Разработка на Quarkus

BIV Group

1.0,

Содержание

1 Определения и сокращения

Наименование	Описание
IDE	Integrated development environment - система программных средств, используемая программистами для разработки программного обеспечения.
Quarkus	Фреймворк для разработки микросервисов от компании RedHat (https://quarkus.io/).
Hibernate	Реализация спецификации JPA, предназначенная для решения задач объектно-реляционного отображения (ORM) (https://hibernate.org/).
DAO	Data access object - абстрактный интерфейс к какому-либо типу базы данных или механизму хранения.
КЛАДР	Классификатор адресов Российской Федерации - ведомственный классификатор ФНС России, созданный для распределения территорий между налоговыми инспекциями и автоматизированной рассылки корреспонденции.
СПАРК	Сервис получения информации о контрагентах (ИП / ЮЛ).
Яндекс.Касса	Российский платёжный провайдер, сервис по приёму платежей через интернет в пользу юридических лиц, индивидуальных предпринимателей и некоммерческих организаций.
Google Analytics	Сервис, предоставляемый Google для создания детальной статистики посетителей веб-сайтов.

2 Введение

Для разработки микросервисов был выбран новый фреймворк Quarkus от компании RedHat. Документацию по фреймворку можно посмотреть на официальном сайте https://quarkus.io/guides/. Исходники фреймворка находятся на GitHub https://github.com/quarkusio/quarkus. В данном документе будут описаны концепции разработки на Quarkus, включая примеры для конкретных случаев и ситуаций.

3 Структура модулей

На текущем этапе разработки, все библиотеки/части микросервисов хранятся в одном репозитории, в каталоге `modules` (ссылка на kallithea). В дальнейшем возможен переход на хранение вида "один модуль":"один репозиторий".

Опишем структуру модулей:

- **core** - модули ядра системы;
  - **common** - общие классы для работы системы, к ним можно отнести: общие сериализаторы, конвертеры, константы, мапперы и т.д.;
  - **utils** - классы утилит: работа с датами, математические методы, методы для работы с base64 и т.д.;
  - **dao-sbi** - модуль, содержащий описание всех используемых сущностей БД, подробнее в следующих пунктах;
  - **contract-service** - сервис работы с договором: генерация договора по указанным параметрам, сохранение/загрузка договора и т.д.;
  - **handbook-service** - сервис работы со справочниками;
  - **product-service** - сервис работы с продуктом: загрузка продукта по наименованию, версии, поддержка кэша продуктов и т.д.;
  - **client-action-log-service** - сервис работы с логом действий клиента;
  - **compliance-service** - сервис проверки контрагентов по compliance (возможно его стоит вынести из core);
  - **kladr-service** - сервис работы с КЛАДР;
  - **platform-caller** - сервис вызова веб-сервисов текущей платформы, необходим для более плавного перехода на микросервисы;
- **product** - продуктовые модули;
  - **travel-online** - Страхование путешественников Онлайн;
  - **...**
- **web-caller** - модули для вызова внешних web-сервисов;
  - **google-analytics-service** - сервис отправки статистики в *Google Analytics*;
  - **spark** - модуль работы со *СПАРК*;
  - **yandex-kassa** - модуль работы с *Яндекс.Касса*;
- **integration** - модули для интеграции с внешними системами;
  - **...**

В следующих пунктах работа части модулей будет рассмотрена более подробно.

4 Разработка DAO

Все сущности, находящиеся в текущей БД Oracle описываются в модуле `dao-sbi` (ссылка на kallithea). Разработка сущностей идет с использованием библиотеки Panache. Документацию Panache можно посмотреть на официальном сайте: https://quarkus.io/guides/hibernate-orm-panache. Разберем разработку сущностей на нескольких примерах.

4.1 Создание сущности

Допустим, нам необходимо реализовать DAO класс для работы с договором. Сначала необходимо определить, в каком пакете будет находиться новый класс. Для договора и всех его сущностей необходимо создать пакет (ссылка на kallithea):

```
package com.bivgroup.core.dao.entity.contract;
```

Имя класса необходимо писать без сокращений (например, для секции договора необходимо создать класс `ContractSection`). Создаем класс `Contract`:

```
/**
 * Сущность "Договор"
 *
 * @author BIV group
 * @version 1.0
 */
@Entity
@Table(name = "B2B_CONTR")
public class Contract extends PanacheEntityBase {
```

Для класса, его полей и методов необходимо создавать комментарии для документирования в формате JavaDoc. Каждая сущность должна наследоваться от класса `PanacheEntityBase`. Библиотека Panache также поддерживает репозитории (аналогично, как в Spring Data), но более правильным считается реализация сущности и методов работы с ее данными внутри сущности. Чтобы не вводить наименования полей вручную, можно скопировать список полей прямо из таблицы БД в IDE. Для сущностей с множеством полей можно отсортировать их в алфавитном порядке. Автоматически это может сделать к примеру плагин `String manipulation` для IntelliJ IDEA. Также следует придерживаться следующего правила расположения полей в классе сущностей:

- Сначала всегда располагается ИД сущности.
- Если сущность является чьей то детализацией, то затем располагается ссылка на родителя.
- Далее идут остальные поля (можно в алфавитном порядке), кроме списков (детализаций).
- В конце располагаются детализации сущности (списки).

Названия колонок в аннотации `@Column` указываются в верхнем регистре. Название поля в классе указывается аналогично названию в БД, но в регистре для стандарта Java. Пример:

```
@Column(name = "INSAMCURRENCYID")
public Long insAmCurrencyId;
```

Используя Panache нет необходимости реализовывать геттеры/сеттеры для полей, он будут сгенерированы автоматически. При этом можно реализовать сеттер и сделать его `private`, если мы не хотим, чтобы поле могло редактироваться в коде. Данный сеттер не будет перетерт при генерации.

4.1.1 Поле *ИД сущности*

Разберем описание поля ИД сущности на примере `Contract`:

```
/** ИД договора (первичный ключ) */
@Id
@Column(name = "CONTRID")
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "B2B_CONTR_SEQ")
@SequenceGenerator(name = "B2B_CONTR_SEQ", sequenceName = "B2B_CONTR_SEQ",
allocationSize = 10)
public Long contrId;
```

Если в сущность предусматривается вставка необходимо обязательно указывать аннотации `@GeneratedValue` и `@SequenceGenerator`. В качестве генератора необходимо указывать sequence, который должен иметь название `ИмяТаблицы_SEQ`. Следует отметить, что для решения проблемы пересечения ИД новых сущностей со старым генератором на `AUTO_PK_SUPPORT` было принято решение делать sequence для Quarkus со стартовым значением `10 000 000 000`. Для тех сущностей, которые в текущей платформе работают на Hibernate, используем существующий sequence.

4.1.2. Поле *Дата*

```
/** Дата расторжения договора */
@Column(name = "CANCELDATE")
@Convert(converter = DoubleDateConverter.class)
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = JSONConstants.DateFormat)
public Date cancelDate;
```

Аннотация `@Convert` необходима для дат, которые в БД хранятся как `Double - FLOAT(*)`. Иначе она не нужна.

4.1.3. Поле *Список (детализация)*

```
@OneToMany(mappedBy = "contract", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
public Set<ContractSection> contractSections = new HashSet<>();
```

В названии поля делаем множественное число: `contractSections`. Для детализации используем связь `@OneToMany`. В `mappedBy` указывается название поля с родителем в дочерней сущности.

Также необходимо реализовать метод добавления элемента детализации. Он необходим, чтобы проставить ссылку на родителя в дочерней сущности:

```
public void addContractSection(ContractSection contractSection) {
    contractSection.contract = this;
    this.contractSections.add(contractSection);
}
```

4.1.3. Поле *Ссылка на родителя*

В дочернем классе необходимо описывать ссылку на родителя следующим образом:

```
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "CONTRID")
@JsonIgnore
public Contract contract;
```

Используем связь `@ManyToOne`, в `@JoinColumn` указываем колонку, которая ссылкается на сущность предка. Аннотация `@JsonIgnore` необходима, чтобы не происходило зацикливание маппера Jackson, который сериализует объект в JSON. Реализуем ссылку через объект `Contract contract`.

4.1.4. Поле *Ссылка на справочник / Однонаправленная связь*

```
@OneToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "CONTACTTYPEID")
public CrmContactType contactType;
```

Используем связь `@OneToOne`, в `@JoinColumn` указываем колонку, которая ссылается на справочник. Под справочником имеется ввиду сущность, которая не будет иметь обратной ссылки. Т.е. связь однонаправленная. Аналогично мы можем задать не только ссылку на справочник, но и просто на однонаправленную ссылку на сущность, которая может редактироваться.

4.1.5. Реализация кастомного поиска

```
public static Contract findByContractNumber(String contractNumber) {
    return find(Contract_.CONTR_NUMBER, contractNumber).firstResult();
}
```

Все методы для работы с сущностью необходимо реализовывать в классе самой сущности в виде статических методов. На примере показан поиск договора по номеру. Т.к. считается, что одинаковых номеров договоров нет, в методе идет получение первого результата `.firstResult()`. Вместо строкового значения номера договора указывается константа, сгенерированная maven плагином по сущности договора. Более подробно можно посмотреть по ссылке. Для подключения данного плагина необходимо в зависимости добавить:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-jpamodelgen</artifactId>
    <version>5.4.4.Final</version>
    <scope>provided</scope>
</dependency>
```

В секцию с плагинами добавить:

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.6</source>
        <target>1.6</target>
        <compilerArguments>
            <processor>
org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor</processor>
        </compilerArguments>
    </configuration>
</plugin>
```

После компиляции модуля в сгенерированных классах будут метаданные по всем сущностям из модуля.

Для кастомных запросов можно также использовать HQL:

```
public static PanacheQuery<ShareContract> findByPhoneNumberAndContractId(String
phoneNumber, Long contractId) {
    return find("select t1 from ShareContract t1 join t1.contract t2 where
t1.phoneNumber = ?1 and t2.contrId = ?2", phoneNumber, contractId);
}
```

В случае, если метод может возвращать различное количество данных, следует возвращать `PanacheQuery<КлассРезультата>`. Затем в месте вызова, данных результат можно

использовать по разному. Использовать пейджинг, получать количество записей, преобразовать в стрим и т.д.

4.2 Вызов нативного SQL

Для вызова нативного SQL необходимо получить `EntityManager` и использовать его метод `createNativeQuery`.

Добавляем переменную:

```
@Inject
EntityManager entityManager;
```

Вызываем SQL:

```
String genNumber = entityManager.createNativeQuery("select t.NUMVALUE from
table(select MASKCOUNTER_NEXT(?1) from dual) t")
    .setParameter(1, maskId)
    .getSingleResult().toString();
```

Параметры передаются через метод `setParameter`. Результат можно получить через соответствующий метод интерфейса `javax.persistence.Query`.

4.3 Доступность классов сущностей во внешних модулях

Для того, чтобы сущности модуля были доступны из вне, по ним необходимо построить индекс. Это реализуется путем добавления в секцию плагины `Jandex index`:

```
<plugin>
    <groupId>org.jboss.jandex</groupId>
    <artifactId>jandex-maven-plugin</artifactId>
    <version>1.0.6</version>
    <executions>
        <execution>
            <id>make-index</id>
            <goals>
                <goal>jandex</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Данный индекс необходимо строить во всех модулях, которые расшаривают свой функционал во вне. Это могут быть какие-либо бины, сервисы, сущности и т.д.

4.4 Генерация Javadoc

Для генерации Javadoc необходимо подключить maven плагин:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>3.1.1</version>
    <configuration>
        <source>1.8</source>
    </configuration>
</plugin>
```

Для генерации документации используется команда:

```
mvn javadoc:javadoc
```

5 Разработка нового продукта

В рамках текущей микросервисной архитектуры разработка каждого продукта ведется в отдельном модуле. Также, после реализации продукта, его установка на стенды происходит в отдельном docker контейнере. Таким образом, установка/обновление одного продукта не будет влиять на работу остальных продуктов системы. Размер обновлений будет небольшим. Рассмотрим разработку нового продукта (онлайн-продукт) на основе продукта Страхование путешественников Онлайн. Исходные коды продукта можно найти по ссылке. Продукты следует называть без сокращений, слова отделяются знаком `-`. Для данного продукта наименование проекта: `travel-online`. Структура исходного кода:

- `.mvn` - каталог с Maven Wrapper.
- `docker` - докер файлы для сборки бэк и фронт части.
- `sql` - SQL скрипты для переноса изменений по БД.
- `src` - JAVA исходники.
- `.dockerignore` - исключения файлов из сборки docker.
- файлы `mvnw` и `mvnw.cmd` - файлы Maven Wrapper.
- `pom.xml` - POM файл для сборки продукта с помощью maven.

Структура каталога `src`:

- `main`.
  - `java` - исходники JAVA.
    - неймспейс продукта, в нашем случае это `com.bivgroup.product.travelonline`.
      - `constants` - продуктовые константы, например: сиснеймы рисков, территории и т.д.
      - `exception` - классы-исключения, в нашем случае один класс: `TravelOnlineException`, который используется для обработки ошибок по продукту.
      - `handbook` - dao-классы со справочниками по продукту, которые могут использоваться например в расчете.
      - `resource` - классы-ресурсы, в них задаются REST точки входа в приложение (иногда ресурсы называют контроллерами). В подкаталогах также находятся POJO классы с описанием входных/выходных данных по запросам.
      - `service` - классы-сервисы, которые реализуют бизнес-логику продукта.
  - `resources` - ресурсы приложения.
    - `META_INF` - папка мета-данных.
      - `beans.xml` - файл необходим для того, чтобы запускался поиск бинов.
    - `application.properties` - настройки приложения.

5.1 Настройки приложения

Рассмотрим настройки приложения на примере файла application.properties. Настройки запуска приложения, порт:

```
quarkus.http.port=8080
```

Настройки подключения к БД:

```
quarkus.datasource.url=jdbc:oracle:thin:@DBIP:1521:SID
quarkus.datasource.driver=oracle.jdbc.OracleDriver
quarkus.datasource.username=username
quarkus.datasource.password=password
quarkus.datasource.max-size=8
quarkus.datasource.min-size=2
```

Настройки Hibernate (очень важно, чтобы параметр `generation` был `none`, чтобы Hibernate не влиял на структуру БД):

```
quarkus.hibernate-orm.database.generation=none
quarkus.hibernate-orm.log.sql=false
quarkus.hibernate-orm.dialect=org.hibernate.dialect.Oracle10gDialect
quarkus.hibernate-orm.jdbc.timezone=UTC
```

Настройки вызова сервисов текущей платформы (при необходимости вызова):

```
com.bivgroup.core.../mp-rest/url=http://localhost:8080/
com.bivgroup.core.../mp-rest/scope=javax.inject.Singleton
com.bivgroup.core.../mp-rest/url=http://localhost:8080/
com.bivgroup.core.../mp-rest/scope=javax.inject.Singleton
auth.user=man1
auth.password-hash=356a192b7913b04c54574d18c28d46e6395428ab
```

Настройки логирования (подробнее о логировании можно посмотреть: https://quarkus.io/guides/logging):

```
quarkus.log.file.enable=true
quarkus.log.file.path=./application.log
quarkus.log.file.level=DEBUG
quarkus.log.file.async=true
quarkus.log.file.format=%d{HH:mm:ss} %-5p [%c{2.}] (%t) %s%e%n
```

Настройки Google Analytics (необходимы, если к продукту требуется подключение статистики):

```
ga.is-enabled=true
ga.url=https://ssl.google-analytics.com/collect
ga.tids=UA-101096572-1
ga.i=1
ga.v=1
ga.ec=online_policy
ga.el=travel
ga.dt=Сбербанк страхование
ga.ignoreRefs=sberbankins.ru;localhost
```

5.2 Настройка сборки приложения, файл POM.xml

```
<modelVersion>4.0.0</modelVersion>
<groupId>com.bivgroup.module.product</groupId>
<artifactId>travel-online</artifactId>
<version>1.0.0-SNAPSHOT</version>
<name>Product "Страхование путешественников Онлайн v2" module</name>
```

Тэг `groupId` для всех продуктов будет одинаковый, в `artifactId` задаем имя каталога с продуктом. При разработке версию устанавливаем в `1.0.0-SNAPSHOT`. При релиза продукта версию можно установить в `1.0.0`. В тэге `name` указываем имя продукта по ФТ.

```
<parent>
    <groupId>com.bivgroup.module</groupId>
    <artifactId>product</artifactId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

Данный тэг `parent` будет одинаковым для всех продуктов.

```
<properties>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <maven.compiler.parameters>true</maven.compiler.parameters>
    <surefire-plugin.version>2.22.0</surefire-plugin.version>
    <quarkus.version>1.1.1.Final</quarkus.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <compiler-plugin.version>3.8.1</compiler-plugin.version>
    <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

В настройках задаются версии Quarkus, плагинов, кодировка и т.д. Для реализации большинства продуктов данных настроек будет достаточно.

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-universe-bom</artifactId>
            <version>${quarkus.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Данная секция используется для управления сторонними зависимостями (в нашем случае мы указываем модуль Quarkus).

```
<dependencies>
    ...
<dependencies>
```

В зависимостях мы указываем требуемые модули Quarkus (без версий), наши модули, а также другие используемые библиотеки.

```
<build>
    <plugins>
        <plugin>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-maven-plugin</artifactId>
            <version>${quarkus.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>build</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${compiler-plugin.version}</version>
        </plugin>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>${surefire-plugin.version}</version>
            <configuration>
                <systemProperties>
                    <java.util.logging.manager>
org.jboss.logmanager.LogManager</java.util.logging.manager>
                </systemProperties>
            </configuration>
        </plugin>
    </plugins>
</build>
```

В секции `build` указываются плагины для компиляции проекта. В общем случае изменений в данной секции делать не нужно.

```
<profiles>
    <profile>
        <id>native</id>
        <activation>
            <property>
                <name>native</name>
            </property>
        </activation>
        <build>
            <plugins>
                <plugin>
                    <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>
        <properties>
            <quarkus.package.type>native</quarkus.package.type>
        </properties>
    </profile>
</profiles>
```

В профиле `native` указываются настройки сборки native модуля. На данный момент native сборку не используем.

5.3 Реализация ресурсов продукта

Для нового продукта главным ресурсом (другое наименование ресурса - контроллер) будет являться класс, в котором описаны главное методы по продукту, такие как: расчет премии, сохранение договора и т.д. В нашем случае это будет класс `TravelOnlineResource` (ссылка на kallithea).

```
@Path(ResourceConfig.BASE_URL)
@Tag(name = "Сервисы продукта Страхование путешественников Онлайн v2")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class TravelOnlineResource extends BaseResource {
```

В аннотации `@Path` мы указываем константу. Для локальной разработки она равна `/`, но для стендов она будет отличаться. В ветках стендов ее необходимо задать, как `/НазваниеМодуляПродукта-gate`. В нашем случае это будет: `/travel-online-gate`. Это необходимо для настройки маршрутизации запросов на стендах на nginx. В `@Tag` описывается имя модуля для Swagger UI. Т.к. практически все сервисы продукта принимают на вход JSON и отдают тоже его, то сразу указываем это в аннотациях `@Produces` и `@Consumes`.

Для чтения требуемых параметров из файла `application.properties` необходимо использовать аннотацию `@ConfigProperty`:

```
@ConfigProperty(name = "auth.user")
String login;
```

Далее в классе идет подключение сервисов через `@Inject`, например:

```
@Inject
CalcService calcService;
```

Рассмотрим реализацию на примере сервиса расчета премий по договору:

```
@POST
@Path("/calcContract")
@Operation(summary = "Расчет договора", description = "Расчет премий по
выбранным опциям с учетом возможности указания промокода")
public BaseResponse<CalcContractResult> calcContract(CalcContractRequest request) {
    BaseResponse<CalcContractResult> response = new BaseResponse<>();
    try {
        calcService.validateCalcRequest(request);
        response.setResult(calcService.calcContract(request));
        response.setStatus(getOKStatus());
    } catch (ConstraintViolationException e) {
        response.setStatus(ConstraintViolationParser
.getConstraintViolationResponseStatus(e));
        logger.error("Ошибка во входных данных сервиса: ", e);
    } catch (TravelOnlineException se) {
        response.setStatus(getStatus(2L, se.getMessage()));
        logger.error("Ошибка работы сервиса: ", se);
    } catch (Exception e) {
        response.setStatus(getOtherErrorStatus());
        logger.error("Непредвиденная ошибка работы сервиса: ", e);
    }
    return response;
}
```

В `@Path` указываем точку входа в сервис относительно `@Path` класса. Тэг `@Operation` - описание метода для Swagger UI. В качестве ответа используется базовый класс, который параметризируется результатом работы сервиса. Валидация входящего запроса выносится всегда в сервис, чтобы выдавать ответ в необходимом нам виде. Также это позволяет реализовывать дополнительную бизнес-логику проверок:

```
calcService.validateCalcRequest(request);
```

Затем идет вызов самого сервиса и формирование результата:

```
response.setResult(calcService.calcContract(request));
response.setStatus(getOKStatus());
```

Далее идет обработка возможных ошибок сервиса:

- ConstraintViolationException - данную ошибку сгенерирует сервис валидации входных параметров.
- TravelOnlineException - обрабатываемая ошибка работы сервиса (это обычно ошибка работы бизнес-логики), текст ошибки мы должны вернуть в ответе.
- Exception - любая непредвиденная ошибка, наружу мы возвращаем текст `Другая ошибка` и при этом логируем ее.

5.4 Реализация сервисов продукта

Вся бизнес-логика продукта реализовывается в классах-сервисах. Разберем реализацию на примере сервиса расчета премий.

```
@ApplicationScoped
public class CalcService {
```

В описании сервиса обязательно должна быть аннотация `@ApplicationScoped`. Она говорит о том, что бин будет доступен на протяжении времени жизни приложения.

Для подключения базовых сервисов из `core` необходимо использовать спецификатор:

```
@Inject
@HandbookServiceQualifier
HandbookService handbookService;
```

Спецификатор необходим, т.к. в модулях продукта может быть сервис с таким же названием, как и в `core`.

Рассмотрим реализацию валидации:

```
public void validateCalcRequest(@Valid CalcContractRequest request) {
    if (request.getIsAnnualPolicy()) {
        if (request.getAnnualPolicyType() == null || request.getAnnualPolicyType()
.isEmpty()) {
            throw new ConstraintViolationException("Не указан тип годового полиса",
null);
        }
    }
    if (request.getInsuredGroups().getGroup0_3() + request.getInsuredGroups()
.getGroup4_60() +
                request.getInsuredGroups().getGroup61_70() + request.getInsuredGroups
().getGroup71_80() > 6) {
        throw new ConstraintViolationException("Максимальное количество
застрахованных не может быть больше шести человек", null);
    }
}
```

В описании метода используем аннотацию `@Valid` для автоматического запуска валидации по POJO классу (javax.validation). Дополнительную валидацию реализуем в теле метода, генерируем ConstraintViolationException с текстом ошибки при необходимости.

Рассмотрим структуру метода расчета:

```
@Transactional
public CalcContractResult calcContract(CalcContractRequest request) throws
TravelOnlineException, HandbookException, ProductException {
    ...
    CalcContractResult result = new CalcContractResult();
    ...
    return result;
}
```

Аннотация `@Transactional` используется, если в методе идет сохранение/модификация данных в БД. На весь метод создается одна транзакция. Метод реализует бизнес-логику расчета премий по договору, создает и заполняет результат расчета.

Важно! Если нет необходимости делать дополнительную бизнес-логику проверок, то отдельный метод валидации можно не делать, а аннотацию `@Valid` можно сразу указывать в самом сервисе.

```
public CalcContractResult calcContract(@Valid CalcContractRequest request) {
    ...
}
```

5.5 Создание POJO классов запроса/ответа сервиса

Для каждого реализуемого сервиса необходимо создавать класс, описывающий запрос и класс-результат. При этом класс с ответом является общим:

```
@JsonPropertyOrder({
    "status",
    "result"
})
public class BaseResponse<T> {

    @Schema(description = "Статус")
    private ResponseStatus status;

    @Schema(description = "Результат работы сервиса")
    private T result;

    public ResponseStatus getStatus() {
        return status;
    }

    public void setStatus(ResponseStatus status) {
        this.status = status;
    }

    public T getResult() {
        return result;
    }

    public void setResult(T result) {
        this.result = result;
    }
}
```

В заголовке метода указывается:

```
public BaseResponse<CalcContractResult> calcContract(CalcContractRequest request) {
```

Рассмотрим реализацию POJO класса запроса:

```
@JsonIgnoreProperties(ignoreUnknown = true)
@JsonPropertyOrder({
    "promoCode",
    ...
    "options"
})
public class CalcContractRequest {
    ...
}
```

Тэг `@JsonIgnoreProperties` используется, если извне могут прийти лишние данные и их необходимо игнорировать (обычно с фронт части приходят лишние значения). В

`@JsonPropertyOrder` задается порядок полей в запросе. Порядок должен соответствовать порядку полей в классе. В теле класса описываются поля с аннотациями для Swagger, ограничениями `javax.validation.constraints`, кастомными сериализаторами/десериализаторами и т.д., например:

```
@Schema(required = true, example = "5", description = "Количество дней поездки")
@NotNull(message = "Не указано количество дней поездки")
@Min(value = 1, message = "Количество дней поездки не может быть меньше 1 дня")
@Max(value = 365, message = "Количество дней поездки не может быть больше 365
дней")
private Long insDayCount;
```

Тэг `@Schema` описывает поле для Swagger. Желательно указывать атрибут `example` для того, чтобы при открытии Swagger UI был виден корректный пример запроса с данными. После запуска приложения сгенерированный документ Swagger будет доступен по ссылке: http://localhost:8080/swagger-ui (порт соответствущий указанному в `application.properties`). Нужно также убедиться, что в проект подключена зависимость:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-smallrye-openapi</artifactId>
</dependency>
```

После описания идут ограничения/валидация. Поле должно быть `private`, по каждому полю необходимо сгенерировать сеттер/геттер.

5.5.1 Описание поля *Дата*

```
@Schema(required = true, example = "2020-01-15", description = "Дата начала поездки
")
@NotNull(message = "Не указана дата начала поездки")
@JsonSerialize(using = CustomDateSerializer.class)
@JsonDeserialize(using = CustomDateDeserializer.class)
private Date dateBeginTravel;
```

Для даты необходимо использовать аннотации `@JsonSerialize`, `@JsonDeserialize` и указать в них класс `CustomDateSerializer.class`. Это требуется для того, чтобы даты могли нормально обрабатываться в различных форматах, а также для корректно сериализации класса в JSON.

5.5.2 Описание поля с валидацией по списку

```
@Schema(required = true, enumeration = {SCHENGEN_NO_RF, ALL_NO_USA_RF, RF_SNG, USA},
example = SCHENGEN_NO_RF, description = "Территория страхования")
@NotEmpty(message = "Не указана территория страхования")
@OneOfArray(values = {SCHENGEN_NO_RF, ALL_NO_USA_RF, RF_SNG, USA}, message = "Указана
неверная территория страхования")
private String insTerritory;
```

Для реализации валидации значение по списку (значение соответствует одному элементу из списка) используется кастомный валидатор `@OneOfArray`. В тэге `@Schema` также указывается, что значение должно соответствовать описанным в списке.

5.5.3 Использование операции trim для удаления ненужных пробелов

Если на вход сервиса в качестве одного из полей приходит строка, возможно по ней необходимо выполнить операцию `trim`. Она удаляет ненужные пробелы в начале и в конце строки. Это может понадобится, например, для полей под ФИО страхователя, которые введены на интерфейсе. Пользователь может случайно ввести в конце пробел и не заметить этого. Либо пользователь может поставить пробел в отчестве, чтобы интерфейс не ругался на его обязательность. Сервис должен контроллировать это. Используем класс `com.bivgroup.common.converter.TrimConverter`. Просто добавляем его в аннотации к необходимому полю:

```
@Schema(required = true, example = "Иван", description = "Имя застрахованного")
@NotEmpty(message = "Не указано имя застрахованного")
@JsonDeserialize(using = TrimConverter.class)
private String name;
```

Если в качестве имени придет пробел, класс его обрежет, строчка станет пустой и сработает ограничение `@NotEmpty`.

5.5.4 Описание класса-результата

Класса результата реализуется аналогично классу запросу, но в нем не нужна валидация данных. Например:

```
@JsonPropertyOrder({
    "hash"
})
public class SaveContractResult {

    @Schema(description = "Хеш договора")
    private String hash;

    public String getHash() {
        return hash;
    }

    public void setHash(String hash) {
        this.hash = hash;
    }
}
```

5.5.5 Использование наследования классов в запросах / ответах

5.6 Реализация проверки работоспособности сервисов продукта

5.7 Подключение лога действий клиента

5.8 Подключение проверок Compliance

5.9 Подключение статистики Google Analytics

5.10 Реализация сборки продукта, установка на DEV стенд

6 Q & A

6.1 Ошибка коммита транзакции в сервисе

```
 javax.persistence.TransactionRequiredException: Transaction is not active, consider
adding @Transactional to your method to automatically activate one.
        at io.quarkus.hibernate.orm.runtime.entitymanager
.TransactionScopedEntityManager.persist(TransactionScopedEntityManager.java:111)
        at io.quarkus.hibernate.orm.runtime.entitymanager.ForwardingEntityManager
.persist(ForwardingEntityManager.java:27)
        at io.quarkus.hibernate.orm.panache.runtime.JpaOperations.persist
(JpaOperations.java:34)
        at io.quarkus.hibernate.orm.panache.runtime.JpaOperations.persist
(JpaOperations.java:29)
        at io.quarkus.hibernate.orm.panache.PanacheEntityBase.persist
(PanacheEntityBase.java:43)
```

Ошибка возникает в случае, если у метода (и у методов, вызвывших метод) нет аннотации `@Transactional`. Если аннотация у метода присутствует, необходимо убедиться, что область видимости метода `public` или `protected`. Для `private` методов аннотация `@Transactional` **не** работает.

6.2 При использовании Java Reflection API не обновляется сущность Hibernate

Допустим мы загрузили сущность договора `Contract`, она находится в состоянии `Persistent`. С помощью класса `java.lang.reflect.Field` и метода `set` мы решили установить значение в загруженный объект договора. Это иногда может понадобиться, если вдруг пришла `Map` со значениями ключей, аналогичными полям класса. Чтобы не маппить руками мы используем Reflection API. При завершении метода и коммите транзакции наблюдаем, что ничего в БД не изменилось. Это происходит из-за того, что при создании объектов классов сущностей Hibernate обертывает их своим прокси классом. Этот класс помимо прочего позволяет наблюдать за изменением полей объекта для того, чтобы понять, нужно ли обновлять данные в БД. Когда мы устанавливаем значение поля напрямую через `Field.set`, Hibernate не видит эти изменения. Необходимо использовать сеттер поля для корректного задания значения (это и по логике работы с объектом корректней, т.к. сеттер может быть кастомный). Для этого необходимо использовать класс `org.apache.commons.beanutils.BeanUtils`. Подключаем зависимость:

```
<dependency>
    <groupId>commons-beanutils</groupId>
    <artifactId>commons-beanutils</artifactId>
    <version>1.9.4</version>
</dependency>
```

В коде пишем:

```
BeanUtils.setProperty(contract, "field", "value");
```

Метод `BeanUtils.setProperty` использует именно сеттер класса, что позволяет Hibernate видеть изменения и успешно коммитить их в БД.

Важно!!! Необходимо в конструктор сервиса добавить строчку:

```
public HandbookService() {
    BeanUtilsBean.getInstance().getConvertUtils().register(false, true, 0);
}
```

Она необходима, чтобы BeanUtils не подставлял дефолтные значения. Например вместо `null` для `Long` поля он подставил бы `0`. Что является некорректным для нас.

7 Предисловие

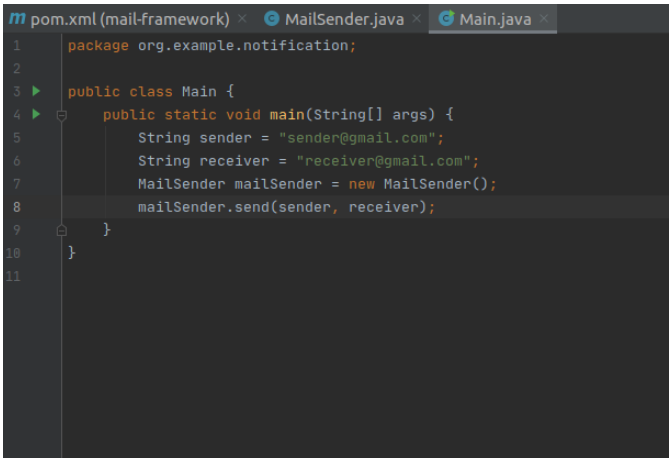
Quarkus - это фреймворк, состоящий из ядра и набора расширений. Ядро основано на внедрении Context и Dependency Injection (CDI), а расширения обычно предназначены для интеграции сторонней инфраструктуры путем предоставления их основных компонентов в виде компонентов CDI.

7.1 Что такое Quakus Extension

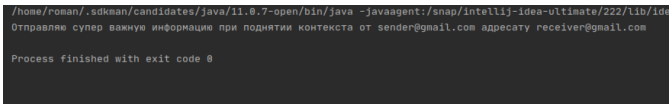
Quarkus Extension- это просто модуль, который может работать поверх приложения Quarkus. Наиболее распространенный вариант использования такого расширения - запуск стороннего фреймворка поверх приложения Quarkus.

7.2 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:



При запуске можно увидеть следующее:



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.

7.3 Как создать расширение на простом примере

Для ознакомления можно попробовать создать расширение самому, пройдя по ссылке

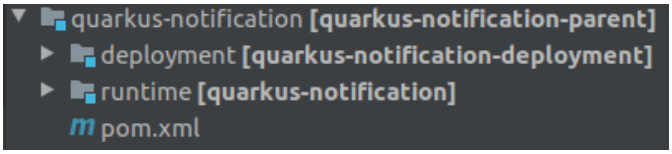
\url{https://quarkus.io/guides/building-my-first-extension}. Все исходники по данной теме можно найти в архиве \textbf{firs-extention.zip}.

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:



7.3.1 Имплементация runtime модуля

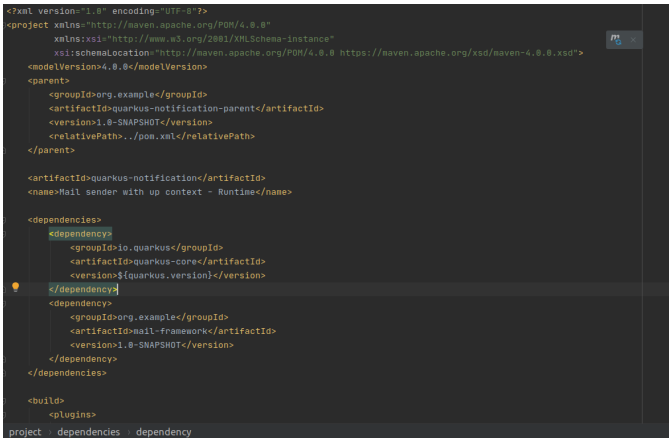
В runtime модуле мы реализуем:

- Конфигурационный класс, который будет хранить в себе настройки адресов отправителя и получателя)
- Поставщика notification API (Предоставление расширением бина, который отвечает за notification API)
- Рекордер, который будет работать как прокси обьект 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:



7.3.2 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;
}
```

7.3.3 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.

7.3.4 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 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).

7.3.5 Реализация deployment

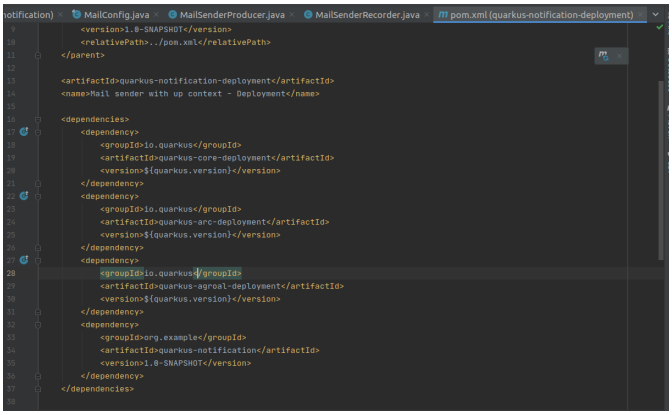
Центральными компонентами расширения Quarkus являются процессоры Build Step. Это методы, аннотированные как @BuildStep, которые генерируют байт-код через устройства записи, и они выполняются во время сборки с помощью цели сборки модуля quarkus-maven-plugin, настроенного в приложении Quarkus.

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.

7.3.6 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:



7.3.7 Реализация Build Step Processors

Как ранее упоминалось, buildStep это такая инструкция, которая позволяет пошагово настраивать бины и записивать 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 внутрь 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);
    }

}
```

7.3.8 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.

```
roman@roman-Lenovo-G700: ~/repository/work/tempextention/code-with-quarkus
roman@roman-Lenovo-G700: ~/repository/work/tempextention/code-with-quarkus
roman@roman-Lenovo-G700:~/repository/work/tempextention/code-with-quarkus$ ./mvnw compile quarkus:dev
[INFO] Scanning for projects...
[INFO]
[INFO] ---------------------< org.acme:code-with-quarkus >---------------------
[INFO] Building code-with-quarkus 1.0.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/quarkus-notification/1.0-SNAPSHOT/
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/quarkus-notification-parent/1.0-SN
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/mail-framework/1.0-SNAPSHOT/maven-
[INFO]
[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]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ code-with-quarkus ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /home/roman/repository/work/tempextention/code-with-quarkus/target/classes
[INFO]
[INFO] --- quarkus-maven-plugin:1.4.2.Final:dev (default-cli) @ code-with-quarkus ---
Listening for transport dt_socket at address: 5005
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/quarkus-notification/1.0-SNAPSHOT/
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/quarkus-notification-parent/1.0-SN
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/mail-framework/1.0-SNAPSHOT/maven-
Downloading from nexus-repo: https://192.168.1.46:8443/repository/maven-public/org/example/quarkus-notification-deployment/1.
Отправляю супер важную информацию при поднятии контекста от sender@gmail.com адресату receiver@gmail.com
 _____ _____ _____ _____ _____ _____ _____ _____
 -/ ___\/ / / / _ \ / __ \/ /_/ / / ___/
 -/ /_/ / /_/ / _ \| , _/ ,< / / /_/ /\ \
--\___\_\____/_/ \_/_/|_/_/|_|\____/___/
2020-05-29 09:51:56,963 INFO  [io.quarkus] (Quarkus Main Thread) code-with-quarkus 1.0.0-SNAPSHOT (powered by Quarkus 1.4.2.F
/0.0.0.0:8080
2020-05-29 09:51:56,972 INFO  [io.quarkus] (Quarkus Main Thread) Profile dev activated. Live Coding activated.
2020-05-29 09:51:56,973 INFO  [io.quarkus] (Quarkus Main Thread) Installed features: [cdi, quarkus-notification, resteasy]
```

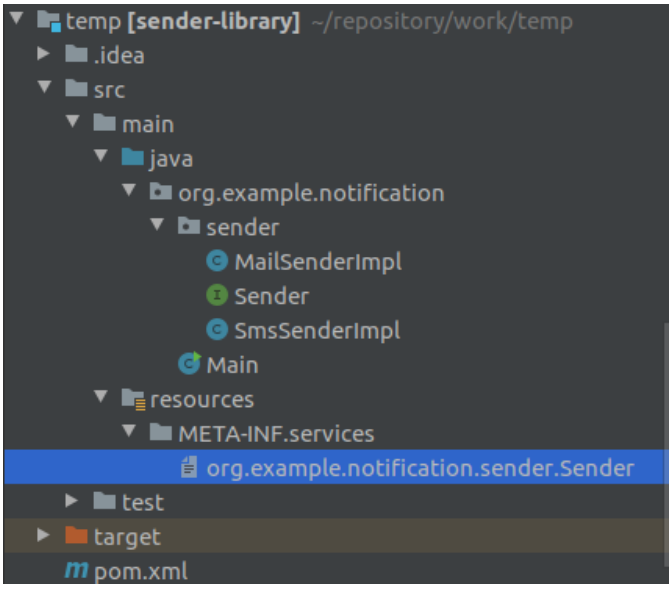
После запуска мы увидим, что отправляется наше импровизированное сообщение, а также увидим наше расширение в списке подключенных

7.4 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.

7.4.1 Структура нового сервиса

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

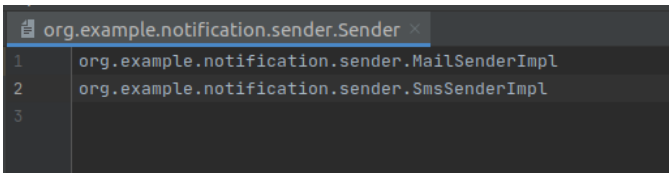


Структура в себе содержит простой интерфейс Sender, который предоставляет контракт по отправке сообщения, и две имплементации EmailSenderImpl и SmsSenderImpl. Первая имплементация подразумевает отправку сообщения через email, вторая через смс.

Конкретный функционал будет упущен, для демонстрации будет пример, который будет печать 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:



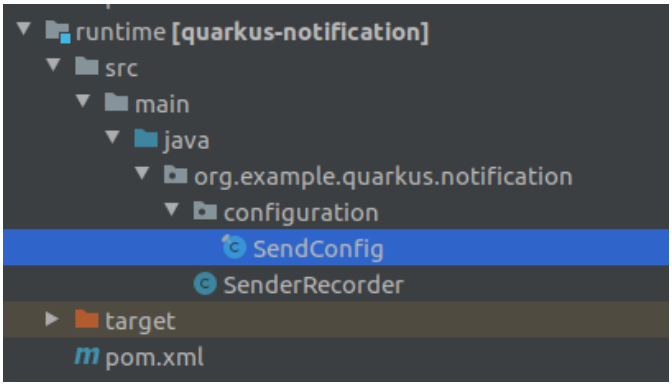
Как видно, тут зарегистрированны доступные реализации интерфейса Sender. Важно отметить, что количество реализаций может быть различным.

7.4.2 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.

7.4.3 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 развертывания, откладываются до времени выполнения. Отсюда и название.

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

Данный класс содержит 2 метода:

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

Метод registry регистрирует бины, предварительно их создавая:

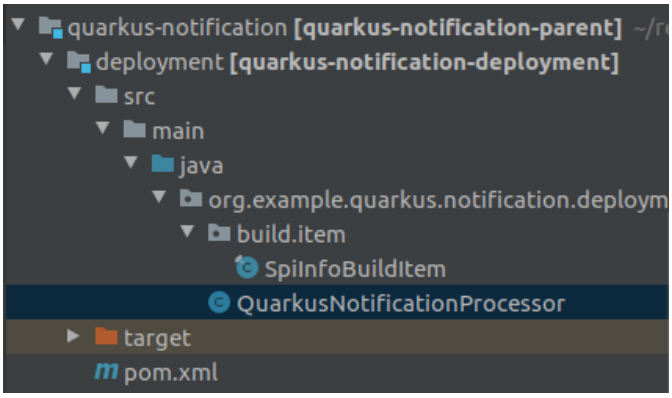
```
/**
 * Регистрация списка имплементаций отправщика
 *
 * @param implementationClasses классы имплементаторы
 * @return
 */
public List<Sender> registry(Collection<Class<? extends Sender>> implementationClasses) {
    // configure our service statically
    List<Sender> services = new ArrayList<>(implementationClasses.size());
    // 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);
    });
}
```

7.4.4 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, оносительно предыдушего 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.

7.4.5 Обзор процессора QuarkusNotificationProcessor

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

- feature
- registerNativeImageResources
- processSend

Часть из 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 фасе, т.е. как только будет запушено 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 попробуете запустить приложение с нашим расширением, то вы столкнетесь с тем, что оно у вас упадет с ошибкой:

```
[INFO] --- maven-resources-plugin:3.0:resources (default-resources) @ code-with-quarkus ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 2 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ code-with-quarkus ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[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.jbo.threads] (main) JBoss Threads version 3.1.1.Final
2020-06-02 12:54:17,970 ERROR [io.qua.ver.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$1.run(StartupActionImpl.java:99)
        at java.base/java.lang.Thread.run(Thread.java:834)
Caused by: java.lang.ExceptionInInitializerError
        at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at java.base/jdk.internal.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at java.base/jdk.internal.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
        at java.base/java.lang.reflect.Constructor.newInstance(Constructor.java:490)
        at java.base/java.lang.Class.newInstance(Class.java:584)
        at io.quarkus.runtime.Quarkus.run(Quarkus.java:68)
        at io.quarkus.runtime.Quarkus.run(Quarkus.java:38)
        at io.quarkus.runtime.Quarkus.run(Quarkus.java:186)
        at io.quarkus.runner.GeneratedMain.main(GeneratedMain.zig:29)
        ... 6 more
Caused by: java.lang.RuntimeException: Failed to start quarkus
        at io.quarkus.runner.ApplicationImpl.<clinit>(ApplicationImpl.zig:470)
        ... 15 more
Caused by: java.lang.IllegalArgumentException: Unable to instantiate service class org.example.notification.sender.SmsSenderImpl
        at org.example.quarkus.notification.SenderRecorder.registry(SenderRecorder.java:28)
        at io.quarkus.deployment.steps.QuarkusNotificationProcessor$registerNativeImageResources5.deploy_0(QuarkusNotificationProcessor$registerNativeImageResources5.zig:
101)
        at io.quarkus.deployment.steps.QuarkusNotificationProcessor$registerNativeImageResources5.deploy(QuarkusNotificationProcessor$registerNativeImageResources5.zig:15
0)
        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
```

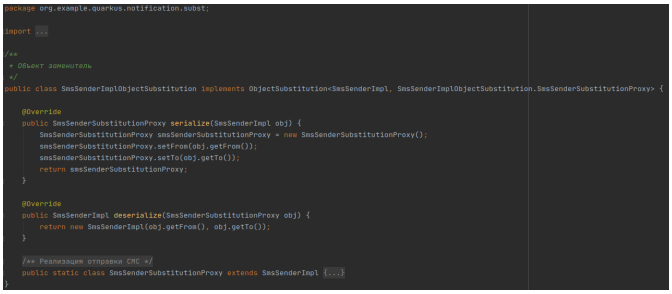
Это сделано специально. Изначально в одной из имплементаций интерфейса Sender я не создал конструктор по умолчанию. Это было сделано для того, чтобы показать, что библиотеки, могут содержать код, который вам нужен, но вы не можете его переписать сами. Для таких случаев необходимо использовать такое понятие,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

7.4.6 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.

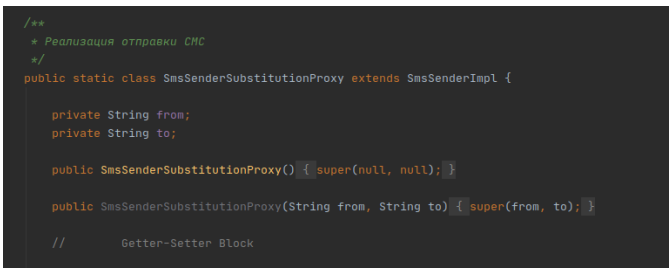
Для этого необходимо создать в runtime модуле класс, который будет будет имплементировать данный интерфейс. В нашем случае пример такого класса можно видеть ниже:



```
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.

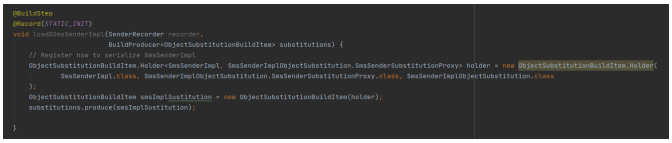
Как видно, интерфейс ObjectSubstitution предоставляет два метода. Метод сериализации и десереализации. Именно эти методы отвечают за конвертацию одного объекта в другой без особых усилий со стороны разработчика. Объект заместитель - это обычная имплементация, которая унаследована от искомого типа SmsSenderImpl:



```
/**
 * 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:



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

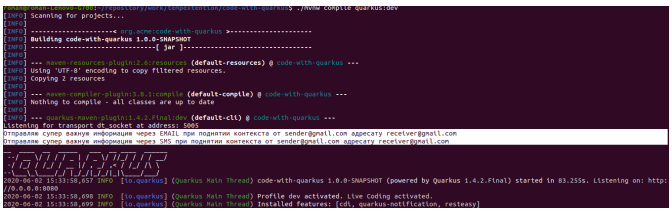
После того, как мы зарегистрируем эту единицу, само собой ничего не заработает, т.к. этим этапом сборки, мы всего лишь дали информацию кваркусу, о доступных единицах.

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

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

7.4.7 Тестирование расширения

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



7.5 Вывод

Расширение Quarkus добавляет новое поведение, ориентированное на разработчиков, к основному приложению и состоят 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 созданние 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.

8 Тестирование приложений на Quarkus

Введение

Прежде чем начать, необходимо быстро пробежаться по основам основ, что же из себя представляет тестирование. Основную информацию о том, как тестировать приложение на Quarkus можно посмотреть на официальном сайте в разделе Quarkus - Testing Your Application

8.1 Основные понятия

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование — это одна из техник контроля качества, включающая в себя активности по планированию работ (Test Management), проектированию тестов (Test Design), выполнению тестирования (Test Execution) и анализу полученных результатов (Test Analysis).

Качество программного обеспечения (Software Quality) — это совокупность характеристик программного обеспечения, относящихся к его способности удовлетворять установленные и предполагаемые потребности. [Quality management and quality assurance]

8.1.1 Цели тестирования

Для того, чтобы было максимально понятно, я выделил несколько целей тестирования. Тестирование направлено для того, чтобы:

- Повысить вероятность того, что приложение, предназначенное для тестирования, будет работать правильно при любых обстоятельствах.
- Повысить вероятность того, что приложение, предназначенное для тестирования, будет соответствовать всем описанным требованиям.
- Предоставление актуальной информации о состоянии продукта на данный момент.

8.1.2 Основные уровни тестирования

1. **Модульное тестирование (Unit Testing)**. Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.). Оно, как правило, наиболее понятное для программиста. Фактически это тестирование методов какого-то класса программы в изоляции от остальной программы. Не всякий класс легко покрыть unit тестами. При проектировании нужно учитывать возможность тестируемости и зависимости класса делать явными. Чтобы гарантировать тестируемость можно применять TDD методологию, которая предписывает сначала писать тест, а потом код реализации тестируемого метода. Тогда архитектура получается тестируемой. Распутывание зависимостей можно осуществить с помощью Dependency Injection. Тогда каждой зависимости явно сопоставляется

интерфейс и явно определяется как инжектируется зависимость — в конструктор, в свойство или в метод. Для осуществления unit тестирования существуют специальные фреймворки, одним из них является JUnit

2. **Интеграционное тестирование (Integration Testing)**. Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.
3. **Системное тестирование (System Testing)**. Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Другими словами это тестирование программы в целом. Для небольших проектов это, как правило, ручное тестирование — запустил, пощелкал, убедился, что (не) работает.
4. **Операционное тестирование (Release Testing)**. Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес моделе системы. Следует учесть, что и бизнес модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации. Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др. Очевидно, что нахождение подобных вещей на стадии внедрения — критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.
5. **Приемочное тестирование (Acceptance Testing)**. Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:
    - определения удовлетворяет ли система приемочным критериям;
    - вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

8.2 Как подключить и пользоваться библиотекой.

8.2.1 Подключаемые модули

Для того, чтобы начать тестировать, необходимо подключить модули, которыми мы в следствии будем пользоваться. В качестве основного мы будем использовать JUnit. На момент написания документации самой свежей версией была JUnit5. Выбор механизма тестирования обусловлен общими рекомендациями разработчиков Quarkus, и все примеры тестов были представлены именно на этой библиотеке.

Стоит упомянуть, что способ подключения зависит от того, что будет тестироваться. Примеры мы будем рассматривать на основе модулей:

- Тестирование библиотеки (на примере queyenne и bivaspect)

- Тестирование расширения (на примере bivcore)
- Тестирование сервиса (на примере tarificator)

Подключаемые зависимости зависят от необходимости поднятия quarkus контекста.

Если поднятие контекста не требуется, то стоит подключать такую зависимость:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.6.2</version>
  <scope>test</scope>
</dependency>
```

Данную версию под капотом предоставляет сам кваркус, и для того, чтобы мы могли потом сделать из нашего модуля нативный запус (через .exe), то стоит использовать именно те библиотеки, которые рекомендованы самим quarkus.

Если все же вам требуется поднимать контекст (для тестирования расширений или сервисов это обычно требуется), то подключаем следующий набор зависимостей:

```
 <dependencies>
        <!-- Расширение для тестирования, которое предоставляет сам Quarkus
-->
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-junit5</artifactId>
            <scope>test</scope>
        </dependency>
        <!-- Тестирование вызово Rest служб  -->
        <dependency>
            <groupId>io.rest-assured</groupId>
            <artifactId>rest-assured</artifactId>
            <scope>test</scope>
        </dependency>
        <!--  Расширение для h2 in memory database  -->
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-jdbc-h2</artifactId>
            <scope>test</scope>
        </dependency>
        <!--  Дополнительная утилита для наполнения тестовой бд данными  -->
        <dependency>
            <groupId>com.github.database-rider</groupId>
            <artifactId>rider-cdi</artifactId>
            <version>1.14.0</version>
            <scope>test</scope>
        </dependency>
        <!--  Дополнительные фичи junit5  -->
        <dependency>
            <groupId>io.quarkus</groupId>
            <artifactId>quarkus-junit5-internal</artifactId>
            <version>${quarkus.platform.version}</version>
            <scope>test</scope>
        </dependency>
        <!-- https://mvnrepository.com/artifact/stax/stax-api -->
        <dependency>
            <groupId>stax</groupId>
            <artifactId>stax</artifactId>
            <version>1.2.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

8.2.1 Немного о JUnit5

Для тех, кто ниразу не пользовался этой библиотекой, то стоит ввести очень быструю инструкцию. Что же такое JUnit? **JUnit** — библиотека для модульного тестирования программного обеспечения на языке Java. Данная библиотека является основой тестирования на Java. Из-за высокой популярности и стабильности он породил экосистему

расширений — JMock, EasyMock, DbUnit, HttpUnit и т. д.

Основную информацию (спецификацию) о данной библиотеке можно найти на сайте Спецификация JUnit

8.2.2 Функциональность JUnit5

Не смотря на то, что JUnit представляет огромную функциональности для тестирования, наиболее частые функции, которыми пользуются многие программисты, представлены ниже:

- junit.framework.Assert: assertEquals, assertFalse, assertNotNull, assertNull, assertNotSame, assertSame, assertTrue
- junit.framework.TestCase extends junit.framework.Assert: run, setUp, tearDown

8.2.3 Пример теста JUnit5

Следующий пример дает представление о минимальных требованиях для написания теста в JUnit:

```
 import static org.junit.jupiter.api.Assertions.assertEquals;

import example.util.Calculator;

import org.junit.jupiter.api.Test;

class MyFirstJUnitJupiterTests {

    private final Calculator calculator = new Calculator();

    @Test
    void addition() {
        assertEquals(2, calculator.add(1, 1));
    }

}
```

JUnit Jupiter поддерживает следующие аннотации для настройки тестов и расширения инфраструктуры.

Все основные аннотации находятся в пакете org.junit.jupiter.api в модуле junit-jupiter-api. Ниже представлен почти полный набор:

Аннотация	Описание
@Test	Column 2, Обозначает, что метод является тестовым методом. В отличие от аннотации @Test в JUnit 4, эта аннотация не объявляет никаких атрибутов, поскольку тестовые расширения в JUnit Jupiter работают на основе своих собственных выделенных аннотаций. Такие методы наследуются, если они не переопределены.
@ParameterizedTest	Обозначает, что метод является параметризованным тестом. Такие методы наследуются, если они не переопределены.
@RepeatedTest	Обозначает, что метод является шаблоном теста для повторного теста. Такие методы наследуются, если они не переопределены.
@TestFactory	Обозначает, что метод является фабрикой тестов для динамических тестов. Такие методы наследуются, если они не переопределены.
@TestTemplate	Обозначает, что метод является шаблоном для тестовых случаев, предназначенных для многократного вызова в зависимости от количества контекстов вызова, возвращаемых зарегистрированными поставщиками. Такие методы наследуются, если они не переопределены.
@TestMethodOrder	Используется для настройки порядка выполнения тестового метода для аннотированного тестового класса; похож на @FixMethodOrder JUnit 4. Такие аннотации наследуются.
@TestInstance	Используется для настройки жизненного цикла экземпляра теста для аннотированного класса теста. Такие аннотации наследуются.
@DisplayName	Объявляет настраиваемое отображаемое имя для класса теста или метода теста. Такие аннотации не наследуются.
@DisplayNameGeneration	Объявляет пользовательский генератор отображаемого имени для тестового класса. Такие аннотации наследуются.

Аннотация	Описание
@BeforeEach	Обозначает, что аннотированный метод должен выполняться перед каждым методом @Test, @RepeatedTest, @ParameterizedTest или @TestFactory в текущем классе; аналогично JUnit 4's @Before. Такие методы наследуются, если они не переопределены.
@AfterEach	Обозначает, что аннотированный метод должен выполняться после каждого метода @Test, @RepeatedTest, @ParameterizedTest или @TestFactory в текущем классе; аналогично JAnit 4's @After. Такие методы наследуются, если они не переопределены.
@BeforeAll	Обозначает, что аннотированный метод должен выполняться перед всеми методами @Test, @RepeatedTest, @ParameterizedTest и @TestFactory в текущем классе; аналогично @BeforeClass JUnit 4. Такие методы наследуются (если они не являются скрытыми или переопределенными) и должны быть статическими (если не используется жизненный цикл тестового экземпляра для каждого класса).
@AfterAll	Обозначает, что аннотированный метод должен выполняться после всех методов @Test, @RepeatedTest, @ParameterizedTest и @TestFactory в текущем классе; аналогично @AfterClass JUnit 4. Такие методы наследуются (если они не являются скрытыми или переопределенными) и должны быть статическими (если не используется жизненный цикл тестового экземпляра для каждого класса).
@Nested	Обозначает, что аннотированный класс является нестатическим вложенным тестовым классом. Методы @BeforeAll и @AfterAll нельзя использовать непосредственно в классе теста @Nested, если не используется жизненный цикл экземпляра для каждого класса. Такие аннотации не наследуются.

Аннотация	Описание
@Tag	Используется для объявления тегов для фильтрации тестов на уровне класса или метода; аналогично тестовым группам в TestNG или категориям в JUnit 4. Такие аннотации наследуются на уровне класса, но не на уровне метода.
@Disabled	Используется для отключения тестового класса или тестового метода; аналогично @Ignore JUnit 4. Такие аннотации не наследуются.
@Timeout	Используется для проверки превышения таймаута теста, фабрики тестов, шаблона теста или метода жизненного цикла, если его выполнение превышает заданную продолжительность. Такие аннотации наследуются.
@ExtendWith	Используется для декларативной регистрации расширений. Такие аннотации наследуются.
@RegisterExtension	Используется для регистрации расширений программно через поля. Такие поля наследуются, если они не затенены.
@TempDir	Используется для предоставления временного каталога посредством введения поля или ввода параметра в методе жизненного цикла или в методе тестирования; находится в пакете org.junit.jupiter.api.io.

8.3 Примеры тестирования.

Мы рассмотрим 3 основных кейса для тестирования, о которых говорили выше. Но прежде чем начать стоит упомянуть кейс, который в некоторых случаях требует подключенной бд. Как правило, тестировать на реальных бд не положено как с технической стороны, так и с идейной.

Все данные могут меняться в соответствии с требованиями разработчика, который тестирует функциональность. Поэтому, для тестов часто используется выделенная бд (такой способ возможен, но не рекомендуется всеравно) или используются in-memory database. В нашем случае мы будем пользоваться вторым вариантом, т.к. он более надежен. В качестве такой бд мы будем использовать H2

Для того, чтобы ее подключить, необходимо подключить соответствующее расширение quarkus и настроить подключение.

Зависимость выглядит следующим образом:

```
  <dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-jdbc-h2</artifactId>
    <scope>test</scope>
  </dependency>
```

Настройки подключения к бд (src/test/resources/application.properties):

```
## TEST IN MEMORY DATABASE
quarkus.datasource.url=jdbc:h2:mem:public
quarkus.datasource.driver=org.h2.Driver
quarkus.datasource.username=sa
quarkus.datasource.password=
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.sql=true
```

После того, как бд подключено, разработчику всегда нужно будет как-то инициализировать нужные данные для теста. Наполнение данными можно производить многими способами, но в данном разделе будут рассмотрены два способа:

- Выполнение скриптов непосредственно через механизмы H2
- Наполнение с помощью библиотеки Database Rider (Документация Database Rider)

Оба способа являются хорошим выбором.

8.3.1 Выполнение скриптов непосредственно через механизмы H2

Для того, чтобы выполнить скрипт, можно использовать **RunScript**. Это класс, который позволяет выполнять скрипты для бд. Он поставляется пакетом org.h2.tools, который будет доступен после добавления зависимости **quarkus-jdbc-h2**.

Для удобства можно создать утильный класс, для того, чтобы можно было пользоваться готовым функционалом (но это не является обязательным условием):

```
public class H2ScriptRunner {

    private final Logger LOGGER = Logger.getLogger(H2ScriptRunner.class.getName());

    private final String url = "jdbc:h2:mem:public";
    private final String userName = "sa";
    private final String password = "";


    public H2ScriptRunner() {
    }

    /**
     * Выполнение скрипта над h2 бд
     *
     * @param filePath путь до выполняемого sql файла
     * @throws SQLException
     */
    public void runSQLFile(String filePath) throws SQLException {
        RunScript.execute(url
                , userName
                , password
                , "classpath:" + filePath
                , Charset.defaultCharset()
                , false
        );
    }

}
```

Пример использования:

```
class h2ScriptRunnerExample {

    private static H2ScriptRunner h2ScriptRunner;

    @BeforeAll
    static void beforeAll() throws SQLException {
        h2ScriptRunner = new H2ScriptRunner();
        h2ScriptRunner.runSQLFile("sql/other/BaseFacadeTest_create.sql");
    }

    @AfterAll
    static void afterAll() throws SQLException {
        h2ScriptRunner.runSQLFile("sql/other/BaseFacadeTest_delete.sql");
    }
}
```

Как видно из кода выше, создается h2ScriptRunner и выполняется нужный скрипт. Кроме этого это делается при помощи BeforeAll и AfterAll, это говорит о том, что даннын скрипты выполняются до и после выполнения тестов, тем самым один наполняет, другой удаляет данные.

8.3.2 Наполнение с помощью библиотеки Database Rider

Database Rider интегрирует JUnit и DBUnit. Эта мощная комбинация позволяет легко подготовить состояние базы данных для тестирования с помощью файлов yaml, xml, json, xls или csv.

Основное вдохновение в Database Rider было взято из постоянного расширения Arquillian - библиотеки для интеграционных тестов базы данных в контейнере.

Для того, чтобы добавить Database Rider необходимо подключить зависимость:

```
    <dependency>
      <groupId>com.github.database-rider</groupId>
      <artifactId>rider-cdi</artifactId>
      <version>1.14.0</version>
      <scope>test</scope>
    </dependency>
```

Для того, чтобы зарегистрировать новые данные, необходимо создать файл с расширением yaml (также можно работать и с другими расширениями, такие как yaml, xml, json, xls или csv, но рассматривать что-то другое мы не будем):

```
    USER:
      - ID: 1
        NAME: "@realpestano"
      - ID: 2
        NAME: "@dbunit"
```

Описание означаает:

- Что мы хотим создать таблицу с именем USER
- ID и NAME являются аттрибутами таблицы
- ID помеченное "-", является первичным ключом
- 1, "@realpestano" значения аттрибутов

Для того, чтобы эти данные добавились в таблицу для конкретного теста, необходимо добавить аннотацию @DBUnitInterceptor (только для класса) и @DataSet("BaseFacadeTest.yml") (как на класс, так и на метод)

Пример использования:

```
@QuarkusTest
@DBUnitInterceptor
@DataSet(value = "BaseFacadeTest.yml",
        cleanBefore = true,
        cleanAfter = true
)
class BaseFacadeTest {

    private static H2ScriptRunner h2ScriptRunner;

    @Inject
    @BOName(value = "BaseFacadeTestFake")
    BaseFacade baseFacadeImpl;

    @BeforeAll
    static void beforeAll() throws SQLException {
        h2ScriptRunner = new H2ScriptRunner();
        h2ScriptRunner.runSQLFile("sql/other/BaseFacadeTest_create.sql");
    }

    @AfterAll
    static void afterAll() throws SQLException {
        h2ScriptRunner.runSQLFile("sql/other/BaseFacadeTest_delete.sql");
    }

    @Test
    void selectQuery() throws Exception {
        Map<String, Object> params = new HashMap<>();
        params.put("SYSTEMBRIEF", "test");

        // Exception
        assertThrows(Exception.class, () -> baseFacadeImpl.selectQuery(
"notFoundQueryName", null, params));
        assertThrows(Exception.class, () -> baseFacadeImpl.selectQuery(
"notFoundQueryName", null, null));

        // Запрос найден и выполняется
        Map<String, Object> queryResult = baseFacadeImpl.selectQuery(
"findedQueryAndFulfilled", null, params);
        assertNotNull(queryResult);
        assertNotNull(queryResult.get("TOTALCOUNT"));
        assertNotNull(queryResult.get("Result"));
    }
}
```

Это означает, что при выполнении теста selectQuery наша бд будет иметь те, данные, которые мы зарегистрировали в файле **BaseFacadeTest.yml**

8.3.3 Тестирование библиотеки (на примере queyenne и bivaspect)

Самое простое тестирование, которое придется выполнять - тестирование утилитного функционала. Оно просто тем, что кроме как подключение аннотации @Test ничего больше не нужно. Такой подход очень прост.

Ниже представлен утилитный класс из модуля **queyenne**:

```
@Tag(value = "unit")
class ConversionUtilTest {

    @Test
    void toInt() {
        // from NUMER
        assertEquals(1, ConversionUtil.toInt(1, 100));
        // from String
        assertEquals(1, ConversionUtil.toInt("1", 100));
        // get default value
        assertEquals(100, ConversionUtil.toInt(null, 100));
        assertEquals(100, ConversionUtil.toInt(Boolean.FALSE, 100));
    }

    @Test
    void toBoolean() {
        assertTrue(ConversionUtil.toBoolean(1));
        // with js implementation... Example "if (t)"
        assertTrue(ConversionUtil.toBoolean(-100));
        assertTrue(ConversionUtil.toBoolean(true));
        assertFalse(ConversionUtil.toBoolean(false));
        assertFalse(ConversionUtil.toBoolean(null));
    }

    @Test
    void toBigDecimal() {
        assertNull(ConversionUtil.toBigDecimal(null));
        assertEquals(1, ConversionUtil.toBigDecimal(BigInteger.valueOf(1L)).intValue(
));
        assertEquals(1, ConversionUtil.toBigDecimal(1).intValue());
        assertThrows(RuntimeException.class,
                () -> ConversionUtil.toBigDecimal("123"),
                "Can't convert to BigDecimal: 123"
        );

    }

    @Test
    void toComparable() {
        assertNull(ConversionUtil.toComparable(null));
```

```
        assertNotNull(ConversionUtil.toComparable("123"));
        assertNotNull(ConversionUtil.toComparable(new char[]{'1', '2', '3'}));
        assertNotNull(ConversionUtil.toComparable(new StringBuffer()));
        assertThrows(ClassCastException.class, () -> ConversionUtil.toComparable(new
Object()));
    }

    @Test
    void testToString() {
        assertNull(ConversionUtil.toComparable(null));
        assertEquals("123", ConversionUtil.toString("123"));
        assertEquals("123", ConversionUtil.toString(new char[]{'1', '2', '3'}));
        assertEquals("123", ConversionUtil.toString(new StringBuffer().append("123"))
);
        assertThrows(ClassCastException.class, () -> ConversionUtil.toString(new
Object()));
    }

    @Test
    void toUpperCase() {
        assertNull(ConversionUtil.toUpperCase(null));
        assertEquals("HELLO TEST WORLD", ConversionUtil.toUpperCase("Hello test world
"));
        assertEquals("HELLO", ConversionUtil.toUpperCase(new char[]{'h', 'e', 'L', 'L
', 'o'}));
        assertEquals("HELLO TEST WORLD", ConversionUtil.toUpperCase(new StringBuffer(
).append("Hello test world")));
        Object object = new Object();
        assertEquals(object, ConversionUtil.toUpperCase(object));
    }
}
```

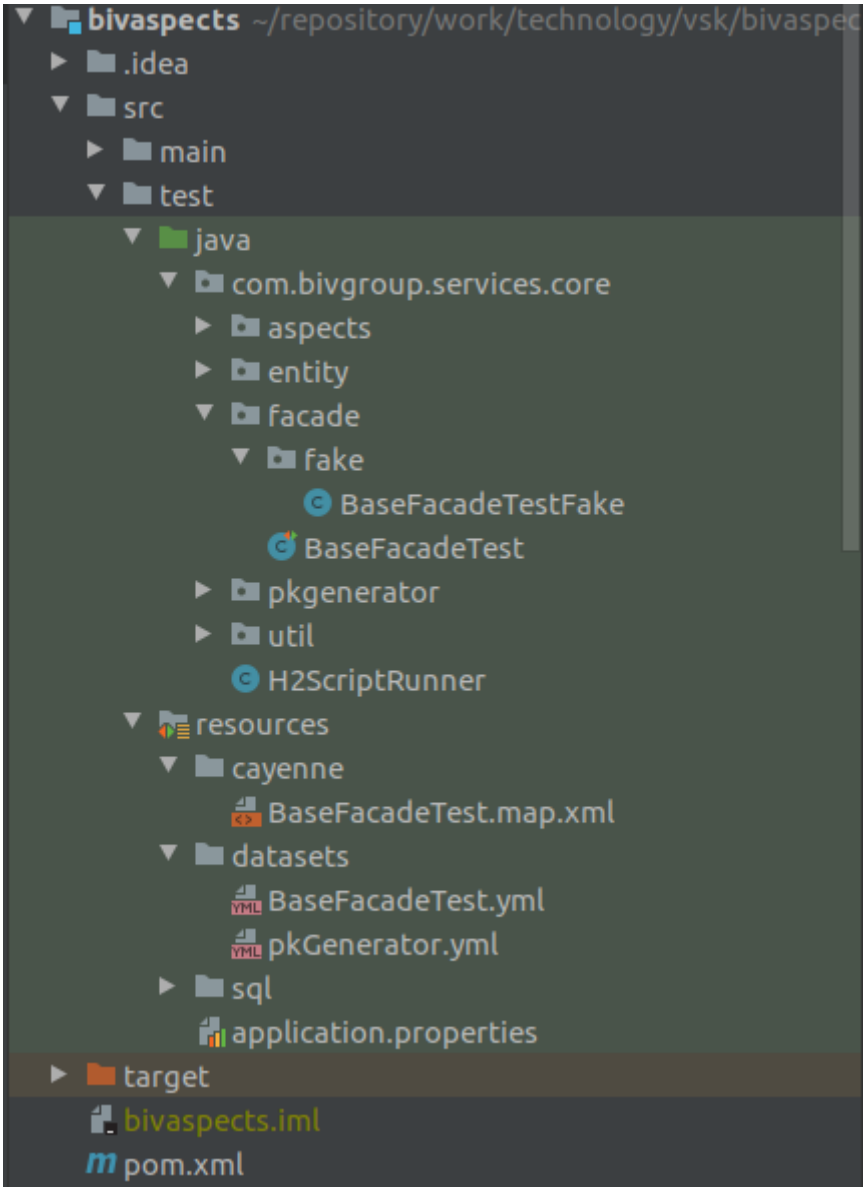
При тестировании модуля **bivaspect** для большинства тестов потребуется использовать несколько механизмов, а именно:

- Создание таблиц с использованием RunScript
- Наполнением таблиц с помощью Database Rider

Такой набор обусловлен тем, что у нас оно используется работу через **queyenne**, это означает, что таблицы не могут быть созданы автоматически, как бы они создавались, например, через JPA. Database Rider помогает нам наполнить таблицы данными (можно было не использовать этот механизм, а выполнять скрипты вручную, но, мне кажется, что такой подход более правильный и легко читаемый). Любой человек, который не знает SQL сможет с ним работать без особого труда.

Возьмем кейс, при котором будем проверять, как отрабатывает механизм запросов в бд.

Структура выглядит следующим образом:



Объектом тестирования здесь выступает класс **BaseFacadeTestFake**. Он представляет из себя фейк объект, который унаследован от класса, он выглядит следующим образом:

```
/**
 * Фейк для симуляции готового бина фасада.
 */
@ApplicationScoped
@BOName(value = "BaseFacadeTestFake")
public class BaseFacadeTestFake extends BaseFacade {

    @PostConstruct
    public void init() {
        if (queyenne != null) {
            QueyenneProject qProject = new QueyenneProject();
            qProject.loadDataMap("cayenne/BaseFacadeTest.map.xml");
            this.queyenne.setQueyenneProject(qProject);
        }
    }
}
```

Все что он делает, лишь переопределяет логику класса **BaseFacade** заменяя загруженные запросы на свои.

Кроме этого, добавлены дополнительные файлы:

- Файлы по созданию и удалению таблицы для тестирования BaseFacadeTest_create.sql и BaseFacadeTest_delete.sql
- Наполнение бд данными BaseFacadeTest.yml
- Регистратор запросов BaseFacadeTest.map.xml

Файл создания талицы (BaseFacadeTest_create) , выглядит так:

```
create table BaseFacadeTest
(
    ID      NUMBER(19)    not null
        primary key
);
```

А удаления (BaseFacadeTest_delete):

```
DROP table IF EXISTS BaseFacadeTest;
```

Наполнение данными (BaseFacadeTest.yml):

```
BaseFacadeTest:
  - ID: 1
  - ID: 2
  - ID: 3
  - ID: 4
  - ID: 5
  - ID: 6
  - ID: 7
  - ID: 8
  - ID: 9
  - ID: 10
  - ID: 11
  - ID: 122176bec92cd3
  - ID: 13
  - ID: 14
  - ID: 15
  - ID: 16
  - ID: 17
  - ID: 18
  - ID: 19
```

Тестирование выглядит следующим образом:

```
@QuarkusTest
@DBUnitInterceptor
@DataSet(value = "BaseFacadeTest.yml",
cleanBefore = true,
cleanAfter = true
)
class BaseFacadeTest {

    private static H2ScriptRunner h2ScriptRunner;

    @Inject
    @BOName(value = "BaseFacadeTestFake")
    BaseFacade baseFacadeImpl;

    @BeforeAll
    static void beforeAll() throws SQLException {
        h2ScriptRunner = new H2ScriptRunner();
        h2ScriptRunner.runSQLFile("sql/other/BaseFacadeTest_create.sql");
    }

    @AfterAll
    static void afterAll() throws SQLException {
        h2ScriptRunner.runSQLFile("sql/other/BaseFacadeTest_delete.sql");
    }

    @Test
    void selectQuery() throws Exception {
        Map<String, Object> params = new HashMap<>();
        params.put("SYSTEMBRIEF", "test");

        // Exception
        assertThrows(Exception.class, () -> baseFacadeImpl.selectQuery(
"notFoundQueryName", null, params));
        assertThrows(Exception.class, () -> baseFacadeImpl.selectQuery(
"notFoundQueryName", null, null));

        // Запрос найден и выполняется
        Map<String, Object> queryResult = baseFacadeImpl.selectQuery(
"findedQueryAndFulfilled", null, params);
        assertNotNull(queryResult);
        assertNotNull(queryResult.get("TOTALCOUNT"));
        assertNotNull(queryResult.get("Result"));
    }
}
```

Алгоритм работы данного кейса следующий:

- @QuarkusTest говорит о том, что для текущего класса требуется поднять контекст quarkus

- С помощью квалификатора (который является фейк объектом) внедряется нужный нам бин @Inject @BOName(value = "BaseFacadeTestFake") BaseFacade baseFacadeImpl;
- Перед тем как выполнять тесты, создаются нужные таблицы в методе beforeAll
- Каждый раз, как у нас выполняется тест, он наполняется данными через @DBUnitInterceptor @DataSet(value = "BaseFacadeTest.yml", cleanBefore = true, cleanAfter = true)
- После выполнения всех тестов,удаляются нужные таблицы в методе afterAll

8.3.4 Тестирование сервиса (на примере tarificator)

Данный модуль необходимо упомянут потому, что он не работает с библиотекой queyenne, а основан на одной из имплементаций JPA, а именно на quarkus-hibernate-orm-panache. Это тоже расширение, которое предоставляется quarkus. Отличие тестирования данного модуля от bivaspect отличается тем, что нам не нужно (необязательно пользоваться ScriptRunner из h2)

Именно здесь библиотека DataBase Rider показывает свою полезность. Из-за того, что у нас используется ORM необходимости в создании таблиц с помощью скриптов у нас отсутствует. Поэтом нам нужно только наполнять ее нужными данными.

Наполнение данными в этом разделе рассматривать не будем, т.к. полный скрипт составляет 13 тысяч строк

```
@QuarkusTest
@DBUnitInterceptor
public class CalculatorTest {
@Inject
EntityManager entityManager;

    @Test
    @DataSet(value = "familyAssetV6.yml")
    public void calculationFamilyAssetTest() throws Exception {
        Calculator calc = Calculator.find("name", "Calculator.FamilyAsset")
.firstResult();
        Assertions.assertNotNull(calc);
        CalculatorVersion calculatorVersion = calc.insCalcVer;
        Assertions.assertNotNull(calculatorVersion);
        CalculatorEngine engine = new CalculatorEngine(entityManager,
calculatorVersion);
        String jsonParamStr = "{\n" +
                "  \"isInsAmFix\": 1,\n" +
                "  \"RISKLIST\": [\n" +
                "    {\n" +
                "      \"SYSNAME\": \"FCC_MAIN_PROGRAM_2\",\n" +
                "      \"PRODSTRUCTID\": 343103,\n" +
                "      \"INSAMVALUE\": 10000000,\n" +
                "      \"isSelected\": true,\n" +
                "      \"ISSELECTED\": 1,\n" +
```

```
"      \"UWCOEF1\": 0,\n" +
"      \"UWCOEF2\": 0\n" +
"    },\n" +
"    {\n" +
"      \"SYSNAME\": \"FCC_EXEMPTION_PAYMENT_DISAB\",\n" +
"      \"PRODSTRUCTID\": 343113,\n" +
"      \"INSAMVALUE\": 5279476.25,\n" +
"      \"isSelected\": true,\n" +
"      \"ISSELECTED\": 1,\n" +
"      \"UWCOEF1\": 0,\n" +
"      \"UWCOEF2\": 0\n" +
"    }\n" +
"  ],\n" +
"  \"gender\": \"ж\",\n" +
"  \"CALCVERID\": 343006,\n" +
"  \"ELEMRISKLIST\": [\n" +
"    {\n" +
"      \"SYSNAME\": \"FAMALYASSETS_LIFE_OF_TERM\",\n" +
"      \"isSelected\": true,\n" +
"      \"ISSELECTED\": 1,\n" +
"      \"UWCOEF1\": null,\n" +
"      \"UWCOEF2\": null\n" +
"    },\n" +
"    {\n" +
"      \"SYSNAME\": \"FAMALYASSETS_DEATH_LIFE\",\n" +
"      \"isSelected\": true,\n" +
"      \"ISSELECTED\": 1,\n" +
"      \"UWCOEF1\": null,\n" +
"      \"UWCOEF2\": null\n" +
"    },\n" +
"    {\n" +
"      \"SYSNAME\": \"FAMALYASSETS_DIAGNOSIS_HIGH_RISK_IC\",\n" +
"      \"isSelected\": null,\n" +
"      \"ISSELECTED\": 1,\n" +
"      \"UWCOEF1\": null,\n" +
"      \"UWCOEF2\": null\n" +
"    },\n" +
"    {\n" +
"      \"SYSNAME\": \"FAMALYASSETS_DEATH_DUE_ACC\",\n" +
"      \"isSelected\": null,\n" +
"      \"ISSELECTED\": 1,\n" +
"      \"UWCOEF1\": null,\n" +
"      \"UWCOEF2\": null\n" +
"    },\n" +
"    {\n" +
"      \"SYSNAME\": \"FAMALYASSETS_DEATH_DUE_ACC_TRANS\",\n" +
"      \"isSelected\": null,\n" +
"      \"ISSELECTED\": 1,\n" +
"      \"UWCOEF1\": null,\n" +
"      \"UWCOEF2\": null\n" +
"    },\n" +
```

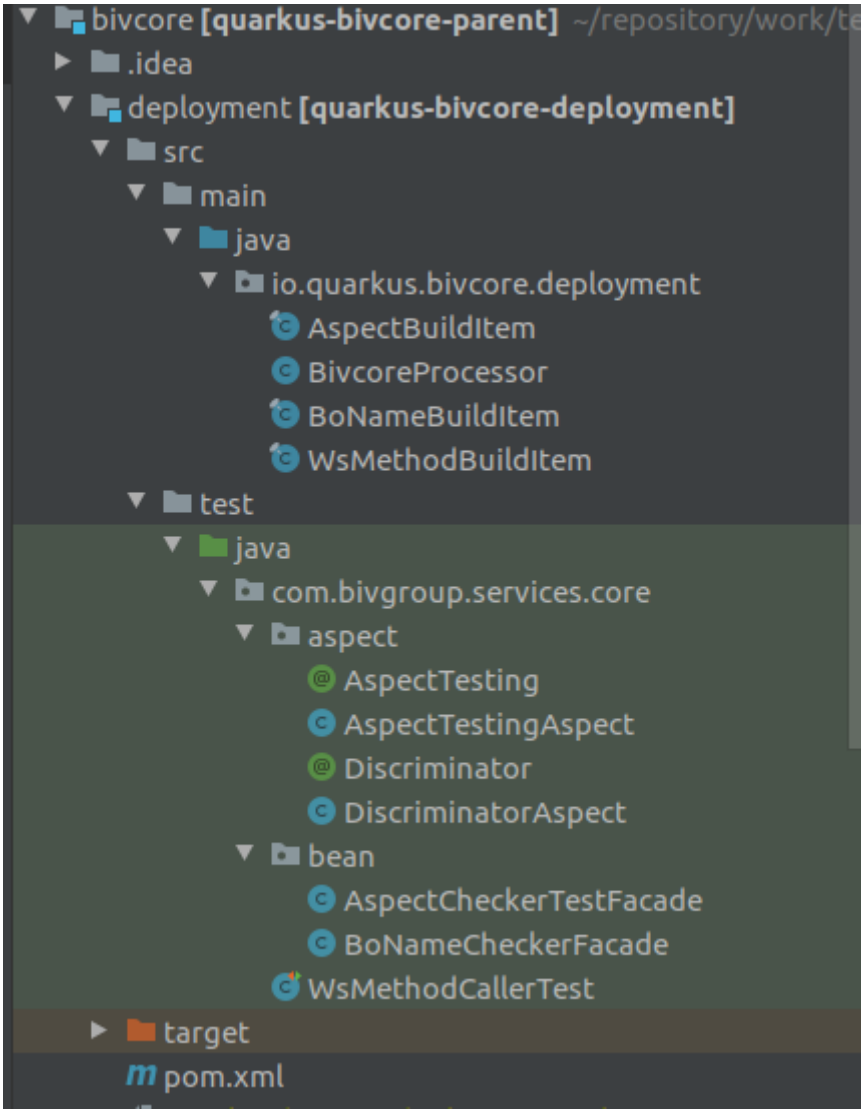
```
                "    {\n" +
                "      \"SYSNAME\": \"FAMALYASSETS_DISABILITY_IA\",\n" +
                "      \"isSelected\": null,\n" +
                "      \"ISSELECTED\": 1,\n" +
                "      \"UWCOEF1\": null,\n" +
                "      \"UWCOEF2\": null\n" +
                "    },\n" +
                "    {\n" +
                "      \"SYSNAME\": \"FAMALYASSETS_DISABILITY_ACCIDENT2\",\n" +
                "      \"isSelected\": null,\n" +
                "      \"ISSELECTED\": 1,\n" +
                "      \"UWCOEF1\": null,\n" +
                "      \"UWCOEF2\": null\n" +
                "    },\n" +
                "    {\n" +
                "      \"SYSNAME\": \"FAMALYASSETS_DISABILITY_ACCIDENT3\",\n" +
                "      \"isSelected\": null,\n" +
                "      \"ISSELECTED\": 1,\n" +
                "      \"UWCOEF1\": null,\n" +
                "      \"UWCOEF2\": null\n" +
                "    },\n" +
                "    {\n" +
                "      \"SYSNAME\": \"FAMALYASSETS_TRAUMA_DUE_ACC_IA\",\n" +
                "      \"isSelected\": null,\n" +
                "      \"ISSELECTED\": 1,\n" +
                "      \"UWCOEF1\": null,\n" +
                "      \"UWCOEF2\": null\n" +
                "    },\n" +
                "    {\n" +
                "      \"SYSNAME\": \"FAMALYASSETS_SURGICAL_DUE_ACC_IA\",\n" +
                "      \"isSelected\": null,\n" +
                "      \"ISSELECTED\": 1,\n" +
                "      \"UWCOEF1\": null,\n" +
                "      \"UWCOEF2\": null\n" +
                "    },\n" +
                "    {\n" +
                "      \"SYSNAME\": \"FAMALYASSETS_DISABILITY_EXEMPTION_PAYMENT\",\n"
+
                "      \"isSelected\": true,\n" +
                "      \"ISSELECTED\": 1,\n" +
                "      \"UWCOEF1\": null,\n" +
                "      \"UWCOEF2\": null\n" +
                "    }\n" +
                "  ],\n" +
                "  \"SALE_CHANNEL_NAME\": \"FIRST\",\n" +
                "  \"term\": 30,\n" +
                "  \"currencyId\": 1,\n" +
                "  \"age\": 37,\n" +
                "  \"PAYVAR\": \"QUARTERLY\"\n" +
                "}";
```

```
        ObjectMapper mapper = new ObjectMapper();
        TypeFactory typeFactory = mapper.getTypeFactory();
        MapLikeType mapLikeType = typeFactory.constructMapLikeType(Map.class, String
.class, Object.class);
        Map<String, Object> params = mapper.readValue(jsonParamStr, mapLikeType);
        SimpleDateFormat formatter = new SimpleDateFormat("dd.MM.yyyy");
        params.put("STARTDATE", formatter.parse("06.02.2020"));
        params.put("BIRTHDATE", formatter.parse("01.01.1983"));
        params.put("FINISHDATE", formatter.parse("05.02.2050"));
        Map<String, Object> returnParams = engine.calculatePremium(params);
        Assertions.assertEquals(8_778_883.2, (Double) returnParams.get(
"PREMVALUEFORALLTERM"));
        Assertions.assertEquals(1.1391, (Double) returnParams.get("sharePremValue"));
        Assertions.assertEquals(73_157.36, (Double) returnParams.get("PREMVALUE_TOTAL
"));
        Assertions.assertEquals(0.17, (Double) returnParams.get("loadMainProgram"));
        Assertions.assertEquals(0.3, (Double) returnParams.get("loadRider"));
        Assertions.assertEquals(1.0, (Double) returnParams.get("commission"));
        Assertions.assertEquals(0.03, (Double) returnParams.get("gnd"));
        Assertions.assertEquals(18.0, (Double) returnParams.get("AGENTTARIFF"));
        Assertions.assertEquals(82.0, (Double) returnParams.get("liabilitiesPercent")
);
    }
}
```

8.3.5 Тестирование расширения (на примере bivcore)

При тестировании расширения, все тестирование сводится к тому, что необходимо протестировать только модуль **deployment**. Это связано с тем, что расширение в основном предназначено для того, чтобы расширить базовый функционал на уровне байт-кода (это не всегда так, но тем не менее). **Runtime** модуль протестировать все-равно не получится, т.к. он ничего не знает о deployment модуле, гда происходит вся магия работы расширений quarkus.

Структура модуля выглядит следующим образом (с течением времени структура может поменяться):



Если коротко, то пакет **aspect** эмулирует тестовые аспекты (которые похожи на настоящие). В данном случае они необходимы для того, чтобы их можно было навесить на тестовые бины, которые зарегистрированы в пакете **bean**.

Основным классом , который выполняет тесты, в текущем кейсе является класс **WsMethodCallerTest** Код данного класса представлен ниже:

```
class WsMethodCallerTest {

    public static final String STATUS = "Status";
    public static final String OK = "OK";
    public static final String ERROR = "ERROR";

    @Inject
    WsMethodCaller wsMethodCaller;

    @RegisterExtension
    static final QuarkusUnitTest configUnitTest = new QuarkusUnitTest()
            .setArchiveProducer(() -> ShrinkWrap.create(JavaArchive.class)
                    .addClasses(WsMethodCallerTest.class
                            , WsMethodCaller.class
```

```
                            , BoNameCheckerFacade.class
                            , AspectCheckerTestFacade.class
                            , DiscriminatorAspect.class
                            , Discriminator.class
                            , AspectTestingAspect.class
                            , AspectTesting.class
                    )
            );

    @Test
    public void callCheckDiscriminatorMethod() throws Exception {
        Map<String, Object> callOKMethod = wsMethodCaller.executeMethod(
"checkDiscriminator", new HashMap<>());

        assertNotNull(callOKMethod);
        assertNotNull(callOKMethod.get(STATUS));
        assertEquals(callOKMethod.get(STATUS), OK);
        assertNotNull(callOKMethod.get("DISCRIMINATOR"));
        assertEquals(1L, callOKMethod.get("DISCRIMINATOR"));
    }

    @Test
    public void callOKMethod() throws Exception {
        Map<String, Object> callOKMethod = wsMethodCaller.executeMethod("callOKMethod
", new HashMap<>());
        assertNotNull(callOKMethod);
        assertNotNull(callOKMethod.get(STATUS));
        assertEquals(callOKMethod.get(STATUS), OK);
    }

    @Test
    public void callErrorMethod() throws Exception {
        Map<String, Object> callErrorMethod = wsMethodCaller.executeMethod(
"callErrorMethod", new HashMap<>());
        assertNotNull(callErrorMethod);
        assertNotNull(callErrorMethod.get(STATUS));
        assertEquals(callErrorMethod.get(STATUS), ERROR);
    }

    @Test
    public void callNotWsMethod() throws Exception {
        Map<String, Object> callNotWsMethod = wsMethodCaller.executeMethod(
"callNotWsMethod", new HashMap<>());
        assertNotNull(callNotWsMethod);
        assertNotNull(callNotWsMethod.get(STATUS));
        assertEquals(callNotWsMethod.get(STATUS), ERROR);
    }

    @Test
    public void callTestReqParam() throws Exception {
        // no req param
```

```
        Map<String, Object> callNoTestReqParam = wsMethodCaller.executeMethod(
"callTestReqParam", new HashMap<>());
        assertNotNull(callNoTestReqParam);
        assertNotNull(callNoTestReqParam.get(STATUS));
        assertEquals(callNoTestReqParam.get(STATUS), ERROR);
        // with req param
        HashMap<String, Object> params = new HashMap<>();
        params.put("TESTREQPARAM", "NOT_EMPTY");
        Map<String, Object> callNotWsMethod = wsMethodCaller.executeMethod(
"callTestReqParam", params);
        assertNotNull(callNotWsMethod);
        assertNotNull(callNotWsMethod.get(STATUS));
        assertEquals(callNotWsMethod.get(STATUS), OK);
    }

}
```

Тестирование расширений Quarkus должно выполняться с расширением iU.quarkus.test.QuarkusUnitTest JUnit 5. Это расширение позволяет выполнять тесты в стиле Arquillian, которые проверяют определенные функциональные возможности. Он не предназначен для тестирования пользовательских приложений, так как это должно быть сделано через io.quarkus.test.junit.QuarkusTest.

Основное отличие состоит в том, что QuarkusTest просто загружает приложение один раз в начале выполнения, а QuarkusUnitTest развертывает пользовательское приложение Quarkus для каждого тестового класса.

Основное что стоит сказать по этому классу:

- Расширение QuarkusUnitTest должно использоваться со статическим полем. При использовании с нестатическим полем тестовое приложение не запускается.
- QuarkusUnitTest используется для создания приложения для тестирования. Он использует Shrinkwrap для создания JavaArchive для тестирования
- Благодаря QuarkusUnitTest можно внедрить bean-компоненты (в нашем случае это wsMethodCaller) из нашего тестового развертывания непосредственно в тестовый набор.

Для того, чтобы проверить вызовы был создан бин BoNameCheckerFacade, который удовлетворяет требованиям расширения.