

# Quarkus - Contexts and Dependency Injection

Quarkus DI solution is based on the [Contexts and Dependency Injection for Java 2.0](#) specification. However, it is not a full CDI implementation verified by the TCK. Only a subset of the CDI features is implemented - see also [the list of supported features](#) and [the list of limitations](#).



Most of the existing CDI code should work just fine but there are some small differences which follow from the Quarkus architecture and goals.

## 1. Bean Discovery

Bean discovery in CDI is a complex process which involves legacy deployment structures and accessibility requirements of the underlying module architecture. However, Quarkus is using a **simplified bean discovery**. There is only single bean archive with the **bean discovery mode annotated** and no visibility boundaries.

The bean archive is synthesized from:

- the application classes,
- dependencies that contain a **beans.xml** descriptor (content is ignored),
- dependencies that contain a Jandex index - **META-INF/jandex.idx**,
- dependencies referenced by **quarkus.index-dependency** in **application.properties**,
- and Quarkus integration code.

Bean classes that don't have a **bean defining annotation** are not discovered. This behavior is defined by CDI. But producer methods and fields and observer methods are discovered even if the declaring class is not annotated with a bean defining annotation (this behavior is different to what is defined in CDI). In fact, the declaring bean classes are considered annotated with **@Dependent**.



Quarkus extensions may declare additional discovery rules. For example, **@Scheduled** business methods are registered even if the declaring class is not annotated with a bean defining annotation.

### 1.1. How to Generate a Jandex Index

A dependency with a Jandex index is automatically scanned for beans. To generate the index just add the following to your **pom.xml**:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jboss.jandex</groupId>
      <artifactId>jandex-maven-plugin</artifactId>
      <version>1.0.7</version>
      <executions>
        <execution>
          <id>make-index</id>
          <goals>
            <goal>jandex</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

If you can't modify the dependency, you can still index it by adding `quarkus.index-dependency` entries to your `application.properties`:

```
quarkus.index-dependency.<name>.group-id=
quarkus.index-dependency.<name>.artifact-id=
quarkus.index-dependency.<name>.classifier=(this one is optional)
```

For example, the following entries ensure that the `org.acme:acme-api` dependency is indexed:

```
quarkus.index-dependency.acme.group-id=org.acme ①
quarkus.index-dependency.acme.artifact-id=acme-api ②
```

① Value is a group id for a dependency identified by name `acme`.

② Value is an artifact id for a dependency identified by name `acme`.

## 2. Native Executables and Private Members

Quarkus is using GraalVM to build a native executable. One of the limitations of GraalVM is the usage of [Reflection](#). Reflective operations are supported but all relevant members must be registered for reflection explicitly. Those registrations result in a bigger native executable.

And if Quarkus DI needs to access a private member it **has to use reflection**. That's why Quarkus users are encouraged *not to use private members* in their beans. This involves injection fields, constructors and initializers, observer methods, producer methods and fields, disposers and interceptor methods.

How to avoid using private members? You can use package-private modifiers:

```

@ApplicationScoped
public class CounterBean {

    @Inject
    CounterService counterService; ①

    void onMessage(@Observes Event msg) { ②
    }
}

```

① A package-private injection field.

② A package-private observer method.

Or constructor injection:

```

@ApplicationScoped
public class CounterBean {

    private CounterService service;

    CounterBean(CounterService service) { ①
        this.service = service;
    }
}

```

① A package-private constructor injection. `@Inject` is optional in this particular case.

## 3. Supported Features

- Programming model
  - Managed beans implemented by a Java class
    - `@PostConstruct` and `@PreDestroy` lifecycle callbacks
  - Producer methods and fields, disposers
  - Qualifiers
  - Alternatives
  - Stereotypes
- Dependency injection and lookup
  - Field, constructor and initializer/setter injection
  - Type-safe resolution
  - Programmatic lookup via `javax.enterprise.inject.Instance`
  - Client proxies

- Injection point metadata
- Scopes and contexts
  - `@Dependent`, `@ApplicationScoped`, `@Singleton`, `@RequestScoped` and `@SessionScoped`
  - Custom scopes and contexts
- Interceptors
  - Business method interceptors: `@AroundInvoke`
  - Interceptors for lifecycle event callbacks: `@PostConstruct`, `@PreDestroy`, `@AroundConstruct`
- Events and observer methods, including asynchronous events and transactional observer methods

## 4. Limitations

- `@ConversationScoped` is not supported
- Decorators are not supported
- Portable Extensions are not supported
- `BeanManager` - only the following methods are implemented: `getBeans()`, `createCreationalContext()`, `getReference()`, `getInjectableReference()`, `resolve()`, `getContext()`, `fireEvent()`, `getEvent()` and `createInstance()`
- Specialization is not supported
- `beans.xml` descriptor content is ignored
- Passivation and passivating scopes are not supported
- Interceptor methods on superclasses are not implemented yet

## 5. Non-standard Features

### 5.1. Eager Instantiation of Beans

#### 5.1.1. Lazy By Default

By default, CDI beans are created lazily, when needed. What exactly "needed" means depends on the scope of a bean.

- A **normal scoped bean** (`@ApplicationScoped`, `@RequestScoped`, etc.) is needed when a method is invoked upon an injected instance (contextual reference per the specification).

In other words, injecting a normal scoped bean will not suffice because a *client proxy* is injected instead of a contextual instance of the bean.

- A **bean with a pseudo-scope** (`@Dependent` and `@Singleton`) is created when injected.

## Lazy Instantiation Example

```
@Singleton // => pseudo-scope
class AmazingService {
    String ping() {
        return "amazing";
    }
}

@ApplicationScoped // => normal scope
class CoolService {
    String ping() {
        return "cool";
    }
}

@Path("/ping")
public class PingResource {

    @Inject
    AmazingService s1; ①

    @Inject
    CoolService s2; ②

    @GET
    public String ping() {
        return s1.ping() + s2.ping(); ③
    }
}
```

- ① Injection triggers the instantiation of **AmazingService**.
- ② Injection itself does not result in the instantiation of **CoolService**. A client proxy is injected.
- ③ The first invocation upon the injected proxy triggers the instantiation of **CoolService**.

### 5.1.2. Startup Event

However, if you really need to instantiate a bean eagerly you can:

- Declare an observer of the **StartupEvent** - the scope of the bean does not matter in this case:

```
@ApplicationScoped
class CoolService {
    void startup(@Observes StartupEvent event) { ①
    }
}
```

① A `CoolService` is created during startup to service the observer method invocation.

- Use the bean in an observer of the `StartupEvent` - normal scoped beans must be used as described in [Lazy By Default](#):

```
@Dependent
class MyBeanStarter {

    void startup(@Observes StartupEvent event, AmazingService
amazing, CoolService cool) { ①
        cool.toString(); ②
    }
}
```

① The `AmazingService` is created during injection.

② The `CoolService` is a normal scoped bean so we have to invoke a method upon the injected proxy to force the instantiation.



Quarkus users are encouraged to always prefer the `@Observes StartupEvent` to `@Initialized(ApplicationScoped.class)` as explained in the [Application Initialization and Termination](#) guide.

## 5.2. Request Context Lifecycle

The request context is also active:

- during notification of a synchronous observer method.

The request context is destroyed:

- after the observer notification completes for an event, if it was not already active when the notification started.



An event with qualifier `@Initialized(RequestScoped.class)` is fired when the request context is initialized for an observer notification. Moreover, the events with qualifiers `@BeforeDestroyed(RequestScoped.class)` and `@Destroyed(RequestScoped.class)` are fired when the request context is destroyed.

## 5.3. Qualified Injected Fields

In CDI, if you declare a field injection point you need to use `@Inject` and optionally a set of qualifiers:

```
@Inject
@ConfigProperty(name = "cool")
String coolProperty;
```

In Quarkus, you can skip the `@Inject` annotation completely if an injected field declares at least one qualifier:

```
@ConfigProperty(name = "cool")
String coolProperty;
```



With the notable exception of one special case discussed below, `@Inject` is still required for constructor and method injection.

## 5.4. Simplified Constructor Injection

In CDI, a normal scoped bean must always declare a no-args constructor (this constructor is normally generated by the compiler unless you declare any other constructor). However, this requirement complicates constructor injection - you need to provide a dummy no-args constructor to make things work in CDI.

```
@ApplicationScoped
public class MyCoolService {

    private SimpleProcessor processor;

    MyCoolService() { // dummy constructor needed
    }

    @Inject // constructor injection
    MyCoolService(SimpleProcessor processor) {
        this.processor = processor;
    }
}
```

There is no need to declare dummy constructors for normal scoped bean in Quarkus - they are generated automatically. Also if there's only one constructor there is no need for `@Inject`.

```
@ApplicationScoped
public class MyCoolService {

    private SimpleProcessor processor;

    MyCoolService(SimpleProcessor processor) {
        this.processor = processor;
    }
}
```



We don't generate a no-args constructor automatically if a bean class extends a class that does not declare a no-args constructor.

## 5.5. Removing Unused Beans

The container attempts to remove all unused beans during build by default. This optimization can be disabled by setting `quarkus.arc.remove-unused-beans` to `none` or `false`.

An unused bean:

- is not a built-in bean or an interceptor,
- is not eligible for injection to any injection point,
- is not excluded by any extension,
- does not have a name,
- does not declare an observer,
- does not declare any producer which is eligible for injection to any injection point,
- is not directly eligible for injection into any `javax.enterprise.inject.Instance` or `javax.inject.Provider` injection point

This optimization applies to all forms of bean declarations: bean class, producer method, producer field.

Users can instruct the container to not remove any of their specific beans (even if they satisfy all the rules specified above) by annotating them with `io.quarkus.arc.Unremovable`. This annotation can be placed on the types, producer methods, and producer fields.

Furthermore, extensions can eliminate possible false positives by producing `UnremovableBeanBuildItem`.

Finally, Quarkus provides a middle ground for the bean removal optimization where application beans are never removed whether or not they are unused, while the optimization proceeds normally for non application classes. To use this mode, set `quarkus.arc.remove-unused-beans` to `fwk` or `framework`.

When using the dev mode (running `mvn clean compile quarkus:dev`), you can see more

information about which beans are being removed by enabling additional logging via the following line in your `application.properties`.

```
quarkus.log.category."io.quarkus.arc.processor".level=DEBUG
```

## 5.6. Default Beans

Quarkus adds a capability that CDI currently does not support which is to conditionally declare a bean if no other bean with equal types and qualifiers was declared by any available means (bean class, producer, synthetic bean, ...) This is done using the `@io.quarkus.arc.DefaultBean` annotation and is best explained with an example.

Say there is a Quarkus extension that among other things declares a few CDI beans like the following code does:

```
@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
    }
}
```

The idea is that the extension auto-configures things for the user, eliminating a lot of boilerplate - we can just `@Inject` a `Tracer` wherever it is needed. Now imagine that in our application we would like to utilize the configured `Tracer`, but we need to customize it a little, for example by providing a custom `Reporter`. The only thing that would be needed in our application would be something like the following:

```

@Dependent
public class CustomTracerConfiguration {

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

```

`@DefaultBean` allows extensions (or any other code for that matter) to provide defaults while backing off if beans of that type are supplied in any way Quarkus supports.

## 6. Build Time Extension Points

### 6.1. Portable Extensions

Quarkus incorporates build-time optimizations in order to provide instant startup and low memory footprint. The downside of this approach is that CDI Portable Extensions cannot be supported. Nevertheless, most of the functionality can be achieved using Quarkus [extensions](#).

### 6.2. Additional Bean Defining Annotations

As described in [Bean Discovery](#) bean classes that don't have a bean defining annotation are not discovered. However, `BeanDefiningAnnotationBuildItem` can be used to extend the set of default bean defining annotations (`@Dependent`, `@Singleton`, `@ApplicationScoped`, `@RequestScoped` and `@Stereotype` annotations):

```

@BuildStep
BeanDefiningAnnotationBuildItem additionalBeanDefiningAnnotation()
{
    return new BeanDefiningAnnotationBuildItem(DotName.
createSimple("javax.ws.rs.Path"));
}

```



Bean registrations that are result of a `BeanDefiningAnnotationBuildItem` are unremovable by default. See also [Removing Unused Beans](#).

### 6.3. Resource Annotations

`ResourceAnnotationBuildItem` is used to specify resource annotations that make it possible to resolve non-CDI injection points, such as Java EE resources.



An integrator must also provide a corresponding `io.quarkus.arc.ResourceReferenceProvider` implementation.

```
@BuildStep
void setupResourceInjection(BuildProducer
<ResourceAnnotationBuildItem> resourceAnnotations, BuildProducer
<GeneratedResourceBuildItem> resources) {
    resources.produce(new GeneratedResourceBuildItem("META-
INF/services/io.quarkus.arc.ResourceReferenceProvider",
        JPAResourceReferenceProvider.class.getName().getBytes()));
    resourceAnnotations.produce(new ResourceAnnotationBuildItem
(DotName.createSimple(PersistenceContext.class.getName())));
}
```

## 6.4. Additional Beans

`AdditionalBeanBuildItem` is used to specify additional bean classes to be analyzed during discovery. Additional bean classes are transparently added to the application index processed by the container.

```
@BuildStep
List<AdditionalBeanBuildItem> additionalBeans() {
    return Arrays.asList(
        new AdditionalBeanBuildItem(SmallRyeHealthReporter.class
),
        new AdditionalBeanBuildItem(HealthServlet.class));
}
```



A bean registration that is a result of an `AdditionalBeanBuildItem` is removable by default. See also [Removing Unused Beans](#).

## 6.5. Synthetic Beans

Sometimes it is very useful to register a synthetic bean, i.e. a bean that doesn't need to have a corresponding java class. In CDI, this could be achieved using `AfterBeanDiscovery.addBean()` methods. In Quarkus, we produce a `BeanRegistrarBuildItem` and leverage the `io.quarkus.arc.processor.BeanConfigurator` API to build a synthetic bean definition.

```

@BuildStep
BeanRegistrarBuildItem syntheticBean() {
    return new BeanRegistrarBuildItem(new BeanRegistrar() {

        @Override
        public void register(RegistrationContext
registrationContext) {
            registrationContext.configure(String.class).types
(String.class).qualifiers(new MyQualifierLiteral()).creator(mc ->
mc.returnValue(mc.load("foo"))).done();
        }
    }));
}

```



The output of a **BeanConfigurator** is recorded as bytecode. Therefore there are some limitations in how a synthetic bean instance is created. See also **BeanConfigurator.creator()** methods.



You can easily filter all class-based beans via the convenient **BeanStream** returned from the **RegistrationContext.beans()** method.

If an extension needs to produce other build items during the "bean registration" phase it should use the **BeanRegistrationPhaseBuildItem** instead. The reason is that injected objects are only valid during a **@BuildStep** method invocation.

```

@BuildStep
void syntheticBean(BeanRegistrationPhaseBuildItem
beanRegistrationPhase,
    BuildProducer<MyBuildItem> myBuildItem,
    BuildProducer<BeanConfiguratorBuildItem>
beanConfigurators) {
    beanConfigurators.produce(new BeanConfiguratorBuildItem
(beanRegistrationPhase.getContext().configure(String.class).types(S
tring.class).qualifiers(new MyQualifierLiteral()).creator(mc -> mc
.returnValue(mc.load("foo"))));
    myBuildItem.produce(new MyBuildItem());
}

```



See **BeanRegistrationPhaseBuildItem** javadoc for more information.

## 6.6. Annotation Transformations

A very common task is to override the annotations found on the bean classes. For example you might want to add an interceptor binding to a specific bean class. Here is how to do it - use the **AnnotationsTransformerBuildItem**:

```

@BuildStep
AnnotationsTransformerBuildItem transform() {
    return new AnnotationsTransformerBuildItem(new
AnnotationsTransformer() {

        public boolean appliesTo(org.jboss.jandex.AnnotationTarget
.Kind kind) {
            return kind == org.jboss.jandex.AnnotationTarget.Kind
.CLASS;
        }

        public void transform(TransformationContext context) {
            if (context.getTarget().asClass().name().toString()
.equals("com.foo.Bar")) {
                context.transform().add(MyInterceptorBinding.class
).done();
            }
        }
    });
}

```

## 6.7. Additional Interceptor Bindings

In rare cases it might be handy to programmatically register an existing annotation as interceptor binding. This is similar to what pure CDI achieves through `BeforeBeanDiscovery#addInterceptorBinding()`. Though here we are going to use `InterceptorBindingRegistrarBuildItem` to get it done. Note that you can register multiple annotations in one go:

```

@BuildStep
InterceptorBindingRegistrarBuildItem addInterceptorBindings() {
    InterceptorBindingRegistrarBuildItem
    additionalBindingsRegistrar = new
    InterceptorBindingRegistrarBuildItem(new
    InterceptorBindingRegistrar() {
        @Override
        public Collection<DotName> registerAdditionalBindings() {
            Collection<DotName> result = new HashSet<>();
            result.add(DotName.createSimple(MyAnnotation.class
            .getName()));
            result.add(DotName.createSimple(MyOtherAnnotation.
            class.getName()));
            return result;
        }
    });
    return additionalBindingsRegistrar;
}

```

## 6.8. Injection Point Transformation

Every now and then it is handy to be able to change qualifiers of an injection point programmatically. You can do just that with `InjectionPointTransformerBuildItem`. The following sample shows how to apply transformation to injection points with type `Foo` that contain qualifier `MyQualifier`:

```

@BuildStep
InjectionPointTransformerBuildItem transformer() {
    return new InjectionPointTransformerBuildItem(new
InjectionPointsTransformer() {

        public boolean appliesTo(Type requiredType) {
            return requiredType.name().equals(DotName.createSimple
(Foo.class.getName()));
        }

        public void transform(TransformationContext context) {
            if (context.getQualifiers().stream()
                .anyMatch(a -> a.name().equals(DotName
.createSimple(MyQualifier.class.getName())))) {
                context.transform()
                    .removeAll()
                    .add(DotName.createSimple(MyOtherQualifier
.class.getName()))
                    .done();
            }
        }
    });
}

```

## 6.9. Observer Transformation

Any `observer method` definition can be vetoed or transformed using an `ObserverTransformerBuildItem`. The attributes that can be transformed include:

- `qualifiers`
- `reception`
- `priority`
- `transaction phase`
- `asynchronous`

```

@BuildStep
ObserverTransformerBuildItem transformer() {
    return new ObserverTransformerBuildItem(new
ObserverTransformer() {

        public boolean appliesTo(Type observedType, Set
<AnnotationInstance> qualifiers) {
            return observedType.name.equals(DotName.createSimple
(MyEvent.class.getName()));
        }

        public void transform(TransformationContext context) {
            // Veto all observers of MyEvent
            context.veto();
        }
    });
}

```

## 6.10. Bean Deployment Validation

Once the bean deployment is ready an extension can perform additional validations and inspect the found beans, observers and injection points. Register a `BeanDeploymentValidatorBuildItem`:

```

@BuildStep
BeanDeploymentValidatorBuildItem beanDeploymentValidator() {
    return new BeanDeploymentValidatorBuildItem(new
BeanDeploymentValidator() {
        public void validate(ValidationContext validationContext)
{
            for (InjectionPointInfo injectionPoint :
validationContext.get(Key.INJECTION_POINTS)) {
                System.out.println("Injection point: " +
injectionPoint);
            }
        }
    });
}

```



You can easily filter all registered beans via the convenient `BeanStream` returned from the `ValidationContext.beans()` method.

If an extension needs to produce other build items during the "validation" phase it should use the `ValidationPhaseBuildItem` instead. The reason is that injected objects are only valid during a `@BuildStep` method invocation.

```

@BuildStep
void validate(ValidationPhaseBuildItem validationPhase,
             BuildProducer<MyBuildItem> myBuildItem,
             BuildProducer<ValidationErrorBuildItem> errors) {
    if (someCondition) {
        errors.produce(new ValidationErrorBuildItem(new
IllegalStateException()));
        myBuildItem.produce(new MyBuildItem());
    }
}

```



See [ValidationPhaseBuildItem](#) javadoc for more information.

## 6.11. Custom Contexts

An extension can register a custom [InjectableContext](#) implementation by means of a [ContextRegistrarBuildItem](#):

```

@BuildStep
ContextRegistrarBuildItem customContext() {
    return new ContextRegistrarBuildItem(new ContextRegistrar() {
        public void register(RegistrationContext
registrationContext) {
            registrationContext.configure(CustomScoped.class)
.normal().contextClass(MyCustomContext.class).done();
        }
    });
}

```

If an extension needs to produce other build items during the "context registration" phase it should use the [ContextRegistrationPhaseBuildItem](#) instead. The reason is that injected objects are only valid during a [@BuildStep](#) method invocation.

```

@BuildStep
void addContext(ContextRegistrationPhaseBuildItem
contextRegistrationPhase,
             BuildProducer<MyBuildItem> myBuildItem,
             BuildProducer<ContextConfiguratorBuildItem> contexts) {
    contexts.produce(new ContextConfiguratorBuildItem
(contextRegistrationPhase.getContext().configure(CustomScoped.class)
.normal().contextClass(MyCustomContext.class)));
    myBuildItem.produce(new MyBuildItem());
}

```



See `ContextRegistrationPhaseBuildItem` javadoc for more information.

## 6.12. Available Build Time Metadata

Any of the above extensions that operates with `BuildExtension.BuildContext` can leverage certain build time metadata that are generated during build. The built-in keys located in `io.quarkus.arc.processor.BuildExtension.Key` are:

- `ANNOTATION_STORE`
  - Contains an `AnnotationStore` that keeps information about all `AnnotationTarget` annotations after application of annotation transformers
- `INJECTION_POINTS`
  - `Collection<InjectionPointInfo>` containing all injection points
- `BEANS`
  - `Collection<BeanInfo>` containing all beans
- `REMOVED_BEANS`
  - `Collection<BeanInfo>` containing all the removed beans; see [Removing Unused Beans](#) for more information
- `OBSERVERS`
  - `Collection<ObserverInfo>` containing all observers
- `SCOPES`
  - `Collection<ScopeInfo>` containing all scopes, including custom ones
- `QUALIFIERS`
  - `Map<DotName, ClassInfo>` containing all qualifiers
- `INTERCEPTOR_BINDINGS`
  - `Map<DotName, ClassInfo>` containing all interceptor bindings
- `STEREOTYPES`
  - `Map<DotName, ClassInfo>` containing all stereotypes

To get hold of these, simply query the extension context object for given key. Note that these metadata are made available as build proceeds which means that extensions can only leverage metadata that were build before they are invoked. If your extension attempts to retrieve metadata that wasn't yet produced, `null` will be returned. Here is a summary of which extensions can access which metadata:

- `AnnotationsTransformer`
  - Shouldn't rely on any metadata as this is one of the first CDI extensions invoked
- `ContextRegistrar`
  - Has access to `ANNOTATION_STORE`

- `InjectionPointsTransformer`
  - Has access to `ANNOTATION_STORE`, `QUALIFIERS`, `INTERCEPTOR_BINDINGS`, `STEREOTYPES`
- `ObserverTransformer`
  - Has access to `ANNOTATION_STORE`, `QUALIFIERS`, `INTERCEPTOR_BINDINGS`, `STEREOTYPES`
- `BeanRegistrar`
  - Has access to `ANNOTATION_STORE`, `QUALIFIERS`, `INTERCEPTOR_BINDINGS`, `STEREOTYPES`, `BEANS`
- `BeanDeploymentValidator`
  - Has access to all build metadata