

Quarkus - Quarkus Extension for Spring DI API

While users are encouraged to use CDI annotations for injection, Quarkus provides a compatibility layer for Spring dependency injection in the form of the `spring-di` extension.

This guide explains how a Quarkus application can leverage the well known Dependency Injection annotations included in the Spring Framework.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

Prerequisites

To complete this guide, you need:

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

Solution

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

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

The solution is located in the `spring-di-quickstart` directory.

Creating the Maven project

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

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=spring-di-quickstart \
  -DclassName="org.acme.spring.di.GreeterResource" \
  -Dpath="/greeting" \
  -Dextensions="spring-di"
cd spring-di-quickstart
```

This command generates a Maven project with a REST endpoint and imports the `spring-di` extension.

If you already have your Quarkus project configured, you can add the `spring-di` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="spring-di"
```

This will add the following to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-spring-di</artifactId>
</dependency>
```

Add beans using Spring annotations

Let's proceed to create some beans using various Spring annotations.

First we will create a `StringFunction` interface that some of our beans will implement and which will be injected into another bean later on. Create a `src/main/java/org/acme/spring/di/StringFunction.java` file and set the following content:

```
package org.acme.spring.di;

import java.util.function.Function;

public interface StringFunction extends Function<String, String> {

}
```

With the interface in place, we will add an `AppConfiguration` class which will use the Spring's Java Config style for defining a bean. It will be used to create a `StringFunction` bean that will capitalize the text passed as parameter. Create a `src/main/java/org/acme/spring/di/AppConfiguration.java` file with the following

content:

```
package org.acme.spring.di;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfiguration {

    @Bean(name = "capitalizeFunction")
    public StringFunction capitalizer() {
        return String::toUpperCase;
    }
}
```

Now we define another bean that will implement `StringFunction` using Spring's stereotype annotation style. This bean will effectively be a no-op bean that simply returns the input as is. Create a `src/main/java/org/acme/spring/di/NoOpSingleStringFunction.java` file and set the following content:

```
package org.acme.spring.di;

import org.springframework.stereotype.Component;

@Component("noopFunction")
public class NoOpSingleStringFunction implements StringFunction {

    @Override
    public String apply(String s) {
        return s;
    }
}
```

Quarkus also provides support for injecting configuration values using Spring's `@Value` annotation. To see that in action, first edit the `src/main/resources/application.properties` with the following content:

```
# Your configuration properties
greeting.message = hello
```

Next create a new Spring bean in `src/main/java/org/acme/spring/di/MessageProducer.java` with the following content:

```
package org.acme.spring.di;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service
public class MessageProducer {

    @Value("${greeting.message}")
    String message;

    public String getPrefix() {
        return message;
    }
}
```

The final bean we will create ties together all the previous beans. Create a `src/main/java/org/acme/spring/di/GreeterBean.java` file and copy the following content:

```

package org.acme.spring.di;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class GreeterBean {

    private final MessageProducer messageProducer;

    @Autowired
    @Qualifier("noopFunction")
    StringFunction noopStringFunction;

    @Autowired
    @Qualifier("capitalizeFunction")
    StringFunction capitalizerStringFunction;

    @Value("${greeting.suffix:!}")
    String suffix;

    public GreeterBean(MessageProducer messageProducer) {
        this.messageProducer = messageProducer;
    }

    public String greet(String name) {
        final String initialValue = messageProducer.getPrefix() + "
" + name + suffix;
        return
noopStringFunction.andThen(capitalizerStringFunction).apply(initial
Value);
    }
}

```

In the code above, we see that both field injection and constructor injection are being used (note that constructor injection does not need the `@Autowired` annotation since there is a single constructor). Furthermore, the `@Value` annotation on `suffix` has also a default value defined, which in this case will be used since we have not defined `greeting.suffix` in `application.properties`.

Update the JAX-RS resource

Open the `src/main/java/org/acme/spring/di/GreeterResource.java` file and update it with the following content:

```
package org.acme.spring.di;

import org.springframework.beans.factory.annotation.Autowired;

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

@Path("/greeting")
public class GreeterResource {

    @Autowired
    GreeterBean greeterBean;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return greeterBean.greet("world");
    }
}
```

Update the test

We also need to update the functional test to reflect the changes made to the endpoint. Edit the `src/test/java/org/acme/spring/di/GreetingResourceTest.java` file and change the content of the `testHelloEndpoint` method to:

```
import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class GreetingResourceTest {

    @Test
    public void testHelloEndpoint() {
        given()
            .when().get("/greeting")
            .then()
                .statusCode(200)
                .body(is("HELLO WORLD!"));
    }
}
```

Package and run the application

Run the application with: `./mvnw compile quarkus:dev`. Open your browser to <http://localhost:8080/greeting>.

The result should be: `HELLO WORLD!`.

Run the application as a native

You can of course create a native image using instructions similar to [this](#) guide.

Important Technical Note

Please note that the Spring support in Quarkus does not start a Spring Application Context nor are any Spring infrastructure classes run. Spring classes and annotations are only used for reading metadata and / or are used as user code method return types or parameter types. What that means for end users, is that adding arbitrary Spring libraries will not have any effect. Moreover Spring infrastructure classes (like `org.springframework.beans.factory.config.BeanPostProcessor` for example) will not be executed.

Conversion Table

The following table shows how Spring DI annotations can be converted to CDI and / or MicroProfile annotations.

Spring	CDI / MicroProfile	Comments
@Autowired	@Inject	
@Qualifier	@Named	
@Value	@ConfigProperty	@ConfigProperty doesn't support an expression language the way @Value does, but makes the typical use cases much easier to handle
@Component	@Singleton	By default Spring stereotype annotations are singleton beans
@Service	@Singleton	By default Spring stereotype annotations are singleton beans
@Repository	@Singleton	By default Spring stereotype annotations are singleton beans
@Configuration	@ApplicationScoped	In CDI a producer bean isn't limited to the application scope, it could just as well be @Singleton or @Dependent
@Bean	@Produces	
@Scope		Doesn't have a one-to-one mapping to a CDI annotation. Depending on the value of @Scope, one of the @Singleton, @ApplicationScoped, @SessionScoped, @RequestScoped, @Dependent could be used

More Spring guides

Quarkus has more Spring compatibility features. See the following guides for more details:

- [Quarkus - Extension for Spring Web](#)
- [Quarkus - Extension for Spring Data JPA](#)
- [Quarkus - Extension for Spring Security](#)
- [Quarkus - Reading properties from Spring Cloud Config Server](#)
- [Quarkus - Extension for Spring Boot properties](#)
- [Quarkus - Extension for Spring Cache](#)
- [Quarkus - Extension for Spring Scheduled](#)