

# Quarkus - Command Mode with Picocli

[Picocli](#) is an open source tool for creating rich command line applications.

Quarkus provides support for using Picocli. This guide contains examples of `picocli` extension usage.



This technology is considered experimental.

In *experimental* mode, early feedback is requested to mature the idea. There is no guarantee of stability nor long term presence in the platform until the solution matures. 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](#).



If you are not familiar with the Quarkus Command Mode, consider reading the [Command Mode reference guide](#) first.

## Configuration

Once you have your Quarkus project configured you can add the `picocli` extension to your project by running the following command in your project base directory.

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

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

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-picocli</artifactId>
</dependency>
```

## Simple command line application

Simple PicocliApplication with only one `Command` can be created as follows:

```

package com.acme.picocli;

import picocli.CommandLine;

import javax.enterprise.context.Dependent;
import javax.inject.Inject;

@CommandLine.Command ①
public class HelloCommand implements Runnable {

    @CommandLine.Option(names = {"-n", "--name"}, description =
    "Who will we greet?", defaultValue = "World")
    String name;

    private final GreetingService greetingService;

    public HelloCommand(GreetingService greetingService) { ②
        this.greetingService = greetingService;
    }

    @Override
    public void run() {
        greetingService.sayHello(name);
    }
}

@Dependent
class GreetingService {
    void sayHello(String name) {
        System.out.println("Hello " + name + "!");
    }
}

```

- ① If there is only one class annotated with `picocli.CommandLine.Command` it will be used as entry point to Picocli CommandLine.
- ② All classes annotated with `picocli.CommandLine.Command` are registered as CDI beans.



Beans with `@CommandLine.Command` should not use proxied scopes (e.g. do not use `@ApplicationScope`) because Picocli will not be able set field values in such beans. This extension will register classes with `@CommandLine.Command` annotation using `@Depended` scope. If you need to use proxied scope, then annotate setter and not field, for example:

```
@CommandLine.Command
@ApplicationScoped
public class EntryCommand {
    private String name;

    @CommandLine.Option(names = "-n")
    public void setName(String name) {
        this.name = name;
    }
}
```

## Command line application with multiple Commands

When multiple classes have the `picocli.CommandLine.Command` annotation, then one of them needs to be also annotated with `io.quarkus.picocli.runtime.annotations.TopCommand`. This can be overwritten with the `quarkus.picocli.top-command` property.

```

package com.acme.picocli;

import io.quarkus.picocli.runtime.annotations.TopCommand;
import picocli.CommandLine;

@TopCommand
@CommandLine.Command(mixinStandardHelpOptions = true, subcommands =
{HelloCommand.class, GoodByeCommand.class})
public class EntryCommand {
}

@CommandLine.Command(name = "hello", description = "Greet World!")
class HelloCommand implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello World!");
    }
}

@CommandLine.Command(name = "goodbye", description = "Say goodbye
to World!")
class GoodByeCommand implements Runnable {

    @Override
    public void run() {
        System.out.println("Goodbye World!");
    }
}

```

## Customizing Picocli CommandLine instance

You can customize CommandLine classes used by the `picocli` extension by producing your own bean instance:

```

package com.acme.picocli;

import io.quarkus.picocli.runtime.PicocliCommandLineFactory;
import io.quarkus.picocli.runtime.annotations.TopCommand;
import picocli.CommandLine;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Produces;

@TopCommand
@CommandLine.Command
public class EntryCommand implements Runnable {
    @CommandLine.Spec
    CommandLine.Model.CommandSpec spec;

    @Override
    public void run() {
        System.out.println("My name is: " + spec.name());
    }
}

@ApplicationScoped
class CustomConfiguration {

    @Produces
    CommandLine customCommandLine(PicocliCommandLineFactory
factory) { ①
        return factory.create().setCommandName("CustomizedName");
    }
}

```

① **PicocliCommandLineFactory** will create an instance of **CommandLine** with **TopCommand** and **CommandLine.IFactory** injected.

## Different entry command for each profile

It is possible to create different entry command for each profile, using **@IfBuildProfile**:

```

@ApplicationScoped
public class Config {

    @Produces
    @TopCommand
    @IfBuildProfile("dev")
    public Object devCommand() {
        return DevCommand.class; ①
    }

    @Produces
    @TopCommand
    @IfBuildProfile("prod")
    public Object prodCommand() {
        return new ProdCommand("Configured by me!");
    }

}

```

① You can return instance of `java.lang.Class` here. In such case `CommandLine` will try to instantiate this class using `CommandLine.IFactory`.

## Configure CDI Beans with parsed arguments

You can use `Event<CommandLine.ParseResult>` or just `CommandLine.ParseResult` to configure CDI beans based on arguments parsed by Picocli. This event will be generated in `QuarkusApplication` class created by this extension. If you are providing your own `@QuarkusMain` this event will not be raised. `CommandLine.ParseResult` is created from default `CommandLine` bean.

```

@CommandLine.Command
public class EntryCommand implements Runnable {

    @CommandLine.Option(names = "-c", description = "JDBC
connection string")
    String connectionString;

    @Inject
    DataSource dataSource;

    @Override
    public void run() {
        try (Connection c = dataSource.getConnection()) {
            // Do something
        } catch (SQLException throwables) {
            // Handle error
        }
    }
}

@ApplicationScoped
class DatasourceConfiguration {

    @Produces
    @ApplicationScoped ❶
    DataSource dataSource(CommandLine.ParseResult parseResult) {
        PGSimpleDataSource ds = new PGSimpleDataSource();

        ds.setURL(parseResult.matchedOption("c").getValue().toString());
        return ds;
    }
}

```

❶ `@ApplicationScoped` used for lazy initialization

## Providing own QuarkusMain

You can also provide your own application entry point annotated with `QuarkusMain` (as described in [Command Mode reference guide](#)).

```

package com.acme.picocli;

import io.quarkus.runtime.QuarkusApplication;
import io.quarkus.runtime.annotations.QuarkusMain;
import picocli.CommandLine;

import javax.inject.Inject;

@QuarkusMain
@CommandLine.Command(name = "demo", mixinStandardHelpOptions =
true)
public class ExampleApp implements Runnable, QuarkusApplication {
    @Inject
    CommandLine.IFactory factory; ❶

    @Override
    public void run() {
        // business logic
    }

    @Override
    public int run(String... args) throws Exception {
        return new CommandLine(this, factory).execute(args);
    }
}

```

❶ Quarkus-compatible `CommandLine.IFactory` bean created by `picocli` extension.

## Native mode support

This extension uses the Quarkus standard build steps mechanism to support GraalVM Native images. In the exceptional case that incompatible changes in a future picocli release cause any issue, the following configuration can be used to fall back to the annotation processor from the picocli project as a temporary workaround:

```

<dependency>
  <groupId>info.picocli</groupId>
  <artifactId>picocli-codegen</artifactId>
</dependency>


```


For Gradle, you need to add the following in `dependencies` section of the `build.gradle` file:



```
annotationProcessor
enforcedPlatform("${quarkusPlatformGroupId}:${quarkusPlatformArtifactId}:${quarkusPlatformVersion}")
annotationProcessor 'info.picocli:picocli-codegen'
```

## Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

| Configuration property   | Type    | Default           |
|--|---------|-------------------|
|  <code>quarkus.picocli.native-image.processing.enable</code><br><br>Set this to false to use the <code>picocli-codegen</code> annotation processor instead of build steps. CAUTION: this will have serious build-time performance impact since this is run on every restart in devmode, use with care! This property is intended to be used only in cases where an incompatible change in the picocli library causes problems in the build steps used to support GraalVM Native images. In such cases this property allows users to make the trade-off between fast build cycles with the older version of picocli, and temporarily accept slower build cycles with the latest version of picocli until the updated extension is available. | boolean | <code>true</code> |
| <code>quarkus.picocli.top-command</code><br><br>Name of bean annotated with <code>io.quarkus.picocli.runtime.annotations.TopCommand</code> or FQCN of class which will be used as entry point for Picocli CommandLine instance. This class needs to be annotated with <code>picocli.CommandLine.Command</code> .   | string  |                   |