Red Hat build of Quarkus 1.7

Apache Maven を使用した Quarkus アプリケーションの開発およびコンパイル

Last Updated: 2021-04-28

Red Hat build of Quarkus 1.7 Apache Maven を使用した Quarkus アプリケーションの開発およびコンパイル

法律上の通知

概要

本ガイドでは、Apache Maven フレームワークを使用して Quarkus アプリケーションを作成する方法を説明します。

目次

前書き

アプリケーション開発者は、Red Hat ビルドの Quarkus を使用して、OpenShift 環境およびサーバーレス環境で実行される Java で書かれたマイクロサービスベースのアプリケーションを作成できます。ネイティブ実行可能ファイルにコンパイルされたアプリケーションは、メモリーのフットプリントが小さく、起動時間は高速です。

本ガイドでは、Apache Maven プラグインを使用して Quarkus プロジェクトを作成する方法を説明します。

前提条件

- OpenJDK (JDK) 11 がインストールされ、**JAVA_HOME** 環境変数が Java SDK の場所を指定していること。
  - Red Hat ビルドの Open JDK は、Red Hat カスタマーポータルの Software Downloads ページから入手可能です (ログインが必要です)。
- Apache Maven 3.6.3 以降がインストールされていること。
  - Maven は Apache Maven Project の Web サイトからダウンロードできます。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社 の CTO、Chris Wright のメッセージ を参照してください。

第1章 RED HAT ビルドの QUARKUS

Red Hat ビルドの Quarkus は、コンテナーおよび Red Hat OpenShift Container Platform と使用するために最適化された Kubernetes ネイティブ Java スタックです。Quarkus は、Eclipse MicroProfile、Apache Kafka、RESTEasy (JAX-RS)、Hibernate ORM (JPA)、Spring、Infinispan、Apache Camel などの一般的な Java 標準、フレームワーク、およびライブラリーと連携するように設計されています。

Quarkus のディペンデンシーインジェクション (依存性の注入) ソリューションは、CDI (コンテキストとディペンデンシーインジェクション) をベースとし、エクステンションフレームワークを備えているので、機能の拡張、およびフレームワークの設定、起動、アプリケーションへの統合が可能です。

Quarkus は、コンテナーファーストという手法で Java アプリケーションをビルドします。この手法により、Java で書かれたマイクロサービスベースのアプリケーションのビルドが大幅に容易になるほか、これらのアプリケーションがサーバーレスコンピューティングフレームワークで実行している関数を呼び出すことができるようになります。これにより、Quarkus アプリケーションのメモリーフットプリントは小さくなり、起動時間は高速化されます。

第2章 APACHE MAVEN および QUARKUS

Apache Maven は、ソフトウェアプロジェクトの作成、管理、ビルドを行う Java アプリケーションの開発で使用される分散型ビルド自動化ツールです。Maven は Project Object Model (POM) ファイルと呼ばれる標準の設定ファイルを使用して、プロジェクトの定義やビルドプロセスの管理を行います。POM ファイルは、モジュールおよびコンポーネントの依存関係、ビルドの順番、結果となるプロジェクトパッケージングのターゲットを定義し、XML ファイルを使用して出力します。この結果、プロジェクトが適切かつ統一された状態でビルドされるようになります。

Maven リポジトリー

Maven リポジトリーには、Java ライブラリー、プラグイン、その他のビルドアーティファクトが格納されています。デフォルトのパブリックリポジトリーは Maven 2 Central Repository ですが、複数の開発チームの間で共通のアーティファクトを共有する目的で、社内のプライベートおよび内部リポジトリーを使用することができます。また、サードパーティーのリポジトリーも利用できます。

Quarkus プロジェクトでオンライン Maven リポジトリーを使用するか、または Red Hat ビルドの Quarkus Maven リポジトリーをダウンロードすることができます。

Maven プラグイン

Maven プラグインは、1 つ以上のゴールを達成する POM ファイルの定義された部分です。Quarkus アプリケーションは以下の Maven プラグインを使用します。

- Quarkus Maven プラグイン (**quarkus-maven-plugin**): Maven による Quarkus プロジェクトの作成を実現、uber-JAR ファイルの生成をサポート、そして開発モードを提供します。
- Maven Surefire プラグイン (**maven-surefire-plugin**): ビルドライフサイクルのテストフェーズで使用され、アプリケーションでユニットテストを実行します。プラグインは、テストレポートが含まれるテキストファイルと XML ファイルを生成します。

2.1. オンラインリポジトリーの MAVEN の SETTINGS.XML ファイルの設定

ユーザーの **settings.xml** ファイルを設定して、オンライン Quarkus リポジトリーを Quarkus Maven プロジェクトで使用することができます。これは推奨される手法です。リポジトリーマネージャーまたは共有サーバー上のリポジトリーと使用する Maven 設定は、プロジェクトの制御および管理を行いやすくします。

注記

Maven の **settings.xml** ファイルを変更してリポジトリーを設定する場合、変更はすべての Maven プロジェクトに適用されます。

手順

1. テキストエディターまたは統合開発環境 (IDE) で、Maven の **~/.m2/settings.xml** ファイルを開きます。

注記

~/.m2/ ディレクトリーに **settings.xml** ファイルがない場合は、**$MAVEN_HOME/.m2/conf/** ディレクトリーの **settings.xml** ファイルを **~/.m2/** ディレクトリーにコピーします。

2. 以下の行を **settings.xml** ファイルの **<profiles>** 要素に追加します。

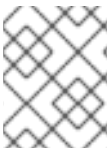
```
<!-- Configure the Quarkus Maven repository -->
<profile>
  <id>red-hat-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>red-hat-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>red-hat-enterprise-maven-repository</id>
      <url>https://maven.repository.redhat.com/ga/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

3. 以下の行を **settings.xml** ファイルの **<activeProfiles>** 要素に追加して、ファイルを保存します。

```
<activeProfile>red-hat-enterprise-maven-repository</activeProfile>
```

2.2. QUARKUS MAVEN リポジトリーのダウンロードおよび設定

オンライン Maven リポジトリーを使用しない場合は、Quarkus Maven リポジトリーをダウンロードして設定し、Maven を使用して Quarkus アプリケーションを作成できます。Quarkus Maven リポジトリーには、Java 開発者がアプリケーションのビルドに使用する必要があるものが数多く含まれています。この手順では、**settings.xml** ファイルを編集して Quarkus Maven リポジトリーを設定する方法を説明します。



注記

Maven の **settings.xml** ファイルを変更してリポジトリーを設定する場合、変更はすべての Maven プロジェクトに適用されます。

手順

1. Red Hat カスタマーポータルの Software Downloads ページから、Quarkus Maven リポジトリーの ZIP ファイルをダウンロードします (ログインが必要です)。

2. ダウンロードしたアーカイブを展開します。

3. ~/**.m2**/ ディレクトリーに移動し、テキストエディターまたは統合開発環境 (IDE) で Maven の **settings.xml** ファイルを開きます。

4. ダウンロードした Quarkus Maven リポジトリーのパスを、**settings.xml** ファイルの **<profiles>** 要素に追加します。Quarkus Maven リポジトリーのパスの形式は、**file://$PATH** である必要があります (例: **file:///home/userX/rh-quarkus-1.7.6.GA-maven-repository/maven-repository**)。

```
<!-- Configure the Quarkus Maven repository -->
<profile>
  <id>red-hat-enterprise-maven-repository</id>
  <repositories>
    <repository>
      <id>red-hat-enterprise-maven-repository</id>
      <url>file:///path/to/Quarkus/Maven/repository/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>red-hat-enterprise-maven-repository</id>
      <url>file:///path/to/Quarkus/Maven/repository/</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </pluginRepository>
  </pluginRepositories>
</profile>
```

5. 以下の行を **settings.xml** ファイルの **<activeProfiles>** 要素に追加して、ファイルを保存します。

```
<activeProfile>red-hat-enterprise-maven-repository</activeProfile>
```

重要

Maven リポジトリーに古いアーティファクトが含まれる場合は、プロジェクトをビルドまたはデプロイしたときに以下のいずれかの Maven エラーメッセージが表示されることがあります。ここで、**<artifact_name>** は不明なアーティファクトの名前、**<project_name>** はビルドを試みているプロジェクトの名前になります。

- **Missing artifact <project_name>**
- **[ERROR] Failed to execute goal on project <artifact_name>; Could not resolve dependencies for <project_name>**

この問題を解決するには、~/**.m2/repository** ディレクトリーにあるローカルリポジトリーのキャッシュバージョンを削除し、最新の Maven アーティファクトを強制的にダウンロードします。

第3章 コマンドラインでの QUARKUS プロジェクトの作成

コマンドラインで Quarkus Maven プラグインを使用し、コマンドラインで属性および値を指定するか、インタラクティブモードでプラグインを使用して、Quarkus プロジェクトを作成できます。作成されるプロジェクトには以下の要素が含まれます。

- Maven の構造
- 関連するユニットテスト
- アプリケーションの起動後に **http://localhost:8080** でアクセス可能なランディングページ
- **src/main/docker** における JVM およびネイティブモード用の **Dockerfile** サンプルファイル
- アプリケーション設定ファイル

手順

1. コマンドターミナルで以下のコマンドを入力し、Maven が JDK 11 を使用していること、そして Maven のバージョンが 3.6.3 以上であることを確認します。

```
mvn --version
```

2. 上記のコマンドで JDK 11 が返されない場合は、JDK 11 へのパスを PATH 環境変数に追加し、上記のコマンドを再度入力します。
3. Quarkus Maven プラグインを使用して新規プロジェクトを作成するには、以下のいずれかの方法を使用します。
   - 以下のコマンドを入力します。

```
mvn io.quarkus:quarkus-maven-plugin:1.7.6.Final-redhat-00014:create \
    -DprojectGroupId=<project_group_id> \
    -DprojectArtifactId=<project_artifact_id> \
    -DplatformGroupId=com.redhat.quarkus \
    -DplatformArtifactId=quarkus-universe-bom \
    -DplatformVersion=1.7.6.Final-redhat-00014 \
    -DclassName="<classname>"
```

     このコマンドで、以下の値を置き換えます。

     - **<project_group_id>**: プロジェクトの一意の識別子。
     - **<project_artifact_id>**: プロジェクトおよびプロジェクトディレクトリーの名前。
     - **<classname>**: 生成されたリソースの完全修飾名 (例: **org.acme.quarkus.sample.HelloResource**)。

   - インタラクティブモードでプロジェクトを作成します。

```
mvn io.quarkus:quarkus-maven-plugin:1.7.6.Final-redhat-00014:create
```

     プロンプトが表示されたら、必要な属性値を入力します。

注記

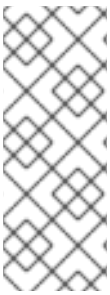
以下のコマンドを入力し、プロジェクト属性のデフォルト値を使用してプロジェクトを作成することもできます。

mvn io.quarkus:quarkus-maven-plugin:1.7.6.Final-redhat-00014:create -B

以下の表は、**create** コマンドで定義できる属性を一覧表示しています。

属性	デフォルト値	説明
projectGroupId	**org.acme.sample**	プロジェクトの一意識別子。
projectArtifactId	なし	プロジェクトおよびプロジェクトディレクトリーの名前。**projectArtifactId** を指定しないと、Maven プラグインはインタラクティブモードを起動します。ディレクトリーがすでに存在する場合、生成は失敗します。
projectVersion	**1.0-SNAPSHOT**	プロジェクトのバージョン。
platformGroupId	**io.quarkus**	プラットフォームのグループ ID。既存のプラットフォームのデフォルト値はすべて、**io.quarkus** です。ただし、デフォルト値は変更することができます。
platformArtifactId	**quarkus-universe-bom**	プラットフォーム BOM のアーティファクト ID。ローカルにビルドされた Quarkus を使用するには、**quarkus-universe-bom** を **pom.xml** ファイルに追加します。
platformVersion	プラットフォームの最新バージョン	プロジェクトに使用するプラットフォームのバージョン。バージョン範囲を指定でき、Maven プラグインは最新バージョンを使用します。

属性	デフォルト値	説明
className	なし	生成されたリソースの完全修飾名。アプリケーションの作成後、REST エンドポイントは以下の URL で公開されます。 **http://localhost:8080/$path** デフォルトの **path** を使用する場合、URL は **http://localhost:8080/hello** になります。
path	**/hello**	リソースパス (**className** を設定した場合のみ)。
extensions	[]	プロジェクトに追加するエクステンションのコンマ区切りリスト。



注記

デフォルトでは、Quarkus Maven プラグインは最新の **quarkus-universe-bom** ファイルを使用します。この BOM は、アプリケーションからエクステンションを参照し、依存関係バージョンを合わせるためにエクステンションを集約します。オフラインの場合、Quarkus Maven プラグインは、ローカルで利用可能な **quarkus-universe-bom** の最新バージョンを使用します。Maven が **quarkus-universe-bom** バージョン 2.0 以前を見つけると、**quarkus-universe-bom** をベースにしたプラットフォームを使用します。

第4章 POM.XML ファイルの設定による QUARKUS プロジェクトの作成

Maven POM XML ファイルを設定して Quarkus プロジェクトを作成できます。

手順

1. テキストエディターで **pom.xml** ファイルを開きます。
2. Quarkus GAV (Group、Artifact、Version) を追加し、**quarkus-universe-bom** ファイルを使用して、Quarkus の異なる依存関係のバージョンを省略します。

```
<dependencyManagement>
  <dependencies>
   <dependency>
    <groupId>${quarkus.platform.group-id}</groupId>
    <artifactId>${quarkus.platform.artifact-id}</artifactId>
    <version>${quarkus-plugin.version}</version>
    <type>pom</type>
    <scope>import</scope>
   </dependency>
  </dependencies>
 </dependencyManagement>
```

3. Quarkus Maven プラグインを追加します。

```
<build>
  <plugins>
    <plugin>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-maven-plugin</artifactId>
      <version>${quarkus-plugin.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>build</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

4. 任意手順: ネイティブアプリケーションをビルドするには、Maven Surefire および Maven Failsafe プラグインを含む特定のネイティブプロファイルを追加し、**native** パッケージタイプを有効化します。

```
<profiles>
  <profile>
    <id>native</id>
    <properties>
      <quarkus.package.type>native</quarkus.package.type>
    </properties>
    <build>
```

```
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>${surefire-plugin.version}</version>
            <executions>
              <execution>
                <goals>
                  <goal>integration-test</goal>
                  <goal>verify</goal>
                </goals>
                <configuration>
                  <systemProperties>

<native.image.path>${project.build.directory}/${project.build.finalName}-
runner</native.image.path>
                  </systemProperties>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </profile>
</profiles>
```

名前に **IT** が含まれるテストは、**@NativeImageTest** アノテーションが付けられ、ネイティブ実行可能ファイルに対して実行されます。

第5章 JAVA コンパイラーの設定

デフォルトでは、Quarkus Maven プラグインはコンパイラーフラグを **maven-compiler-plugin** プラグインから **javac** コマンドに渡します。

手順

- 開発モードで使用されるコンパイラーフラグをカスタマイズするには、**configuration** セクションを **plugin** ブロックに追加し、**compilerArgs** プロパティーを設定します。**source**、**target**、**jvmArgs** を設定することもできます。たとえば、**--enable-preview** を JVM と **javac** の両方に渡すには、以下の行を追加します。

```
<plugin>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-maven-plugin</artifactId>
  <version>${quarkus-plugin.version}</version>

  <configuration>
    <source>${maven.compiler.source}</source>
    <target>${maven.compiler.target}</target>
    <compilerArgs>
      <arg>--enable-preview</arg>
    </compilerArgs>
    <jvmArgs>--enable-preview</jvmArgs>
  </configuration>

  ...
</plugin>
```

第6章 QUARKUS アプリケーションを使用した JAVA エクステンションのインストールおよび管理

Java エクステンションを使用して、アプリケーションの機能を拡張し、フレームワークの設定、起動、アプリケーションへの統合が可能です。この手順では、エクステンションを検索して Quarkus プロジェクトに追加する方法を説明します。

前提条件

- Quarkus Maven プロジェクトがあること。

手順

1. Quarkus プロジェクトディレクトリーに移動します。
2. 利用可能なエクステンションを一覧表示するには、以下のコマンドを入力します。

```
./mvnw quarkus:list-extensions
```

3. プロジェクトにエクステンションを追加するには、以下のコマンドを入力します。ここで、**<extension>** は、追加するエクステンションの Group、Artifact、Version (GAV) に置き換えます。

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

たとえば、Agroal エクステンションを追加するには、以下のコマンドを入力します。

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

4. 特定のエクステンションを検索するには、**-Dextensions=** の後にエクステンション名または名前の一部を入力します。以下の例では、名前に、**jdbc**、**agroal**、および **non-exist-ent** のテキストが含まれるエクステンションを検索します。

```
./mvnw quarkus:add-extension -Dextensions=jdbc,agroal,non-exist-ent
```

このコマンドは、以下の結果を返します。

```
  Multiple extensions matching 'jdbc'
    * io.quarkus:quarkus-jdbc-h2
    * io.quarkus:quarkus-jdbc-mariadb
    * io.quarkus:quarkus-jdbc-postgresql
    Be more specific e.g using the exact name or the full gav.
  Adding extension io.quarkus:quarkus-agroal
  Cannot find a dependency matching 'non-exist-ent', maybe a typo?
[...]
```

5. 特定のテキスト文字列が返すすべてのエクステンションをインストールするには、**-Dextensions=** の後にエクステンション名または名前の一部を入力します。以下の例では、**hibernate-** で始まるすべてのエクステンションを検索し、インストールします。

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

第7章 QUARKUS プロジェクトの IDE へのインポート

テキストエディターで Quarkus プロジェクトを開発することは可能ですが、統合開発環境 (IDE) を使用した方がプロジェクトの作業がしやすいかもしれません。以下の手順では、Quarkus プロジェクトを特定の IDE にインポートする方法を説明します。

前提条件

- Quarkus Maven プロジェクトがあること。

手順

以下のいずれかのセクションの手順を実行します。

CodeReady Studio または Eclipse

1. CodeReady Studio または Eclipse で、**File → Import** とクリックします。
2. **Maven → Existing Maven Project** と選択します。
3. 次の画面で、プロジェクトのルートロケーションを選択します。見つかったモジュールの一覧が表示されます。
4. 生成されたプロジェクトを選択し、**Finish** をクリックします。
5. アプリケーションを起動するには、新しいターミナルウィンドウに以下のコマンドを入力します。

```
./mvnw compile quarkus:dev
```

IntelliJ

1. IntelliJ で、以下のタスクのいずれかを実行します。
   - **File → New → Project From Existing Sources** と選択します。
   - **Welcome** ページで **Import project** を選択します。
2. プロジェクトの root ディレクトリーを選択します。
3. **Import project from external model** を選択してから、**Maven** を選択します。
4. オプションを確認して **Next** をクリックします。
5. **Finish** をクリックします。
6. アプリケーションを起動するには、新しいターミナルウィンドウに以下のコマンドを入力します。

```
./mvnw compile quarkus:dev
```

Apache NetBeans

1. **File → Open Project** と選択します。

2. プロジェクトの **root** ディレクトリーを選択します。
3. **Open Project** クリックします。
4. アプリケーションを起動するには、新しいターミナルウィンドウに以下のコマンドを入力します。

```
./mvnw compile quarkus:dev
```

Visual Studio Code

1. Java Extension Pack をインストールします。
2. Visual Studio Code でプロジェクトディレクトリーを開きます。プロジェクトは Maven プロジェクトとしてロードされます。

第8章 QUARKUS プロジェクトの出力の設定

アプリケーションをビルドする前に、**application.properties** ファイルのアプリケーションプロパティーのデフォルト値を変更することで、ビルドコマンドの出力を制御できます。

前提条件

- Quarkus Maven プロジェクトがあること。

手順

1. テキストエディターで **application.properties** ファイルを開きます。
2. 変更するプロパティーの値を編集し、ファイルを保存します。
   以下の表は、変更可能なプロパティーをまとめたものです。

プロパティー	説明	タイプ	デフォルト
quarkus.package.main-class	アプリケーションのエントリーポイント。ほとんどの場合、この値は変更する必要があります。	string	**io.quarkus.runner.GeneratedMain**
quarkus.package.type	要求された出力タイプ。	string	**jar**
quarkus.package.uber-jar	Java ランナーを uber-JAR としてパックすべきかどうか。	boolean	**false**
quarkus.package.manifest.add-implementation-entries	実装情報をランナー JAR ファイルの **MANIFEST.MF** ファイルに含めるかどうか。	boolean	**true**
quarkus.package.user-configured-ignored-entries	出力アーティファクトにコピーしてはならないファイル。	string (list)	
quarkus.package.runner-suffix	ランナー JAR ファイルに適用される接尾辞。	string	**-runner**
quarkus.package.output-directory	アプリケーションビルドの出力フォルダー。これは、ビルドシステムのターゲットディレクトリーと相対的に解決されます。	string	
quarkus.package.output-name	最終的なアーティファクトの名前。	string	

第9章 QUARKUS アプリケーションのテスト

デフォルトでは、Quarkus アプリケーションをテストする場合、Maven は **test** 設定プロファイルを使用します。ただし、Maven Surefire プラグインを使用して、テスト用のカスタム設定プロファイルを作成することができます。

前提条件

- Apache Maven を使用して作成した Quarkus プロジェクトがあること。

手順

- テスト要件を満たすように以下の例を編集します。**<profile_name>** は、テストプロファイルの名前に置き換えます。

```
<project>
 [...]
 <build>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${surefire-plugin.version}</version>
    <configuration>
     <systemPropertyVariables>
      <quarkus.test.profile><profile_name></quarkus.test.profile>
      <buildDirectory>${project.build.directory}</buildDirectory>
      [...]
     </systemPropertyVariables>
    </configuration>
   </plugin>
  </plugins>
 </build>
 [...]
</project>
```

注記

ネイティブモードでは、カスタムのテスト設定プロファイルは使用できません。ネイティブテストは常に、**prod** プロファイルを使用して実行されます。

第10章 QUARKUS アプリケーションのビルドクラスパスツリーのロギング

Quarkus ビルドプロセスでは、アプリケーションで使用するエクステンションのデプロイメント依存関係が、元のアプリケーションクラスパスに追加されます。ビルドクラスパスに含まれる依存関係およびバージョンを確認することができます。**quarkus-bootstrap** Maven プラグインには、アプリケーションのビルド依存関係ツリーを表示する **build-tree** ゴールが含まれます。

前提条件

- Quarkus Maven アプリケーションがあること。

手順

1. プラグイン設定を **pom.xml** ファイルに追加します。

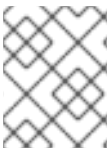
```
<project>
  [...]
    <plugin>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-bootstrap-maven-plugin</artifactId>
      <version>${quarkus-plugin.version}</version>
    </plugin>
  [...]
</project>
```

2. アプリケーションのビルド依存関係ツリーを一覧表示するには、以下のコマンドを入力します。

```
./mvnw quarkus-bootstrap:build-tree
```

3. このコマンドの出力は、以下のようになります。

```
[INFO] --- quarkus-bootstrap-maven-plugin:1.7:build-tree (default-cli) @ getting-started ---
[INFO] org.acme:getting-started:jar:1.0-SNAPSHOT
[INFO] └─ io.quarkus:quarkus-resteasy-deployment:jar:1.7 (compile)
[INFO]    ├─ io.quarkus:quarkus-resteasy-server-common-deployment:jar:1.7 (compile)
[INFO]    │  ├─ io.quarkus:quarkus-core-deployment:jar:1.7 (compile)
[INFO]    │  │  ├─ commons-beanutils:commons-beanutils:jar:1.9.3 (compile)
[INFO]    │  │  │  ├─ commons-logging:commons-logging:jar:1.2 (compile)
[INFO]    │  │  │  └─ commons-collections:commons-collections:jar:3.2.2 (compile)
...
```



注記

mvn dependency:tree コマンドは、アプリケーションのランタイム依存関係のみを表示します。

第11章 ネイティブ実行可能ファイルの作成

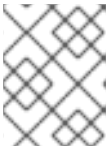
Podman または Docker などのコンテナーランタイムを使用して、Quarkus アプリケーションからネイティブ実行可能ファイルを作成することができます。Quarkus は、ビルダーイメージを使用してバイナリー実行可能ファイルを作成します。これは、Red Hat Universal Base Images RHEL8-UBI および RHEL8-UBI minimal と共に使用することができます。Red Hat ビルドの Quarkus 1.7 は、**quarkus.native.builder-image** プロパティーのデフォルトとして **registry.access.redhat.com/quarkus/mandrel-20-rhel8:20.3** を使用します。

お使いのアプリケーションのネイティブ実行可能ファイルには、アプリケーションコード、必須ライブラリー、Java API、および仮想マシン (VM) の縮小版が含まれます。縮小された仮想マシンベースは、アプリケーションの起動時間を高速化し、ディスクのフットプリントを小さくします。

手順

1. Getting Started プロジェクトの**pom.xml** ファイルを開き、**native** プロファイルが含まれていることを確認します。

```
<profiles>
  <profile>
    <id>native</id>
    <properties>
      <quarkus.package.type>native</quarkus.package.type>
    </properties>
  </profile>
</profiles>
```



注記

Quarkus **native** プロファイルを使用すると、ネイティブ実行可能ファイルおよびネイティブイメージテストの両方を実行することができます。

2. 以下のいずれかの方法を使用して、ネイティブ実行可能ファイルをビルドします。

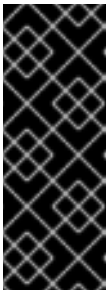
   a. Docker を使用してネイティブ実行可能ファイルをビルドします。

```
./mvnw package -Pnative -Dquarkus.native.container-build=true
```

   b. Podman を使用してネイティブ実行可能ファイルをビルドします。

```
./mvnw package -Pnative -Dquarkus.native.container-build=true -
Dquarkus.native.container-runtime=podman
```

   これらのコマンドは、**target** ディレクトリーに **getting-started-*-runner** バイナリーを作成します。



重要

Quarkus アプリケーションをネイティブ実行可能ファイルにコンパイルすると、分析および最適化の際にメモリーを大量に消費します。**quarkus.native.native-image-xmx** 設定プロパティーを設定することで、ネイティブコンパイル時に使用されるメモリーの量を制限することができます。メモリー制限を低く設定すると、ビルド時間が長くなる可能性があります。

3. ネイティブ実行可能ファイルを実行します。

```
./target/getting-started-*-runner
```

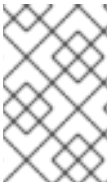
ネイティブ実行可能ファイルをビルドする場合、**prod** プロファイルが有効化され、Quarkus ネイティブテストは、**prod** プロファイルを使用して実行されます。これは、**quarkus.test.native-image-profile** プロパティーを使用して変更することができます。

11.1. 手動でのコンテナーの作成

本セクションでは、Linux X86_64 向けにアプリケーションを使用してコンテナーイメージを手動で作成する方法を説明します。Quarkus Native コンテナーを使用してネイティブイメージを作成する場合、Linux X86_64 オペレーティングシステムをターゲットとする実行可能ファイルを作成します。お使いのホストオペレーティングシステムが別のものである場合は、バイナリーを直接実行することはできないので、コンテナーを手動で作成する必要があります。

Quarkus Getting Started プロジェクトには、以下の内容と共に **src/main/docker** ディレクトリーに **Dockerfile.native** が含まれます。

```
FROM registry.access.redhat.com/ubi8/ubi-minimal
WORKDIR /work/
COPY target/*-runner /work/application
RUN chmod 775 /work
EXPOSE 8080
CMD ["./application", "-Dquarkus.http.host=0.0.0.0"]
```



UNIVERSAL BASE IMAGE (UBI)

Dockerfiles は、ベースイメージとして UBI を使用します。このベースイメージは、コンテナーで機能するように設計されています。**Dockerfiles** は、ベースイメージの minimal バージョン を使用して、作成されたイメージのサイズを縮小します。

手順

1. 以下のいずれかの方法を使用して、ネイティブ Linux 実行可能ファイルをビルドします。

a. Docker を使用してネイティブ実行可能ファイルをビルドします。

```
./mvnw package -Pnative -Dquarkus.native.container-build=true
```

b. Podman を使用してネイティブ実行可能ファイルをビルドします。

```
./mvnw package -Pnative -Dquarkus.native.container-build=true -
Dquarkus.native.container-runtime=podman
```

2. 以下のいずれかの方法を使用して、コンテナーイメージをビルドします。
   a. Docker を使用してコンテナーイメージをビルドします。

```
docker build -f src/main/docker/Dockerfile.native -t quarkus-quickstart/getting-started .
```

   b. Podman を使用してコンテナーイメージをビルドします。

```
podman build -f src/main/docker/Dockerfile.native -t quarkus-quickstart/getting-started .
```

3. コンテナーを実行します。
   a. Docker を使用してコンテナーを実行します。

```
docker run -i --rm -p 8080:8080 quarkus-quickstart/getting-started
```

   b. Podman を使用してコンテナーを実行します。

```
podman run -i --rm -p 8080:8080 quarkus-quickstart/getting-started
```

Red Hat OpenShift Container Platform での Quarkus Maven アプリケーションのデプロイに関する詳細は、**『Red Hat OpenShift Container Platform での Quarkus アプリケーションのデプロイ』**を参照してください。

第12章 ネイティブ実行可能ファイルのテスト

ネイティブ実行可能ファイルの機能をテストするために、ネイティブモードで実行するアプリケーションをテストします。**@NativeImageTest** アノテーションを使用して、ネイティブ実行可能ファイルをビルドし、http エンドポイントに対してテストを実行します。

手順

1. **pom.xml** ファイルを開き、**native** プロファイルに以下の要素が含まれていることを確認します。

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>${surefire-plugin.version}</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
      <configuration>
        <systemPropertyVariables>
          <native.image.path>${project.build.directory}/${project.build.finalName}-
runner</native.image.path>

<java.util.logging.manager>org.jboss.logmanager.LogManager</java.util.logging.manager>
          <maven.home>${maven.home}</maven.home>
        </systemPropertyVariables>
      </configuration>
    </execution>
  </executions>
</plugin>
```

failsafe-maven-plugin はインテグレーションテストを実行し、作成されたネイティブ実行可能ファイルの場所を示します。

2. **src/test/java/org/acme/quickstart/NativeGreetingResourceIT.java** ファイルを開き、以下の内容が含まれていることを確認します。

```
package org.acme.quickstart;


import io.quarkus.test.junit.NativeImageTest;

@NativeImageTest (1)
public class NativeGreetingResourceIT extends GreetingResourceTest { (2)

    // Run the same tests

}
```

(1) テストの前に、ネイティブファイルからアプリケーションを開始する別のテストランナーを使用します。実行可能ファイルは、**Failsafe Maven Plugin** に設定された **native.image.path** システムプロパティーを使用して取得されます。

2 この例は、**GreetingResourceTest** を拡張しますが、新しいテストを作成することも可能です。

3. テストを実行します。

```
./mvnw verify -Pnative
```

以下の例は、このコマンドの出力を示しています。

```
./mvnw verify -Pnative
...
[getting-started-1.0-SNAPSHOT-runner:18820]     universe:     587.26 ms
[getting-started-1.0-SNAPSHOT-runner:18820]      (parse):   2,247.59 ms
[getting-started-1.0-SNAPSHOT-runner:18820]     (inline):   1,985.70 ms
[getting-started-1.0-SNAPSHOT-runner:18820]    (compile):  14,922.77 ms
[getting-started-1.0-SNAPSHOT-runner:18820]      compile:  20,361.28 ms
[getting-started-1.0-SNAPSHOT-runner:18820]        image:   2,228.30 ms
[getting-started-1.0-SNAPSHOT-runner:18820]        write:     364.35 ms
[getting-started-1.0-SNAPSHOT-runner:18820]      [total]:  52,777.76 ms
[INFO]
[INFO] --- maven-failsafe-plugin:2.22.1:integration-test (default) @ getting-started ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running org.acme.quickstart.NativeGreetingResourceIT
Executing [/data/home/gsmet/git/quarkus-quickstarts/getting-started/target/getting-started-
1.0-SNAPSHOT-runner, -Dquarkus.http.port=8081, -Dtest.url=http://localhost:8081, -
Dquarkus.log.file.path=build/quarkus.log]
2019-04-15 11:33:20,348 INFO  [io.quarkus] (main) Quarkus 999-SNAPSHOT started in
0.002s. Listening on: http://[::]:8081
2019-04-15 11:33:20,348 INFO  [io.quarkus] (main) Installed features: [cdi, resteasy]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.387 s - in
org.acme.quickstart.NativeGreetingResourceIT
...
```

注記

Quarkus は、ネイティブイメージの開始まで 60 秒間待機し、その後ネイティブテストに自動的に失敗します。この待機時間は、**quarkus.test.native-image-wait-time** システムプロパティーを使用して変更可能です。

以下のコマンドを使用して、待機時間を延長することができます。**<duration>** は、秒単位の待機時間になります。

```
./mvnw verify -Pnative -Dquarkus.test.native-image-wait-time=<duration>
```

第13章 QUARKUS 開発モードの使用

開発モードはバックグラウンドコンパイルによるホットデプロイメントを可能にします。つまり、Java ファイルまたはリソースファイルを変更してブラウザーを更新すると、変更が自動的に反映されます。これは、設定プロパティーファイルなどのリソースファイルでも同じく反映されます。

前提条件

- Quarkus Maven アプリケーションがあること。

手順

1. 開発モードで Quarkus を起動するには、Quarkus アプリケーションの **pom.xml** ファイルが含まれるディレクトリーで以下のコマンドを入力します。

```
./mvnw quarkus:dev
```

2. アプリケーションに変更を加え、ファイルを保存します。

3. ブラウザーを更新して、ワークスペースのスキャンをトリガーします。
変更が検出されると、Java ファイルが再コンパイルされ、アプリケーションが再デプロイされます。その後、要求は再デプロイされたアプリケーションによって処理されます。コンパイルまたはデプロイメントに問題がある場合には、エラーページが表示されます。

   開発モードでは、デバッガーがアクティベートされ、ポート **5005** をリッスンします。

4. 任意手順: アプリケーションの実行前にデバッガーが割り当てられるのを待つには、**-Dsuspend** を追加します。

```
./mvnw quarkus:dev -Dsuspend
```

5. 任意手順: デバッガーが実行されないようにするには、**-Ddebug=false** を追加します。

```
./mvnw quarkus:dev -Ddebug=false
```

第14章 QUARKUS プロジェクトのデバッグ

Quarkus が開発モードで起動すると、デバッグはデフォルトで有効になります。デバッガーは、JVM を一時停止せずにポート **5005** でリッスンします。

前提条件

- Quarkus Maven プロジェクトがあること。

手順

デバッグを制御するには、以下のいずれかの方法を使用します。

システムプロパティーを使用したデバッガーの制御

1. 以下の **debug** システムプロパティーの値の 1 つを変更します。ここで、**PORT** はデバッガーがリッスンするポートです。
   - **false**: JVM はデバッグモードを無効にして開始します。
   - **true**: JVM はデバッグモードで開始され、ポート **5005** でリッスンしています。
   - **client**: JVM はクライアントモードで起動され、**localhost:5005** への接続を試みます。
   - **PORT**: JVM はデバッグモードで開始され、**PORT** をリッスンしています。
2. **suspend** システムプロパティーの値を変更します。このプロパティーは、Quarkus がデバッグモードで開始する際に使用されます。
   - **y** または **true**: デバッグモードの JVM の起動が一時停止します。
   - **n** または **false**: デバッグモードの JVM は一時停止せずに起動します。

コマンドラインからのデバッガーの制御

1. デバッグモードの JVM で Quarkus アプリケーションを起動するには、以下のコマンドを入力します。

```
./mvnw compile quarkus:dev -Ddebug
```

2. **localhost:5005** にデバッガーを割り当てます。

第15章 その他のリソース

- 『Red Hat OpenShift Container Platform での Quarkus アプリケーションのデプロイ』
- 『Quarkus アプリケーションのネイティブ実行可能ファイルへのコンパイル』
- 『Quarkus アプリケーションのテスト』
- Apache Maven Project

改訂日時: 2021-04-27 03:44:04 UTC