Quarkus - Introduction to Contexts and Dependency Injection

In this guide we're going to describe the basic principles of the Quarkus programming model that is based on the Contexts and Dependency Injection for Java 2.0 specification.

1. *Q: OK. Let's start simple. What is a bean?*

A: Well, a bean is a *container-managed* object that supports a set of basic services, such as injection of dependencies, lifecycle callbacks and interceptors.

2. *Q: Wait a minute. What does "container-managed" mean?*

A: Simply put, you don't control the lifecycle of the object instance directly. Instead, you can affect the lifecycle through declarative means, such as annotations, configuration, etc. The container is the *environment* where your application runs. It creates and destroys the instances of beans, associates the instances with a designated context, and injects them into other beans.

3. *Q: What is it good for?*

A: An application developer can focus on the business logic rather than finding out "where and how" to obtain a fully initialized component with all of its dependencies.

> ℹ️ You've probably heard of the *inversion of control* (IoC) programming principle. Dependency injection is one of the implementation techniques of IoC.

4. *Q: What does a bean look like?*

A: There are several kinds of beans. The most common ones are class-based beans:

Simple Bean Example

```
import javax.inject.Inject;
import javax.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.metrics.annotation.Counted;

@ApplicationScoped ①
public class Translator {

    @Inject
    Dictionary dictionary; ②

    @Counted   ③
    String translate(String sentence) {
      // ...
    }
}
```

① This is a scope annotation. It tells the container which context to associate the bean instance with. In this particular case, a **single bean instance** is created for the application and used by all other beans that inject `Translator`.

② This is a field injection point. It tells the container that `Translator` depends on the `Dictionary` bean. If there is no matching bean the build fails.

③ This is an interceptor binding annotation. In this case, the annotation comes from the MicroProfile Metrics. The relevant interceptor intercepts the invocation and updates the relevant metrics. We will about interceptors later.

5. *Q: Nice. How does the dependency resolution work? I see no names or identifiers.*

A: That's a good question. In CDI the process of matching a bean to an injection point is **type-safe**. Each bean declares a set of bean types. In our example above, the `Translator` bean has two bean types: `Translator` and `java.lang.Object`. Subsequently, a bean is assignable to an injection point if the bean has a bean type that matches the *required type* and has all the *required qualifiers*. We'll talk about qualifiers later. For now, it's enough to know that the bean above is assignable to an injection point of type `Translator` and `java.lang.Object`.

6. *Q: Hm, wait a minute. What happens if multiple beans declare the same type?*

A: There is a simple rule: **exactly one bean must be assignable to an injection point, otherwise the build fails**. If none is assignable the build fails with `UnsatisfiedResolutionException`. If multiple are assignable the build fails with `AmbiguousResolutionException`. This is very useful because your application fails fast whenever the container is not able to find an unambiguous dependency for any injection point.

Your can use programmatic lookup via `javax.enterprise.inject.Instance` to resolve ambiguities at runtime and even iterate over all beans implementing a given type:

```
public class Translator {

    @Inject
    Instance<Dictionary> dictionaries;  ①

    String translate(String sentence) {
      for (Dictionary dict : dictionaries) {  ②
          // ...
      }
    }
}
```

① This injection point will not result in an ambiguous dependency even if there are multiple beans that implement the `Dictionary` type.

② `javax.enterprise.inject.Instance` extends `Iterable`.

7. *Q: Can I use setter and constructor injection?*

A: Yes, you can. In fact, in CDI the "setter injection" is superseded by more powerful initializer methods. Intializers may accept multiple parameters and don't have to follow the JavaBean naming conventions.

Initialized and Constructor Injection Example

```
@ApplicationScoped
public class Translator {

    private final TranslatorHelper helper

    Translator(TranslatorHelper helper) {  ①
        this.helper = helper;
    }

    @Inject  ②
    void setDeps(Dictionary dic, LocalizationService locService) {
③
      / ...
    }
}
```

① This is a constructor injection. In fact, this code would not work in regular CDI implementations where a bean with a normal scope must always declare a no-args constructor and the bean constructor must be annotated with `@Inject`. However, in Quarkus we detect the absence of no-

args constructor and "add" it directly in the bytecode. It's also not necessary to add `@Inject` if there is only one constructor present.

② An initializer method must be annotated with `@Inject`.

③ An initializer may accept multiple parameters - each one is an injection point.

8. *Q: You talked about some qualifiers?*

A: Qualifiers are annotations that help the container to distinguish beans that implement the same type. As we already said a bean is assignable to an injection point if it has all the required qualifiers. If you declare no qualifier at an injection point the `@Default` qualifier is assumed.

A qualifier type is a Java annotation defined as `@Retention(RUNTIME)` and annotated with the `@javax.inject.Qualifier` meta-annotation:

Qualifier Example

```
@Qualifier
@Retention(RUNTIME)
@Target({METHOD, FIELD, PARAMETER, TYPE})
public @interface Superior {}
```

The qualifiers of a bean are declared by annotating the bean class or producer method or field with the qualifier types:

Bean With Custom Qualifier Example

```
@Superior  ①
@ApplicationScoped
public class SuperiorTranslator extends Translator {

    String translate(String sentence) {
      // ...
    }
}
```

① @Superior is a qualifier annotation.

This bean would be assignable to `@Inject @Superior Translator` and `@Inject @Superior SuperiorTranslator` but not to `@Inject Translator`. The reason is that `@Inject Translator` is automatically transformed to `@Inject @Default Translator` during typesafe resolution. And since our `SuperiorTranslator` does not declare `@Default` only the original `Translator` bean is assignable.

9. *Q: Looks good. What is the bean scope?*

The scope of a bean determines the lifecycle of its instances, i.e. when and where an instance should be created and destroyed.

> **i**   Every bean has exactly one scope.

10. *Q: What scopes can I actually use in my Quarkus application?*

A: You can use all the built-in scopes mentioned by the specification except for `javax.enterprise.context.ConversationScoped`.

Annotation	Description
`@javax.enterprise.context.ApplicationScoped`	A single bean instance is used for the application and shared among all injection points. The instance is created lazily, i.e. once a method is invoked upon the client proxy.
`@javax.inject.Singleton`	Just like @ApplicationScoped except that no client proxy is used. The instance is created when an injection point that resolves to a @Singleton bean is being injected.
`@javax.enterprise.context.RequestScoped`	The bean instance is associated with the current *request* (usually an HTTP request).
`@javax.enterprise.context.Dependent`	This is a pseudo-scope. The instances are not shared and every injection point spawns a new instance of the dependent bean. The lifecycle of dependent bean is bound to the bean injecting it - it will be created and destroyed along with the bean injecting it.
`@javax.enterprise.context.SessionScoped`	This scope is backed by an `javax.servlet.http.HttpSession` object. It's only available if the `quarkus-undertow` extension is used.

> **i**   There can be other custom scopes provided by Quarkus extensions. For example, `quarkus-narayana-jta` provides `javax.transaction.TransactionScoped`.

11. *Q: I don't undestand the concept of client proxies.*

Indeed, the client proxies could be hard to grasp but they provide some useful functionality. A client proxy is basically an object that delegates all method invocations to a target bean instance. It's a container construct that implements `io.quarkus.arc.ClientProxy` and extends the bean class.

Generated Client Proxy Example

```
@ApplicationScoped
class Translator {

    String translate(String sentence) {
      // ...
    }
}

// The client proxy class is generated and looks like...
class Translator_ClientProxy extends Translator { ①

    String translate(String sentence) {
       // Find the correct translator instance...
       Translator translator =
getTranslatorInstanceFromTheApplicationContext();
       // And delegate the method invocation...
       return translator.translate(sentence);
    }
}
```

① The `Translator_ClientProxy` instance is always injected instead of a direct reference to a contextual instance of the `Translator` bean.

Client proxies allow for:

- Lazy instantiation - the instance is created once a method is invoked upon the proxy.

- Ability to inject a bean with "narrower" scope to a bean with "wider" scope; i.e. you can inject a `@RequestScoped` bean into an `@ApplicationScoped` bean.

- Circular dependencies in the dependency graph. Having circular dependencies is often an indication that a redesign should be considered, but sometimes it's inevitable.

- In rare cases it's practical to destroy the beans manually. A direct injected reference would lead to a stale bean instance.

12. *Q: OK. You said that there are several kinds of beans?*

A: Yes. In general, we distinguish:

1. Class beans

2. Producer methods

3. Producer fields

4. Synthetic beans

> ℹ️ Synthetic beans are usually provided by extensions. Therefore, we not going to cover them in this guide.

Producer methods and fields are useful if you need additional control over instantiation of a bean. They are also useful when integrating third-party libraries where you don't control the class source and may not add additional annotations etc.

Producers Example

```
@ApplicationScoped
public class Producers {

    @Produces ①
    double pi = Math.PI; ②

    @Produces ③
    List<String> names() {
        List<String> names = new ArrayList<>();
        names.add("Andy");
        names.add("Adalbert");
        names.add("Joachim");
        return names; ④
    }
}

@ApplicationScoped
public class Consumer {

    @Inject
    double pi;

    @Inject
    List<String> names;

    // ...
}
```

① The container analyses the field annotations to build a bean metadata. The *type* is used to build the set of bean types. In this case, it will be `double` and `java.lang.Object`. No scope annotation is declared and so it's defaulted to `@Dependent`.

② The container will read this field when creating the bean instance.

③ The container analyses the method annotations to build a bean metadata. The *return type* is used to build the set of bean types. In this case, it will be `List<String>`, `Collection<String>`, `Iterable<String>` and `java.lang.Object`. No scope annotation is declared and so it's defaulted to `@Dependent`.

④ The container will call this method when creating the bean instance.

There's more about producers. You can declare qualifiers, inject dependencies into the producer methods parameters, etc. You can read more about producers for example in the Weld docs.

13. *Q: OK, injection looks cool. What other services are provided?*

13.1. Lifecycle Callbacks

A bean class may declare lifecycle `@PostConstruct` and `@PreDestroy` callbacks:

Lifecycle Callbacks Example

```java
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@ApplicationScoped
public class Translator {

    @PostConstruct ①
    void init() {
        // ...
    }

    @PreDestroy ②
    void destroy() {
        // ...
    }
}
```

① This callback is invoked before the bean instance is put into service. It is safe to perform some initialization here.

② This callback is invoked before the bean instance is destroyed. It is safe to perform some cleanup tasks here.

It's a good practice to keep the logic in the callbacks "without side effects", i.e. you should avoid calling other beans inside the callbacks.

13.2. Interceptors

Interceptors are used to separate cross-cutting concerns from business logic. There is a separate specification - Java Interceptors - that defines the basic programming model and semantics.

Simple Interceptor Example

```
import javax.interceptor.Interceptor;
import javax.annotation.Priority;

@Logged ①
@Priority(2020) ②
@Interceptor ③
public class LoggingInterceptor {

    @Inject ④
    Logger logger;

    @AroundInvoke ⑤
    Objec logInvocation(InvocationContext context) {
        // ...log before
        Objec ret = context.proceed(); ⑥
        // ...log after
        return ret;
    }

}
```

① This is an interceptor binding annotation that is used to bind our interceptor to a bean. Simply annotate a bean class with `@Logged`.

② `Priority` enables the interceptor and affects the interceptor ordering. Interceptors with smaller priority values are called first.

③ Marks an interceptor component.

④ An interceptor instance may be the target of dependency injection.

⑤ `AroundInvoke` denotes a method that interposes on business methods.

⑥ Proceed to the next interceptor in the interceptor chain or invoke the intercepted business method.

Instances of interceptors are dependent objects of the bean instance they intercept, i.e. a new interceptor instance is created for each intercepted bean.

13.3. Events and Observers

Beans may also produce and consume events to interact in a completely decoupled fashion. Any Java object can serve as an event payload. The optional qualifiers act as topic selectors.

Simple Event Example

```java
class TaskCompleted {
  // ...
}

@ApplicationScoped
class ComplicatedService {

    @Inject
    Event<Task> event;  ①

    void doSomething() {
        // ...
        event.fire(new TaskCompleted());  ②
    }

}

@ApplicationScoped
class Logger {

    void onTaskCompleted(@Observes TaskCompleted task) {  ③
        // ...log the task
    }

}
```

① `javax.enterprise.event.Event` is used to fire events.

② Fire the event synchronously.

③ This method is notified when a `TaskCompleted` event is fired.

> ☼ For more info about events/observers visit Weld docs.

14. Conclusion

In this guide, we've covered some of the basic topics of the Quarkus programming model that is based on the Contexts and Dependency Injection for Java 2.0 specification. However, a full CDI implementation is not used under the hood. Quarkus only implements a subset of the CDI features - see also the list of supported features and the list of limitations. On the other hand, there are quite a few non-standard features and Quarkus-specific APIs. We believe that our efforts will drive the innovation of the CDI specification towards the build-time oriented developer stacks in the future.

If you wish to learn more about Quarkus-specific features and limitations there is a Quarkus CDI Reference Guide. We also recommend you to read the CDI specification and the Weld documentation (Weld is a CDI Reference Implementation) to get acquainted with more complex topics.