

Пишем свое первое расширение на Quarkus

Иванов Роман Александрович

2 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 на простом примере	4
4.1	Имплементация runtime модуля	4
4.2	Реализация конфигурации	5
4.3	Имплементация поставщика notification API	6
4.4	Имплементация Рекордера	6
4.5	Реализация deployment	7
4.6	Сборка и настройка зависимостей	7
4.7	Реализация Build Step Processors	8
4.8	Тестирование	10
5	Пример реализации расширения на основе SPI	10
5.1	Структура нового сервиса	11
5.2	Пишем расширения для SPI	12
5.2.1	Обзор runtime модуля	12
5.2.2	Обзор deployment модуля	14
5.2.3	Обзор процессора QuarkusNotificationProcessor	15
5.3	Подмена объектов (Object Substitution)	16
5.4	Тестирование расширения	19
5.5	Вывод	19

1 Предисловие

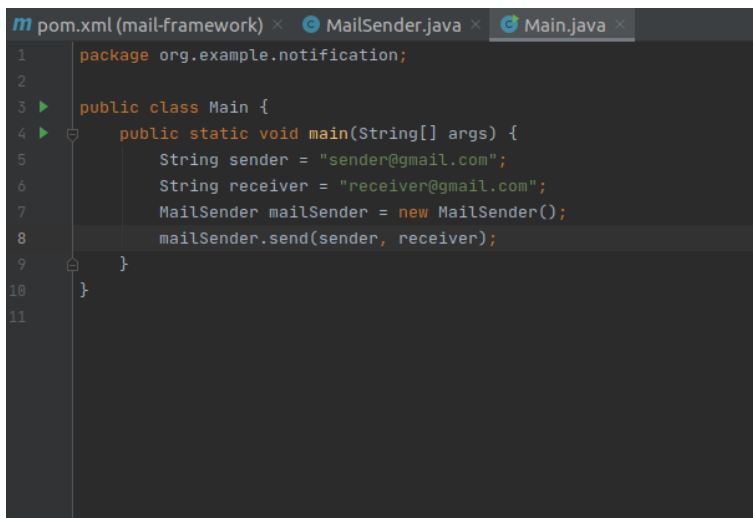
Quarkus - это фреймворк, состоящий из ядра и набора расширений. Ядро основано на внедрении Context и Dependency Injection (CDI), а расширения обычно предназначены для интеграции сторонней инфраструктуры путем предоставления их основных компонентов в виде компонентов CDI.

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.

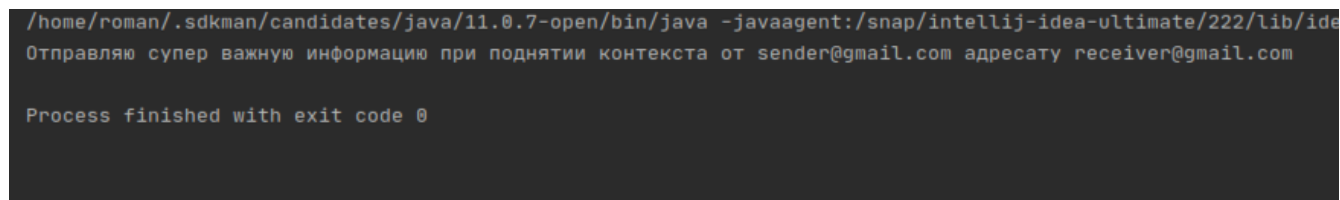
3 Запуск приложения в простом java приложении

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/ide
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 на простом примере

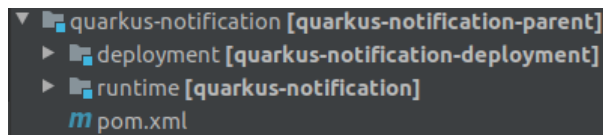
Для ознакомления можно попробовать создать расширение самому, пройдя по ссылке <https://quarkus.io/guides/building-my-first-extension>. Все исходники по данной теме можно найти в архиве **firs-extension.zip**.

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:



4.1 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 нашей импровизированной библиотеки:

```
<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 http://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>

```

4.2 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 доступен 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;
}

```

4.3 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.

4.4 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 бунов
     */
    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).

4.5 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.

4.6 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  </parent>
```

4.7 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);
    }
}
```

4.8 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]

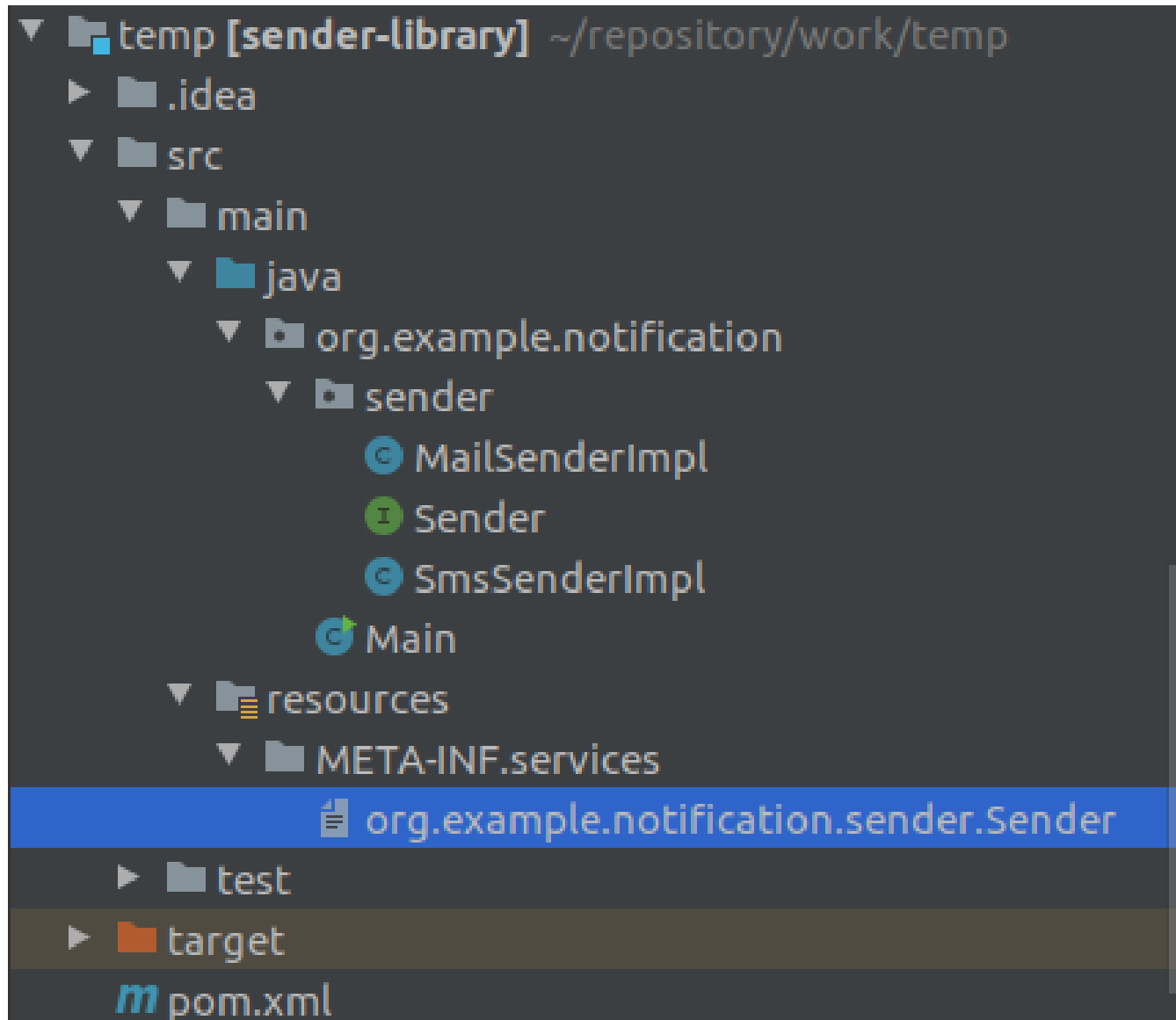
После запуска мы увидим, что отправляется наше импровизированное сообщение, а также увидим наше расширение в списке подключенных

5 Пример реализации расширения на основе SPI

Усложним прошлый пример тем, что пусть подключаемая библиотека предоставляет бины на основе SPI механизма. Что такое SPI? Service Provider Interface (SPI, Интерфейс поставщика услуг) это API, предназначенный для реализации или расширения третьей стороной. Его можно использовать для включения расширения каркаса и сменных компонентов.

5.1 Структура нового сервиса

Предположим, что вышла новая версия нашей импровизированной библиотеки. Теперь, она предоставляет бины посредством SPI и структура имеет следующий вид:

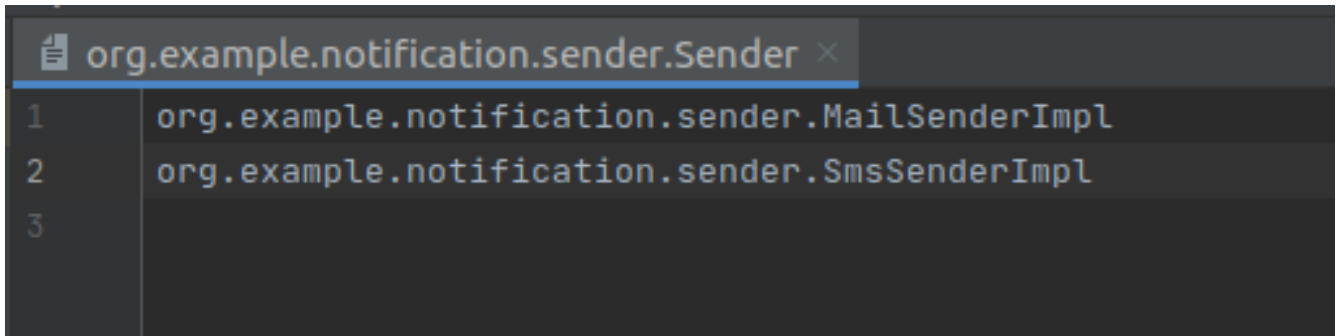


Структура в себе содержит простой интерфейс `Sender`, который предоставляет контракт по отправке сообщения, и две имплементации `EmailSenderImpl` и `SmsSenderImpl`. Первая имплементация подразумевает отправку сообщения через email, вторая через смс.

Конкретный функционал будет упущен, для демонстрации будет пример, который будет печатать в консоль эмуляцию отправки.

Разумеется, нужно взглянуть на SPI механизм. Предоставление расширений (бинов) происходит за счет регистрации конкретных реализации в файле `org.example.notification.sender.Sender`, который имеет такое название согласно своему имени класса.

Содержимое этого файла имеет следующий вид:



```
org.example.notification.sender.Sender x
1 org.example.notification.sender.MailSenderImpl
2 org.example.notification.sender.SmsSenderImpl
3
```

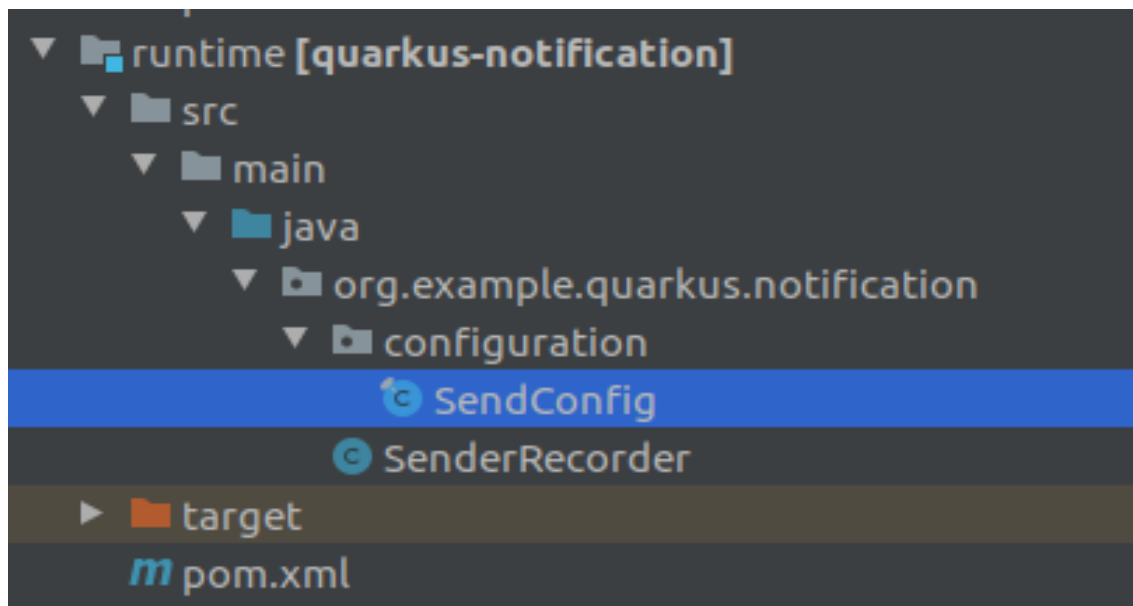
Как видно, тут зарегистрированы доступные реализации интерфейса Sender. Важно отметить, что количество реализаций может быть различным.

5.2 Пишем расширения для SPI

Расширение для этого примера, реализовано точно также, как и в пункте 4, но с новыми изменениями, которые никак не касаются структуры, направлены лишь на работу с SPI. Все изменения мы рассмотрим далее.

5.2.1 Обзор runtime модуля

Рантайм модуль состоит из двух классов. Структура кода представлена ниже:



Первый класс SendConfig. Он отвечает за хранение значений в application.properties. Второй, отвечает за верхнеуровневую работу по записи байт-кода. Код, который он может исполнять как во время статической инициализации, так и в рантайме. По сути, это форма отложенного выполнения, когда вызовы, сделанные во время разворачивания, откладываются до времени выполнения. Отсюда и название.

Просмотр SendConfig опустим. Для ознакомления вы можете посмотреть исходный код в папке sources (**firs-extention-spi.zip**).

Данный класс содержит 2 метода:

1. public List<Sender> registry(Collection<Class<? extends Sender>> implementationClasses)
2. public void sendAll(BeanContainer beanContainer, List<Sender> services, String from, String to)

Метод registry регистрирует бины, предварительно их создавая:

```
/**
 * Регистрация списка имплементаций отправщика
 *
 * @param implementationClasses классы имплементаторы
 * @return
 */
public List<Sender> registry(Collection<Class<? extends Sender>> implementationClasses) {
    // configure our service statically
    List<Sender> services = new ArrayList<>(implementationClasses.size());
    // instantiate the service implementations
    for (Class<? extends Sender> implementationClass : implementationClasses) {
        try {
            services.add(implementationClass.getConstructor().newInstance());
        } catch (Exception e) {
            throw new IllegalArgumentException("Unable to instantiate service " + implementationClass, e);
        }
    }

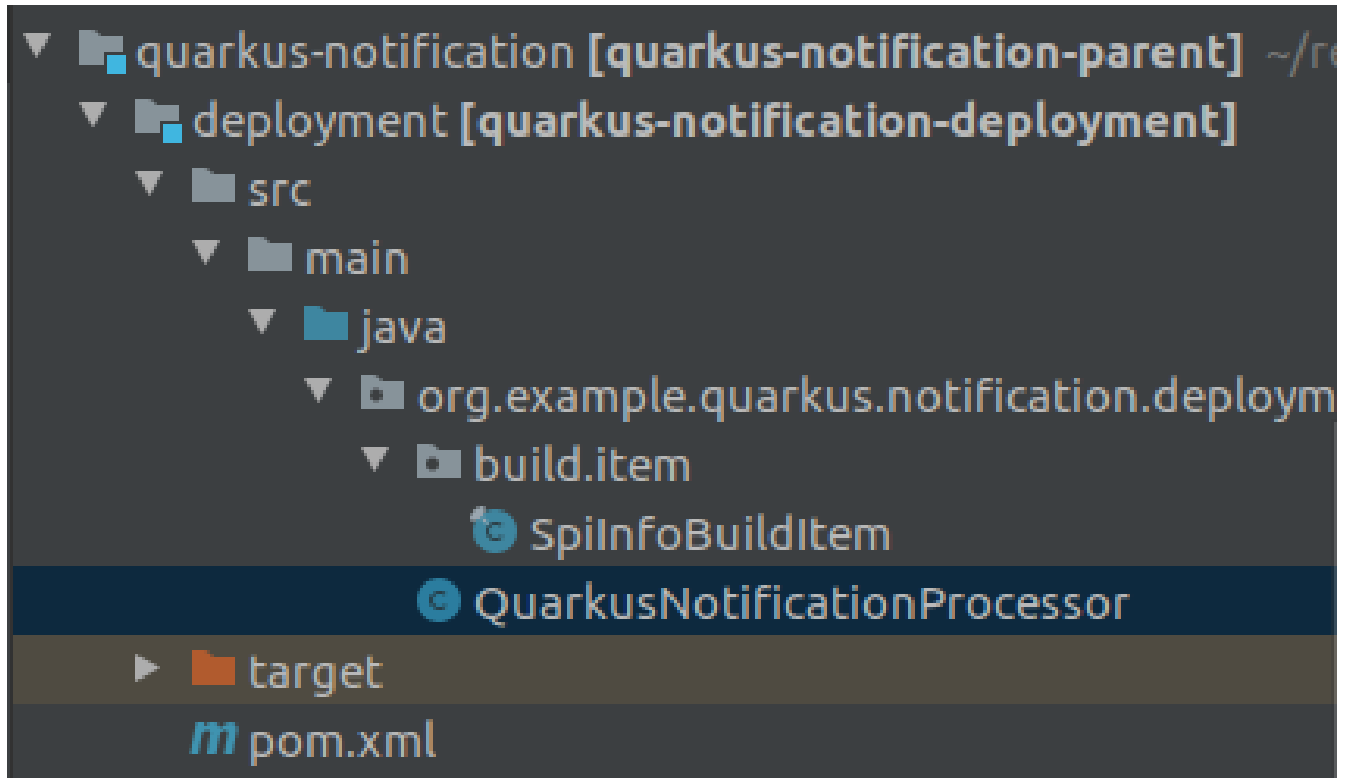
    return services;
}
```

Второй метод sendAll, берет все имплементации и для каждой вызывает отправку.

```
/**
 * Отправка всеми сервисами соответствующих сообщений
 *
 * @param beanContainer контейнер бинов
 * @param services список зарегистрированных методов
 * @param from от кого
 * @param to кому
 */
public void sendAll(BeanContainer beanContainer, List<Sender> services, String from, String to) {
    services.forEach(s -> {
        Sender instance = beanContainer.instance(s.getClass());
        instance.send(from, to);
    });
}
```

5.2.2 Обзор deployment модуля

Деплоймент модуль тоже претерпел маленьких изменений. Если для простого расширения, который практически ничего не делал не требовалось ничего, то сейчас можно увидеть, что структура немного поменялась и имеет вид:



Как видим, структура почти не поменялась, относительно предыдущего примера. SpiInfoBuildItem это новый класс, который необходимо для создания контекста при выполнении шагов сборки, которые помечены @BuildStep.

Он имеет следующий вид:

```
public final class SpiInfoBuildItem extends SimpleBuildItem {

    private final List<Sender> service;

    public SpiInfoBuildItem(List<Sender> service) { this.service = service; }

    public List<Sender> getService() { return service; }

}
```

Что такое BuildItem? Это конкретные конечные подклассы абстрактного класса, предоставляемый самим кваркусом, io.quarkus.builder.item.BuildItem. Каждый элемент сборки представляет некоторую единицу информации, которая должна быть передана с одного этапа на другой. Базовый класс

BuildItem не может быть непосредственно разделен на подклассы; скорее, существуют абстрактные подклассы для каждого из подклассов элементов сборки, которые могут быть созданы: простые, множественные и пустые.

Из содержания видно, что данная единица сборки предоставляет информацию о зарегистрированных сервисах. В данном случае он реализован для того, чтобы в момент регистрации он создавался, а в момент выполнения предоставлял зарегистрированные имплементации сервисов.

Рассмотрим процессор.

5.2.3 Обзор процессора QuarkusNotificationProcessor

Класс QuarkusNotificationProcessor содержит 3 метода, при этом являясь BuildStep-ом:

1. feature
2. registerNativeImageResources
3. processSend

Часть из них вам могут показаться уже знакомыми, т.к. за основу был взят прошлый пример.

Метод feature мы пропустим, т.к. он генерируется автоматически при создании расширения. Стоит только отметить, что он нужен для того, чтобы зарегистрировать текущее расширение в ядре кваркуса.

Всю работу по регистрации внешних сервисов, которые нам поступают через SPI берет на себя метод registerNativeImageResources. Данный метод выглядит следующим образом:

```
@BuildStep
@Record(ExecutionTime.STATIC_INIT)
void registerNativeImageResources(BuildProducer<SpiInfoBuildItem> spiInfoBuildItemBuildProducer,
                                RecorderContext recorderContext,
                                SenderRecorder recorder) throws IOException {

    String service = "META-INF/services/" + Sender.class.getName();

    // read the implementation classes
    Collection<Class<? extends Sender>> implementationClasses = new LinkedHashSet<>();
    Set<String> implementations = ServiceUtil.classNamesNamedIn(Thread.currentThread().getContextClassLoader(),
        service);
    for (String implementation : implementations) {
        implementationClasses.add((Class<? extends Sender>) recorderContext.classProxy(implementation));
    }

    // produce a static-initializer with those classes
    List<Sender> services = recorder.registry(implementationClasses);
    spiInfoBuildItemBuildProducer.produce(new SpiInfoBuildItem(services));
}
```

Если говорить коротко, то он собирает все описания из SPI файла (для простоты назовем его так), через рекордер создает для каждой имплементации соответствующий объект, а далее регистрируем наш SpiInfoBuildItem с этими сервисами. Как можно заметить, что все эти манипуляции будут происходить во время статической инициализации, т.е. в момент сборки расширения.

Метод processSend зарегистрирован в runtime фазе, т.е. как только будет запущено приложение, так сразу будет выполнен этот метод.

Код метод очень простой:

```
@BuildStep
@Record(ExecutionTime.RUNTIME_INIT)
void processSend(SenderRecorder recorder, BeanContainerBuildItem beanContainerBuildItem, SpiInfoBuildItem spiInfoBuildItem) {
    recorder.sendAll(beanContainerBuildItem.getValue(), spiInfoBuildItem.getService(), sendConfig.from, sendConfig.to);
}
```

По коду видно, что в метод приходит наш `SpiInfoBuildItem`, который мы проинициализировали на этапе деплоя, и для всех этих сервисов будет сделан вызов отправки.

После того, как вы попытаетесь запустить приложение с нашим расширением, то вы столкнетесь с тем, что оно у вас упадет с ошибкой:

```
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ code-with-quarkus ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ code-with-quarkus ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- quarkus-maven-plugin:1.4.2.Final:dev (default-cli) @ code-with-quarkus ---
Listening for transport dt_socket at address: 5005
2020-06-02 12:54:17,098 INFO [org.jboss.threads] (main) JBoss Threads version 3.1.1.Final
2020-06-02 12:54:17,970 ERROR [io.qua.run.boo.StartupActionImpl] (Quarkus Main Thread) Error running Quarkus: java.lang.reflect.InvocationTargetException
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.base/java.lang.reflect.Method.invoke(Method.java:566)
    at io.quarkus.runner.bootstrap.StartupActionImpl$1.run(StartupActionImpl.java:99)
    at java.base/java.lang.Thread.run(Thread.java:834)
Caused by: java.lang.ExceptionInInitializerError
    at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
    at java.base/jdk.internal.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
    at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:490)
    at java.base/java.lang.Class.newInstance(Class.java:584)
    at io.quarkus.runtime.Quarkus.run(Quarkus.java:60)
    at io.quarkus.runtime.Quarkus.run(Quarkus.java:38)
    at io.quarkus.runtime.Quarkus.run(Quarkus.java:106)
    at io.quarkus.runner.GeneratedMain.main(GeneratedMain.zig:29)
    ... 6 more
Caused by: java.lang.RuntimeException: Failed to start quarkus
    at io.quarkus.runner.ApplicationImpl.<clinit>(ApplicationImpl.zig:470)
    ... 15 more
Caused by: java.lang.IllegalArgumentException: Unable to instantiate service class org.example.notification.sender.SmsSenderImpl
    at org.example.quarkus.notification.SenderRecorder.registry(SenderRecorder.java:28)
    at io.quarkus.deployment.steps.QuarkusNotificationProcessor$registerNativeImageResources5.deploy_0(QuarkusNotificationProcessor$registerNativeImageResources5.zig:12
101)    at io.quarkus.deployment.steps.QuarkusNotificationProcessor$registerNativeImageResources5.deploy(QuarkusNotificationProcessor$registerNativeImageResources5.zig:12
6)    at io.quarkus.runner.ApplicationImpl.<clinit>(ApplicationImpl.zig:405)
    ... 15 more
Caused by: java.lang.NoSuchMethodException: org.example.notification.sender.SmsSenderImpl.<init>()
    at java.base/java.lang.Class.getConstructor0(Class.java:3349)
    at java.base/java.lang.Class.getConstructor(Class.java:2151)
    at org.example.quarkus.notification.SenderRecorder.registry(SenderRecorder.java:26)
    ... 18 more
```

Это сделано специально. Изначально в одной из имплементаций интерфейса `Sender` я не создал конструктор по умолчанию. Это было сделано для того, чтобы показать, что библиотеки, могут содержать код, который вам нужен, но вы не можете его переписать сами. Для таких случаев необходимо использовать такое понятие, как "Подмена объектов" или `Object Substitution`.

5.3 Подмена объектов (Object Substitution)

Объекты, созданные на этапе сборки и переданные во время выполнения, должны иметь конструктор по умолчанию, чтобы их можно было создавать и настраивать при запуске среды выполнения из состояния времени сборки. Если у объекта нет конструктора по умолчанию, вы увидите ошибку.

Существует интерфейс `io.quarkus.runtime.ObjectSubstitution`, который можно реализовать, чтобы сообщить Quarkus, как обрабатывать такие классы.

Для этого необходимо создать в runtime модуле класс, который будет имплементировать данный интерфейс. В нашем случае пример такого класса можно видеть ниже:

```
package org.example.quarkus.notification.subst;

import ...

/**
 * Объект заменитель
 */
public class SmsSenderImplObjectSubstitution implements ObjectSubstitution<SmsSenderImpl, SmsSenderImplObjectSubstitution.SmsSenderSubstitutionProxy> {

    @Override
    public SmsSenderSubstitutionProxy serialize(SmsSenderImpl obj) {
        SmsSenderSubstitutionProxy smsSenderSubstitutionProxy = new SmsSenderSubstitutionProxy();
        smsSenderSubstitutionProxy.setFrom(obj.getFrom());
        smsSenderSubstitutionProxy.setTo(obj.getTo());
        return smsSenderSubstitutionProxy;
    }

    @Override
    public SmsSenderImpl deserialize(SmsSenderSubstitutionProxy obj) {
        return new SmsSenderImpl(obj.getFrom(), obj.getTo());
    }

    /** Реализация отправки СМС */
    public static class SmsSenderSubstitutionProxy extends SmsSenderImpl {...}
}
```

Интерфейс ObjectSubstitution принимает два типа. Первый, тип (SmsSenderImpl) нашего исходного класса, который в нашем случае не имеет конструктора по умолчанию. Второй, (SmsSenderImplObjectSubstitution.SmsSenderSubstitutionProxy) - это класс заменитель, на который мы будем менять. Этот класс зарегистрирован прямо внутри SmsSenderImplObjectSubstitution - это не обязательное условие, вы также можете его создать отдельно.

Как видно, интерфейс ObjectSubstitution предоставляет два метода. Метод сериализации и десериализации. Именно эти методы отвечают за конвертацию одного объекта в другой без особых усилий со стороны разработчика. Объект заместитель - это обычная имплементация, которая унаследована от искомого типа SmsSenderImpl:

```
/**
 * Реализация отправки СМС
 */
public static class SmsSenderSubstitutionProxy extends SmsSenderImpl {

    private String from;
    private String to;

    public SmsSenderSubstitutionProxy() { super(null, null); }

    public SmsSenderSubstitutionProxy(String from, String to) { super(from, to); }

    // Getter-Setter Block
}
```

Все что нам необходимо, это просто создать конструктор по умолчанию, а методы оставить как есть. Нужно отметить, что работу методов тоже можно менять, но делать это стоит в зависимости от ваших требований.

Кроме этого, нам нужно зарегистрировать эту замену в расширении. Для этого нужно добавить такой код в `QuarkusNotificationProcessor`:

```
@BuildStep
@Record(STATIC_INIT)
void loadSmsSenderImpl(SenderRecorder recorder,
    BuildProducer<ObjectSubstitutionBuildItem> substitutions) {
    // Register how to serialize SmsSenderImpl
    ObjectSubstitutionBuildItem.Holder<SmsSenderImpl, SmsSenderImplObjectSubstitution.SmsSenderSubstitutionProxy> holder = new ObjectSubstitutionBuildItem.Holder(
        SmsSenderImpl.class, SmsSenderImplObjectSubstitution.SmsSenderSubstitutionProxy.class, SmsSenderImplObjectSubstitution.class
    );
    ObjectSubstitutionBuildItem smsImplSubstitution = new ObjectSubstitutionBuildItem(holder);
    substitutions.produce(smsImplSubstitution);
}
```

В методе происходит регистрация объекта заменителя, и с помощью специальной единицы сборки (`ObjectSubstitutionBuildItem`) мы предоставляем информацию для других сборок.

После того, как мы регистрируем эту единицу, само собой ничего не заработает, т.к. этим этапом сборки, мы всего лишь дали информацию кваркусу, о доступных единицах.

Последнее, что нужно сделать, это заменить нужный объект на объект прокси, для этого перейдем в функцию `registerNativeImageResources` и напишем следующий код:

```
@BuildStep
@Record(STATIC_INIT)
void registerNativeImageResources(BuildProducer<SpiInfoBuildItem> spiInfoBuildItemBuildProducer,
    RecorderContext recorderContext,
    SenderRecorder recorder,
    List<ObjectSubstitutionBuildItem> substitutionBuildItems) throws IOException {

    String service = "META-INF/services/" + Sender.class.getName();
    // read the implementation classes
    Collection<Class<? extends Sender>> implementationClasses = new LinkedHashSet<>();
    Set<String> implementations = ServiceUtil.classNamesNamedIn(Thread.currentThread().getContextClassLoader(),
        service);
    for (String implementation : implementations) {
        Optional<ObjectSubstitutionBuildItem> first = substitutionBuildItems.stream()
            .filter(f -> f.getHolder() != null && f.getHolder().from.getName().equals(implementation))
            .findFirst();
        if (first.isPresent()) {
            implementationClasses.add((Class<? extends Sender>) recorderContext.classProxy(first.get().getHolder().to.getName()));
        } else {
            implementationClasses.add((Class<? extends Sender>) recorderContext.classProxy(implementation));
        }
    }
    // produce a static-initializer with those classes
    List<Sender> services = recorder.registry(implementationClasses);
    spiInfoBuildItemBuildProducer.produce(new SpiInfoBuildItem(services));
}
```

В месте, где мы добавляли имя настоящего класса, теперь мы добавляем имя прокси-класса, тем самым решая проблему отсутствия конструктора по-умолчанию.

5.4 Тестирование расширения

После того, как мы добавим расширение и запустим приложение, мы увидим следующие сообщения:

```
romang@roman-Lenovo-U700:~/repository/work/temprexention/code-with-quarkus$ ./mvnw compile quarkus:dev
[INFO] Scanning for projects...
[INFO] -----< org.acme:code-with-quarkus >-----
[INFO] Building code-with-quarkus 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ code-with-quarkus ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ code-with-quarkus ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- quarkus-maven-plugin:1.4.2.Final:dev (default-cli) @ code-with-quarkus ---
[INFO] Listening for transport dt_socket at address: 5005
Отправляю супер важную информацию через EMAIL при поднятии контекста от sender@gmail.com адресату receiver@gmail.com
Отправляю супер важную информацию через SMS при поднятии контекста от sender@gmail.com адресату receiver@gmail.com
-- QUARKUS --
2020-06-02 15:33:58,657 INFO [io.quarkus] (Quarkus Main Thread) code-with-quarkus 1.0.0-SNAPSHOT (powered by Quarkus 1.4.2.Final) started in 83.255s. Listening on: http://0.0.0.0:8080
2020-06-02 15:33:58,698 INFO [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2020-06-02 15:33:58,699 INFO [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, quarkus-notification, resteasy]
```

5.5 Вывод

Расширение Quarkus добавляет новое поведение, ориентированное на разработчиков, к основному приложению и состоит из двух отдельных частей: блоку сборки и блоку выполнения. Первая часть отвечает за всю сборку метаданных, такую как чтение аннотаций, XML-дескрипторы и т.д. Результатом этой фазы является записанный байт-код, который отвечает за непосредственное создание соответствующих служб времени выполнения.

Это означает, что метаданные обрабатываются только один раз во время сборки, что экономит время запуска, а также использование памяти, поскольку классы и другие структуры, которые используются для обработки, не загружаются (или даже не присутствуют) в JVM времени выполнения.