

Quarkus - Writing JSON REST Services

JSON is now the *lingua franca* between microservices.

In this guide, we see how you can get your REST services to consume and produce JSON payloads.



there is another guide if you need a [REST client](#) (including support for JSON).

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

Architecture

The application built in this guide is quite simple: the user can add elements in a list using a form and the list is updated.

All the information between the browser and the server are formatted as JSON.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `rest-json-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.4.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=rest-json-quickstart \
  -DclassName="org.acme.rest.json.FruitResource" \
  -Dpath="/fruits" \
  -Dextensions="resteasy-jsonb"
cd rest-json-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS and [JSON-B](#) extensions.

Quarkus also supports [Jackson](#) so, if you prefer Jackson over JSON-B, you can create a project relying on the RESTEasy Jackson extension instead:

```
mvn io.quarkus:quarkus-maven-plugin:1.4.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=rest-json-quickstart \
  -DclassName="org.acme.rest.json.FruitResource" \
  -Dpath="/fruits" \
  -Dextensions="resteasy-jackson"
cd rest-json-quickstart
```

To improve user experience, Quarkus registers the three Jackson [Java 8 modules](#) so you don't need to do it manually.

Creating your first JSON REST service

In this example, we will create an application to manage a list of fruits.

First, let's create the **Fruit** bean as follows:

```
package org.acme.rest.json;

public class Fruit {

    public String name;
    public String description;

    public Fruit() {
    }

    public Fruit(String name, String description) {
        this.name = name;
        this.description = description;
    }
}
```

Nothing fancy. One important thing to note is that having a default constructor is required by the JSON serialization layer.

Now, edit the `org.acme.rest.json.FruitResource` class as follows:

```

package org.acme.rest.json;

import java.util.Collections;
import java.util.LinkedHashMap;
import java.util.Set;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {

    private Set<Fruit> fruits = Collections.newSetFromMap
(Collections.synchronizedMap(new LinkedHashMap<>()));

    public FruitResource() {
        fruits.add(new Fruit("Apple", "Winter fruit"));
        fruits.add(new Fruit("Pineapple", "Tropical fruit"));
    }

    @GET
    public Set<Fruit> list() {
        return fruits;
    }

    @POST
    public Set<Fruit> add(Fruit fruit) {
        fruits.add(fruit);
        return fruits;
    }

    @DELETE
    public Set<Fruit> delete(Fruit fruit) {
        fruits.removeIf(existingFruit -> existingFruit.name
.contentEquals(fruit.name));
        return fruits;
    }
}

```

The implementation is pretty straightforward and you just need to define your endpoints using the JAX-RS annotations.

The **Fruit** objects will be automatically serialized/deserialized by **JSON-B** or **Jackson**, depending on the extension you chose when initializing the project.



While RESTEasy supports auto-negotiation, when using Quarkus, it is very important to define the **@Produces** and **@Consumes** annotations. They are analyzed at build time and Quarkus restricts the number of JAX-RS providers included in the native executable to the minimum required by the application. It allows to reduce the size of the native executable.

Configuring JSON support

JSON-B

Quarkus makes it very easy to configure various JSON-B settings via CDI beans. The simplest (and suggested) approach is to define a CDI bean of type **io.quarkus.jsonb.JsonbConfigCustomizer** inside of which any JSON-B configuration can be applied.

If for example a custom serializer named **FooSerializer** for type **com.example.Foo** needs to be registered with JSON-B, the addition of a bean like the following would suffice:

```
import io.quarkus.jsonb.JsonbConfigCustomizer;
import javax.inject.Singleton;
import javax.json.bind.JsonbConfig;
import javax.json.bind.serializer.JsonbSerializer;

@Singleton
public class FooSerializerRegistrationCustomizer implements
    JsonbConfigCustomizer {

    public void customize(JsonbConfig config) {
        config.withSerializers(new FooSerializer());
    }
}
```

A more advanced option would be to directly provide a bean of **javax.json.bind.JsonbConfig** or in the extreme case to provide a bean of type **javax.json.bind.Jsonb**. If the latter approach is leveraged it is very important to manually inject and apply all **io.quarkus.jsonb.JsonbConfigCustomizer** beans in the CDI producer that produces **javax.json.bind.Jsonb**. Failure to do so will prevent JSON-B specific customizations provided by various extensions from being applied.

Jackson

As stated above, Quarkus provides the option of using **Jackson** instead of JSON-B via the use of the **quarkus-resteasy-jackson** extension.

Following the same approach as described in the previous section, Jackson's **ObjectMapper** can be

configured using a `io.quarkus.jackson.ObjectMapperCustomizer` bean. An example where a custom module needs to be registered would like so:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import io.quarkus.jackson.ObjectMapperCustomizer;
import javax.inject.Singleton;

@Singleton
public class RegisterCustomModuleCustomizer implements
ObjectMapperCustomizer {

    public void customize(ObjectMapper mapper) {
        mapper.registerModule(new CustomModule());
    }
}
```

Users can even provide their own `ObjectMapper` bean if they so choose. If this is done, it is very important to manually inject and apply all `io.quarkus.jackson.ObjectMapperCustomizer` beans in the CDI producer that produces `ObjectMapper`. Failure to do so will prevent Jackson specific customizations provided by various extensions from being applied.

Creating a frontend

Now let's add a simple web page to interact with our `FruitResource`. Quarkus automatically serves static resources located under the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, add a `fruits.html` file with the content from this [fruits.html](#) file in it.

You can now interact with your REST service:

- start Quarkus with `./mvnw compile quarkus:dev`
- open a browser to `http://localhost:8080/fruits.html`
- add new fruits to the list via the form

Building a native executable

You can build a native executable with the usual command `./mvnw package -Pnative`.

Running it is as simple as executing `./target/rest-json-quickstart-1.0-SNAPSHOT-runner`.

You can then point your browser to `http://localhost:8080/fruits.html` and use your application.

About serialization

JSON serialization libraries use Java reflection to get the properties of an object and serialize them.

When using native executables with GraalVM, all classes that will be used with reflection need to be registered. The good news is that Quarkus does that work for you most of the time. So far, we haven't registered any class, not even `Fruit`, for reflection usage and everything is working fine.

Quarkus performs some magic when it is capable of inferring the serialized types from the REST methods. When you have the following REST method, Quarkus determines that `Fruit` will be serialized:

```
@GET
@Produces("application/json")
public List<Fruit> list() {
    // ...
}
```

Quarkus does that for you automatically by analyzing the REST methods at build time and that's why we didn't need any reflection registration in the first part of this guide.

Another common pattern in the JAX-RS world is to use the `Response` object. `Response` comes with some nice perks:

- you can return different entity types depending on what happens in your method (a `Legume` or an `Error` for instance);
- you can set the attributes of the `Response` (the status comes to mind in the case of an error).

Your REST method then looks like this:

```
@GET
@Produces("application/json")
public Response list() {
    // ...
}
```

It is not possible for Quarkus to determine at build time the type included in the `Response` as the information is not available. In this case, Quarkus won't be able to automatically register for reflection the required classes.

This leads us to our next section.

Using Response

Let's create the `Legume` class which will be serialized as JSON, following the same model as for our `Fruit` class:

```
package org.acme.rest.json;

public class Legume {

    public String name;
    public String description;

    public Legume() {
    }

    public Legume(String name, String description) {
        this.name = name;
        this.description = description;
    }
}
```

Now let's create a **LegumeResource** REST service with only one method which returns the list of legumes.

This method returns a **Response** and not a list of **Legume**.


```

package org.acme.rest.json;

import java.util.Collections;
import java.util.LinkedHashSet;
import java.util.Set;

import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/legumes")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class LegumeResource {

    private Set<Legume> legumes = Collections.synchronizedSet(new
LinkedHashSet<>());

    public LegumeResource() {
        legumes.add(new Legume("Carrot", "Root vegetable, usually
orange"));
        legumes.add(new Legume("Zucchini", "Summer squash"));
    }

    @GET
    public Response list() {
        return Response.ok(legumes).build();
    }
}

```

Now let's add a simple web page to display our list of legumes. In the `src/main/resources/META-INF/resources` directory, add a `legumes.html` file with the content from this [legumes.html](#) file in it.

Open a browser to <http://localhost:8080/legumes.html> and you will see our list of legumes.

The interesting part starts when running the application as a native executable:

- create the native executable with `./mvnw package -Pnative`.
- execute it with `./target/rest-json-quickstart-1.0-SNAPSHOT-runner`
- open a browser and go to <http://localhost:8080/legumes.html>

No legumes there.

As mentioned above, the issue is that Quarkus was not able to determine the `Legume` class will require

some reflection by analyzing the REST endpoints. The JSON serialization library tries to get the list of fields of `Legume` and gets an empty list so it does not serialize the fields' data.



At the moment, when JSON-B or Jackson tries to get the list of fields of a class, if the class is not registered for reflection, no exception will be thrown. GraalVM will simply return an empty list of fields.

Hopefully, this will change in the future and make the error more obvious.

We can register `Legume` for reflection manually by adding the `@RegisterForReflection` annotation on our `Legume` class:

```
import io.quarkus.runtime.annotations.RegisterForReflection;

@registerForReflection
public class Legume {
    // ...
}
```

Let's do that and follow the same steps as before:

- hit `Ctrl+C` to stop the application
- create the native executable with `./mvnw package -Pnative`.
- execute it with `./target/rest-json-quickstart-1.0-SNAPSHOT-runner`
- open a browser and go to <http://localhost:8080/legumes.html>

This time, you can see our list of legumes.

Being reactive

You can return *reactive types* to handle asynchronous processing. Quarkus recommends the usage of `Mutiny` to write reactive and asynchronous code.

To integrate Mutiny and RESTEasy, you need to add the `quarkus-resteasy-mutiny` dependency to your project:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-mutiny</artifactId>
</dependency>
```

Then, your endpoint can return `Uni` or `Multi` instances:

```

@GET
@Path("/{name}")
public Uni<Fruit> getOne(@PathParam String name) {
    return findByName(name);
}

@GET
public Multi<Fruit> getAll() {
    return findAll();
}

```

Use **Uni** when you have a single result. Use **Multi** when you have multiple items that may be emitted asynchronously.

You can use **Uni** and **Response** to return asynchronous HTTP responses: **Uni<Response>**.

More details about Mutiny can be found in the [Getting Started with Reactive guide](#).

HTTP filters and interceptors

Both HTTP request and response can be intercepted by providing **ContainerRequestFilter** or **ContainerResponseFilter** implementations respectively. These filters are suitable for processing the metadata associated with a message: HTTP headers, query parameters, media type, and other metadata. They also have the capability to abort the request processing, for instance when the user does not have the permissions to access the endpoint.

Let's use **ContainerRequestFilter** to add logging capability to our service. We can do that by implementing **ContainerRequestFilter** and annotating it with the **@Provider** annotation:

```

package org.acme.rest.json;

import io.vertx.core.http.HttpServerRequest;
import org.jboss.logging.Logger;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.UriInfo;
import javax.ws.rs.ext.Provider;

@Provider
public class LoggingFilter implements ContainerRequestFilter {

    private static final Logger LOG = Logger.getLogger(
(LoggingFilter.class));

    @Context
    UriInfo info;

    @Context
    HttpServerRequest request;

    @Override
    public void filter(ContainerRequestContext context) {

        final String method = context.getMethod();
        final String path = info.getPath();
        final String address = request.remoteAddress().toString();

        LOG.infof("Request %s %s from IP %s", method, path,
address);
    }
}

```

Now, whenever a REST method is invoked, the request will be logged into the console:

```

2019-06-05 12:44:26,526 INFO [org.acm.res.jso.LoggingFilter]
(executor-thread-1) Request GET /legumes from IP 127.0.0.1
2019-06-05 12:49:19,623 INFO [org.acm.res.jso.LoggingFilter]
(executor-thread-1) Request GET /fruits from IP 0:0:0:0:0:0:0:1
2019-06-05 12:50:44,019 INFO [org.acm.res.jso.LoggingFilter]
(executor-thread-1) Request POST /fruits from IP 0:0:0:0:0:0:0:1
2019-06-05 12:51:04,485 INFO [org.acm.res.jso.LoggingFilter]
(executor-thread-1) Request GET /fruits from IP 127.0.0.1

```

CORS filter

[Cross-origin resource sharing](#) (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.

Quarkus comes with a CORS filter. Read the [HTTP Reference Documentation](#) to learn how to use it.

GZip Support

Quarkus comes with GZip support (even though it is not enabled by default). The following configuration knobs allow to configure GZip support.

```
quarkus.resteasy.gzip.enabled=true ①  
quarkus.resteasy.gzip.max-input=10M ②
```

- ① Enable Gzip support.
- ② Configure the upper limit on deflated request body. This is useful to mitigate potential attacks by limiting their reach. The default value is **10M**. This configuration option would recognize strings in this format (shown as a regular expression): `[0-9]+[KkMmGgTtPpEeZzYy]?`. If no suffix is given, assume bytes.

Servlet compatibility

In Quarkus, RESTEasy can either run directly on top of the Vert.x HTTP server, or on top of Undertow if you have any servlet dependency.

As a result, certain classes, such as `HttpServletRequest` are not always available for injection. Most use-cases for this particular class are covered by JAX-RS equivalents, except for getting the remote client's IP. RESTEasy comes with a replacement API which you can inject: `HttpRequest`, which has the methods `getRemoteAddress()` and `getRemoteHost()` to solve this problem.

What's Different from Jakarta EE Development

No Need for `Application` Class

Configuration via an application-supplied subclass of `Application` is supported, but not required.

Lifecycle of Resources

In Quarkus all JAX-RS resources are treated as CDI beans. It's possible to inject other beans via `@Inject`, bind interceptors using bindings such as `@Transactional`, define `@PostConstruct` callbacks, etc.

If there is no scope annotation declared on the resource class then the scope is defaulted. The default

scope can be controlled through the `quarkus.resteasy.singleton-resources` property. If set to `true` (default) then a **single instance** of a resource class is created to service all requests (as defined by `@javax.inject.Singleton`). If set to `false` then a **new instance** of the resource class is created per each request. An explicit CDI scope annotation (`@RequestScoped`, `@ApplicationScoped`, etc.) always overrides the default behavior and specifies the lifecycle of resource instances.

Conclusion

Creating JSON REST services with Quarkus is easy as it relies on proven and well known technologies.

As usual, Quarkus further simplifies things under the hood when running your application as a native executable.

There is only one thing to remember: if you use `Response` and Quarkus can't determine the beans that are serialized, you need to annotate them with `@RegisterForReflection`.