Quarkus - Writing Your Own Extension

Quarkus extensions add a new developer focused behavior to the core offering, and consist of two distinct parts, buildtime augmentation and runtime container. The augmentation part is responsible for all metadata processing, such as reading annotations, XML descriptors etc. The output of this augmentation phase is recorded bytecode which is responsible for directly instantiating the relevant runtime services.

This means that metadata is only processed once at build time, which both saves on startup time, and also on memory usage as the classes etc that are used for processing are not loaded (or even present) in the runtime JVM.

1. Extension philosophy

This section is a work in progress and gathers the philosophy under which extensions should be designed and written.

1.1. Why an extension framework

Quarkus's mission is to transform your entire application including the libraries it uses, into an artifact that uses significantly less resources than traditional approaches. These can then be used to build native applications using GraalVM. To do this you need to analyze and understand the full "closed world" of the application. Without the full and complete context, the best that can be achieved is partial and limited generic support. By using the Quarkus extension approach, we can bring Java applications in line with memory footprint constrained environments like Kubernetes or cloud platforms.

The Quarkus extension framework results in significantly improved resource utilization even when GraalVM is not used (e.g. in HotSpot). Let's list the actions an extension performs:

- Gather build time metadata and generate code

  - This part has nothing to do with GraalVM, it is how Quarkus starts frameworks "at build time"

  - The extension framework facilitates reading metadata, scanning classes as well as generating classes as needed

  - A small part of the extension work is executed at runtime via the generated classes, while the bulk of the work is done at build time (called deployment time)

- Enforce opinionated and sensible defaults based on the close world view of the application (e.g. an application with no `@Entity` does not need to start Hibernate ORM)

- An extension hosts Substrate VM code substitution so that libraries can run on GraalVM

  - Most changes are pushed upstream to help the underlying library run on GraalVM

  - Not all changes can be pushed upstream, extensions host Substrate VM substitutions - which is a form of code patching - so that libraries can run

- Host Substrate VM code substitution to help dead code elimination based on the application needs
  - This is application dependant and cannot really be shared in the library itself
  - For example, Quarkus optimizes the Hibernate code because it knows it only needs a specific connection pool and cache provider
- Send metadata to GraalVM for example classes in need of reflection
  - This information is not static per library (e.g. Hibernate) but the framework has the semantic knowledge and knows which classes need to have reflection (for example @Entity classes)

1.2. Favor build time work over runtime work

As much as possible favor doing work at build time (deployment part of the extension) as opposed to let the framework do work at startup time (runtime). The more is done there, the smaller Quarkus applications using that extension will be and the faster they will load.

1.3. How to expose configuration

Quarkus simplifies the most common usages. This means that its defaults might be different than the library it integrates.

To make the simple experience easiest, unify the configuration in `application.properties` via MicroProfile Config. Avoid library specific configuration files, or at least make them optional: e.g. `persistence.xml` for Hibernate ORM is optional.

Extensions should see the configuration holistically as a Quarkus application instead of focusing on the library experience. For example `quarkus.database.url` and friends are shared between extensions as defining a database access is a shared task (instead of a `hibernate.` property for example). The most useful configuration options should be exposed as `quarkus.[extension].` instead of the natural namespace of the library. Less common properties can live in the library namespace.

To fully enable the close world assumptions that Quarkus can optimize best, it is better to consider configuration options as build time settled vs overridable at runtime. Of course properties like host, port, password should be overridable at runtime. But many properties like enable caching or setting the JDBC driver can safely require a rebuild of the application.

1.4. Expose your components via CDI

Since CDI is the central programming model when it comes to component composition, frameworks and extensions should expose their components as beans that are easily consumable by user applications. For example, Hibernate ORM exposes `EntityManagerFactory` and `EntityManager` beans, the connection pool exposes `DataSource` beans etc. Extensions must register these bean definitions at build time.

1.4.1. Beans backed by classes

An extension can produce an `AdditionalBeanBuildItem` to instruct the container to read a bean definition from a class as if it was part of the original application:

Bean Class Registered by `AdditionalBeanBuildItem`

```
@ApplicationScoped ①
public class Echo {

    public String echo(String val) {
        return val;
    }
}
```

① If a bean registered by an `AdditionalBeanBuildItem` does not specify a scope then `@Dependent` is assumed.

All other beans can inject such a bean:

Bean Injecting a Bean Produced by an `AdditionalBeanBuildItem`

```
@Path("/hello")
public class ExampleResource {

    @Inject
    Echo echo;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(String foo) {
        return echo.echo(foo);
    }
}
```

And vice versa - the extension bean can inject application beans and beans provided by other extensions:

Extension Bean Injection Example

```
@ApplicationScoped
public class Echo {

    @Inject
    DataSource dataSource;   ①

    @Inject
    Instance<List<String>> listsOfStrings; ②

    //...
}
```

① Inject a bean provided by other extension.

② Inject all beans matching the type `List<String>`.

1.4.2. Bean initialization

Some components may require additional initialization based on information collected during augmentation. The most straightforward solution is to obtain a bean instance and call a method directly from a build step. However, it is *illegal* to obtain a bean instance during the augmentation phase. The reason is that the CDI container is not started yet. It's started during the Static init bootstrap phase.

> `BUILD_AND_RUN_TIME_FIXED` and `RUN_TIME` config roots can be injected in any bean. `RUN_TIME` config roots should only be injected after the bootstrap though.

It is possible to invoke a bean method from a recorder method though. If you need to access a bean in a `@Record(STATIC_INIT)` build step then is must either depend on the `BeanContainerBuildItem` or wrap the logic in a `BeanContainerListenerBuildItem`. The reason is simple - we need to make sure the CDI container is fully initialized and started. However, it is safe to expect that the CDI container is fully initialized and running in a `@Record(RUNTIME_INIT)` build step. You can obtain a reference to the container via `CDI.current()` or Quarkus-specific `Arc.container()`.

> Don't forget to make sure the bean state guarantees the visibility, e.g. via the `volatile` keyword.

> There is one significant drawback of this "late initialization" approach. An *uninitialized* bean may be accessed by other extensions or application components that are instantiated during bootstrap. We'll cover a more robust solution in the Synthetic beans.

1.4.3. Default beans

A very useful pattern of creating such beans but also giving application code the ability to easily override some of the beans with custom implementations, is to use the `@DefaultBean` that Quarkus provides. This is best explained with an example.

Let us assume that the Quarkus extension needs to provide a `Tracer` bean which application code is meant to inject into its own beans.

```
@Dependent
public class TracerConfiguration {

    @Produces
    public Tracer tracer(Reporter reporter, Configuration
configuration) {
        return new Tracer(reporter, configuration);
    }

    @Produces
    @DefaultBean
    public Configuration configuration() {
        // create a Configuration
    }

    @Produces
    @DefaultBean
    public Reporter reporter(){
        // create a Reporter
    }
}
```

If for example application code wants to use `Tracer`, but also needs to use a custom `Reporter` bean, such a requirement could easily be done using something like:

```
@Dependent
public class CustomTracerConfiguration {

    @Produces
    public Reporter reporter(){
        // create a custom Reporter
    }
}
```

1.4.4. How to Override a Bean Defined by a Library/Quarkus Extension that doesn't use @DefaultBean

Although `@DefaultBean` is the recommended approach, it is also possible for application code to override beans provided by an extension by marking beans as a CDI `@Alternative` and including `@Priority` annotation. Let's show a simple example. Suppose we work on an imaginary "quarkus-parser" extension and we have a default bean implementation:

```
@Dependent
class Parser {

  String[] parse(String expression) {
    return expression.split("::");
  }
}
```

And our extension also consumes this parser:

```
@ApplicationScoped
class ParserService {

  @Inject
  Parser parser;

  //...
}
```

Now, if a user or even some other extension needs to override the default implementation of the Parser the simplest solution is to use CDI @Alternative + @Priority:

```
@Alternative ①
@Priority(1) ②
@Singleton
class MyParser extends Parser {

  String[] parse(String expression) {
    // my super impl...
  }
}
```

① MyParser is an alternative bean.

② Enables the alternative. The priority could be any number to override the default bean but if there are multiple alternatives the highest priority wins.

> CDI alternatives are only considered during injection and type-safe resolution. For example the default implementation would still receive observer notifications.

1.4.5. Synthetic beans

Sometimes it is very useful to be able to register a synthetic bean. Bean attributes of a synthetic bean are not derived from a java class, method or field. Instead, the attributes are specified by an extension.

Since the CDI container does not control the instantiation of a synthetic bean the dependency injection and other services (such as interceptors) are not supported. In other words, it's up to the extension to provide all required services to a synthetic bean instance.

There are several ways to register a synthetic bean in Quarkus. In this chapter, we will cover a use case that can be used to initialize extension beans in a safe manner (compared to Bean initialization).

The `SyntheticBeanBuildItem` can be used to register a synthetic bean:

- whose instance can be easily produced through a recorder,

- to provide a "context" bean that holds all the information collected during augmentation so that the real components do not need any "late initialization" because they can inject the context bean directly.

Instance Produced Through Recorder

```
@BuildStep
@Record(STATIC_INIT)
SyntheticBeanBuildItem syntheticBean(TestRecorder recorder) {
    return
SyntheticBeanBuildItem.configure(Foo.class).scope(Singleton.class)
                .runtimeValue(recorder.createFoo("parameters are
recorder in the bytecode")) ①
                .done();
}
```

① The string value is recorded in the bytecode and used to initialize the instance of `Foo`.

"Context" Holder

```
@BuildStep
@Record(STATIC_INIT)
SyntheticBeanBuildItem syntheticBean(TestRecorder recorder) {
    return
SyntheticBeanBuildItem.configure(TestContext.class).scope(Singleton
.class)
                .runtimeValue(recorder.createContext("parameters
are recorder in the bytecode")) ①
                .done();
}
```

① The "real" components can inject the `TestContext` directly.

1.5. Some types of extensions

There exist multiple stereotypes of extension, let's list a few.

Bare library running

This is the less sophisticated extension. It consists of a set of patches to make sure a library runs on GraalVM. If possible, contribute these patches upstream, not in extensions. Second best is to write Substrate VM substitutions, which are patches applied during native image compilation.

Get a framework running

A framework at runtime typically reads configuration, scan the classpath and classes for metadata (annotations, getters etc), build a metamodel on top of which it runs, find options via the service loader pattern, prepare invocation calls (reflection), proxy interfaces, etc.
These operations should be done at build time and the metamodel be passed to the recorder DSL that will generate classes that will be executed at runtime and boot the framework.

Get a CDI portable extension running

The CDI portable extension model is very flexible. Too flexible to benefit from the build time boot promoted by Quarkus. Most extension we have seen do not make use of these extreme flexibilities capabilities. The way to port a CDI extension to Quarkus is to rewrite it as a Quarkus extension which will define the various beans at build time (deployment time in extension parlance).

2. Technical aspect

2.1. Three Phases of Bootstrap and Quarkus Philosophy

There are three distinct bootstrap phases of a Quarkus app:

Augmentation

This is the first phase, and is done by the Build Step Processors. These processors have access to Jandex annotation information and can parse any descriptors and read annotations, but should not attempt to load any application classes. The output of these build steps is some recorded bytecode, using an extension of the ObjectWeb ASM project called Gizmo(ext/gizmo), that is used to actually bootstrap the application at runtime. Depending on the `io.quarkus.deployment.annotations.ExecutionTime` value of the `@io.quarkus.deployment.annotations.Record` annotation associated with the build step, the step may be run in a different JVM based on the following two modes.

Static Init

If bytecode is recorded with `@Record(STATIC_INIT)` then it will be executed from a static init method on the main class. For a native executable build, this code is executed in a normal JVM as part of the native build process, and any retained objects that are produced in this stage will be directly serialized into the native executable via an image mapped file. This means that if a framework can boot in this phase then it will have its booted state directly written to the image, and so the boot code does not need to be executed when the image is started.

There are some restrictions on what can be done in this stage as the Substrate VM disallows some objects in the native executable. For example you should not attempt to listen on a port or start threads in this phase. In addition, it is disallowed to read run time configuration during static initialization.

In non-native pure JVM mode, there is no real difference between Static and Runtime Init, except that Static Init is always executed first. This mode benefits from the same build phase augmentation as native mode as the descriptor parsing and annotation scanning are done at build time and any associated class/framework dependencies can be removed from the build output jar. In servers like WildFly, deployment related classes such as XML parsers hang around for the life of the application, using up valuable memory. Quarkus aims to eliminate this, so that the only classes loaded at runtime are actually used at runtime.

As an example, the only reason that a Quarkus application should load an XML parser is if the user is using XML in their application. Any XML parsing of configuration should be done in the Augmentation phase.

Runtime Init

If bytecode is recorded with `@Record(RUNTIME_INIT)` then it is executed from the application's main method. This code will be run on native executable boot. In general as little code as possible should be executed in this phase, and should be restricted to code that needs to open ports etc.

Pushing as much as possible into the `@Record(STATIC_INIT)` phase allows for two different optimizations:

1. In both native executable and pure JVM mode this allows the app to start as fast as possible since processing was done during build time. This also minimizes the classes/native code needed in the application to pure runtime related behaviors.

2. Another benefit with native executable mode is that Substrate can more easily eliminate features that are not used. If features are directly initialized via bytecode, Substrate can detect that a method is never called and eliminate that method. If config is read at runtime, Substrate cannot reason about the contents of the config and so needs to keep all features in case they are required.

2.2. Maven setup

Your extension project should be setup as a multi-module project with two submodules:

1. A deployment time submodule that handles the build time processing and bytecode recording.

2. A runtime submodule that contains the runtime behavior that will provide the extension behavior in the native executable or runtime JVM.

> You may want to use the `create-extension` mojo of `io.quarkus:quarkus-maven-plugin` to create these Maven modules - see the next section.

Your runtime artifact should depend on `io.quarkus:quarkus-core`, and possibly the runtime artifacts of other Quarkus modules if you want to use functionality provided by them. You will also need to include the `io.quarkus:quarkus-bootstrap-maven-plugin` to generate the Quarkus extension descriptor included into the runtime artifact, if you are using the Quarkus parent pom it will automatically inherit the correct configuration. Futhermore, you'll need to configure the `maven-compiler-plugin` to detect the `quarkus-extension-processor` annotation processor.

> By convention the deployment time artifact has the `-deployment` suffix, and the runtime artifact has no suffix (and is what the end user adds to their project).

```xml
<dependencies>
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-core</artifactId>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-bootstrap-maven-plugin</artifactId>
            <!-- Executions configuration can be inherited from
quarkus-build-parent -->
            <executions>
                <execution>
                    <goals>
                        <goal>extension-descriptor</goal>
                    </goals>
                    <configuration>

<deployment>${project.groupId}:${project.artifactId}-
deployment:${project.version}</deployment>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>io.quarkus</groupId>
                        <artifactId>quarkus-extension-
processor</artifactId>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

> ℹ The above `maven-compiler-plugin` configuration requires version 3.5+.

> ⚠️ Under no circumstances can the runtime module depend on a deployment artifact. This would result in pulling all the deployment time code into runtime scope, which defeats the purpose of having the split.

Your deployment time module should depend on `io.quarkus:quarkus-core-deployment`, your runtime artifact, and possibly the deployment artifacts of other Quarkus modules if you want to use functionality provided by them. You will also need to configure the `maven-compiler-plugin` to detect the `quarkus-extension-processor` annotation processor.

```xml
<dependencies>
    <dependency>
        <groupId>io.quarkus</groupId>
        <artifactId>quarkus-core-deployment</artifactId>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>io.quarkus</groupId>
                        <artifactId>quarkus-extension-processor</artifactId>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

2.2.1. Create new extension modules using Maven

The `create-extension` mojo of `io.quarkus:quarkus-maven-plugin` can be used to generate stubs of Maven modules needed for implementing a new Quarkus extension.

> ⚠️ This mojo can be currently used only for adding extensions to an established source tree hosting multiple extensions in one subdirectory, such as Quarkus or Camel Quarkus. Creating extension projects from scratch is not supported yet.

As and example, let's add a new extension called `my-ext` to the Quarkus source tree:

```
git clone https://github.com/quarkusio/quarkus.git
cd quarkus
cd extensions
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create-extension -N
\
    -Dquarkus.artifactIdBase=my-ext \
    -Dquarkus.artifactIdPrefix=quarkus- \
    -Dquarkus.nameBase="My Extension"
```

The above sequence of commands does the following:

- Creates four new Maven modules:

  - `quarkus-my-ext-parent` in the `extensions/my-ext` directory

  - `quarkus-my-ext` in the `extensions/my-ext/runtime` directory

  - `quarkus-my-ext-deployment` in the `extensions/my-ext/deployment` directory; a basic `MyExtProcessor` class is generated in this module.

  - `quarkus-my-ext-integration-test` in the `integration-tests/my-ext/deployment` directory; an empty JAX-RS Resource class and two test classes (for JVM mode and native mode) are generated in this module.

- Links these three modules where necessary:

  - `quarkus-my-ext-parent` is added to the `<modules>` of `quarkus-extensions-parent`

  - `quarkus-my-ext` is added to the `<dependencyManagement>` of the runtime BOM (Bill of Materials) `bom/runtime/pom.xml`

  - `quarkus-my-ext-deployment` is added to the `<dependencyManagement>` of the deployment BOM (Bill of Materials) `bom/deployment/pom.xml`

  - `quarkus-my-ext-integration-test` is added to the `<modules>` of `quarkus-integration-tests-parent`

A Maven build performed immediately after generating the modules should fail due to a `fail()` assertion in one of the test classes.

There is one step (specific to the Quarkus source tree) that you should do manually when creating a new extension: create a `quarkus-extension.yaml` file that describe your extension inside the runtime module `src/main/resources/META-INF` folder.

This is the `quarkus-extension.yaml` of the `quarkus-agroal` extension, you can use it as an example:

```
name: "Agroal — Database connection pool"
metadata:
  keywords:
  — "agroal"
  — "database-connection-pool"
  — "datasource"
  — "jdbc"
  guide: "https://quarkus.io/guides/datasource"
  categories:
  — "data"
  status: "stable"
```

Note that the parameters of the mojo that will be constant for all the extensions added to this source tree are configured in `extensions/pom.xml` so that they do not need to be passed on the command line each time a new extension is added:

```xml
<plugin>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-maven-plugin</artifactId>
    <version>${quarkus.version}</version>
    <inherited>false</inherited>
    <!-- Settings for stubbing new extensions via
            ./mvnw quarkus:create-extension -N
-Dquarkus.artifactIdBase=my-ext -Dquarkus.nameBase="My Extension"
    -->
    <configuration>
        <namePrefix xml:space="preserve">Quarkus - </namePrefix>
        <runtimeBomPath>../bom/runtime/pom.xml</runtimeBomPath>

<deploymentBomPath>../bom/deployment/pom.xml</deploymentBomPath>
        <itestParentPath>../integration-
tests/pom.xml</itestParentPath>
    </configuration>
</plugin>
```

The `nameBase` parameter of the mojo is optional. If you do not specify it on the command line, the plugin will derive it from `artifactIdBase` by replacing dashes with spaces and uppercasing each token. So you may consider omitting explicit `nameBase` in some cases.

Please refer to CreateExtensionMojo JavaDoc for all the available options of the mojo.

2.3. Build Step Processors

Work is done at augmentation time by *build steps* which produce and consume *build items*. The build

steps found in the deployment modules that correspond to the extensions in the project build are automatically wired together and executed to produce the final build artifact(s).

2.3.1. Build steps

A *build step* is a method which is annotated with the `@io.quarkus.deployment.annotations.BuildStep` annotation. Each build step may consume items that are produced by earlier stages, and produce items that can be consumed by later stages. Build steps are normally only run when they produce a build item that is ultimately consumed by another step.

Build steps are normally placed on plain classes within an extension's deployment module. The classes are automatically instantiated during the augment process and utilize injection.

2.3.2. Build items

Build items are concrete, final subclasses of the abstract `io.quarkus.builder.item.BuildItem` class. Each build item represents some unit of information that must be passed from one stage to another. The base `BuildItem` class may not itself be directly subclassed; rather, there are abstract subclasses for each of the kinds of build item subclasses that *may* be created: simple, multi, and empty.

Think of build items as a way for different extensions to communicate with one another. For example, a build item can:

- expose the fact that a database configuration exists

- consume that database configuration (e.g. a connection pool extension or an ORM extension)

- ask an extension to do work for another extension: e.g. an extension wanting to define a new CDI bean and asking the ArC extension to do so

This is a very flexible mechanism.

> `BuildItem` instances should be immutable, as the producer/consumer model does not allow for mutation to be correctly ordered. This is not enforced but failure to adhere to this rule can result in race conditions.

2.3.2.1. Simple build items

Simple build items are final classes which extend `io.quarkus.builder.item.SimpleBuildItem`. Simple build items may only be produced by one step in a given build; if multiple steps in a build declare that they produce the same simple build item, an error is raised. Any number of build steps may consume a simple build item. A build step which consumes a simple build item will always run *after* the build step which produced that item.

Example of a single build item

```java
/**
 * The build item which represents the Jandex index of the
application,
 * and would normally be used by many build steps to find usages
 * of annotations.
 */
public final class ApplicationIndexBuildItem extends
SimpleBuildItem {

    private final Index index;

    public ApplicationIndexBuildItem(Index index) {
        this.index = index;
    }

    public Index getIndex() {
        return index;
    }
}
```

2.3.2.2. Multi build items

Multiple or "multi" build items are final classes which extend `io.quarkus.builder.item.MultiBuildItem`. Any number of multi build items of a given class may be produced by any number of steps, but any steps which consume multi build items will only run *after* every step which can produce them has run.

Example of a multiple build item

```
public final class ServiceWriterBuildItem extends MultiBuildItem {
    private final String serviceName;
    private final List<String> implementations;

    public ServiceWriterBuildItem(String serviceName, String...
implementations) {
        this.serviceName = serviceName;
        // Make sure it's immutable
        this.implementations = Collections.unmodifiableList(
            Arrays.asList(
                implementations.clone()
            )
        );
    }

    public String getServiceName() {
        return serviceName;
    }

    public List<String> getImplementations() {
        return implementations;
    }
}
```

Example of multiple build item usage

```
/**
 * This build step produces a single multi build item that declares
two
 * providers of one configuration-related service.
 */
@BuildStep
public ServiceWriterBuildItem registerOneService() {
    return new ServiceWriterBuildItem(
        Converter.class.getName(),
        MyFirstConfigConverterImpl.class.getName(),
        MySecondConfigConverterImpl.class.getName()
    );
}

/**
 * This build step produces several multi build items that declare
multiple
 * providers of multiple configuration-related services.
 */
@BuildStep
```

```java
public void registerSeveralServices(
    BuildProducer<ServiceWriterBuildItem> providerProducer
) {
    providerProducer.produce(new ServiceWriterBuildItem(
        Converter.class.getName(),
        MyThirdConfigConverterImpl.class.getName(),
        MyFourthConfigConverterImpl.class.getName()
    ));
    providerProducer.produce(new ServiceWriterBuildItem(
        ConfigSource.class.getName(),
        MyConfigSourceImpl.class.getName()
    ));
}

/**
 * This build step aggregates all the produced service providers
 * and outputs them as resources.
 */
@BuildStep
public void produceServiceFiles(
    List<ServiceWriterBuildItem> items,
    BuildProducer<GeneratedResourceBuildItem> resourceProducer
) throws IOException {
    // Aggregate all of the providers

    Map<String, Set<String>> map = new HashMap<>();
    for (ServiceWriterBuildItem item : items) {
        String serviceName = item.getName();
        for (String implName : item.getImplementations()) {
            map.computeIfAbsent(
                serviceName,
                (k, v) -> new LinkedHashSet<>()
            ).add(implName);
        }
    }

    // Now produce the resource(s) for the SPI files
    for (Map.Entry<String, Set<String>> entry : map.entrySet()) {
        String serviceName = entry.getKey();
        try (ByteArrayOutputStream os = new
ByteArrayOutputStream()) {
            try (OutputStreamWriter w = new OutputStreamWriter(os,
StandardCharsets.UTF_8)) {
                for (String implName : entry.getValue()) {
                    w.write(implName);
                    w.write(System.lineSeparator());
                }
            }
            w.flush();
```

```
            resourceProducer.produce(
                new GeneratedResourceBuildItem(
                    "META-INF/services/" + serviceName,
                    os.toByteArray()
                )
            );
        }
    }
}
```

2.3.2.3. Empty build items

Empty build items are final (usually empty) classes which extend `io.quarkus.builder.item.EmptyBuildItem`. They represent build items that don't actually carry any data, and allow such items to be produced and consumed without having to instantiate empty classes. They cannot themselves be instantiated.

Example of an empty build item

```
public final class NativeImageBuildItem extends EmptyBuildItem {
    // empty
}
```

Empty build items can represent "barriers" which can impose ordering between steps. They can also be used in the same way that popular build systems use "pseudo-targets", which is to say that the build item can represent a conceptual goal that does not have a concrete representation.

Example of usage of an empty build item in a "pseudo-target" style

```
/**
 * Contrived build step that produces the native image on disk.
The main augmentation
 * step (which is run by Maven or Gradle) would be declared to
consume this empty item,
 * causing this step to be run.
 */
@BuildStep
@Produce(NativeImageBuildItem.class)
void produceNativeImage() {
    // ...
    // (produce the native image)
    // ...
}
```

Example of usage of an empty build item in a "barrier" style

```
/**
 * This would always run after {@link #produceNativeImage()}
completes, producing
 * an instance of {@code SomeOtherBuildItem}.
 */
@BuildStep
@Consume(NativeImageBuildItem.class)
SomeOtherBuildItem secondBuildStep() {
    return new SomeOtherBuildItem("foobar");
}
```

2.3.3. Injection

Classes which contain build steps support the following types of injection:

- Constructor parameter injection

- Field injection

- Method parameter injection (for build step methods only)

Build step classes are instantiated and injected for each build step invocation, and are discarded afterwards. State should only be communicated between build steps by way of build items, even if the steps are on the same class.

> Final fields are not considered for injection, but can be populated by way of constructor parameter injection if desired. Static fields are never considered for injection.

The types of values that can be injected include:

- Build items produced by previous build steps

- Build producers to produce items for subsequent build steps

- Configuration root types

- Template objects for bytecode recording

> Objects which are injected into a build step method or its class *must not* be used outside of that method's execution.

> Injection is resolved at compile time via an annotation processor, and the resulting code does not have permission to inject private fields or invoke private methods.

2.3.4. Producing values

A build step may produce values for subsequent steps in several possible ways:

- By returning a simple build item or multi build item instance

- By returning a `List` of a multi build item class

- By injecting a `BuildProducer` of a simple or multi build item class

- By annotating the method with `@io.quarkus.deployment.annotations.Produce`, giving the class name of a empty build item

If a simple build item is declared on a build step, it *must* be produced during that build step, otherwise an error will result. Build producers which are injected into steps *must not* be used outside of that step.

Note that a `@BuildStep` method will only be called if it produces something that another consumer or the final output requires. If there is no consumer for a particular item then it will not be produced. What is required will depend on the final target that is being produced. For example, when running in developer mode the final output will not ask for GraalVM-specific build items such as `ReflectiveClassBuildItem`, so methods that only produce these items will not be invoked.

2.3.5. Consuming values

A build step may consume values from previous steps in the following ways:

- By injecting a simple build item

- By injecting an `Optional` of a simple build item class

- By injecting a `List` of a multi build item class

- By annotating the method with `@io.quarkus.deployment.annotations.Consume`, giving the class name of a empty build item

Normally it is an error for a step which is included to consume a simple build item that is not produced by any other step. In this way, it is guaranteed that all of the declared values will be present and non-`null` when a step is run.

Sometimes a value isn't necessary for the build to complete, but might inform some behavior of the build step if it is present. In this case, the value can be optionally injected.

> ℹ Multi build values are always considered *optional*. If not present, an empty list will be injected.

2.3.5.1. Weak value production

Normally a build step is included whenever it produces any build item which is in turn consumed by any other build step. In this way, only the steps necessary to produce the final artifact(s) are included, and steps which pertain to extensions which are not installed or which only produce build items which are not relevant for the given artifact type are excluded.

For cases where this is not desired behavior, the `@io.quarkus.deployment.annotations.Weak` annotation may be used. This annotation indicates that the build step should not automatically be included solely on the basis of producing the annotated value.

Example of producing a build item weakly

```
/**
 * This build step is only run if something consumes the
ExecutorClassBuildItem.
 */
@BuildStep
void createExecutor(
        @Weak BuildProducer<GeneratedClassBuildItem> classConsumer,
        BuildProducer<ExecutorClassBuildItem> executorClassConsumer
) {
        ClassWriter cw = new ClassWriter(Gizmo.ASM_API_VERSION);
        String className = generateClassThatCreatesExecutor(cw); ①
        classConsumer.produce(new GeneratedClassBuildItem(true,
className, cw.toByteArray()));
        executorClassConsumer.produce(new
ExecutorClassBuildItem(className));
}
```

① This method (not provided in this example) would generate the class using the ASM API.

Certain types of build items are generally always consumed, such as generated classes or resources. An extension might produce a build item along with a generated class to facilitate the usage of that build item. Such a build step would use the @Weak annotation on the generated class build item, while normally producing the other build item. If the other build item is ultimately consumed by something, then the step would run and the class would be generated. If nothing consumes the other build item, the step would not be included in the build process.

In the example above, GeneratedClassBuildItem would only be produced if ExecutorClassBuildItem is consumed by some other build step.

Note that when using bytecode recording, the implicitly generated class can be declared to be weak by using the optional attribute of the @io.quarkus.deployment.annotations.Record annotation.

Example of using a bytecode recorder where the generated class is weakly produced

```
/**
 * This build step is only run if something consumes the
ExecutorBuildItem.
 */
@BuildStep
@Record(value = ExecutionTime.RUNTIME_INIT, optional = true) ①
ExecutorBuildItem createExecutor( ②
        ExecutorTemplate executorTemplate,
        ThreadPoolConfig threadPoolConfig
) {

    return new ExecutorBuildItem(
        setupTemplate.setupRunTime(
            shutdownContextBuildItem,
            threadPoolConfig,
            launchModeBuildItem.getLaunchMode()
        )
    );
}
```

① Note the `optional` attribute.

② This example is using recorder proxies; see the section on bytecode recording for more information.

2.3.6. Capabilities

The `@BuildStep` annotation has a `providesCapabilities` property that can be used to provide capability information to other extensions about what is present in the current application. Capabilities are simply strings that are used to describe an extension. Capabilities should generally be named after an extensions root package, for example the transactions extension will provide `io.quarkus.transactions`.

To check if a capability is present you can inject the `io.quarkus.deployment.Capabilities` object and call `isCapabilityPresent`.

Capabilities should be used when checking for the presence of an extension rather than class path based checks.

2.3.7. Application Archives

The `@BuildStep` annotation can also register marker files that determine which archives on the class path are considered to be 'Application Archives', and will therefore get indexed. This is done via the `applicationArchiveMarkers`. For example the ArC extension registers `META-INF/beans.xml`, which means that all archives on the class path with a `beans.xml` file will be indexed.

> **ℹ** `BuildStep.applicationArchiveMarkers()` is deprecated and will be removed at some point post Quarkus 1.1. Extensions are encouraged to use `io.quarkus.deployment.builditem.AdditionalApplicationArchiveMarkerBuildItem` instead.

2.3.8. Using Thread's Context Class Loader

The build step will be run with a TCCL that can load user classes from the deployment in a transformer-safe way. This class loader only lasts for the life of the augmentation, and is discarded afterwards. The classes will be loaded again in a different class loader at runtime. This means that loading a class during augmentation does not stop it from being transformed when running in the development/test mode.

2.3.9. Adding external JARs to the indexer with IndexDependencyBuildItem

The index of scanned classes will not automatically include your external class dependencies. To add dependencies, create a `@BuildStep` that produces `IndexDependencyBuildItem` objects, for a `groupId` and `artifactId`.

> **ℹ** It is important to specify all the required artifacts to be added to the indexer. No artifacts are implicitly added transitively.

The `Amazon Alexa` extension adds dependent libraries from the Alexa SDK that are used in Jackson JSON transformations, in order for the reflective classes to identified and included at `BUILD_TIME`.

```
    @BuildStep
    void addDependencies(BuildProducer<IndexDependencyBuildItem>
indexDependency) {
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-runtime"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-model"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-lambda-
support"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-servlet-
support"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-dynamodb-
persistence-adapter"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-apache-
client"));
        indexDependency.produce(new
IndexDependencyBuildItem("com.amazon.alexa", "ask-sdk-model-
runtime"));
    }
```

With the artifacts added to the Jandex indexer, you can now search the index to identify classes implementing an interface, sub-classes of a specific class, or classes with a target annotation.

For example, the Jackson extension uses code like below to search for annotations used in JSON deserialization, and add them to the reflective hierarchy for BUILD_TIME analysis.

```
    DotName JSON_DESERIALIZE =
DotName.createSimple(JsonDeserialize.class.getName());

    IndexView index = combinedIndexBuildItem.getIndex();

    // handle the various @JsonDeserialize cases
    for (AnnotationInstance deserializeInstance :
index.getAnnotations(JSON_DESERIALIZE)) {
        AnnotationTarget annotationTarget =
deserializeInstance.target();
        if (CLASS.equals(annotationTarget.kind())) {
            DotName dotName = annotationTarget.asClass().name();
            Type jandexType = Type.create(dotName,
Type.Kind.CLASS);
            reflectiveHierarchyClass.produce(new
ReflectiveHierarchyBuildItem(jandexType));
        }

    }
```

2.4. Configuration

Configuration in Quarkus is based on SmallRye Config, an implementation of the MicroProfile Config specification. All of the standard features of MP-Config are supported; in addition, there are several extensions which are made available by the SmallRye Config project as well as by Quarkus itself.

The value of these properties is configured in a `application.properties` file that follows the MicroProfile config format.

Configuration of Quarkus extensions is injection-based, using annotations.

2.4.1. Configuration Keys

Leaf configuration keys are mapped to non-`private` fields via the `@io.quarkus.runtime.annotations.ConfigItem` annotation.

> ℹ️ Though the SmallRye Config project is used for implementation, the standard `@ConfigProperty` annotation does not have the same semantics that are needed to support configuration within extensions.

Configuration keys are normally derived from the field names that they are tied to. This is done by de-camel-casing the name and then joining the segments with hyphens (-). Some examples:

- `bindAddress` becomes `bind-address`
- `keepAliveTime` becomes `keep-alive-time`
- `requestDNSTimeout` becomes `request-dns-timeout`

The name can also be explicitly specified by giving a `name` attribute to the `@ConfigItem` annotation.

> **ℹ** Though it is possible to override the configuration key name using the `name` attribute of `@ConfigItem`, normally this should only be done in cases where (for example) the configuration key name is the same as a Java keyword.

2.4.2. Configuration Value types

The type of the field with the `@ConfigItem` annotation determines the conversion that is applied to it. Quarkus extensions may use the full range of configuration types made available by SmallRye Config, which includes:

- All primitive types and primitive wrapper types

- `String`

- Any type which has a constructor accepting a single argument of type `String` or `CharSequence`

- Any type which has a static method named `of` which accepts a single argument of type `String`

- Any type which has a static method named `valueOf` or `parse` which accepts a single argument of type `CharSequence` or `String`

- `java.time.Duration`

- `java.util.regex.Pattern`

- `java.nio.file.Path`

- `io.quarkus.runtime.configuration.MemorySize` to represent data sizes

- `java.net.InetSocketAddress`, `java.net.InetAddress` and `org.wildfly.common.net.CidrAddress`

- A `List` or `Optional` of any of the above types

- `OptionalInt`, `OptionalLong`, `OptionalDouble`

In addition, custom converters may be registered by adding their fully qualified class name in file `META-INF/services/org.eclipse.microprofile.config.spi.Converter`.

Though these implicit converters use reflection, Quarkus will automatically ensure that they are loaded at the appropriate time.

2.4.2.1. Optional Values

If the configuration type is one of the optional types, then empty values are allowed for the configuration key; otherwise, specification of an empty value will result in a configuration error which prevents the application from starting. This is especially relevant to configuration properties of inherently emptiable values such as `List`, `Set`, and `String`. Such value types will never be empty; in the event of an empty value, an empty `Optional` is always used.

2.4.3. Configuration Default Values

A configuration item can be marked to have a default value. The default value is used when no

matching configuration key is specified in the configuration.

Configuration items with a primitive type (such as `int` or `boolean`) implicitly use a default value of `0` or `false`. The sole exception to this rule is the `char` type which does not have an implicit default value.

A property with a default value is not implicitly optional. If a non-optional configuration item with a default value is explicitly specified to have an empty value, the application will report a configuration error and will not start. If it is desired for a property to have a default value and also be optional, it must have an `Optional` type as described above.

2.4.4. Configuration Groups

Configuration values are always collected into grouping classes which are marked with the `@io.quarkus.runtime.annotations.ConfigGroup` annotation. These classes contain a field for each key within its group. In addition, configuration groups can be nested.

2.4.4.1. Optional Configuration Groups

A nested configuration group may be wrapped with an `Optional` type. In this case, the group is not populated unless one or more properties within that group are specified in the configuration. If the group is populated, then any required properties in the group must also be specified otherwise a configuration error will be reported and the application will not start.

2.4.5. Configuration Maps

A `Map` can be used for configuration at any position where a configuration group would be allowed. The key type of such a map **must** be `String`, and its value may be either a configuration group class or a valid leaf type. The configuration key segment following the map's key segment will be used as the key for map values.

2.4.6. Configuration Roots

Configuration roots are configuration groups that appear in the root of the configuration tree. A configuration property's full name is determined by joining the string `quarkus.` with the hyphenated name of the fields that form the path from the root to the leaf field. For example, if I define a configuration root group called `ThreadPool`, with a nested group in a field named `sizing` that in turn contains a field called `minSize`, the final configuration property will be called `quarkus.thread-pool.sizing.min-size`.

A configuration root's name can be given with the `name` property, or it can be inferred from the class name. If the latter, then the configuration key will be the class name, minus any `Config` or `Configuration` suffix, broken up by camel-case, lowercased, and re-joined using hyphens (-).

A configuration root's class name can contain an extra suffix segment for the case where there are configuration roots for multiple Configuration Root Phases. Classes which correspond to the `BUILD_TIME` and `BUILD_AND_RUN_TIME_FIXED` may end with `BuildTimeConfig` or `BuildTimeConfiguration`, and classes which correspond to the `RUN_TIME` phase may end with `RuntimeConfig`, `RunTimeConfig`, `RuntimeConfiguration` or `RunTimeConfiguration`.

Note: The current implementation is still using injection site to determine the root set, so to avoid

migration problems, it is recommended that the injection site (field or parameter) have the same name as the configuration root class until this change is complete.

2.4.6.1. Configuration Root Phases

Configuration roots are strictly bound by configuration phase, and attempting to access a configuration root from outside of its corresponding phase will result in an error. A configuration root dictates when its contained keys are read from configuration, and when they are available to applications. The phases defined by `io.quarkus.runtime.annotations.ConfigPhase` are as follows:

Phase name	Read & avail. at build time	Avail. at run time	Read during static init	Re-read during startup (native executable)	Notes
BUILD_TIME	✓				Appropriate for things which affect build.
BUILD_AND_RUN_TIME_FIXED	✓	✓			Appropriate for things which affect build and must be visible for run time code. Not read from config at run time.
RUN_TIME		✓	✓	✓	Not available at build, read at start in all modes.

For all cases other than the `BUILD_TIME` case, the configuration root class and all of the configuration groups and types contained therein must be located in, or reachable from, the extension's run time artifact. Configuration roots of phase `BUILD_TIME` may be located in or reachable from either of the extension's run time or deployment artifacts.

2.4.7. Configuration Example

```
import io.quarkus.runtime.annotations.ConfigItem;
import io.quarkus.runtime.annotations.ConfigGroup;
import io.quarkus.runtime.annotations.DefaultConverter

import java.io.File;
import java.util.logging.Level;

@ConfigGroup ①
public class FileConfig {

    /**
     * Enable logging to a file.
     */
```

```
    @ConfigItem(defaultValue = "true")
    boolean enable;

    /**
     * The log format.
     */
    @ConfigItem(defaultValue = "%d{yyyy-MM-dd HH:mm:ss,SSS} %h
%N[%i] %-5p [%c{1.}] (%t) %s%e%n")
    String format;

    /**
     * The level of logs to be written into the file.
     */
    @ConfigItem(defaultValue = "ALL")
    Level level;

    /**
     * The name of the file in which logs will be written.
     */
    @ConfigItem(defaultValue = "application.log")
    File path;

}

/**
 * Logging configuration.
 */
@ConfigRoot(phase = ConfigPhase.RUN_TIME) ②
public class LogConfiguration {

    // ...

    /**
     * Configuration properties for the logging file handler.
     */
    FileConfig file;
}

public class LoggingProcessor {
    // ...

    /**
     * Logging configuration.
     */
    ③
    LogConfiguration config;
}
```

A configuration property name can be split into segments. For example, a property name like

`quarkus.log.file.enable` can be split into the following segments:

- `quarkus` - a namespace claimed by Quarkus which is a prefix for all `@ConfigRoot` classes,

- `log` - a name segment which corresponds to the `LogConfiguration` class annotated with `@ConfigRoot`,

- `file` - a name segment which corresponds to the `file` field in this class,

- `enabled` - a name segment which corresponds to `enable` field in `FileConfig` class annotated with `@ConfigGroup`.

① The `FileConfig` class is annotated with `@ConfigGroup` to indicate that this is an aggregate configuration object containing a collection of configurable properties, rather than being a simple configuration key type.

② The `@ConfigRoot` annotation indicates that this object is a configuration root group, in this case one which corresponds to a `log` segment. A class name is used to link configuration root group with the segment from a property name. The `Configuration` part is stripped off from a `LogConfiguration` class name and the remaining `Log` is lowercased to become a `log`. Since all `@ConfigRoot` annotated classes uses `quarkus` as a prefix, this finally becomes `quarkus.log` and represents the properties which names begin with `quarkus.log.*`.

③ Here the `LoggingProcessor` injects a `LogConfiguration` instance automatically by detecting the `@ConfigRoot` annotation.

A corresponding `application.properties` for the above example could be:

```
quarkus.log.file.enable=true
quarkus.log.file.level=DEBUG
quarkus.log.file.path=/tmp/debug.log
```

Since `format` is not defined in these properties, the default value from `@ConfigItem` will be used instead.

2.4.8. Enhanced conversion

You can use enhanced conversion of a config item by using the `@ConvertWith` annotation which accepts a `Converter` class object. If the annotation is present on a config item, the implicit or custom built in converter in use will be overridden by the value provided. To do, see the example below which converts `YES` or `NO` values to `boolean`.

```
@ConfigRoot
public class SomeConfig {
    /**
     * Config item with enhanced converter
     */
    @ConvertWith(YesNoConverter.class) ①
    @ConfigItem(defaultValue = "NO")
    Boolean answer;


    public static class YesNoConverter implements
Converter<Boolean> {

        public YesNoConverter() {}

        @Override
        public Boolean convert(String s) {
            if (s == null || s.isEmpty()) {
                return false;
            }

            switch (s) {
                case "YES":
                    return true;
                case "NO":
                    return false;
            }

            throw new IllegalArgumentException("Unsupported value "
+ s + " given");
        }
    }
}
```

1. Override the default `Boolean` converter and use the provided converter which accepts a `YES` or `NO` config values.

The corresponding `application.properties` will look like.

```
quarkus.some.answer=YES
```

Enum values (config items) are translated to skewed-case (hyphenated) by default. The table below illustrates an enum name and their canonical equivalence:

Java enum	Canonical equivalent
DISCARD	discard
READ_UNCOMMITTED	read-uncommitted
SIGUSR1	sigusr1
JavaEnum	java-enum
MAKING_LifeDifficult	making-life-difficult
YeOldeJBoss	ye-olde-jboss
camelCaseEnum	camel-case-enum

To use the default behaviour which is based on implicit converter or a custom defined one add `@DefaultConverter` annotation to the configuration item

```
@ConfigRoot
public class SomeLogConfig {
    /**
     * The level of logs to be written into the file.
     */
    @DefaultConverter ①
    @ConfigItem(defaultValue = "ALL")
    Level level;
}
```

1. Use the default converter (built in or a custom converter) to convert `Level.class` enum.

2.5. Conditional Step Inclusion

It is possible to only include a given `@BuildStep` under certain conditions. The `@BuildStep` annotation has two optional parameters: `onlyIf` and `onlyIfNot`. These parameters can be set to one or more classes which implement `BooleanSupplier`. The build step will only be included when the method returns `true` (for `onlyIf`) or `false` (for `onlyIfNot`).

The condition class can inject configuration roots as long as they belong to a build-time phase. Run time configuration is not available for condition classes.

The condition class may also inject a value of type `io.quarkus.runtime.LaunchMode`. Constructor parameter and field injection is supported.

An example of a conditional build step

```java
@BuildStep(onlyIf = IsDevMode.class)
LogCategoryBuildItem enableDebugLogging() {
    return new LogCategoryBuildItem("org.your.quarkus.extension",
Level.DEBUG);
}

static class IsDevMode implements BooleanSupplier {
    LaunchMode launchMode;

    public boolean getAsBoolean() {
        return launchMode == LaunchMode.DEVELOPMENT;
    }
}
```

2.6. Bytecode Recording

One of the main outputs of the build process is recorded bytecode. This bytecode actually sets up the runtime environment. For example, in order to start Undertow, the resulting application will have some bytecode that directly registers all Servlet instances and then starts Undertow.

As writing bytecode directly is complex, this is instead done via bytecode recorders. At deployment time, invocations are made on recorder objects that contain the actual runtime logic, but instead of these invocations proceeding as normal they are intercepted and recorded (hence the name). This recording is then used to generate bytecode that performs the same sequence of invocations at runtime. This is essentially a form of deferred execution where invocations made at deployment time get deferred until runtime.

Let's look at the classic 'Hello World' type example. To do this the Quarkus way we would create a recorder as follows:

```java
@Recorder
class HelloRecorder {

  public void sayHello(String name) {
    System.out.println("Hello" + name);
  }

}
```

And then create a build step that uses this recorder:

```
@Record(RUNTIME_INIT)
@BuildStep
public void helloBuildStep(HelloRecorder recorder) {
    recorder.sayHello("World");
}
```

When this build step is run nothing is printed to the console. This is because the `HelloRecorder` that is injected is actually a proxy that records all invocations. Instead if we run the resulting Quarkus program we will see 'Hello World' printed to the console.

Methods on a recorder can return a value, which must be proxiable (if you want to return a non-proxiable item wrap it in `io.quarkus.runtime.RuntimeValue`). These proxies may not be invoked directly, however they can be passed into other recorder methods. This can be any recorder method, including from other `@BuildStep` methods, so a common pattern is to produce `BuildItem` instances that wrap the results of these recorder invocations.

For instance, in order to make arbitrary changes to a Servlet deployment Undertow has a `ServletExtensionBuildItem`, which is a `MultiBuildItem` that wraps a `ServletExtension` instance. I can return a `ServletExtension` from a recorder in another module, and Undertow will consume it and pass it into the recorder method that starts Undertow.

At runtime the bytecode will be invoked in the order it is generated. This means that build step dependencies implicitly control the order that generated bytecode is run. In the example above we know that the bytecode that produces a `ServletExtensionBuildItem` will be run before the bytecode that consumes it.

The following objects can be passed to recorders:

- Primitives

- String

- Class<?> objects

- Objects returned from a previous recorder invocation

- Objects with a no-arg constructor and getter/setters for all properties (or public fields)

- Objects with a constructor annotated with `@RecordableConstructor` with parameter names that match field names

- Any arbitrary object via the `io.quarkus.deployment.recording.RecorderContext#registerSubstitution(Class, Class, Class)` mechanism

- Arrays, Lists and Maps of the above

2.6.1. RecorderContext

`io.quarkus.deployment.recording.RecorderContext` provides some convenience methods to enhance bytecode recording, this includes the ability to register creation functions for classes without no-arg constructors, to register an object substitution (basically a transformer from a non-

serializable object to a serializable one and vice versa), and to create a class proxy. This interface can be directly injected as a method parameter into any `@Record` method.

Calling `classProxy` with a given class name will create a `Class` that can be passed into recorder methods, and at runtime will be substituted with the class whose name was passed in to `classProxy`. This is basically a convenience to avoid the need to explicitly load classes in the recorders.

2.6.2. Printing step execution time

At times, it can be useful to know how the exact time each startup task (which is the result of each bytecode recording) takes when the application is run. The simplest way to determine this information is to set the `quarkus.debug.print-startup-times` property to `true` when running the application. The output will look something like:

```
Build step LoggingResourceProcessor.setupLoggingRuntimeInit
completed in: 42ms
Build step ConfigGenerationBuildStep.checkForBuildTimeConfigChange
completed in: 4ms
Build step SyntheticBeansProcessor.initRuntime completed in: 0ms
Build step ConfigBuildStep.validateConfigProperties completed in:
1ms
Build step ResteasyStandaloneBuildStep.boot completed in: 95ms
Build step VertxHttpProcessor.initializeRouter completed in: 1ms
Build step VertxHttpProcessor.finalizeRouter completed in: 4ms
Build step LifecycleEventsBuildStep.startupEvent completed in: 1ms
Build step VertxHttpProcessor.openSocket completed in: 93ms
Build step ShutdownListenerBuildStep.setupShutdown completed in:
1ms
```

2.7. Contexts and Dependency Injection

2.7.1. Extension Points

As a CDI based runtime, Quarkus extensions often make CDI beans available as part of the extension behavior. However, Quarkus DI solution does not support CDI Portable Extensions. Instead, Quarkus extensions can make use of various Build Time Extension Points.

2.8. Extension Health Check

Health checks are provided via the `quarkus-smallrye-health` extension. It provides both liveness and readiness checks capabilities.

When writing an extension, it's beneficial to provide health checks for the extension, that can be automatically included without the developer needing to write their own.

In order to provide a health check, you should do the following:

- Import the `quarkus-smallrye-health` extension as an **optional** dependency in your runtime module so it will not impact the size of the application if health check is not included.

- Create your health check following the Quarkus - MicroProfile Health guide. We advise providing only readiness check for an extension (liveness check is designed to express the fact that an application is up and needs to be lightweight).

- Import the `quarkus-smallrye-health-spi` library in your deployment module.

- Add a build step in your deployment module that produces a `HealthBuildItem`.

- Add a way to disable the extension health check via a config item `quarkus.<extension>.health.enabled` that should be enabled by default.

Following is an example from the Agroal extension that provides a `DataSourceHealthCheck` to validate the readiness of a datasource.

```
@BuildStep
HealthBuildItem addHealthCheck(AgroalBuildTimeConfig
agroalBuildTimeConfig) {
    return new
HealthBuildItem("io.quarkus.agroal.runtime.health.DataSourceHealthCheck",
            agroalBuildTimeConfig.healthEnabled);
}
```

2.9. Extension Metrics

The `quarkus-smallrye-metrics` extension (and other 3rd party or experimental extensions) provide support for collecting metrics. There are two broad patterns that extensions can use to interact with an optional metrics extension to add their own metrics:

- Consumer pattern: An extension declares a `MetricsFactoryConsumerBuildItem` and uses that to provide a bytecode recorder to the metrics extension. When the metrics extension has initialized, it will iterate over registered consumers to initialize them with a `MetricsFactory`. This factory can be used to declare API-agnostic metrics, which can be a good fit for extensions that provide an instrumentable object for gathering statistics (e.g. Hibernate's `Statistics` class).

- Binder pattern: An extension can opt to use completely different gathering implementations depending on the metrics system. An `Optional<MetricsCapabilityBuildItem> metricsCapability` build step parameter can be used to declare or otherwise initialize API-specific metrics based on the active metrics extension (e.g. "smallrye-metrics" or "micrometer"). This pattern can be combined with the consumer pattern by using `MetricsFactory::metricsSystemSupported()` to test the active metrics extension within the recorder.

Remember that support for metrics is optional. Extensions can use an `Optional<MetricsCapabilityBuildItem> metricsCapability` parameter in their build step to test for the presence of an enabled metrics extension. Consider using additional configuration

to control behavior of metrics. Datasource metrics can be expensive, for example, so additional configuration flags are used enable metrics collection on individual datasources.

When adding metrics for your extension, you may find yourself in one of the following situations:

1. An underlying library used by the extension is using a specific Metrics API directly (either MP Metrics, micrometer, or some other).

2. An underlying library uses its own mechanism for collecting metrics and makes them available at runtime using its own API, e.g. Hibernate's `Statistics` class, or Vert.x `MetricsOptions`.

3. An underlying library does not provide metrics (or there is no library at all) and you want to add instrumentation.

2.9.1. Case 1: The library uses a metrics library directly

If the library directly uses a metrics API, there are two options:

- Use an `Optional<MetricsCapabilityBuildItem> metricsCapability` parameter to test which metrics API is supported (e.g. "smallrye-metrics" or "micrometer") in your build step, and use that to selectively declare or initialize API-specific beans or build items.

- Create a separate build step that consumes a `MetricsFactory`, and use the `MetricsFactory::metricsSystemSupported()` method within the bytecode recorder to initialize required resources if the desired metrics API is supported (e.g. "smallrye-metrics" or "micrometer").

Extensions may need to provide a fallback if there is no active metrics extension or the extension doesn't support the API required by the library.

2.9.2. Case 2: The library provides its own metric API

There are two examples of a library providing its own metrics API:

- The extension defines an instrumentable object as Agroal does with `io.agroal.api.AgroalDataSourceMetrics`, or

- The extension provides its own abstraction of metrics, as Jaeger does with `io.jaegertracing.spi.MetricsFactory`.

2.9.2.1. Observing instrumentable objects

Let's take the instrumentable object (`io.agroal.api.AgroalDataSourceMetrics`) case first. In this case, you can do the following:

- Define a `BuildStep` that produces a `MetricsFactoryConsumerBuildItem` that uses a `RUNTIME_INIT` or `STATIC_INIT` Recorder to define a `MetricsFactory` consumer. For example, the following creates a `MetricsFactoryConsumerBuildItem` if and only if metrics are enabled both for Agroal generally, and for a datasource specifically:

```java
@BuildStep
@Record(ExecutionTime.RUNTIME_INIT)
void registerMetrics(AgroalMetricsRecorder recorder,
        DataSourcesBuildTimeConfig dataSourcesBuildTimeConfig,
        BuildProducer<MetricsFactoryConsumerBuildItem>
datasourceMetrics,
        List<AggregatedDataSourceBuildTimeConfigBuildItem>
aggregatedDataSourceBuildTimeConfigs) {

    for (AggregatedDataSourceBuildTimeConfigBuildItem
aggregatedDataSourceBuildTimeConfig :
aggregatedDataSourceBuildTimeConfigs) {
        // Create a MetricsFactory consumer to register metrics
for a data source
        // IFF metrics are enabled globally and for the data
source
        // (they are enabled for each data source by default if
they are also enabled globally)
        if (dataSourcesBuildTimeConfig.metricsEnabled &&

aggregatedDataSourceBuildTimeConfig.getJdbcConfig().enableMetric
s.orElse(true)) {
            datasourceMetrics.produce(new
MetricsFactoryConsumerBuildItem(

recorder.registerDataSourceMetrics(aggregatedDataSourceBuildTime
Config.getName())));
        }
    }
}
```

- The associated recorder should use the provided `MetricsFactory` to register metrics. For Agroal, this means using the `MetricFactory` API to observe `io.agroal.api.AgroalDataSourceMetrics` methods. For example:

```
/* RUNTIME_INIT */
public Consumer<MetricsFactory> registerDataSourceMetrics(String
dataSourceName) {
    return new Consumer<MetricsFactory>() {
        @Override
        public void accept(MetricsFactory metricsFactory) {
            String tagValue =
DataSourceUtil.isDefault(dataSourceName) ? "default" :
dataSourceName;
            AgroalDataSourceMetrics metrics =
getDataSource(dataSourceName).getMetrics();

            // When using MP Metrics, the builder uses the
VENDOR registry by default.
            metricsFactory.builder("agroal.active.count")
                    .description(
                            "Number of active connections. These
connections are in use and not available to be acquired.")
                    .tag("datasource", tagValue)
                    .buildGauge(metrics::activeCount);
            ....
```

The `MetricsFactory` provides a fluid builder for registration of metrics, with the final step constructing gauges or counters based on a `Supplier` or `ToDoubleFunction`. Timers can either wrap `Callable`, `Runnable`, or `Supplier` implementations, or can use a `TimeRecorder` to accumulate chunks of time. The underlying metrics extension will create appropriate artifacts to observe or measure the defined functions.

2.9.2.2. Using a Metrics API-specific implementation

Using metrics-API specific implementations may be preferred in some cases.nJaeger, for example, defines its own metrics interface, `io.jaegertracing.spi.MetricsFactory`, that it uses to define counters and gauges. A direct mapping from that interface to the metrics system will be the most efficient. In this case, it is important to isolate these specialized implementations and to avoid eager classloading to ensure the metrics API remains an optional, compile-time dependency.

`Optional<MetricsCapabilityBuildItem> metricsCapability` can be used in the build step to selectively control initialization of beans or the production of other build items. The Jaeger extension, for example, can use the following to control initialization of specialized Metrics API adapters:

+

```
/* RUNTIME_INIT */
@BuildStep
@Record(ExecutionTime.RUNTIME_INIT)
void setupTracer(JaegerDeploymentRecorder jdr,
JaegerBuildTimeConfig buildTimeConfig, JaegerConfig jaeger,
        ApplicationConfig appConfig,
Optional<MetricsCapabilityBuildItem> metricsCapability) {

    // Indicates that this extension would like the SSL support to
be enabled
    extensionSslNativeSupport.produce(new
ExtensionSslNativeSupportBuildItem(Feature.JAEGER.getName()));

    if (buildTimeConfig.enabled) {
        boolean metricsEnabled = buildTimeConfig.metricsEnabled &&
metricsCapability.isPresent();
        if ( metricsEnabled &&
metricsCapability.get().metricsSupported(MetricsFactory.MP_METRICS)
) {
            jdr.registerTracerWithMpMetrics(jaeger, appConfig);
        } else if ( metricsEnabled &&
metricsCapability.get().metricsSupported(MetricsFactory.MICROMETER)
) {
            jdr.registerTracerWithMicrometer(jaeger, appConfig);
        } else {
            jdr.registerTracerWithoutMetrics(jaeger, appConfig);
        }
    }
}
```

A recorder consuming a `MetricsFactory` can use `MetricsFactory::metricsSystemSupported()` can be used to control initialization of metrics objects during bytecode recording in a similar way.

2.9.3. Case 3: It is necessary to collect metrics within the extension code

To define your own metrics from scratch, you have two basic options: Use the generic `MetricFactory` builders, or follow the binder pattern, and create instrumentation specific to the enabled metrics extension.

To use the extension-agnostic `MetricFactory` API, your processor can define a `BuildStep` that produces a `MetricsFactoryConsumerBuildItem` that uses a `RUNTIME_INIT` or `STATIC_INIT` Recorder to define a `MetricsFactory` consumer.

+

```
@BuildStep
@Record(ExecutionTime.RUNTIME_INIT)
MetricsFactoryConsumerBuildItem registerMetrics(MyExtensionRecorder
recorder) {
    return new
MetricsFactoryConsumerBuildItem(recorder.registerMetrics());
}
```

+ - The associated recorder should use the provided `MetricsFactory` to register metrics, for example

+

```
final LongAdder extensionCounter = new LongAdder();

/* RUNTIME_INIT */
public Consumer<MetricsFactory> registerMetrics() {
    return new Consumer<MetricsFactory>() {
        @Override
        public void accept(MetricsFactory metricsFactory) {
            metricsFactory.builder("my.extension.counter")
                    .buildGauge(extensionCounter::longValue);
            ....
```

Remember that metrics extensions are optional. Keep metrics-related initialization isolated from other setup for your extension, and structure your code to avoid eager imports of metrics APIs.

Gathering metrics can be expensive. Consider using additional extension-specific configuration to control behavior of metrics if the presence/absence of metrics support isn't sufficient. Datasource metrics can be expensive, for example, so there are additional flags to selectively enable collection of metrics on datasources.

2.10. Customizing JSON handling from an extension

Extensions often need to register serializers and/or deserializers for types the extension provides.

For this, both JSON-B and Jackson extension provide a way to register serializer/deserializer from within an extension deployment module.

Keep in mind that not everybody will need JSON, so you need to make it optional.

If an extension intends to provide JSON related customization, it is strongly advised to provide customization for both JSON-B and Jackson.

2.10.1. Customizing JSON-B

First, add an **optional** dependency to `quarkus-jsonb` on your extension's runtime module.

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jsonb</artifactId>
    <optional>true</optional>
</dependency>
```

Then create a serializer and/or a deserializer for JSON-B, an example of which can be seen in the mongodb-panache extension.

```java
public class ObjectIdSerializer implements
JsonbSerializer<ObjectId> {
    @Override
    public void serialize(ObjectId obj, JsonGenerator generator,
SerializationContext ctx) {
        if (obj != null) {
            generator.write(obj.toString());
        }
    }
}
```

Add a dependency to quarkus-jsonb-spi on your extension's deployment module.

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jsonb-spi</artifactId>
</dependency>
```

Add a build step to your processor to register the serializer via the JsonbSerializerBuildItem.

```java
@BuildStep
JsonbSerializerBuildItem registerJsonbSerializer() {
    return new
JsonbSerializerBuildItem(io.quarkus.mongodb.panache.jsonb.ObjectIdS
erializer.class.getName()));
}
```

The JSON-B extension will then use the produced build item to register your serializer/deserializer automatically.

If you need more customization capabilities than registering a serializer or a deserializer, you can produce a CDI bean that implements `io.quarkus.jsonb.JsonbConfigCustomizer` via an `AdditionalBeanBuildItem`. More info about customizing JSON-B can be found on the JSON guide Configuring JSON support

2.10.2. Customizing Jackson

First, add an **optional** dependency to `quarkus-jackson` on your extension's runtime module.

```xml
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jackson</artifactId>
  <optional>true</optional>
</dependency>
```

Then create a serializer or a deserializer (or both) for Jackson, an example of which can be seen in the `mongodb-panache` extension.

```java
public class ObjectIdSerializer extends StdSerializer<ObjectId> {
    public ObjectIdSerializer() {
        super(ObjectId.class);
    }
    @Override
    public void serialize(ObjectId objectId, JsonGenerator
jsonGenerator, SerializerProvider serializerProvider)
            throws IOException {
        if (objectId != null) {
            jsonGenerator.writeString(objectId.toString());
        }
    }
}
```

Add a dependency to `quarkus-jackson-spi` on your extension's deployment module.

```xml
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jackson-spi</artifactId>
</dependency>
```

Add a build step to your processor to register a Jackson module via the `JacksonModuleBuildItem`. You need to name your module in a unique way across all Jackson modules.

```
@BuildStep
JacksonModuleBuildItem registerJacksonSerDeser() {
    return new JacksonModuleBuildItem.Builder("ObjectIdModule")

.add(io.quarkus.mongodb.panache.jackson.ObjectIdSerializer.class.ge
tName(),

io.quarkus.mongodb.panache.jackson.ObjectIdDeserializer.class.getNa
me(),
                          ObjectId.class.getName())
                .build();
}
```

The Jackson extension will then use the produced build item to register a module within Jackson automatically.

If you need more customization capabilities than registering a module, you can produce a CDI bean that implements `io.quarkus.jackson.ObjectMapperCustomizer` via an `AdditionalBeanBuildItem`. More info about customizing Jackson can be found on the JSON guide Configuring JSON support

2.11. Testing Extensions

Testing of Quarkus extensions should be done with the `io.quarkus.test.QuarkusUnitTest` JUnit 5 extension. This extension allows for Arquillian-style tests that test specific functionalities. It is not intended for testing user applications, as this should be done via `io.quarkus.test.junit.QuarkusTest`. The main difference is that `QuarkusTest` simply boots the application once at the start of the run, while `QuarkusUnitTest` deploys a custom Quarkus application for each test class.

These tests should be placed in the deployment module, if additional Quarkus modules are required for testing their deployment modules should also be added as test scoped dependencies.

Note that `QuarkusUnitTest` is in the `quarkus-junit5-internal` module.

An example test class may look like:

```
package io.quarkus.health.test;

import static org.junit.jupiter.api.Assertions.assertEquals;

import java.util.ArrayList;
import java.util.List;

import javax.enterprise.inject.Instance;
import javax.inject.Inject;

import org.eclipse.microprofile.health.Health;
```

```java
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import io.quarkus.test.QuarkusUnitTest;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.asset.EmptyAsset;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

import io.restassured.RestAssured;

public class FailingUnitTest {

    @RegisterExtension
①
    static final QuarkusUnitTest config = new QuarkusUnitTest()
            .setArchiveProducer(() ->
                    ShrinkWrap.create(JavaArchive.class)
②
                            .addClasses(FailingHealthCheck.class)

.addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml")
            );

    @Inject
③
    @Health
    Instance<HealthCheck> checks;

    @Test
    public void testHealthServlet() {
        RestAssured.when().get("/health").then().statusCode(503);
④
    }

    @Test
    public void testHealthBeans() {
        List<HealthCheck> check = new ArrayList<>();
⑤
        for (HealthCheck i : checks) {
            check.add(i);
        }
        assertEquals(1, check.size());
        assertEquals(HealthCheckResponse.State.DOWN,
check.get(0).call().getState());
    }
}
```

① The `QuarkusUnitTest` extension must be used with a static field. If used with a non-static field,

the test application is not started.

② This producer is used to build the application to be tested. It uses Shrinkwrap to create a JavaArchive to test

③ It is possible to inject beans from our test deployment directly into the test case

④ This method directly invokes the health check Servlet and verifies the response

⑤ This method uses the injected health check bean to verify it is returning the expected result

If you want to test that an extension properly fails at build time, use the `setExpectedException` method:

```
package io.quarkus.hibernate.orm;

import io.quarkus.deployment.configuration.ConfigurationError;
import io.quarkus.test.QuarkusUnitTest;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.RegisterExtension;

public class PersistenceAndQuarkusConfigTest {

    @RegisterExtension
    static QuarkusUnitTest runner = new QuarkusUnitTest()
            .setExpectedException(ConfigurationError.class)
①
            .setArchiveProducer(() ->
ShrinkWrap.create(JavaArchive.class)
                    .addAsManifestResource("META-INF/some-
persistence.xml", "persistence.xml")
                    .addAsResource("application.properties"));

    @Test
    public void testPersistenceAndConfigTest() {
        // should not be called, deployment exception should happen
first:
        // it's illegal to have Hibernate configuration properties
in both the
        // application.properties and in the persistence.xml
        Assertions.fail();
    }

}
```

① This tells JUnit that the Quarkus deployment should fail with a specific exception

2.12. Testing hot reload

It is also possible to write tests that verify an extension works correctly in development mode and can correctly handle updates.

For most extensions this will just work 'out of the box', however it is still a good idea to have a smoke test to verify that this functionality is working as expected. To test this we use QuarkusDevModeTest:

```java
public class ServletChangeTestCase {

    @RegisterExtension
    final static QuarkusDevModeTest test = new QuarkusDevModeTest()
            .setArchiveProducer(new Supplier<JavaArchive>() {
                @Override
                public JavaArchive get() {
                    return ShrinkWrap.create(JavaArchive.class)    ①

                            .addClass(DevServlet.class)
                            .addAsManifestResource(new
StringAsset("Hello Resource"), "resources/file.txt");
                }
            });

    @Test
    public void testServletChange() throws InterruptedException {
        RestAssured.when().get("/dev").then()
                .statusCode(200)
                .body(is("Hello World"));

        test.modifySourceFile("DevServlet.java", new
Function<String, String>() {    ②

            @Override
            public String apply(String s) {
                return s.replace("Hello World", "Hello Quarkus");
            }
        });

        RestAssured.when().get("/dev").then()
                .statusCode(200)
                .body(is("Hello Quarkus"));
    }

    @Test
    public void testAddServlet() throws InterruptedException {
        RestAssured.when().get("/new").then()
                .statusCode(404);
```

```
            test.addSourceFile(NewServlet.class);

        RestAssured.when().get("/new").then()
                .statusCode(200)
                .body(is("A new Servlet"));
    }

    @Test
    public void testResourceChange() throws InterruptedException {
        RestAssured.when().get("/file.txt").then()
                .statusCode(200)
                .body(is("Hello Resource"));

        test.modifyResourceFile("META-INF/resources/file.txt", new
Function<String, String>() {

            @Override
            public String apply(String s) {
                return "A new resource";
            }
        });

        RestAssured.when().get("file.txt").then()
                .statusCode(200)
                .body(is("A new resource"));
    }

    @Test
    public void testAddResource() throws InterruptedException {

        RestAssured.when().get("/new.txt").then()
                .statusCode(404);

        test.addResourceFile("META-INF/resources/new.txt", "New
File");

        RestAssured.when().get("/new.txt").then()
                .statusCode(200)
                .body(is("New File"));

    }
}
```

③ (next to `test.addSourceFile(NewServlet.class);`)

④ (next to `Function<String, String>() {`)

⑤ (next to `File");`)

① This starts the deployment, your test can modify it as part of the test suite. Quarkus will be restarted between each test method so every method starts with a clean deployment.

② This method allows you to modify the source of a class file. The old source is passed into the function, and the updated source is returned.

③ This method adds a new class file to the deployment. The source that is used will be the original

source that is part of the current project.

④ This method modifies a static resource

⑤ This method adds a new static resource

2.13. Native Executable Support

There Quarkus provides a lot of build items that control aspects of the native executable build. This allows for extensions to programmatically perform tasks such as registering classes for reflection or adding static resources to the native executable. Some of these build items are listed below:

`io.quarkus.deployment.builditem.nativeimage.NativeImageResourceBuildItem`

Includes static resources into the native executable.

`io.quarkus.deployment.builditem.nativeimage.NativeImageResourceDirectoryBuildItem`

Includes directory's static resources into the native executable.

`io.quarkus.deployment.builditem.nativeimage.RuntimeReinitializedClassBuildItem`

A class that will be reinitialized at runtime by Substrate. This will result in the static initializer running twice.

`io.quarkus.deployment.builditem.nativeimage.NativeImageSystemPropertyBuildItem`

A system property that will be set at native executable build time.

`io.quarkus.deployment.builditem.nativeimage.NativeImageResourceBundleBuildItem`

Includes a resource bundle in the native executable.

`io.quarkus.deployment.builditem.nativeimage.ReflectiveClassBuildItem`

Registers a class for reflection in Substrate. Constructors are always registered, while methods and fields are optional.

`io.quarkus.deployment.builditem.nativeimage.RuntimeInitializedClassBuildItem`

A class that will be initialized at runtime rather than build time. This will cause the build to fail if the class is initialized as part of the native executable build process, so care must be taken.

`io.quarkus.deployment.builditem.nativeimage.NativeImageConfigBuildItem`

A convenience feature that allows you to control most of the above features from a single build item.

`io.quarkus.deployment.builditem.NativeImageEnableAllCharsetsBuildItem`

Indicates that all charsets should be enabled in native image.

`io.quarkus.deployment.builditem.ExtensionSslNativeSupportBuildItem`

A convenient way to tell Quarkus that the extension requires SSL and it should be enabled during native image build. When using this feature, remember to add your extension to the list of extensions that offer SSL support automatically on the native and ssl guide.

2.14. IDE support tips

2.14.1. Writing Quarkus extensions in Eclipse

The only particular aspect of writing Quarkus extensions in Eclipse is that APT (Annotation Processing Tool) is required as part of extension builds, which means you need to:

- Install `m2e-apt` from https://marketplace.eclipse.org/content/m2e-apt

- Define this property in your `pom.xml`: `<m2e.apt.activation>jdt_apt</m2e.apt.activation>`, although if you rely on `io.quarkus:quarkus-build-parent` you will get it for free.

- If you have the `io.quarkus:quarkus-extension-processor` project open at the same time in your IDE (for example, if you have the Quarkus sources checked out and open in your IDE) you will need to close that project. Otherwise, Eclipse will not invoke the APT plugin that it contains.

- If you just closed the extension processor project, be sure to do `Maven > Update Project` on the other projects in order for Eclipse to pick up the extension processor from the Maven repository.

2.15. Troubleshooting / Debugging Tips

2.15.1. Dump the Generated Classes to the File System

During the augmentation phase Quarkus extensions generate new and modify existing classes for various purposes. Sometimes you need to inspect the generated bytecode to debug or understand an issue. There are three system properties that allow you to dump the classes to the filesystem:

- `quarkus.debug.generated-classes-dir` - to dump the generated classes, such as bean metadata

- `quarkus.debug.transformed-classes-dir` - to dump the transformed classes, e.g. Panache entities

- `quarkus.debug.generated-sources-dir` - to dump the ZIG files; ZIG file is a textual representation of the generated code that is referenced in the stack traces

These properties are especially useful in the development mode or when running the tests where the generated/transformed classes are only held in memory in a class loader.

For example, you can specify the `quarkus.debug.generated-classes-dir` system property to have these classes written out to disk for inspection in the development mode:

```
./mvnw quarkus:dev -Dquarkus.debug.generated-classes-dir=dump
-classes
```

> ℹ️ The property value could be either an absolute path, such as `/home/foo/dump` on a Linux machine, or a path relative to the user working directory, i.e. `dump` corresponds to the `{user.dir}/target/dump` in the dev mode and `{user.dir}/dump` when running the tests.

You should see a line in the log for each class written to the directory:

```
INFO  [io.qua.run.boo.StartupActionImpl] (main) Wrote
/path/to/my/app/target/dump-
classes/io/quarkus/arc/impl/ActivateRequestContextInterceptor_Bean.
class
```

The property is also honored when running tests:

```
./mvnw clean test -Dquarkus.debug.generated-classes-dir=target/dump
-generated-classes
```

Analogously, you can use the `quarkus.debug.transformed-classes-dir` and `quarkus.debug.transformed-classes-dir` properties to dump the relevant output.

2.15.2. Multi-module Maven Projects and the Development Mode

It's not uncommon to develop an extension in a multi-module Maven project that also contains an "example" module. However, if you want to run the example in the development mode then the `-DnoDeps` system property must be used in order to exclude the local project dependencies. Otherwise, Quarkus attempts to monitor the extension classes and this may result in weird class loading issues.

```
./mvnw compile quarkus:dev -DnoDeps
```

2.15.3. Indexer does not include your external dependency

Remember to add `IndexDependencyBuildItem` artifacts to your `@BuildStep`.

2.16. Sample Test Extension

We have an extension that is used to test for regressions in the extension processing. It is located in https://github.com/quarkusio/quarkus/tree/master/core/test-extension directory. In this section we touch on some of the tasks an extension author will typically need to perform using the test-extension code to illustrate how the task could be done.

2.16.1. Features and Capabilities

2.16.1.1. Features

A *feature* represents a functionality provided by an extension. The name of the feature gets displayed in the log during application bootstrap.

Example Startup Lines

```
2019-03-22 14:02:37,884 INFO  [io.quarkus] (main) Quarkus 999-
SNAPSHOT started in 0.061s.
2019-03-22 14:02:37,884 INFO  [io.quarkus] (main) Installed
features: [cdi, test-extension] ①
```

① A list of features installed in the runtime image

A feature can be registered in a Build Step Processors method that produces a `FeatureBuildItem`:

TestProcessor#feature()

```
    @BuildStep
    FeatureBuildItem feature() {
        return new FeatureBuildItem("test-extension");
    }
```

The name of the feature should only contain lowercase characters, words are separated by dash; e.g. `security-jpa`. An extension should provide at most one feature and the name must be unique. If multiple extensions register a feature of the same name the build fails.

The feature name should also map to a label in the extension's `devtools/common/src/main/filtered/extensions.json` entry so that the feature name displayed by the startup line matches a label that one can used to select the extension when creating a project using the Quarkus maven plugin as shown in this example taken from the Writing JSON REST Services guide where the `resteasy-jsonb` feature is referenced:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=rest-json \
    -DclassName="org.acme.rest.json.FruitResource" \
    -Dpath="/fruits" \
    -Dextensions="resteasy-jsonb"
cd rest-json
```

2.16.1.2. Capabilities

A *capability* represents a technical capability that can be queried by other extensions. An extension may provide multiple capabilities and multiple extensions can provide the same capability. By default, capabilities are not displayed to users.

Capabilities can be registered in a Build Step Processors method that produces a `CapabilityBuildItem`:

TestProcessor#capability()

```
    @BuildStep
    void capabilities(BuildProducer<CapabilityBuildItem>
capabilityProducer) {
        capabilityProducer.produce(new
CapabilityBuildItem("org.acme.test-transactions"));
        capabilityProducer.produce(new
CapabilityBuildItem("org.acme.test-metrics"));
    }
```

Extensions can consume registered capabilities using the `Capabilities` build item:

TestProcessor#doSomeCoolStuff()

```
    @BuildStep
    void doSomeCoolStuff(Capabilities capabilities) {
        if (capabilities.isPresent(Capability.TRANSACTIONS)) {
          // do something only if JTA transactions are in...
        }
    }
```

Capabilities should follow the naming conventions of Java packages; e.g. `io.quarkus.security.jpa`. Capabilities provided by core extensions should be listed in the `io.quarkus.deployment.Capability` enum and their name should always start with the `io.quarkus` prefix.

2.16.2. Bean Defining Annotations

The CDI layer processes CDI beans that are either explicitly registered or that it discovers based on bean defining annotations as defined in 2.5.1. Bean defining annotations. You can expand this set of annotations to include annotations your extension processes using a `BeanDefiningAnnotationBuildItem` as shown in this `TestProcessor#registerBeanDefinningAnnotations` example:

```
import javax.enterprise.context.ApplicationScoped;
import org.jboss.jandex.DotName;
import io.quarkus.extest.runtime.TestAnnotation;

public final class TestProcessor {
    static DotName TEST_ANNOTATION =
DotName.createSimple(TestAnnotation.class.getName());
    static DotName TEST_ANNOTATION_SCOPE =
DotName.createSimple(ApplicationScoped.class.getName());

...

    @BuildStep
    BeanDefiningAnnotationBuildItem registerX() {
        ①
        return new BeanDefiningAnnotationBuildItem(TEST_ANNOTATION,
TEST_ANNOTATION_SCOPE);
    }
...
}

/**
 * Marker annotation for test configuration target beans
 */
@Target({ TYPE })
@Retention(RUNTIME)
@Documented
@Inherited
public @interface TestAnnotation {
}

/**
 * A sample bean
 */
@TestAnnotation ②
public class ConfiguredBean implements IConfigConsumer {

...
```

① Register the annotation class and CDI default scope using the Jandex `DotName` class.

② `ConfiguredBean` will be processed by the CDI layer the same as a bean annotated with the CDI standard @ApplicationScoped.

2.16.3. Parsing Config to Objects

One of the main things an extension is likely to do is completely separate the configuration phase of behavior from the runtime phase. Frameworks often do parsing/load of configuration on startup that

can be done during build time to both reduce the runtime dependencies on frameworks like xml parsers as well as reducing the startup time the parsing incurs.

An example of parsing a XML config file using JAXB is shown in the `TestProcessor#parseServiceXmlConfig` method: .Parsing an XML Configuration into Runtime XmlConfig Instance

```
    @BuildStep
    @Record(STATIC_INIT)
    RuntimeServiceBuildItem parseServiceXmlConfig(TestRecorder
 recorder) throws JAXBException {
        RuntimeServiceBuildItem serviceBuildItem = null;
        JAXBContext context =
JAXBContext.newInstance(XmlConfig.class);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        InputStream is =
getClass().getResourceAsStream("/config.xml"); ①
        if (is != null) {
            log.info("Have XmlConfig, loading");
            XmlConfig config = (XmlConfig)
unmarshaller.unmarshal(is); ②
...
        }
        return serviceBuildItem;
    }
```

① Look for a config.xml classpath resource

② If found, parse using JAXB context for `XmlConfig.class`

> ℹ️ If there was no /config.xml resource available in the build environment, then a null `RuntimeServiceBuildItem` would be returned and no subsequent logic based on a `RuntimeServiceBuildItem` being produced would execute.

Typically one is loading a configuration to create some runtime component/service as `parseServiceXmlConfig` is doing. We will come back to the rest of the behavior in `parseServiceXmlConfig` in the following Manage Non-CDI Service section.

If for some reason you need to parse the config and use it in other build steps in an extension processor, you would need to create an `XmlConfigBuildItem` to pass the parsed XmlConfig instance around.

> 💡 If you look at the XmlConfig code you will see that it does carry around the JAXB annotations. If you don't want these in the runtime image, you could clone the XmlConfig instance into some POJO object graph and then replace XmlConfig with the POJO class. We will do this in Replacing Classes in the Native Image.

2.16.4. Scanning Deployments Using Jandex

If your extension defines annotations or interfaces that mark beans needing to be processed, you can locate these beans using the Jandex API, a Java annotation indexer and offline reflection library. The following `TestProcessor#scanForBeans` method shows how to find the beans annotated with our `@TestAnnotation` that also implement the `IConfigConsumer` interface:

Example Jandex Usage

```
    static DotName TEST_ANNOTATION =
DotName.createSimple(TestAnnotation.class.getName());
...

    @BuildStep
    @Record(STATIC_INIT)
    void scanForBeans(TestRecorder recorder,
BeanArchiveIndexBuildItem beanArchiveIndex, ①
            BuildProducer<TestBeanBuildItem> testBeanProducer) {
        IndexView indexView = beanArchiveIndex.getIndex(); ②
        Collection<AnnotationInstance> testBeans =
indexView.getAnnotations(TEST_ANNOTATION); ③
        for (AnnotationInstance ann : testBeans) {
            ClassInfo beanClassInfo = ann.target().asClass();
            try {
                boolean isConfigConsumer =
beanClassInfo.interfaceNames()
                        .stream()
                        .anyMatch(dotName ->
dotName.equals(DotName.createSimple(IConfigConsumer.class.getName()
))); ④
                if (isConfigConsumer) {
                    Class<IConfigConsumer> beanClass =
(Class<IConfigConsumer>)
Class.forName(beanClassInfo.name().toString(), false,
Thread.currentThread().getContextClassLoader());
                    testBeanProducer.produce(new
TestBeanBuildItem(beanClass)); ⑤
                    log.infof("Configured bean: %s", beanClass);
                }
            } catch (ClassNotFoundException e) {
                log.warn("Failed to load bean class", e);
            }
        }
    }
```

① Depend on a `BeanArchiveIndexBuildItem` to have the build step be run after the deployment has been indexed.

② Retrieve the index.

③ Find all beans annotated with @TestAnnotation.

④ Determine which of these beans also has the IConfigConsumer interface.

⑤ Save the bean class in a TestBeanBuildItem for use in a latter RUNTIME_INIT build step that will interact with the bean instances.

2.16.5. Interacting With Extension Beans

You can use the io.quarkus.arc.runtime.BeanContainer interface to interact with your extension beans. The following configureBeans methods illustrate interacting with the beans scanned for in the previous section:

Using CDI BeanContainer Interface

```
// TestProcessor#configureBeans
    @BuildStep
    @Record(RUNTIME_INIT)
    void configureBeans(TestRecorder recorder,
 List<TestBeanBuildItem> testBeans, ①
            BeanContainerBuildItem beanContainer, ②
            TestRunTimeConfig runTimeConfig) {

        for (TestBeanBuildItem testBeanBuildItem : testBeans) {
            Class<IConfigConsumer> beanClass =
testBeanBuildItem.getConfigConsumer();
            recorder.configureBeans(beanContainer.getValue(),
beanClass, buildAndRunTimeConfig, runTimeConfig); ③
        }
    }

// TestRecorder#configureBeans
    public void configureBeans(BeanContainer beanContainer,
Class<IConfigConsumer> beanClass,
            TestBuildAndRunTimeConfig buildTimeConfig,
            TestRunTimeConfig runTimeConfig) {
        log.info("Begin BeanContainerListener callback\n");
        IConfigConsumer instance =
beanContainer.instance(beanClass); ④
        instance.loadConfig(buildTimeConfig, runTimeConfig); ⑤
        log.infof("configureBeans, instance=%s\n", instance);
    }
```

① Consume the `TestBeanBuildItem`s produced from the scanning build step.

② Consume the BeanContainerBuildItem to order this build step to run after the CDI bean container has been created.

③ Call the runtime recorder to record the bean interactions.

④ Runtime recorder retrieves the bean using its type.

⑤ Runtime recorder invokes the `IConfigConsumer#loadConfig(…)` method passing in the configuration objects with runtime information.

2.16.6. Manage Non-CDI Service

A common purpose for an extension is to integrate a non-CDI aware service into the CDI based Quarkus runtime. Step 1 of this task is to load any configuration needed in a STATIC_INIT build step as we did in Parsing Config to Objects. Now we need to create an instance of the service using the configuration. Let's return to the `TestProcessor#parseServiceXmlConfig` method to see how this can be done.

Creating a Non-CDI Service

```
// TestProcessor#parseServiceXmlConfig
    @BuildStep
    @Record(STATIC_INIT)
    RuntimeServiceBuildItem parseServiceXmlConfig(TestRecorder
recorder) throws JAXBException {
        RuntimeServiceBuildItem serviceBuildItem = null;
        JAXBContext context =
JAXBContext.newInstance(XmlConfig.class);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        InputStream is =
getClass().getResourceAsStream("/config.xml");
        if (is != null) {
            log.info("Have XmlConfig, loading");
            XmlConfig config = (XmlConfig)
unmarshaller.unmarshal(is);
            log.info("Loaded XmlConfig, creating service");
            RuntimeValue<RuntimeXmlConfigService> service =
recorder.initRuntimeService(config); ①
            serviceBuildItem = new
RuntimeServiceBuildItem(service); ③
        }
        return serviceBuildItem;
    }

// TestRecorder#initRuntimeService
    public RuntimeValue<RuntimeXmlConfigService>
initRuntimeService(XmlConfig config) {
        RuntimeXmlConfigService service = new
RuntimeXmlConfigService(config); ②
        return new RuntimeValue<>(service);
    }

// RuntimeServiceBuildItem
    final public class RuntimeServiceBuildItem extends
SimpleBuildItem {
    private RuntimeValue<RuntimeXmlConfigService> service;

    public
RuntimeServiceBuildItem(RuntimeValue<RuntimeXmlConfigService>
service) {
        this.service = service;
    }

    public RuntimeValue<RuntimeXmlConfigService> getService() {
        return service;
    }
}
```

① Call into the runtime recorder to record the creation of the service.

② Using the parsed `XmlConfig` instance, create an instance of `RuntimeXmlConfigService` and wrap it in a `RuntimeValue`. Use a `RuntimeValue` wrapper for non-interface objects that are non-proxiable.

③ Wrap the return service value in a `RuntimeServiceBuildItem` for use in a RUNTIME_INIT build step that will start the service.

2.16.6.1. Starting a Service

Now that you have recorded the creation of a service during the build phase, you need to record how to start the service at runtime during booting. You do this with a RUNTIME_INIT build step as shown in the `TestProcessor#startRuntimeService` method.

Starting/Stopping a Non-CDI Service

```
// TestProcessor#startRuntimeService
    @BuildStep
    @Record(RUNTIME_INIT)
    ServiceStartBuildItem startRuntimeService(TestRecorder
recorder, ShutdownContextBuildItem shutdownContextBuildItem , ①
            RuntimeServiceBuildItem serviceBuildItem) throws
IOException { ②
        if (serviceBuildItem != null) {
            log.info("Registering service start");
            recorder.startRuntimeService(shutdownContextBuildItem,
serviceBuildItem.getService()); ③
        } else {
            log.info("No RuntimeServiceBuildItem seen, check
config.xml");
        }
        return new
ServiceStartBuildItem("RuntimeXmlConfigService"); ④
    }

// TestRecorder#startRuntimeService
    public void startRuntimeService(ShutdownContext
shutdownContext, RuntimeValue<RuntimeXmlConfigService>
runtimeValue)
            throws IOException {
        RuntimeXmlConfigService service = runtimeValue.getValue();
        service.startService(); ⑤
        shutdownContext.addShutdownTask(service::stopService); ⑥
    }
```

① We consume a ShutdownContextBuildItem to register the service shutdown.

② We consume the previously initialized service captured in `RuntimeServiceBuildItem`.

③ Call the runtime recorder to record the service start invocation.

④ Produce a `ServiceStartBuildItem` to indicate the startup of a service. See Startup and Shutdown Events for details.

⑤ Runtime recorder retrieves the service instance reference and calls its `startService` method.

⑥ Runtime recorder registers an invocation of the service instance `stopService` method with the Quarkus `ShutdownContext`.

The code for the `RuntimeXmlConfigService` can be viewed here: RuntimeXmlConfigService.java

The testcase for validating that the `RuntimeXmlConfigService` has started can be found in the `testRuntimeXmlConfigService` test of `ConfiguredBeanTest` and `NativeImageIT`.

2.16.7. Startup and Shutdown Events

The Quarkus container supports startup and shutdown lifecycle events to notify components of the container startup and shutdown. There are CDI events fired that components can observe are illustrated in this example:

Observing Container Startup

```
import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;

puclic class SomeBean {
    /**
     * Called when the runtime has started
     * @param event
     */
    void onStart(@Observes StartupEvent event) { ①
        System.out.printf("onStart, event=%s%n", event);
    }

    /**
     * Called when the runtime is shutting down
     * @param event
     */
    void onStop(@Observes ShutdownEvent event) { ②
        System.out.printf("onStop, event=%s%n", event);
    }
}
```

① Observe a `StartupEvent` to be notified the runtime has started.

② Observe a 'ShutdownEvent` to be notified when the runtime is going to shutdown.

What is the relevance of startup and shutdown events for extension authors? We have already seen the use of a `ShutdownContext` to register a callback to perform shutdown tasks in the Starting a Service section. These shutdown tasks would be called after a `ShutdownEvent` had been sent.

A   `StartupEvent`   is   fired   after   all

`io.quarkus.deployment.builditem.ServiceStartBuildItem` producers have been consumed. The implication of this is that if an extension has services that application components would expect to have been started when they observe a `StartupEvent`, the build steps that invoke the runtime code to start those services needs to produce a `ServiceStartBuildItem` to ensure that the runtime code is run before the `StartupEvent` is sent. Recall that we saw the production of a `ServiceStartBuildItem` in the previous section, and it is repeated here for clarity:

Example of Producing a ServiceStartBuildItem

```
// TestProcessor#startRuntimeService
    @BuildStep
    @Record(RUNTIME_INIT)
    ServiceStartBuildItem startRuntimeService(TestRecorder
recorder, ShutdownContextBuildItem shutdownContextBuildItem,
            RuntimeServiceBuildItem serviceBuildItem) throws
IOException {
...
        return new
ServiceStartBuildItem("RuntimeXmlConfigService"); ①
    }
```

① Produce a `ServiceStartBuildItem` to indicate that this is a service starting step that needs to run before the `StartupEvent` is sent.

2.16.8. Register Resources for Use in Native Image

Not all configuration or resources can be consumed at build time. If you have classpath resources that the runtime needs to access, you need to inform the build phase that these resources need to be copied into the native image. This is done by producing one or more `NativeImageResourceBuildItem` or `NativeImageResourceBundleBuildItem` in the case of resource bundles. Examples of this are shown in this sample `registerNativeImageResources` build step:

```
public final class MyExtProcessor {
    @Inject
    BuildProducer<NativeImageResourceBuildItem> resource;
    @Inject
    BuildProducer<NativeImageResourceBundleBuildItem>
resourceBundle;

    @BuildStep
    void registerNativeImageResources() {
        resource.produce(new
NativeImageResourceBuildItem("/security/runtime.keys")); ①

        resource.produce(new NativeImageResourceBuildItem(
                "META-INF/my-descriptor.xml")); ②

        resourceBundle.produce(new
NativeImageResourceBuildItem("javax.xml.bind.Messages")); ③
    }
}
```

① Indicate that the /security/runtime.keys classpath resource should be copied into native image.

② Indicate that the `META-INF/my-descriptor.xml` resource should be copied into native image

③ Indicate that the "javax.xml.bind.Messages" resource bundle should be copied into native image.

2.16.9. Service files

If you are using `META-INF/services` files you need to register the files as resources so that your native image can find them, but you also need to register each listed class for reflection so they can be instantiated or inspected at run-time:

```
public final class MyExtProcessor {

    @BuildStep
    void
registerNativeImageResources(BuildProducer<ServiceProviderBuildItem
> services) {
        String service = "META-INF/services/" +
io.quarkus.SomeService.class.getName();

        // find out all the implementation classes listed in the
service files
        Set<String> implementations =

ServiceUtil.classNamesNamedIn(Thread.currentThread().getContextClas
sLoader(),

                                      service);

        // register every listed implementation class so they can
be instantiated
        // in native-image at run-time
        services.produce(
            new
ServiceProviderBuildItem(io.quarkus.SomeService.class.getName(),

implementations.toArray(new String[0])));
    }
}
```

⚠️ ServiceProviderBuildItem takes a list of service implementation classes as parameters: if you are not reading them from the service file, make sure that they correspond to the service file contents because the service file will still be read and used at run-time. This is not a substitute for writing a service file.

ℹ️ This only registers the implementation classes for instantiation via reflection (you will not be able to inspect its fields and methods). If you need to do that, you can do it this way:

```
public final class MyExtProcessor {

    @BuildStep
    void
registerNativeImageResources(BuildProducer<NativeImageResourceBuild
Item> resource,

BuildProducer<ReflectiveClassBuildItem> reflectionClasses) {
        String service = "META-INF/services/" +
io.quarkus.SomeService.class.getName();

        // register the service file so it is visible in native-
image
        resource.produce(new
NativeImageResourceBuildItem(service));

        // register every listed implementation class so they can
be inspected/instantiated
        // in native-image at run-time
        Set<String> implementations =

ServiceUtil.classNamesNamedIn(Thread.currentThread().getContextClas
sLoader(),

                                        service);
        reflectionClasses.produce(
            new ReflectiveClassBuildItem(true, true,
implementations.toArray(new String[0])));
    }
}
```

While this is the easiest way to get your services running natively, it's less efficient than scanning the implementation classes at build time and generating code that registers them at static-init time instead of relying on reflection.

You can achieve that by adapting the previous build step to use a static-init recorder instead of registering classes for reflection:

```
public final class MyExtProcessor {

    @BuildStep
    @Record(ExecutionTime.STATIC_INIT)
    void registerNativeImageResources(RecorderContext
recorderContext,

                                        SomeServiceRecorder recorder)
{
        String service = "META-INF/services/" +
io.quarkus.SomeService.class.getName();
```

```java
        // read the implementation classes
        Collection<Class<? extends io.quarkus.SomeService>>
implementationClasses = new LinkedHashSet<>();
        Set<String> implementations =
ServiceUtil.classNamesNamedIn(Thread.currentThread().getContextClas
sLoader(),

service);
        for(String implementation : implementations) {
            implementationClasses.add((Class<? extends
io.quarkus.SomeService>)
                recorderContext.classProxy(implementation));
        }

        // produce a static-initializer with those classes
        recorder.configure(implementationClasses);
    }
}

@Recorder
public class SomeServiceRecorder {

    public void configure(List<Class<? extends
io.quarkus.SomeService>> implementations) {
        // configure our service statically
        SomeServiceProvider serviceProvider =
SomeServiceProvider.instance();
        SomeServiceBuilder builder =
serviceProvider.getSomeServiceBuilder();

        List<io.quarkus.SomeService> services = new
ArrayList<>(implementations.size());
        // instantiate the service implementations
        for (Class<? extends io.quarkus.SomeService>
implementationClass : implementations) {
            try {

services.add(implementationClass.getConstructor().newInstance());
            } catch (Exception e) {
                throw new IllegalArgumentException("Unable to
instantiate service " + implementationClass, e);
            }
        }

        // build our service
        builder.withSomeServices(implementations.toArray(new
io.quarkus.SomeService[0]));
        ServiceManager serviceManager = builder.build();
```

```
        // register it
        serviceProvider.registerServiceManager(serviceManager,
Thread.currentThread().getContextClassLoader());
    }
}
```

2.16.10. Object Substitution

Objects created during the build phase that are passed into the runtime need to have a default constructor in order for them to be created and configured at startup of the runtime from the build time state. If an object does not have a default constructor you will see an error similar to the following during generation of the augmented artifacts:

DSAPublicKey Serialization Error

```
    [error]: Build step
io.quarkus.deployment.steps.MainClassBuildStep#build threw an
exception: java.lang.RuntimeException: Unable to serialize objects
of type class sun.security.provider.DSAPublicKeyImpl to bytecode as
it has no default constructor
    at io.quarkus.builder.Execution.run(Execution.java:123)
    at
io.quarkus.builder.BuildExecutionBuilder.execute(BuildExecutionBuil
der.java:136)
    at
io.quarkus.deployment.QuarkusAugmentor.run(QuarkusAugmentor.java:11
0)
    at io.quarkus.runner.RuntimeRunner.run(RuntimeRunner.java:99)
    ... 36 more
```

There is a `io.quarkus.runtime.ObjectSubstitution` interface that can be implemented to tell Quarkus how to handle such classes. An example implementation for the `DSAPublicKey` is shown here:

DSAPublicKeyObjectSubstitution Example

```
package io.quarkus.extest.runtime.subst;

import java.security.KeyFactory;
import java.security.NoSuchAlgorithmException;
import java.security.interfaces.DSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.X509EncodedKeySpec;
import java.util.logging.Logger;

import io.quarkus.runtime.ObjectSubstitution;

public class DSAPublicKeyObjectSubstitution implements
ObjectSubstitution<DSAPublicKey, KeyProxy> {
    private static final Logger log =
Logger.getLogger("DSAPublicKeyObjectSubstitution");
    @Override
    public KeyProxy serialize(DSAPublicKey obj) { ①
        log.info("DSAPublicKeyObjectSubstitution.serialize");
        byte[] encoded = obj.getEncoded();
        KeyProxy proxy = new KeyProxy();
        proxy.setContent(encoded);
        return proxy;
    }

    @Override
    public DSAPublicKey deserialize(KeyProxy obj) { ②
        log.info("DSAPublicKeyObjectSubstitution.deserialize");
        byte[] encoded = obj.getContent();
        X509EncodedKeySpec publicKeySpec = new
X509EncodedKeySpec(encoded);
        DSAPublicKey dsaPublicKey = null;
        try {
            KeyFactory kf = KeyFactory.getInstance("DSA");
            dsaPublicKey = (DSAPublicKey)
kf.generatePublic(publicKeySpec);

        } catch (NoSuchAlgorithmException | InvalidKeySpecException
e) {
            e.printStackTrace();
        }
        return dsaPublicKey;
    }
}
```

① The serialize method takes the object without a default constructor and creates a `KeyProxy` that contains the information necessary to recreate the `DSAPublicKey`.

② The deserialize method uses the `KeyProxy` to recreate the `DSAPublicKey` from its encoded form using the key factory.

An extension registers this substitution by producing an `ObjectSubstitutionBuildItem` as shown in this `TestProcessor#loadDSAPublicKey` fragment:

Registering an Object Subtitution

```
    @BuildStep
    @Record(STATIC_INIT)
    PublicKeyBuildItem loadDSAPublicKey(TestRecorder recorder,
            BuildProducer<ObjectSubstitutionBuildItem>
substitutions) throws IOException, GeneralSecurityException {
...
        // Register how to serialize DSAPublicKey
        ObjectSubstitutionBuildItem.Holder<DSAPublicKey, KeyProxy>
holder = new ObjectSubstitutionBuildItem.Holder(
                DSAPublicKey.class, KeyProxy.class,
DSAPublicKeyObjectSubstitution.class);
        ObjectSubstitutionBuildItem keysub = new
ObjectSubstitutionBuildItem(holder);
        substitutions.produce(keysub);

        log.info("loadDSAPublicKey run");
        return new PublicKeyBuildItem(publicKey);
    }
```

2.16.11. Replacing Classes in the Native Image

The Graal SDK supports substitutions of classes in the native image. An example of how one could replace the `XmlConfig/XmlData` classes with versions that have no JAXB annotation dependencies is shown in these example classes:

Substitution of XmlConfig/XmlData Classes Example

```
package io.quarkus.extest.runtime.graal;
import java.util.Date;
import com.oracle.svm.core.annotate.Substitute;
import com.oracle.svm.core.annotate.TargetClass;
import io.quarkus.extest.runtime.config.XmlData;

@TargetClass(XmlConfig.class)
@Substitute
public final class Target_XmlConfig {

    @Substitute
    private String address;
    @Substitute
    private int port;
```

```java
    @Substitute
    private ArrayList<XData> dataList;

    @Substitute
    public String getAddress() {
        return address;
    }

    @Substitute
    public int getPort() {
        return port;
    }

    @Substitute
    public ArrayList<XData> getDataList() {
        return dataList;
    }

    @Substitute
    @Override
    public String toString() {
        return "Target_XmlConfig{" +
                "address='" + address + '\'' +
                ", port=" + port +
                ", dataList=" + dataList +
                '}';
    }
}

@TargetClass(XmlData.class)
@Substitute
public final class Target_XmlData {

    @Substitute
    private String name;
    @Substitute
    private String model;
    @Substitute
    private Date date;

    @Substitute
    public String getName() {
        return name;
    }

    @Substitute
    public String getModel() {
        return model;
    }
```

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

    @Substitute
    @Override
    public String toString() {
        return "Target_XmlData{" +
                "name='" + name + '\'' +
                ", model='" + model + '\'' +
                ", date='" + date + '\'' +
                '}';
    }
}
```

3. Configuration reference documentation

The configuration is an important part of each extension and therefore needs to be properly documented. Each configuration property should have a proper Javadoc comment.

While it is handy to have the documentation available when coding, this configuration documentation must also be available in the extension guides. The Quarkus build automatically generates the configuration documentation for you based on the Javadoc comments but you need to explicitly include it in your guide.

In this section, we will explain everything you need to know about the configuration reference documentation.

3.1. Writing the documentation

For each configuration property, you need to write some Javadoc explaining its purpose.

> Always make the first sentence meaningful and self-contained as it is included in the summary table.

You can either use standard Javadoc comments or Asciidoc directly as a Javadoc comment.

We assume you are familiar with writing Javadoc comments so let's focus on our Asciidoc support. While standard Javadoc comments are perfectly fine for simple documentation (recommended even), if you want to include tips, source code extracts, lists... Asciidoc comes in handy.

Here is a typical configuration property commented with Asciidoc:

```
/**
 * Class name of the Hibernate ORM dialect. The complete list of
bundled dialects is available in the
 *
https://docs.jboss.org/hibernate/stable/orm/javadocs/org/hibernate/
dialect/package-summary.html[Hibernate ORM JavaDoc].
 *
 * [NOTE]
 * ====
 * Not all the dialects are supported in GraalVM native
executables: we currently provide driver extensions for PostgreSQL,
 * MariaDB, Microsoft SQL Server and H2.
 * ====
 *
 * @asciidoclet
 */
@ConfigItem
public Optional<String> dialect;
```

This is the simple case: you just have to write Asciidoc and mark the comment with the `@asciidoclet` tag. This tag has two purposes: it is used as a marker for our generation tool but it is also used by the `javadoc` process for proper Javadoc generation.

Now let's consider a more complicated example:

```
// @formatter:off
/**
 * Name of the file containing the SQL statements to execute when
Hibernate ORM starts.
 * Its default value differs depending on the Quarkus launch mode:
 *
 * * In dev and test modes, it defaults to `import.sql`.
 *    Simply add an `import.sql` file in the root of your resources
directory
 *    and it will be picked up without having to set this property.
 *    Pass `no-file` to force Hibernate ORM to ignore the SQL import
file.
 * * In production mode, it defaults to `no-file`.
 *    It means Hibernate ORM won't try to execute any SQL import
file by default.
 *    Pass an explicit value to force Hibernate ORM to execute the
SQL import file.
 *
 * If you need different SQL statements between dev mode, test
(`@QuarkusTest`) and in production, use Quarkus
 * https://quarkus.io/guides/config#configuration-
profiles[configuration profiles facility].
 *
 * [source,property]
 * .application.properties
 * ----
 * %dev.quarkus.hibernate-orm.sql-load-script = import-dev.sql
 * %test.quarkus.hibernate-orm.sql-load-script = import-test.sql
 * %prod.quarkus.hibernate-orm.sql-load-script = no-file
 * ----
 *
 * [NOTE]
 * ====
 * Quarkus supports `.sql` file with SQL statements or comments
spread over multiple lines.
 * Each SQL statement must be terminated by a semicolon.
 * ====
 *
 * @asciidoclet
 */
// @formatter:on
@ConfigItem
public Optional<String> sqlLoadScript;
```

A few comments on this one:

- Every time you will need the indentation to be respected in the Javadoc comment (think list items

spread on multiple lines or indented source code), you will need to disable temporarily the automatic Eclipse formatter (this, even if you don't use Eclipse as the formatter is included in our build). To do so, use the `// @formatter:off`/`// @formatter:on` markers. Note the fact that they are separate comments and there is a space after the `//` marker. This is required.

- As you can see, you can use the full power of Asciidoctor (except for the limitation below)

> ⚠️ You cannot use open blocks (`--`) in your Asciidoctor documentation. All the other types of blocks (source, admonitions…) are supported.

> 💡 By default, the doc generator will use the hyphenated field name as the key of a `java.util.Map` configuration item. To override this default and have a user friendly key (independent of implementation details), you may use the `io.quarkus.runtime.annotations.ConfigDocMapKey` annotation. See the following example,
>
> ```
> @ConfigRoot
> public class SomeConfig {
>     /**
>      * Namespace configuration.
>      */
>     @ConfigItem(name = ConfigItem.PARENT)
>     @ConfigDocMapKey("cache-name") ①
>     Map<String, CaffeineNamespaceConfig> namespace;
> }
> ```

1. This will generate a configuration map key named `quarkus.some."cache-name"` instead of `quarkus.some."namespace"`.

3.2. Writing section documentation

If you wish to generate configuration section of a given `@ConfigGroup`, Quarkus has got you covered with the `@ConfigDocSection` annotation. See the code example below:

```
/**
* Config group related configuration.
* Amazing introduction here
*/
@ConfigItem
@ConfigDocSection ①
public ConfigGroupConfig configGroup;
```

1. This will add a section documentation for the `configGroup` config item in the generated documentation. Section's title and introduction will be derived from the javadoc of the configuration item. The first sentence from the javadoc is considered as the section title and the remaining sentences used as section introduction. You can also use the `@asciidoclet` tag as

shown above.

3.3. Generating the documentation

Generating the documentation is easy:

- Running `./mvnw clean install -DskipTests -DskipITs` will do.
- You can either do it globally or in a specific extension directory (e.g. `extensions/mailer`).

The documentation is generated in the global `target/asciidoc/generated/config/` located at the root of the project.

3.4. Including the documentation in the extension guide

Now that you have generated the configuration reference documentation for your extension, you need to include it in your guide (and review it).

This is simple, include the generated documentation in your guide:

```
include::{generated-dir}/config/quarkus-your-
extension.adoc[opts=optional, leveloffset=+1]
```

If you are interested in including the generated documentation for the config group, you can use the include statement below

```
include::{generated-dir}/config/hyphenated-config-group-class-name-
with-runtime-or-deployment-namespace-replaced-by-config-group-
namespace.adoc[opts=optional, leveloffset=+1]
```

For example, the `io.quarkus.vertx.http.runtime.FormAuthConfig` configuration group will be generated in a file named `quarkus-vertx-http-config-group-form-auth-config.adoc`.

A few recommendations:

- `opts=optional` is mandatory as we don't want the build to fail if only part of the configuration documentation has been generated
- The documentation is generated with a title level of 2 (i.e. `==`). You usually need to adjust it. It can be done with `leveloffset=+N`.

It is not recommended to include the whole configuration documentation in the middle of your guide as it's heavy. If you have an `application.properties` extract with your configuration, just do as follows.

First, include a tip just below your `application.properties` extract:

```
[TIP]
For more information about the extension configuration please refer
to the <<configuration-reference, Configuration Reference>>.
```

Then, at the end of your documentation, include the extensive documentation:

```
[[configuration-reference]]
== Configuration Reference

include::{generated-dir}/config/quarkus-your-
extension.adoc[opts=optional, leveloffset=+1]
```

Finally, generate the documentation and check it out.

4. Continuous testing of your extension

In order to make it easy for extension authors to test their extensions daily against the latest snapshot of Quarkus, Quarkus has introduced the notion of Ecosystem CI. The Ecosystem CI README has all the details on how to set up a GitHub Actions job to take advantage of this capability, while this video provides an overview of what the process looks like.