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.

Testing the simple REST service

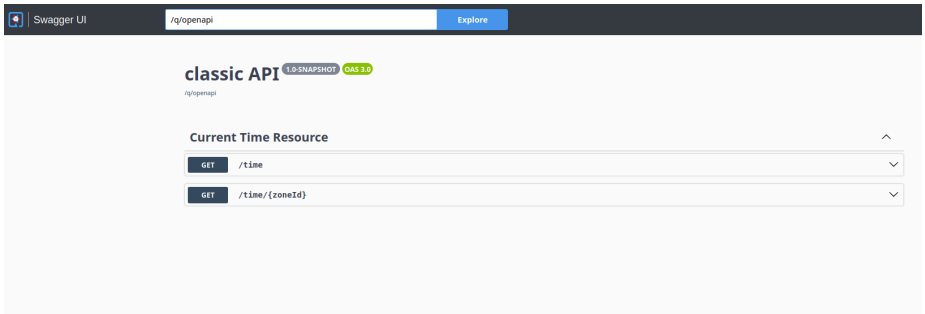
You can test this REST service by either running Quarkus in dev mode, using Maven command `mvn quarkus:dev` or, if you prefer, by executing the *fast JAR*, as follows:

```
$ 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 unit tests

Testing REST endpoints with the `curl` utility or with Swagger UI is very practical,however, it requires manual operation. Despite being repetitive and fastidious, this is also error-prone. This is where the unit testing becomes very important.

With Quarkus, unit testing has never been so easy. The `quarkus-junit5` extension is required, as well as annotating your test classes with `@QuarkusTest`. All the more so Quarkus integrates with RESTassured, a very well-known Java DSL (*Domain Specific Language*) for testing.

Bur REST is vast field and, hence, other test packages and libraries are used since years. Accordingly, while testing Quarkus REST endpoints with RESTassured becomes a formality, we cannot limit ourselves to only these kind of tests, and we need to cover the full spectrum.

Consequently, we provide the following test categories with the endpoints samples:

- RESTassured tests, as discussed above;
- Eclipse MP REST Client tests. We already mentioned that in one of the preceding paragraphs, Eclipse MP (MicroProfile) is a relatively new project of the Eclipse Foundation which aims at optimizing the enterprise grade microservices architecture. As such, it defines a number of standards, one of which most important is REST Client, that dramatically simplifies the REST clients architecture.
- Jakarta REST clients. Jakarta REST specifications, formerly JAX-RS, define a standard mechanism to invoke REST services. We're using it, among others, for tests purposes.
- Java 11 HTTP Client. In its 11th release back in 2023, Java offers a new HTTP client API that might equally be used for test purposes.

All these test categories are integrated, of course, with JUnit 5. Let's have now a look at them all.

The RESTassured tests

This is probably the simplest and the most practical way to test REST endpoints. The listing below shows an example.

```
@QuarkusTest
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public class TestCurrentTimeResource extends BaseRestAssured
{
  @TestHTTPEndpoint(CurrentTimeResource.class)
  @TestHTTPResource
  URL timeSrvUrl;

  @BeforeAll
```

```
  public void beforeAll() throws URISyntaxException
  {
    timeSrvUri = timeSrvUrl.toURI();
    assertThat(timeSrvUri).isNotNull();
  }

  @AfterAll
  public void afterAll()
  {
    timeSrvUri = null;
  }
}
```

In order to factorize the common behaviour of the RESTassured based unit tests and to minimize the amount of the boilerplate code, we defined a based class called `BaseRestAssured` that contains recurrent in these tests. Hence, our test above extends this class.

The annotation `@QuarkusTest` which decorates this code is used to flag it as a Quarkus unit test. The other annotation at the class level, `@TestInstance`, is a JUnit 5 annotation, not a Quarkus one, and it aims at switching to the "class mode", from the default "method mode" of running tests. In "class mode" the JUnit 5 test executor is set such that to run all the test methods on the same class instance, as opposed to the default "method mode" where each test is executed on its own class instance. The "class mode" has also the side effect of making possible to declare the methods `@BeforeAll` and `@AfterAll` as non static.

These methods are also JUnit 5 annotations, not Quarkus ones, and they are executed once, before and, respectively, after all the tests were run. Here we need to handle the non static property `timeSrvUri` in the `@BeforeAll` method, meaning that this method should be non static as well, hence the use of the annotation.

The couple of annotations `TestHTTPEndpoint` and `TestHTTPResource` are specific Quarkus annotations and their role here is to capture the base URL of the endpoint under the test. A Quarkus unit test will start a local HTTP server, running the REST service defined in the current project. But if the host on which this HTTP server runs will always be `localhost`, the TCP port on which the HTTP listener is listening, which default value is 8081, might be random. Hence, it's the role of this couple of annotations to capture this information.

In our case, the `timeSrvUrl` property of type `URL` will have the value `http://localhost:<port-number>/time`, where `<port-number>` is the TCP port number randomly allocated by Quarkus upon running the test HTTP server. This URL corresponds to the one of the REST service defined in this project, by the class `CurrentTimeResource`, passed as a parameter to the `TestHTTPEndpoint` annotation.

Now, the code of the base class `BaseRestAssured` is shown below:

```
public class BaseRestAssured
{
  private static final String FMT = "d MMM uuuu, HH:mm:ss XXX z";
  protected URI timeSrvUri;

  @Test
  public void testCurrentTime()
  {
    Response response = given().when().get(timeSrvUri);
    assertThat(response.statusCode()).isEqualTo(HttpStatus.SC_OK);
    assertThat(response.getBody()).isNotNull();
    assertThat(LocalDateTime.parse(response.prettyPrint(),
      DateTimeFormatter.ofPattern(FMT)))
        .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.HOURS));
}

@Test
public void testCurrentTimeWithZoneId()
{
  Response response = given().baseUri(timeSrvUri.toString())
   .pathParam("zoneId", URLEncoder.encode("Europe/Kaliningrad", StandardCharsets.UTF_8))
   .when().get("{zoneId}");
  assertThat(response.statusCode()).isEqualTo(HttpStatus.SC_OK);
  assertThat(response.getBody()).isNotNull();
  assertThat(LocalDateTime.parse(response.prettyPrint(),
    DateTimeFormatter.ofPattern(FMT)))
    .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.HOURS));
}
```

The RESTassured syntax, in the most authentic DSL style, is clear and explicit. It leverages the "given-when" statements borrowed from the functional test notations and, for this reason, is easy understandable. For example `given().when().get(timeSrvUri)` sends a GET request to the endpoint which value is stored by the `timSrvUri` property. When a path parameter is required, for example, this becomes:

```
given().baseUri(...).pathParam("zoneId", ...).when().get("{zoneId}")
```

Easy, right ? A bit more complicated is the assertion:

```
assertThat(LocalDateTime.parse(response.prettyPrint(),
  DateTimeFormatter.ofPattern(FMT)))
    .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.HOURS));
```

We're using AssertJ, a library which complements JUnit 5 with more explicit assertions. Here we assert that the date and time returned by our endpoint is less than 1 hour closed to the local date and time.

The MP REST Client tests

MP REST Client is a specification which complements Jakarta REST 2.1 and provides a type-safe approach to invoke REST endpoints on HTTP. RESTeasy, as the REST engine used by Quarkus, implements this specification via the `quarkus-rest-client` extension. Also, a 2nd extension is required to provide support for JSON marshalling and unmarshalling operations and here we have the choice between using Jackson, an open-source library supported by the community, or the standard Jakarta JSON Binding (formerly JSON-B). We choose the standard, of course, hence the `quarkus-rest-client-jsonb` extension.

This having been said, using MP REST Client with Quarkus is as simple as creating an interface matching the service's endpoints. If the service implements itself an interface, which is generally the case, then this same interface should be used for its clients. Let's see how things are working in our specific case.

The following interface is located in the `classic` directory of the GitHub project:

```
@RegisterRestClient(configKey = "base_uri")
@Path("time")
public interface CurrentTimeResourceClient
{
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  String getCurrentDateAndTimeAtDefaultZone();
  @GET
  @Path("{zoneId}")
  @Produces(MediaType.TEXT_PLAIN)
  String getCurrentDateAndTimeAtZone(@PathParam("zoneId") String zoneId);
}
```

This is the interface that exposes the `CurrentTimeResource` REST service that we've seen previously in this chapter. The only new element is the annotation `@RegisterRestClient` which signals to the Quarkus compile time that the given interface is meant for being injected as a CDI (*Context and Dependencies Injection*) component. The other annotations are the old good JAX-RS ones, renamed now as Jakarta REST.

And that's all ! At the compile time, the Quarkus REST processor performs a step named "augmentation", during which it will generate the byte-code associated to the REST client. Accordingly, in order to test a REST service, all that the unit test needs to do is to inject the generated REST client via its interface, and to call the endpoints. Look at the following listing:

```
@QuarkusTest
public class TestCurrentTimeResourceMpClient
{
  @Inject
  @RestClient
```

```
  CurrentTimeResourceClient currentTimeResourceClient;
  private static final String FMT = "d MMM uuuu, HH:mm:ss XXX z";

  @Test
  public void testCurrentTime()
  {
    assertThat(LocalDateTime.parse(currentTimeResourceClient
     .getCurrentDateAndTimeAtDefaultZone(), DateTimeFormatter.ofPattern(FMT)))
       .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.HOURS));
  }

  @Test
  public void testCurrentTimeWithZoneId()
  {
    assertThat(LocalDateTime.parse(currentTimeResourceClient
     .getCurrentDateAndTimeAtDefaultZone(), DateTimeFormatter.ofPattern(FMT)))
       .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.HOURS));
  }
}
```

As you can see, the annotation `@RestClient` injects the service's interface and all that remains to do is to call the associated endpoints against that interface.

The sharp-eyed reader has probably observed that this unit test doesn't mention any URL. This is because the `application.properties` file located in the `resources` directory contains the following property:

```
base_uri/mp-rest/url=http://localhost:8081
```

It defines the default URL of the test HTTP server run by Quarkus. The name `base_uri` is the one used as the value of the argument `configKey` of the `RegisterRestClient` annotation, as shown above.

Jakarta REST Client tests

Java 11 HTTP Server based tests

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.TestCurrentTimeResource
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 drawbacks

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 JAX-RS specifications, back in 2011. The idea is simple: once the consumer has invoked a service endpoint, it doesn't wait for completion, instead it returns control immediately. But now the question is: how does the consumer get the response to its request ? And here purists classify the asynchronous invocation process in two categories:

- blocking asynchronous invocations where the consumer needs to poll the task status in order to check for completion;

- non-blocking asynchronous invocations where the consumer doesn't do any polling, as it is notified when the task is complete.

Later, in 2014, Java 8 has been released and, with it, a couple of new classes have been added to the `java.util.concurrent` package. Some of them, like `CompletableFuture` and `CompletionStage` have greatly improved the way that the asynchronous processing was implemented in REST services. We'll see how.

And in order to further complicate the whole context, in 2017, the 2.1 release of the JAX-RS specifications have brought some more improvements. This is the reason why, when it comes to asynchronous processing with REST services, we need to address all these flavours: JAX-RS 2.0, Java 8 and JAX-RS 2.1.

That's what we'll be trying to do in the next paragraphs.

JAX-RS 2.0: Asynchronously invoking REST services

Let's have a look at some examples of how this kind of asynchronous consumers, blocking and non-blocking, might be implemented based on the JAX-RS 2.0 specifications.

Blocking asynchronous consumers The listing below shows a Quarkus integration test that invokes in a blocking while asynchronous mode the endpoint `/time` of the `CurrentTimeResource` REST service.

```
@Test
public void testCurrentTime()
{
  try (Client client = ClientBuilder.newClient())
  {
    Future<String> timeFuture = client.target(TIME_SRV_URL).request().async().get(String.cla
    String time = timeFuture.get(5, TimeUnit.SECONDS);
    assertThat(parseTime(time)).isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.MINU
  }
  catch (Exception ex)
  {
    fail("### BlockingAsyncCurrentTimeResourceIT.testCurrentTime(): Unexpected exception %s'
  }
}
```

This code may be found in the `/home/nicolas/50-shades-of-rest/async-clients /blocking-async-clients/` directory of the GitHub repository. Please notice the `async()` verb in the request definition.

As you can see, this time the endpoint invocation doesn't return a `String`, but a `Future<String>`, i.e a kind of promise that, once the operation completed, the result will be returned. The call to the service returns immediately, before it had a chance to complete. Then, the `get()` method will block the current thread waiting for completion during 5 seconds. Hence, the blocking side of the call.

Non-blocking asynchronous consumers Now let's look at a 2nd example implementing the same integration test but in a non-blocking way.

```
@Test
public void testCurrentTime()
{
  try (Client client = ClientBuilder.newClient())
  {
    CountDownLatch latch = new CountDownLatch(1);
    client.target(TIME_SRV_URL).request().async().get(new InvocationCallback<String>()
    {
      @Override
      public void completed(String t)
      {
        ldt = parseTime(t);
        latch.countDown();
      }

      @Override
      public void failed(Throwable throwable)
      {
        fail("""
          ### NonBlockingAsyncCurrentTimeResourceIT.testCurrentTime():
          Unexpected exception %s""", throwable.getMessage());
        latch.countDown();
      }
    });
    if (latch.await(5, TimeUnit.SECONDS))
      assertThat(ldt)
        .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.MINUTES));
  }
  catch (Exception ex)
  {
    fail("""
      ### NonBlockingAsyncCurrentTimeResourceIT.testCurrentTime()):
      Unexpected exception %s""", ex.getMessage());
  }
}
```

We're still using the `async()` method to invoke our endpoint but, this time, the `get()` method won't take anymore the type of the expected result as its input parameter, but an instance of the interface `jakarta.ws.rs.client.InvocationCallback`. This interface has two methods:

```
public void completed(T t);
public void failed(Throwable throwable);
```

The first one notifies the task completion. It takes as its input parameter the operation result, i.e. the `String` containing the current date and time. The 2nd one notifies the task failure and takes as its input parameter the current exception which prevented it to complete.

Our callback is executed by a different thread than the one making the call. This thread, called *worker thread*, will be started automatically. In order to synchronize execution, that is to say, to make sure that our main thread doesn't exit before the worker one finishes the job, a count-down with the initial value of 1 is armed. It will be decremented by both `completed()` and `failed()` methods, such that, by using the `await()` function, we can wait the end of the worker thread, before exiting the main one.

Java 8: Asynchronously invoking REST services

Using the `java.util.concurrent.Future` class for asynchronously invoking REST endpoints was already great progress compared to the initial 1.x release of the JAX-RS specifications which only allowed synchronous calls. Introduced in Java 5, this class represents,as we've seen, the result of an asynchronous processing. It contains all the required methods to poll and wait for the result, however it remains quite basic and limited.

Several years later, Java 8 improves the asynchronous processing by introducing the class `CompletableFuture` as an enhancement of `Future`. It not only represents a future result but also provides a plethora of methods to compose, combine, execute asynchronous tasks, and handle their results without blocking.

So let's examine how to use these new classes in the Java 8 style.

Blocking asynchronous consumers Have a look at the listing below, that you can found in the `/home/nicolas/ 50-shades-of-rest/async-clients/java8-async-clients/` directory of the GitHub repository.

```
@Test
public void testCurrentTimeWithZoneId()
{
  try (Client client = ClientBuilder.newClient())
  {
    CompletableFuture<String> timeFuture = CompletableFuture.supplyAsync(() ->
      client.target(TIME_SRV_URL).path(ENCODED).request().get(String.class))
      .exceptionally(ex -> fail("""
         ### BlockingJava8CurrentTimeResourceIT.testCurrentTimeWithZoneId():\s"""
         + ex.getMessage()));
    assertThat(parseTime(timeFuture.join())).isCloseTo(LocalDateTime.now(),
      byLessThan(1, ChronoUnit.MINUTES));
  }
}
```

What you see here is that the endpoint invocation is done now using `CompletableFuture.supplyAsync()` to which we provide the JAX-RS client request as a supplier, implemented on the behalf of a Lambda function. The `supplyAsync()` method executes the given task on the behalf of a new thread that it creates. We have here the possibility to specify an `Executor` or, in the default case, it will fork a new thread from the common thread pool shared across the application.

This method will return immediately, without waiting the task completion. The supplied task (the lambda function) will run on this separate thread. Since this is a blocking asynchronous endpoint invocation, we use the `join()` method in order to wait for the task completion.

To resume, this example is very similar to the one in `/home/nicolas/50-shades-of-rest/async-clients/jaxrs20-async-clients/`, with the only difference that it returns a `CompletableFuture` instead of a `Future`. Also, the `async()` method isn't anymore required when instantiating the JAX-RS client request.

Non-Blocking asynchronous consumers The same similarities that we noticed above are also in effect as far as the non-blocking asynchronous invocation are concerned. Here is a code fragment:

```
@Test
public void testCurrentTimeWithZoneId()
{
  try (Client client = ClientBuilder.newClient())
  {
    Callback callback = new Callback();
    CompletableFuture.supplyAsync(() ->
      client.target(TIME_SRV_URL).path(ENCODED).request().async().get(callback))
      .exceptionally(ex -> fail("""
        ### NonBlockingJava8CurrentTimeResourceIT.testCurrentTimeWithZoneId():
        \s""" + ex.getMessage()));
  }
}
```

If we didn't need to use the `async()` method in the previous example, here we do as, otherwise, we cannot pass a `Callback` instance to our `get()` method. For the rest, everything works nearly the same, differing slightly only that the `CompletionFuture` class is now used instead of `Future`. Also, we separately defined the `Callback` class instead of providing it in-line as this facilitates its reuse. Here's the listing:

```
public class Callback implements InvocationCallback<String>
{
  private final CountDownLatch latch;
  private final AtomicReference<String> time;
```

```
  public Callback()
  {
    this.latch = new CountDownLatch(1);
    this.time = new AtomicReference<>();
  }

  @Override
  public void completed(String strTime)
  {
    time.set(strTime);
    latch.countDown();
  }

  @Override
  public void failed(Throwable throwable)
  {
    latch.countDown();
    fail("### Callback.failed(): Unexpected exception", throwable);
  }

  public String getTime() throws InterruptedException
  {
    String strTime = null;
    if (latch.await(5, TimeUnit.SECONDS))
      strTime = time.get();
    return strTime;
  }
}
```

Very few things have changed here, if any, compared to the preceding version.

JAX-RS 2.1: Asynchronously invoking REST services

As we have seen, asynchronously invoking REST services in an either blocking or non-blocking way, as defined by the JAX-RS 2.0 specifications, would either result in polling the response, by calling `get()`, or registering a callback that would be invoked when the HTTP response is available. Both of these alternatives are interesting but things usually get complicated when you want to nest callbacks or add conditional cases in the asynchronous execution flows. JAX-RS 2.1 offers a new way to overcome these problems. It is called *Reactive Client API* and it simply consists in invoking the `rx()` method, instead of `async()`, as it was the case with JAX-RS 2.0.

Blocking asynchronous consumers In the listing below, using `rx()` returns a response of type `CompletionStage`. Then the method `thenAccept()` will

execute the given action in a blocking mode, on the same thread.

```
@Test
public void testCurrentTime()
{
  try (Client client = ClientBuilder.newClient())
  {
    client.target(TIME_SRV_URL).request().rx().get(String.class)
    .thenAccept(time -> {
      assertThat(parseTime(time)).isCloseTo(LocalDateTime.now(),
        byLessThan(1, ChronoUnit.MINUTES));
    })
    .exceptionally(ex -> {
      fail("""
        ### BlockingRxCurrentTimeResourceIT.testCurrentTime():
         Unexpected exception %s""", ex.getMessage());
      return null;
    })
    .toCompletableFuture().join();
  }
}
```

Here, the call to `join()` will block the current thread until the operation completes.

Non-Blocking asynchronous consumers Let's have a look now at the non-blocking asynchronous consumer:

```
@Test
public void testCurrentTime()
{
  try (Client client = ClientBuilder.newClient())
  {
    client.target(TIME_SRV_URL).request().rx().get(String.class)
      .thenApply(time -> {
        assertThat(parseTime(time)).isCloseTo(LocalDateTime.now(),
          byLessThan(1, ChronoUnit.MINUTES));
         return time;
      })
      .exceptionally(ex -> {
        fail("""
          ### BlockingRxCurrentTimeResourceIT.testCurrentTime():
          Unexpected exception %s""", ex.getMessage());
        return null;
      });
  }
}
```

We have replaced the call to `thenAccept(...)` by `thenApply(...)`. Doing that, we don't wait anymore the task completion, as we did previously by calling `join()`. Instead, `thenApply()` returns a `CompletionStage` with the result of the endpoint call which, in our case, is a `String`. So, as opposed to the approach using `thenAccept()` where `join()` blocks the current thread until the operation completes, the one using `thenApply()` remains non-blocking as it returns a `CompletionStage` that will complete in the future. However, as opposed to the first one which guarantees that the assertion has been evaluated before processing, the last one doesn't.

This is furthermore what happens because, modifying the assertion as follows:

```
assertThat(LocalDateTime.of(2019, Month.MARCH, 28, 14, 33))
  .isCloseTo(LocalDateTime.now(), byLessThan(1, ChronoUnit.MINUTES));
```

the test will still pass, showing that the assertion isn't in fact evaluated, otherwise an `AssertionError` would have been raised. Accordingly, while non-blocking calls are still possible with the JAX-RS 2.1 `rx()` flavour, they are more suitable for being integrated in larger asynchronous flows than in tests, where a terminal `join()` is at some point required.