

Quarkus - Tips for writing native applications

This guide contains various tips and tricks for getting around problems that might arise when attempting to run Java applications as native executables.

Note that we differentiate two contexts where the solution applied might be different:

- in the context of an application, you will rely on configuring the `native-image` configuration by tweaking your `pom.xml`;
- in the context of an extension, Quarkus offers a lot of infrastructure to simplify all of this.

Please refer to the appropriate section depending on your context.

Supporting native in your application

GraalVM imposes a number of constraints and making your application a native executable might require a few tweaks.

Including resources

By default, when building a native executable, GraalVM will not include any of the resources that are on the classpath into the native executable it creates. Resources that are meant to be part of the native executable need to be configured explicitly.

Quarkus automatically includes the resources present in `META-INF/resources` (the web resources) but, outside of this directory, you are on your own.

To include more resources in the native executable, create a `resources-config.json` (the most common location is within `src/main/resources`) JSON file defining which resources should be included:

```
{
  "resources": [
    {
      "pattern": ".*\\.xml$"
    },
    {
      "pattern": ".*\\.json$"
    }
  ]
}
```

The patterns are valid Java regexps. Here we include all the XML files and JSON files into the native

executable.



You can find more information about this topic in [the GraalVM documentation](#).

The final order of business is to make the configuration file known to the `native-image` executable by adding the proper configuration to `application.properties`:

```
quarkus.native.additional-build-args =-
H:ResourceConfigurationFiles=resources-config.json
```

In the previous snippet we were able to simply use `resources-config.json` instead of specifying the entire path of the file simply because it was added to `src/main/resources`. If the file had been added to another directory, the proper file path would have had to be specified manually.



Multiple options may be separated by a comma. For example, one could use:

```
quarkus.native.additional-build-args =\
    -H:ResourceConfigurationFiles=resources
    -config.json,\
    -H:ReflectionConfigurationFiles=reflection
    -config.json
```

in order to ensure that various resources are included and additional reflection is registered.

If for some reason adding the aforementioned configuration to `application.properties` is not desirable, it is possible to configure the build tool to effectively perform the same operation.

When using Maven, we could use the following configuration:

```
<profiles>
  <profile>
    <id>native</id>
    <properties>
      <quarkus.package.type>native</quarkus.package.type>
      <quarkus.native.additional-build-args>
-H:ResourceConfigurationFiles=resources-
config.json</quarkus.native.additional-build-args>
    </properties>
  </profile>
</profiles>
```

When using Gradle, configuring the `buildNative` task like so suffices:

```
buildNative {  
    additionalBuildArgs = [  
        '-H:ResourceConfigurationFiles=resources-config.json'  
    ]  
}
```

Registering for reflection

When building a native executable, GraalVM operates with a closed world assumption. It analyzes the call tree and removes all the classes/methods/fields that are not used directly.

The elements used via reflection are not part of the call tree so they are dead code eliminated (if not called directly in other cases). To include these elements in your native executable, you need to register them for reflection explicitly.

This is a very common case as JSON libraries typically use reflection to serialize the objects to JSON:

```

public class Person {
    private String first;
    private String last;

    public String getFirst() {
        return first;
    }

    public void setFirst(String first) {
        this.first = first;
    }

    public String getLast() {
        return last;
    }

    public void setValue(String last) {
        this.last = last;
    }
}

@Path("/person")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class PersonResource {

    private final Jsonb jsonb;

    public PersonResource() {
        jsonb = JsonbBuilder.create(new JsonbConfig());
    }

    @GET
    public Response list() {
        return Response.ok(jsonb.fromJson("{\"first\": \"foo\", \"last\": \"bar\"}", Person.class)).build();
    }
}

```

If we were to use the code above, we would get an exception like the following when using the native executable:

```

Exception handling request to /person:
org.jboss.resteasy.spi.UnhandledException:
javax.json.bind.JsonbException: Can't create instance of a class:
class org.acme.jsonb.Person, No default constructor found

```

or if you are using Jackson:

```
com.fasterxml.jackson.databind.exc.InvalidDefinitionException: No
serializer found for class org.acme.jsonb.Person and no properties
discovered to create BeanSerializer (to avoid exception, disable
SerializationFeature.FAIL_ON_EMPTY_BEANS)
```

An even nastier possible outcome could be for no exception to be thrown, but instead the JSON result would be completely empty.

There are two different ways to fix this type of issues.

Using the `@RegisterForReflection` annotation

The easiest way to register a class for reflection is to use the `@RegisterForReflection` annotation:

```
@RegisterForReflection
public class MyClass {
}
```

Using a configuration file

Obviously, adding `@RegisterForReflection` is not possible if the class is in a third-party jar.

In this case, you can use a configuration file to register classes for reflection.

As an example, in order to register all methods of class `com.acme.MyClass` for reflection, we create `reflection-config.json` (the most common location is within `src/main/resources`)

```
[
  {
    "name" : "com.acme.MyClass",
    "allDeclaredConstructors" : true,
    "allPublicConstructors" : true,
    "allDeclaredMethods" : true,
    "allPublicMethods" : true,
    "allDeclaredFields" : true,
    "allPublicFields" : true
  }
]
```



For more details on the format of this file, please refer to [the GraalVM documentation](#).

The final order of business is to make the configuration file known to the `native-image` executable

by adding the proper configuration to `application.properties`:

```
quarkus.native.additional-build-args =-
H:ReflectionConfigurationFiles=reflection-config.json
```

In the previous snippet we were able to simply use `reflection-config.json` instead of specifying the entire path of the file simply because it was added to `src/main/resources`. If the file had been added to another directory, the proper file path would have had to be specified manually.



Multiple options may be separated by a comma. For example, one could use:

```
quarkus.native.additional-build-args =\
-H:ResourceConfigurationFiles=resources
-config.json,\
-H:ReflectionConfigurationFiles=reflection
-config.json
```

in order to ensure that various resources are included and additional reflection is registered.

If for some reason adding the aforementioned configuration to `application.properties` is not desirable, it is possible to configure the build tool to effectively perform the same operation.

When using Maven, we could use the following configuration:

```
<profiles>
  <profile>
    <id>native</id>
    <properties>
      <quarkus.package.type>native</quarkus.package.type>
      <quarkus.native.additional-build-args>
-H:ReflectionConfigurationFiles=reflection-
config.json</quarkus.native.additional-build-args>
    </properties>
  </profile>
</profiles>
```

When using Gradle, configuring the `buildNative` task like so suffices:

```
buildNative {
    additionalBuildArgs = [
        '-H:ReflectionConfigurationFiles=reflection-
config.json'
    ]
}
```

Delaying class initialization

By default, Quarkus initializes all classes at build time.

There are cases where the initialization of certain classes is done in a static block needs to be postponed to runtime. Typically omitting such configuration would result in a runtime exception like the following:

```
Error: No instances are allowed in the image heap for a class that
is initialized or reinitialized at image runtime:
sun.security.provider.NativePRNG
Trace: object java.security.SecureRandom
method
com.amazonaws.services.s3.model.CryptoConfiguration.<init>(CryptoMo
de)
Call path from entry point to
com.amazonaws.services.s3.model.CryptoConfiguration.<init>(CryptoMo
de):
```

If you need to delay the initialization of a class, you can use the `--initialize-at-run-time=<package or class>` configuration knob.

It should be added to the `native-image` configuration using an `<additionalBuildArg>` as shown in the examples above.



You can find more information about all this in [the GraalVM documentation](#).

When multiple classes or packages need to be specified via the `quarkus.native.additional-build-args` configuration property, the `,` symbol needs to be escaped. An example of this is the following:



```
quarkus.native.additional-build-args=--initialize-at-
run-
time=com.example.SomeClass\\,org.acme.SomeOtherClass
```

Managing Proxy Classes

While writing native application you'll need to define proxy classes at image build time by specifying the list of interfaces that they implement.

In such a situation, the error you might encounter is:

```
com.oracle.svm.core.jdk.UnsupportedFeatureError: Proxy class
defined by interfaces [interface
org.apache.http.conn.HttpClientConnectionManager, interface
org.apache.http.pool.ConnPoolControl, interface
com.amazonaws.http.conn.Wrapped] not found. Generating proxy
classes at runtime is not supported. Proxy classes need to be
defined at image build time by specifying the list of interfaces
that they implement. To define proxy classes use
-H:DynamicProxyConfigurationFiles=<comma-separated-config-files>
and -H:DynamicProxyConfigurationResources=<comma-separated-config
-resources> options.
```

Solving this issue requires adding the `-H:DynamicProxyConfigurationResources=<comma-separated-config-resources>` option and to provide a dynamic proxy configuration file. You can find all the information about the format of this file in [the GraalVM documentation](#).

Supporting native in a Quarkus extension

Supporting native in a Quarkus extension is even easier as Quarkus provides a lot of tools to simplify all this.



Everything described here will only work in the context of Quarkus extensions, it won't work in an application.

Register reflection

Quarkus makes registration of reflection in an extension a breeze by using `ReflectiveClassBuildItem`, thus eliminating the need for a JSON configuration file.

To register a class for reflection, one would need to create a Quarkus processor class and add a build step that registers reflection:


```
public class SaxParserProcessor {

    @BuildStep
    ReflectiveClassBuildItem reflection() {
        // since we only need reflection to the constructor of the
        // class, we can specify `false` for both the methods and the fields
        // arguments.
        return new ReflectiveClassBuildItem(false, false,
            "com.sun.org.apache.xerces.internal.parsers.SAXParser");
    }

}
```



More information about reflection in GraalVM can be found [here](#).

Alternative with @RegisterForReflection

As for applications, you can also use the `@RegisterForReflection` annotation if the class is in your extension and not in a third-party jar.

Including resources

In the context of an extension, Quarkus eliminates the need for a JSON configuration file by allowing extension authors to specify a `NativeImageResourceBuildItem`:

```
public class ResourcesProcessor {

    @BuildStep
    NativeImageResourceBuildItem nativeImageResourceBuildItem() {
        return new NativeImageResourceBuildItem("META-INF/extra.properties");
    }

}
```



For more information about GraalVM resource handling in native executables please refer to [the GraalVM documentation](#).

Delay class initialization

Quarkus simplifies things by allowing extensions authors to simply register a `RuntimeInitializedClassBuildItem`. A simple example of doing so could be:

```
public class S3Processor {

    @BuildStep
    RuntimeInitializedClassBuildItem cryptoConfiguration() {
        return new RuntimeInitializedClassBuildItem
(CryptoConfiguration.class.getCanonicalName());
    }

}
```

Using such a construct means that a `--initialize-at-run-time` option will automatically be added to the `native-image` command line.



For more information about `--initialize-at-run-time`, please read [the GraalVM documentation](#).

Managing Proxy Classes

Very similarly, Quarkus allows extensions authors to register a `NativeImageProxyDefinitionBuildItem`. An example of doing so is:

```
public class S3Processor {

    @BuildStep
    NativeImageProxyDefinitionBuildItem httpProxies() {
        return new NativeImageProxyDefinitionBuildItem(
"org.apache.http.conn.HttpClientConnectionManager",
        "org.apache.http.pool.ConnPoolControl",
"com.amazonaws.http.conn.Wrapped");
    }

}
```

Using such a construct means that a `-H:DynamicProxyConfigurationResources` option will automatically be added to the `native-image` command line.



For more information about Proxy Classes you can read [the GraalVM documentation](#).