ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ RUBBLES APPLICATION DEVELOPMENT FRAMEWORK

Инструкция пользователя

Листов 61

2022

Аннотация

В настоящем документе приведена инструкция пользователя программного обеспечения Rubbles Application Development Framework (технологической платформы для создания бизнес-приложений).

Содержание

Аннотация 2

Содержание 3

Перечень терминов и сокращений 5

1. Введение 7
   1.1. Область применения 7
   1.2. Уровень подготовки пользователя 7
   1.3. Перечень эксплуатационной документации, с которой необходимо ознакомиться пользователю 7

2. Программное и техническое обеспечение 8

3. Инициализация проекта 9
   3.1. Установка фреймворка 9
   3.2. Создание и запуск приложения с использованием фреймворка 9
   3.3. Запуск фронтенда и бэкенда через docker-compose 11

4. Проектирование навигационной модели приложения 13
   4.1. Создание структуры меню 14
   4.2. Создаем страницы и привязываем их к пунктам меню 15

5. Проектирование ролевой модели приложения 17
   5.1. Создание пользователя с ролью «Администратор» и подключение «Административной панели» 17
   5.2. Создание новых ролей 18
   5.3. Внедрение проверки доступа к страницам на основе ролей 18
   5.4. Создание тестовых пользователей и делегирование им прав 20

6. Проектирование модели данных приложений (БД) 23
   6.1. Пример настройки миграции с помощью flyway 23

7. Реализация CRUD-менеджеров (создание CRUD-сущностей) 27
   7.1. Утилита crudcp.py 27
   7.2. Расширение базовой функциональности 28
   7.3. Корректировка State 29
   7.4. Корректировка Vacancy 30
   7.5. Корректировка Candidate 31
   7.6. Корректировки VacancyRelation и StateRelation 33

8. Ролевая модель (горизонтальная) 35
   8.1. Вакансии 35
   8.2. Кандидаты 36

9. Проектирование событийной модели приложения 38
   9.1. Создание события 38

9.2. Оформление события в бизнес-логику 40
9.3. Добавление заготовки обработчика событий 41

10. Проектирование модели нотификаций пользователей 44
10.1 Настройки приложения 44
10.2. Встроенные подписки - декларативная настройка отправки уведомлений 44
10.3. Уточнение релевантности уведомления 46

11. Проектирование титульного дашборда 49
11.1.Подготовка витрины данных 49
11.2. Datasource, Dataset, Monitor + Сборка страницы 50
11.3. Реализация кастомного расчетного показателя для мониторов 53
11.4. Реализация недостающих вспомогательных методов сервисов 54
11.5. Генерация тестовых данных 56

12. CRUD 59
12.1. Базовый функционал 59
12.2. Оператор присоединения (join) 60
12.3. Фильтры 60
12.4. Деревья 61

14. Спецификация пользовательского интерфейса 63

Перечень терминов и сокращений

Сокращение	Полное наименование
БД	База данных
Бэкенд	(англ. – back-end) – программно-аппаратная часть сервиса, отвечающая за функционирование его внутренней части
Виджет	Элемент интерфейса – примитив графического интерфейса пользователя, имеющий стандартный внешний вид и выполняющий стандартные действия
Дашборд	Интерактивная информационная панель, которая наглядно представляет, визуализирует, объясняет и анализирует данные
Нода	Узел в системе связанных объектов
Предикат	Выражение, использующее одну или более величин с результатом логического типа
Приложение, бизнес-приложение	Продукт, который реализуется с помощью Rubbles Application Development Framework
Проект	Процесс реализации приложения либо кодовая база конкретной системы
Сиквенс	Структура для генерации уникальных целочисленных значений
Сокет	Название программного интерфейса для обеспечения обмена данными между процессами
Фреймворк	(англ. – framework) – программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного продукта
Фронтенд	(англ. – front-end) – клиентская сторона пользовательского интерфейса к программно-аппаратной части сервиса
Эндпоинт	(англ. — конечная точка) - шлюз, который соединяет серверные процессы приложения с внешним интерфейсом, то есть адрес, на который отправляются сообщения
Appix	Внутреннее наименование Rubbles Application Development Framework
CRUD	Акроним, обозначающий четыре базовые функции, используемые при работе с базами данных: создание (англ. create), чтение (read), модификация (update), удаление (delete)

Сокращение	Полное наименование
DevOps Engineer	Специалист, обеспечивающий автоматизацию процесса разработки продукта
Entity-модель	Модель объектов фреймворка
FrontEnd Developer	Специалист, который создает пользовательские интерфейсы и отвечает за всю внешнюю часть сайта или приложения, с которой взаимодействуют пользователи
Keycloak	Продукт с открытым кодом для реализации single sign-on с возможностью управления доступом, нацелен на современные приложения и сервисы
QA Engineer	Специалист, который создаёт сценарии тестирования, прогнозирует сбои и находит ошибки в продуктах (тестировщик)

1. Введение

1.1. Область применения

Rubbles Application Development Framework – это фреймворк высокого уровня абстракции, построенный поверх Spring Boot и предназначенный для разработки типовых бизнес-приложений. Основная цель фреймворка – ускорить и удешевить процесс разработки бизнес-приложения за счет предоставления готовых, протестированных и проинтегрированных между собой высокоуровневых компонентов и сервисов, образующих консистентную платформу.

Rubbles Application Development Framework позволяет сократить затраты на разработку типовых бизнес-приложений, но не в ущерб качеству итогового продукта: предоставляемые фреймворком решения выполнены с соблюдением лучших практик и становятся частью кодовой базы конечного продукта, открывая возможность дальнейшего развития этой кодовой базы вне зависимости от фреймворка и плавного перехода к полноценной командной разработке.

При разработке типового бизнес-приложения Rubbles Application Development Framework позволяет обойтись без привлечения таких ролей как:

- FrontEnd Developer.
- DevOps Engineer.
- QA Engineer.

1.2. Уровень подготовки пользователя

Пользователь Rubbles Application Development Framework должен иметь опыт работы с языками программирования Java/Kotlin и компонентами Spring Framework.

1.3. Перечень эксплуатационной документации, с которой необходимо ознакомиться пользователю

Перед началом работы пользователю необходимо ознакомиться с настоящей инструкцией пользователя.

2. Программное и техническое обеспечение

Таблица 2.1 Требования к программному и техническому обеспечению

Требование	Минимальные требования	Рекомендуемые требования
ОЗУ	2 GB свободной оперативной памяти	8 ГБ общей системной оперативной памяти
Центральный процессор	Любой современный процессор	Многоядерный процессор IntelliJ IDEA, который поддерживает многопоточность для различных операций и процессов
Дисковое пространство	2,5 Гб и еще 1 ГБ для кэшей	Твердотельный накопитель с объемом свободного места не менее 5 Гб
Разрешение монитора	1024×768	1920×1080
Операционная система	Официально выпущенные 64-разрядные версии: Microsoft Windows 8 или более поздней версии macOS 10.14 или более поздней версии. Любой дистрибутив Linux, поддерживающий Gnome, KDE или Unity DE. Предварительные версии не поддерживаются	Последняя 64-разрядная версия Windows, macOS или Linux (например, Debian, Ubuntu или RHEL)

3. Инициализация проекта

3.1. Установка фреймворка

1) Установить через любой пакетный менеджер opdejdk-11.
2) Установить через любой пакетный менеджер maven.
3) Разархивировать исходные коды библиотеки из архива appix-rosres.zip.
4) Перейти в директорию `/home/devop/appix/appix/backend` и выполнить `./gradlew publishToMavenLocal`
5) После сборки и публикации проекта в директории `/home/devop/.m2/repository/com/sbdagroup` должны появиться сборки библиотек с актуальными версиями компонент.

3.2. Создание и запуск приложения с использованием фреймворка

Для создания демонстрационного приложения необходимо выполнить следующие действия:

1) Создать новый Spring-boot проект с помощью IntelliJ IDEA или Spring Initializr. При этом в IDEA проект можно создать через пункт Spring Initializr. Никаких дополнительных модулей спринга для простого приложения выбирать не требуется.

2) В качестве языка разработки выбрать Kotlin и систему сборки Gradle. Обратите внимание, что должна быть установлена минимальная версия Java 11 и рекомендуемая (проверенная) версия Spring Boot 2.6.1.

3) Документация соответствует (с незначительными изменениями в конфигурационных файлах) пунктам создания демо приложения “HrHero” с использованием фреймворка Rubbles Application Development Framework и опубликована по адресу https://vue.rosres.rubbles-crm.ru/content/docs.

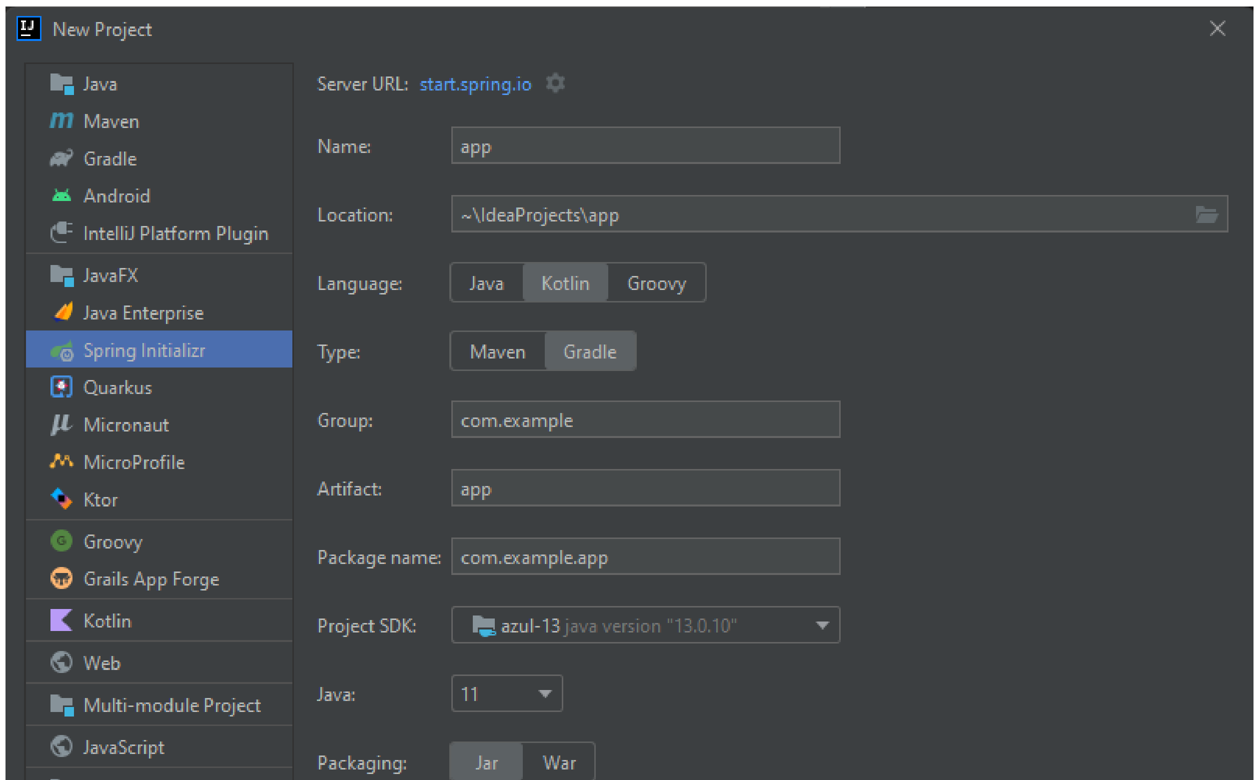

Рис. 3.1 – Создание проекта через пункт Spring Initializr

3) Добавить в `build.gradle.kts` репозиторий с appix:

```
repositories {
      mavenLocal() // Локальный maven репозиторий, в котором мы опубликовали компоненты Appix
    mavenCentral()
}
```

4) Подключить ядро фреймворка и другие зависимости и добавить в блок dependencies файла `build.gradle.kts`:

```
plugins {
    id("org.springframework.boot") version "2.6.1"
}

dependencies {
    implementation("org.jetbrains.kotlin:kotlin-reflect")
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk8")
    implementation("org.springframework.boot:spring-boot-starter:2.6.1")
    implementation("org.springframework.boot:spring-boot-starter-web:2.6.1")

implementation("org.springframework.boot:spring-boot-starter-data-jpa:2.6.1")

testImplementation("org.springframework.boot:spring-boot-starter-test:2.6.1")

    // Flyway support
    implementation("org.flywaydb:flyway-core:8.0.2")

    // Json support
    implementation("com.fasterxml.jackson.module:jackson-module-kotlin:2.13.0")
```

```
    // Appix Core
    implementation("com.sbdagroup:appix-core:0.0.32-SNAPSHOT")
}
```

5) Перейти в `main` класс spring-boot приложения и добавить аннотацию `@EnableAppix`.

```
package com.example.app

import appix.core.config.context.EnableAppix      // Импортируем аннотацию из
ядра
import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

@EnableAppix                // Навешиваем на класс Application
@EntityScan
@EnableJpaRepositories
@SpringBootApplication
class Application

fun main(args: Array<String>) {
    runApplication<Application>(*args)  // Запускаем приложение
}
```

6) Запустить `main application` класс в IDEA.

7) Проверить, что backend запустился и работает:

```
> curl 'http://localhost:8080/'
---------------------------------------------------

< StatusCode        : 200
< Response          : {"code":"ok","result":{...}}
```

3.3. Запуск фронтенда и бэкенда через docker-compose

Для запуска демонстрационного приложения через docker-compose необходимо выполнить следующие действия:

1) Перейти в директорию `/home/devop/appix/appix/appix-demo` и выполнить сборку приолжения командой `./gradlew -x test clean build`
2) Создать `docker-compose.yml` файл и описать frontend и backend сервисы:

```
version: '3.3'
services:

  #####  APPIX-DEMO  #####
  appix-demo-backend:
    container_name: appix-demo-backend
    build: ./appix/appix-demo
    restart: always
    environment:
      - SERVER_PORT=8080
      - SPRING_APPLICATION_NAME=appix-demo
      - BACKEND_PUBLIC_URL=appix-demo-backend.rosres.rubbles-crm.ru
    ports:
      - "8085:8080"
```

```
  appix-demo:
    container_name: appix-demo
    build:
      context: ./appix/frontend/vue1
      args:
        BACKEND_PUBLIC_URL: appix-demo-backend.rosres.rubbles-crm.ru
        VUE_PUBLIC_URL: appix-demo.rosres.rubbles-crm.ru
    restart: always
    ports:
      - "8086:80"
```

Frontend часть приложения собирается непосредственно из frontend компонент библиотеки Appix и не требует внесения дополнительных изменений в код Appix.

`Frontend компоненты расположены в директории /home/devop/appix/appix/frontend/vue1`

3) Запустить контейнер. Для этого в том каталоге, где был создан файл docker-compose.yml, выполнить: `docker-compose up -d`
4) Открыть в браузере адрес https://appix-demo.rosres.rubbles-crm.ru/content/auth ввести параметры тестового пользователя `login:root password:12345` и проверить, что авторизация произойдет успешно и откроется главное меню приложения .
5) На демонстрационном стенде уже создан /home/devop/appix/docker-compose.yml файл для запуска приложения и стенда с документацией. Стенд с документацией запускается неспосредственно из компонент библиотеки Appix.

4. Проектирование навигационной модели приложения

При первом запуске навигационная структура выглядит следующим образом:

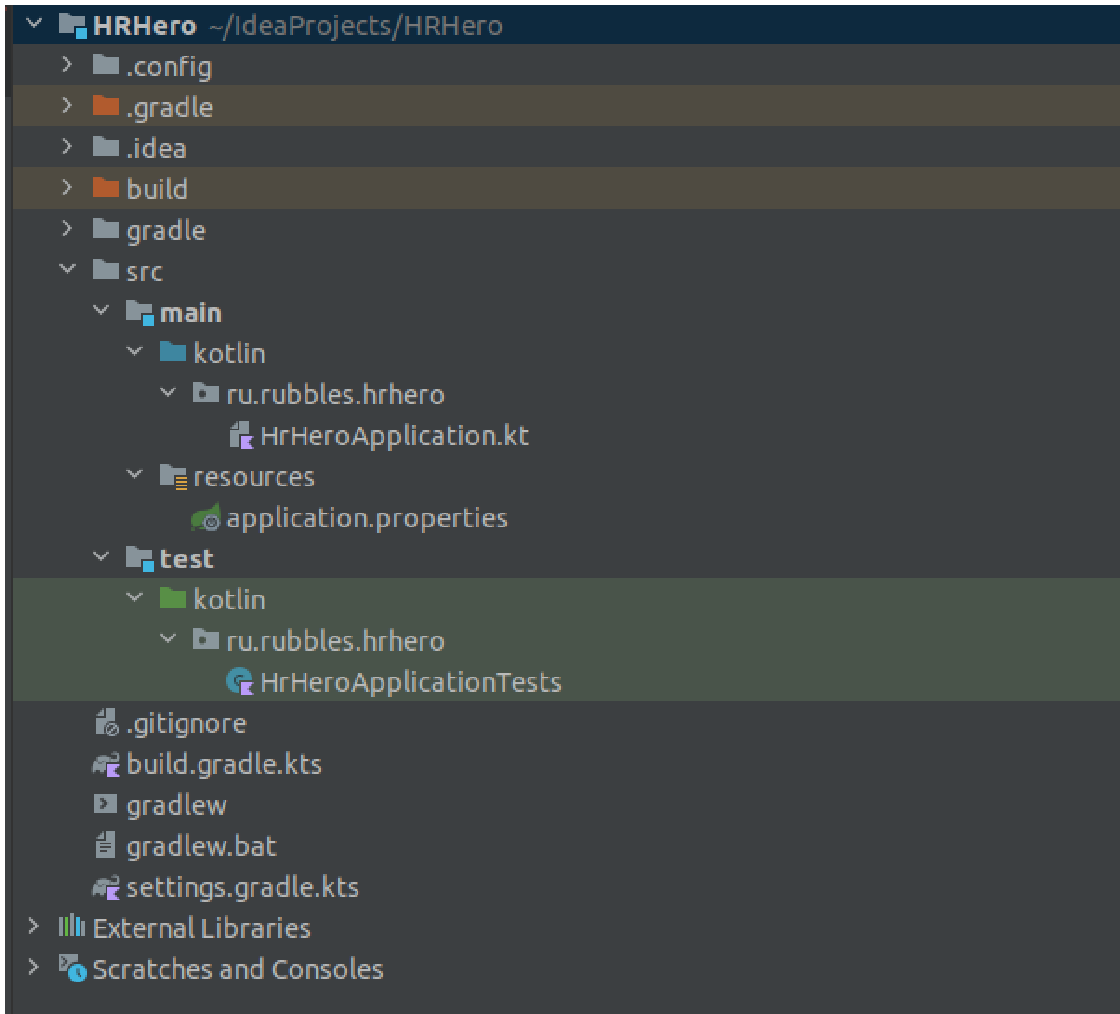

Рис. 4.1 – Навигационная структура (пример)

Стартовая страница выглядит следующим образом:

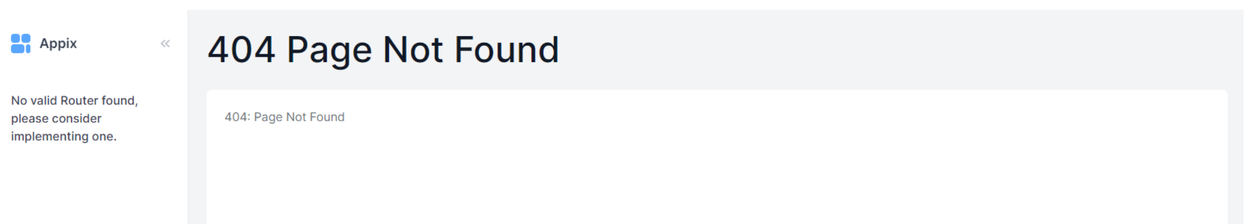

Рис. 4.2 – Стартовая страница

Структура сайта при первом запуске отсутствует, также отсутствуют страницы, поэтому титульная (стартовая) страница отображает ошибку 404.

4.1. Создание структуры меню

Мы планируем создать всего 3 пункта меню:

- Картина дня - тут будет дашборд, показывающий состояние нашей системы в цифрах и графиках
- Вакансии - тут будет раздел, где мы управляем списком вакансий
- Кандидаты - тут будет раздел, где мы ведем учет кандидатов по той или иной вакансии

Какое-то время мы не будем думать о содержании этих страниц и сосредоточимся только на общей структуре приложения. Чтобы создать свое меню нам понадобится создать класс новый класс компонент (`@Component`) `HrHeroRouter`, унаследовав его от класса `appix.core.ui.router.BaseRouter` и реализовать в нем метод `getAppMenu()`.

Создадим пакет ui и разместим новый класс в нем:

```
package ru.rubbles.hrhero.ui

import appix.core.ui.layout.Menu
import appix.core.ui.layout.MenuItem
import appix.core.ui.router.BaseRouter
import org.springframework.stereotype.Component

@Component
class HrHeroRouter : BaseRouter() {
    override fun getAppMenu(): Menu {
        return Menu(
            listOf(
                MenuItem(
                    title = "Картина дня",
                    path = "/",
                ),
                MenuItem(
                    title = "Вакансии",
                    path = "/vacancies",
                ),
                MenuItem(
                    title = "Кандидаты",
                    path = "/candidates",
                )
            )
        )
    }
}
```

Перезапускаем наше приложение и видим только что созданное меню:

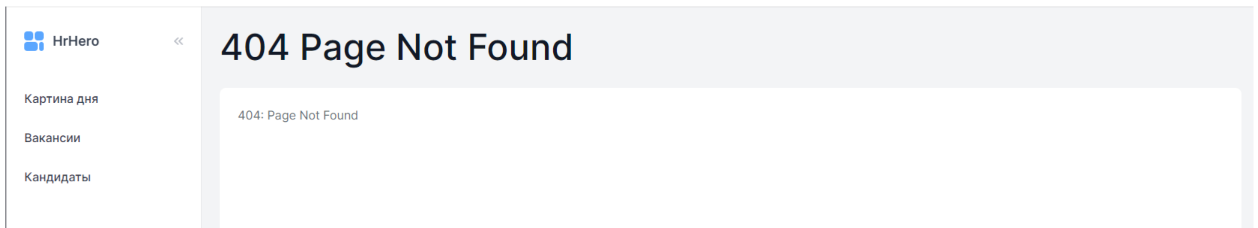

Рис. 4.3 – Стартовая страница

Теперь мы можем передвигаться по меню, но везде выдается заглушка, так как страниц мы все еще не создали.

4.2. Создаем страницы и привязываем их к пунктам меню

Создание страницы осуществляется с помощью создания компонента путем наследования от абстрактного класса `PageBuilder`.

Создадим новый пакет ui.pages и в нем классы `DashboardPage`, `VacancyPage` и `CandidatePage`. Приведем пример класса `DashboardPage`, остальные классы создаются по аналогии.

```
package ru.rubbles.hrhero.ui.pages

import appix.core.ui.layout.Page
import appix.core.ui.layout.PageBuilder
import org.springframework.stereotype.Component

@Component
class DashboardPage : PageBuilder(
    title = "Главная страница",
    public = true,                    // Пока будем делать все страницы доступными для всех, даже без авторизации
) {
    override fun build(authorization: String?): Page {
        return Page(
            pageBuilder = this
        )
    }
}
```

Теперь подключим наши страницы к нашим пунктам меню:

```
package ru.rubbles.hrhero.ui

@Component
class HrHeroRouter(
    private val dashboardPage: DashboardPage,             // Инжектим наши страницы-компоненты в конструктор роутера
    private val vacancyPage: VacancyPage,
    private val candidatePage: CandidatePage,
) : BaseRouter() {
    override fun getAppMenu(): Menu {
        return Menu(
            mutableListOf(
                MenuItem(
                    title = "Картина дня",
                    path = "/",
```

```
                            pageBuilder = dashboardPage,        // Указываем их в
свойстве pageBuilder класса MenuItem
                      ),
                      MenuItem(
                          title = "Вакансии",
                          path = "/vacancies",
                          pageBuilder = vacancyPage,
                      ),
                      MenuItem(
                          title = "Кандидаты",
                          path = "/candidates",
                          pageBuilder = candidatePage,
                      )
                  )
              )
          }
      }
```

Теперь при переходе на любой из пунктов меню мы видим как меняется заголовок заголовок страницы:

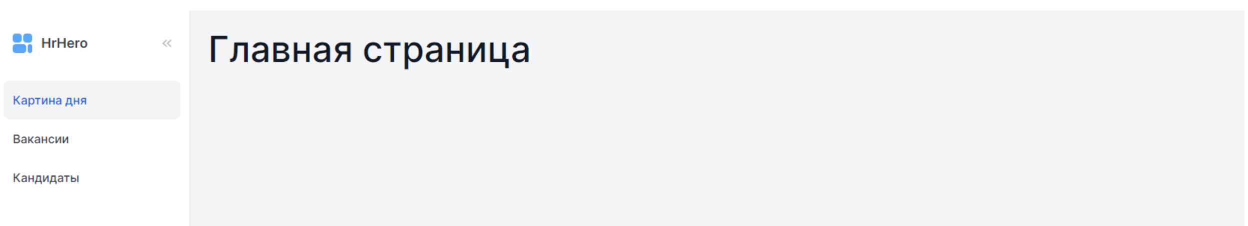

Рис. 4.4 – Стартовая страница

5. Проектирование ролевой модели приложения

В нашей ролевой модели будет 3 роли:

• Админ - встроенная роль `BuiltinRoles.ROOT`. Он будет иметь возможность создавать новых пользователей, создавать и редактировать вакансии, назначать пользователям роли и давать доступ к тем или иным вакансиям.

• Рекрутер - может смотреть справочники вакансий и редактировать таблицу кандидатов, в соответствии с назначенными ему вакансиями.

• Ревьюер - может смотреть справочники вакансий и таблицу кандидатов, в соответствии с назначенными вакансиями.

Отдельно мы создадим `permission` на просмотр финансовых ожиданий кандидата, чтобы это право мы могли делегировать независимо от ролей. То есть, наличие роли Ревьюер само по себе не будет означать возможность видеть финансовые ожидания.

5.1. Создание пользователя с ролью «Администратор» и подключение «Административной панели»

Роль Администратор специально создавать не требуется, она доступна "из коробки". Но чтобы ей полноценно воспользоваться нам необходимо включить встроенную админку, где администратор сможет создавать новых пользователей и делегировать им права.

Для этого надо:

1) Подключаем модуль appix-admin как зависимость к нашему проекту

```
implementation("com.sbdagroup:appix-admin:0.0.10-SNAPSHOT")
```

2) Включаем админку в `application.properties`, она будет находиться по адресу `/content/auth`:

```
appix.admin.enabled = true
```

3) Пересобираем проект и авторизуемся за пользователя `root/12345` и видим в нашем меню новый раздел Admin

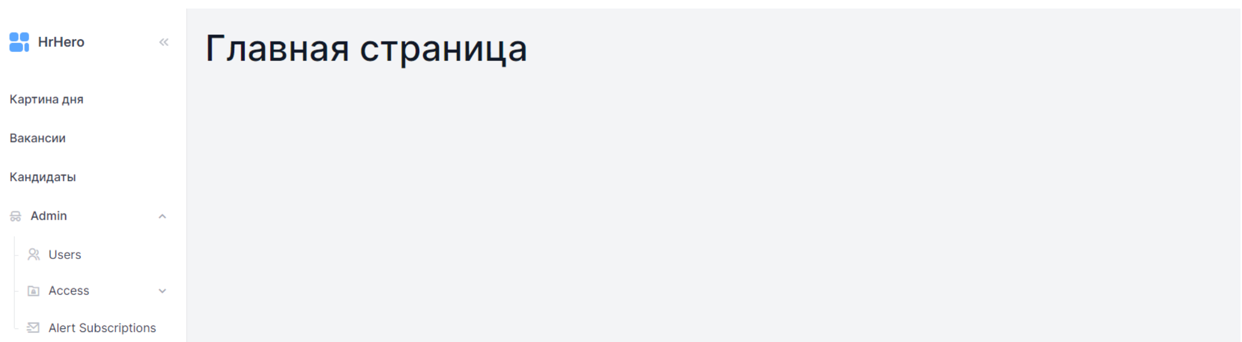

Рис. 5.1 – Раздел Admin

5.2. Создание новых ролей

Добавить в систему новые роли очень просто, для этого всего лишь необходимо создать новый `enum class`, имплементировав интерфейс `AccessEntityType`. Создадим новый пакет acs и поместим в него 2 новых класса:

```
package ru.rubbles.hrhero.acs

import appix.core.acs.AccessEntityType

@AccessEntityType.Extend("HrHeroRoles")
enum class HrHeroRoles(override val alias: String) : AccessEntityType {
    EDITOR("Рекрутер"),
    REVIEWER("Ревьювер"),
}

@AccessEntityType.Extend("HrHeroPermissions")
enum class HrHeroPermissions(override val alias: String) : AccessEntityType {
    FINANCES("Финансовые данные"),
}
```

5.3. Внедрение проверки доступа к страницам на основе ролей

Как мы помним, на текущий момент благодаря параметру public=true все страницы видны всем пользователям, даже если они не авторизованы в системе. Настало время это исправить. Добавим к страницам ограничения по ролям:

```
@Component
class DashboardPage :
    PageBuilder(
        title = "Дашбоард",
        // Вместо параметра public,
        // задаем параметр access и передаем в него список тех ролей, которыми должен обладать пользователь,
        // чтобы доступ был предоставлен:
        access = accessList(BuiltinRoles.ROOT, HrHeroRoles.REVIEWER, HrHeroRoles.EDITOR),
    ) {
    override fun build(authorization: String?): Page {
        return Page(pageBuilder = this)
    }
}

@Component
class CandidatePage : PageBuilder(
    title = "Кандидаты",
    access = accessList(HrHeroRoles.REVIEWER, HrHeroRoles.EDITOR),
) {
    override fun build(authorization: String?): Page {
        return Page(pageBuilder = this)
    }
}

@Component
class VacancyPage : PageBuilder(
```

```
    title = "Вакансии",
    access = accessList(HrHeroRoles.REVIEWER),
) {
    override fun build(authorization: String?): Page {
        return Page(pageBuilder = this)
    }
}
```

Как вы могли заметить, страница с дашбоардом сейчас доступна всем ролям, а страницы "Кандидаты" и "Вакансии" - только ревьюверам. Админ и рекрутер, получается, не имеют к ним доступа.

Как мы понимаем, это не совсем соответствует нашим целям.

Существует 2 способа "персонализации" страниц под разные роли:

1) Динамическое построение страницы в зависимости от роли пользователя, который ее запрашивает

2) Разные предсозданные страницы для разных ролей

В рамках данного курса мы постараемся проиллюстрировать оба способа:

1) Все роли получили доступ к одной и той же страницы "Дашбоарды", а значит элементы "персонализации" нам необходимо реализовать динамически при сборке контента этой страницы - Способ 1

2) Админ получит доступ к "специальной" версии страницы "Вакансии", с расширенными возможностями, а другие роли будут видеть версию страницы "Только для чтения". Эти 2 версии по сути будут реализованы как 2 разные физические страницы - Способ 2

3) Рекрутер получат доступ к "специальной" версии страницы "Кандидаты", которая будет позволять добавлять новых кандидатов - Способ 2

Создаем новые страницы для роли Админ и Рекрутер

Полностью по аналогии с созданными ранее страницами, создаем еще 2 страницы:

```
@Component
class VacancyMngtPage : PageBuilder(
    title = "Вакансии (управление)",
    access = accessList(BuiltinRoles.ROOT),          // Только для админа
) {
    override fun build(authorization: String?): Page {
        return Page(pageBuilder = this)
    }
}

@Component
class CandidateMngtPage : PageBuilder(
    title = "Кандидаты (управление)",
      access = accessList(HrHeroRoles.EDITOR),          // Только для
рекрутера
) {
    override fun build(authorization: String?): Page {
        return Page(pageBuilder = this)
    }
}
```

И поправим роутер:

```
@Component
class HrHeroRouter(
    private val dashboardPage: DashboardPage,
    private val vacancyPage: VacancyPage,
    private val vacancyMngtPage: VacancyMngtPage,
    private val candidatePage: CandidatePage,
    private val candidateMngtPage: CandidateMngtPage
) : BaseRouter() {
    override fun getAppMenu(): Menu {
        return Menu(
            mutableListOf(
                MenuItem(
                    title = "Картина дня",
                    path = "/",
                    pageBuilder = dashboardPage,
                ),
                MenuItem(
                    title = "Вакансии",
                    path = "/vacancies",
                    pageBuilder = vacancyPage,
                ),
                MenuItem(
                    title = "Вакансии",
                    path = "/mngt/vacancies",
                    pageBuilder = vacancyMngtPage,
                ),
                MenuItem(
                    title = "Кандидаты",
                    path = "/candidates",
                    pageBuilder = candidatePage
                ),
                MenuItem(
                    title = "Кандидаты",
                    path = "/mngt/candidates",
                    pageBuilder = candidateMngtPage
                ),
            )
        )
    }
}
```

Итог. Раздел "Картина дня" будет доступна для всех пользователей. Разделы "Вакансии" и "Кандидаты" будут также видны всем ролям, но суть этих страниц будет для них отличаться, в зависимости от роли пользователя. Убедиться в этом можно просто обратив внимание на заголовок этих страниц при переходе на них после выполнения следующего пункта.

5.4. Создание тестовых пользователей и делегирование им прав

Создавать пользователей и делегировать им роли и права можно 2 способами:

• Через админку, используя авторизацию под учетной записью root-пользователя

• С помощью Init-компонента в коде

Так как мы пока не используем настоящую базу данных и не можем расчитывать, что созданные таким образом пользователи переживут перезапуск системы, мы выберем другой путь - создадим пользователей из кода с помощью компонента инициализации

Создадим Init-компонент в корневом пакете приложения:

```
package ru.rubbles.hrhero

@Component
class InitData(
    usersService: BuiltinUsersService,
    acsService: BuiltinAcsService
) {

    init {

        // Создаем пользователей:
        val editor1 = usersService.createOne(
                BuiltinUser(null, "editor1@hrhero.pro", "Рекрутер 1", "editor1")
        )

        val editor2 = usersService.createOne(
                BuiltinUser(null, "editor2@hrhero.pro", "Рекрутер 2", "editor2")
        )

        val reviewer1 = usersService.createOne(
               BuiltinUser(null, "reviewer1@hrhero.pro", "Ревьювер 1", "reviewer1")
        )

        val reviewer2 = usersService.createOne(
               BuiltinUser(null, "reviewer2@hrhero.pro", "Ревьювер 2", "reviewer2")
        )

        // Даем роль рекрутера:
        acsService.give(editor1.toUser(), HrHeroRoles.EDITOR)
        acsService.give(editor2.toUser(), HrHeroRoles.EDITOR)

        // Даем роль ревьювера:
        acsService.give(reviewer1.toUser(), HrHeroRoles.REVIEWER)
        acsService.give(reviewer2.toUser(), HrHeroRoles.REVIEWER)

            // Дополнительно, наделяем 2-ого ревьювера правом просмотра финансовых ожиданий
        acsService.give(reviewer2.toUser(), HrHeroPermissions.FINANCES)
    }
}
```

Теперь можно авторизоваться под созданными пользователями и убедиться в том, что каждый "видит" приложения так, как мы это задумали:

- Admin видит разделы "Картина дня", "Вакансии" и "Кандидаты", причем последние два раздела открываются в режиме "Управление"
- Editor видит разделы "Картина дня" и "Кандидаты", причем последний раздел открывается в режиме "Управление"

• Reviewer видит разделы "Картина дня", "Вакансии" и "Кандидаты", причем последние два раздела открываются в режиме "Просмотр"

6. Проектирование модели данных приложений (БД)

В данной статье мы спроектируем нашу модель данных и реализуем POJO объекты

На самом деле, материал этого урока имеет очень мало отношения к фреймворку Appix. Он нужен только лишь потому, что без этого материала сложно описать процесс создания приложения. Вне зависимости от того, на каком фреймворке реализуется приложение - эта часть в любом случае осталась бы плюс-минус неизменной.

Итак, наша целевая схема модели данных выглядит так:

Рис. 6.1 – Целевая схема модели данных

По умолчанию во фреймворке используется база данных H2 и все таблицы создаются автоматически при описании @Entity модели. Это удобно для быстрого старта разработки первого прототипа.

При желании можно настроить миграции. Для применения миграций в Appix используется Flyway, хотя в целом ничто не мешает вам использовать Liquibase или другую библиотеку.

6.1. Пример настройки миграции с помощью flyway

Прежде всего, Flyway необходимо подключить, если он еще не подключен. Для этого в build.gradle.kts проверяем наличие зависимости:

```
dependencies {

    // ... другие зависимости

    // Flyway support
    implementation("org.flywaydb:flyway-core:8.0.2")
}
```

Теперь создаем папку `resources > db > migration` (стандартное расположение файлов миграции flyway) и помещаем туда файл

`V0_<цифра>__<любое_имя>.sql`, например `V1_1__hrhero_init.sql` (начинаем с 1 так как 0 зарезервирован ядром фреймворка):

```
CREATE TABLE ru_rubbles_hrhero_model_vacancy
(
    id   bigint primary key,
    name varchar(255),
    url varchar(512)
);
create sequence ru_rubbles_hrhero_model_vacancy_id_seq start with 1;

CREATE TABLE ru_rubbles_hrhero_model_state
(
    id   bigint primary key,
    name varchar(255),
    warningDays bigint default null,
    dangerDays bigint default null
);
create sequence ru_rubbles_hrhero_model_state_id_seq start with 1;

CREATE TABLE ru_rubbles_hrhero_model_candidate
(
    id          bigint primary key,
    name        varchar(255),
    vacancy_id bigint references ru_rubbles_hrhero_model_vacancy(id),
    state_id    bigint references ru_rubbles_hrhero_model_state(id),
    url         varchar(255)
);
create sequence ru_rubbles_hrhero_model_candidate_id_seq start with 1;

CREATE TABLE ru_rubbles_hrhero_model_vrelation
(
    id          bigint primary key,
    vacancy_id bigint references ru_rubbles_hrhero_model_vacancy(id),
    user_id     varchar(32) references appix_users(id)
);
create sequence ru_rubbles_hrhero_model_vrelation_id_seq start with 1;

CREATE TABLE ru_rubbles_hrhero_model_srelation
(
    id          bigint primary key,
    state_id     bigint references ru_rubbles_hrhero_model_state(id),
    user_id     varchar(32) references appix_users(id)
);
create sequence ru_rubbles_hrhero_model_srelation_id_seq start with 1;

create table ru_rubbles_hrhero_model_log
(
    id             bigint primary key,
    creation_date timestamp not null,
    event_type     varchar(32) not null,
                candidate_id          bigint     not     null     references
ru_rubbles_hrhero_model_candidate(id),
    state_id       bigint not null references ru_rubbles_hrhero_model_state(id),
                 vacancy_id                bigint     not     null     references
ru_rubbles_hrhero_model_vacancy(id),
    actor_user_id varchar(32) default null references appix_users(id)
);
create sequence ru_rubbles_hrhero_model_log_id_seq start with 1;

create table ru_rubbles_hrhero_model_comment
```

```
(
    id              bigint primary key,
    creation_date timestamp not null,
    candidate_id bigint not null references ru_rubbles_hrhero_model_candidate(id),
    actor_user_id varchar(32) default null references appix_users(id),
    comment         text
);
create sequence ru_rubbles_hrhero_model_comment_id_seq start with 1;

create table ru_rubbles_hrhero_model_flow
(
    id              bigint primary key,
    definition      text
);
create sequence ru_rubbles_hrhero_model_flow_id_seq start with 1;
```

Также можно включить веб-консоль H2, чтобы посмотреть на созданные таблицы. Для этого надо добавить в `application.properties`:

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2
spring.h2.console.settings.web-allow-others=true
```

Запускаем приложение и проверяем результат Адрес подключения к базе данных можно найти в логе приложения:

```
INFO 7421 --- [           main]
o.f.c.i.database.base.BaseDatabaseType   : Database:
**jdbc:h2:mem:6f377de7-cb80-421a-8cc5-91cf72476678**
```

Переходим по адресу `http://localhost:8080/h2`, заполняем JDBC URL:

English Preferences Tools Help
Login
Saved Settings: Generic H2 (Embedded)
Setting Name: Generic H2 (Embedded) Save Remove
Driver Class: org.h2.Driver
JDBC URL: jdbc:h2:mem:6f377de7-cb80-421a-8cc5-91cf72476678
User Name: sa
Password:
Connect Test Connection

Рис. 6.2 – Настройки

Нажимаем "Connect" и смотрим, какие таблицы существует в БД. Убеждаемся, что наши таблички на месте.

7. Реализация CRUD-менеджеров (создание CRUD-сущностей)

Для создания CRUD-сущностей необходимо имплементировать логику работы основных сущностей Системы.

Классы, необходимые для реализации полноценного Appix CRUD:

- Entity (сущность, может, но не обязана, быть выражена как JpaEntity).
- Repository (если используется реализация CrudServiceJpa: CrudService).
- Имплементация интерфейса CrudService.
- Имплементация интерфейса CrudController.
- Имплементация интерфейса TableBuilder для описания непосредственно Crud-таблицы.

7.1. Утилита crudcp.py

Все эти классы можно создавать вручную, но для ускорения этой весьма рутиной процедуры во фреймворке предусмотрена утилита `crudcp.py`, которая позволяет создать весь необходимый набор буквально с помощью одной команды.

Чтобы ей пользоваться, необходимо ее подключить.

Важно заметить, что это потребуется сделать единожды, а пользоваться ей вы будете достаточно часто

Подключаем `crudcp.py`:

1) Скопируем папку `crudcp` в корень нашего проекта.

2) Скопируем `crudcp.py` в корень нашего проекта.

У нас есть 3 таблицы-сущности, с которыми мы хотим начать работать: `Vacancy`, `Candidate`, `State`, а также 2 таблицы-связи, определяющие доступ пользователей к сущностям `Vacancy` и `State`.

Сгенеририуем для них заготовки кода. Для этого из корня проекта, куда мы скопировали `crudcp.py` выполним 5 практически идентичных команд:

```
# python3 crudcp.py <ENTITY NAME> <TARGET PACKAGE>
python3 crudcp.py Vacancy ru.rubbles.hrhero.model.vacancy
python3 crudcp.py Candidate ru.rubbles.hrhero.model.candidate
python3 crudcp.py State ru.rubbles.hrhero.model.state
python3 crudcp.py VacancyRelation ru.rubbles.hrhero.model.vrelation
python3 crudcp.py StateRelation ru.rubbles.hrhero.model.srelation
```

Так как у нас нет сабмодулей, то мы использовали синтаксис команды, в котором указание имени модуля опущено. Если нужно создать сущности для какого-то определенного модуля, то стоит использовать более полный формат вызова:

```
python3 crudcp.py <TARGET_MODULE> <ENTITY NAME> <TARGET PACKAGE>
```

7.2. Расширение базовой функциональности

Сгенерированные классы CandidatePage и VacancyPage нам не нужны, так как мы уже создали их в предыдущем уроке, их можно смело удалить.

```
rm src/main/kotlin/ru/rubbles/hrhero/model/candidate/CandidatePage.kt

rm src/main/kotlin/ru/rubbles/hrhero/model/vacancy/VacancyPage.kt
```

А вот страницы `StatePage`, `StateRelationPage` и `VacancyRelationPage` нам пригодятся, поэтому перенесем их в pages для консистентности. Перенесите их средствами IDE для автоматической корректировки имени пакета. Определим для них режим доступа, чтобы они были доступны только для админа. На примере страницы `StatePage` (повторить для `StateRelationPage` и `VacancyRelationPage`):

```
@Component
class StatePage(
    val stateTable: StateTable,
) : PageBuilder(
    title = "Этапы воронки",                          // Отредактируем параметр title
     access = accessList(BuiltinRoles.ROOT),         // Определим режим доступа к
этой страницы - только для админа
) {
   // тут без изменений
}
```

Теперь добавим страницы в роутер:

```
@Component
class HrHeroRouter(
    // ...
    // Инжектим нужные нам страницы в роутер
    private val statePage: StatePage,
    private val vacancyRelationPage: VacancyRelationPage,
    private val stateRelationPage: StateRelationPage,
) : BaseRouter() {
    override fun getAppMenu(): Menu {
        return Menu(
            mutableListOf(
                // ...
                // Добавляем пункты в меню
                MenuItem(
                    title = "Настройка этапов",
                    path = "/mngt/states",
                    pageBuilder = statePage
                ),
                MenuItem(
                    title = "Доступ к вакансиям",
                    path = "/mngt/vrelation",
                    pageBuilder = vacancyRelationPage
                ),
```

```
                MenuItem(
                    title = "Доступ к этапам",
                    path = "/mngt/srelation",
                    pageBuilder = stateRelationPage
                )
            )
        )
    }
}
```

Теперь, авторизовавшись под учетной записью root мы можем увидеть пункты меню “Настройка этапов”, “Доступ к этапам” и “Доступ к вакансиям”. Правда, они пока толком не грузятся, потому что есть ряд нестыковок в тех классах сущностей, которые были сгенерированы. Нам предстоит их поправить.

7.3. Корректировка State

В классе State корректируем поля сущности в соответствии с нашей схемой данных:

```
@Entity
@JsonIgnoreProperties(ignoreUnknown = true)
@Table(name = "ru_rubbles_hrhero_model_state")
class State(

    @Id
    @SequenceGenerator(
        name = "ru_rubbles_hrhero_model_state_id_seq",
        sequenceName = "ru_rubbles_hrhero_model_state_id_seq",
        allocationSize = 1
    )
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "ru_rubbles_hrhero_model_state_id_seq"
    )
    @Column(name = "id")
    @IntProperty(alias = "ID", subType = DataSubType.PK)
    var id: Long?,

    @Column(name = "name")
    @StringProperty(alias = "Название этапа", subType = DataSubType.NAME)
    var name: String,

    @Column(name = "warningdays")
    @IntProperty(alias = "Тревожно (кол-во дней)")
    var warningDays: Long? = null,

    @Column(name = "dangerdays")
    @IntProperty(alias = "Критично (кол-во дней)")
    var dangerDays: Long? = null,
) : Serializable {
    constructor() : this(null, "", 0, 0)
}
```

Убрав поле description из класса State и добавив новые поля, мы нарушили компилируемость класса `StateTable`. Вносим соответствующие коррективы в этот класс:

```
@Component("ru.rubbles.hrhero.model.state.StateTable")
class StateTable : TableBuilder<State>() {
```

```
        override  fun  build(ctx:  Map<String,  Any?>,  authorization:  String?):
Table<State> {
        return Table(
            entityClass = State::class.java,
            api = crudApi(StateController.api),
            // Меняем заголовок тут:
            caption = "Управление статусами кандидатов в воронке",
            pagination = false,
            defaultPageSize = 10,
                            actions  =  listOf(CrudAction.ADD,  CrudAction.EDIT,
CrudAction.DELETE),
            // Меняем наборы полей тут:
            presence = mapOf(
                CrudAction.READ to listOf(State::id.name, State::name.name),
                                   CrudAction.ADD  to  listOf(State::name.name,
State::warningDays.name, State::dangerDays.name),
                                  CrudAction.EDIT  to  listOf(State::name.name,
State::warningDays.name, State::dangerDays.name),
            ),
            // Выключаем поиск, так как в таблице будет мало записей:
            search = mapOf()
        )
    }
}
```

7.4. Корректировка Vacancy

Повторяем примерно тот же набор модификаций и в отношении `Vacancy`:

- Корректируем поля сущности `Vacancy`, заменяя `description` на url
- Вносим соответствующие правки в `VacancyTable`

Теперь создадим полную копию класса `VacancyTable` и назовем ее `VacancyMngtTable`, а в исходной версии `VacancyTable` уберем возможности редатирования и добавления записей:

```
@Component("ru.rubbles.hrhero.model.vacancy.VacancyTable")
class VacancyTable : TableBuilder<Vacancy>() {
        override  fun  build(ctx:  Map<String,  Any?>,  authorization:  String?):
Table<Vacancy> {
        return Table(
            entityClass = Vacancy::class.java,
            api = crudApi(VacancyController.api),
             caption  =  "Просмотр  справочника  вакансий",   //  Меняем  заголовок,
подчеркиваем что только просмотр
            pagination = false,
            defaultPageSize = 10,
             actions = listOf(),                            // Запрещаем действия
ADD/EDIT/DELETE
              presence = mapOf(                                // Здесь оставляем
только CrudAction.READ
                CrudAction.READ to listOf(Vacancy::id.name, Vacancy::name.name,
Vacancy::url.name),
            ),
            search = mapOf()
        )
    }
}
```

Чтобы закончить работу с `Vacancy` нам остается только прописать `VacancyMngtTable` в страницу `VacancyMngtPage`, а `VacancyTable` в `VacancyPage`: На примере `VacancyMngtPage`:

```
@Component
class VacancyMngtPage(
    // Инжектим соответствующую версию таблицы:
    private val vacancyMngtTable: VacancyMngtTable
) : PageBuilder(
    title = "Вакансии (управление)",
    access = accessList(BuiltinRoles.ROOT),
) {
    override fun build(authorization: String?): Page {
        return Page(
            pageBuilder = this,
            // Добавляем таблицу в тело страницы простейшим образом:
            content = Container(
                listOf(
                    Row(
                        listOf(
                            Column(
                                TableWidget(
                                    id = "table",
                                     table = vacancyMngtTable.build(mapOf(),
authorization)
                                )
                            )
                        )
                    )
                )
            )
        )
    }
}
```

Полностью по аналогии поступаем с парой `VacancyTable` и `VacancyPage`. Авторизовавшись как root мы можем видеть раздел “Вакансии” и таблица, представленная в нем, позволяет нам редактировать записи, добавлять их и удалять. Авторизовавшись под учетной записью `reviewer1@hrhero.pro/reviewer1`, мы тоже видим раздел Вакансии, но тут он работает только на чтение.

7.5. Корректировка Candidate

Как и с другими сущностями, корректируем набор полей в соответствии со схемой данных:

```
@Entity
@JsonIgnoreProperties(ignoreUnknown = true)
@Table(name = "ru_rubbles_hrhero_model_candidate")
class Candidate(

    @Id
    @SequenceGenerator(
        name = "ru_rubbles_hrhero_model_candidate_id_seq",
        sequenceName = "ru_rubbles_hrhero_model_candidate_id_seq",
```

```
        allocationSize = 1
    )
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "ru_rubbles_hrhero_model_candidate_id_seq"
    )
    @Column(name = "id")
    @IntProperty(alias = "ID", subType = DataSubType.PK)
    var id: Long?,

    @Column(name = "name")
    @StringProperty(alias = "Имя")
    var name: String,

    @Column(name = "url")
    @StringProperty(alias = "Ссылка на резюме")
     var url: String,                                    // Просто строковое
поле для хранения ссылки на резюме

    @Column(name = "state_id")
     @SelectOptionsProperty(                                       // Здесь мы
используем технику "соединения" crud-сущностей
        alias = "Статус",
        required = true,
         joinApi = StateController.joinApi,                  // Прописываем URL
для получения имен по айди сущности
         selectOptionsApi = StateController.dictApi,         // Прописываем URL
для получения выпадающего списка опций
    )
    var stateId: Long?,

    @Column(name = "vacancy_id")
     @SelectOptionsProperty(                                       // Здесь мы
используем технику "соединения" crud-сущностей
        alias = "Вакансия",
        required = true,
         joinApi = VacancyController.joinApi,                // Прописываем URL
для получения имен по айди сущности
         selectOptionsApi = VacancyController.dictApi,       // Прописываем URL
для получения выпадающего списка опций
    )
    var vacancyId: Long?,

    ) : Serializable {
    constructor() : this(null, "", "", 0L, 0L)
}
```

Полностью по аналогии с `Vacancy`, мы должны создать 2 версии определения crud-таблицы: `CandidateTable` и `CandidateMngtTable`, где первая только для READ, а вторая - полноценный CRUD.

При этом, удаляя `description` из `presense` таблиц вы можете добавить туда новые поля сущности `stateId`, `vacancyId`, `url`, а также использовать их при определения параметра таблицы `search`:

```
@Component("ru.rubbles.hrhero.model.candidate.CandidateTable")
class CandidateTable : TableBuilder<Candidate>() {
        override  fun  build(ctx:  Map<String,  Any?>,  authorization:  String?):
Table<Candidate> {
        return Table(
            entityClass = Candidate::class.java,
```

```
            api = crudApi(CandidateController.api),
            caption = "Просмотр базы кандидатов",
            pagination = false,
            defaultPageSize = 10,
            actions = listOf(),
            presence = mapOf(
                CrudAction.READ to listOf(
                    Candidate::id.name,
                    Candidate::name.name,
                    Candidate::vacancyId.name,
                    Candidate::stateId.name,
                    Candidate::url.name,
                ),
            ),
             // Это будет означать возможность фильтрации таблицы не только по
имени кандидата,
                // но и по их статусам, а также принадлежностям тем или иным
вакансиям:
            search = mapOf(
                               Candidate::name.name  to  listOf(EqualityOperator.EQ,
EqualityOperator.LIKE),
                Candidate::vacancyId.name to listOf(EqualityOperator.EQ),
                Candidate::stateId.name to listOf(EqualityOperator.EQ),
            )
        )
    }
}
```

Полностью по аналогии с `Vacancy`, мы должны прописать 2 созданные версии таблицы в страницы `CandidatePage` и `CandidateMngtPage`, соответственно.

7.6. Корректировки VacancyRelation и StateRelation

Корректируем наборы полей для сущностей `VacancyRelation` и `StateRelation`. На примере `VacancyRelation`:

```
@Entity
@JsonIgnoreProperties(ignoreUnknown = true)
@Table(name = "ru_rubbles_hrhero_model_vrelation")
class VacancyRelation(

    @Id
    @SequenceGenerator(
        name = "ru_rubbles_hrhero_model_vrelation_id_seq",
        sequenceName = "ru_rubbles_hrhero_model_vrelation_id_seq",
        allocationSize = 1
    )
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "ru_rubbles_hrhero_model_vrelation_id_seq"
    )
    @Column(name = "id")
    @IntProperty(alias = "ID", subType = DataSubType.PK)
    var id: Long?,

    @Column(name = "vacancy_id")
    @SelectOptionsProperty(
        alias = "Вакансия",
        required = true,
        joinApi = VacancyController.joinApi,
        selectOptionsApi = VacancyController.dictApi,
```

```
    )
    var vacancyId: Long?,

    @Column(name = "user_id")
    @SelectOptionsProperty(
        alias = "Сотрудник",
        joinApi = BuiltinUsersController.joinApi,
        selectOptionsApi = BuiltinUsersController.dictApi,
    )
    var userId: String? = null,

    ) : Serializable {

    constructor() : this(null, 0, "")
}
```

Для сущности `StateRelation` все также, но вместо поля `vacancyId`, которое использовало `VacancyController`, будет уже `stateId`, использующее `StateController`. Для обоих сущностей корректируем их `*Table` классы.

Итоги урока. Мы запрограммировали все основные сущности и разместили их в соответствующих страницах, при этом корректно учли функциональные ограничения нашей ролевой модели. Увидели, как можно создавать разные "проекции" crud-таблиц для разных ролей. Поняли, как можно связывать сущности друг с другом, чтобы при работе с ними были выпадающие списки и вместо id подтягивались бы имена записей. Поверхностно познакомились с тем, как можно размещать контент внутри страниц. При старте системы, у нас еще нет данных в справочниках статусов и вакансий. Их может создавать админ системы, но при первом старте - справочники пусты..

8. Ролевая модель (горизонтальная)

В целом уже сейчас все работает, однако мы добавили функционал привязки пользователей к вакансиям и статусам. То есть, помимо ранее заложенной функциональной ролевой модели мы теперь закладываем и элементы горизонтальной ролевой модели - когда полномочия контролируются не по функии, а по набору доступных объектов, на которые эти функции могут распространяться.

Это означает для нас необходимость внести еще пару правок.

8.1. Вакансии

Управление вакансиями осуществляется админом, для которого привязки не актуальны. Администратор видит все вакансии и управляет всеми вакансиями. Таким образом классы `VacancyMngtTable` и `VacancyMngtPage` нам трогать не надо. А вот просмотр справочника вакансий у нас доступен для ролей ревьюевера и рекрутера. Следовательно, мы должны ограничить то, что видят эти роли, чтобы они видели только те вакансии, к которым у них есть доступ.

То есть, нам необходимо модифицировать класс `VacancyTable`:

```
@Component("ru.rubbles.hrhero.model.vacancy.VacancyTable")
class VacancyTable(
     private val authService: AuthService,                          // Инжектим
сервис аутентификации
      private val vacancyRelationService: VacancyRelationService   // Инжектим
сервис доступа к вакансиям
) : TableBuilder<Vacancy>() {
       override fun build(ctx: Map<String, Any?>, authorization: String?):
Table<Vacancy> {

        // Получаем пользователя:
        val user = authService.getUser(authorization)

        // Получаем список идентификаторов вакансий, доступных пользователю:
        val vacancyIds = vacancyRelationService.getList(
            searchData = mapOf(
                                               VacancyRelation::userId.name   to
SearchCriteria(EqualityOperator.EQ, user.id!!)
            )
        ).content.map { it.vacancyId }.toSet()

        return Table(
            // ... без изменений

              // Используем полученный список для ограничения области видимомсти
таблицы:
            bounded = mapOf(
                        Vacancy::id.name to SearchCriteria(EqualityOperator.IN,
vacancyIds)
            )
        )
    }
}
```

Теперь эта таблица будет учитывать выданные пользователям полномочия в отношении вакансий.

8.2. Кандидаты

В отношении кандидатов история похожа. Отличие лишь в том, что админ вообще не имеет доступа к этим страницам. А обе имеющиеся "линии" функционала - для рекрутера и для ревьюевера - должны быть ограничены списком назначенных вакансий. Плюс к этому, список доступных кандидатов для ревьюевера должен быть ограничен еще и по статусу воронки. Получается, правки вносим в оба класса `CandidateTable` и `CandidateMngtTable`:

```
CandidateTable
@Component("ru.rubbles.hrhero.model.candidate.CandidateTable")
class CandidateTable(
    // Инжектим сервис аутентификации и сервисы связей с вакансиями и этапами:
    private val authService: AuthService,
    private val vacancyRelationService: VacancyRelationService,
    private val stateRelationService: StateRelationService,
) : TableBuilder<Candidate>() {
        override fun build(ctx: Map<String, Any?>, authorization: String?): Table<Candidate> {

        // Получаем пользователя
        val user = authService.getUser(authorization)

        // Получаем доступный ему список вакансий
        val vacancyIds = vacancyRelationService.getList(
            searchData = mapOf(
                                                VacancyRelation::userId.name to SearchCriteria(EqualityOperator.EQ, user.id!!)
            )
        ).content.map { it.vacancyId }.toSet()

        // Получаем доступный ему список этапов
        val stateIds = stateRelationService.getList(
            searchData = mapOf(
                                                StateRelation::userId.name to SearchCriteria(EqualityOperator.EQ, user.id!!)
            )
        ).content.map { it.stateId }.toSet()

        return Table(
            // ... без изменений

            // Ограничиваем область видимости таблицы:
            bounded = mapOf(
                                                Candidate::vacancyId.name to SearchCriteria(EqualityOperator.IN, vacancyIds),
                  Candidate::stateId.name to SearchCriteria(EqualityOperator.IN, stateIds),
            )
        )
    }
}
```

Точно такие же изменения делаем в рамках `CandidateMngtTable`, но ограничиваем только по делегированным вакансиям. Уже сейчас разные роли видит только те записи, к которым у них так или иначе есть доступ. Остается правда один нюанс: когда мы авторизованы как рекрутер и хотим добавить нового кандидата - в форме добавления нам доступны все вакансии. Да, если мы добавим в "чужую" вакансию, то потом не будем его видеть, но хочется ограничить и возможность добавления. Открываем класс `Candidate` и видим, что для его свойства `vacancyId` задано:

```
selectOptionsApi = VacancyController.dictApi
```

Переходим в данный метод и видим, что он унаследован из `CrudController`, а тот в свою очередь работает с классом `CrudFacade`, вызывая метод `getByNameMatch`. `CrudFacade` является всего лишь оберткой над нашим `VacancyService: CrudService`, поэтому мы можем переопределить этот метод:

```
@Service
class VacancyService(
    vacancyRepository: VacancyRepository,
    // Как и ранее с таблицами, добавляем нужные нам сервисы
    private val authService: AuthService,
    private val acsService: BuiltinAcsService,
    private val vacancyRelationService: VacancyRelationService,
) : CrudServiceJpa<Vacancy, Long>(vacancyRepository, Vacancy::class.java) {

    override fun getByNameMatch(
        query: String?,
        ctx: Map<String, Any?>?,
        authorization: String?
    ): List<SelectOption<Long>> {
        // Получаем пользователя:
        val user = authService.getUser(authorization)

            // Если пользователь админ - ограничений нет, оставляем дефолтную
логику,
        // Иначе делаем более сложную проверку:
        if (!acsService.check(user, accessList(BuiltinRoles.ROOT))) {

            // Получаем список доступных вакансий:
            val vacancyIds = vacancyRelationService.getList(
                searchData = mapOf(
                                                    VacancyRelation::userId.name to
SearchCriteria(EqualityOperator.EQ, user.id!!)
                )
            ).content.mapNotNull { it.vacancyId }.plus(0)

                // Удовлетворяем контракту getByNameMatch воспользовшись удобным
getByIds:
                                                return super.getByIds(vacancyIds, ctx,
authorization).values.toList()
        }
        // Дефолтный ответ:
        return super.getByNameMatch(query, ctx, authorization)
    }
}
```

9. Проектирование событийной модели приложения

В рамках функционирования любого приложения возникают те или иные события, которые важны с точки зрения этого приложения, которые являются значимыми с доменной точки зрения. Appix считает это важным концептом, поэтому строится вокруг концепции Событийная модель.

Проектируя событийную модель, мы должны задаться вопросом: "Какие события релевантны для такого приложения как HrHero?" Не долго думая можно прийти к такому минимальному списку:

- Добавление нового кандидата
- Изменение статуса кандидата

В этом уроке мы создадим соответствующие типы событий, проинжектим их в бизнес-логику, а также реализуем их сохранение, чтобы в дальнейшем использовать при создании титульного дашборда.

9.1. Создание события

Сперва необходимо создать `enum class`, расширив им интерфейс `EventType`, и перечислить в нем оба события, которые мы придумали и ключи контекста этих событий, то есть то, что описывает произошедшее событие:

```
package ru.rubbles.hrhero.events

import appix.core.crud.table.Join
import appix.core.dto.SelectOption
import appix.core.events.EventType
import appix.core.events.notifications.NotificationChannel

enum class CandidateEventType(
    override val join: Join? = null,
    override val options: List<SelectOption<String>>? = listOf(),
     override val contextKeys: Set<String> = setOf(),                    // На
данном этапе нам нужно только это свойство
    override val channels: Set<NotificationChannel> = setOf()
) : EventType {

    NEW_CANDIDATE(
        // Прописываем те ключи контекста, которыми будет описываться событие:
        contextKeys = setOf(
            "id", "name",
            "stateId", "stateName",
            "vacancyId", "vacancyName",
            "actorId", "actorName"
        )
    ),
    CANDIDATE_STATE_UPDATE(
        // Прописываем те ключи контекста, которыми будет описываться событие:
        contextKeys = setOf(
            "id", "name",
            "stateId", "stateName",
```

```
            "vacancyId", "vacancyName",
            "actorId", "actorName"
        ),
    )
}
```

Это позволит Appix "видеть" эти события, как "события" и обрабатывать их соответственно. Ключи контекста в принципе выбираются произвольно, исходя из здравого смысла, однако важно, что именно такой набор ключей необходимо будет передавать при вызове этого события в рамках работы приложения. Теперь создадим классы событий под каждое значение в enum:

```
package ru.rubbles.hrhero.events

import appix.core.events.BaseEvent

// Сигнатуру конструктора класса делаем максимально приближенной к доменной области
// а также соответствующей тем данным, что нам надо будет передать в контекст события
class NewCandidateEvent(
    source: Any,
     val actor: User, val candidate: Candidate, val vacancy: Vacancy, val state: State) :
    BaseEvent(
        source = source,
            eventType = CandidateEventType.NEW_CANDIDATE,                // Тут хардкодим тип, чтобы в будущем избежать ошибок
         contextData = mapOf(                                           // Собираем контекст, который "описывает" событие
            "id" to candidate.id as Any,
            "name" to candidate.name as Any,
            "stateId" to state.id as Any,
            "stateName" to state.name as Any,
            "vacancyId" to vacancy.id as Any,
            "vacancyName" to vacancy.name as Any,
            "actorId" to actor.id as Any,
            "actorName" to actor.name as Any,
        ),
        candidate.id.toString()
    )

class CandidateChangedEvent(
    source: Any,
     val actor: User, val candidate: Candidate, val vacancy: Vacancy, val state: State) :
    BaseEvent(
                       // полностью по аналогии, только eventType = CandidateEventType.CANDIDATE_STATE_UPDATE,
    )
```

По большому счету, можно было бы обойтись и 1 классом, так как они по сути идентичны, но использовать 2 разных предпочтительнее просто из соображений типизации. Плюс, они так похожи скорее благодаря частному случаю - сами события, которые мы придумали, максимально близки друг к другу, но если бы события были бы другими, то реализация соответствующих классов тоже отличалась бы гораздо серьезнее.

9.2. Оформление события в бизнес-логику

Как мы понимаем, бизнес-логика концентрируется внутри классов `@Service`. Поэтому, очевидно, точкой инжекта этих событий станет класс `CandidateService`.

Посмотрим, что у нас сгенерировано с помощью `crudcp.py`:

```
@Service
class CandidateService(
    candidateRepository: CandidateRepository
) : CrudServiceJpa<Candidate, Long>(candidateRepository, Candidate::class.java)
```

Лаконично. Но это благодаря тому, что логика "спрятана" в базовом классе `CrudServiceJpa`, который в свою очередь реализует интерфейс `CrudService`. Это, в свою очередь, означает, что мы можем переопределить нужные нам методы для инжекта вызова событий:

```
@Service
class CandidateService(
    candidateRepository: CandidateRepository,
     private val authService: AuthService,          // Добавим AuthService для получения актора действия
     private val vacancyService: VacancyService, // Добавим VacancyService для работы с вакансиями
     private val stateService: StateService,       // Добавим StateService для работы со статусами
     private val eventService: EventService,       // Добавим EventService для эмита событий
) : CrudServiceJpa<Candidate, Long>(candidateRepository, Candidate::class.java)
{

    /**
     * Переопределяем метод создания нового кандидата
     */
       override fun createOne(newItem: Candidate, ctx: Map<String, Any?>?, authorization: String?): Candidate {

        // Получим актора действия:
        val actor = authService.getUser(authorization)

        // Создаем кандидата, используя базовый класс:
        val candidate = super.createOne(newItem, ctx, authorization)

        // Получаем объекты "статуса" и "вакансии":
                  val state = stateService.getOne(candidate.stateId!!, ctx, authorization)
              val vacancy = vacancyService.getOne(candidate.vacancyId!!, ctx, authorization)

        // Осуществляем вызов события:
                eventService.emitBase(NewCandidateEvent(source = this, actor, candidate, vacancy, state))
        return candidate
    }

    /**
     * Переопределяем метод обновления кандидата
     */
     override fun updateOne(id: Long, changedItem: Candidate, ctx: Map<String, Any?>?, authorization: String?): Candidate {
```

```
        // Получим актора действия:
        val actor = authService.getUser(authorization)

        // Обновляем кандидата, используя базовый класс:
        val candidate = super.updateOne(id, changedItem, ctx, authorization)

        // Получаем объекты "статуса" и "вакансии":
        val state = stateService.getOne(candidate.stateId!!, ctx, authorization)
        val vacancy = vacancyService.getOne(candidate.vacancyId!!, ctx, authorization)

        // Осуществляем вызов события:
        eventService.emitBase(CandidateChangedEvent(source = this, actor, candidate, vacancy, state))
        return candidate
    }
}
```

В целом первая часть работы выполнена: событийная модель спроектирована и реализована, она уже работает. Просто события, которая она производит никак не используются, "эмитятся" в холостую.

9.3. Добавление заготовки обработчика событий

Добавим класс-компонент (пока пустой), который будет отвечать за обработку некоторых (или всех) событий общей событийной модели приложения и обрабатывать их каким-то специальным образом. Один тип события может предусматривать один алгоритм обработки, другой тип события - другой алгоритм обработки.

```
@Component
class CandidateEventListener {

    @EventListener
    fun onNewCandidate(event: NewCandidateEvent) {}

    @EventListener
    fun onUpdateCandidate(event: CandidateChangedEvent) {}

}
```

Теперь, подумаем о том, как мы хотим обрабатывать наши события. Один из вариантов, который может прийти в голову - отправка уведомлений пользователям. Однако этой части мы коснемся в одном из следующих уроков, который будет посвящен нотификационной модели. Другой вариант, на котором мы и остановимся в рамках данного урока, предполагает реализацию некого журналирования событий, сохранение их в базе данных в виде `timeseries` датасета, для дальнейшего его использования для построения титульного дашбоарда приложения. Прежде всего сгенируем новый crud-пакет, для хранения логов:

```
python3 crudcp.py Log ru.rubbles.hrhero.model.log
```

Как обычно, после генерации кода, мы должны внести правки в класс сущности, чтобы она соответствовала нашей модели БД.

```
@Entity
@JsonIgnoreProperties(ignoreUnknown = true)
@Table(name = "ru_rubbles_hrhero_model_log")
class Log(

    @Id
    @SequenceGenerator(
        name = "ru_rubbles_hrhero_model_log_id_seq",
        sequenceName = "ru_rubbles_hrhero_model_log_id_seq",
        allocationSize = 1
    )
    @GeneratedValue(
        strategy = GenerationType.SEQUENCE,
        generator = "ru_rubbles_hrhero_model_log_id_seq"
    )
     @Column(name = "id")                                                  //
Просто автоинкрементный id
    @IntProperty(alias = "ID", subType = DataSubType.PK)
    var id: Long?,

     @Column(name = "creation_date")                                       //
Метка времени события
    @DateTimeProperty(alias = "Дата и время", required = true)
    var creationDate: LocalDateTime,

     @Column(name = "event_type")                                      // Тип
события (для фильтров в таблице)
    @SelectOptionsProperty(alias = "Событие", required = true)
    var eventType: String,

     @Column(name = "candidate_id")                                        //
Привязка события к кандидату
    @SelectOptionsProperty(
        alias = "Кандидат",
        required = true,
        joinApi = CandidateController.joinApi,
        selectOptionsApi = CandidateController.dictApi,
    )
    var candidateId: Long,

     @Column(name = "state_id")                                            //
Привязка события к статусу
    @SelectOptionsProperty(
        alias = "Статус",
        required = true,
        joinApi = StateController.joinApi,
        selectOptionsApi = StateController.dictApi,
    )
    var stateId: Long,

     @Column(name = "vacancy_id")                                          //
Привязка события к вакансии
    @SelectOptionsProperty(
        alias = "Вакансия",
        required = true,
        joinApi = VacancyController.joinApi,
        selectOptionsApi = VacancyController.dictApi,
    )
    var vacancyId: Long,
```

```
    @Column(name = "actor_user_id")                                       //
Привязка события к действующему лицу
    @SelectOptionsProperty(
        alias = "Сотрудник",
         joinApi = BuiltinUsersController.joinApi,                  // Тут
используемый готовый контроллер из ядра
        selectOptionsApi = BuiltinUsersController.dictApi,
    )
    var actorUserID: String? = null,
    ) : Serializable {
    constructor() : this(
        null, LocalDateTime.now(), "", 0, 0, 0
    )
}
```

Вносим правки в связанный класс `LogTable`, просто чтобы код компилировался (самостоятельно). Теперь нам требуется лишь прописать сохранение логов в рамках пока пустующего класса `CandidateEventListener`:

```
@Component
class CandidateEventListener(
    private val logService: LogService              // Инжектим себе сервис
LogService
) {
    @EventListener
    fun onNewCandidate(event: NewCandidateEvent) {
        logService.createOne(
            Log(
                id = null,
                creationDate = LocalDateTime.now(),
                eventType = event.eventType.toString(),
                candidateId = event.candidate.id!!,
                stateId = event.state.id!!,
                vacancyId = event.vacancy.id!!,
                actorUserID = event.actor.id,
            )
        )
    }
    @EventListener
    fun onUpdateCandidate(event: CandidateChangedEvent) {
        // полностью аналогично коду onNewCandidate
    }

}
```

Проделаем небольшую проверку

1. Авторизуемся как root/12345
2. Идем в раздел Вакансии и добавляем первую вакансию Первая вакансия
3. Идем в раздел Настройка этапов и добавляем простой flow из одного статуса Добавлен
4. Даем одному из рекрутеров доступ к созданной вакансии и авторизуемся под ним.
5. Идем в раздел Кандидаты и добавляем первого кандидата
6. Идем в h2 (`http://localhost:8080/h2`, jdbc-адрес базы в логах старта приложения) и смотрим таблицу `RU_RUBBLES_HRHERO_MODEL_LOG`

10. Проектирование модели нотификаций пользователей

Теперь, когда у нас спроектирована событийная модель мы можем перейти к проектированию модели нотификаций. С точки зрения HrHero нам важно предусмотреть систему уведомлений пользователей о ключевых событиях, происходящих в рамках приложения - чтобы каждый пользователь своевременно мог узнавать о том, что его касается.

10.1 Настройки приложения

Прежде всего, проверим настройки приложения. Необходимо убедиться в том, что канал нотификаций email включен, а настройки SMTP spring.mail полны и корректны:

```
appix.admin.enabled=true
appix.alerting.channel.email.enabled=true
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=appixtest@gmail.com
spring.mail.password=appix141021
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

10.2. Встроенные подписки - декларативная настройка отправки уведомлений

В прошлом уроке мы уже видели, как можно написать свой кастомный обработчик событий и выполнять произвольные действия при их возникновении. Теоретически, мы могли пойти таким путем - реализовали бы свои `@EventListener` для каждого типа событий, заинжектили бы себе `NotificationService` и воспользовались бы его публичным методом `send()` в рамках реализации этих обработчиков. Вполне рабочий способ. Более того, ровно такая техника использована в ядре фреймворка при реализации функционала регистрации и отправки писем с кодом подтверждения имейла. Такой подход мы можем назвать императивной отправкой. Но в данном случае нам будет удобнее использовать другую технику и другую возможность фреймворка - встроенные подписки, которые помогают нам реализовать модель декларативной отправки. Чтобы это сделать нужно реализовать интерфейс `NotificationDecisionResolver`.

Создадим новый пакет `events.notifications` и новый класс в нем `NewCandidateNotificationDecisionResolver`:

```
@Component
class NewCandidateNotificationDecisionResolver(
    final override val eventType: EventType = CandidateEventType.NEW_CANDIDATE
) : NotificationDecisionResolver {
```

```
    private val builtinSubscriptions = listOf(
        SubscriptionRecord(
            eventType = eventType,
            notificationChannel = NotificationChannel.EMAIL,
            receivers = HrHeroRoles.REVIEWER,
        )
    )

    override fun getStaticSubscriptions(): List<SubscriptionRecord> =
builtinSubscriptions
}
```

Метод `getStaticSubscriptions()` возвращает список подписок, для которых мы не указываем пользователя, не указываем адрес, а только тип события, канал и роль получателей. Как только мы добавили такую "встроенную" подписку фреймворк начинает контролировать процесс. Если наш `EventType` указан в рамках модели `SubscriptionRecord` в паре с каким-либо `NotificationChannel`, то сам `EventType` должен быть объявлен как "привязанный" к этому каналу. А если мы "привязываем" событие к каналу, то следом фреймворк проверит наличие у нас специального класса для сборки шаблона уведомления для этого события и для этого канала (связки NotificationChannel+EventType). Если одно из этих условий не выполняется, то фреймворк не позволит запустить приложение.

EventNEW_CANDIDATEcan not be distributed viaEMAILchannel

Вносим коррективы. Поправим объявления наших `EventType`, чтобы показать, что они могут генерировать уведомления по каналу EMAIL, а следовательно - требуют включения этого канала и наличия шаблона:

```
enum class CandidateEventType(
    override val join: Join? = null,
    override val options: List<SelectOption<String>>? = listOf(),
    override val contextKeys: Set<String> = setOf(),                // Ранее мы
добавляли ключи контекста
    override val channels: Set<NotificationChannel> = setOf()       // Теперь
мы добавляем каналы уведомлений
) : EventType {

    NEW_CANDIDATE(
        contextKeys = setOf(
            // ... без изменений
        ),
        // Добавляем канал EMAIL:
        channels = setOf(NotificationChannel.EMAIL)),

    CANDIDATE_STATE_UPDATE(
        // Полностью по аналогии с NEW_CANDIDATE
    )
}
```

```
EventType NEW_CANDIDATE has no notification message builder implementation for NotificationChannel EMAIL
```

Теперь система требует наличие сборщика шаблона уведомления. То есть, нам нужно наследовать класс `EmailNotificationMessageBuilder` и добавить новый класс `EmailNewCandidateMessageBuilder` в пакет `notifications`.

```
@Component
@ConditionalOnNotificationEnabled(NotificationChannel.EMAIL)
class NewCandidateEmailMessageBuilder :
    EmailNotificationMessageBuilder(CandidateEventType.NEW_CANDIDATE) {

    override fun getTemplate(): String {
        return """
            Новый кандидат на позицию `${getPlaceholder("vacancyName")}`!
            Имя: ${getPlaceholder("name")}"
        """
    }
}
```

Обратите внимание на использование метода `getPlaceholder()`. Его использование здесь считается хорошей практикой, так как наличие именно такого ключа контекста события будет таким образом автоматически контроллироваться фреймворком и вероятность ошибки будет сведена к нулю. Класс `CandidateChangedEmailMessageBuilder` имплементируем полностью по аналогии с классом `NewCandidateEmailMessageBuilder`, заменяя тип ивента и внося смысловые коррективы в текст уведомления.

10.3. Уточнение релевантности уведомления

Благодаря тому, что мы определили метод `getStaticSubscriptions` в интерфейсе `NotificationDecisionResolver`, все ревьюверы получают уведомления о новых поступающих кандидатах. Однако, как мы помним, в нашем приложении и рекрутеры и ревьюверы должны иметь специализацию по вакансии. Конкретному ревьюеверу мы должны отправлять только те уведомления, которые сообщают о новых кандидатах, которые проходят по вакансиям этого ревьюевера. Чтобы реализовать эту логику мы определяем 2-ой метод интерфейса `NotificationDecisionResolver`, который называется `confirmRecepients`.

```
@Component
class NewCandidateNotificationDecisionResolver(
    override val eventType: EventType = CandidateEventType.NEW_CANDIDATE,
    private val vacancyRelationService: VacancyRelationService,     // Наш
новый сервис связей
) : NotificationDecisionResolver {

    override fun confirmRecepients(                                  //
Переопределяем метод
```

```
        event: BaseEvent,
        subscriptionRecords: Set<SubscriptionRecord>
    ): Set<SubscriptionRecord> {

        // Строим мапу пользователей на список их вакансий:
        val relMap = vacancyRelationService
            .getList()
            .groupBy { it.userId }
            .map { it.key to it.value.mapNotNull { r -> r.vacancyId } }
            .toMap()

        // Относимся к ивенту как к `NewCandidateEvent` и получаем доступ к
вакансии кандидата:
        val vacancyId = (event as NewCandidateEvent).vacancy.id

        // Непосредственно проверяем, что пользователь указанный в подписке
имеет доступ к вакансии:
        return subscriptionRecords.filter {
relMap[it.userId]?.contains(vacancyId) == true }.toSet()
    }

    // метод getStaticSubscriptions() без изменений
}
```

В случае с событием CANDIDATE_STATE_UPDATE все почти также, правда помимо учета релевантности по признаку вакансии, добавляется еще и учет релевантности уведомлений по признаку "Этапа" воронки.

Мы хотим чтобы рекрутеры получали уведомления по всем этапам воронки, а ревьюеверы только по тем из них, к которым им выдан доступ:

```
@Component
class CandidateChangedNotificationDecisionResolver(
    final override val eventType: EventType = CandidateEventType.NEW_CANDIDATE,
    private val userService: UserService,                           // Сервис
для работы с пользователями
    private val acsService: BuiltinAcsService,                      // Сервис
для проверки прав и ролей
    private val vacancyRelationService: VacancyRelationService,     // Сервис
доступа к вакансиям
    private val stateRelationService: StateRelationService,         // Сервис
доступа к этапам
) : NotificationDecisionResolver {

    // Тут у нас будет не 1, а 2 встроенные подписки
    private val builtinSubscriptions = listOf(
        // Для рекрутера
        SubscriptionRecord(
            eventType = eventType,
            notificationChannel = NotificationChannel.EMAIL,
            receivers = HrHeroRoles.EDITOR,
        ),
        // Для ревьювера
        SubscriptionRecord(
            eventType = eventType,
            notificationChannel = NotificationChannel.EMAIL,
            receivers = HrHeroRoles.REVIEWER,
        ),
    )

    override fun getStaticSubscriptions(): List<SubscriptionRecord> =
builtinSubscriptions
```

```
    override fun confirmRecepients(
        event: BaseEvent,
        subscriptionRecords: Set<SubscriptionRecord>
    ): Set<SubscriptionRecord> {

        // Строим мапу пользователей на список их вакансий:
        val vrelMap = vacancyRelationService
            .getList()
            .groupBy { it.userId }
            .map { it.key to it.value.mapNotNull { r -> r.vacancyId } }
            .toMap()

        // Строим мапу пользователей на список доступных им этапов:
        val srelMap = stateRelationService
            .getList()
            .groupBy { it.userId }
            .map { it.key to it.value.mapNotNull { r -> r.stateId } }
            .toMap()

        // Строим мапу идентификаторов пользователей на объекты пользователей:
        val usersMap = userService
            .findByIds(subscriptionRecords.mapNotNull { it.userId }.toSet())
            .associateBy { it.id!!.toString() }

        // Относимся к ивенту как к `NewCandidateEvent` и получаем доступ к
данным события:
        val vacancyId = (event as NewCandidateEvent).vacancy.id
        val stateId = event.state.id

        // Непосредственно проверяем, что пользователь указанный в подписке
имеет отношение к событию:
        return subscriptionRecords
            .filter {
                // Есть совпадение по вакансии:
                vrelMap[it.userId]?.contains(vacancyId) == true && (
                        // Или пользователь имеет роль рекрутера (и тогда этапы
не важны):
                        acsService.check(usersMap[it.userId]!!,
accessList(HrHeroRoles.EDITOR)) ||
                                // Или у пользователя явно прописан этап, как
доступный ему:
                                srelMap[it.userId]?.contains(stateId) == true
                        )
            }
            .toSet()
    }
}
```

В этом уроке мы разобрали, как можно "подключить" к событийной модели фреймворка еще и нотификационную. Разобрали как подключать нотификации в выбранные каналы, как создавать шаблоны уведомлений для каждого из каналов, как создавать встроенные подписки и настраивать релевантность уведомлений, основываясь на ролевой модели.

11. Проектирование титульного дашборда

Чтобы лишний раз не напрягать систему отправкой писем для дальнейшей работы над приложением мы отключим канал уведомлений целиком:

```
appix.alerting.channel.email.enabled=false
```

Теперь настало время заполнить контентом нашу титульную страницу. Ранее мы реализовали Событийную модель нашего приложения и, как мы помним, благодаря этому наша система пишет все возникающие события в специальную табличку `ru_rubbles_hrhero_model_log`. Также мы планируем использовать непосредственно таблицу `ru_rubbles_hrhero_model_candidate`, которая может давать нам актуальную на текущий момент времени картину положения дел в системе. Благодаря этому и пакету `appix.core.data` мы можем довольно легко добавить нашему приложению немного красивых графиков.

11.1.Подготовка витрины данных

В качестве основной витрины мы будем использовать таблицу `ru_rubbles_hrhero_model_candidate` и разработанный ранее `CandidateService`. Теоретически, мы могли бы использовать его напрямую для построения графиков, но этот сервис оперирует сущностью Candidate, которая как мы помним содержит только идентификаторы вакансий и этапов, а на графиках, разумеется, мы хотели бы видеть человекочитаемые имена. Поэтому создадим маленькую витринку поверх `CandidateService`. Сперва создадим новую сущность в рамках пакета model.candidate и назовем ее `CandidateForDashboard`:

```
data class CandidateForDashboard(
    val candidateId: Long, val candidateName: String,
    val vacancyId: Long?, val vacancyName: String,
    val stateId: Long?, val stateName: String,
)
```

Здесь помимо идентификаторов ключевых сущностей мы будем иметь и их имена. Далее, создаем саму витрину, имплементируя простейший однометодный интерфейс `DataStorage`:

```
@Service
class CandidateServiceFacade(
    // Инжектим себе основные доменные сервисы
    private val candidateService: CandidateService,
    private val vacancyService: VacancyService,
    private val stateService: StateService,
    // Указываем наш новый класс-сущность, с которым будет работать витрина:
    override val entityClass: Class<CandidateForDashboard> = CandidateForDashboard::class.java
) : DataStorage<CandidateForDashboard> {
```

```
    // Определяем один единственный метод интерфейса:
    override fun getList(
        projection: List<String>?, searchData: Map<String, SearchCriteria>?,
            pageRequest: PageRequest?, ctx: Map<String, Any?>?, authorization:
String?
    ): Page<CandidateForDashboard> {

        // Основу данных просто забираем из candidateService:
                val page = candidateService.getList(projection, searchData,
pageRequest, ctx, authorization)

        // Дальше собираем идентфикаторы вакансий и этапов из кандидатов:
        val vacanciesIds = page.content.mapNotNull { it.vacancyId }.toSet()
        val stateIds = page.content.mapNotNull { it.stateId }.toSet()

        // Получаем сами сущности вакансий и этапов, сохраняем их в мапу:
        val vacancies = vacancyService.getByIds(vacanciesIds.toList())
        val states = stateService.getByIds(stateIds.toList())

         // Формируем итоговый список кандидатов, пригодных для использования в
витрине:
        val candidates = page.content.map {
            CandidateForDashboard(
                candidateId = it.id!!,
                candidateName = it.name,
                vacancyId = it.vacancyId,
                    vacancyName = vacancies[it.vacancyId]?.name?:"-", // Джойн
вакансий
                stateId = it.stateId,
                    stateName = states[it.stateId]?.name?:"-",         // Джойн
этапов
            )
        }

        // Отдаем данные витрины:
        return PageImpl(candidates, page.pageable, page.totalElements)
    }
}
```

Витрина данных готова. Можно переходить к построению дашбордов.

11.2. Datasource, Dataset, Monitor + Сборка страницы

В пакете `appix.core.data` существуют вспомогательные классы, которые упрощают нашу дальнейшую задачу. Откроем класс `DashboardPage` и реализуем его следующим образом:

```
@Component
class DashboardPage(
      // Заинжектим основные сервисы, благодаря которым мы сможем знать, что
именно
    // показывать пользователю на дашбоарде:
    private val authService: AuthService,
    private val vacancyService: VacancyService,
    private val vacancyRelationService: VacancyRelationService,
) : PageBuilder(
    title = "Главная страница",
             access = accessList(BuiltinRoles.ROOT, HrHeroRoles.REVIEWER,
HrHeroRoles.EDITOR)
```

```
) {

    // Сперва определяем datasource
    // Так как мы будем работать с витриной встроенной внутри приложения, а не внешней БД,
    // используем DatasourceDriver.CRUD_SERVICE:
    private val datasource = Datasource("BuiltinCruds", DatasourceDriver.CRUD_SERVICE)

    // Теперь определяем dataset
    private val candidatesDataset = Dataset(
        id = "Candidates",           // id может быть любым
        datasource = datasource,     // созданный ранее источник данных
        mapping = Mapping(           // описываем маппинг данных нашей витрины
            mapOf(
                // Для каждого поля сущности витрины мы определяем "алиас" и "тип данных":
                CandidateForDashboard::candidateId.name to Mapping.Value("ID", DataType.INTEGER),
                CandidateForDashboard::candidateName.name to Mapping.Value("Имя", DataType.STRING),
                CandidateForDashboard::vacancyName.name to Mapping.Value("Вакансия", DataType.STRING),
                CandidateForDashboard::stateName.name to Mapping.Value("Статус", DataType.STRING),
            ),
            // Указываем первичный ключ витрины:
            primaryDatetimeField = CandidateForDashboard::candidateId.name
        ),
        // И основная магия: указываем имя сервиса витрины
        crudServiceName = "candidateServiceFacade"
    )

    // Теперь создадим пару "мониторов" (aka графиков):
    val breakdownMonitor = Monitor(
        id = "breakdownMonitor",                                             // Любой уникальный id
        name = "Вакансии и статусы",                                         // Заголовок графика
        description = "Разбивка кандидатов по вакансиям и статусам",         // Подзаголовок
        dataset = candidatesDataset,                                         // Привязка к датасету
        visualizationType = MonitorVisualizationType.SIMPLE_HISTOGRAM,   // Тип визуализации (гистограмма)
        dataType = MonitorDataType.CATEGORIES_BREAKDOWN,                 // Тип данных - категория с разбивкой
        categoryTree = listOf(listOf(CandidateForDashboard::vacancyName.name)), // Первичная группировка по вакансиям
        valueBreakdownTree = listOf(CandidateForDashboard::stateName.name) // Разбивка по "этапам"
    )

    // Монитор "эффективности найма"
    val efficiencyMonitor = Monitor(
        id = "",
        name = "Конверсия в найм",
        description = "Конверсия в найм",
        dataset = candidatesDataset,
        visualizationType = MonitorVisualizationType.SIMPLE_GAUGE,   // Тип визуализации - шкала
        dataType = MonitorDataType.COUNT,
```

```
         url = CandidateController.efficiencyApi,                     // URL для
загрузки данных в монитор
        chartParams = ChartParams(
            gauges = listOf(
                Gauge(
                    minValue = 0.0,          // Минимальное значение шкалы
                    maxValue = 100.0,        // Максимальное значение шкалы
                    unit = Units.PERCENT,    // Единица измерения - проценты
                         coloringRules = listOf( // Правила цветовой индикации
шкалы:
                                 ColoringRule(min  = 0.0, max  = 4.99, color  =
Color.DANGER, rangeDescription = "Не эффективный найм"),
                                ColoringRule(min  = 5.0, max  = 19.99, color  =
Color.WARNING, rangeDescription = "Сложный найм"),
                                ColoringRule(min  = 20.0, max  = 100.0, color  =
Color.SUCCESS, rangeDescription = "Хороший найм"),
                    )
                )
            )
        )
    )

    // Теперь собственно метод построения странички:
    override fun build(authorization: String?): Page {

        // Вынесем в переменную "фильтры" для мониторов эффективности
        // Просто для удобства:
        val monitorFilters = { vacancyId: Long ->
            MonitorFilters(
                         listOf(MonitorFilter("vacancyId", EqualityOperator.EQ,
vacancyId.toString()))
            )
        }

        // Получаем список релевантных пользователю вакансий:
        val user = authService.getUser(authorization)
        val relatedVacancies = vacancyRelationService.getList(
            searchData = mapOf(
                                      VacancyRelation::userId.name   to
SearchCriteria(EqualityOperator.EQ, user.id!!)
            )
        )
            .content.mapNotNull { it.vacancyId }
            .let {
                vacancyService.getByIds(it)
            }

        return Page(
            pageBuilder = this,
            content = Container(
                listOf(
                       // В первой строке - итерируемся по доступным пользователю
вакансиям
                    // И добавляем "колонки" с MonitorWidget:
                    Row(
                        relatedVacancies.map {
                            val vacancyId = it.value.id
                            val vacancyName = it.value.name
                            Column(
                                MonitorWidget(
                                    // ID любой, но уникальный:
                                    id = "monitor_vacancy_$it.id",
```

```
                                                // Данные монитора копируем из шаблона,
добавляя только фильтры:
                                            monitorData = efficiencyMonitor.copy(
                                                name = vacancyName,
                                                conditions = monitorFilters(vacancyId)
                                            )
                                        )
                                    )
                                }
                            ),
                            // Во второй строке пустим нашу гистограмму с разбивками:
                                      Row(MonitorWidget(id = "monitor1", monitorData =
breakdownMonitor))
                    )
                )
            )
    }
}
```

Имплементация страничке получилась не короткой, да. Однако она довольно понятным образом делится на 2 логические части:

- Определение данных
- Сборка разметки страницы

11.3. Реализация кастомного расчетного показателя для мониторов

Как вы, очевидно, заметили - в коде есть ошибка, не позволяющая приложению компилироваться - референс к `CandidateController.efficiencyApi` в рамках определения шаблона монитора эффективности найма… Это сделано сознательно.

Вообще, встроенный внутри ядра фреймворка контроллер позволяет "из коробки" возвращать приличное количество данных и мы могли бы обойтись без создания своего контроллера. Но это распространяется на случай, например, если бы мы хотели вывести в монитор:

- среднее значение из датасета
- последнее значение из датасета
- абсолютное количество записей в датасете по критерию
- и некоторые другие стандартные показатели

В нашем же случай мы замахнулись на некий расчетный показатель - эффективность найма, который расчитывается по не совсем стандартным для фреймворка правилам. Поэтому алгоритм его расчета мы и выносим в отдельный контроллер, который рефернсим в параметра монитора url, то есть это механика для вывода произвольных расчетных метрик, которую в этом уроке и хотелось продемонстрировать. Реализуем этот ендпойнт и алгоритм расчета метрики в рамках контроллера `CandidateController`:

```
@RestController
@RequestMapping(CandidateController.api)
class CandidateController(
```

```
    // Как обычно инжектим необходимые сервисы:
    private val candidateService: CandidateService,
    private val stateService: StateService,
) : CrudController<Candidate, Long>(candidateService) {

    companion object {
        const val api = "/ru.rubbles.hrhero.model.candidate/candidate"
        const val dictApi = "$api/${CrudController.dictApi}"
        const val joinApi = "$api/${CrudController.joinApi}"

        // Добавляем "адрес" для нового ендпойнта как константу
        // Это является для нас просто хорошим тоном:
        const val efficiencyApi = "$api/efficiency"
    }

    // Ну и реализуем непосредственно ендпойнт с логикой расчета:
    @PostMapping("efficiency")
    @ResponseStatus(HttpStatus.OK)
    fun efficiency(
        // Сигнатура стандартная для всех "мониторов"
        // Ее всегда можно "позаимствовать" из класса
        // appix.core.data.MonitorContoller:
        @RequestHeader(name = "Authorization") authorization: String? = null,
        @ModelAttribute(RequestContext.header) requestContext: RequestContext,
        @RequestBody monitor: Monitor,
    ): ResponseWrapper<Double?> {

        // Обычно в начале мы "валидируем" присланные нам данные монитора
        // относительно ожидаемого типа возвращаемых данных:
        monitor.validate(MonitorDataType.COUNT)

        // Из фильтров монитора вытаскиваем привязку к вакансии:
          val vacancyId = monitor.conditions?.filters?.values?.first { it.field
== "vacancyId" }!!.value.toLong()

        // Получаем "этап", который у нас считается "успехом найма":
        val conversionState = stateService.getConversionState()
        // И следом количество кандидатов, который до этого этапа добрались:
        val conversionStateCount = candidateService
            .countByStateAndVacancy(conversionState?.id?:0, vacancyId)

        // Получаем общее количество кандидатов по анализируемой вакансии:
        val totalCount = candidateService
            .countByVacancy(vacancyId)
            .let { if (it == 0L) 1 else it}

        // Ну и считаем банальный процент успешных от общего числа:
        return (conversionStateCount * 100.0 / totalCount)
            .toBigDecimal()
            .setScale(2, RoundingMode.UP)
            .toDouble()
            .respondSuccess()
    }
}
```

11.4. Реализация недостающих вспомогательных методов сервисов

Итак, все на местах, но почти… опять мы видим пару мест, которых пока у нас не хватает:

- Метод StateService, который сообщал мы нам, какой этап считается "успехом"
- Метод CandidateService, который сообщал бы нам количество кандидатов по заданным вакансии и этапу
- Метод CandidateService, который сообщал бы нам общее количество кандидатов по заданной вакансии (по всем этапам)

Последний рывок. Реализуем:

```
@Service
class StateService(
    stateRepository: StateRepository
) : CrudServiceJpa<State, Long>(stateRepository, State::class.java) {

    fun getConversionState(): State? {
        return getList(
            searchData = mapOf(
                   State::name.name to SearchCriteria(EqualityOperator.EQ, "Офер
принят")
            )
        ).content.firstOrNull()
    }
}
```

Теперь пара методов `CandidateService`:

```
@Service
class CandidateService(
    // Обозначаем используемый репозиторий как val,
    // чтобы он был доступен в методах класса
    val candidateRepository: CandidateRepository,
    // ... остальные инжекты без изменений
) : CrudServiceJpa<Candidate, Long>(candidateRepository, Candidate::class.java)
{

    // ...
    // Тут все без изменений с прошлых уроков

    fun countByVacancy(vacancyId: Long): Long =
        candidateRepository.countByVacancyId(vacancyId)

    fun countByStateAndVacancy(stateId: Long, vacancyId: Long): Long =
        candidateRepository.countByStateIdAndVacancyId(stateId, vacancyId)
}
```

И, так как мы используем JPA-репозиторий `CandidateRepository`, добавляем и ему тоже 2 новых метода выборки данных:

```
interface CandidateRepository : JpaRepository<Candidate, Long> {
    fun countByVacancyId(vacancyId: Long): Long
    fun countByStateIdAndVacancyId(stateId: Long, vacancyId: Long): Long
}
```

Теперь все должно компилироваться.

11.5. Генерация тестовых данных

Чтобы безболезненно оценить результат нашей работы, дополним класс `InitData` генерацией дополнительных тестовых данных:

```
@Component
class InitData(
    // Добавляем необходимых сервисов:
    usersService: BuiltinUsersService,
    acsService: BuiltinAcsService,
    vacancyService: VacancyService,
    stateService: StateService,
    vacancyRelationService: VacancyRelationService,
    stateRelationService: StateRelationService,
    candidateService: CandidateService,
) {

    // Простейший генератор имен =)
    private val names = listOf(
        "Андрей", "Иван", "Алексей", "Виктор", "Петр", "Артем", "Дмитрий",
"Кирилл", "Олег",
        "Михаил", "Игорь", "Василий", "Сергей", "Виктория", "Любовь", "Анна",
"Мария", "Юлия",
        "Никита", "Николай", "Антон", "Александра"
    ).let {
        listOf(it, it, it, it, it)
    }.flatten().shuffled()

    init {

        // ...
        // Тут прежние данные по созданию пользователей и настройке доступов:
        // ...

        // Создадим пару вакансий:
        val vacancy1 = vacancyService.createOne(Vacancy(null, "Тестировщик"))
        val vacancy2 = vacancyService.createOne(Vacancy(null,
"Java-разработчик"))
        val vacancy3 = vacancyService.createOne(Vacancy(null, "Системный
аналитик"))

        // Создадим несколько статусов:
        val state1 = stateService.createOne(State(null, "Новые"))
        val state2 = stateService.createOne(State(null, "Собеседование
назначено"))
        val state3 = stateService.createOne(State(null, "Офер выставлен"))
        val state4 = stateService.createOne(State(null, "Офер принят"))
        val state5 = stateService.createOne(State(null, "Отказ"))

        // Дадим доступ reviewer1 и editor1 ко всем 3 вакансиям:
        listOf(reviewer1, editor1).forEach {
            vacancyRelationService.createOne(
                VacancyRelation(null, vacancy1.id!!, it.id.toString())
            )
            vacancyRelationService.createOne(
                VacancyRelation(null, vacancy2.id!!, it.id.toString())
            )
            vacancyRelationService.createOne(
                VacancyRelation(null, vacancy3.id!!, it.id.toString())
            )
        }

        // Дадим доступ reviewer1 и reviewer2 ко всем некоторым этапам:
        listOf(reviewer1, reviewer2).forEach {
            stateRelationService.createOne(
                StateRelation(null, state1.id!!, it.id.toString())
            )
            stateRelationService.createOne(
```

```
                StateRelation(null, state2.id!!, it.id.toString())
            )
        }

        // Сгенерируем таблицу кандидатов:
        repeat(60) {
            candidateService.createOne(
                Candidate(
                    id = null,
                    name = "${names[it]}
${"АБВГДЛПРСТФЛНКЯ".toList().random()}.",
                    url = "",
                    vacancyId = listOf(vacancy1, vacancy2,
vacancy3).random().id,
                    stateId = listOf(state1, state2, state3, state4,
state5).random().id,
                ),
                mapOf(),
                listOf(editor1, editor2).random().token,
            )
        }
    }
}
```

В этом уроке мы разбирали не только возможности фреймворка, но и продолжили работать по части бизнес-логики: "успешные" этапы найма, оценка эффективности, и так далее. Мы разобрали определение datasource, dataset, а также познакомились с несколькими (из многих) типов мониторов (графиков). Как итог, нам удалось построить симпатичную страничку, которая показывает пользователю данные по найму и при этом учитывает ролевую модель, заложенную в приложении ранее. Выглядит этак примерно так:

Рис.11.1 – Титульный дашборд

12. CRUD

12.1. Базовый функционал

Ядро CRUD слоя находится в пакете `appix.core.crud`. Основными классами в нем являются `CrudService`, `CrudServiceJpa` и `CrudController`. CRUD слой в фреймворке Appix реализован в generic виде, что позволяет максимально переиспользовать общий функционал и не дублировать boilerplate-код.

Для того чтобы получить возможность использования встроенных CRUD-операций, разработчику необходимо выполнить ряд действий, описанных ниже. После выполнения указанных действий будут доступны методы встроенные в `CrudController`, но их можно будет переопределить, а также добавить новые. Методы встроенные в `CrudController`:

- `getList POST '/list';`
- `getOne GET '/{id}';`
- `post POST '/';`
- `patch PATCH '/{id}';`
- `move PATCH '/move';`
- `delete DELETE '/{id}';`
- `getPage GET '/{id}/page';`
- `getTableDefinition GET '/tableDefinition';`
- `dict GET '/dict';`
- `join POST '/join';`
- `joinWithDependency POST 'join/dep';`
- `addFile POST '/addFile'.`

Для того чтобы имплементировать работу с CRUD для новой кастомной сущности, и тем самым получить возможность воспользоваться встроенными CRUD-операциями, необходимо сделать следующее:

1) Создать новую сущность `Entity` (если новая сущность связанна с таблицей в БД). На поля сущности необходимо навесить аннотацию `@Property` с нужным значением `Property.subType`. Например для полей, которые являются первичными ключами, нужно использовать аннотацию со значением: `@Property(subType = DataSubType.PK)`. Для полей, которые требуется использовать для декларативной сортировки, необходимо использовать аннотацию со значением: `@Property(subType = DataSubType.ORDINAL)`. Порядок полей для сортировки соответствует порядку полей с аннотацией `@Property(subType = DataSubType.ORDINAL)` в классе сущности.

2) Создать файл миграции (.sql) со схемой таблицы для сущности (опционально).

3) Создать репозиторий наследник от `JpaRepository` (если была создана новая сущность, которая связанная с таблицей в БД).

4) Создать сервис наследник от CrudService или CrudServiceJpa.

5) Создать контроллер наследник от CrudController и указать в нем корневой путь для REST API.

6) Создать таблицу и виджет, объекты класса Table, TableWidget.

7) Создать страницу, объект класса Page. В составе страницы должен находиться созданный виджет TableWidget.

8) Объект класса Table необходимо сконфигурировать, а именно:

- Указать в поле path CRUD API для данной сущности.
- Указать параметры pagination.
- Указать в поле presence соответствие доступных CRUD действий -> поля сущности (Entity).

В результате фронтенд будет иметь все данные для отображения необходимых компонентов (виджетов) для операций с данной сущностью.

12.2. Оператор присоединения (join)

Для связки сущностей существует возможность использовать оператор соединения – join. Реализовать join в Appix, можно следующим образом:

- В объекте класса Table, для кастомной сущности, необходимо задать поле Table.joins. Это поле содержит соответствие Foreign Key -> CRUD API path сущности, которую необходимо присоединить.
- Generic CrudController имеет методы join() и joinWithDependency(), которые будут использованы для join-запросов, поэтому необходимо иметь наследника от CrudController.

12.3. Фильтры

В Appix существует возможность задать кастомную фильтрацию для полей сущности по определенным правилам. Правила фильтрации содержатся в enum SearchCriteriaOperator.

Реализовать кастомную фильтрацию в Appix, можно следующим образом:

- В объекте класса Table, для кастомной сущности, необходимо задать поле Table.search. Это поле содержит соответствие: Поле сущности -> Значение из enum SearchCriteriaOperator.
- На фронтенде появится возможно заполнить указанные поля для фильтрации.
- Generic CrudController в методе getList() принимает параметры заполненных полей в виде соответствия из п. 1 трансформирует их в предикаты Criteria API и использует для фильтрации, поэтому необходимо иметь наследника от CrudController.

12.4. Деревья

Дополнительно к стандартным CRUD-операциям, в фреймворке Appix, доступны дополнительные методы Tree CRUD API для древовидных (иерархичных) сущностей.

Для того чтобы получить возможность воспользоваться дополнительными CRUD-операциями, разработчику необходимо выполнить ряд действий, описанных ниже. После выполнения указанных действий будут доступны методы, встроенные в CrudController, но их можно будет переопределить, а также добавить новые.

Методы Tree CRUD API встроенные в CrudController:

- getTree POST '/tree';
- treeMove PATCH '/tree/move';
- deleteTree DELETE '/tree/delete/{id}'.

Для того чтобы получить возможность использовать дополнительные Tree CRUD-операции, необходимо выполнить действия, описанные в статье Implementing CRUD (Part 1 - Basics), кроме этого нужно сделать следующее:

- Кастомная сущность Entity должна иметь ровно одно поле, необходимое для хранения древовидной структуры. Пример такого поля:

```
@javax.persistence.Column(name = "parentId")
    @IntProperty(subType = DataSubType.PARENT_ID)
    var parentId: Long? = null
```

Как видно из примера, поле должно содержать поле DataSubType.PARENT_ID.

1) Создать виджет для дерева-таблицы, объект TreeTableWidget.

2) Создать страницу, объект класса Page. В составе страницы должен находиться созданный виджет TreeTableWidget.

Примером использования Tree CRUD в текущей кодовой базе являются объекты для построения меню (MenuItem): MenuItemEntity, MenuService, MenuController, menuItemsTable, TreeTableWidget.

1)

14. Спецификация пользовательского интерфейса

Актуальная детальная техническая спецификация контрактов, моделей и виджетов находится по адресу
https://vue.rosres.rubbles-crm.ru/content/docs/frontend/contracts/api-wrapping