Red Hat build of Quarkus 3.2

YAML ファイルを使用して Red Hat build of Quarkus アプリケーションを設定する

Last Updated: 2024-05-10

Red Hat build of Quarkus 3.2 YAML ファイルを使用して Red Hat build of Quarkus アプリケーションを設定する

法律上の通知

概要

このガイドでは、YAML ファイルを使用して Red Hat build of Quarkus アプリケーションを設定する方法について説明します。

目次

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、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 プロパティーファイルを使用して RED HAT BUILD OF QUARKUS アプリケーションを設定する

アプリケーション開発者は、Red Hat build of 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。

次のいずれかの方法を使用して、Quarkus アプリケーションを設定できます。

- **application.properties** ファイルのプロパティーを設定する
- **application.yaml** ファイルを更新することで、YAML フォーマットで構造化設定を適用する

次の手順を実行して、アプリケーションの設定を拡張およびカスタマイズすることもできます。

- プロパティー式を使用して、設定プロパティー値の置き換えまたは設定を行います。
- さまざまな外部ソースから設定値を読み取るカスタム設定ソースコンバーターを使用して、MicroProfile 準拠のクラスを実装します。
- 設定プロファイルを使用して、開発環境、テスト環境、実稼働環境用に個別の設定値のセットを維持します。

この手順には、Quarkus **config-quickstart** 演習を使用して作成された設定例が含まれています。

前提条件

- OpenJDK 11 または 17 をインストールし、**JAVA_HOME** 環境変数を設定して Java SDK の場所を指定している。
  - Red Hat build of OpenJDK をダウンロードするには、Red Hat カスタマーポータルにログインし、ソフトウェアダウンロード に移動します。
- Apache Maven 3.8.6 以降がインストールされている。
  - Maven は Apache Maven Project の Web サイトからダウンロードできます。
- Maven を、Quarkus Maven repository のアーティファクトを使用するように設定している。
  - Maven の設定方法は、Quarkus スタートガイド を参照してください。

1.1. RED HAT 設定オプション

設定オプションを使用すると、1 つの設定ファイルでアプリケーションの設定を変更できます。Red Hat build of Quarkus は、関連するプロパティーをグループ化し、必要に応じてプロファイルを切り替えるために使用できる設定プロファイルをサポートしています。

デフォルトでは、Quarkus は **src/main/resources** ディレクトリーにある **application.properties** ファイルからプロパティーを読み取ります。代わりに、YAML ファイルからプロパティーを読み取るように Quarkus を設定することもできます。

quarkus-config-yaml 依存関係をプロジェクトの **pom.xml** ファイルに追加すると、**application.yaml** ファイルでアプリケーションのプロパティーを設定および管理できます。詳細は、YAML 設定サポートを追加する を参照してください。

Red Hat build of Quarkus は MicroProfile Config もサポートしているため、アプリケーションの設定を他のソースからロードすることもできます。

Eclipse MicroProfile プロジェクトの MicroProfile Config 仕様を使用して、設定プロパティーをアプリケーションに注入し、コードに定義されたメソッドを使用してアクセスできます。

Quarkus は、次のソースを含むさまざまなソースからアプリケーションプロパティーを読み取ることもできます。

- ファイルシステム
- データベース
- Kubernetes または OpenShift Container Platform の **ConfigMap** またはシークレットオブジェクト
- Java アプリケーションでロードできる任意のソース

1.2. 設定クイックスタートプロジェクトを作成する

config-quickstart プロジェクトを使用すると、Apache Maven と Quarkus Maven プラグインを使用して、シンプルな Quarkus アプリケーションを起動し、実行できます。次の手順では、Quarkus Maven プロジェクトの作成方法を説明します。

前提条件

- OpenJDK 11 または 17 をインストールし、**JAVA_HOME** 環境変数を設定して Java SDK の場所を指定している。
  - Red Hat build of OpenJDK をダウンロードするには、Red Hat カスタマーポータルにログインし、ソフトウェアダウンロード に移動します。
- Apache Maven 3.8.6 以降がインストールされている。
  - Maven は Apache Maven Project の Web サイトからダウンロードできます。

手順

1. Maven が OpenJDK 11 または 17 を使用していること、および Maven のバージョンが 3.8.6 以降であることを確認します。

```
mvn --version
```

2. **mvn** コマンドが OpenJDK 11 または 17 を返さない場合は、システム上で OpenJDK 11 または 17 がインストールされているディレクトリーが **PATH** 環境変数に含まれていることを確認します。

```
export PATH=$PATH:<path_to_JDK>
```

3. プロジェクトを生成するには、以下のコマンドを入力します。

```
mvn com.redhat.quarkus.platform:quarkus-maven-plugin:3.2.12.Final-redhat-00002:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=config-quickstart \
    -DplatformGroupId=com.redhat.quarkus.platform \
```

```
    -DplatformVersion=3.2.12.Final-redhat-00002 \
    -DclassName="org.acme.config.GreetingResource" \
    -Dpath="/greeting"
cd config-quickstart
```

検証

前述の **mvn** コマンドにより、**config-quickstart** ディレクトリーに次の項目が作成されます。

- Maven プロジェクトディレクトリー構造
- An**org.acme.config.GreetingResource** リソース
- 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
- ネイティブモードおよび JVM モードでアプリケーションをテストするための関連するユニットテスト
- **src/main/docker** サブディレクトリー内の **Dockerfile.jvm** ファイルおよび **Dockerfile.native** ファイルの例
- アプリケーションの設定ファイル

注記

このチュートリアルで使用する Quarkus Maven プロジェクトを Quarkus quickstart archive からダウンロードしたり、**Quarkus Quickstarts** Git リポジトリーをクローンしたりすることもできます。Quarkus の **config-quickstart** の演習は **config-quickstart** ディレクトリーにあります。

1.3. RED HAT BUILD OF QUARKUS アプリケーションに設定値を注入する

Red Hat build of Quarkus は、MicroProfile Config 機能を使用して設定データをアプリケーションに注入します。コンテキストと依存性注入 (CDI) を使用するか、コード内でメソッドを定義することで、設定にアクセスできます。

@ConfigProperty アノテーションを使用して、オブジェクトプロパティーをアプリケーションの **MicroProfile Config Sources** ファイルのキーにマップします。

次の手順と例は、Red Hat build of Quarkus アプリケーション設定ファイル **application.properties** を使用して、Quarkus **config-quickstart** プロジェクトに個別のプロパティー設定を注入する方法を示しています。

注記

MicroProfile Config configuration file (**src/main/resources/META-INF/microprofile-config.properties**) は、**application.properties** ファイルとまったく同じ方法で使用できます。

application.properties ファイルを使用することが推奨されます。

前提条件

config-quickstart プロジェクトを作成している。

手順

1. **src/main/resources/application.properties** ファイルを開きます。
2. 設定プロパティーを設定ファイルに追加します。**<property_name>** はプロパティー名に、**<value>** はプロパティーの値に置き換えます。

```
<property_name>=<value>
```

以下の例は、Quarkus **config-quickstart** プロジェクトで **greeting.message** プロパティーおよび **greeting.name** プロパティーの値を設定する方法を示しています。

application.properties ファイルの例

```
greeting.message = hello
greeting.name = quarkus
```

重要

アプリケーションを設定する際に、アプリケーション固有プロパティーの前に **quarkus** の文字列を付けないでください。**quarkus** の接頭辞は、フレームワークレベルで Quarkus を設定するために予約されています。**quarkus** をアプリケーション固有プロパティーの接頭辞として使用すると、アプリケーションの実行時に予期しない結果が生じる可能性があります。

3. プロジェクトの **GreetingResource.java** ファイルを確認します。このファイルには、**/greeting** エンドポイントで HTTP リクエストを送信した場合にメッセージを返す **hello()** メソッドが含まれる **GreetingResource** クラスがあります。

GreetingResource.java ファイルの例

```
import java.util.Optional;

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

import org.eclipse.microprofile.config.inject.ConfigProperty;

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

    String message;
    Optional<String> name;
    String suffix;


    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return message + " " + name.orElse("world") + suffix;
    }
}
```

ここで示した例では、**hello()** メソッドの **message** および **name** 文字列の値は初期化されていません。エンドポイントが呼び出され、この状態で正常に開始されると、アプリケーションは **NullPointerException** をスローします。

4. **message**、**name**、および **suffix** フィールドを定義し、アノテーション **@ConfigProperty** を付け、**greeting.message** および **greeting.name** プロパティーに定義した値と一致させます。アノテーション **@ConfigProperty** を使用して、各文字列の設定値を注入します。以下に例を示します。

GreetingResource.java ファイルの例

```
import java.util.Optional;

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

import org.eclipse.microprofile.config.inject.ConfigProperty;

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

  @ConfigProperty(name = "greeting.message") 1
  String message;

  @ConfigProperty(name = "greeting.suffix", defaultValue="!") 2
  String suffix;

  @ConfigProperty(name = "greeting.name")
  Optional<String> name; 3

  @GET
  @Produces(MediaType.TEXT_PLAIN)
  public String hello() {
     return message + " " + name.orElse("world") + suffix;
  }
}
```

1. **greeting.message** 文字列の値を設定しない場合、アプリケーションは失敗し、**jakarta.enterprise.inject.spi.DeploymentException: io.quarkus.runtime.configuration.ConfigurationException: Failed to load config value of type class java.lang.String for: greeting.message** の例外をスローします。
2. **greeting.suffix** の値を設定しないと、Quarkus はその値をデフォルト値に解決します。
3. **greeting.name** プロパティーを定義しない場合、**name** の値は使用できません。**name** に **Optional** パラメーターを設定しているため、この値が使用できない場合でもアプリケーションは実行されます。

注記

設定した値を注入するには、**@ConfigProperty** を使用します。**@ConfigProperty** アノテーションが付けられたメンバーの **@Inject** アノテーションを含める必要はありません。

5. 開発モードでアプリケーションをコンパイルして起動します。

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

6. エンドポイントがメッセージを返すことを確認するには、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。

```
curl http://localhost:8080/greeting
```

このコマンドは、以下の出力を返します。

```
hello quarkus!
```

7. アプリケーションを停止するには、CTRL+C を押します。

1.4. 設定変更を検証するために機能テストを更新する

アプリケーションの機能テストを実行する前に、機能テストを更新してアプリケーションのエンドポイントに加えた変更を反映する必要があります。以下の手順では、Quarkus **config-quickstart** プロジェクトで **testHelloEndpoint** メソッドを更新する方法を説明します。

手順

1. **GreetingResourceTest.java** ファイルを開きます。
2. **testHelloEndpoint** メソッドの内容を更新します。

```
package org.acme.config;

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 quarkus!")); // Modified line
    }

}
```

-

1.5. 設定プロパティーを設定する

デフォルトでは、Quarkus は **src/main/resources** ディレクトリーにある **application.properties** ファイルからプロパティーを読み取ります。ビルドプロパティーを変更した場合は、必ずアプリケーションを再パッケージ化してください。

Quarkus はビルド時にほとんどのプロパティーを設定します。エクステンションは、ターゲット環境に固有のデータベース URL、ユーザー名、パスワードなどのプロパティーを、実行時にオーバーライド可能として定義できます。

前提条件

Quarkus Maven プロジェクトがある。

手順

1. Quarkus プロジェクトをパッケージ化するには、以下のコマンドを入力します。

```
./mvnw clean package
```

2. 以下のメソッドのいずれかを使用して、設定プロパティーを設定します。

- システムプロパティーの設定
  以下のコマンドを、**<property_name>** は追加する設定プロパティーの名前に、**<value>** はプロパティーの値に置き換えて入力します。

```
java -D<property_name>=<value> -jar target/myapp-runner.jar
```

  たとえば、**quarkus.datasource.password** プロパティーの値を設定するには、以下のコマンドを入力します。

```
java -Dquarkus.datasource.password=youshallnotpass -jar target/myapp-runner.jar
```

- 環境変数の設定
  以下のコマンドを、**<property_name>** は設定する設定プロパティーの名前に、**<value>** はプロパティーの値に置き換えて入力します。

```
export <property_name>=<value> ; java -jar target/myapp-runner.jar
```

注記

環境変数名は、Eclipse MicroProfile の変換ルールに従います。名前を大文字に変換し、英数字以外の文字をすべてアンダースコア (_) に置き換えます。

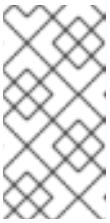
- 環境ファイルの使用
  現在の作業ディレクトリーに **.env** ファイルを作成し、**<PROPERTY_NAME>** はプロパティー名に、**<value>** はプロパティーの値に置き換えて設定プロパティーを追加します。

```
<PROPERTY_NAME>=<value>
```

注記

開発モードでは、このファイルはプロジェクトのルートディレクトリーにあります。バージョン管理でファイルは追跡しないでください。プロジェクトのルートディレクトリーに **.env** ファイルを作成する場合は、プログラムがプロパティーとして読み取るキーおよび値を定義できます。

- **application.properties** ファイルを使用します。
  アプリケーションが実行される **$PWD/config/application.properties** ディレクトリーに設定ファイルを配置し、そのファイルで定義されているランタイムプロパティーがデフォルトの設定をオーバーライドできるようにします。



注記

開発モードで **config/application.properties** 機能を使用することもできます。**config/application.properties** ファイル を **target** ディレクトリーに配置します。ビルドツールからのクリーニング操作 (**mvn clean** など) では、**config** ディレクトリーも削除されます。

1.6. 高度な設定マッピング

次に記載する高度なマッピングの手順は、Red Hat build of Quarkus 固有のエクステンションであり、MicroProfile Config 仕様の範囲には含まれません。

1.6.1. @ConfigMapping を使用してインターフェイスにアノテーションを付ける

関連する複数の設定値を個別に注入する代わりに、**@io.smallrye.config.ConfigMapping** アノテーションを使用して設定プロパティーをグループ化します。次の手順では、Quarkus **config-quickstart** プロジェクトで **@ConfigMapping** アノテーションを使用する方法を説明します。

前提条件

- **config-quickstart** プロジェクトを作成している。
- プロジェクトの **application.properties** ファイルに **greeting.message** プロパティーおよび **greeting.name** プロパティーを定義している。

手順

1. プロジェクト内の **GreetingResource.java** ファイルで、次の例に示す内容が含まれていることを確認します。**@ConfigPoperties** アノテーションを使用して、別の設定ソースからこのクラスに設定プロパティーを注入するには、**java.util.Optional** および **org.eclipse.microprofile.config.inject.ConfigProperty** パッケージをインポートする必要があります。

   GreetingResource.java ファイルの例

```
import java.util.Optional;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
```

```
import org.eclipse.microprofile.config.inject.ConfigProperty;

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

    @ConfigProperty(name = "greeting.message")
    String message;

    @ConfigProperty(name = "greeting.suffix", defaultValue="!")
    String suffix;

    @ConfigProperty(name = "greeting.name")
    Optional<String> name;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return message + " " + name.orElse("world") + suffix;
    }
}
```

2. **src/main/java/org/acme/config** ディレクトリーに **GreetingConfiguration.java** ファイルを作成します。そのファイルに、**ConfigMapping** と **Optional** のインポートステートメントを追加します。

GreetingConfiguration.java ファイルの例

```
import io.smallrye.config.ConfigMapping;
import io.smallrye.config.WithDefault;
import java.util.Optional;

@ConfigMapping(prefix = "greeting") (1)
public interface GreetingConfiguration {
    String message();

    @WithDefault("!") (2)
    String suffix();

    Optional<String> name();
}
```

(1) **prefix** プロパティーはオプションです。たとえば、このシナリオの接頭辞は **greeting** です。

(2) **greeting.suffix** が設定されていない場合、**!** はデフォルト値として使用されます。

3. 次に示すとおり、**@Inject** アノテーションを使用して **GreetingConfiguration** インスタンスを **GreetingResource** クラスに注入します。

注記

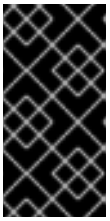
この抜粋は、**config-quickstart** プロジェクトの初期バージョンにある、**@ConfigProperty** アノテーションを持つ 3 つのフィールドを置き換えます。

GreetingResource.java ファイルの例

```
@Path("/greeting")
public class GreetingResource {

    @Inject
    GreetingConfiguration config;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return config.message() + " " + config.name().orElse("world") + config.suffix();
    }
}
```

4. 開発モードでアプリケーションをコンパイルして起動します。

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



重要

クラスプロパティーの値を指定しない場合、アプリケーションはコンパイルに失敗し、値が指定されていないことを示す **io.smallrye.config.ConfigValidationException** エラーが返されます。これは、オプションフィールドやデフォルト値のフィールドには適用されません。

5. エンドポイントがメッセージを返すことを確認するには、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。

```
curl http://localhost:8080/greeting
```

6. 以下のメッセージが表示されます。

```
hello quarkus!
```

7. アプリケーションを停止するには、CTRL+C を押します。

1.6.2. ネストされたオブジェクト設定の使用

別のインターフェイス内にネストされたインターフェイスを定義できます。この手順では、Quarkus **config-quickstart** プロジェクトでネストされたインターフェイスを作成および設定する方法を説明します。

前提条件

- **config-quickstart** プロジェクトを作成している。
- プロジェクトの **application.properties** ファイルに **greeting.message** プロパティーおよび **greeting.name** プロパティーを定義している。

手順

1. プロジェクトの **GreetingResource.java** を確認します。このファイルには、**/greeting** エンドポイントで HTTP リクエストを送信した場合にメッセージを返す **hello()** メソッドが含まれる **GreetingResource** クラスがあります。

   GreetingResource.java ファイルの例

```
import java.util.Optional;

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

import org.eclipse.microprofile.config.inject.ConfigProperty;

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

    @ConfigProperty(name = "greeting.message")
    String message;

    @ConfigProperty(name = "greeting.suffix", defaultValue="!")
    String suffix;

    @ConfigProperty(name = "greeting.name")
    Optional<String> name;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return message + " " + name.orElse("world") + suffix;
    }
}
```

2. **GreetingConfiguration** インスタンスを使用して **GreetingConfiguration.java** クラスファイルを作成します。このクラスには、**GreetingResource** クラスで定義される **hello()** メソッドの外部化設定が含まれます。

   GreetingConfiguration.java ファイルの例

```
import io.smallrye.config.ConfigMapping;
import io.smallrye.config.WithDefault;
import java.util.Optional;

@ConfigMapping(prefix = "greeting")
public interface GreetingConfiguration {
    String message();

    @WithDefault("!")
    String suffix();

    Optional<String> name();
}
```

3. 次の例のように、**GreetingConfiguration** インスタンス内にネストされた **ContentConfig** クラスを作成します。

GreetingConfiguration.java ファイルの例

```
import io.smallrye.config.ConfigMapping;
import io.smallrye.config.WithDefault;

import java.util.List;
import java.util.Optional;

@ConfigMapping(prefix = "greeting")
public interface GreetingConfiguration {
    String message();

    @WithDefault("!")
    String suffix();

    Optional<String> name();

    ContentConfig content();

    interface ContentConfig {
        Integer prizeAmount();

        List<String> recipients();
    }
}
```

注記

ContentConfig クラスのメソッド名は **content** です。プロパティーを正しいインターフェイスに確実にバインドするために、このクラスの設定プロパティーを定義する際の接頭辞に **content** を使用します。そうすることで、プロパティー名の競合やアプリケーションの予期しない動作も防げます。

4. **application.properties** ファイルで **greeting.content.prize-amount** および **greeting.content.recipients** 設定プロパティーを定義します。
次の例は、**GreetingConfiguration** インスタンスと **ContentConfig** クラスに定義されたプロパティーを示しています。

application.properties ファイルの例

```
greeting.message = hello
greeting.name = quarkus
greeting.content.prize-amount=10
greeting.content.recipients=Jane,John
```

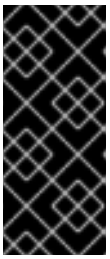
5. 次の例に示すように、3 つの **@ConfigProperty** フィールドアノテーションの代わりに、**@Inject** アノテーションを使用して **GreetingConfiguration** インスタンスを **GreetingResource** クラスに注入します。また、**/greeting** エンドポイントが返すメッセージ文字列を、追加した新しい **greeting.content.prize-amount** および **greeting.content.recipients** プロパティーに設定した値で更新する必要があります。

GreetingResource.java ファイルの例

```
import java.util.Optional;

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

import jakarta.inject.Inject;

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

    @Inject
    GreetingConfiguration config;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return config.message() + " " + config.name().orElse("world") + config.suffix() + "\n" +
config.content().recipients() + " receive total of candies: " + config.content().prizeAmount();
    }
}
```

6. 開発モードでアプリケーションをコンパイルして起動します。

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



重要

クラスプロパティーの値を指定しない場合、アプリケーションはコンパイルに失敗し、値が指定されていないことを示す **jakarta.enterprise.inject.spi.DeploymentException** 例外を受け取ります。これは、**Optional** フィールドおよびデフォルト値のフィールドには適用されません。

7. エンドポイントがメッセージを返すことを確認するには、新しいターミナルウィンドウを開いて次のコマンドを入力します。

```
curl http://localhost:8080/greeting
```

2 行の出力を含むメッセージが表示されます。次のように、1 行目にはグリーティングが表示され、2 行目には賞品の受取人および賞品の量が報告されます。

```
hello quarkus!
Jane,John receive total of candies: 10
```

8. アプリケーションを停止するには、CTRL+C を押します。

注記

@ConfigMapping アノテーションが付けられたクラスには、次の例のような Bean Validation アノテーションを付けることができます。

```
@ConfigMapping(prefix = "greeting")
public class GreetingConfiguration {

    @Size(min = 20)
    public String message;
    public String suffix = "!";
}
```

プロジェクトには **quarkus-hibernate-validator** 依存関係が含まれている必要があります。

1.7. プログラムで設定にアクセスする

コード内にメソッドを定義して、アプリケーションの設定プロパティー値を取得できます。そうすることで、設定プロパティー値を動的に検索したり、CDI Bean または Jakarta REST (旧名 JAX-RS) リソースのいずれかのクラスから設定プロパティー値を取得したりできます。

org.eclipse.microprofile.config.ConfigProvider.getConfig() メソッドを使用して設定にアクセスできます。**config** オブジェクトの **getValue()** メソッドは、設定プロパティー値を返します。

前提条件

- Quarkus Maven プロジェクトがある。

手順

- メソッドを使用して、アプリケーションコード内のクラスまたはオブジェクトの設定プロパティーの値にアクセスします。取得する値がプロジェクトの設定ソースに設定されているかどうかに応じて、以下のいずれかのメソッドを使用できます。
  - たとえば、**application.properties** ファイルで、プロジェクトの設定ソースで設定されるプロパティーの値にアクセスするには、**getValue()** メソッドを使用します。

    ```
    String <variable-name> = ConfigProvider.getConfig().getValue("<property-name>",
    <data-type-class-name>.class);
    ```

    たとえば、データ型が **String** で、コード内の **message** 変数に割り当てられている **greeting.message** プロパティーの値を取得するには、次の構文を使用します。

    ```
    String message = config.getValue("greeting.message", String.class);
    ```

  - 任意の値またはデフォルト値を取得し、**application.properties** ファイルまたはアプリケーションの別の設定ソースで定義されていない場合は、**getOptionalValue()** メソッドを使用します。

    ```
    String <variable-name> = ConfigProvider.getConfig().getOptionalValue("<property-
    name>", <data-type-class-name>.class);
    ```

たとえば、オブションで、データ型が **String** で、コード内の **name** 変数に割り当てられている、オプションの **greeting.name** プロパティーの値を取得するには、次の構文を使用します。

```
Optional<String> name = config.getOptionalValue("greeting.name", String.class);
```

次の抜粋は、プログラムによる設定へのアクセスを使用して、前述の **GreetingResource** クラスのバリアントを示します。

src/main/java/org/acme/config/GreetingResource.java

```
package org.acme.config;

import java.util.Optional;

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

import org.eclipse.microprofile.config.Config;
import org.eclipse.microprofile.config.ConfigProvider;
import org.eclipse.microprofile.config.inject.ConfigProperty;

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

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        Config config = ConfigProvider.getConfig();
        String message = config.getValue("greeting.message", String.class);
        String suffix = config.getOptionalValue("greeting.suffix", String.class).orElse("!");
        Optional<String> name = config.getOptionalValue("greeting.name", String.class);


        return message + " " + name.orElse("world") + suffix;
    }
}
```

1.8. プロパティー式

プロパティー式は、設定内のプロパティーの値を置き換えるために使用できるプロパティー参照とプレーンテキスト文字列の組み合わせになります。

変数と同様に、Quarkus でプロパティー式を使用して、設定プロパティーの値をハードコーディングする代わりに置き換えることができます。プロパティー式は、**java.util.Properties** がアプリケーションの設定ソースからプロパティーの値を読み取ると解決されます。

これは、コンパイル時に設定プロパティーが設定から読み取られる場合に、プロパティー式もコンパイル時に解決されることを意味します。設定プロパティーが実行時にオーバーライドされる場合、その値は実行時に解決されます。

複数の設定ソースを使用してプロパティー式を解決できます。これは、ある設定ソースで定義されているプロパティーの値を使用して、別の設定ソースで使用するプロパティー式を拡張できることを意味します。

式のプロパティーの値を解決できず、式のデフォルト値を設定しない場合は、アプリケーションで **NoSuchElementException** が発生します。

1.8.1. プロパティー式の使用例

Quarkus アプリケーション設定時に柔軟性を持たせるために、次の例に示すプロパティー式を使用できます。

- 設定プロパティーの値の置き換え:
  プロパティー式を使用して、設定内のプロパティー値のハードコーディングを回避できます。以下の例のように、**${<property_name>}** 構文を使用して、設定プロパティーを参照する式を作成します。

  application.properties ファイルの例

  ```
  remote.host=quarkus.io
  callable.url=https://${remote.host}/
  ```

  callable.url プロパティーの値は **https://quarkus.io/** に解決されます。

- 特定の設定プロファイルに固有のプロパティー値の設定:
  以下の例では、**%dev** 設定プロファイルとデフォルトの設定プロファイルは、異なるホストアドレスでデータソース接続 URL を使用するように設定されます。

  application.properties ファイルの例

  ```
  %dev.quarkus.datasource.jdbc.url=jdbc:mysql://localhost:3306/mydatabase?useSSL=false
  quarkus.datasource.jdbc.url=jdbc:mysql://remotehost:3306/mydatabase?useSSL=false
  ```

  データソースドライバーは、アプリケーションの起動に使用する設定プロファイルに応じて、プロファイルに設定したデータベース URL を使用します。

  カスタム **application.server** プロパティーに対して、設定プロファイルごとに異なる値を設定することで、簡単に同じ結果を得ることができます。その後、次の例に示すように、アプリケーションのデータベース接続 URL でプロパティーを参照できます。

  application.properties ファイルの例

  ```
  %dev.application.server=localhost
  application.server=remotehost

  quarkus.datasource.jdbc.url=jdbc:mysql://${application.server}:3306/mydatabase?
  useSSL=false
  ```

  application.server プロパティーは、アプリケーションの実行時に選択するプロファイルに応じて適切な値に解決されます。

- プロパティー式のデフォルト値の設定:

プロパティー式のデフォルト値を定義できます。Quarkus は、式を展開するために必要なプロパティーの値が設定ソースのいずれからも解決されない場合に、デフォルト値を使用します。次の構文を使用して、式のデフォルト値を設定できます。

```
${<property_name>:<default_value>}
```

以下の例では、データソース URL のプロパティー式は、**application.server** プロパティーのデフォルト値として **mysql.db.server** を使用します。

application.properties ファイルの例

```
quarkus.datasource.jdbc.url=jdbc:mysql://${application.server:mysql.db.server}:3306/mydatabas
e?useSSL=false
```

- プロパティー式のネスト化:
  プロパティー式を別のプロパティー式内にネスト化することで、プロパティー式を作成できます。ネスト化されたプロパティー式を展開する際には、内部の式が最初に展開されます。プロパティー式をネストするには、次の構文を使用します。

  ```
  ${<outer_property_name>${<inner_property_name>}}
  ```

- 複数のプロパティー式を組み合わせます。
  次の構文を使用して、2 つ以上のプロパティー式を結合できます。

  ```
  ${<first_property_name>}${<second_property_name>}
  ```

- プロパティー式と環境変数の組み合わせ:
  プロパティー式を使用して、環境変数の値を置き換えることができます。次に示す例の式は、**HOST** 環境変数に設定される値を **application.host** プロパティーの値として置き換えます。

  application.properties ファイルの例

  ```
  remote.host=quarkus.io
  application.host=${HOST:${remote.host}}
  ```

HOST 環境変数が設定されていない場合、**application.host** プロパティーは、**remote.host** プロパティーの値をデフォルトとして使用します。

1.9. 設定プロファイルの使用

お使いの環境に応じて、異なる設定プロファイルを使用できます。設定プロファイルを使用すると、同じファイルに複数の設定を含めることができ、プロファイル名を使用してそれらを選択できます。

Red Hat build of Quarkus には、次の 3 つのデフォルト設定プロファイルがあります。

- **dev**: 開発モードでのアクティブ化
- **test**: テストの実行時のアクティブ化
- **prod**: 開発またはテストモードで実行されていない場合のデフォルトプロファイル

注記

さらに、独自のカスタムプロファイルを作成することもできます。

前提条件

Quarkus Maven プロジェクトがある。

手順

1. Java リソースファイルを開き、以下のインポートステートメントを追加します。

```
import io.quarkus.runtime.configuration.ProfileManager;
```

2. 現在の設定プロファイルを表示するには、**ProfileManager.getActiveProfile()** メソッドを呼び出してログを追加します。

```
LOGGER.infof("The application is starting with profile `%s`",
ProfileManager.getActiveProfile());
```

注記

@ConfigProperty("quarkus.profile") メソッドを使用して現在のプロファイルにアクセスすることはできません。

1.9.1. カスタム設定プロファイルの設定

設定プロファイルは、必要なだけ作成できます。同じファイル内に複数の設定を含めることができ、プロファイル名を使用して設定を選択できます。

手順

1. カスタムプロファイルを設定するには、**application.properties** ファイルのプロファイル名で設定プロパティーを作成します。**<property_name>** はプロパティー名に、**<value>** はプロパティーの値に、そして **<profile>** はプロファイル名に置き換えます。

設定プロパティーを作成します

```
%<profile>.<property_name>=<value>
```

以下の設定例では、**quarkus.http.port** の値はデフォルトで **9090** で、**dev** プロファイルがアクティブ化されると **8181** になります。

設定例

```
quarkus.http.port=9090
%dev.quarkus.http.port=8181
```

2. プロファイルを有効にするには、以下のいずれかの方法を使用します。

- **quarkus.profile** システムプロパティーを設定します。

- **quarkus.profile** システムプロパティーを使用してプロファイルを有効にするには、以下のコマンドを入力します。

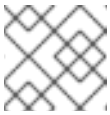
  quarkus.profile プロパティーを使用して、プロファイルを有効にします

  ```
  mvn -Dquarkus.profile=<value> quarkus:dev
  ```

- **QUARKUS_PROFILE** 環境変数を設定します。
  - 環境変数を使用してプロファイルを有効にするには、以下のコマンドを入力します。

    環境変数を使用してプロファイルを有効にします

    ```
    export QUARKUS_PROFILE=<profile>
    ```

    注記

    システムプロパティーの値は環境変数の値よりも優先されます。

3. アプリケーションを再パッケージしてプロファイルを変更するには、以下のコマンドを入力します。

   プロフィールを変更します

   ```
   ./mvnw package -Dquarkus.profile=<profile>
   java -jar target/myapp-runner.jar
   ```

   以下の例は、**prod-aws** プロファイルをアクティブ化するコマンドを示しています。

   プロファイルをアクティブにするコマンドの例

   ```
   ./mvnw package -Dquarkus.profile=prod-aws
   java -jar target/myapp-runner.jar
   ```

注記

Quarkus アプリケーションのデフォルトのランタイムプロファイルは、アプリケーションのビルドに使用されるプロファイルに設定されます。Red Hat build of Quarkus は、環境モードに応じてプロファイルを自動的に選択します。たとえば、JAR としてアプリケーションを実行中の場合、Quarkus は **prod** モードになります。

1.10. カスタム設定ソースの設定

Quarkus アプリケーションは、デフォルトでプロジェクトの **src/main/resources** サブディレクトリー内の **application.properties** ファイルからプロパティーを読み取ります。Quarkus では、外部化された設定の MicroProfile Config 仕様に従い、他のソースからアプリケーション設定プロパティーをロードすることもできます。**org.eclipse.microprofile.config.spi.ConfigSource** および **org.eclipse.microprofile.config.spi.ConfigSourceProvider** インターフェイスを実装するクラスを定義することで、アプリケーションは他のソースから設定プロパティーをロードできるようになります。以下の手順では、Quarkus プロジェクトにカスタム設定ソースを実装する方法を説明します。

前提条件

Quarkus **config-quickstart** プロジェクトがある。

手順

1. **org.eclipse.microprofile.config.spi.ConfigSourceProvider** インターフェイスを実装するプロジェクトでクラスファイルを作成します。**ConfigSource** オブジェクトのリストを返すには、**getConfigSources()** メソッドをオーバーライドする必要があります。

   org.acme.config.InMemoryConfigSourceProvider の例

```
package org.acme.config;

import org.eclipse.microprofile.config.spi.ConfigSource;
import org.eclipse.microprofile.config.spi.ConfigSourceProvider;

import java.util.List;

public class InMemoryConfigSourceProvider implements ConfigSourceProvider {

    @Override
    public Iterable<ConfigSource> getConfigSources(ClassLoader classLoader) {
    return List.of(new InMemoryConfigSource());
    }
}
```

2. **org.eclipse.microprofile.config.spi.ConfigSource** インターフェイスを実装する **InMemoryConfigSource** クラスを作成します。

   org.acme.config.InMemoryConfigSource の例

```
package org.acme.config;

import org.eclipse.microprofile.config.spi.ConfigSource;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class InMemoryConfigSource implements ConfigSource {
private static final Map<String, String> configuration = new HashMap<>();


  static {
     configuration.put("my.prop", "1234");
  }

   @Override
   public int getOrdinal() { 1
     return 275;
   }

   @Override
   public Set<String> getPropertyNames() {
     return configuration.keySet();
   }
```

```
    @Override
    public String getValue(final String propertyName) {
      return configuration.get(propertyName);
    }

    @Override
    public String getName() {
      return InMemoryConfigSource.class.getSimpleName();
    }
  }
```

1. **getOrdinal()** メソッドは、**ConfigSource** クラスの優先度を返します。そのため、複数の設定ソースが同じプロパティーを定義している場合でも、Quarkus は最も優先度の高い**ConfigSource** クラスで定義されている適切な値を選択できます。

3. プロジェクトの **src/main/resources/META-INF/services/** サブディレクトリーに **org.eclipse.microprofile.config.spi.ConfigSourceProvider** という名前のファイルを作成し、作成したファイルに **ConfigSourceProvider** を実装するクラスの完全修飾名を入力します。

   org.eclipse.microprofile.config.spi.ConfigSourceProvider ファイルの例:

   ```
   org.acme.config.InMemoryConfigSourceProvider
   ```

   アプリケーションをコンパイルして開始する際に、作成した **ConfigSourceProvider** が確実に登録およびインストールされているようにするには、前の手順を完了する必要があります。

4. プロジェクト内の **GreetingResource.java** ファイルを編集して、次の更新を追加します。

   ```
   @ConfigProperty(name="my.prop") int value;
   ```

5. **GreetingResource.java** ファイルで、**hello** メソッドを展開して新しいプロパティーを使用します。

   ```
   @GET
   @Produces(MediaType.TEXT_PLAIN)
   public String hello() {
     return message + " " + name.orElse("world") + " " + value;
   }
   ```

6. 以下のコマンドを入力して、開発モードでアプリケーションをコンパイルし、起動します。

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

7. ターミナルウィンドウを開いて次のコマンドを入力し、**/greeting** エンドポイントが予期したメッセージを返すことを確認します。

   要求の例

   ```
   curl http://localhost:8080/greeting
   ```

8. アプリケーションがカスタム設定を正常に読み取ると、コマンドは次のレスポンスを返します。

```
hello world 1234
```

1.11. カスタム設定コンバーターの設定値としての使用

org.eclipse.microprofile.config.spi.Converter<T> を実装し、その完全修飾クラス名を **META-INF/services/org.eclipse.microprofile.config.spi.Converter** に追加することで、カスタムタイプを設定値として保存できます。コンバーターを使用すると、値の文字列表現をオブジェクトに変換できます。

前提条件

config-quickstart プロジェクトを作成している。

手順

1. **org.acme.config package** で、次の内容を含む **org.acme.config.MyCustomValue** クラスを作成します。

   カスタム設定値の例

```
package org.acme.config;

public class MyCustomValue {

  private final int value;

  public MyCustomValue(Integer value) {
    this.value = value;
  }

  public int value() {
    return value;
  }
}
```

2. コンバータークラスを実装し、convert メソッドをオーバーライドして **MyCustomValue** インスタンスを生成します。

   コンバータークラスの実装例

```
package org.acme.config;

import org.eclipse.microprofile.config.spi.Converter;

public class MyCustomValueConverter implements Converter<MyCustomValue> {

    @Override
    public MyCustomValue convert(String value) {
        return new MyCustomValue(Integer.valueOf(value));
    }
}
```

3. 次の例に示すように、**META-INF/services/org.eclipse.microprofile.config.spi.Converter** サービスファイルにコンバーターの完全修飾クラス名を含めます。

org.eclipse.microprofile.config.spi.Converter ファイルの例

```
org.acme.config.MyCustomValueConverter
org.acme.config.SomeOtherConverter
org.acme.config.YetAnotherConverter
```

4. **GreetingResource.java** ファイルに、**MyCustomValue** プロパティーを注入します。

```
@ConfigProperty(name="custom")
    MyCustomValue value;
```

5. この値を使用するように **hello** メソッドを編集します。

```
@GET
 @Produces(MediaType.TEXT_PLAIN)
 public String hello() {
     return message + " " + name.orElse("world") + " - " + value.value();
 }
```

6. **application.properties** ファイルに、変換する文字列表現を追加します。

```
custom=1234
```

7. 以下のコマンドを入力して、開発モードでアプリケーションをコンパイルし、起動します。

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

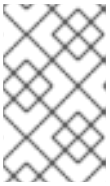
8. ターミナルウィンドウを開いて次のコマンドを入力し、/**greeting** エンドポイントが予期したメッセージを返すことを確認します。

要求の例

```
curl http://localhost:8080/greeting
```

9. アプリケーションがカスタム設定を正常に読み取ると、コマンドは次のレスポンスを返します。

```
hello world - 1234
```



注記

カスタムコンバータークラスは **public** で、引数なしのコンストラクター **public** がなければなりません。カスタムコンバータークラスは **abstract** であってはなりません。

関連情報:

- List of converters in the **microprofile-config** GitHub repository

1.11.1. カスタムコンバーターの優先度の設定

Quarkus コアコンバータの優先度は、すべてデフォルトで 200 です。他のコンバーターの優先度は、すべてデフォルトで 100 です。**jakarta.annotation.Priority** アノテーションを使用して、カスタムコンバーターの優先度を上げることができます。

次の手順は、優先度 150 のカスタムコンバーター **AnotherCustomValueConverter** の実装を示しています。これは、デフォルトの優先度 100 に設定されている前のセクションの **MyCustomValueConverter** よりも優先されます。

前提条件

- **config-quickstart** プロジェクトを作成している。
- アプリケーションのカスタム設定コンバーターを作成している。

手順

1. クラスに **@Priority** アノテーションを付け、優先度の値を渡して、カスタムコンバーターの優先度を設定します。次の例では、優先度の値は **150** に設定されています。

   AnotherCustomValueConverter.java ファイルの例

   ```
   package org.acme.config;

   import jakarta.annotation.Priority;
   import org.eclipse.microprofile.config.spi.Converter;

   @Priority(150)
   public class AnotherCustomValueConverter implements Converter<MyCustomValue> {

     @Override
     public MyCustomValue convert(String value) {
     return new MyCustomValue(Integer.valueOf(value));
     }
   }
   ```

2. プロジェクトの **src/main/resources/META-INF/services/** サブディレクトリーに **org.eclipse.microprofile.config.spi.Converter** という名前のファイルを作成し、作成したファイルに **Converter** を実装するクラスの完全修飾名をファイルに入力します。

   org.eclipse.microprofile.config.spi.Converter ファイルの例

   ```
   org.acme.config.AnotherCustomValueConverter
   ```

   アプリケーションをコンパイルして起動する際に、作成した **Converter** が確実い登録およびインストールされているようにするには、前の手順を完了する必要があります。

検証

必要な設定を完了してから、次のステップとして Quarkus アプリケーションをコンパイルしてパッケージ化します。詳細と例については、Quarkus スタートガイド でコンパイルおよびパッケージ化のセクションを参照してください。

1.12. 関連情報

- 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

- OpenShift Container Platform に Quarkus アプリケーションをデプロイする
- 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

改訂日時: 2024-05-10