

Пишем свое первое расширение на Quarkus

Иванов Роман Александрович

1 июня 2020 г.

Содержание

1	Предисловие	3
2	Что такое Quakus Extension	3
3	Запуск приложения в простом java приложении	3
4	Как создать расширение	4
5	Имплементация runtime модуля	4
6	Реализация конфигурации	5
7	Имплементация поставщика notification API	6
8	Имплементация Рекордера	7
9	Реализация deployment	8
10	Сборка и настройка зависимостей	9
11	Реализация Build Step Processors	10
12	Тестирование	12

1 Предисловие

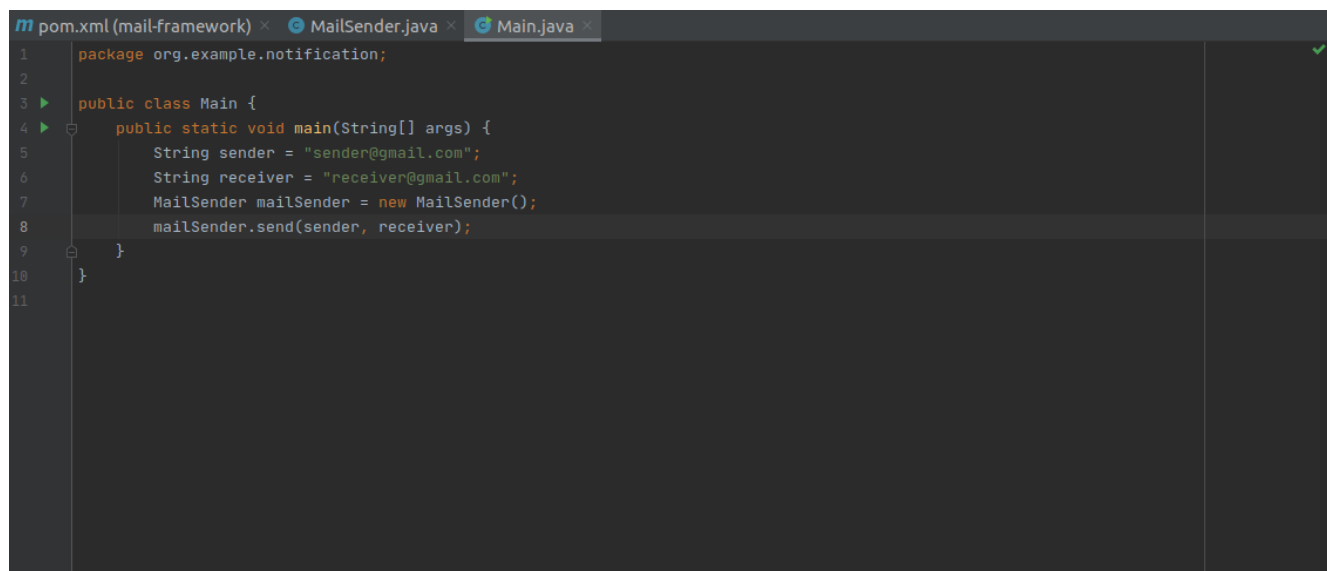
Quarkus - это фреймворк, состоящий из ядра и набора расширений. Ядро основано на внедрении Context и Dependency Injection (CDI), а расширения обычно предназначены для интеграции сторонней инфраструктуры путем предоставления их основных компонентов в виде компонентов CDI.

2 Что такое Quarkus Extension

Quarkus Extension- это просто модуль, который может работать поверх приложения Quarkus. Наиболее распространенный вариант использования такого расширения - запуск стороннего фреймворка поверх приложения Quarkus.

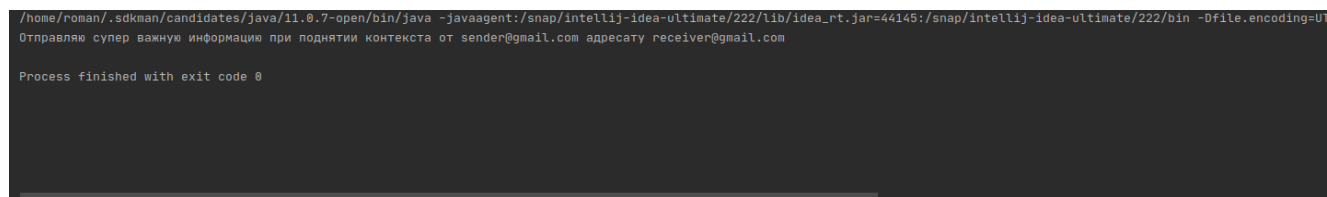
3 Запуск приложения в простом java приложении

Давайте попробуем реализовать расширение для отправки почты во время поднятия контекста приложения (будем его дальше называть notification). Но прежде чем мы углубимся, нам сначала нужно показать, как отправлять сообщения из основного метода Java. Это значительно облегчит реализацию расширения. Точкой входа для notification является API notification. Чтобы использовать это, нам нужен адрес отправителя и получателя:



```
1 package org.example.notification;
2
3 public class Main {
4     public static void main(String[] args) {
5         String sender = "sender@gmail.com";
6         String receiver = "receiver@gmail.com";
7         MailSender mailSender = new MailSender();
8         mailSender.send(sender, receiver);
9     }
10 }
11
```

При запуске можно увидеть следующее:



```
/home/roman/.sdkman/candidates/java/11.0.7-open/bin/java -javaagent:/snap/intellij-idea-ultimate/222/lib/idea_rt.jar=44145:/snap/intellij-idea-ultimate/222/bin -Dfile.encoding=UTF
Отправляю супер важную информацию при поднятии контекста от sender@gmail.com адресату receiver@gmail.com

Process finished with exit code 0
```

Цель состоит в том, чтобы выставить notification как расширение Quarkus. То есть, предоставляя конфигурацию (адрес отправителя и получателя) Quarkus Configuration, а затем создавая notification API в качестве компонента CDI. Это обеспечит средство для отправки сообщения в момент поднятия контекста приложения.

4 Как создать расширение

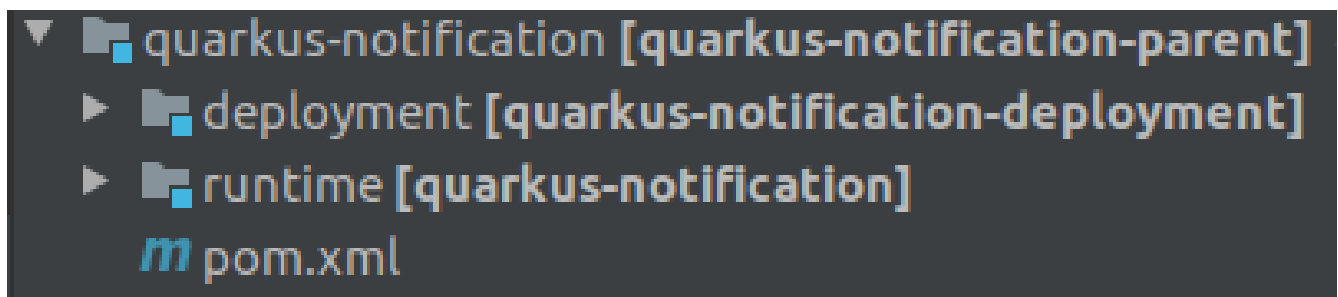
Для ознакомления можно попробовать создать расширение самому, пройдя по ссылке <https://quarkus.io/guides/building-my-first-extension>.

Для того, чтобы создать расширение, необходимо выполнить инструкцию:

```
mvn io.quarkus:quarkus-maven-plugin:1.4.2.Final:create-extension -N \
  -DgroupId=org.example \
  -DartifactId=quarkus-notification \
  -Dversion=1.0-SNAPSHOT \
  -Dquarkus.nameBase="Mail_sender_with_up_context "
```

Обратите внимание, что указанная версия 1.4.2.Final является самой актуальной на момент написания, в зависимости от версии ее нужно будет поменять.

Технически говоря, расширение Quarkus - это многомодульный проект Maven, состоящий из двух модулей. Первый - это модуль времени выполнения, в котором мы реализуем требования. Второй - это модуль развертывания для обработки конфигурации и генерации кода времени выполнения. Итак, начнем с создания многомодульного проекта Maven под названием quarkus-notification-parent, который содержит два подмодуля: время выполнения и развертывание:



5 Имплементация runtime модуля

В runtime модуле мы реализуем:

1. Конфигурационный класс, который будет хранить в себе настройки адресов отправителя и получателя)
2. Поставщика notification API (Предоставление расширением бина, который отвечает за notification API)
3. Рекордер, который будет работать как прокси объект для вызова API

Модуль runtime будет зависеть от модуля ядра Quarkus и, в конечном итоге, модулей времени выполнения необходимых расширений. Здесь нам нужна зависимость нашего импровизированного фреймворка:

```
<groupId>org.example</groupId>
<artifactId>mail-framework</artifactId>
<version>1.0-SNAPSHOT</version>
```

После добавления этой зависимости в runtime модуль он будет выглядеть следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.example</groupId>
    <artifactId>quarkus-notification-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>quarkus-notification</artifactId>
  <name>Mail sender with up context - Runtime</name>

  <dependencies>
    <dependency>
      <groupId>io.quarkus</groupId>
      <artifactId>quarkus-core</artifactId>
      <version>${quarkus.version}</version>
    </dependency>
    <dependency>
      <groupId>org.example</groupId>
      <artifactId>mail-framework</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
```

6 Реализация конфигурации

Мы аннотируем класс с помощью `@ConfigRoot`, а свойства - с помощью `@ConfigItem`. Таким образом, поля `from` и `to`, будут представлены через ключ `quarkus.mail.from` и `quarkus.mail.to` в файле `application.properties`, расположенном в пути к классам приложения `Quarkus`.

Также стоит обратить внимание на `ConfigRoot.phase`. Значение `BUILD_AND_RUN_TIME_FIXED` означает, что значения ключей конфигурации читаются во время развертывания и доступны во время выполнения.

Конфигурационный класс будет иметь следующий вид:

```

package org.example.quarkus.notification.configuration;

import io.quarkus.runtime.annotations.ConfigItem;
import io.quarkus.runtime.annotations.ConfigPhase;
import io.quarkus.runtime.annotations.ConfigRoot;

/**
 * Конфигурация для отправки
 */
@ConfigRoot(name = "mail", phase = ConfigPhase.BUILD_AND_RUN_TIME_FIXED)
public final class MailConfig {
    /**
     * Адрес отправителя
     */
    @ConfigItem
    public String from;

    /**
     * Адрес получателя
     */
    @ConfigItem
    public String to;
}

```

7 Имплементация поставщика notification API

Выше мы видели, как работать с notification через простой вызов. Теперь мы воспроизведем тот же код, но в виде компонента CDI, и для этой цели будем использовать производителя CDI:

```

package org.example.quarkus.notification.producer;

import org.example.notification.MailSender;
import org.example.quarkus.notification.configuration.MailConfig;

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

/**
 * Поставщик
 */
@ApplicationScoped
public class MailSenderProducer {

    private MailConfig mailConfig;

    @Produces
    public MailSender mailSender() {
        return new MailSender(mailConfig.from, mailConfig.to);
    }

    /**
     * Установка конфигурации
     */
    * @param mailConfig конфигурация
    */
    public void setMailConfig(MailConfig mailConfig) {
        this.mailConfig = mailConfig;
    }
}

```

Метод, помеченный аннотацией @Produces, предоставляет бин с настройками, которые будут указаны в application.properties.

8 Имплементация Рекордера

На этом шаге мы напишем класс Рекордера, который действует как прокси для записи байт-кода и настройки логики времени выполнения:

```

@Recorder
public class MailSenderRecorder {

    /**
     * Установка конфигурации
     *
     * @param mailConfig конфигурация
     * @return слушатель контейнера бинов
     */
    public BeanContainerListener setMailSenderConfig(MailConfig mailConfig) {
        return beanContainer -> {
            MailSenderProducer producer = beanContainer.instance(MailSenderProducer.class);
            producer.setMailConfig(mailConfig);
        };
    }

    /**
     * Отправка сообщения
     *
     * @param container контейнер
     * @param from      отправитель
     * @param to        получатель
     */
    public void send(BeanContainer container, String from, String to) {
        MailSender mailSender = container.instance(MailSender.class);
        mailSender.send(from, to);
    }
}

```

Класс рекордер должен содержать аннотацию @Recorder. Через него мы устанавливаем конфигурацию, а также отправляем сообщение.

Обратите внимание, что когда мы вызываем эти методы записи во время сборки, инструкции не выполняются, а записываются для последующего выполнения во время запуска.

Далее рассмотрим содержимое модуля развертывания (deployment).

9 Реализация deployment

Центральными компонентами расширения Quarkus являются процессоры Build Step. Это методы, аннотированные как @BuildStep, которые генерируют байт-код через устройства записи, и они выполняются во время сборки с помощью цели сборки модуля quarkus-maven-plugin, настроенного в приложении Quarkus.

BuildSteps потребляют элементы сборки, созданные на ранних этапах сборки, и могут также сами создавать элементы сборки для других этапов сборки.

Сгенерированный код всеми упорядоченными шагами сборки, найденными в модулях развертывания приложения, фактически является кодом времени выполнения.

10 Сборка и настройка зависимостей

Самым важным аспектом зависимостей данного модуля является то, что он должен зависеть от соответствующего модуля времени выполнения и, в конечном итоге, от модулей развертывания необходимых расширений. Это означает, что вам необходимо добавить зависимость runtime модуля в модуль deployment. На нашем примере это выглядит следующим образом:

```
notification) x MailConfig.java x MailSenderProducer.java x MailSenderRecorder.java x m pom.xml (quarkus-notification-deployment) x v
9      <version>1.0-SNAPSHOT</version>
10     <relativePath>../pom.xml</relativePath>
11  </parent>
12
13     <artifactId>quarkus-notification-deployment</artifactId>
14     <name>Mail sender with up context - Deployment</name>
15
16     <dependencies>
17       <dependency>
18         <groupId>io.quarkus</groupId>
19         <artifactId>quarkus-core-deployment</artifactId>
20         <version>${quarkus.version}</version>
21       </dependency>
22       <dependency>
23         <groupId>io.quarkus</groupId>
24         <artifactId>quarkus-arc-deployment</artifactId>
25         <version>${quarkus.version}</version>
26       </dependency>
27       <dependency>
28         <groupId>io.quarkus</groupId>
29         <artifactId>quarkus-agroal-deployment</artifactId>
30         <version>${quarkus.version}</version>
31       </dependency>
32       <dependency>
33         <groupId>org.example</groupId>
34         <artifactId>quarkus-notification</artifactId>
35         <version>1.0-SNAPSHOT</version>
36       </dependency>
37     </dependencies>
38
```

11 Реализация Build Step Processors

Как ранее упоминалось, `buildStep` это такая инструкция, которая позволяет пошагово настраивать бины и записывать их байткод через инструмент кваркус ARC.

Теперь давайте реализуем три пошаговых процессора для записи байт-кода. Процессором первого шага сборки является метод `feature()`. Он отвечает за регистрацию расширения в ядро кваркуса.

За второй шаг сборки отвечает метод `build`. Он отвечает за регистрацию необходимых `BuildItem` внутри других `BuildItem`. Такой подход позволяет гибко настраивать бины. На этом этапе метод записывает байт-код для выполнения в статическом методе `init`. Мы настраиваем это через значение `STATIC_INIT`, который указан в аннотации `@Record`.

Третий шаг сборки описывает метод `processSend`. Данный метод записывает байт код, который будет выполняться в момент выполнения. Т.е. как только приложение запустится, сразу будет вызван метод отправки сообщения.

Код процессора выглядит следующим образом:

```
import ...

class QuarkusNotificationProcessor {

    private MailConfig mailConfig;

    private static final String FEATURE = "quarkus-notification";

    @BuildStep
    FeatureBuildItem feature() { return new FeatureBuildItem(FEATURE); }

    @BuildStep
    @Record(ExecutionTime.STATIC_INIT)
    void build(MailSenderRecorder recorder,
               BuildProducer<AdditionalBeanBuildItem> additionalBeanProducer,
               BuildProducer<BeanContainerListenerBuildItem> containerListenerProducer) {

        AdditionalBeanBuildItem unremovableProducer = AdditionalBeanBuildItem.unremovableOf(MailSenderProducer.class);
        additionalBeanProducer.produce(unremovableProducer);

        containerListenerProducer.produce(
            new BeanContainerListenerBuildItem(recorder.setMailSenderConfig(mailConfig)));
    }

    @BuildStep
    @Record(ExecutionTime.RUNTIME_INIT)
    void processSend(MailSenderRecorder recorder, BeanContainerBuildItem beanContainer) {
        recorder.send(beanContainer.getValue(), mailConfig.from, mailConfig.to);
    }
}
```

12 Тестирование

Теперь можно протестировать работу расширения. Для этого подключаем ее как зависимость в наш новый проект. Для подключения нужно использовать группу и артефакт от модуля runtime.

[illegible]

После запуска мы увидим, что отправляется наше импровизированное сообщение, а также увидим наше расширение в списке подключенных