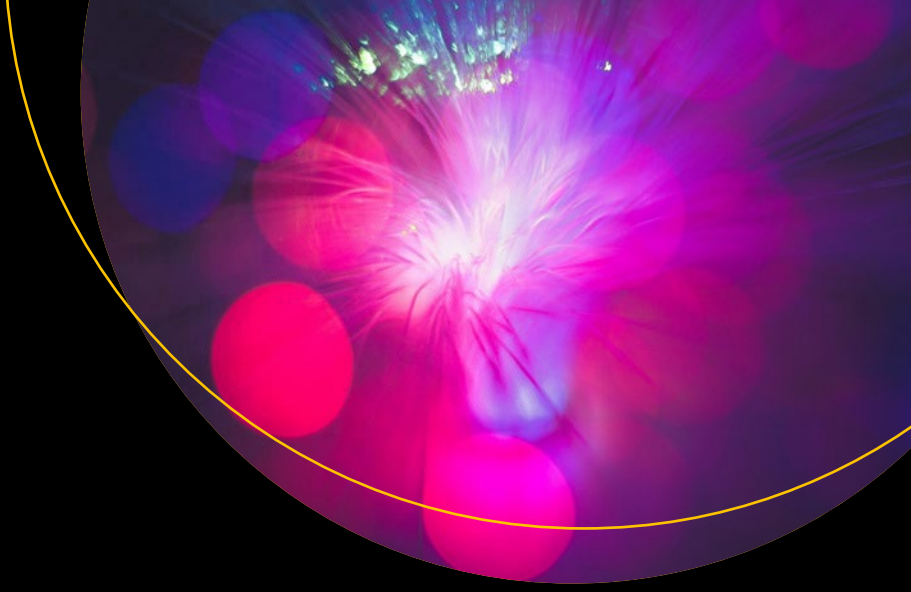

# Beginning Quarkus Framework

## Build Cloud-Native Enterprise Java Applications and Microservices

Tayo Koleoso

Apress®

# Beginning Quarkus Framework

## Build Cloud-Native Enterprise Java Applications and Microservices

Tayo Koleoso

Apress®

*Beginning Quarkus Framework: Build Cloud-Native Enterprise Java Applications and Microservices*

Tayo Koleoso
Silver Spring, MD, USA

ISBN-13 (pbk): 978-1-4842-6031-9 ISBN-13 (electronic): 978-1-4842-6032-6
https://doi.org/10.1007/978-1-4842-6032-6



Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Shahadat Rahman on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484260319. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

*I dedicate this book to my grandma,*
*who with her grade school education would*
*constantly implore me to keep studying, up till today.*

*"You know I wasn't a particularly brilliant student;*
*but you need to be far ahead of me;*
*never stop your education and passing your exams."*

*—Mrs. Cecilia Vincent*

# Table of Contents

# About the Author

**Tayo Koleoso** is a full-time technical lead and consulting architect with a burning passion for learning, because he knows there's too much he doesn't know, and teaching, because it's the best way to reinforce knowledge. He's an in-person instructor and author, dedicated to topics and technologies he'll have to study religiously to deliver. His journey started from Lagos, in Nigeria, bringing him to the United States as an immigrant software engineer. Across industries, from finance to cybersecurity, he has led teams, architected complicated integrations, and broken and built many fun and mission-critical projects in the enterprise space, with Java and Python, in the cloud. Quarkus is his latest victim.

Outside of technology, he's very passionate about personal finance and the securities market. Throw a couple of habanero peppers in, and he's happy! You can watch and follow his courses on LinkedIn Learning.

# About the Technical Reviewer

**Mouhamadou D. Sylla** is an electrical and software engineer with extensive experience and works for a hi-technology, defense and biomedical research company that provides scientific, engineering, system integration, and technical services. As a senior software engineer, he is responsible for the development integration products produced by the company. Mouhamadou has been working in software development for a decade and has participated and led several development projects in Java. His primary interests include the integration of security into software development lifecycles and emerging technologies such as Quarkus. Mouhamadou graduated from the University of Maryland, College Park, with a BS in Electrical Engineering and minor in Computer Science.

# Acknowledgments

I want to thank God, my mother, and sister for all the support and prodding to push this book through.

To my muse and #1 cheerleader, Keni, I say a huge thank you.

My technical reviewer, Mo Sylla, keeping me accurate and on target, thank you so much; it was an honor working on this book with you.

To Eden and her mum, for granting my first book "interview," thank you.

Finally, to the person that set me on this path so many years ago – he probably doesn't even remember – Femi Temowo, thank you so much.

Thanks to the Quarkus crew for building a game-changing platform! Thank you Java for existi... [Cue the walk-off music].

# Introduction

Java is dead. Long live Java.

"Cloud-native" is being thrown around in the industry a lot these days; many are joining the "microservice" train as well. Many are not ready. To be truly microservice, cloud-native, or even "cloud-friendly" takes a major mindset shift and technological realignment. Traditional popular Java application design, frameworks, and thinking can no longer deliver the goods.

Quarkus literally puts the "native" in "cloud-native"; this is not your grandma's web service framework.

This book is a view of how Java enterprise applications and microservices will be built and deployed in the future. I've carefully selected the extensions and practices to demonstrate in this text. The intent is to cover the most common use cases in the enterprise, combined with as many easily digestible and functional examples as possible. I aim to demonstrate how to

- Build scalable and cost-effective applications on premises and in the cloud
- Reliably deploy RESTful Java services in the containerized world
- Prepare you and your organization for the architectural and operational changes that are necessary for a successful migration to microservice architecture

- Run resource-efficient Java applications in deployment form factors that were hitherto impractical
- Reuse your existing code and components in the new world of Quarkus

You're not going to want to build Java the old way again. After seeing how much more "performant" your web services can become with the tools in this platform, you'll find there are very few reasons to continue to do Java as of old.

To get the most out of this book you'll need

- Familiarity with Java in an enterprise setting
- A basic understanding of web services
- A basic understanding of the cloud

This book is example-heavy; a lot of the examples can be copy-pasted straight into your IDE and run. Because of the active development going on in this book, it's highly likely that by the time you're reading it, some of it is outdated or functioning differently. This is a good thing: the Quarkus project is *very* actively developed.

After reading this book, you will be able to build and package a production-strength Java application that is natively compiled and deployable on-premise and in the cloud.

# CHAPTER 1

# Welcome to Quarkus

Quarkus is the latest entrant into the microservice arena, brought to you by our friends over at Red Hat. Now it's not like there aren't enough microservice frameworks out there, but ladies and gentlemen, this one's different. This is one of the precious few microservice frameworks engineered from the ground up for… [drum roll] the cloud.

The market is dominated arguably by the Spring Framework, Spring Boot being its flagship platform for microservices. The Spring Framework does everything and a little more, but one thing needs to be said: its cloud offerings are bolted on; afterthoughts added to a platform born before the era of cloud-*everything*, serverless, and containerization.

Quarkus is a framework built with modern software development in mind, not as an afterthought. It's a platform built to excel as a cloud deployment: as a containerized deployment, inside a stand-alone server, or in one of the common serverless frameworks. Quarkus provides almost everything we've grown accustomed to in a microservice framework like Spring Boot or Micronaut, with a lot of added benefits that put it ahead of the pack. You can run it on-premise, in the cloud, and everywhere in between.

In this chapter, we're going to take a window-shopper look at the framework and even take it for a test drive. Thank you for purchasing this book and choosing to explore this game-changing platform with me.


T. Koleoso, *Beginning Quarkus Framework*, https://doi.org/10.1007/978-1-4842-6032-6_1

# Write Once, Run ~~*Anywhere*~~ Predictably (WORP)

***Write Once, Run Anywhere*** (WORA) was the original promise of Java: you write your Java code one time, and it's good to run on any platform where. The way it fulfills that promise is by adding a lot of insulation in the JVM that protects the code from all the peculiarities of various operating systems and platforms. This is intended to mitigate any platform-specific weirdness that might cause code to behave differently.[1] The cost of that insulation is a degradation in the speed of execution, not to mention the bloat in the Java platform code that causes the size of deployment packages to swell considerably. Some even rewrote that aphorism to become "Write Once, Break Everywhere" because among other reasons, once you added application servers to the mix, things got decidedly less predictable.

***Enter the age of containerization***. Technologies like Docker and VMware Vagrant have rendered the need to write or run insulated code basically unnecessary. Containerization, the cloud, and serverless technology take a lot of the guesswork out of running code. Why should you need to keep yourself guessing what platform your code will be deployed to, when you can reliably deploy to a docker container? You no longer need to "Write Once, Run Anywhere"; you need to "Write Once, Run *Predictably*." WORP code, baby! With a WORP mindset, we can shed all the baggage of insulation that the JDK saddles us with. We can now get much smaller deployment packages. Heck, maybe our code could run a lot *fas*-.

[1]The famous "...but it works on my machine?!"

## Supersonic Subatomic!

*Supersonic* and *Subatomic* aren't 1980s-era compliments (though Quarkus is totally tubular and radical, dudes and dudettes!). No, it's a tagline that refers to two of Quarkus' biggest differentiators: this framework will usually generate much smaller deployment packages with small memory footprints (subatomic) and deploy faster (supersonic) than most other microservice frameworks on the market.

The folks over at Red Hat mean business with this framework. Quarkus contains most of all the features you've come to expect from a modern microservice framework in a shockingly compact deployment package, a package that's then engineered to start up faster than the competition [*hold for thunderous applause from the serverless crowd]*. It's truly a container-first and cloud-native microservice platform, engineered for

- Fast application startup times to enable quick scaling up or down of applications in a container
- Small memory footprint to minimize the cost of running applications in the cloud
- Predictable deployment scenarios

How is the package so small and fast? The secret sauce is a relatively new Java feature known as *ahead-of-time* (AOT) compilation. A little background on this feature, for the uninitiated (and a trip down memory lane for the platform veterans).

## A Brief Primer on JVM Internals

Today's Java is both an interpreted and a compiled language platform. Java started as an interpreted language platform: you save a source file with the `.java` extension and run the javac command to generate a `.class` file. That *class* file contains what's called Java *bytecode*, a java

language-specific interpretation of all the java code that you wrote. When you now run "`java yourcode.class`", the class file is **interpreted** by the JRE into the OS-specific CPU instructions. That intermediate step of translating the class file into CPU-friendly instructions is carried out every time the code is run – every method is reinterpreted for every time it needs to be run. In the modern-day JVM, this would go on for a while, until some methods in your program or chunks of code are marked as "*hotspots*" – meaning the JVM has run these portions of the code many, many times.

At this point, the JVM will then execute a *Just-In-Time* (JIT) compilation of those hotspot portions. The thing is interpretation of the java bytecode slows down the execution; JIT compilation produces durable assembly instructions that can be executed directly by the CPU. This means that there will be no need for the repeat interpretation. As you can imagine, that speeds up those specific parts of the application. The JIT-compiled parts (and only those parts) of the application will become faster to execute than the interpreted parts.

Figure 1-1 illustrates the process.

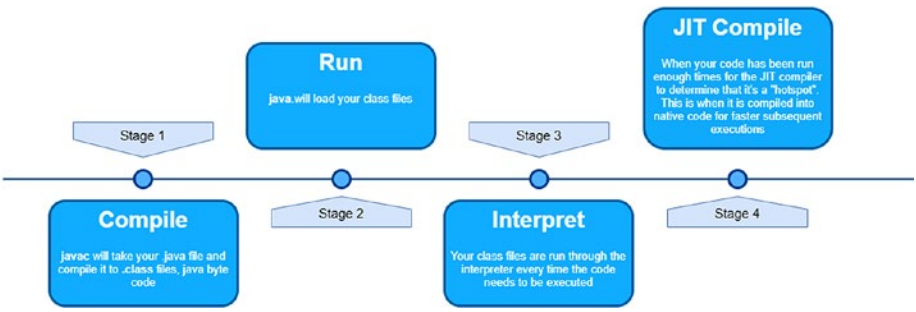


***Figure 1-1.*** *Traditional Hotspot compilation*

## Ahead-of-Time Compilation to the Rescue!

Ahead-of-time (AOT) compilation takes compilation further, or rather brings it nearer. AOT compilation takes your `.java` files straight to compiled native binaries that can be immediately executed by the CPU, skipping the relentless interpretation step and passing the savings on to you! Your application starts up significantly faster, and most of the code enjoys the benefit of near immediate computability by the CPU. Additionally, the memory usage drastically shrinks. The performance gains from this process are comparable to what the likes of C++ can boast of. Your entire application, if AOT-compiled, can even become a self-contained executable, without the need for a running, OS-supplied JVM.

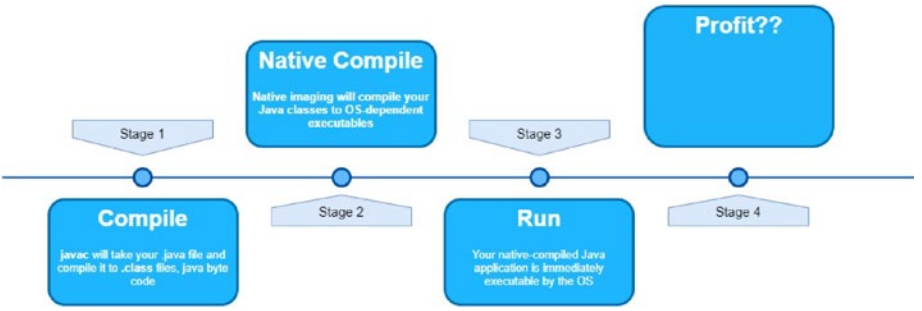


***Figure 1-2.*** *Ahead-of-Time compilation*

But there are a few catches, because there's no free lunch: Quarkus and AOT compilation weave their performance sorcery by stripping the Java runtime to the essentials. Code that's compiled using AOT contains only exactly what that code needs from the JDK, trimming a lot of fat. This upfront compilation step means that it will take a little bit longer than most Java devs are used to. Other time-consuming operations like using the Reflection API are diminished somewhat. For example, if you plan to use reflection, you're going to have to configure your build to prepare to use

specific classes reflectively. It's counterintuitive I know, but in practice, it's a minor inconvenience at worst. At best you're guaranteeing the behavior of your application at runtime! *WORP WORP, baby!*

There are some minor sacrifices that are made at the altar of performance that we will examine later in this book. Writing WORP code means deploying predictably. It means knowing what JVM features your application might need ahead of deployment (TLS, Reflection, Injection, etc.). At the end of it all, you're still getting a lot of bang for your buck.

# Quarkus Feature Tour

Any good microservice framework must provide a minimum set of features, like running your application without a stand-alone server and opinionated configurations with sensible defaults. Now I'm not going to say that Quarkus is going to make you smarter, wealthier, or more attractive,[2] but I'm also not going to say it won't do all those things. But what *else* can it do?

## Native Image Support

As I've mentioned before, a key feature of the Quarkus framework is the ability to generate *native images* from Java code. The native image that's generated skips the interpretation stage of running regular java code, helping it start faster and consume less system resources.[3] The process of generating a native image using AOT strips several layers of "fat" from the JRE and Java code, allowing the finished image to operate with significantly

[2]Editor's note: It will not.

[3]The performance gains manifest immediately during startup of the microservice app. There are some special conditions where you may not get the most performance benefits out of the framework, but on the whole, Quarkus is much faster and lighterweight than most of the competition.

less resources than a traditional Java application. This isn't the only mode of Quarkus mind; you can run your Quarkus app as a traditional Java application (so-called JVM mode) without any problems – and still get significant performance boosts. It's just that now, any talk of "Java is too slow to do `${someUseCase}`" or "Java is not suitable for embedded deployment" is no longer valid. Cheers to that!

## Serverless and Container-Friendly

For the uninitiated, serverless deployment is an application deployment environment available only in the cloud. It's a deployment model that's offered by cloud providers where you don't have to deal with the application server onto which your microservice will be deployed. All you'll need to do as a customer of a serverless-providing vendor is to supply your deployment package – a WAR or in the case of Quarkus, a JAR.

Kubernetes (K8s for short) deployment is a first love for Quarkus – it was designed with K8s in mind, from the container orchestration perspective.[4] With the support for native compilation using GraalVM, Quarkus yields

- Dramatically smaller deployment units
- Much lower memory demands
- Quick startup times

These are factors that you should care about if you're operating in a containerized or serverless environment. You want your dockerized application to start fast and utilize as little RAM as reasonably possible. Why? So that your K8s, Elastic Container Service, or other container management service can quickly scale out your microservice in response

[4]Kubernetes is a container management platform that allows you to scale and manage the deployment of a containerized application.

to load. In a serverless scenario, you *really* want your application to start up as quickly as possible; a delay in startup could prove expensive: some cloud providers charge by the amount of time for which a serverless application runs. The native compilation doesn't apply to just your code; many third-party libraries and frameworks that you're used to (Kafka, AWS libraries, etc.) have been engineered using Quarkus' extension API to make them native compilable. This means you can get container-friendly levels of performance out of things like JDBC operations and dependency injection. Even without native compilation, Quarkus as a framework does a lot of upfront optimization to the deployment artifacts that improves startup time. Quarkus ships with in-built support for Amazon Web Services, Azure, and OpenShift.

## Hot Reload of Live Code

Developer productivity is another focus of the Quarkus framework. The hot reloading capability in Quarkus allows developers to see their changes to code reflected live. So, when you crack open your favorite IDE (that's right, you get this feature regardless of IDE), and run the project, you don't need to shut down a server or kill the application to see further changes to your code. Simply save the change in the IDE and keep testing the code – no need to restart anything. Even config files! It's pretty awesome to add new dependencies to your Maven POM.xml in a running project and have the new libraries pulled down, all without restarting the app!

## Robust Framework Support

Quarkus supports a lot of frameworks out of the box. It also provides a robust extension framework that allows you to add support for your favorite third-party libraries and frameworks. If you've worked with any of

- JavaEE
- MicroProfile

- Apache Camel
- And yes, Spring Framework[5]

you can use all those frameworks inside your Quarkus-based code. As I'll cover in a little bit, Quarkus also covers a lot of the standards we've grown accustomed to: JAX-RS, JAX-B, JSON-B, and so on. It's built to enable fresh microservice development, as well as migrating existing microservices into a Quarkus project. Now as at the time of this writing, Quarkus is still a pretty young platform, so the support for some frameworks is still in preview mode, so your mileage may vary.

## Developer-Friendly Tooling

Quarkus provides a rich option set for working with and within the framework. There are feature-rich plugins for both IntelliJ and Microsoft's Visual Studio Code for a GUI-led bootstrapping of a project. There's also the `https://code.quarkus.io/` project starter page, like you get with Spring Boot.

Once you've gotten the project going, there's a healthy ecosystem of extensions that cover most use cases in the microservice world. The *quarkus* Maven plugin gives you handy access to all the functionality you'll need to manage your Quarkus project; my favorite function gives you handy access to plugins just like *Homebrew* (for macOS) and *Node Package Manager* (for Node.js). We see these tools and plugins in action shortly.

[5]In preview mode at the time of this writing.

## Reactive SQL

Now this one, I got excited when I first read about it. Many facets of Java standard and enterprise programming have gotten the reactive treatment: RESTful service endpoints, core Java,[6] and so on. With Quarkus, database programming is getting the reactive treatment also! Reactive programming as a programming style provides a responsive, flow-driven, and message-oriented approach to handling data. It's designed for high-throughput, robust error handling and a fluent programming style; and it's a very welcome addition to SQL. What does that buy you?

- Being able to operate on database query results as a streaming flow of data, instead of having to iterate over the results one by one
- Processing results of a query in an asynchronous, event-driven manner
- A publish-subscribe relationship between your business logic and the database
- All within a scalable, CPU-efficient, and responsive framework

That's the promise of reactive SQL with Quarkus. As at the time of this writing, only MySQL, DB2 and PostgreSQL support are available in reactive mode in Quarkus.

[6]JDK 9 introduced core support for reactive programming with the Flow API, so framework providers can now base their reactive implementations on core java.

## Cloud-Native and Microservices-Ready

As anyone who's had to decompose a monolithic application into microservices can attest, it's not a walk in the park. When your architecture is built with the assumption that everything your application will ever need is in a single deployment unit, you're going to find some peculiar challenges breaking it down into microservices. Then double that trouble for pushing the application into the cloud. Quarkus is loaded with extensions that make the transition to microservices a breeze. All of Quarkus' features are in support of a full application living in the highly distributed and disconnected world of the cloud:

- Foundationally, almost everything in Quarkus is reactive for efficient CPU usage and flow control.
- With OpenTracing, MicroProfile Metrics, and Health Checks, you will have eyes and ears over everything your application is doing, especially when a single business process spans multiple independent components "up there," in the sky.
- Your application doesn't have to spontaneously combust every time a black box dependency isn't available for whatever reason: fault tolerance is supported, also via MicroProfile.

## JVM Language Support: Scala and Kotlin

Now I'm neither a career Scala programmer nor a Kotlin one, and even I think this is awesome: you can use Quarkus in your Scala and Kotlin projects – and a handful of other JVM-compatible languages! Pretty *hip and with it*, as the kids say.[7]

[7]Editor's note: "The kids" haven't said this in over 3 decades. Please stop this.

# Getting Started with Quarkus

Red Hat lets you *have it your way* – there are a few options for starting off with a brand new Quarkus project. I'll cover the usual suspects.

## Java

Quarkus deprecated JDK 8 support with version 1.4.1 (this book is based on v1.6). The Quarkus team plans to drop support for JDK 8 altogether version 1.6 of Quarkus. It's JDK 11 from there on out.

## IDEs

There are plugins for IntelliJ and Microsoft Visual Studio Code; for IntelliJ, go to **File ➤ Settings ➤ Plugins** and search for "Quarkus" and follow the instructions from there. Similarly, for VS Code hit **File ➤ Preferences ➤ Settings** and search for "Quarkus". IntelliJ offers the trademark intuitive interface for starting a project from the Quarkus plugin; my experience with the Visual Studio Code plugin was not as intuitive. Nothing on the Eclipse or NetBeans front as at the time of this writing. Boooo!

## Maven

With Apache Maven, you can bootstrap a project from the command line with

```
mvn io.quarkus:quarkus-maven-plugin:1.6.0.Final:create
-DprojectGroupId=com.apress.samples
-DprojectArtifactId=code-with-quarkus
-DprojectVersion=1.0.0-SNAPSHOT
-DclassName=org.acme.ExampleResource
-Dpath=/hello
```

This command uses the `quarkus-maven-plugin`, version **1.6.0.Final** from the `io.quarkus` group. On that plugin, I'm using the `create` goal, passing in additional properties like `className` to generate a REST resource class and `path` to set the path on the REST resource class. The command generates a project named "code-with-quarkus".

The resulting kit is a completely functional application – you can run basic maven commands on it immediately:

```
mvn clean install
```

## Quarkus Plugin

The quarkus maven plugin is not your average framework plugin. It's loaded with way more functionality than one would expect. Apart from using it to generate, test, and package a project, you also get to

- Run your project in hot-reload mode with the *dev* goal. In this mode, changes you make to your project's code will be reflected without needing to shut down and restart the application. From within the welcome-to-quarkus directory, run

  ```
  mvn quarkus:dev
  ```

  This starts your Quarkus project in development mode:

  ```
  [io.quarkus] (main) code-with-quarkus 1.0.0-SNAPSHOT
  (running on Quarkus 1.6.0.Final) started in 2.012s.
  Listening on: http://0.0.0.0:8080
       [io.quarkus] (main) Profile dev activated. Live
       Coding activated.
  [io.quarkus] (main) Installed features: [cdi, resteasy]
  ```

Starting Quarkus in dev mode shows you the extensions that are running as the final startup line.

- List the available extensions[8] within the Quarkus ecosystem with the `list-extensions` goal

  ```
  mvn quarkus:list-extensions
  ```

  This produces a two-column list of all the available extensions you can spice up your application with:

  ```
  Current Quarkus extensions available:
  Quarkus - Core                          quarkus-core
  JAXB                                    quarkus-jaxb
  Jackson                                 quarkus-jackson
  JSON-B                                  quarkus-jsonb
  ...
  To get more information, append
  -Dquarkus.extension.format=full to your command line.

  Add an extension to your project by adding the
  dependency to your pom.xml or use `./mvnw
  quarkus:add-extension -Dextensions="artifactId"`
  ```

- Add new extensions to your project with the `add-extension` and `add-extensions` goals

  ```
  mvn quarkus:add-extension
  -Dextension=quarkus-spring-web
  ```

[8]It's important to note that the Quarkus-supplied dependencies you add are not your usual maven third-party libs – they've been engineered to fit into the Quarkus platform, many of them re-engineered to make them compatible for native compilation. They're not your average add-ons.

```
[INFO] --- quarkus-maven-plugin:1.6.0.Final:add-
extension (default-cli) @ code-with-quarkus ---
? Adding extension io.quarkus:quarkus-spring-web
[INFO] ------------------------------------------
[INFO] BUILD SUCCESS
```

The preceding command adds the Spring Web extension to your project, without you having to go fish out the maven coordinates yourself. The `add-extension` and `add-extensions` goals will update the POM.xml file with the Maven details of any extensions you add without your intervention.

- See a list of available options and other useful information with help

```
mvn quarkus:help

This plugin has 12 goals:
quarkus:add-extension
    Allow adding an extension to an existing pom.xml
    file. Because you can add one or several extension
    in one go, there are two mojos: add-extensions
    and add-extension. Both support the extension and
    extensions parameters.

  quarkus:analyze-call-tree
          Analyze call tree of a method or a
          class based on an existing report
          produced by Substrate when using
          -H:+PrintAnalysisCallTree, and does a more
          meaningful analysis of what is causing a
          type to be retained...
```

## Starter Website

For the ultimate in convenience, head on over to code.quarkus.io; point and click your way through to set the multiple attributes about the eventual Maven or Gradle[9] project that it will generate. There, you can also select all the extensions you want to include with the project. The starter page generates a package containing all the basic project files.

## Quarkus Maven Project Kit

Whichever approach you take to bootstrap, your generated project should contain at least the following:

- The standard Java project directory structure: `src/main/java`, src/main/test, `src/main/resources`, `pom.xml`, `README.md`
- Docker artifacts: `src/main/docker/Dockerfile.jvm`, `src/main/docker/Dockerfile.native`
- Property file for holding application configuration properties: `src/main/resources/application.properties`
- An informational page that you can just delete if you don't want it in your setup: `src/main/META-INF.resources/index.html`

---

**Tip** Use the `quarkus:generate-config` maven goal to generate an `application.properties` file that contains all available Quarkus framework properties, all disabled. With this, you can have a look at what's available to be configured.

---

[9]Gradle support is in preview mode.