Quarkus: 50 Shades of REST

REST is now 25 years old. The birth certificate of this almost impossible to remember acronym (*REpresentational State Transfer*) is considered to be the Y2K doctoral dissertation of Roy Fielding, which aimed at creating a *standard* software architecture, making easier the communication between systems using HTTP (*HyperText Transfer Protocol*).

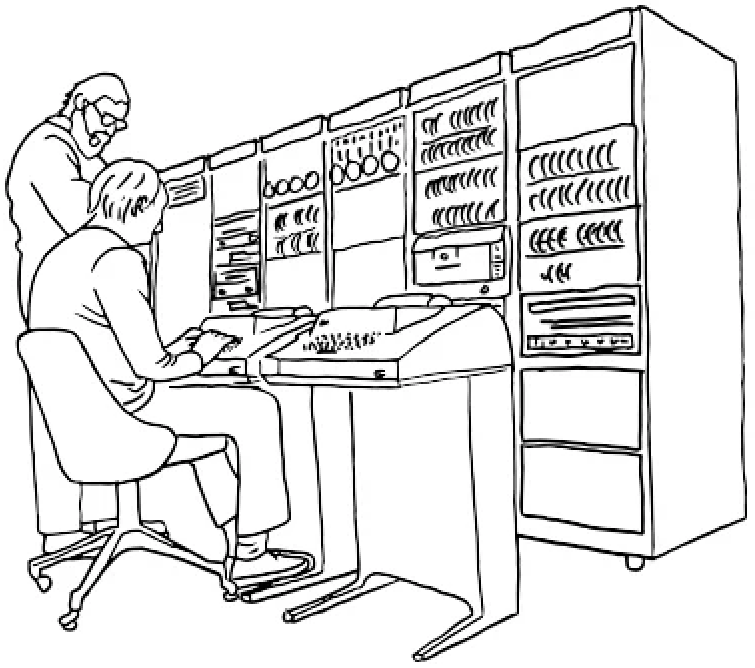
25 years is a long time and, at the IT scale, it's even much longer. We could think that, after a so long period of practising and testing, this paradigm has yielded up all its secrets. But no, screams often our daily activity, constraining us to observe exactly the opposite.

Hence, the idea of this posts series which tries to address the essential aspects of this old, yet unknown, web technology, from its most basic features, like verbs and resource naming conventions, to the most advanced ones, like non-blocking, asynchronous or reactive processing, together with the whole diversity of the REST clients, blocking or non-blocking, synchronous or asynchronous, reactive or classic.

And since in order to illustrate my discourse I need code examples, I chose to write them in Java, with its supersonic subatomic dedicated stack, that doesn't need any longer presentation.

A bit of history

In the begging there was nothing. Before the 70s, there weren't networks at all. Computers were standalone boxes, like the one below, which didn't communicate each other. No files transfer, no remote access, no email, no internet, nothing.



Arpanet, IBM SNA, DECnet and friends

This started to change in the beginning '70s with the birth of the ARPANET. But it wasn't until the end of the '70s that ARPANET became a backbone with several hundreds of nodes, using as processors minicomputers that later became routers. ARPANET was a network of networks, an inter-network, hence the Internet that it became later.

ARPANET was based on TCP/IP (*Transmission Control Protocol/Internet Protocol*), the first network protocol which continues to be the *lingua franca* of our nowadays internet. However, starting with the beginning of the '80s, other network protocols have raised as well. Among the most famous IBM SNA (*System Network Architecture*) and DECnet (*Digital Equipment Coorportaion net*). They were both proprietary, yet popular network architectures connecting mainframes, minicomputers peripheral devices like teleprinters and displays, etc.

IBM SNA and DECnet have competed until early '80s when another one, OSI (*Open System Interconnect*), backed by European telephone monopolies and most governments, was favored. Well-defined and supported by many state organizations, OSI became quickly an ISO standard and was in the process of imposing itself on the market, but it suffered from too much complexity and, finally, it gave way to TCP/IP which was already a *de-facto* standard. Accordingly, at the end of the '80s, the network protocols war was finished and

TCP/IP was the winner.

RPC

But the network protocols wasn't the only war that took place in that period. Once that the computer networks became democratic and affordable, new distributed software applications started to emerge. These applications weren't designed anymore to run in a single isolated box but as standalone components, on different nodes of the network. And in order to communicate, in order that local components be able to call remote ones, software communication protocols were required.

The first major software communication protocol was RPC (*Remote Procedure Call*), developed by SUN Microsystems in the 80s. One of the first problems that the distributed computing had to solve was the fact that, in order to perform a remote call, the caller needs to capture the essence of the callee. A call to another component, be it local or remote, needs to be compiled and, in order to be compiled, the callee procedure needs to be known by the compiler, such that it can translate its name to a memory address. But when the callee is located on a different node than the caller, the compiler cannot know the callee procedure since it is remote. Hence the notion of *stub*, i.e. a local proxy of the remote procedure that, when called, transforms the local call into remote one.

Which means that, in order to call a remote procedure, in addition of the two components, the caller and the callee, a caller stub able to transform the local call into a remote one, as well as a callee stub, able to transform the local return into a remote one, are required. These stubs are very complex artifacts and coding them manually would have been a nightmare for the poor programmer. One of the greatest merits of RPC was to recognize the difficulty of such an undertaking and to provide `genrpc`, a dedicated tool to generate the stubs using a standard format, named XDR (*eXternal Data Representation*), in a way the grandfather of the nowadays' XML and JSON.

RPC was a big success as it proposed a rather straightforward model for distributed applications development in C, on Unix operating system. But like any success, it has been forgotten as soon as other new and more interesting paradigm have emerged.

DCOM

Nevertheless, the success of RPC has encouraged other software editors to take a real interest in this new paradigm called henceforth *middleware* and which allowed programs on different machines to talk each other. Microsoft, for example, living up to its reputation, adopted RPC but modified it and, in the early '90s, released a specific Windows NT version of it, known as MSRPC. Several years later, in September 1996, Microsoft launched DCOM (*Distributed Component Object Model*).

Based on MSRPC and on RPC, which underlying mechanism it was using, DCOM imposed itself as a new middleware construct supporting OOP (*Object Oriented Programming*). The OOP support provided by DCOM was great progress compared with the RPC layer as it allowed a higher abstraction level and to manipulate complex types instead of the XDR basic ones.

Unlike RPC and MSRPC accessible only in C, DCOM supported MS Visual C/C++ and Visual Basic. However, like all the Microsoft products, DCOM was tied to Windows and, hence, unable to represent a reliable middleware, especially in heterogeneous environments involving different hardware, operating systems and programming languages.

CORBA

The *Common Object Request Broker Architecture* is an OMG (*Object Management Group*) standard that emerged in 1991 and which aimed at bringing solutions to the DCOM's most essential concerns, especially its associated vendor lock-in pattern that made its customer dependent on the Windows operating system.

As a multi-language, multi-os and multi-constructor platform, Corba was the first true distributed object-oriented model. It replaced the `rpcgen` utility, inherited from RPC and MSRPC, by IDL (*Interface Definition Language*), a plain text notation. And instead of the old XDR, the IDL compiler generated C++ or Java code directly.

Corba has definitely been a major player in the middleware history thanks to its innovative architecture based on components like POA (*Portable Object Adapter*), PI (*Portable Interceptors*), INS (*Interoperable Naming Service*) and many others.

But Corba was complex and required a quite steep learning curve. Its approach was powerful but using it carelessly could have led to terrible applications, impacting dramatically the infrastructure performances. Moreover, it was based on IIOP (*Internet Inter ORB Protocol*), an unfriendly firewall communication protocol that used raw TCP:IP connections to transmit data.

All these aspects made feel like, despite Corba's great qualities, the community wasn't yet ready to adopt the first distributed object-oriented model.

RMI

Positioned initially as the natural outgrowth of Corba, RMI (*Remote Method Invocation*) has been introduced with JDK (*Java Development Kit*) 1.1, in 1997. One year later, JDK 1.2 introduced Java IDL and `idl2java`, the Java counterpart of Corba's IDL, supporting IIOP.

In 1999, the RMI/IIOP extension to the JDK 1.2 enabled the remote access of any Java distributed object from any IIOP supported language. This was a major evolution as it delivered Corba distributed capabilities to the Java platform.

Two years later, in 2001, the JDK 1.4 introduced support for POA, PI and INS, signing this way the Corba's death sentence. A couple of the most widespread implementations, like Borland's VisiBroker or Iona's Orbix, have still subsisted until 2003, when they got lost into oblivion.

From now on, Java RMI became the universal distributed object-oriented object model.

Jakarta Enterprise Beans (EJB)

In 1999, SUN Microsystems has released the first version of what they're calling the Java Enterprise platform, named a bit confusing J2EE (*Java 2 Enterprise Edition*). This new Java based framework was composed of 4 specifications: JDBC (*Java Data Base Connection*), EJB (*Enterprise Java Beans*), Servlet and JSP (*Java Server Pages*). In 2006 J2EE became Java EE and, 11 years later, in 2017, it changed again its name to become Jakarta EE.

Between 1999 and today, the Jakarta EE specifications have evolved dramatically. Started with the previous mentioned 4 subprojects, they represent today more than 30. But the EJB specifications, currently named Jakarta Enterprise Beans, remain among the most the innovative Java APIs, the legitimate heir of Java RMI.

Enhanced under the JCP (*Java Community Process*) as JSR (*Java Specification Request*) 19 (EJB 2.0), JSR 153 (EJB 2.1), JSR 220 (EJB 3.0), JSR 318 (EJB 3.1) and JSR 345 (EJB 3.2), these specifications provide even today the standard way to implement the server-side components, often called the backend. They handle common concerns in enterprise grade applications, like security, persistence, transactional integrity, concurrency, remote access, race conditions management, and others.

Jakarta XML Web Services (JAX-WS)

While Jakarta Enterprise Beans compliant components were the standard solution to implement and encapsulate business logic, a new markup notation for storing, transmitting and reconstructing arbitrary data, has emerged. This notation, named XML (*eXtended Markup Language*), finished by being adopted as a standard by WWW (*World Wide Web*) consortium, in 1999. And as that's often the case in the IT history, barely adopted, it immediately became so essential, so much so that it was quickly considered that any XML application was mandatory great.

Consequently, it didn't need much to architectures boards to consider that exchanging XML documents, instead of RMI/IIOP Jakarta Enterprise Beans payloads, would be easier and more proficient. It was also considered that Jakarta Enterprise Beans was heavy because it required stubs to be automatically downloaded from servers to clients and, once downloaded, these stubs acted like client-side objects, making remote calls. This required that the byte-code for the various programmer-defined Java classes be available on the client machine and, this setup was considered a significant challenge.

The alternative was the so-called *web services*, a newly coined concept supposed to simplify the distributed processing. According to this new paradigm, clients and servers would exchange XML documents, i.e. text data. This documents grammar is described by a new notation, called XSD (*XML Schema Defintion*), having the same capabilities as an object-oriented programming language, supporting inheritance, polymorphism, etc. This XSD notation was to the web services what XDR was to RPC.

As for the interface contract between clients and servers, another new XML based notation, called WSDL (*Web Service Definition Language*), was required. Last but not least, the payload exchanged between clients and servers was expressed using a yet another new XML based notation, called SOAP (*Simple Object Access Protocol*) which, despite its name, was anything but simple. The funy thing is that all this huge labyrinth was considered simpler that the old good Jakarta Enterprise Beans component.

Nevertheless, all this madness became standard in 2003, as JSR 101, known also under the name of JAX-RPC (*Java API for XML-Based RPC*) and later, in 2017 as JSR 224, named JAX-WS (*Java API for XML-Based Services*). These specifications gave rise to a lot other, including but not limited to WS-I Basic Profile, WS-I Attachments, WS-Addressing, SAAJ, etc.

Jakarta RESTful Web Services (JAX-RS)

After this so convoluted piece of history, we come finally at the end of our journey, in 2009, when the specifications JAX-RS became a part of Java EE 6. Today, in 2024, they are named Jakarta RESTful Web Services and are a part of Jakarta EE 11. Since 15 years they represent the main substratum making service and microservices to communicate each-other, as well as with the external world.

In this post series we'll examine all their 50 shades :-).

The Classics

In Java, more precisely in Java Enterprise or, if you prefer, the *server side* Java, called today Jakarta EE, the REST web services come under the Jakarta RESTfull Web Services specifications, formerly named JAX-RS (*Java API for Restful Web Services*). These specifications are considered as *classics* because they are the only existent ones, as far as Java is concerned. Other implementations exist, for example the one proposed by Spring REST but, as opposed to the Jakarta EE ones, defined under a formalised mechanism and maintained by big consortia, these specification-less open-source libraries are guided by their maintainer's goals alone, not by the collabotation between companies, user groups and communities.

Jakarta REST (short for Jakarta RESTfull Web Services) is an old specification, its 1.1 release being issued in 2009 under the Java EE 6 auspices. Its current version is the 4.0.0, published the 30th of April 2024, as a part of Jakarta EE 11.

Different editors have adopted it, over the years, and provided different implementations, of which among the most prominent are Apache CXF, Oracle Jersey and Red Hat RESTeasy. It comes without surprise that Quarkus offers support for RESTeasy, via its `quarkus-rest*` extension set.

REST, as proposed by Roy Fielding, is a protocol-agnostic architecture, however, all its major implementations use HTTP (*HyperText Transfer Protocol*) as its application layer protocol. HTTP has been designed on the purpose such that to provide request/response based interactions. Despite this particularity, which could be considered a weakness, HTTP is nowadays ubiquitously used as front tier to expose APIs consumable by other services.

The fact that HTTP, as an application layer protocol (HTTPS is a transport layer protocol), has been designed with this request/response pattern in mind, makes it intrinsically synchronous. In order to serve HTTP requests, an HTTP server is required. This server listens on a dedicated TCP port (8080 for example) and waits for incoming requests. When it receives a new HTTP request the server parses it, determines the HTTP method (for example GET, POST, etc.) as well as the path and the request's body. If HTTP filters or interceptors are present, then they will be executed, after which the endpoint aimed at processing the request will be looked up. Once this endpoint found, it will be invoked, the request is processed and the result captured in a HTTP response that will be sent back to the consumer.

As we can see, the described process is eminently synchronous. It implies that the same single thread is used for the entire request processing lifetime. This is the most common and basic use case of the HTTP protocol. Let's look at a first REST service example which leverages this pattern.

A simple REST service

In the `/50-shades-of-rest/classic/` directory of the GitHub repository you'll find a simple Quarkus application which exposes a REST service having two endpoints, as follows:

- an endpoint named `/time` which returns the current local date and time in the default time zone;
- an endpoint named `/time/{zoneId}` which returns the current date and time in the time zone which ID is passed as path parameter.

The Listing 2.1 below shows a short code excerpt.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String getCurrentDateAndTimeAtDefaultZone()
```

```
{
  return ZonedDateTime.now().format(DateTimeFormatter.ofPattern(FMT)
    .withZone(ZoneId.systemDefault()));
}
```

Listing 2.1: A Simple Jakarta REST service with Quarkus

As you see, it couldn't be simpler. The endpoint above serves `GET` requests and the result it produces is plain ASCII text. The 2nd endpoint is just as simple. It servers `GET` requests as well but having as a path parameter a timezone ID.

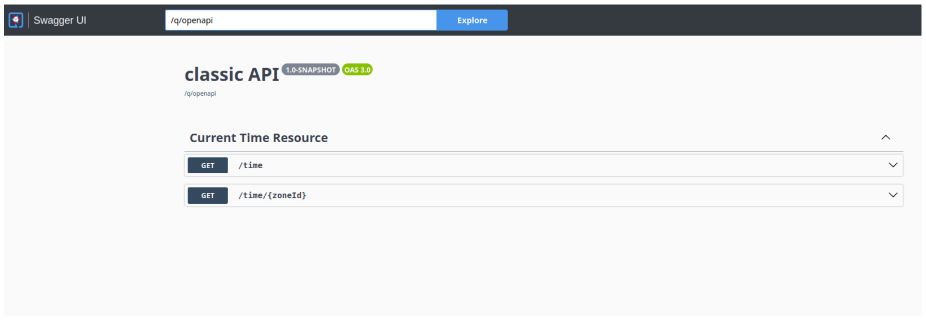
You can test this REST service by either running Quarkus in dev mode, using Maven command `mvn quarkus:dev` or, if you prefer, by executing the *fast JAR*, as follows:

```
$ cd classic
$ mvn clean package
$ java -jar target/quarkus-app/quarkus-run.jar
```

Once that the Quarkus application has started, be it in dev mode or running the *fast JAR*, you have to fire your preferred browser at the service's endpoints, as show below:

```
$ curl http://localhost:8080/time && echo
13 Oct 2024, 00:33:49 +02:00 CEST
$ curl http://localhost:8080/time/Europe%2FMoscow && echo
13 Oct 2024, 01:35:59 +03:00 MSK
$ curl http://localhost:8080/time/America%2FNew_York && echo
12 Oct 2024, 18:31:47 -04:00 EDT
```

Another practical way to test this service is using Swagger UI. Just fire your browser at `http://localhost:8080/s/swagger-ui/` and you'll be presented with the following screen:



Here you can test the endpoints by clicking on the `GET` button and, then, selecting `Try out`.

The `classic` Maven project contains also two unit tests. They are decorated with the `@QuarkusTest` annotation and use the RESTassured, a Java DSL (*Domain*

Specific Language) for testing. Running the Maven test phase will display the following result:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running fr.simple_software.fifty_shades_of_rest.classic.tests.TestCurrentTimeResource
2024-10-13 01:38:24,185 INFO  [io.quarkus] (main) classic 1.0-SNAPSHOT on JVM (powered by Qu
2024-10-13 01:38:24,186 INFO  [io.quarkus] (main) Profile test activated.
2024-10-13 01:38:24,187 INFO  [io.quarkus] (main) Installed features: [cdi, rest, rest-clien
13 Oct 2024, 01:38:24 +02:00 CEST
13 Oct 2024, 01:38:24 +02:00 EET
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.811 s -- in fr.simp
2024-10-13 01:38:25,068 INFO  [io.quarkus] (main) classic stopped in 0.132s
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary for 50 Shades of REST :: The master POM 1.0-SNAPSHOT:
[INFO]
[INFO] 50 Shades of REST :: The master POM ............... SUCCESS [  0.002 s]
[INFO] 50 Shades of REST :: the classic module ........... SUCCESS [  5.475 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.201 s
[INFO] Finished at: 2024-10-13T01:38:25+02:00
[INFO] ------------------------------------------------------------------------
```

In addition, of these unit tests, a couple of integration tests are provided as well. In order to demonstrate different test APIs available for REST services testing, these integration tests use the Jakarta REST client instead of RESTassured. The class `jakarta.ws.rs.client.ClientBuilder` is used here in order to instantiate the implementation of the `jakarta.ws.rs.client.Client` interface on the behalf of which the REST endpoints are called. Look carefully at the code to see how `try-with-resource` statements are used.

To run the integration tests, start first the Quarkus application by executing the *fast JAR* as you already did above, then run `mvn failsafe:integration-test`. A summary similar to the one for the unit tests will be shown, for example:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running fr.simple_software.fifty_shades_of_rest.classic.tests.CurrentTimeResourceIT
2024-10-14 00:22:35,841 INFO  [io.quarkus] (main) classic 1.0-SNAPSHOT on JVM (powered by Qu
2024-10-14 00:22:35,843 INFO  [io.quarkus] (main) Profile test activated.
2024-10-14 00:22:35,843 INFO  [io.quarkus] (main) Installed features: [cdi, reactive-routes,
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.352 s -- in fr.simp
2024-10-14 00:22:36,376 INFO  [io.quarkus] (main) classic stopped in 0.144s
```

The synchronous communication drawback

As we already mentioned, using synchronous communication for REST services, as well as for web services in general, is very common. You send a request and wait for the associated response. While synchronous processing is easy to understand and to reason about because the code structure is sequential, it has a drawback: it leads to *time coupling*. This is one of the most complex form of coupling consisting in the fact that it requires all the participants and the network to be operational for the complete duration of the exchange.

But this isn't the case in the real live where the nodes we want to interact with might not be reachable, or they are reachable but fail to process the request, or even worst, the nodes are reachable, they succeed to receive and process the requests but the network connection is lost while the responses are written back to the consumers.

We can simulate all these outcomes using a Quarkus interceptor that catches up the network traffic and inserts different HTTP headers in the incoming requests and the outgoing responses, such that to simulate failures or to introduce delays and loses.

A fault simulator

The class `FaultSimulator` in the `failures` module of our Maven project performs the operations described above. Looking at the code, you can see the following `enum` structure which defines all the failure types that can happen:

```
private enum Fault
{
  NONE,
  INBOUND_REQUEST_LOSS,
  SERVICE_FAILURE,
  OUTBOUND_RESPONSE_LOSS
}
```

Here we have the following categories:

- no failure (`NONE`)
- the request is lost between the consumer and the producer (`INBOUND_REQUEST_LOSS`)
- the producer receives the request but fails to process it (`SERVICE_FAILURE`)
- the consumer receives and processes the request but fails to write it back to the consumer (`RESPONSE_LOSS`)

The `FaultSimulator` is a CDI (*Contexts and Dependency Injection*) bean having an application scope and being decorated with the `@Route` annotation, to serve and endpoint named `fail`. It takes the following query parameters:

- the type of failure: `INBOUND_REQUEST_LOSS`, `SERVICE_FAILURE`, `OUTBOUND_REQUEST_LOSS`. `NONE` is the default value.
- the fault ratio: a value in the interval [0, 1) defining how much percent of the requests will randomly fail. 0.5 is the default value.

To illustrate the synchronous communication drawback you need first to start the Quarkus application, in either dev mode or as a *fast JAR*, for example:

```
$ cd classic
$ mvn clean package
$ java -jar target/quarkus-app/quarkus-run.jar
```

Then, use the `curl` command below in order to configure the `FaultSimulator` to lose 50% of the incoming requests:

```
curl http://localhost:8080/fail?failure=INBOUND_REQUEST_LOSS && echo
Faults are enabled: fault = INBOUND_REQUEST_LOSS, failure rate = 0.5
```

Now let's test again our service endpoints `/time` and `/time/{zoneId}` as we did previously:

```
$ curl --max-time 5 http://localhost:8080/time && echo
13 Oct 2024, 17:27:00 +02:00 CEST
$ curl --max-time 5 http://localhost:8080/time && echo
curl: (28) Operation timed out after 5001 milliseconds with 0 bytes received
```

We've been lucky, in only two trials we managed to get a timeout after 5 seconds (the `--max-time` parameter) as if the HTTP request has never attended the producer. Now, let's try the 2nd type of failure:

```
$ curl http://localhost:8080/fail?failure=SERVICE_FAILURE
Faults are enabled: fault = SERVICE_FAILURE, failure rate = 0.5
$ curl http://localhost:8080/time && echo
FAULTY RESPONSE!
```

Here again we've been lucky as we managed to get from the first try the faulty response, showing that, this time, the request has successfully been received by the producer, but it failed to process, whatever the reason might be.

Last but not least:

```
$ curl http://localhost:8080/fail?failure=OUTBOUND_RESPONSE_LOSS
Faults are enabled: fault = OUTBOUND_RESPONSE_LOSS, failure rate = 0.5
$ curl http://localhost:8080/time && echo
curl: (52) Empty reply from server
```

Now the network connection has been abruptly closed before the response reaches the consumer.

These examples aim at helping to understand the impact of not only the uncertainty of certain communication operations, but also to illustrate the strong coupling induced by the synchronous request/response based pattern. This type of communication is often used because of its simplicity, but it only works if all the infrastructure elements are simultaneously operational, which is never the case in the real live.

So, what are the options in that case ? Well, we can still continue to use sync communication and to try to gracefully manage exceptional conditions using time- outs, retries, etc. For example, the following `curl` command:

```
$ curl --max-time 5 --retry 100 --retry-all-errors http://localhost:8080/time && echo
curl: (52) Empty reply from server
Warning: Problem (retrying all errors). Will retry in 1 seconds. 100 retries
Warning: left.
13 Oct 2024, 18:01:32 +02:00 CEST
```

This command waits during maximum 5 seconds for a result and, failing that, it exits on time-out. It will retry 100 times, every second or until success. And as a matter of fact, you can see that, during the first invocation, the HTTP response is lost before attending the consumer. Never mind, we'll try again in a sec and, this time, we're successful.

So, this would be one of the options: synchronous communication gracefully handled. But there is another option, more interesting: using asynchronous communication. This is what we'll be seeing in the next sections.

Asynchronous processing with REST services

There are two levels of asynchronous processing as far as REST services are concerned:

- the asynchronous client processing: in this scenario the consumer invokes an endpoint which returns immediately. Depending on the type of asynchronous invocation the return might be of type `Future`, `CompletionStage`, etc. But the operation didn't finish yet at the moment when the consumer call returns. Perhaps it did even start yet. In order to get the result, the consumer has different options, for example to do polling or to use callbacks and continuations.
- the asynchronous server processing: in this scenario, the producer itself processes the request asynchronously.

The two cases above may be combined such that to have asynchronous consumers with synchronous producers, asynchronous consumers with asynchronous producers and synchronous consumers with asynchronous producers. Let's examine them closer with the help of some examples.

Asynchronously invoking REST services

Asynchronous REST services consumers have been introduced with the 2.0 release of the specifications, back in 2011. The idea is simple.