

**Дипломна работа**

**Тема:**

„Проектиране и реализация на система за кандидатстудентски прием“

Специалност: Софтуерно инженерство

Дипломант:......................... Ръководител: …………

/Данаил Пламенов Михайлов/ /ас. Велислав Колесниченко/

фак.№ 23651231

Съдържание

[1. Увод 4](#__RefHeading___Toc3581_2427203700)

[1.1. Дефиниране на целите на бизнеса спрямо проучването на конкурентите 5](#__RefHeading___Toc4589_3569542567)

[2.0 Анализ на проблема и избор на технологии 6](#__RefHeading___Toc3581_2427203700_Copy_1)

[2.1 Инженерен проблем 6](#__RefHeading___Toc4466_3569542567)

[2.2 Избор на технологии 9](#__RefHeading___Toc4468_3569542567)

[2.2.1 Език за програмиране 9](#__RefHeading___Toc4591_3569542567)

[2.2.1.1 Java 9](#__RefHeading___Toc4404_3569542567)

[2.2.1.2 Javascript 9](#__RefHeading___Toc5821_3082079080)

[2.2.2 Фреймуърк (Framework) 10](#__RefHeading___Toc4593_3569542567)

[2.2.2.1 Quarkus 10](#__RefHeading___Toc4406_3569542567)

[2.2.2.2 ReactJS 11](#__RefHeading___Toc4410_3569542567)

[2.2.3 Контейнеризация 11](#__RefHeading___Toc15800_1525090346)

[2.2.3.1 Docker 12](#__RefHeading___Toc4416_3569542567)

[2.2.4 База от данни 13](#__RefHeading___Toc4597_3569542567)

[2.3 Функционалности: 13](#__RefHeading___Toc1305_3579324838)

[3.0 Проектиране на системата 15](#__RefHeading___Toc3581_2427203700_Copy_2)

[3.1 Концептуален модел на базата от данни 17](#__RefHeading___Toc4470_3569542567)

[3.2 Модел на Чен диаграма 20](#__RefHeading___Toc3968_953106589)

[3.3 Architecture диаграма 21](#__RefHeading___Toc3970_953106589)

[3.4 UML Диаграми 23](#__RefHeading___Toc13000_1525090346)

[3.4.1 Class диаграма 24](#__RefHeading___Toc13002_1525090346)

[3.4.2 Use case диаграма 26](#__RefHeading___Toc11420_1525090346)

[4. Реализация на системата 27](#__RefHeading___Toc3972_953106589)

[4.1 База данни 70](#__RefHeading___Toc1913_2923609595)

[5. Тестове 75](#__RefHeading___Toc3974_953106589)

[6. Заключение 89](#__RefHeading___Toc3978_953106589)

[Литература 91](#__RefHeading___Toc3980_953106589)

# **1.** **Увод**

Системите за кандидатстудентски прием играят ключова роля в модернизирането на процесите, свързани с образованието и достъпа до висши учебни заведения. Те не само улесняват административните дейности, но също така осигуряват прозрачност, ефективност и персонализирано обслужване за кандидат-студентите. С напредването на технологиите дигитализацията на тези системи се превръща в основен инструмент за управление на големи обеми от данни и за автоматизиране на сложни процедури, свързани с изпити, класирания и записване.

Дигитализираната система за кандидатстудентски прием предоставя възможност на кандидат-студентите да се регистрират, да заявяват участие в изпити, да подават документи за класиране и да следят своя статус в процеса на прием. Това води до по-добра организация и спестява време както за кандидатите, така и за административните служители. Освен това, системата предлага адаптивност и прозрачност чрез функционалности като проверка на документи, избор на предпочитани специалности и известяване за предстоящи срокове.

От гледна точка на администратора, системата дава възможност за централизирано управление на процесите по класиране, както и за анализ и справки, които могат да се използват за подобрение на бъдещите кампании. Благодарение на интегрираните инструменти за генериране на отчети и управление на данни, администраторите могат да вземат информирани решения, които отговарят на нуждите както на кандидатите, така и на институцията.

Тази система представлява не просто технологично средство, а ключов фактор за подобряване на достъпа до образование и за осигуряване на по-добро планиране и управление в процеса на кандидатстудентския прием.

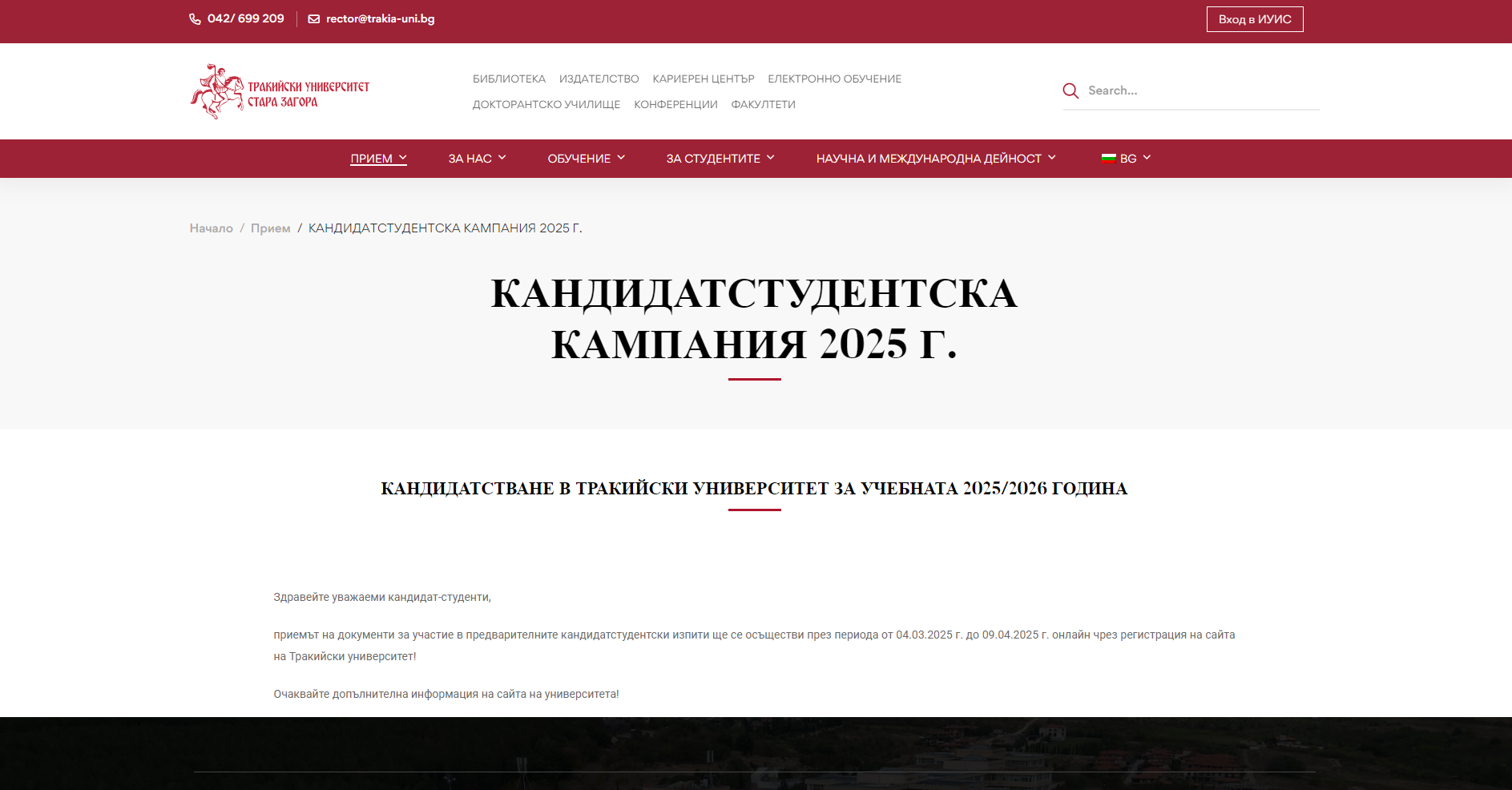
## **Дефиниране на целите на бизнеса спрямо проучването на конкурентите**

**Примерен прием на УННС**

****

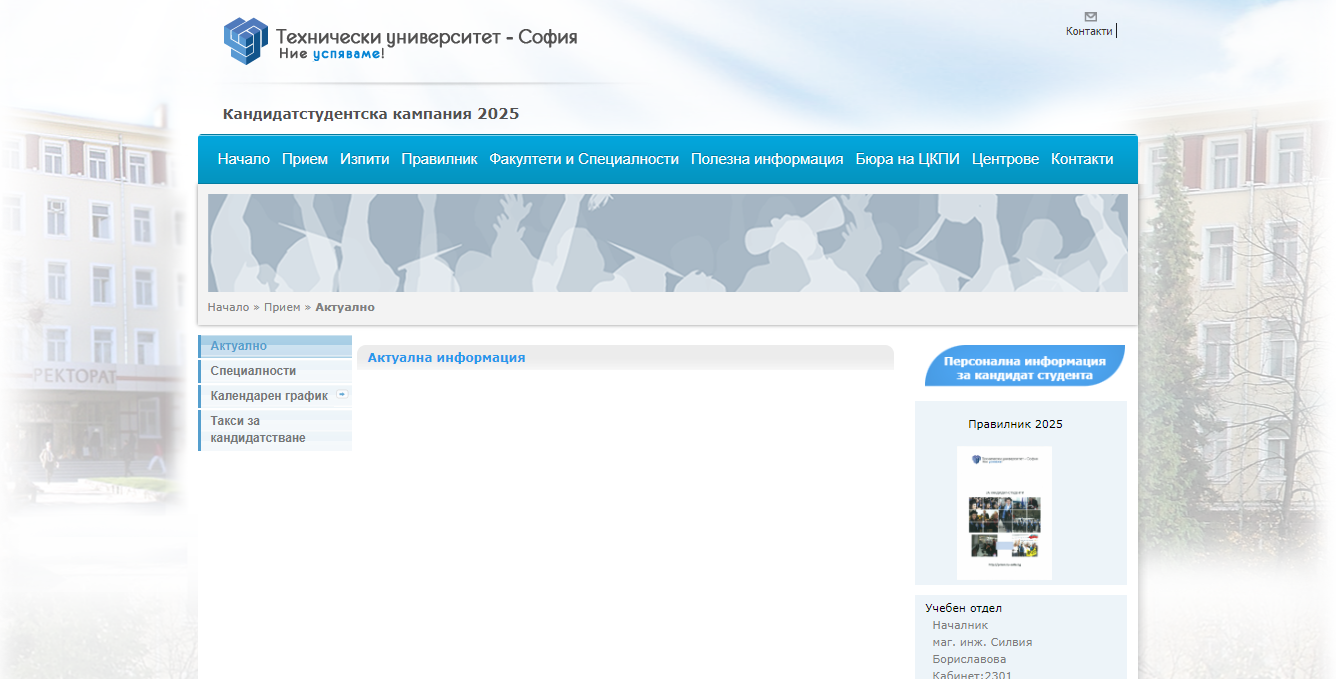
Фигура 1

**Примерен прием на Тракийски университет**



Фигура 2

**Примерен прием на ТУ София**

Фигура 3

# **2.0** **Анализ на проблема и избор на технологии**

Разработването на система за кандидатстудентски прием е предизвикателство, което изисква внимателно планиране и анализ на функционалните изисквания, за да се осигури лесен и ефективен процес за кандидатите и администрацията. Целта на тази система е да дигитализира и оптимизира приема на нови студенти, като предостави интуитивен интерфейс за кандидатите и мощни инструменти за управление на процесите за административния персонал.

## **2.1 Инженерен проблем**

Проектирането на система за кандидатстудентски прием изисква задълбочено разбиране на нуждите на различните участници в процеса. След анализ на конкуриращи системи и съществуващи решения, както и идентифициране на техните предимства и недостатъци, бяха дефинирани основните функционалности, които приложението трябва да поддържа.

**Изисквания на системата:**

1. **Регистрация и управление на профили:**
   * Кандидат-студентите трябва да могат да се регистрират в системата, като попълват всички задължителни данни.
2. **Записване за изпити:**
   * Възможност за избор на изпит и предпочитана дата за явяване.
   * Автоматично изчисляване и предоставяне на такса за изпита.
   * Качване на платежни документи и тяхната проверка от администратор, след което кандидатите получават право за явяване на изпита.
3. **Кандидатстване за класиране:**
   * Заявяване на използване на оценки от държавни зрелостни изпити (ДЗИ).
   * Прикачване на документи, необходими за класирането.
   * Подреждане на предпочитаните специалности и уведомяване при липса на право за кандидатстване в определени специалности, като се посочват изискваните изпити.
4. **Класиране и записване:**
   * След провеждането на класирането кандидатите трябва да могат да проверят специалността, в която са приети, както и сроковете за записване.

**Функционалности за администраторите:**

1. **Управление на класирането:**
   * Стартиране на процеса на класиране въз основа на предварително дефинирани правила.
   * Въвеждане и редактиране на информация за специалностите и критериите за класиране.
2. **Справки и отчети:**
   * Проследяване на класирането на студентите и специалностите.
   * Изтегляне на справки във формати като PDF и CSV.
3. **Управление на изпити:**
   * Проверка и въвеждане на резултати от кандидатстудентските изпити (КСИ).

Тази система цели да минимизира времето и ресурсите, необходими за обработката на данни, като предостави удобен достъп до информация и автоматизирани решения за основните процеси в кандидатстудентския прием.

Задачи на приложението:

Разработването на система за кандидатстудентски прием изисква внимателно планиране на основните задачи, които ще гарантират ефективност, удобство за потребителите и лесно управление от страна на администрацията. Въз основа на зададените изисквания, задачите на приложението са следните:

**1. Системен дизайн:**

Проектиране на интуитивен потребителски интерфейс, който осигурява лесна навигация за кандидат-студентите и администраторите. Интерфейсът трябва да покрива всички ключови функции, като регистрация, записване за изпити, класиране и управление на документи. Внедряването на сигурна система за управление на идентичността е от съществено значение, за да се осигури защита на личните данни.

**2. Интеграция:**

Създаване на единен интерфейс за комуникация между бекенд и фронтенд компоненти на системата. Тази интеграция ще гарантира надеждността на процесите като подаване на документи, обработка на плащания и актуализация на данни в реално време.

**3. Автоматизирано тестване:**

Въвеждане на система за автоматично тестване, която ще гарантира функционалността и стабилността на приложението при промени в кода. Товавключва тестване на модули за регистрация, записване за изпити, класиране и административен панел.

**4. Среда за разработване:**

Създаване на лесна за използване и възпроизводима (reproducible) среда за разработване. Тя трябва да описва всички зависимости на приложението по ясен и декларативен начин, което ще улесни бъдещото разширяване и поддръжка на системата.

**5. Система за уведомяване:**

Разработване на надеждна система за известяване, която да информира кандидат-студентите за статуса на техните кандидатури, записване за изпити, резултати от класирането и срокове за записване. Уведомленията ще се изпращат чрез имейл или вътрешни съобщения в системата.

**6. Административен панел**:

Създаване на сигурен и удобен за използване административен панел, който позволява управление на профили, проверка на платежни документи, стартиране и управление на класиране, както и генериране на справки и отчети в PDF или CSV формат.

## **2.2 Избор на технологии**

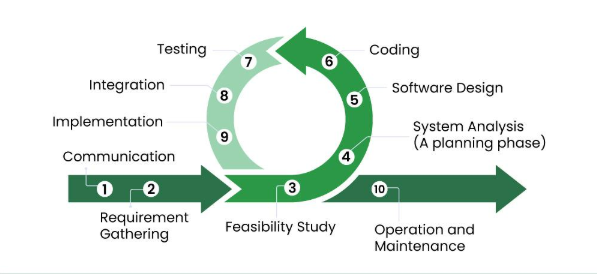
Необходимите програмни компоненти за имплементирането на информационната система са разпределени по следните категории:

### **2.2.1 Език за програмиране**

#### 2.2.1.1 Java

Java е обектно-ориентиран програмен език на високо ниво, който предоставя стабилност и широк набор от инструменти за създаване на надеждни и мащабируеми приложения.

* **Компилация и изпълнение:** Java изходният код се компилира в байткод, който се изпълнява от Java Virtual Machine (JVM), осигурявайки преносимост на приложенията.
* **Многоплатформеност:** Java приложенията могат да работят върху различни операционни системи без промени в кода.

*Фигура 4*

#### 2.2.1.2 Javascript

JavaScript е динамичен език, използван за създаване на интерактивни уеб приложения.

* **Скриптове:** Скриптовете могат да се интегрират директно в HTML и да се изпълняват от уеб браузъра на клиента.
* **Гъвкавост:** Интерпретаторният характер на езика позволява бързо разработване и тестване на уеб функционалности.

### **2.2.2 Фреймуърк (Framework)**

#### 2.2.2.1 Quarkus

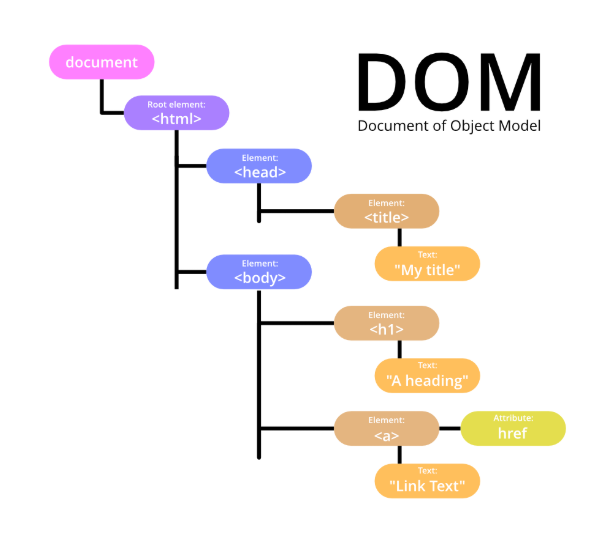
Quarkus е модерен Java фреймуърк, оптимизиран за съвременни облачни и контейнеризирани среди.

* **Основни предимства:** Позволява създаване на „serverless“ приложения, интегрира се с GraalVM за изпълнение на native executables и предлага бързо стартиране и ниско потребление на памет.
* **Dependency Injection:** Поддържа инжектиране на зависимости, което улеснява разширяемостта и поддръжката на приложенията.

#### 2.2.2.2 ReactJS

ReactJS е JavaScript библиотека за изграждане на динамични потребителски интерфейси.

* **Компонентно-базиран подход:** Позволява създаването на многократно използваеми компоненти.
* **Виртуален DOM:** Осигурява бърза и ефективна работа с елементи на потребителския интерфейс.

*Фигура 5*

### **2.2.3 Контейнеризация**

Контейнеризацията е техника за пакетиране на приложения с всички необходими зависимости и библиотеки, за да бъдат лесно преносими и изпълними в различни среди.

* **Използвани технологии:** Контейнерите осигуряват изолация и последователност при разгръщане на приложенията в облачни среди или локални инфраструктури.
* **Предимства:** Лесно разгръщане, мащабируемост и управление на версиите.

#### **2.2.3.1 Docker**

Docker е платформа, базирана на архитектура клиент-сървър, която позволява множество клиенти да контролират внедряването и управлението на контейнери върху един сървър. Тя предоставя стандартизиран начин за изграждане, пренасяне и изпълнение на приложения в изолирани среди.

**1.Шаблон**  
Преди създаването на контейнер, е необходимо да се изготви шаблон („template“), наричан още Dockerfile.

* **Dockerfile:** Файл, съдържащ инструкции за изграждане на Docker изображение.
* **Процес на изграждане:** Docker Engine изпълнява инструкциите в Dockerfile, за да създаде изображение.

**2.Изображение**  
Docker изображението представлява изпълним пакет от софтуер, включващ всички необходими зависимости за приложението.

* **Роля:** Изображението дефинира как да се създаде контейнер.
* **Използваемост:** След построяване, изображението може да бъде многократно използвано за създаване на множество идентични контейнери.

**3.Контейнер**  
Контейнерът е изолирана виртуална среда за изпълнение на приложения, изградена въз основа на Docker изображение.

* **Съдържание:** Всеки контейнер включва код, библиотеки, системни и програмни зависимости, необходими за стартиране на приложението.
* **Изолация:** Контейнерите са независими един от друг, което гарантира стабилност и сигурност при изпълнението на множество приложения на една и съща система.

Docker значително опростява процеса на внедряване и управление на софтуерни приложения, като осигурява висока преносимост и лесна интеграция в различни инфраструктури, включително облачни среди и локални сървъри.

### **2.2.4 База от данни**

Информационната система за кандидатстудентски прием изисква надеждна, мащабируема и функционално богата база от данни за управление на информацията за кандидатите, изпитите и класиранията.

**2.2.4.1PostgreSQL**  
PostgreSQL е мощна, отворена обектно-релационна система за управление на бази данни (DBMS), която разширява SQL със съвременни функции за обработка и съхранение на данни.

## **2.3 Функционалности:**

1. **Обектно-релационен модел:**
   * PostgreSQL поддържа обектно-релационен модел, което позволява обработка на сложни структури и операции.
   * Функции: сложни SQL заявки, управление на транзакции, многопотребителско управление, наследяване на таблици и масиви.
2. **Разширяемост:**
   * Системата позволява дефиниране на потребителски оператори, функции, типове данни и методи за достъп.
   * Тази разширяемост я прави подходяща за интегриране на специфични бизнес изисквания.
3. **Референтна цялост:**
   * Осигурява надеждна валидност на данните чрез използване на първични и външни ключове, както и ограничения.
4. **Между-програмно езиково API:**
   * PostgreSQL предоставя интерфейси за различни програмни езици, включително:
     + **Pascal**
     + **Python**
     + **Perl**
     + **PHP**
     + **ODBC**
     + **Java/JDBC**
     + **Ruby**
     + **TCL**
     + **C/C++**
     + **Pike**

PostgreSQL е избрана заради своята стабилност, възможност за работа с големи обеми данни и широката поддръжка на програмни езици. Тези качества я правят идеален избор за база данни на системата за кандидатстудентски прием.

# **3.0 Проектиране на системата**

В процеса на разработка на системата за кандидатстудентски прием едно от най-важните стъпала е **проектирането на системата**. То представлява целенасочен процес, при който се изгражда логическа и техническа структура, съобразена с бизнес изискванията, нуждите на крайните потребители и добрите практики в софтуерното инженерство.

Проектирането започва с анализ на функционалностите и нуждите на всички участници – кандидат-студенти и администрация – и завършва с изработване на конкретни технологични решения. Системата е базирана на клиент-сървърна архитектура, използваща REST API за комуникация между потребителския интерфейс и сървърната логика. Бекендът е реализиран с помощта на Java и фреймуърка Quarkus, който е подходящ за съвременни облачни и контейнеризирани среди, а за клиентската част е използван ReactJS, комбиниран с NextJS за динамично зареждане на съдържанието. За съхранение и управление на данните е използвана релационна база данни PostgreSQL, а за лесно внедряване и управление в различни среди е приложена Docker-базирана контейнеризация.

**Структурата на проекта** е организирана модулно, като всеки модул отговаря за конкретна функционалност в системата. Проектът е разделен на три основни слоя – клиентски, сървърен и слой за база от данни. Клиентският слой предоставя удобен интерфейс на крайния потребител, сървърният обработва логиката и заявките, а базата от данни съхранява цялата необходима информация. Кодът на проекта е структуриран така, че всеки модул съдържа свои контролери, сервизни класове и репозитории. Така се осигурява ясно разделение на отговорностите и улеснява поддръжката и бъдещото разширяване на системата.

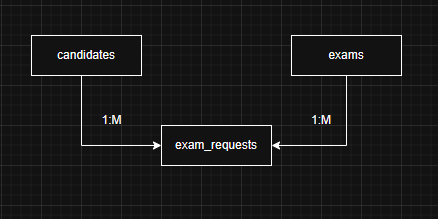
**Дефинирани са отделни модули**, всеки от които изпълнява конкретна роля в общото функциониране на системата. Модулът за удостоверяване на самоличност (Authentication) обхваща регистрацията на нови потребители, хеширане на паролите, изпращане на имейл с начална парола и генериране на токени за вход. Модулът за управление на изпити позволява на администратора да създава, редактира или изтрива изпити, както и да преглежда всички текущи. Заявките за изпит се обработват чрез отделен модул, който проверява валидността на подадените документи, приема или отхвърля заявления и следи за статуса на всяка заявка. Модулът за класиране изчислява крайния бал на кандидатите по зададена формула и ги подрежда според резултатите. Съществува и модул за уведомяване, чрез който системата автоматично изпраща имейли при успешна регистрация, промяна в заявките или резултатите от класирането.

След като са дефинирани модулите, всеки от тях е **проектиран детайлно с конкретна логика и взаимодействие между компонентите**. Например, при регистрация на кандидат, контролерът получава данните от потребителя и ги предава към сервизен клас, който ги валидира, създава нов потребителски обект и го записва в базата данни. Същият процес включва генериране на потребителско име, хеширане на паролата и изпращане на автоматичен имейл. Всеки изпит, който се създава от администратор, минава през контролер, който проверява входните данни, и при валидност ги подава към съответното хранилище, което записва изпита в базата. Подобна структура е приложена и при останалите модули. Всеки метод е логически обособен, със стриктна валидация, обработка на грешки и обратна връзка към потребителя.

Последният ключов елемент в разработката е **прототипът на потребителския интерфейс**. Системата разполага с два основни изгледа – за кандидат-студенти и за администратори. Кандидатите имат достъп до форма за регистрация, страница за вход и интерфейс за подаване на заявки за изпити. След като са се регистрирали, те могат да преглеждат своите заявки, да видят резултатите от изпитите си, както и да получават информация относно класирането. Интерфейсът е изграден така, че да бъде интуитивен и лесен за използване, с падащи менюта, бутони за действие и таблици за визуализиране на данните. От страна на администратора е осигурен специален панел, който позволява управление на всички кандидати, изпити и заявки. Там могат да се добавят нови изпити, да се одобряват или отхвърлят заявления, да се въвеждат оценки и да се преглежда класирането по специалности. Интерфейсът е реализиран с помощта на React, който позволява динамично зареждане на съдържание и бърза реакция при действия от страна на потребителя. Структурата на интерфейса отговаря напълно на съвременните изисквания за уеб приложения, включително мобилна адаптивност, ясна навигация и визуална обратна връзка при всяко действие.

## **3.1 Концептуален модел на базата от данни**

Таблиците на приложението са разпределени, според изискванията на заданието.



Фигура 6

Диаграмата представя структурата на база данни, която съдържа три основни таблици: candidates, exam\_requests и exams. Таблицата candidates съхранява информация за кандидатите, включително техните лични данни като име, фамилия, ЕГН, адрес, имейл, училище и др. Всеки запис в тази таблица представлява един кандидат.

Таблицата exams съдържа информация за изпитите. За всеки изпит се пазят данни като дата и час на провеждане, предмет и специалност. Всеки запис в таблицата exams представлява един конкретен изпит.

Централна роля в структурата на базата от данни играе таблицата exam\_requests, която съхранява заявките на кандидатите за участие в изпити. Тази таблица е свързваща и съдържа външни ключове към двете останали таблици – candidate\_id, който сочи към candidates.id, и exam\_id, който сочи към exams.id. Освен идентификаторите, тя съдържа и допълнителна информация за самата заявка, като например резултат от изпита, статус на заявката, оценка от дипломата и дата на подаване на заявката.

Връзките между таблиците са от тип "един към много" (1:M). Един кандидат може да подаде множество заявки за различни изпити, което означава, че между таблицата candidates и таблицата exam\_requests има връзка тип "един към много". По същия начин, един изпит може да бъде заявен от множество кандидати, което създава връзка тип "един към много" между таблицата exams и exam\_requests.

1. Candidate

* id: идентификатор на кандидата – int
* name: първо име на кандидата – string
* surname: фамилно име – string
* lastname: бащино име – string
* latin\_name: първо име на латиница – string
* latin\_surname: фамилно име на латиница – string
* latin\_lastname: бащино име на латиница – string
* date\_of\_birth: дата на раждане – date
* egn: ЕГН – string
* place\_of\_birth: място на раждане – string
* city: населено място – string
* district: област – string
* municipality: община – string
* address: адрес – string
* phone: телефонен номер – string
* email: електронна поща – string
* password: парола – string
* admin: администраторски права – boolean
* reset\_token: токен за възстановяване на парола – string
* reset\_token\_expiry: срок на валидност на токена – datetime
* id\_number: номер на лична карта – string
* id\_issued\_by: издател на личната карта – string
* date\_id\_created: дата на издаване на личната карта – date
* school\_name: име на училището – string
* school\_city: населено място на училището – string
* secondary\_education: вид средно образование – string

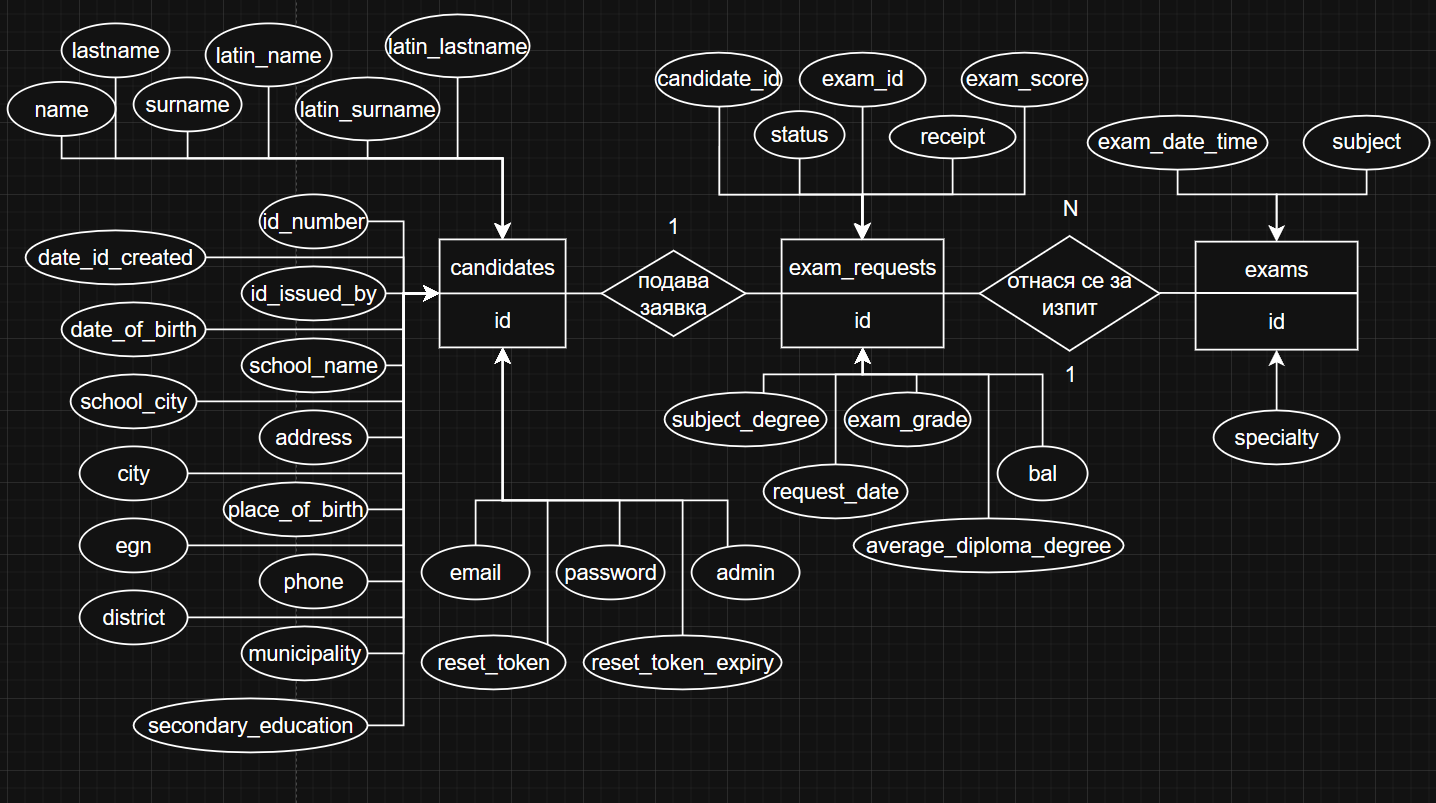
2. ExamRequest

* id: идентификатор на заявката – int
* candidate\_id: идентификатор на кандидата – int
* exam\_id: идентификатор на изпита – int
* exam\_score: резултат от изпита – float
* status: статус на заявката – string
* receipt: потвърждение за получен документ – boolean
* exam\_grade: оценка от изпита – float
* subject\_degree: оценка по профилиращ предмет – string
* average\_diploma\_degree: среден успех от дипломата – float
* bal: сбор от оценки (бал) – float
* request\_date: дата на подаване на заявката – date

3. Exam

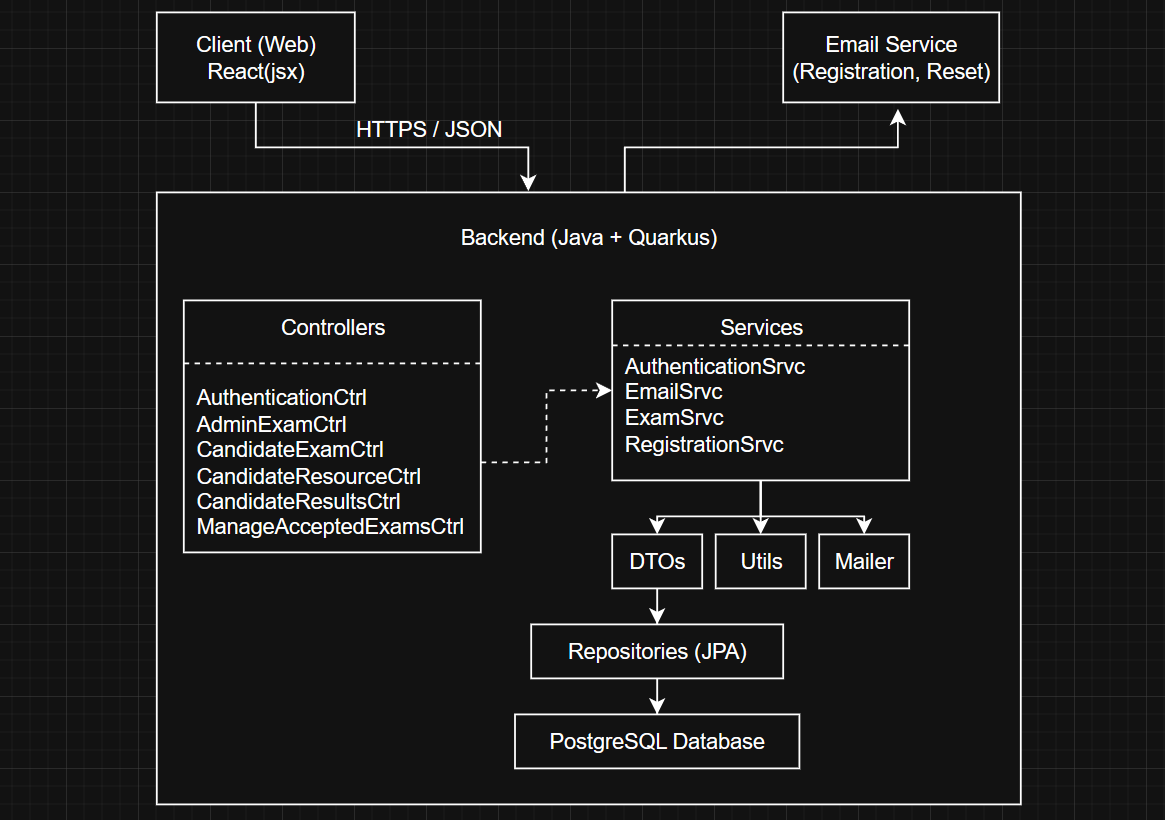
* id: идентификатор на изпита – int
* exam\_date\_time: дата и час на провеждане – datetime
* subject: предмет на изпита – string
* specialty: специалност, към която се отнася изпитът – string

## **3.2 Модел на Чен диаграма**



Фигура 7

## **3.3 Architecture диаграма**

Фигура 8

Системата за кандидатстудентски прием е изградена по модел на **трислойна архитектура**, която разделя приложението на три основни слоя: клиентски (frontend), сървърен (backend) и база от данни. Всеки от тези слоеве изпълнява строго определена роля и комуникира с останалите чрез добре дефинирани интерфейси. Тази архитектура е внедрена и изпълнена с помощта на модерни технологии като React, Java с Quarkus и PostgreSQL.

Потребителят работи с **уеб клиент**, създаден с помощта на React JSX. Този клиент предоставя интерактивен и интуитивен интерфейс, чрез който кандидатът може да се регистрира, да подава документи, да избира изпити и да следи класирането си. Всички действия, които потребителят извършва през браузъра, се изпращат до сървъра под формата на HTTP заявки (REST API).

В сървъра се намира **бекенд частта**, реализирана с Java и фреймуърка Quarkus. Това е сърцето на системата, където се случва цялата логика. Тук се намират така наречените **контролери**, които приемат входните заявки от клиента. Контролерите са отговорни за това да получат данни, да ги валидират и да ги препратят към **сервизните класове (services)**, където е концентрирана бизнес логиката. Например, когато потребителят се регистрира, контролерът за автентикация (AuthenticationController) получава информацията от формата и я предава към AuthenticationService, който създава парола, хешира я, генерира потребителско име, създава обект за кандидат и го записва в базата. След това изпраща автоматичен имейл на кандидата с данните му за вход. Това се извършва с помощта на специален **Mailer клас**, който използва SMTP за изпращане на съобщения.

Сервизните класове работят с така наречените **репозитории (repositories)**. Това са компоненти, които работят директно с **базата от данни** – записват, извличат, актуализират или изтриват обекти. За разлика от контролерите, които се занимават с входа и изхода, и от услугите, които обработват логиката, репозиторитата комуникират с релационната база от данни PostgreSQL, където се съхранява цялата информация за кандидатите, изпитите, заявките, класиранията и др.

Данните, които се предават между слоевете, не се подават директно като ентитети от базата, а се капсулират в така наречените **DTO обекти (Data Transfer Objects)**. Това е добър подход, който изолира вътрешната структура на данните от външния свят и позволява лесно контролиране каква информация се показва и каква не. Например, CandidateDTO съдържа само необходимите за регистрация полета, без да излага цялата вътрешна логика или чувствителна информация като хеширани пароли.

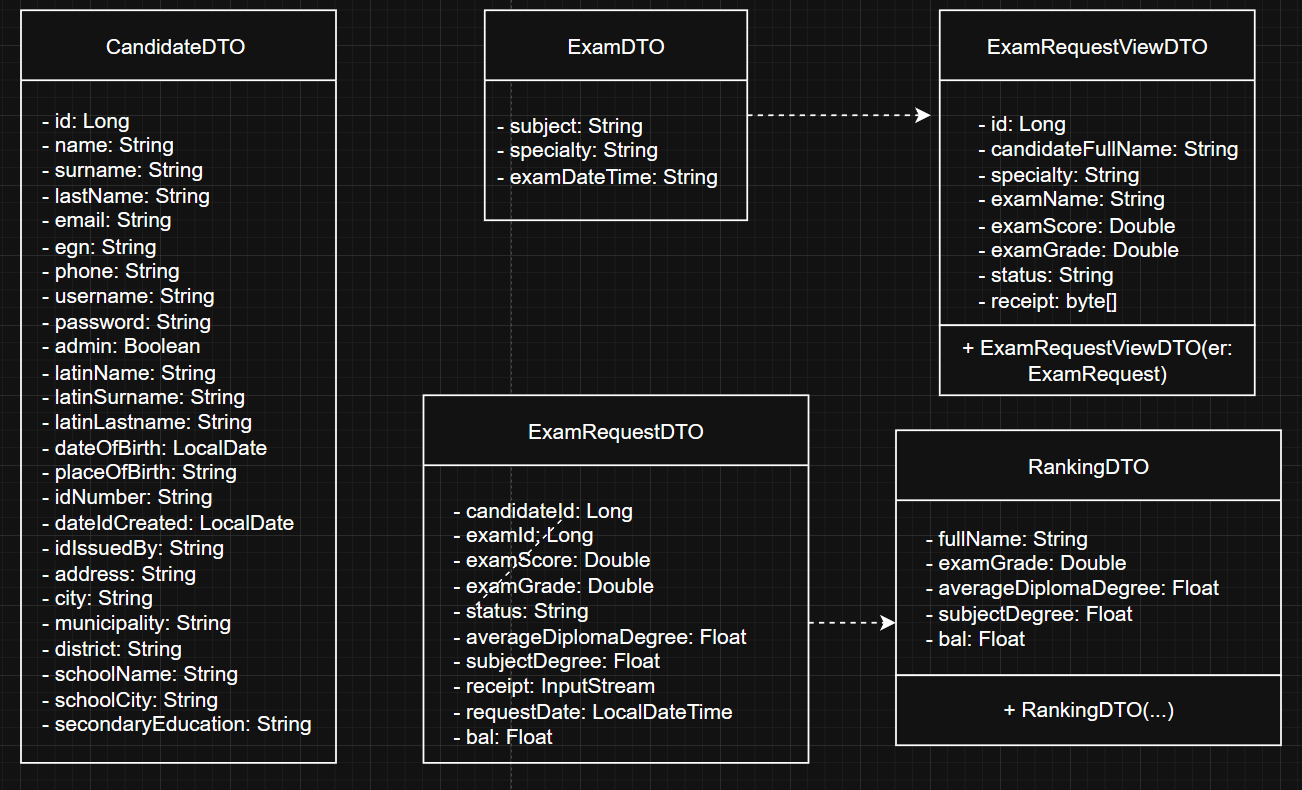
В рамките на бизнес логиката присъстват и **помощни класове (utils)**. Те не са част от конкретна бизнес операция, но изпълняват общи задачи, като например генериране на случайни пароли (PasswordUtil), проверка на съвпадение между пароли (bcrypt), или създаване на уникални потребителски имена (UsernameUtil), така че кандидатите да получават автоматично уникални идентификатори при регистрация.

Накрая, сигурността е осигурена чрез използване на **JWT (JSON Web Tokens)** за удостоверяване на самоличността и контрол на достъпа. Всеки потребител, който се логне успешно, получава токен, който използва за достъп до защитени ресурси. Администраторите имат разширени права и достъп до модули за управление на изпити, заявки, справки и класирания.

Системата е стабилна, добре модулно структурирана, и отговаря на добрите практики в съвременното уеб разработване, като осигурява ясно разделение на отговорностите, висока сигурност и лесна поддръжка.

## **3.4 UML Диаграми**

### **3.4.1 Class диаграма**

Фигура 9

В основата на този модел стои кандидатът, представен чрез обекта CandidateDTO. Този обект съдържа цялата основна информация за един кандидат, като например неговия уникален идентификатор, име, фамилия и презиме. Освен това има данни за електронната поща, личния номер (EGN), телефонен номер и потребителско име с парола, което предполага, че кандидатът има регистрация в системата. Интересно е, че има и булевата променлива „admin“, която вероятно показва дали кандидатът има административни права в системата. Кандидатът също така е описан с латински версии на имената си, което вероятно е необходимо за международна употреба или за по-лесно съвместимост със системи, които не поддържат кирилица. Налични са и лични данни като дата и място на раждане, както и различни документи като номер на лична карта, дата на издаване и от кой е издадена тя. Освен това, има информация за адреса, включително град, община и район, както и за образованието му – име на училището, град и степен на средното образование.

След това имаме ExamDTO, който описва конкретен изпит. Този обект включва предмета на изпита, специалността и датата и часа на провеждането му. По този начин системата знае какъв изпит предстои и кога трябва да се проведе.

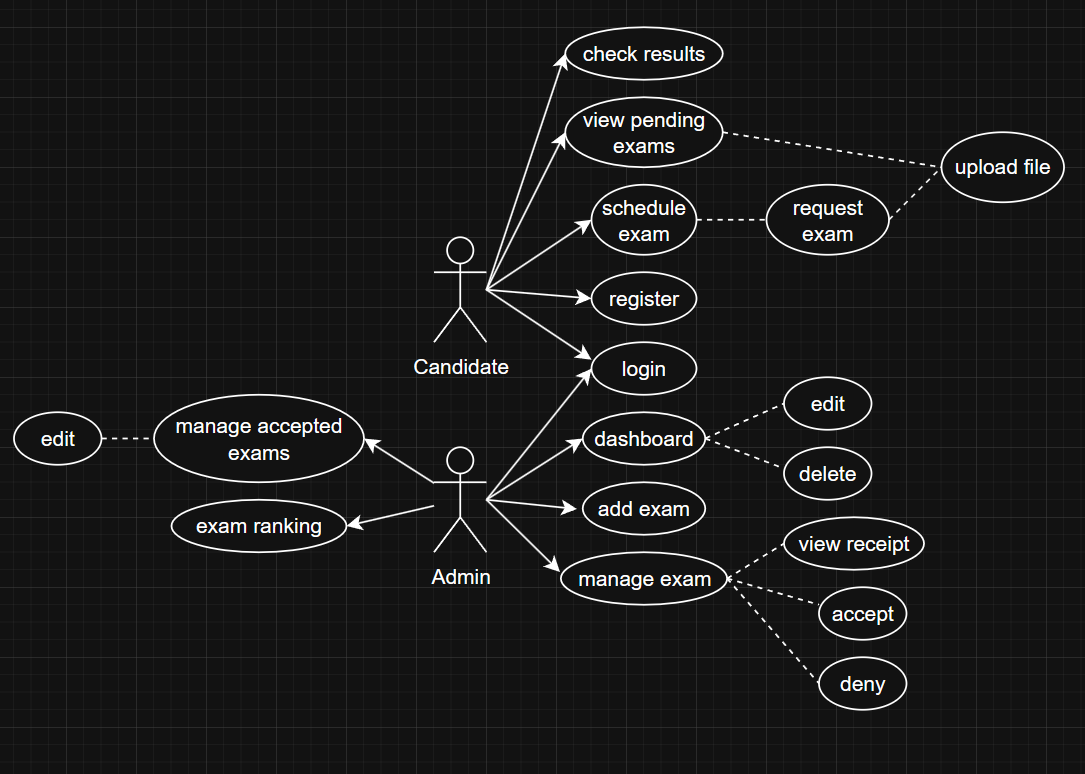
Свързан с изпитите е ExamRequestDTO. Този обект се използва, за да се запише кандидатът за даден изпит. Той съдържа идентификаторите на кандидата и изпита, резултатите от изпита, както оценка, така и точки. В допълнение, има статус, който показва дали молбата е приета, обработена или отхвърлена, и различни други оценки като средна дипломна степен и степен по предмет. Тук се съхранява и квитанцията за плащане във вид на поток от данни (InputStream), както и дата и час на подаване на заявката. Това дава възможност да се проследи цялата информация, свързана с процеса на кандидатстване и подаване на изпит.

ExamRequestViewDTO пък е обект, който се използва за визуализиране на заявките за изпити по по-удобен и обобщен начин. Той съдържа идентификатора на заявката, пълното име на кандидата, специалността, името на изпита, резултатите и статуса на молбата. Освен това включва и квитанцията, но вече под формата на масив от байтове, което улеснява показването ѝ в потребителския интерфейс.

Накрая имаме RankingDTO, който е свързан с класирането на кандидатите. Този обект обобщава информация за пълното име на кандидата, оценката от изпита, средната дипломна степен, степента по предмет и общия бал (бал). Това позволява да се подредят кандидатите по техните резултати и да се направи сравнителен анализ на представянето им.

Обобщено, тези обекти и техните полета изграждат една цялостна система за управление на кандидатстване и провеждане на изпити. Те позволяват събиране, съхраняване и визуализиране на важна информация, свързана с кандидатите, изпитите, молбите и класирането, което е от съществено значение за административната и логистична дейност в контекста на образование или подбор на кандидати. Ако искаш, мога да ти обясня и как се свързват тези обекти помежду си в процеса на работа.

### **3.4.2 Use case диаграма**

Фигура 10

Кандидатът има достъп до няколко основни функционалности в системата. Първото му действие е да се **регистрира (register)**, след което може да се **впише (login)** в системата. След успешен вход, кандидатът има достъп до **профилът си**, откъдето може да навигира към други функции.

Кандидатът може избере изпит (schedule exam) и да **заяви участие за изпит (request exam)**. И двете действия са свързани с възможността да **качва файл (upload file)** – например документ или платежна бележка, която служи за кандидатстване. Освен това, кандидатът може да **разглежда чакащи изпити (view pending exams)**, където може да обнови своят документ или платежна бележка ако е направил грешка.

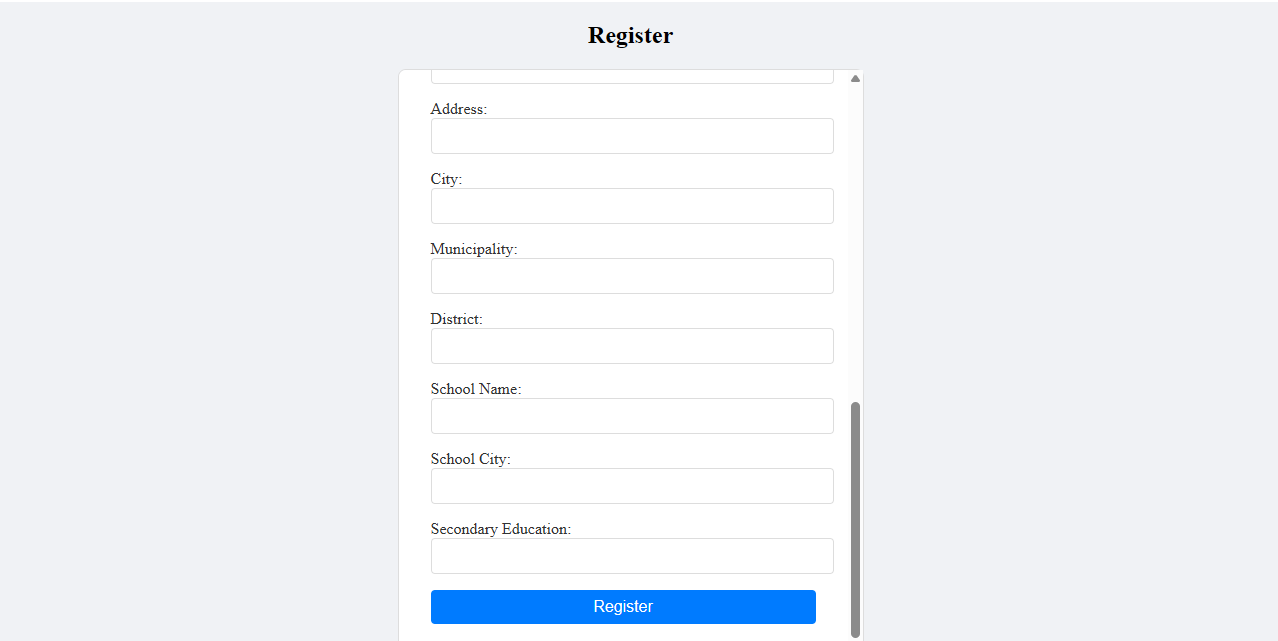
След провеждане на изпитите, кандидатът има достъп до функционалността **check results (проверка на резултати)**, чрез която вижда своите оценки или статус.

Администраторът също започва с **вход в системата (login)** и получава достъп до **таблото (dashboard)**. От таблото той може да изпълнява действия като **edit (редакция)** и **delete (изтриване) на изпитите**, което предполага управление на различни записи.

Администраторът има възможност да **добавя нов изпит (add exam)** и да **управлява изпити (manage exam)**. Управлението на изпитите включва действия като **преглед на получен документ или платежна форма (view receipt)**, **одобрение на кандидатура (accept)** или **отказ (deny)**.

След като кандидатите бъдат приети, администраторът може да използва функционалността **manage accepted exams (управление на приети изпити)**. Тя е свързана с възможността за **редактиране (edit)** на вече приетите записи като точки и оценки. Освен това, администраторът може да използва **exam ranking (класиране по изпити)**, което генерира списък с класирани кандидати по оценки и бал за съответната специалност и изпит.

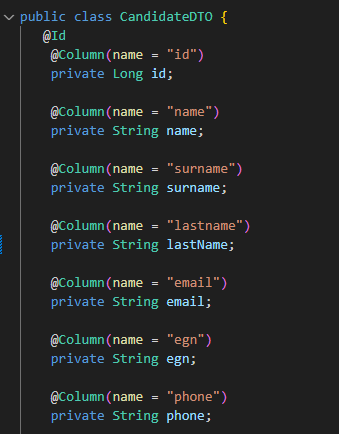
## **4. Реализация на системата**

Фигура 11

Когато кандидат иска да се регистрира в системата, той попълва регистрационна форма с лични данни – като име, имейл, ЕГН, телефон, адрес и т.н. След това натиска бутона **„Register“**, с което се изпраща заявка към системата. Тази заявка отива към специален интернет адрес, наречен **API** в случая това е /api/auth/registration.

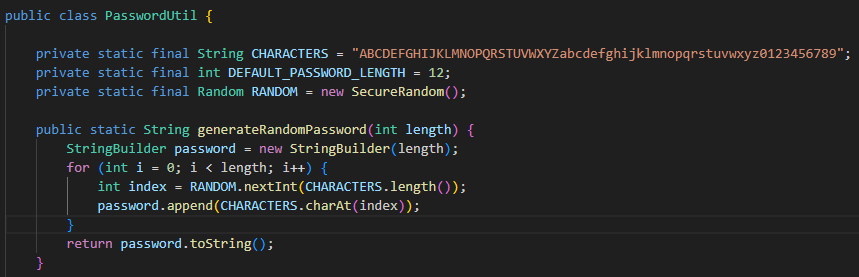
Тази част от системата се управлява от клас, наречен AuthenticationController, който отговаря за логина (вход в системата), регистрацията и профила на потребителя. При регистрацията, AuthenticationController получава информацията от формата и я предава на друг клас – RegistrationService, който се грижи за реалното създаване и запазване на нов кандидат в базата данни.

В RegistrationService първата стъпка е да се вземат въведените от потребителя данни и да се преобразуват във вътрешна структура, която се нарича Candidate. Това е обект, чрез който системата общува с базата данни. Тъй като данните, изпратени от потребителя, са в малко по-различен формат (CandidateDTO), те се преобразуват чрез специален метод, който копира полетата от едната структура в другата.



Фигура 12

След това системата създава произволна и сигурна парола за новия кандидат. Това се прави с помощта на класа PasswordUtil, който има метод, наречен generateRandomPassword(). Този метод генерира случайна комбинация от букви и цифри, с дължина 12 символа, като например Xy2Af93LpZ8d. Паролата, която се създава, обаче **не се записва директно** в базата данни. Вместо това, тя се **хешира**. Хеширането е процес, при който истинската парола се преобразува в дълъг, нечетим код, който не може да бъде обърнат обратно. Това се прави с цел сигурност – дори ако някой получи достъп до базата данни, той няма да знае истинската парола на потребителя. Хеширането се извършва чрез друг метод от PasswordUtil, наречен encodePassword().

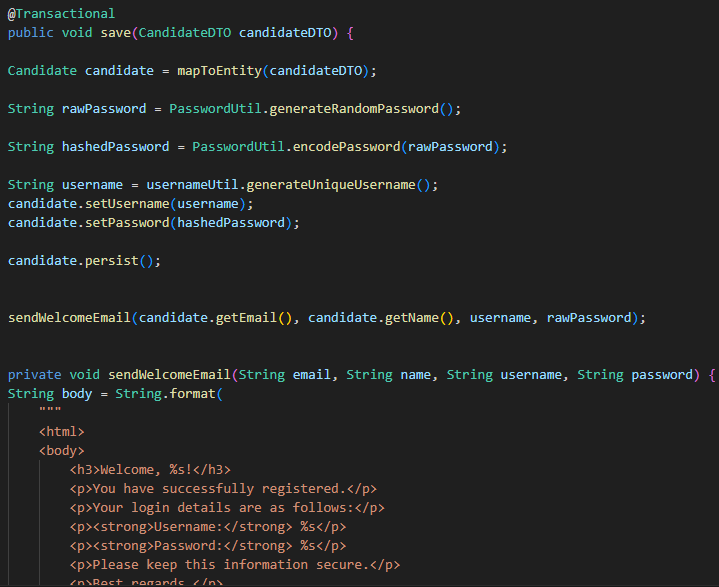
Фигура 13

След като имаме парола, системата трябва да създаде и **уникално потребителско име** за кандидата. Това се извършва от класа UsernameUtil, който генерира име, базирано на текущата година и случайни цифри. Например, ако годината е 2025, името може да бъде 251234. Класът проверява дали това име вече не се използва от друг кандидат и ако е нужно, пробва ново, докато намери свободно.

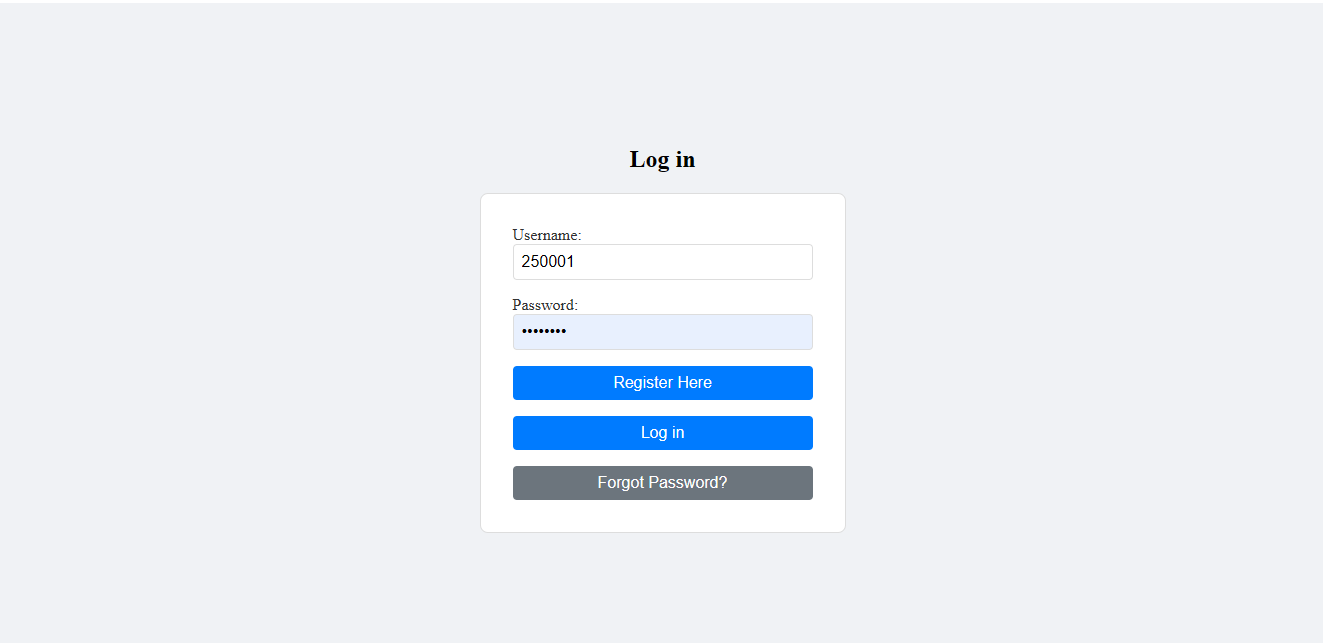
Когато са готови както потребителското име, така и паролата, те се записват към кандидата, заедно с останалите лични данни. След това системата запазва целия кандидат в базата данни. Това е моментът, в който потребителят вече официално съществува в системата.

След като записването приключи успешно, системата прави последна стъпка, тя изпраща **имейл** на кандидата. Имейлът съдържа приветствие, потребителското име и оригиналната (нехеширана) парола, за да може потребителят да влезе в системата. Имейлът е форматиран като HTML съобщение и се изпраща автоматично чрез обект, наречен mailer.

В този имейл се казва, например: „Добре дошъл, Иван! Успешно си се регистрирал. Данните ти за вход са: Потребителско име: 251234 и Парола: Xy2Af93LpZ8d. Моля, пази тази информация.“ Изпращането на имейла се извършва в отделен процес, така че потребителят не трябва да чака дълго време.

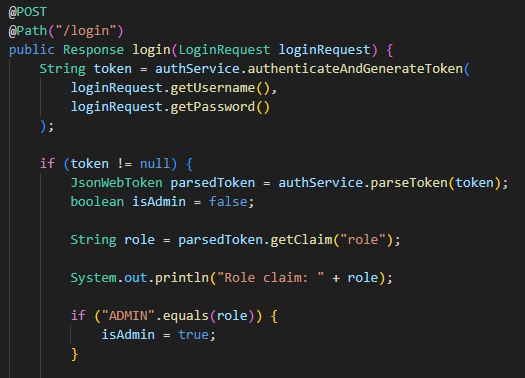
Фигура 14

След като потребителят получи имейла, той може да използва изпратеното потребителско име и парола, за да се логне в системата чрез друго място в AuthenticationController, наречено /api/auth/login. Там се проверява дали въведените данни са верни – използва се същият клас PasswordUtil, който сравнява въведената парола с хешираната, съхранена в базата. Ако всичко е наред, потребителят получава **токен за достъп** – това е цифров код, който му позволява да използва защитени части на системата. Освен това системата проверява дали потребителят има специални права – например дали е администратор.



Фигура 15

Когато един потребител иска да влезе в системата, той въвежда своето **потребителско име** и **парола** в екрана за вход (login). Тази информация се изпраща като заявка към системата, по-точно към /api/auth/login, която е управлявана от AuthenticationController. Оттам заявката се подава на специален помощен клас, наречен AuthenticationService, който извършва проверката на потребителските данни.

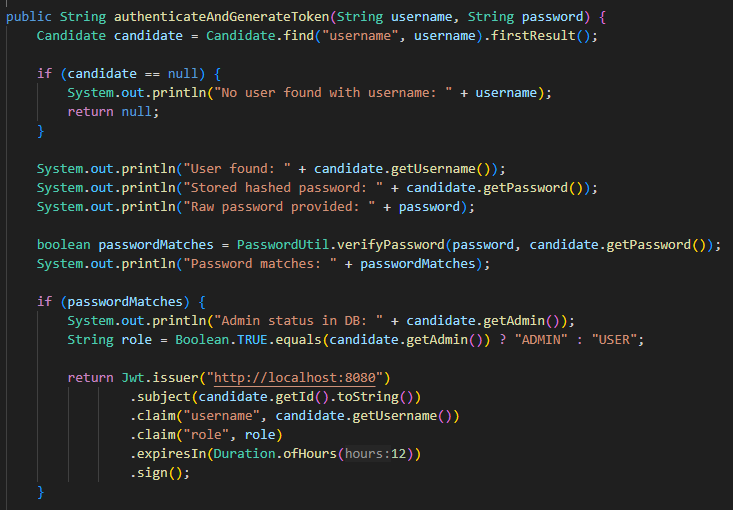


Фигура 16

В метода authenticateAndGenerateToken първата стъпка е системата да потърси в базата данни дали съществува кандидат с такова потребителско име. Ако такъв не бъде открит, т.е. ако потребителят е въвел грешно име, методът просто връща null и входът се счита за неуспешен.

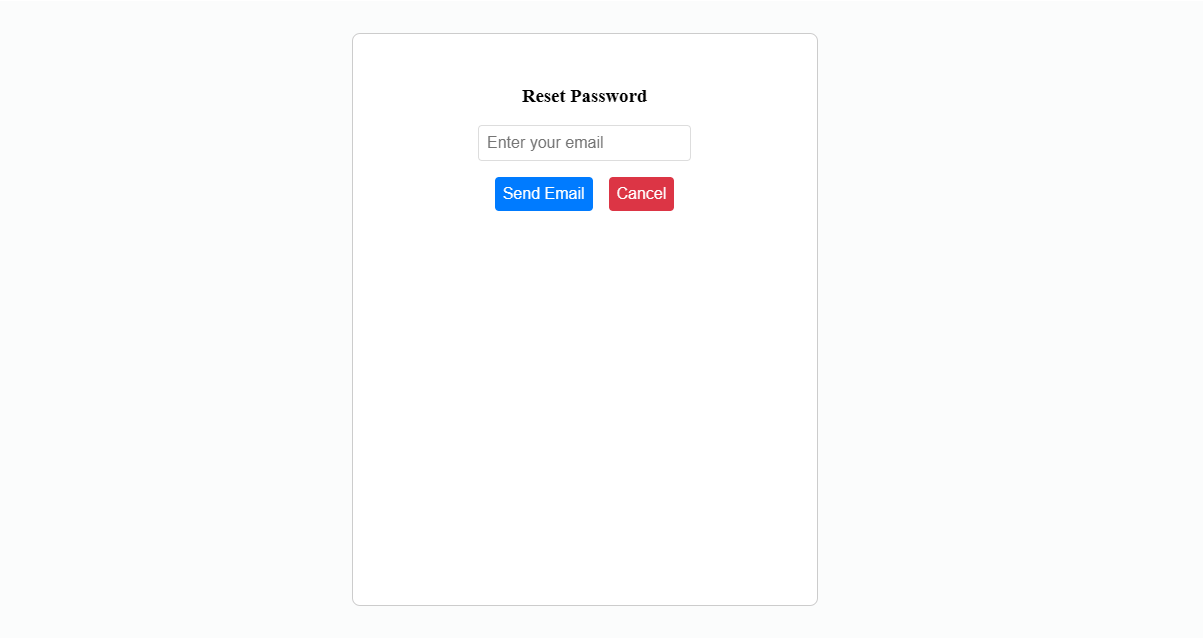
Ако кандидатът бъде намерен, системата започва да проверява дали въведената от него парола съвпада с тази, която е записана в базата. Важно е да се знае, че в базата не се пази истинската парола, а само нейната **хеширана версия**, т.е. защитен криптиран вариант. За да провери съвпадението, системата използва метод от PasswordUtil, който сравнява въведената (обикновена) парола с криптираната. Ако паролата съвпада, тогава проверката е успешна.

След успешна проверка системата трябва да създаде **токен за достъп**, който служи като „пропуск“ за потребителя. Този токен се нарича JWT (JSON Web Token) и съдържа в себе си важна информация – кой е потребителят, какво е неговото потребителско име, дали има администраторски права (роли), и докога е валиден токенът. Ако в базата данни кандидатът е маркиран като администратор, тогава в токена се добавя роля „ADMIN“, в противен случай – „USER“. Токенът има валидност 12 часа, и след това трябва да се създаде нов.

Фигура 17

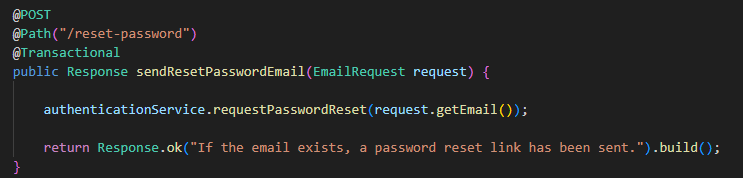
След като токенът бъде създаден, той се връща обратно на потребителя. От този момент нататък, когато потребителят използва токена, системата може да го идентифицира и да знае дали той е обикновен потребител или администратор. Този токен може да се използва за достъп до други части на системата, като например за извличане на профилна информация.

Ако обаче паролата, която е въведена от потребителя, **не съвпада** с тази в базата, тогава методът също връща null, с което входът се счита за неуспешен.

Фигура 18

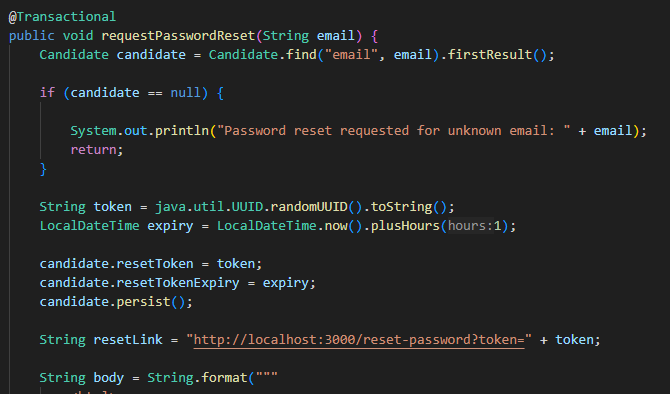
В AuthenticationService има и допълнителна функционалност – **забравена парола**. Ако потребителят е забравил паролата си, той може да въведе своя имейл, и чрез метода requestPasswordReset системата ще генерира **специален временен код (token)**. Този код представлява дълга уникална комбинация от символи, която е валидна за определено време – например 1 час. Той се записва в базата данни към съответния кандидат, заедно с информация кога изтича валидността му.

След това системата създава специален линк, съдържащ този временен код, например http://localhost:3000/reset-password?token=.... Този линк се изпраща чрез имейл на потребителя с текст, който го приканва да натисне линка, за да промени паролата си. Ако по някаква причина в базата не бъде намерен потребител с такъв имейл, системата не казва това директно, за да не разкрива дали даден имейл е регистриран – това е мярка за сигурност срещу хакери.

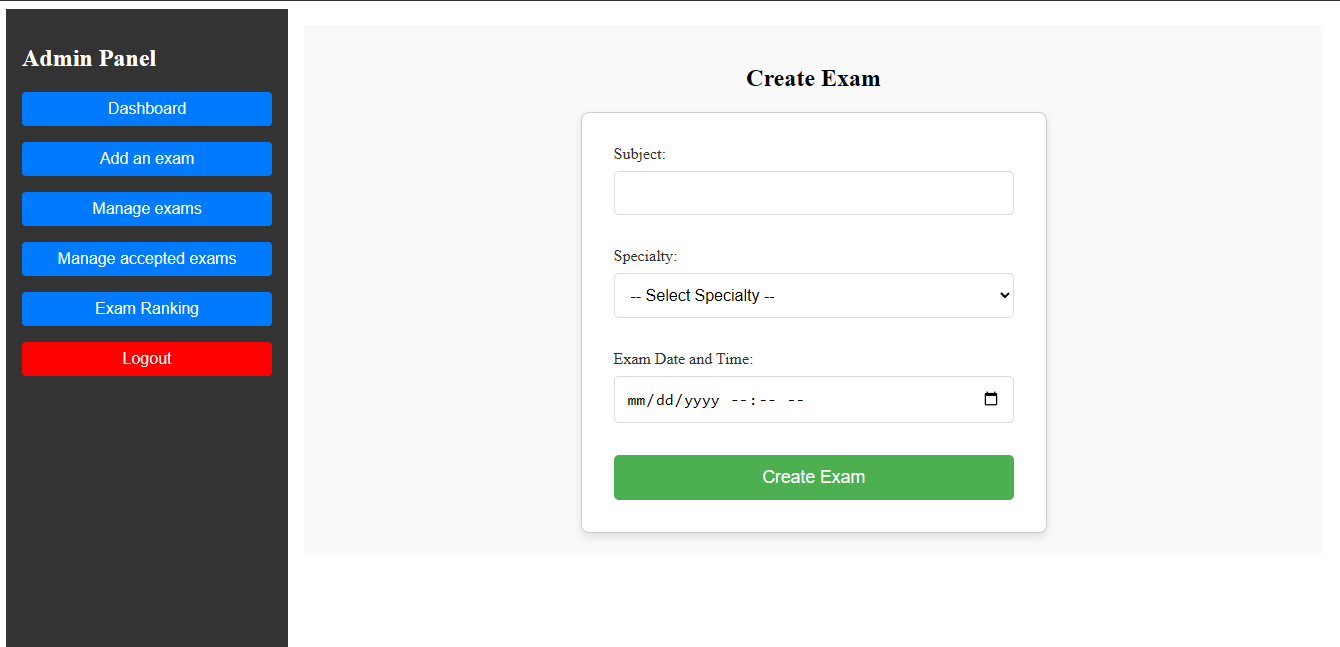
Фигура 19

Фигура 20

Когато потребителят натисне линка и въведе новата си парола, методът resetPassword проверява дали кодът (token) все още е валиден. Ако токенът е правилен и не е изтекъл, тогава новата парола се **хешира** и се записва в базата, като същевременно временният токен и времето му на валидност се изтриват, за да не може линкът да бъде използван повторно.



Фигура 21

Фигура 22

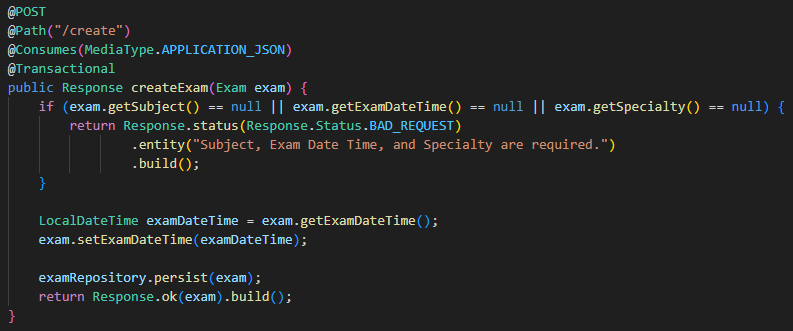
Класът AdminExamController представлява част от **административния интерфейс** на една система за управление на кандидатстване и изпити. Той отговаря за действия, свързани с **изпитите** – например създаване, редактиране или изтриване на изпит. В конкретния случай разглеждаме метода createExam, който има за цел да **създаде нов изпит в системата**.

Всичко започва, когато администратор (човек с права да управлява системата) натисне бутон в административния панел, чрез който подава информация за нов изпит – какъв е предметът (напр. „Математика“), кога ще се проведе изпитът (конкретна дата и час), и за коя специалност е предназначен (напр. „Софтуерни и Интернет Технологии“). Тези данни се изпращат в системата чрез специален **API адрес** – /api/exam/create.

Този адрес е дефиниран с помощта на анотация @Path("/create"), което означава, че ако някой изпрати POST заявка до адрес /api/exam/create, ще бъде извикан точно този метод. Заявката трябва да съдържа информация за изпита в **JSON формат**, което е стандартен начин за изпращане на структурирани данни в интернет. Това е указано чрез @Consumes(MediaType.APPLICATION\_JSON) – казваме на системата, че очакваме входът да бъде в JSON.

Когато заявката пристигне, методът createExam я приема, и първото нещо, което прави, е **проверка дали подадената информация е пълна**. В този контекст се проверява дали са попълнени задължителните полета – предметът на изпита (subject), датата и часът на провеждането (examDateTime), както и специалността, за която се отнася (specialty). Това се прави чрез условие, което казва: ако някое от тези три полета е празно (т.е. не е въведено), тогава заявката се смята за невалидна и системата връща съобщение за грешка. Това съобщение е стандартен отговор със статус код 400 BAD REQUEST, което означава, че клиентът (в случая администраторът) е направил грешка, като не е попълнил нещо задължително.

Ако обаче всички нужни полета са попълнени, тогава се преминава към следващата стъпка – запазване на изпита в базата данни. За целта се използва специален обект, наречен ExamRepository, който отговаря за взаимодействието между приложението и базата данни по отношение на обекти от тип „Exam“ (Изпит).

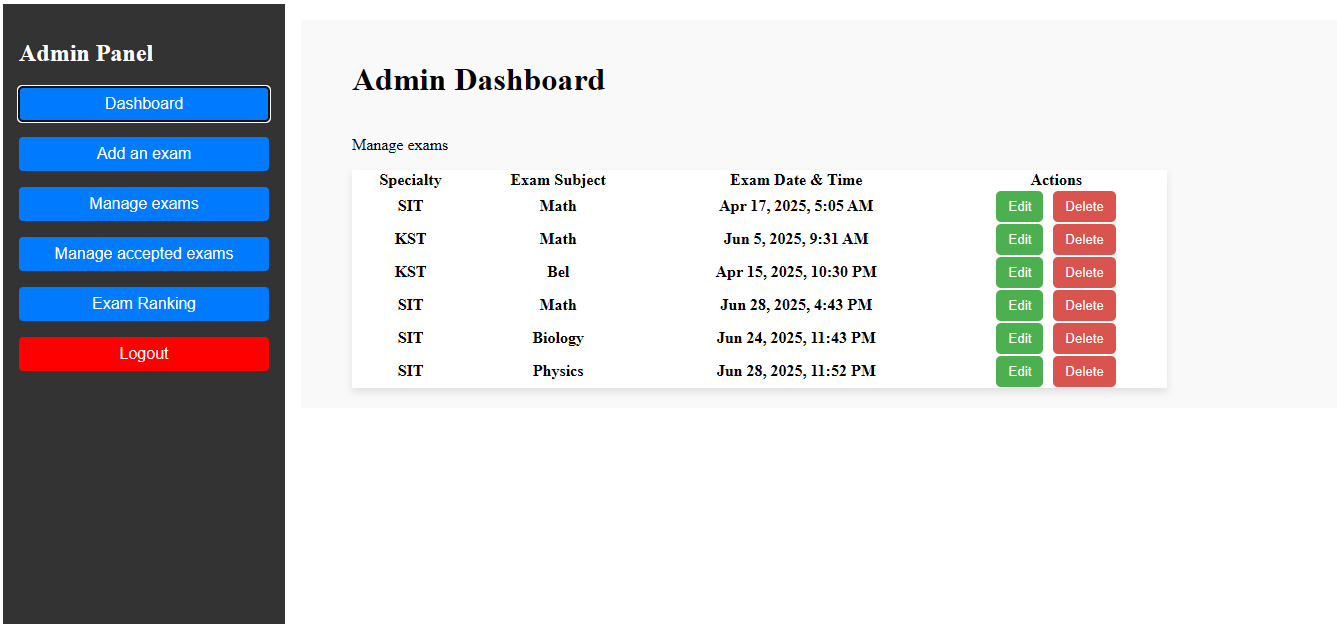
Фигура 23

Преди да се извърши записът, кодът създава обект examDateTime, който просто взема стойността на подадената дата и час от входните данни. Това не е строго необходимо, но е добър начин да се подготви стойността преди запис. След това методът извиква examRepository.persist(exam), което означава „запази този изпит в базата данни“.

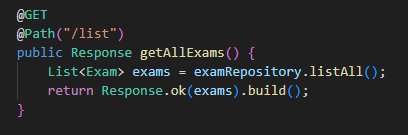
Така информацията за новия изпит – включително предметът, часът, датата и специалността – се съхранява и остава достъпна за бъдещи операции, като например показване на изпитите на кандидатите, редактиране, изтриване и т.н.

След успешното записване на изпита в базата, системата връща обратно потвърждение към администратора, че операцията е успешна. Това се прави с Response.ok(exam).build();, което означава: „операцията беше успешна и тук е информацията за новия изпит, която току-що запазихме“. Тази информация може да се използва, за да се покаже визуално потвърждение на екрана или за други автоматични действия в административния интерфейс.

Важно е да отбележим и още нещо – методът е маркиран с @Transactional, което означава, че цялата операция по създаване на изпита се извършва в така наречената **транзакция**. Това е нещо като гаранция, че ако нещо се обърка в процеса (например се загуби връзка с базата), записът няма да се направи частично. Или ще бъде изцяло успешен, или изцяло ще бъде върнат назад – така се гарантира **надеждност и консистентност** на данните.

Фигура 24

Методът getAllExams, който се съдържа в този контролер, е предназначен за извличане на **пълен списък с всички изпити** в системата. Това се осъществява чрез HTTP GET заявка към /api/exam/list. Методът използва examRepository.listAll(), за да извлече всички записи от таблицата с изпити в базата данни и ги връща под формата на списък. Отговорът от метода е от тип Response.ok(exams).build();, което означава, че резултатът ще бъде подаден на клиента в стандартен JSON формат. Тази функционалност е изключително полезна както за интерфейси, които визуализират изпити в таблична форма, така и за програми или скриптове, които се нуждаят от обобщени данни за анализ или отчет. Методът не съдържа параметри за филтриране или сортиране, но такива могат лесно да бъдат добавени в бъдещи разширения на API-я.



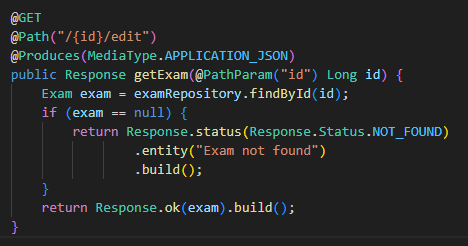
Фигура 25



Фигура 26

Mетодът GetExam, който се намира в контролера, е предназначен за преглед

на конкретен изпит по неговото уникално идентификационно число (ID). Това се реализира чрез HTTP GET заявка към адрес /api/exam/{id}/edit, където параметърът {id} представлява уникалният идентификатор на изпита в базата данни. При получаване на такава заявка, методът използва examRepository.findById(id), за да опита да извлече съответния обект от базата. Ако такъв обект не бъде намерен, методът отговаря със статус код 404 Not Found и съобщение „Exam not found“, което ясно информира клиента, че търсеният ресурс не съществува. В случай че изпитът бъде успешно намерен, той се връща на клиента в JSON формат, с което се дава възможност на потребителския интерфейс или друг клиент на API да визуализира информацията за изпита и при необходимост да извърши последващи действия, като редакция или анализ.

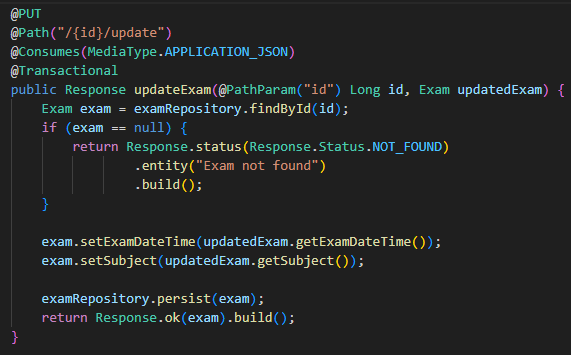


Фигура 27

Методът updateExam се използва за редактиране или актуализиране на съществуващ изпит. Това е POST или PUT операция, в случая реализирана с PUT заявка към пътя /api/exam/{id}/update. Методът приема като аргумент обект от тип Exam, който съдържа новите стойности за редактиране. В рамките на метода първо се извършва проверка дали изпитът с даденото ID съществува, като отново се използва examRepository.findById(id).

Aко изпитът не бъде открит, се връща същият отговор като в предходния метод – 404 Not Found, с което се избягва възможността да се редактира несъществуващ обект. Ако изпитът бъде открит, се продължава с актуализиране на неговите свойства, като се презаписват новите стойности за examDateTime и subject, подадени в тялото на заявката.

След като стойностите са зададени, с помощта на examRepository.persist(exam) се извършва запазване на обекта в базата данни в рамките на активна транзакция, маркирана с анотацията @Transactional, което гарантира целостта и консистентността на операцията. В отговор се връща обновеният изпит в JSON формат, така че клиентът да може да потвърди успешно извършената промяна и да види как изглеждат данните след редакцията.

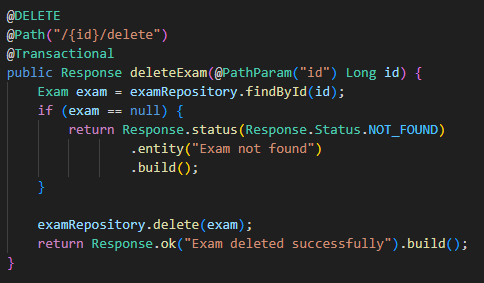


Фигура 28

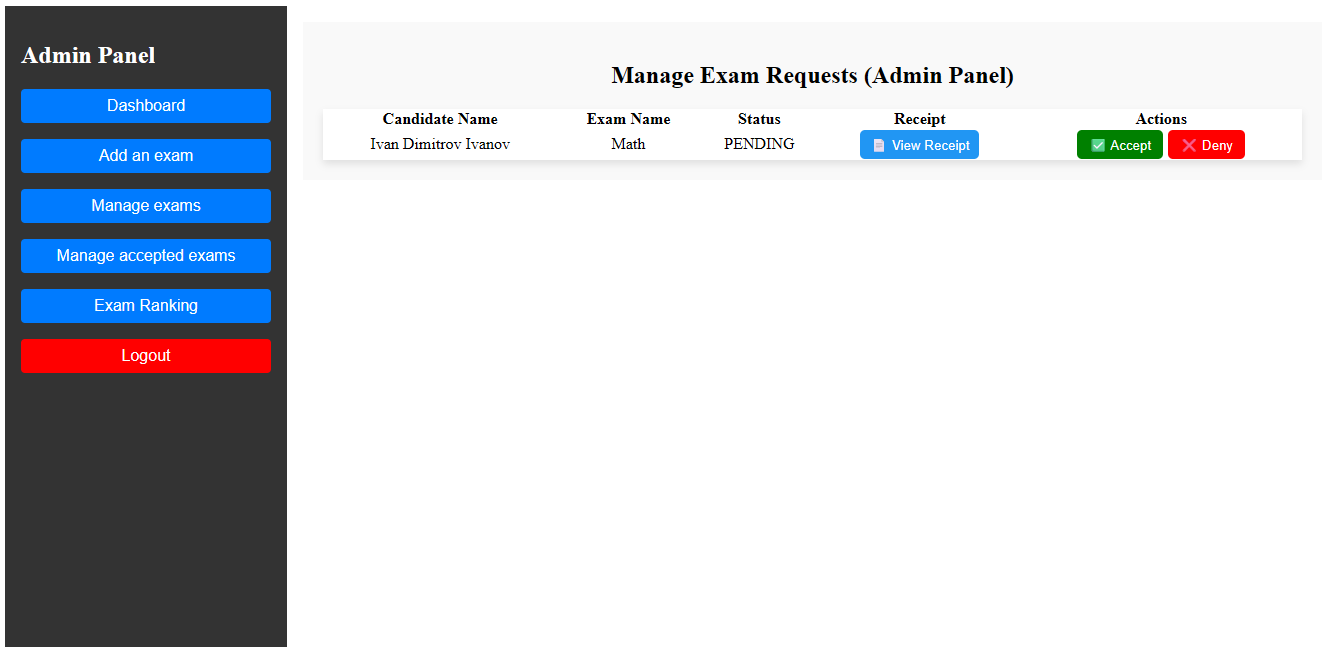
Методът deleteExam в системата е реализирано чрез DELETE заявка към /api/exam/{id}/delete. Методът за изтриване също започва с проверка за съществуването на изпита в базата, използвайки същата логика чрез examRepository.findById(id).

При липса на съответен обект в базата се връща статус 404, което осигурява надеждност и яснота в комуникацията с клиентските приложения.

Ако обектът бъде намерен, той се предава на метода examRepository.delete(exam), който физически го премахва от базата данни. Това премахване е твърдо изтриване, при което записът изчезва напълно. След успешното изтриване се връща текстов отговор със съобщение „Exam deleted successfully“ и статус код 200, което потвърждава, че операцията е изпълнена успешно. Това е особено важно при административни интерфейси, където ясната обратна връзка към потребителя е критично важна.



Фигура 29

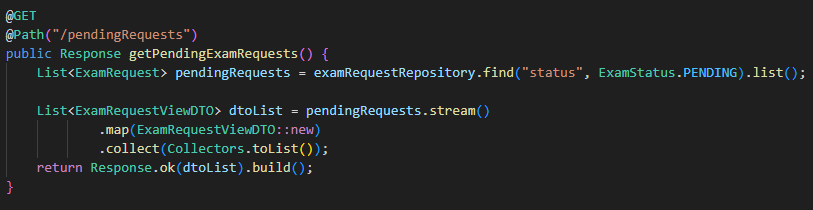


Фигура 30

Методът getPendingExamRequests, се използва за извличане на списък от всички чакащи потвърждение заявки за участие в изпит. Това е HTTP GET заявка към пътя /api/exam/pendingRequests, която не изисква параметри, тъй като нейното предназначение е да върне всички заявки, чийто статус е отбелязан като PENDING, тоест очакват одобрение от администратор.

Вътрешно методът използва examRequestRepository.find("status", ExamStatus.PENDING).list();, което означава, че извлича от базата данни всички записи, в които колоната status има стойност PENDING. След като списъкът със заявки бъде извлечен, той се трансформира към по-удобен за визуализация формат чрез мапване на всеки обект към ExamRequestViewDTO.

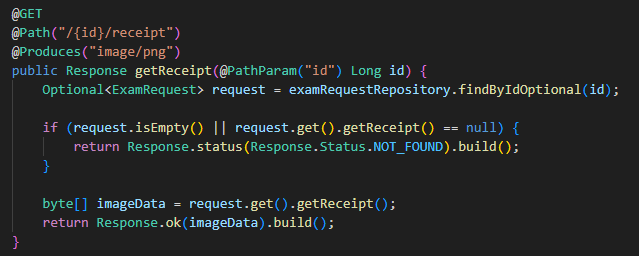
Това се извършва с помощта на Java Stream API, като за всяка заявка се създава нов DTO обект, предназначен да съдържа само релевантната информация, която клиентът трябва да види. Резултатът от тази операция е списък от DTO обекти, който се връща като JSON отговор с HTTP статус 200, което позволява на интерфейса на администратора да представи лесно преглед на чакащите заявки в табличен или списъчен изглед.



Фигура 31

Фигура 32

Следващият метод, наречен getReceipt, е отговорен за връщането на изображение, което представлява прикачената разписка за платена такса, подадена от кандидата като част от заявката за изпит. Това е GET заявка с път /api/exam/{id}/receipt, където {id} е уникалният идентификатор на заявката. Методът използва examRequestRepository.findByIdOptional(id);, за да намери заявката по дадения ID. Ако заявка с този идентификатор не съществува или ако полето receipt е празно (тоест не е предоставена разписка), тогава се връща отговор със статус 404 Not Found, като по този начин се избягва погрешна обработка на заявката. Ако обаче разписката е налична, методът връща нейното съдържание като масив от байтове с MIME тип image/png, което означава, че клиентът може директно да визуализира изображението в браузър или друго приложение, което поддържа показване на PNG изображения. Това е особено полезно за администраторите, тъй като им позволява да проверят валидността на плащането, преди да вземат решение дали да одобрят или отхвърлят съответната заявка.

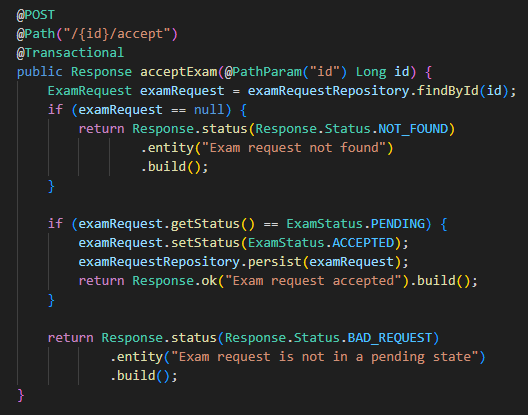


Фигура 33



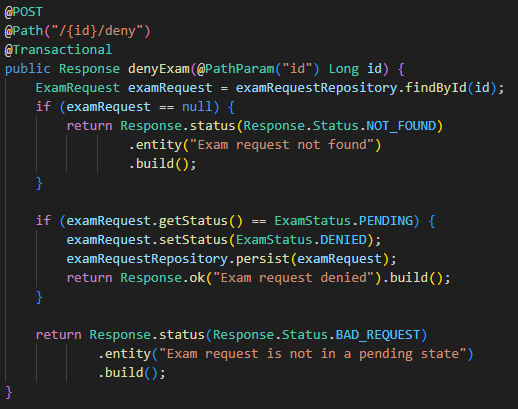
Фигура 34

Методът acceptExam в този контролер е предназначен за **одобряване на заявка за изпит**. Това става чрез HTTP POST заявка към /api/exam/{id}/accept, където отново се използва уникалният идентификатор на заявката. Методът е анотиран с @Transactional, което означава, че всички промени, извършени в рамките на този метод, ще бъдат извършени в една транзакция и няма да останат в базата данни, ако възникне грешка по време на изпълнение. Вътре в метода първо се извършва проверка дали съществува заявка с дадения ID. Ако такава не бъде намерена, методът връща 404 Not Found, като по този начин се избягва неволна модификация на невалиден запис. Ако заявката е валидна и нейният статус е PENDING, то тя се приема за прегледана и се отбелязва като ACCEPTED. След тази промяна обектът се запазва чрез examRequestRepository.persist(examRequest);, и клиентът получава отговор със съобщение „Exam request accepted“, което потвърждава, че системата е обработила правилно промяната. Ако обаче заявката вече не е в състояние PENDING, например ако вече е одобрена или отказана, тогава методът връща статус 400 Bad Request със съобщение, което ясно показва, че тази заявка вече не подлежи на обработка.

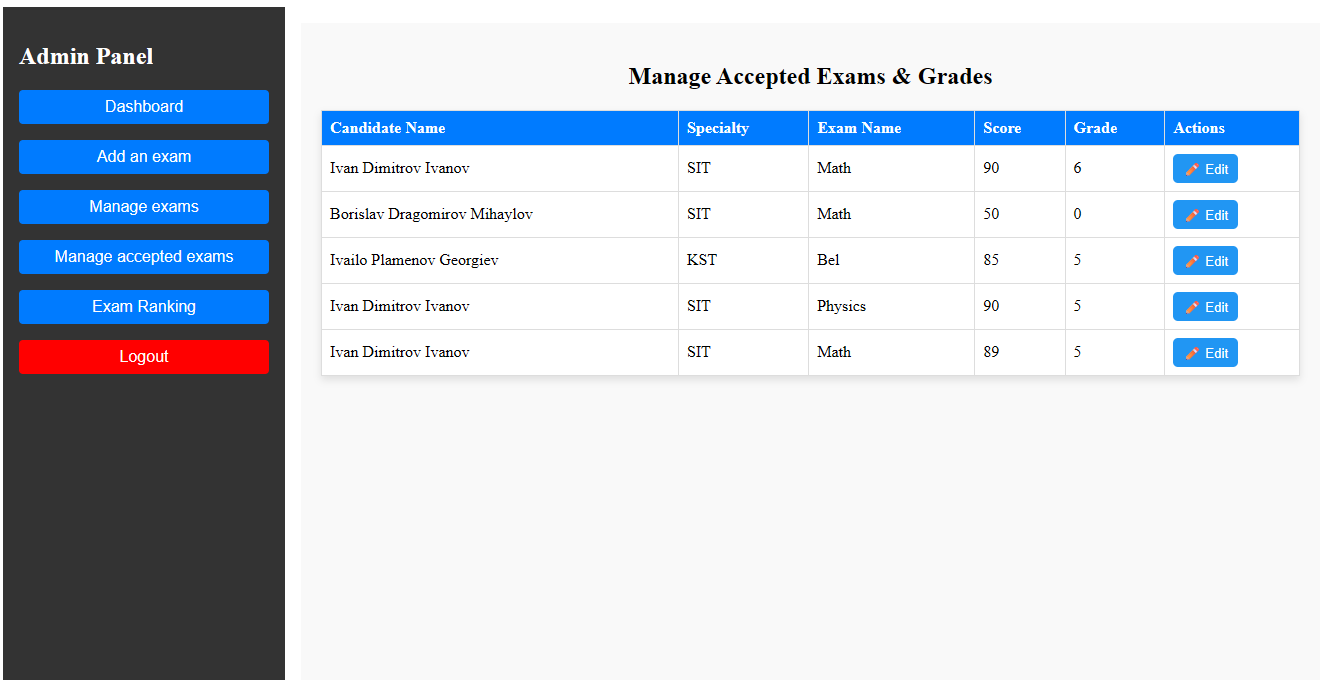


Фигура 35

Методът denyExam изпълнява противоположната операция, а именно – **отхвърляне на заявка**. Той също използва POST заявка, като адресът е /api/exam/{id}/deny, и също е анотиран с @Transactional. Логиката му следва същата структура като при приемането на заявка – първо се търси обектът по ID, и ако не бъде намерен, се връща 404 Not Found. Ако обектът съществува и неговият статус е PENDING, тогава той се маркира със статус DENIED, след което се извършва запис в базата. Клиентът получава потвърждение под формата на текст „Exam request denied“. Ако обаче заявката вече не е в изчакващ статус, отново се връща 400 Bad Request, тъй като не е допустимо да се отхвърля нещо, което вече е било прието или отказано. Този контролен механизъм предотвратява грешки и гарантира целостта на данните.



Фигура 36

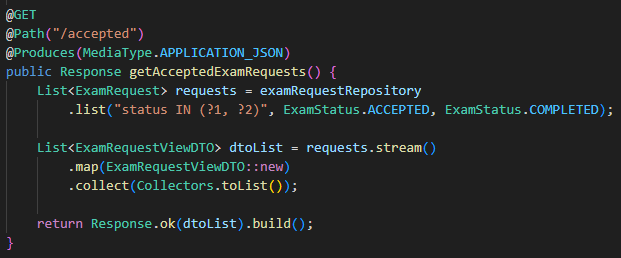
Фигура 37

Контролерът ManageAcceptedExamsController е съществен компонент от административния модул на системата, който се използва основно за управление на заявките за изпити, които вече са били **одобрени** от администраторите или вече са **завършени**. В контекста на процеса на кандидатстване за изпити, веднъж след като дадена заявка бъде одобрена, тя следва да премине през няколко допълнителни етапа – провеждане на изпита, оценяване, изчисление на бал и определяне на окончателния резултат.

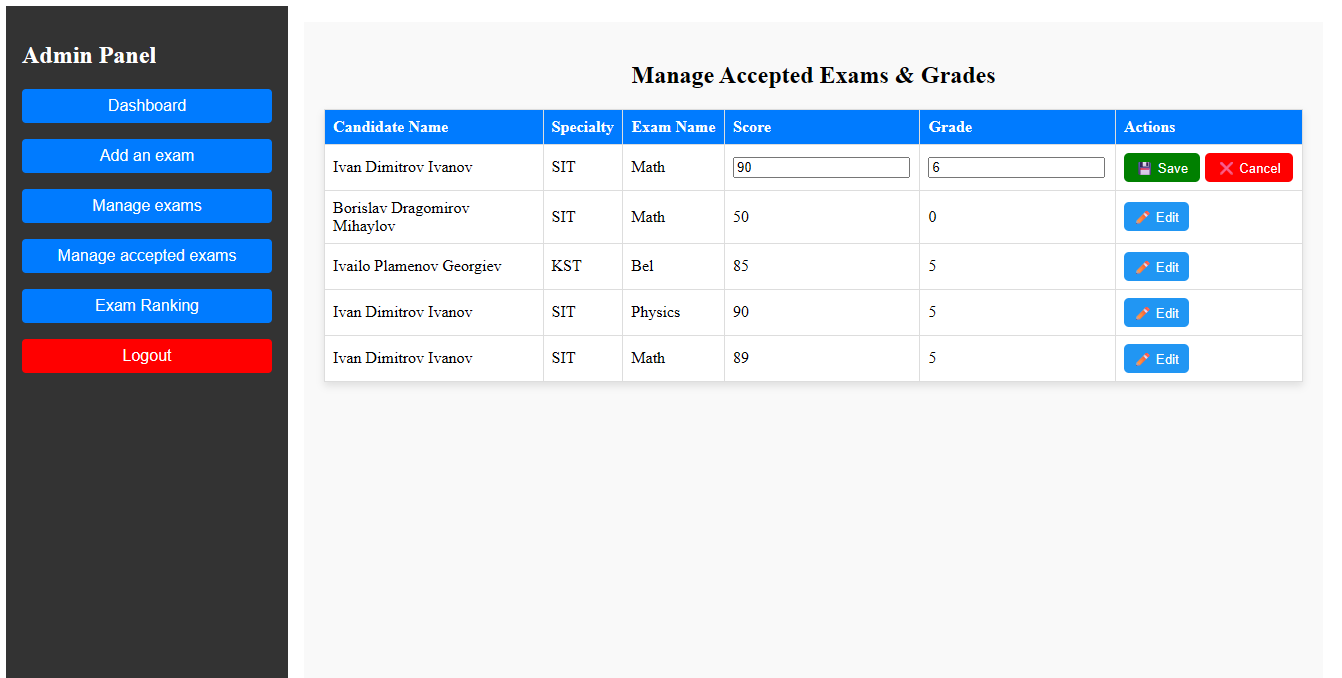
Всички тези задачи се изпълняват чрез методите, съдържащи се в този контролер, които предоставят интерфейс за извличане на приети заявки и обновяване на техния статус и резултати.

Методът getAcceptedExamRequests, дефиниран с анотацията @GET и достъпен чрез пътя /api/exam/manage/accepted, е отговорен за връщане на списък с всички заявки за изпити, които вече са били **приети** или **приключени**.

Това се реализира чрез заявка към базата данни, в която се използва условие status IN (?1, ?2) с параметри ExamStatus.ACCEPTED и ExamStatus.COMPLETED, което позволява едновременно филтриране на няколко възможни статуса. Важно е да се отбележи, че това не просто връща обектите от базата, а ги преобразува в по-опростена и подходяща за визуализация форма – чрез клас ExamRequestViewDTO. Това трансформиране осигурява по-чиста, по-малка и по-разбираема структура на данните, което улеснява клиентските приложения или потребителския интерфейс да обработват и визуализират необходимата информация. Списъкът от DTO обекти след това се изпраща като JSON отговор, който клиентът може директно да използва, например за показване на табличен изглед на всички текущо приети кандидати за изпити.

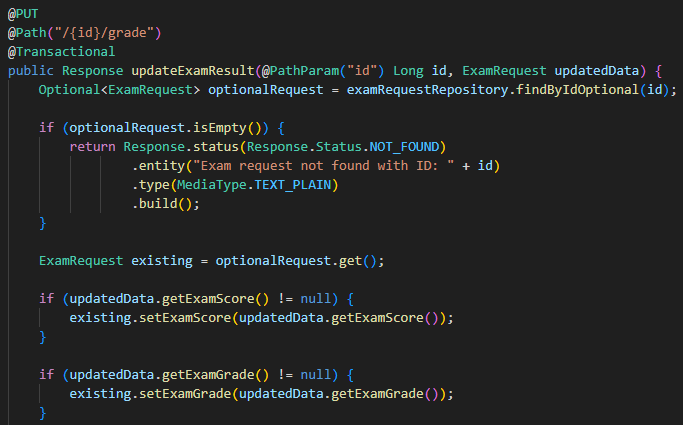


Фигура 38

Фигура 39

Методът updateExamResult значително по-сложен и критичен от гледна точка на функционалност, тъй като се използва за **актуализиране на оценките и крайното състояние на заявките**, след като изпитът е бил проведен. Този метод приема PUT заявка към пътя /api/exam/manage/{id}/grade, където {id} е идентификаторът на конкретната заявка за изпит, чиято информация следва да бъде обновена. Методът е анотиран с @Transactional, което гарантира, че всички операции, извършени в рамките на метода, ще бъдат изпълнени в рамките на една транзакция и няма да бъдат частично записани в случай на грешка. Основната му задача е да позволи на администраторите или преподавателите да въведат и актуализират резултатите от изпита, дипломната средна оценка, оценката по предмета, както и окончателния бал, който е обобщена стойност, използвана при класиране.

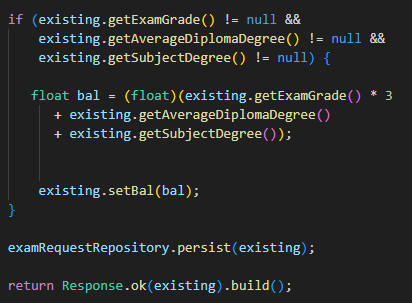
Когато заявката достигне до метода, първата стъпка е да се провери дали в базата съществува заявка с подадения идентификатор. Това се извършва чрез метода findByIdOptional(id), който връща Optional обект – по този начин се избягват потенциални NullPointerException. Ако такава заявка не бъде намерена, методът незабавно връща отговор със статус 404 Not Found, като по този начин уведомява клиента, че не е открит запис с такъв идентификатор. Ако обаче заявката съществува, тогава методът преминава към проверка и актуализация на предоставените данни.

Фигура 40

Конкретно, методът проверява дали обектът updatedData, който се получава в тялото на заявката, съдържа стойности за едно или повече от следните полета: examScore, examGrade, status, averageDiplomaDegree, subjectDegree. Ако някое от тях не е null, то съответното поле в оригиналния обект existing се обновява със стойността от входа. Това позволява частична актуализация на данните – например, може да се промени само оценката от изпита, без да се докосват дипломната оценка или общият бал. Това е важна характеристика на API=я, тъй като позволява гъвкавост при работа с различни форми на вход.

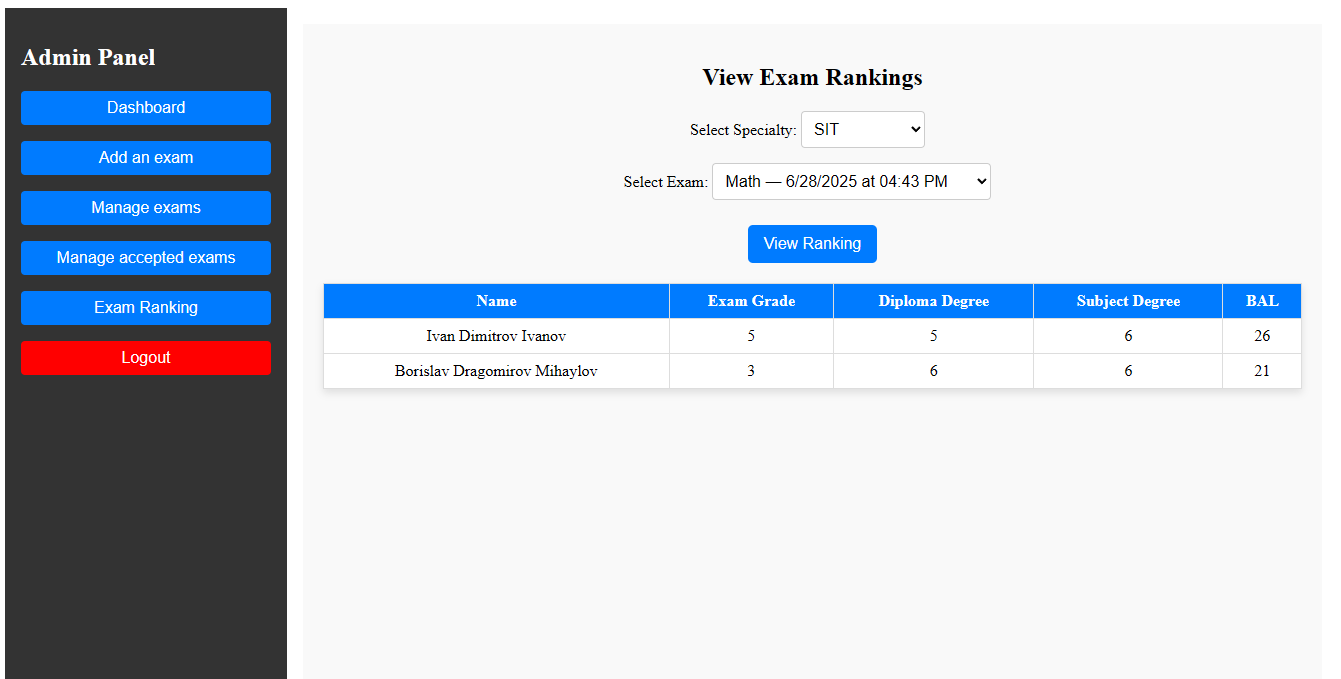
След това, когато всички необходими полета са попълнени – т.е. налични са examGrade, averageDiplomaDegree и subjectDegree, системата автоматично изчислява **бала на кандидата**. Изчислението използва следната формула: балът се изчислява като (оценка от изпита, умножена по 3) плюс средната дипломна оценка плюс оценката по конкретния предмет. Формулата е реализирана като:  
bal = examGrade \* 3 + averageDiplomaDegree + subjectDegree.

Този резултат се съхранява в полето bal на обекта ExamRequest, което го прави достъпен както за класиране, така и за по-нататъшна обработка и визуализация. Накрая заявката се запазва чрез persist, с което всички направени промени се прилагат в базата данни и се връща отговор със статус 200 и самия обновен обект като JSON.



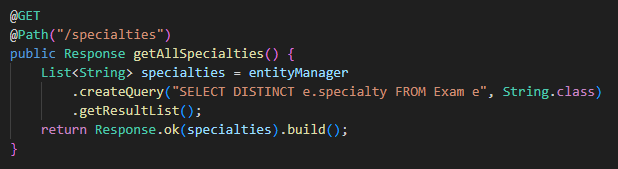
Фигура 41

Този метод е от ключово значение за крайния етап от процеса на изпитване, тъй като обединява в себе си няколко критични операции – актуализация на резултатите, изчисляване на крайния бал, както и промяна на състоянието на заявката. Той е гъвкав, сигурен и съобразен със стандартите за изграждане на RESTful API, като позволява точна и структурирана манипулация на данните, необходими за правилното оценяване и класиране на кандидатите.



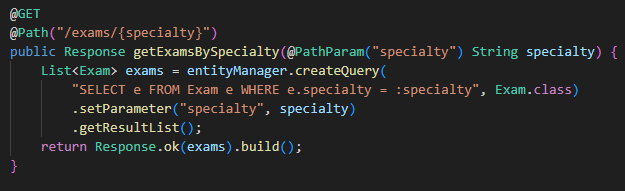
Фигура 42

Първоначално, когато администраторът отвори панела за преглед на класиране, интерфейсът изпраща заявка към ендпойнта /specialties, чрез която се извличат всички уникални специалности, по които са обявени изпити. Извличането става чрез EntityManager, който връща само уникалните наименования на специалностите, без дублиране. Тези специалности се използват за попълване на падащото меню „Select Specialty“, което позволява на администратора да избере конкретна специалност, като например „SIT“, както е показано на изображението.



Фигура 43

След като бъде направен избор от първото меню, се изпълнява нова заявка към ендпойнта /exams/{specialty}, чрез която се извличат всички изпити, асоциирани с избраната специалност. Тези изпити автоматично се зареждат във второто падащо меню – „Select Exam“. За всеки изпит в списъка се показва предметът, както и точната дата и час на провеждането му, което осигурява яснота и конкретика за потребителя. Това дава възможност на администратора да избере конкретен изпит, например „Math – 6/28/2025 at 04:43 PM“, както се вижда на примера.

Фигура 44

След като са избрани както специалността, така и съответният изпит, потребителят натиска бутона „View Ranking“, което задейства заявка към ендпойнта /ranking/{specialty}/{examId}. Този метод служи за извличане на резултатите от класирането, като първо филтрира всички обекти от тип ExamRequest. В селекцията се включват само онези, които отговарят едновременно на избраната специалност, посочения изпит и имат статус „COMPLETED“, тоест кандидатът е положил изпита и е получил окончателна оценка.

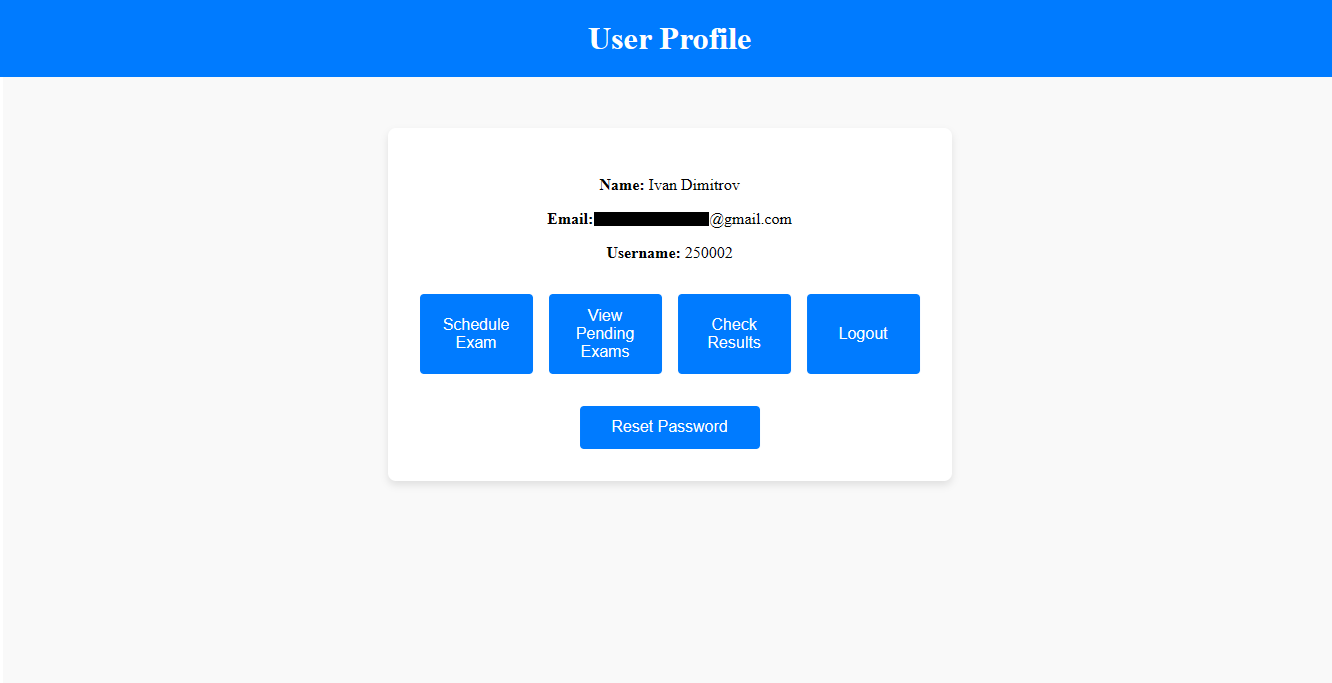
Фигура 45

За всеки кандидат, отговарящ на тези критерии, се изчислява балът по предварително зададена формула. Формулата включва следните компоненти: оценката от изпита се умножава по три, след което към получения резултат се прибавят дипломната оценка и оценката по профилиращия предмет. Така полученият сбор представлява крайният бал, на базата на който се формира класирането.

Всички кандидатури се сортират в низходящ ред спрямо този бал, така че кандидатите с най-високи резултати се показват най-отгоре в таблицата. Всеки запис съдържа имената на кандидата, оценката от изпита, дипломната оценка, оценката по предмет и крайния бал. Тези данни се капсулират в обекти от тип RankingDTO и се визуализират в табличен формат на клиентската страна.

В конкретния пример от изображението е избрана специалността „SIT“ и изпит по математика, проведен на 28 юни 2025 г. в 16:43 ч. В таблицата са показани резултатите на двама кандидати. Иван Димитров Иванов е получил оценка 5 от изпита, дипломна оценка 5 и оценка по предмет 6, което води до бал 26 (5 × 3 + 5 + 6). Вторият кандидат, Борислав Драгомиров Михайлов, има изпитна оценка 3, диплома 6 и оценка по предмет 6, което води до бал 21 (3 × 3 + 6 + 6). Таблицата ясно показва сравнението между кандидатите и подреждането им по резултати.

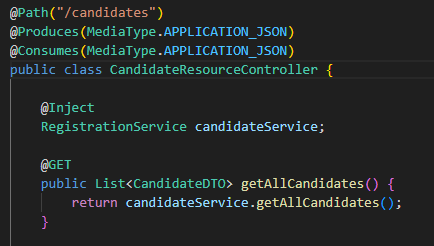
Цялата система е изградена така, че да предоставя бърз достъп до актуална и точна информация за резултатите, като елиминира нуждата от ръчно изчисляване. Това я прави изключително ефективна и надеждна в процеса на подбор и класиране на кандидат-студенти.

Фигура 46

Контролерът, отговарящ за управление на кандидатите, предоставя RESTful интерфейс, чрез който клиентската част на системата може динамично да зарежда информация за всички регистрирани кандидати, както и детайлите за конкретен кандидат по идентификатор.

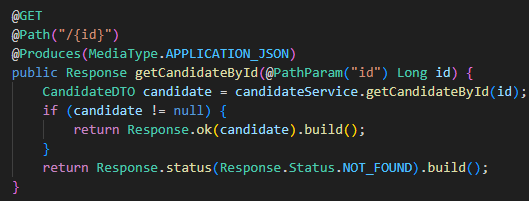
Тази функционалност е особено важна за приемната комисия или администрацията, тъй като осигурява бърз, структуриран и централизиран достъп до кандидатурите в системата.

При отваряне на административния интерфейс или секцията за преглед на кандидатите, се извършва заявка към ендпойнта /candidates. Този метод използва услугата RegistrationService, чрез която се извличат всички налични кандидати от базата данни. Получените данни се връщат като списък от обекти от тип CandidateDTO, който съдържа информация за всеки кандидат — като име, контакти, подадени документи или друга релевантна информация. Тези данни могат да се използват за визуализиране в табличен формат или като основа за филтриране и сортиране.



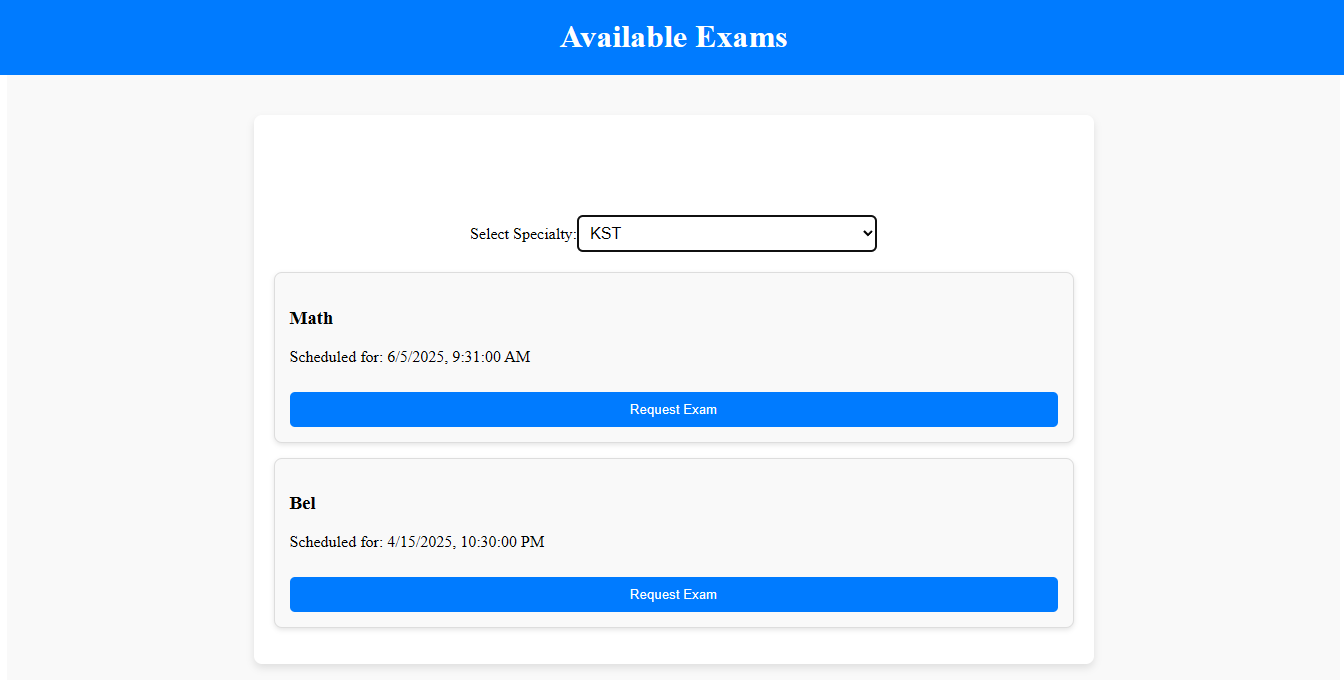
Фигура 47

Допълнително, при избор на конкретен кандидат от интерфейса — например при преглед на индивидуалният му профил — се извършва GET заявка към /candidates/{id}, където {id} е уникалният идентификатор на кандидата. Контролерът използва @PathParam, за да извлече идентификатора от пътя на заявката, и отново използва RegistrationService, за да намери съответния кандидат в базата. Ако кандидат с такъв идентификатор съществува, се връща HTTP отговор с код 200 (OK) и тялото на отговора съдържа обект от тип CandidateDTO. Ако кандидатът не бъде намерен, се връща отговор с код 404 (Not Found), което позволява на клиентската част да визуализира съответното съобщение за грешка или да пренасочи потребителя.



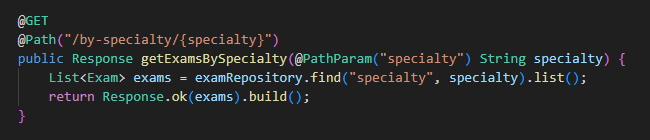
Фигура 48

Тази система осигурява стабилен и стандартизиран подход за достъп до данни за кандидатите. RESTful архитектурата на контролера улеснява интеграцията с клиентски приложения, като например уеб портали или вътрешни административни панели, и гарантира, че данните се предават в структуриран JSON формат, удобен за обработка и визуализация.

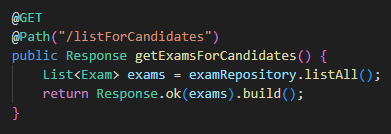
Фигура 49

Контролерът CandidateExamController, отговарящ за визуализиране на наличните изпити, предоставя RESTful интерфейс, който улеснява клиентската част на системата при зареждането на данни за изпити, свързани със специалности или всички достъпни изпити в системата. Тази функционалност е ключова както за кандидатите, така и за административния персонал, тъй като осигурява точна и навременна информация относно провежданите изпити.

При избор на конкретна специалност от страна на потребителя (например при попълване на заявление за явяване на изпит), интерфейсът извършва заявка към ендпойнта /api/exam/by-specialty/{specialty}. Методът използва @PathParam, за да извлече наименованието на специалността от URL адреса, и чрез ExamRepository извлича всички изпити, асоциирани с тази специалност. Това позволява динамично попълване на падащо меню или списък с възможни изпити за избраната специалност, което е особено полезно при кандидатстване или планиране на изпитна сесия.

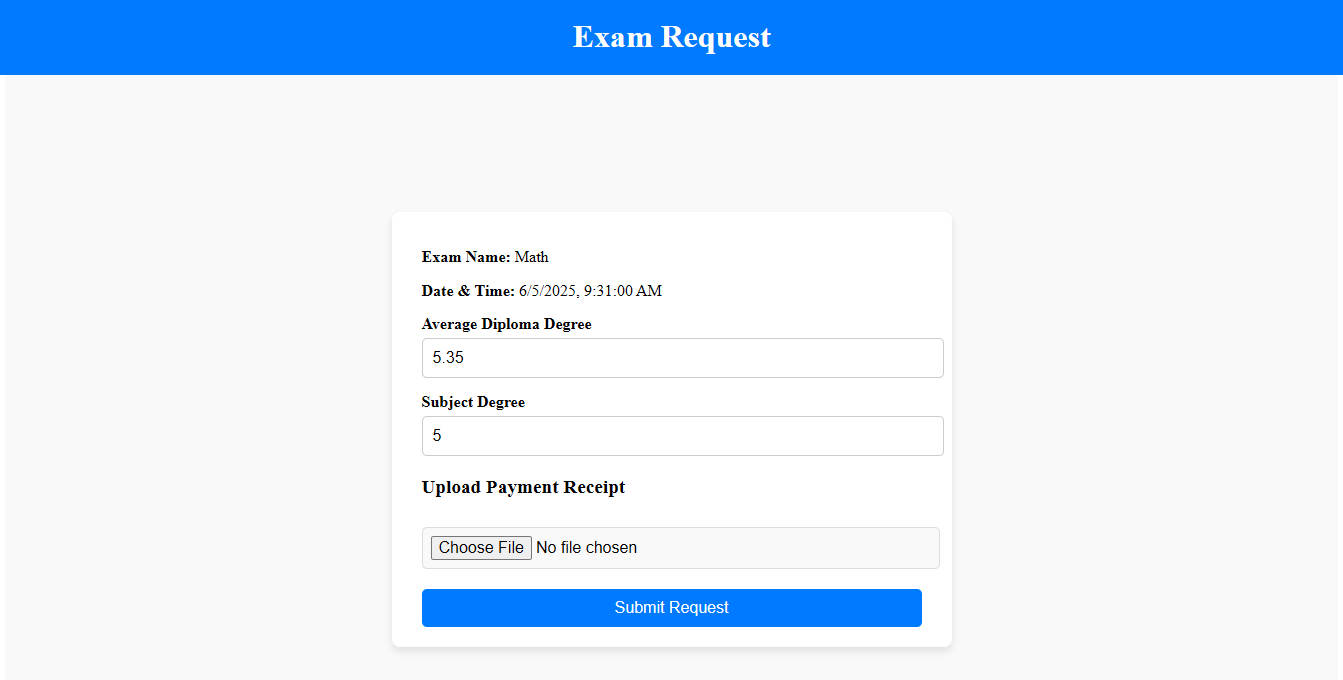
Фигура 50

Допълнително, чрез заявка към ендпойнта /api/exam/listForCandidates се извлича пълен списък с всички изпити, налични в системата, независимо от специалността. Този метод е полезен както за административни цели (например при поддръжка или анализ на провеждани изпити), така и за потребители, които желаят да прегледат общо разписание или да получат информация за изпити, към които все още не са заявили участие. Използването на examRepository.listAll() гарантира, че се връщат всички записи от съответната таблица в базата данни.



Фигура 51

И двата метода връщат резултатите в JSON формат, което позволява лесна интеграция с уеб клиентски приложения и гарантира стандартизирано представяне на данните. Чрез внедрените репозитории (ExamRepository, CandidateRepository, ExamRequestRepository), контролерът разчита на ефективен и структуриран достъп до базата от данни, което осигурява бърза обработка на заявките.

******

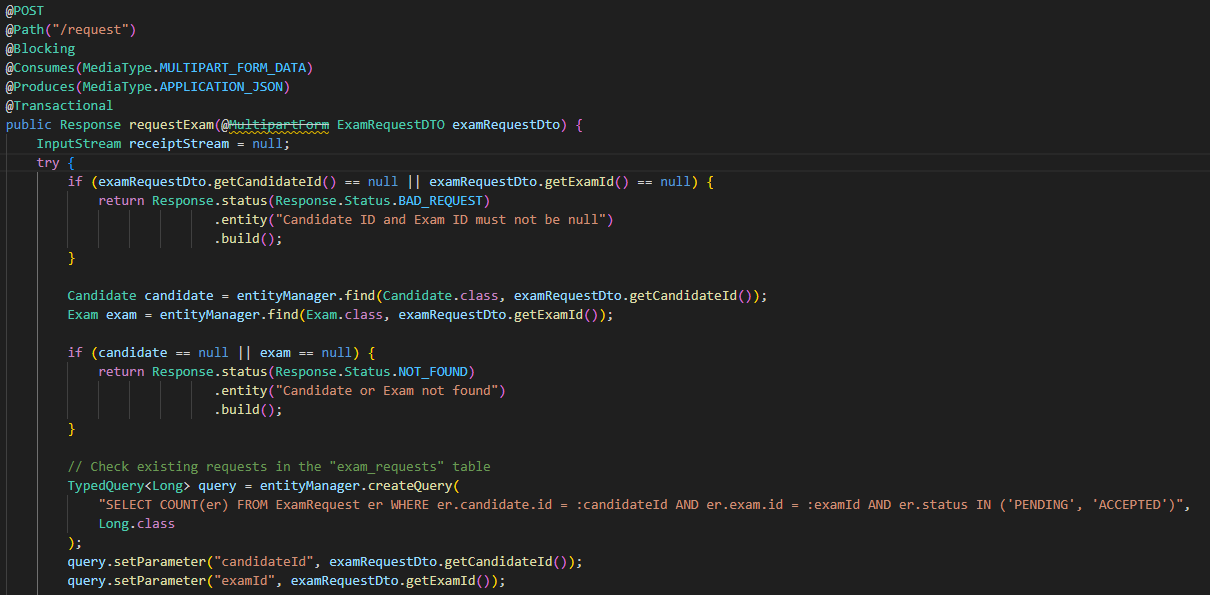
Фигура 52

Когато кандидатът попълни формата за записване в интерфейса на системата и натисне бутона за изпращане, се задейства заявка към ендпойнта /api/exam/request. Тази заявка се изпраща във формат multipart/form-data, който позволява подаване както на текстови стойности, така и на файл. Цялата информация се капсулира в обект от тип ExamRequestDTO, който включва идентификатора на кандидата, идентификатора на изпита, статуса на заявката, дата на подаване, оценки и самата прикачена разписка.

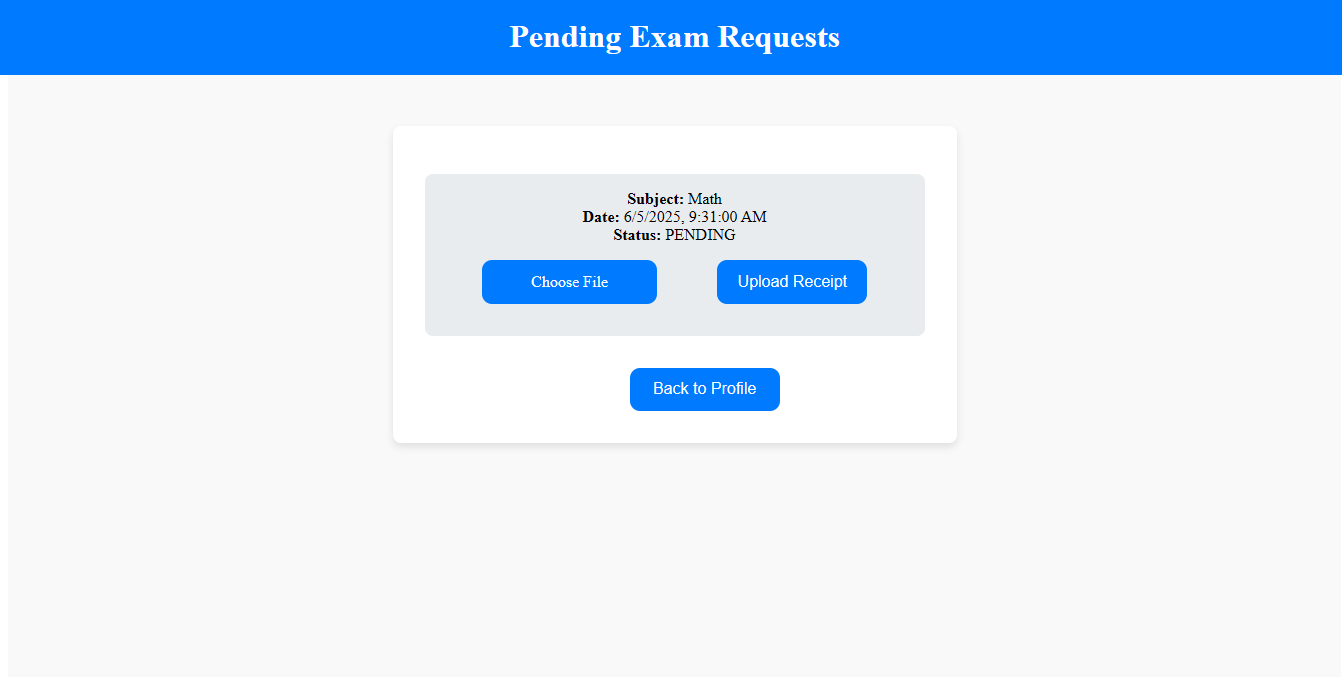
Първото, което методът прави, е да провери дали са попълнени задължителните полета – идентификатор на кандидат и изпит. Ако някое от тях липсва, системата връща грешка, с което се гарантира, че не се приемат непълни или невалидни заявки. След тази проверка се извличат съответните обекти кандидат и изпит от базата данни чрез EntityManager. Ако някой от тях не бъде открит, заявката се прекратява и се връща съобщение, че кандидатът или изпитът не съществуват.

След като се потвърди, че и кандидатът, и изпитът съществуват, се извършва проверка дали вече е подадена заявка за същия изпит от същия кандидат със статус PENDING или ACCEPTED. Това се прави с цел предотвратяване на дублиране. Ако такава заявка вече съществува, системата връща съобщение за конфликт, с което информира потребителя, че вече е кандидатствал за този изпит.

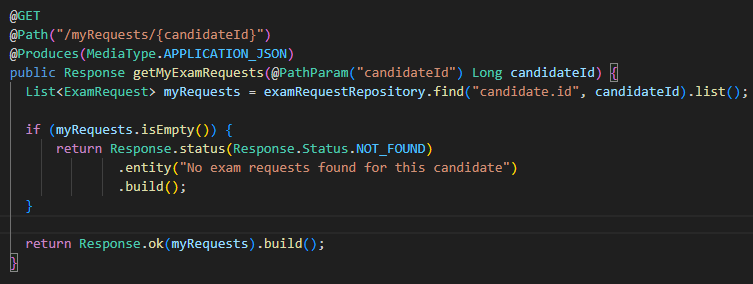
Ако няма предишна активна заявка, се създава нов обект от тип ExamRequest, в който се записват данните от формата. Прикаченият файл се прочита от потока и се преобразува в байтов масив, който също се съхранява в базата. След това заявката се запазва чрез репозитори и ако всички действия приключат успешно, методът връща потвърждение, че кандидатстването е извършено успешно.

Фигура 53

Тази функционалност позволява на системата да регистрира ясно, точно и надеждно всички подадени кандидатури за изпити. Същевременно осигурява автоматизация и защита от повторно записване, а наличието на прикачен файл предоставя възможност за по-пълна документация и бъдещи справки. Това прави процеса по кандидатстване прозрачен, лесен за използване и надежден от административна гледна точка.

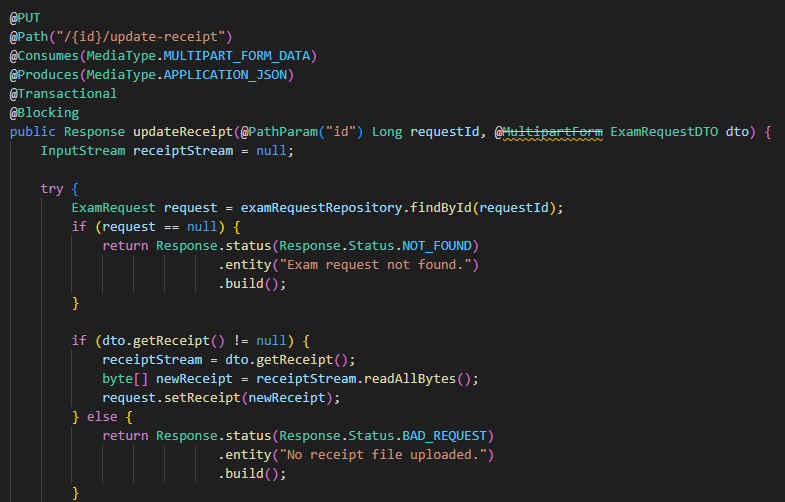
Фигура 54

Когато кандидатът отвори секцията „Моите заявки“ в клиентския интерфейс, се извършва заявка към ендпойнта /myRequests/{candidateId}. Методът получава като параметър идентификатора на кандидата и използва репозитория, за да извлече всички записани заявки от базата, свързани с този кандидат. Ако такива заявки не са намерени, се връща съобщение, че няма подадени заявления, което може да се визуализира на екрана като нотификация. В случай че има налични заявки, те се връщат като списък в JSON формат, който клиентът може да използва за визуализация в таблица или списък.

Фигура 55

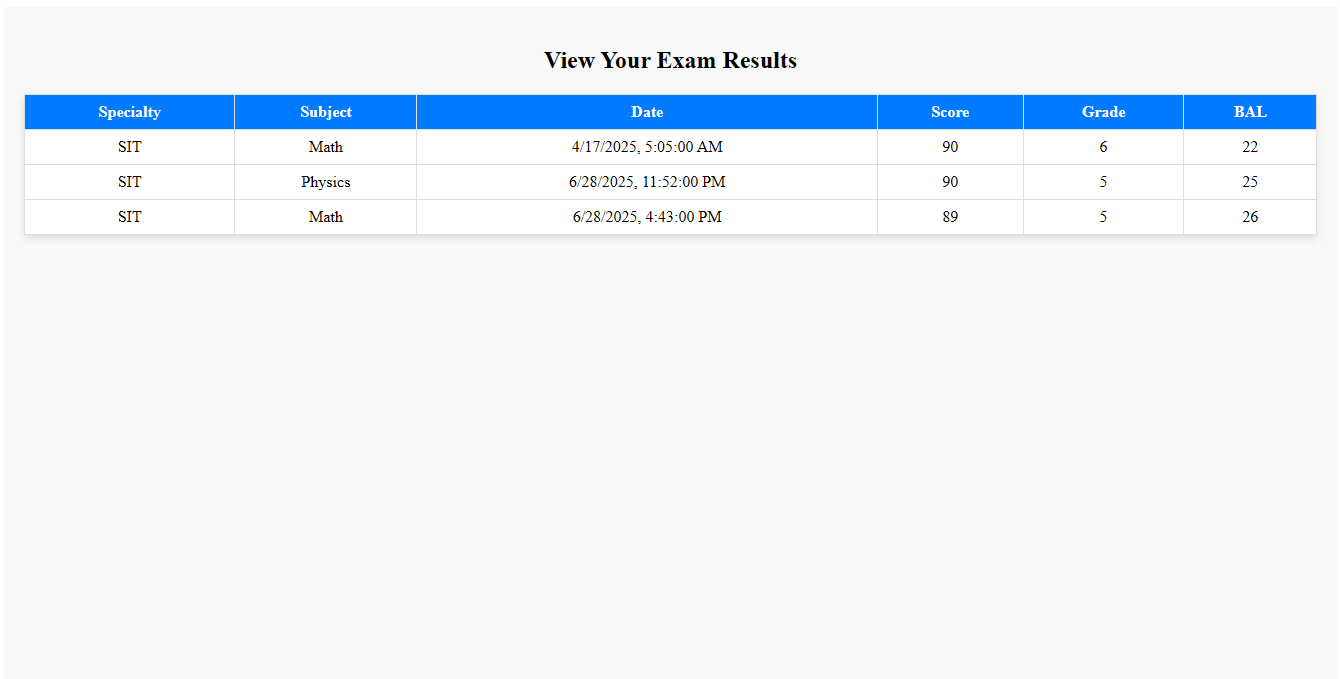
В допълнение към прегледа на заявките, контролерът предоставя и възможност за актуализиране на вече подадена разписка чрез ендпойнта /{id}/update-receipt. Този метод очаква идентификатора на заявката, която трябва да се актуализира, и новата разписка като файл, подаден във формат multipart/form-data. Първо се проверява дали заявката съществува в базата чрез подадения ID. Ако не бъде намерена, системата връща съобщение за грешка с код „не е намерено“. В противен случай, ако е прикачен нов файл, той се прочита от входния поток, преобразува се в байтов масив и се задава като нова стойност на разписката в съществуващия обект.

След успешно обновяване на платежната форма, заявката се записва отново в базата и се връща потвърждение, че разписката е обновена успешно. Ако не е подаден файл, методът връща съобщение, че не е открита прикачена разписка. При възникване на грешки по време на четене на файла, се връща съобщение за сървърна грешка с описание на проблема.



Фигура 56

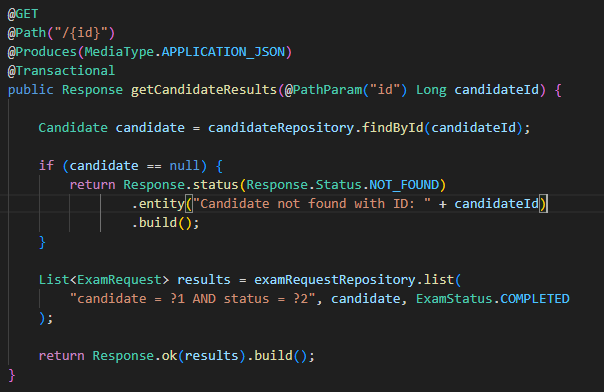
Тази функционалност е особено полезна в случаи, когато кандидатът е подал грешна или непълна разписка и иска да я подмени с вярна. Това гарантира по-голяма гъвкавост и точност в комуникацията между кандидатите и системата, както и по-ефективна обработка на кандидатурите от страна на административния екип.

******

Фигура 57

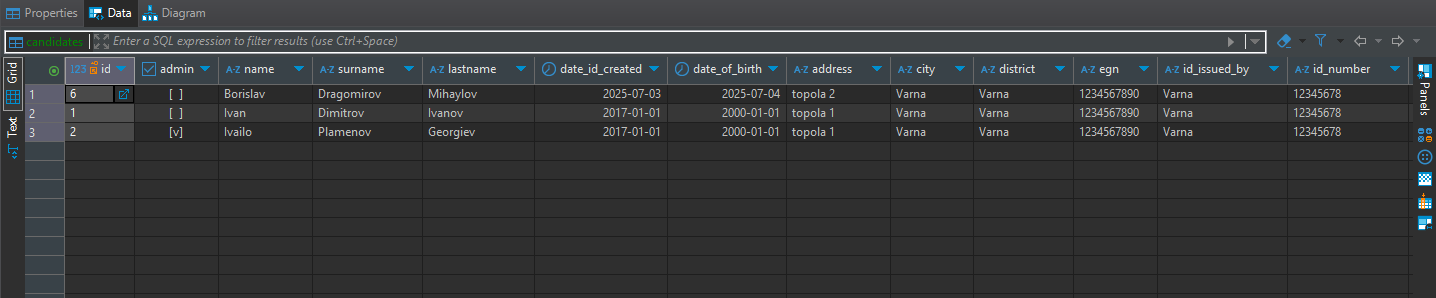
Когато кандидатът влезе в секцията „Моите резултати“, клиентският интерфейс извършва заявка към ендпойнта /api/exam/result/{id}, където {id} е уникалният идентификатор на кандидата. Методът първо проверява дали съществува кандидат с такъв идентификатор в базата чрез candidateRepository. Ако не бъде намерен, се връща съобщение за грешка със статус „не е намерен“, което позволява на клиентския интерфейс да информира потребителя, че такъв профил липсва.

Ако кандидатът съществува, се извличат всички записи от таблицата exam\_requests, които са свързани с този кандидат и имат статус „COMPLETED“, което означава, че кандидатът е участвал в съответния изпит и е оценен. Резултатите се връщат като списък от обекти ExamRequest, които съдържат информация за самия изпит, оценки и допълнителни данни, като бал и оценка по предмет.

Фигура 58

Тази справка е полезна както за самите кандидати, които искат да проследят своето представяне, така и за административния екип, който може да използва тази информация за статистика или последващи стъпки в процеса на класиране и прием. Системата по този начин осигурява пълна проследимост на процеса и достъпност на резултатите в реално време.

## **4.1 База данни**

Фигура 59

Колоната admin е от логически тип (bool) и показва дали даден потребител е администратор. Ако стойността е true, това е администраторски акаунт; ако е false, това е обикновен потребител.

Колоната date\_id\_created съдържа дата и показва кога е създаден записът за потребителя в базата данни.

date\_of\_birth е също от тип date и съдържа датата на раждане на потребителя.

id е уникален числов идентификатор (от тип int8), който автоматично се генерира. Това е основният идентификатор на реда.

Следващите колони съдържат лична информация за потребителя, като всички са текстови (varchar(255)), което позволява въвеждането на низове (т.е. текст до 255 символа).

* address, city, district, и municipality съхраняват адресна информация – улица, град, район и община.
* egn съхранява единния граждански номер – уникален идентификатор за всеки гражданин.
* email съдържа имейл адреса на потребителя.
* id\_issued\_by указва институцията, издала личната карта.
* id\_number е номерът на личната карта.

След това имаме няколко колони, свързани с имената:

* name, surname и lastname са различните части на името на потребителя.
* latin\_name, latin\_surname, и latin\_lastname са съответните имена, изписани с латински букви.

Колоната password съхранява паролата на потребителя, която е хеширана.

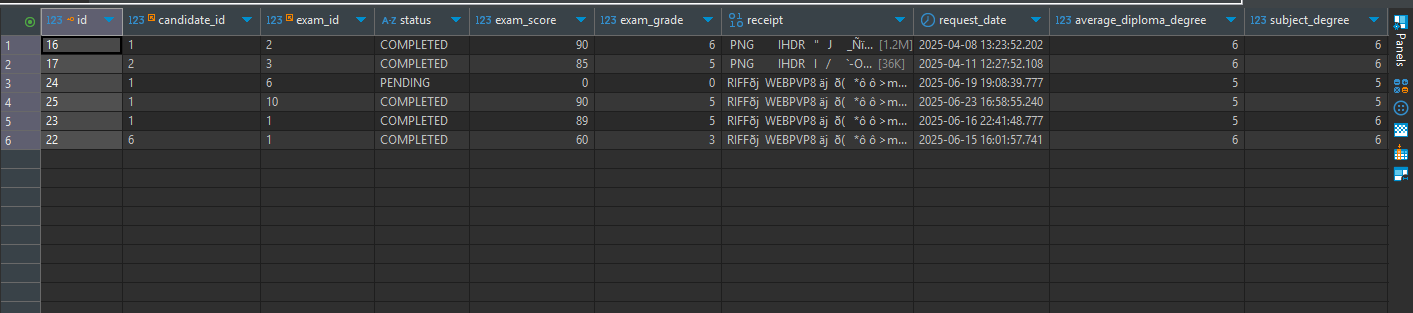
phone е за телефонния номер.

place\_of\_birth е мястото, където потребителят е роден.

Следващите три колони – school\_city, school\_name, и secondary\_education – съдържат информация за завършеното средно образование: градът на училището, неговото име и вида на образованието.

Колоната username е потребителското име, с което човекът се идентифицира при вход в системата.

Колоната reset\_token съдържа токен, използван за възстановяване на паролата – например когато потребителят е поискал нова парола чрез имейл. Свързано с нея е полето reset\_token\_expiry, което е от тип timestamp(6) и определя кога изтича валидността на този токен.

Фигура 60

Тази таблица съдържа информация, свързана с изпити и кандидати. Всеки ред в таблицата представлява участие на даден кандидат в даден изпит, с различни свързани данни.

Колоната exam\_score съдържа резултата от изпита. Тя използва числов тип float8, което означава, че може да съхранява десетични числа с висока точност. Задължително е да има стойност за тази колона (отбелязано е с „Not Null“).

Следва candidate\_id, който е цяло число (тип int8) и служи за идентификация на кандидата. Това поле не е задължително – може да има празни стойности.

Колоната exam\_id също е от тип int8 и указва конкретния изпит, към който се отнасят данните. И тази колона не е задължителна.

Колоната id е уникален идентификатор на реда в таблицата и също е от тип int8. Тя е маркирана като Identity, което означава, че стойността ѝ се генерира автоматично (например с автоинкремент).

Колоната status съдържа текстова стойност (от тип varchar(255)) и може да описва текущото състояние на нещо – например дали записът е активен, обработен, изчакващ и т.н. Тази колона има дефинирана стойност по подразбиране, но не е задължителна.

Колоната receipt е от тип bytea, което означава, че съдържа двоични данни – например сканирана разписка или друг файл, който е прикачен към записа. Не е задължителна.

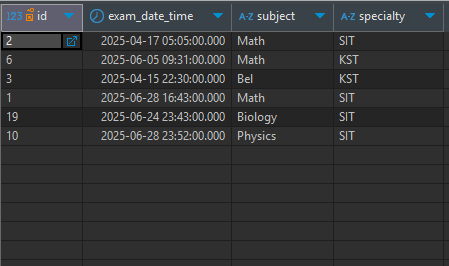
Колоната exam\_grade е отново числова (float8) и вероятно съдържа финалната оценка от изпита, различна от exam\_score. Това поле е задължително.

request\_date е дата и час (тип timestamp(6)) и показва кога е направено заявлението, вероятно за участие в изпита или за получаване на резултати. Това поле не е задължително.

average\_diploma\_degree съдържа средната дипломна оценка на кандидата и е от тип float8. Това поле също не е задължително.

subject\_degree вероятно съдържа оценка по конкретен предмет, който е свързан с изпита, и също е числова стойност.

Накрая, bal е още едно числово поле (float8), което може да се използва за изчисления като баланс или допълнителен коефициент, свързан с кандидатурата. И това поле не е задължително.

******

Фигура 61

Колоната id е уникалният идентификатор на записа. Тя е от тип int8 и се генерира автоматично (маркирана е като Identity By Default). Това е основният ключ, който различава всеки запис от останалите.

Колоната exam\_date\_time е от тип timestamp(6), което означава, че съдържа точна дата и час на провеждане на изпита, включително до наносекунди. Това поле не е задължително, но очевидно е ключово за графика на изпитите.

Колоната subject съхранява предмета на изпита – например „Математика“ или „Физика“. Типът ѝ е varchar(255), т.е. текст до 255 символа. Това поле не е задължително, но е важно за съдържанието на изпита.

Колоната specialty указва специалността, за която е предназначен изпитът – например „Софтуерно инженерство“ или „Медицина“. И тя е текстова с максимална дължина 255 символа и не е задължителна.

С други думи, тази таблица вероятно представлява списък с изпити, включващ предмета, специалността, и точния момент, в който се провежда всеки от тях.

## **5. Тестове**

**Използвал съм JUnit тестове**, които използват **библиотеката RestAssured** за изпращане и проверка на HTTP заявки и отговори. JUnit е най-популярната библиотека за писане на автоматизирани тестове в Java и служи като основа за организиране, изпълнение и валидиране на резултатите от тестовете.

В разработения тестов клас са използвани както правилни, така и неправилни тестове, които обхващат основни сценарии, свързани с логването в системата и създаването на нов изпит. Правилните тестове проверяват дали системата функционира правилно при коректно подадени входни данни – например при въвеждане на валиден потребител с вярна парола или при опит за създаване на изпит с пълно и валидно съдържание. Неправилните тестове от своя страна симулират неправилни или непълни заявки, с цел да се провери дали системата правилно открива грешките и отказва изпълнението на съответните операции. Такъв е случаят например при опит за вход с грешна парола или при създаване на изпит без задължителните полета като предмет, специалност и дата.

Използването на този тип тестове носи няколко важни предимства. На първо място, те автоматизират процеса по проверка на основните функционалности, като премахват нуждата от ръчно тестване, което би било бавно, неточно и податливо на човешки грешки. На второ място, чрез ясно дефинирани очаквания за всеки отделен сценарий се осигурява предсказуемо поведение на системата – ако тя не отговори по очаквания начин, тестът веднага ще сигнализира за проблем. Това значително улеснява поддръжката на системата и ускорява откриването на грешки, особено след промени в кода. Освен това, автоматичните тестове могат да бъдат интегрирани в процеса на непрекъсната интеграция (CI/CD), така че да се изпълняват автоматично при всяко обновяване на системата, което допринася за по-висока стабилност и надеждност на приложението в дългосрочен план. Така чрез внимателно подбрана комбинация от правилни и неправилни сценарии, тестовете гарантират, че системата реагира адекватно както при правилно, така и при неправилно подадени заявки.

******

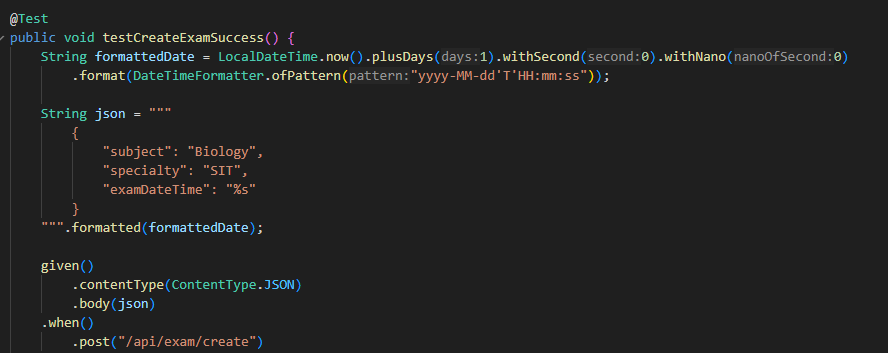
Фигура 62

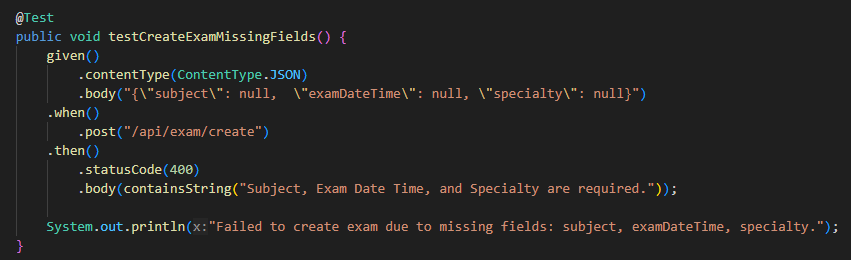
Този тестов клас в програмата служи за проверка дали модулът, който отговаря за вход в системата (тоест логване), работи правилно. В него са дефинирани два основни тестови сценария. Това не е част от самата програма, която се използва от кандидатите, а е инструмент, чрез който програмистите проверяват дали системата реагира както се очаква при подаване на определени входни данни.

Първият метод се казва testLoginSuccess. Той симулира сценарий, при който се прави опит за вход със съществуващ и валиден потребител, чиито имейл и парола вече са запазени в системата – в случая това е имейлът 250001 и паролата TPtp50$%. Програмата автоматично изпраща тези данни като част от заявка към сървъра, по същия начин, по който би го направил кандидатът, когато попълва формата за вход и натисне бутона Login. Сървърът трябва да върне отговор със статус код 200, което в уеб терминологията означава „успешна операция“. Освен това се проверява дали сървърът е върнал обратно „token“, което е уникален код за достъп, издаден при успешно влизане. Именно този токен позволява на системата да разпознае потребителя при следващи заявки, без той да се логва наново. Допълнително, се проверява и дали в тялото на отговора съществува свойството admin със стойност true. Това показва, че въпросният потребител има администраторски права и може да извършва действия, които обикновен потребител не може – като например да разглежда всички кандидати или да одобрява техни заявки. Ако всичко това се потвърди, програмата отпечатва съобщението „Login successful, token received.“ – то не се вижда от крайните потребители, но служи на разработчиците като индикация, че тестът е минал успешно.

Вторият метод е testLoginFailure, който цели да провери как ще реагира системата, ако се опитаме да влезем със съществуващо потребителско име, но с грешна парола. Използва се същият имейл адрес като при успешния сценарий, но паролата този път е грешна – „12345“. Системата в такъв случай трябва да откаже достъпа и да върне код 401, което означава „неоторизиран достъп“. Освен това се очаква в отговора да бъде включено текстово съобщение „Invalid credentials“ (грешни данни за вход). Това е ключово поведение за всяка защитена система – при грешна комбинация от потребител и парола, достъпът трябва да бъде отказан, за да се предотврати неоторизирано влизане. Ако това се случи, програмата отпечатва в конзолата съобщение „Login failed, invalid credentials.“

Целият този тестов код е написан с помощта на библиотеката RestAssured, която позволява автоматично тестване на уеб API – това са частите от програмата, които приемат заявки и връщат отговори между клиент и сървър. Тези тестове могат да бъдат стартирани автоматично всеки път, когато се прави нова версия на системата, и така се гарантира, че функционалността за вход работи правилно и не е била повредена при последни промени в кода. Без такива тестове, разработчиците биха били принудени ръчно да проверяват дали логването все още функционира, което е бавно, неточно и податливо на човешки грешки. Затова този подход не само спестява време, но също така осигурява много по-висока степен на надеждност и предсказуемост в поведението на системата.

Фигура 63

Фигура 64

Първият тестов метод се казва testCreateExamSuccess и симулира сценарий, при който администраторът създава изпит с напълно коректни данни. Най-напред в този метод се изчислява дата и час за изпита, които винаги са поне един ден напред от текущия момент – това е логична мярка, тъй като не може да се създава изпит в миналото или за текущия момент. След като датата се форматира във формат, който системата очаква, се изгражда текстова структура във вид на JSON – това е стандартен формат за обмен на данни между клиент и сървър. В нея ясно се вижда, че изпитът ще бъде по предмета „Biology“, за специалността „SIT“, и с конкретната изчислена дата. Този JSON се изпраща като тяло на заявка към адрес /api/exam/create, което е именно пътят, по който сървърната логика приема нови изпити.

След това се извършва проверка на отговора, който идва от сървъра. Очакваният статус е 200 – това е стандартният код, който показва, че операцията е успешна. Освен това, в отговора трябва да бъдат върнати данните за новосъздадения изпит – те се проверяват едно по едно: предметът трябва да е Biology, специалността – Sit (в случая с главна буква само S, което може би подсказва, че системата автоматично преобразува стойността), а датата трябва да съдържа точно тази стойност, която е била изпратена. Ако всичко това е вярно, това потвърждава, че сървърът е създал изпита успешно и че комуникацията между клиент и сървър е протекла точно според очакванията. Програмата отпечатва в конзолата съобщение, че изпитът е създаден успешно – това не е нещо, което се вижда от крайните потребители, но е важно за разработчиците и тестерите, които следят логовете.



Фигура 65

Вторият метод, наречен testCreateExamMissingFields, тества напълно противоположна ситуация. Тук целта е да се види как ще реагира системата, ако някой се опита да създаде изпит, без да предостави никаква информация – нито предмет, нито дата, нито специалност. В този случай се изпраща празен JSON, в който всички полета са зададени като null, тоест празни стойности. Сървърът трябва да разпознае тази заявка като невалидна и да откаже създаването на изпита, като върне статус код 400 – това е международен стандарт за грешка при „лоша заявка“ (bad request). Освен това се очаква сървърът да върне и ясно текстово съобщение, което да обяснява причината за отказа – в случая това е фразата „Subject, Exam Date Time, and Specialty are required.“, което показва, че липсата на който и да е от тези три елемента прави заявката невалидна. След това отново се отпечатва съобщение в конзолата, което информира, че създаването е отказано именно поради липса на задължителни полета.

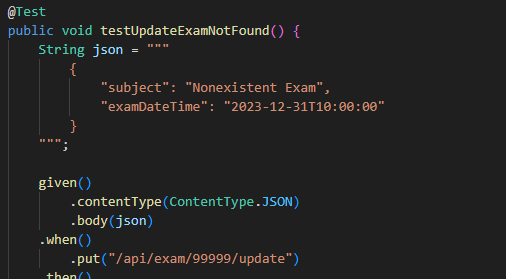


Фигура 66

Този тестов клас осигурява критично важна проверка на функционалността, която стои в основата на управлението на изпити. Без такива автоматични тестове, рискът от това да се допусне грешка, която позволява създаването на изпит с невалидна информация, би бил сериозен. Това би могло да обърка стотици кандидати, да повреди базата данни или дори да доведе до провал на цялата сесия. Затова този код не просто проверява техническа логика, а защитава целостта и надеждността на системата в реална работна среда.

******

Фигура 67

******

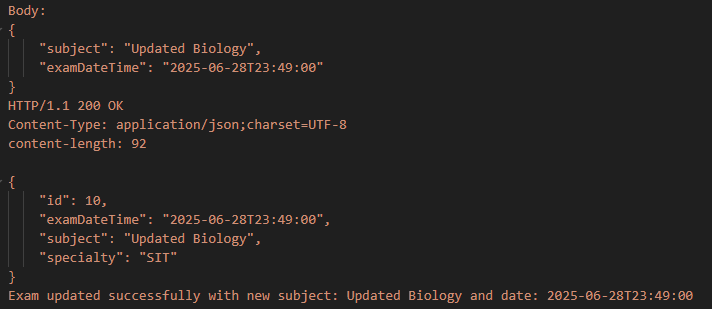
Фигура 68

Тестовият клас AdminUpdateExamTest съдържа два тестови метода, които са фокусирани върху функционалността за **редактиране на съществуващ изпит** в системата. От гледна точка на административния панел, това е съществена операция, тъй като е възможно да се допусне грешка при създаване на изпит – било то в името на предмета, в часа на провеждане или в други данни – и съответно е необходимо да има механизъм, който да позволява тези данни да бъдат коригирани.

Първият метод – testUpdateExam – симулира точно такъв случай, при който администраторът иска да промени вече съществуващ изпит. В примера се използва конкретен идентификатор – 10, което предполага, че в базата данни вече съществува изпит с това ID. Преди да се подаде заявката към сървъра, се създава нова дата за изпита, която е пет дни напред във времето – това гарантира, че тестът ще е валиден независимо кога се изпълнява. Тази дата се форматира във вид, който сървърът разбира – това е ISO 8601 формат, който съчетава дата и час.

Създава се JSON структура, в която се задават новото име на предмета – „Updated Biology“ – и новата дата. Тази структура е предназначена за подава към сървъра като тяло на PUT заявка, която е насочена към адрес /api/exam/10/update. Това е пътят, по който сървърът приема заявки за редакция на съществуващ изпит. От изпълнението на тази заявка се очаква да върне HTTP статус код 200 – знак, че операцията е успешна. Освен това, отговорът трябва да съдържа отразените нови стойности – предметът трябва да е актуализиран с новото име, а датата да съвпада с новата стойност. Ако всичко това е налице, това е знак, че редакцията е минала успешно, данните в базата са обновени, и сървърът коректно е възприел новите стойности.

След теста системата отпечатва в логовете съобщение, което допълнително потвърждава успеха – това е полезно при анализ на тестови отчети.

Фигура 69

Вторият метод – testUpdateExamNotFound – проверява как системата ще реагира, ако някой се опита да редактира изпит, който **не съществува**. В случая се използва ID 99999, което най-вероятно не отговаря на реално съществуващ запис в базата. Същата JSON структура се подава към сървъра, но при опит за редакция на невалиден изпит сървърът трябва да върне грешка. Конкретно, очакваният HTTP статус е 404, което означава „не е намерено“. Това е важна защита, защото ако подобна заявка се обработваше без проверка, съществува риск данни да бъдат добавяни или променяни некоректно, което може да доведе до объркване, загуба на информация или компрометиране на надеждността на цялата платформа.

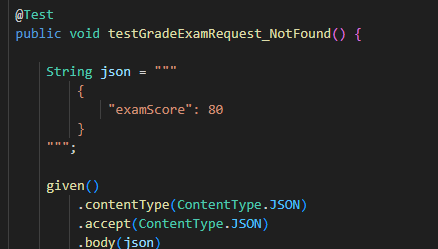
В този тест отново се отпечатва лог съобщение, което потвърждава, че системата правилно е отказала да редактира несъществуващ изпит и че грешката е засечена навреме.

Фигура 70

Този тестов клас играе важна роля в гарантирането на стабилността и целостта на административния модул за управление на изпити. Чрез него системата автоматично проверява, че съществуващите изпити могат да бъдат редактирани без проблем и че невалидни заявки се отхвърлят с подходящи съобщения и статус кодове. Това създава увереност, че администраторите могат безопасно да поддържат актуална информация за провежданите изпити, без риск от грешки или несъгласуваности в базата данни.

******

Фигура 71

******

Фигура 72

Тестовият клас AdminGradeExamTest представлява автоматизиран начин за проверка дали системата коректно обработва оценяването на кандидатски изпити. Той се използва в административната част на платформата – там, където администраторите ръчно въвеждат или актуализират оценките на кандидатите след провеждане на съответния изпит.

Първият метод в този клас – testGradeExamRequest – симулира реалистичен сценарий, при който администратор оценява вече подадена заявка за явяване на изпит. За да извърши тази операция, той изпраща към сървъра структура от данни, форматирана в JSON, която съдържа всичко необходимо, за да бъде пресметнат крайният бал (наречен BAL) на кандидата. В частност, тази структура включва: броя точки, получени на изпита (examScore), оценката от самия изпит (examGrade), текущия статус на заявката (status, който в случая е COMPLETED – тоест завършен), както и други данни, които са част от цялостната формула за оценяване – средната дипломна оценка (averageDiplomaDegree) и оценката по съответния предмет (subjectDegree).

Балът BAL в този пример се пресмята с формула, която явно е вградена логически в сървъра: (examGrade \* 3) + averageDiplomaDegree + subjectDegree. С конкретните числа в теста това се равнява на (5 \* 3) + 5 + 5 = 25. Преди да се направи заявката, тестовият метод изчислява тази стойност самостоятелно в променливата expectedBal, за да има с какво да сравни получения отговор от сървъра.

След това се изпраща PUT заявка към сървъра, насочена към пътя /api/exam/manage/25/grade. Числото 25 е ID на заявка за изпит, която предполагаемо вече съществува в базата данни и очаква да бъде оценена. В отговора от сървъра се очакват няколко неща: първо, че заявката ще бъде приета (статус код 200); второ, че всички полета, изпратени към сървъра, ще бъдат върнати обратно – това е индикация, че те са били правилно обработени и записани. И трето, че полето bal в отговора ще съдържа точно стойността 20.0, изчислена по формулата. Всички тези проверки са важни, защото те гарантират, че системата коректно обработва оценъчната логика и не допуска грешки в пресмятането или в съхраняването на данните.

След завършването на теста, системата извежда съобщение в конзолата с очаквания бал – това е от помощ при ръчно наблюдение на тестовете или при търсене на грешки.



Фигура 73

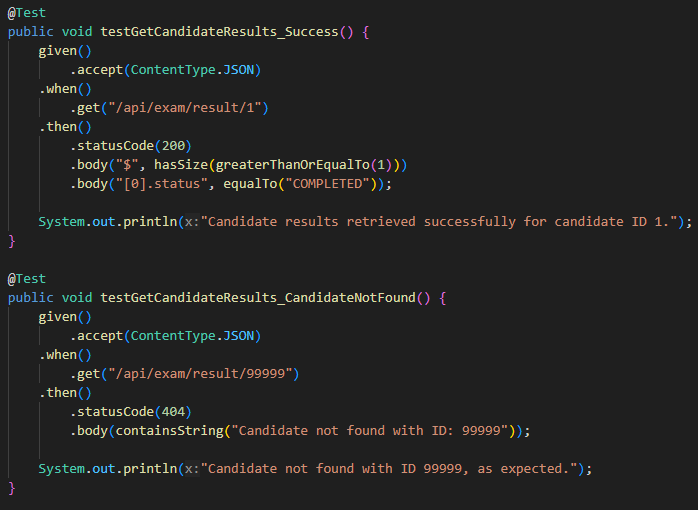
Вторият метод – testGradeExamRequest\_NotFound – разглежда ситуация, при която администратор се опитва да оцени заявка за изпит, която не съществува в системата. Това е важен тест от гледна точка на сигурността и надеждността – системата не трябва да обработва данни, свързани с невалидни или несъществуващи заявки. В този случай се използва ID 99999, което се предполага, че няма съответстваща заявка в базата данни. Подобно на предишния тест, се изпраща PUT заявка, но този път сървърът трябва да отговори със статус код 404, който означава "Not Found" – тоест, не е открит запис с такова ID. Освен това, в отговора трябва да присъства ясно съобщение, че заявка с такова ID не съществува.

И тук, както в първия случай, тестовият метод приключва с извеждане на съобщение в конзолата, което потвърждава, че системата правилно е отклонила невалидната заявка.



Фигура 74

С този тестов клас се проверява цялостната логика на **оценяване** в системата: от обработката на коректни заявки с правилна формула за изчисление на бала, до реакцията на системата при опит за манипулация или грешка. Такива тестове са изключително ценни в образователни платформи, където точността на оценките и доверието на кандидатите зависят изцяло от сигурността и предвидимостта на сървърната логика.

******

Фигура 75

Класът CandidateResultsTest представлява автоматизиран тест, който проверява дали сървърът правилно връща резултатите от изпити за конкретен кандидат в системата. Този тест е насочен към функционалността, чрез която кандидатите могат да видят своите приключени изпити и постигнати резултати, като комуникацията между клиент (уеб страница или приложение) и сървър се осъществява чрез HTTP заявки.

Първият метод в теста, testGetCandidateResults\_Success, проверява позитивен сценарий – ситуация, в която кандидат с ID 1 съществува в базата данни и има поне един завършен изпит. Заявката се изпраща до сървъра чрез GET метод, насочен към пътя /api/exam/result/1, където числото 1 е идентификаторът на кандидата. Изисква се отговор в JSON формат, което се задава с accept(ContentType.JSON). След като заявката бъде изпратена, се очаква сървърът да отговори със статус код 200, който означава успешно изпълнена операция.

След това следват две проверки върху самото съдържание на отговора. Първо, се проверява дали списъкът с резултати съдържа поне един елемент – това показва, че кандидатът има поне един завършен изпит. Проверката hasSize(greaterThanOrEqualTo(1)) гарантира това. Втората проверка потвърждава, че статусът на първия резултат в списъка е COMPLETED, което означава, че изпитът е проведен и оценен. Успешното преминаване на тези проверки показва, че системата правилно извлича и връща нужната информация за вече приключили изпити на конкретен кандидат. В края на теста се отпечатва съобщение, което информира, че резултатите са получени успешно.



Фигура 76

Вторият метод, testGetCandidateResults\_CandidateNotFound, проверява грешен сценарий – опит да се получат резултати за несъществуващ кандидат, в случая с ID 99999. Отново се изпраща GET заявка, но този път сървърът трябва да реагира различно, защото кандидат с такъв идентификатор няма. Очакваният статус код тук е 404, което означава „Not Found“. Освен това, отговорът трябва да съдържа ясно съобщение, че кандидат с това ID не е намерен. Тази проверка гарантира, че сървърът не връща празни резултати или технически грешки, а дава смислен отговор, когато кандидатът не съществува. И в този случай, в конзолата се отпечатва съобщение, потвърждаващо, че тестът е минал успешно – тоест, системата правилно е идентифицирала, че такъв кандидат няма.



Фигура 77

С този тестов клас се потвърждава надеждността на една от най-важните функционалности в системата – достъпа на кандидатите до собствените им резултати. Проверките не само гарантират, че се връщат правилни данни, но и че в случаите на грешка (например при търсене на несъществуващ кандидат), системата реагира адекватно и предсказуемо.

## **6. Заключение**

Системата за прием на кандидати за студенти е разработена с цел да отговори на изискванията от началото на дипломната работа. Тя осигурява платформа с модулен дизайн и лесна поддръжка за удобство както на кандидатите за студенти в университета. Главната й цел е да автоматизира и оптимизира процесите свързани с приема на студенти - запазване за изпити и подаване на документи и комуникация с кандидатите чрез последните технологии в софтуерно инженерство.

Системата съчетава клиентско приложение и сървърна логика, които комуникират чрез REST API. Използвани са технологии като Java и Quarkus за сървърната част, React и NextJS за разработката на потребителски интерфейс и PostgreSQL за съхранение на структурирана информация. Използването на контейнери през Docker улеснява внедрянето на системата в различни околности включително облачни платформи и осигурява стабилност, безопасност и контрол над конфигурацията.

Сред ключовите предимности на изграденото решение са модулният подход и ясно разделените отговорности между компонентите – от регистрация на потребители и управление на профили през подаването на заявки и качването на документи до проверката на платежни разписки. Системата е оборудвана също със съвременни механизми за защита като хеширане на пароли и употреба на токени за удостоверяване в добавка към защитата на информацията. Освен това има възможности за изпращане на автоматизирани електронни писма.

Освен това, предвиден е и административен панел с удобен интерфейс, чрез който админите могат да управляват всички аспекти на кандидатстудентския процес от добавяне и редактиране на изпити, през проверка на подадените документи, до генериране на справки и наблюдение на резултатите от класиранията. Благодарение на дефинирания подход за изчисляване на бал, кандидатите се класират по прозрачен и обективен начин, който елиминира възможността за грешки и спестява време на администраторите.

Особено внимание е обърнато на потребителския интерфейс, той е структуриран така, че да бъде ясен и логичен както за неопитни потребители, така и за специалисти. Предвидени са секции като „Моите заявки“, „Моите резултати“ и „Изпити“, които улесняват навигацията и позволяват на кандидатите да бъдат винаги информирани за състоянието на техните документи и класирания. Всичко това е постигнато чрез внимателно планиране, обмислен дизайн и последователно внедряване на функционалностите, които обединяват усилията на програмиране, сигурност, UX дизайн и системна архитектура.

След проведените тестове може да се заключи, че създадената система е не само стабилна и добре структурирана, но и напълно готова за реално внедряване в учебна институция. Тя отговаря на изискванията за надеждност и сигурност, като същевременно е проектирана така, че да позволява бъдещо разширяване и адаптиране към различни учебни заведения. В заключение, системата представлява едно ефективно решение за дигитализация на процеса по кандидатстване във висше учебно заведение.

## **Литература**

1. Java Documentation: <https://docs.oracle.com/en/java/>

2. Quarkus Documentation : <https://quarkus.io/guides/>[#](https://quarkus.io/guides/" \l "/)

3. Javascript: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

4. ReactJS Documentation: <https://react.dev/>

5. PostgreSQL Docs: <https://www.postgresql.org/docs/>

6. Docker Documentation: <https://docs.docker.com/>

7.PostgreSQL:[https://books.google.bg/books?hl=en&lr=&id=G8dh95j5NgcC&oi=fnd&pg=PR4&dq=postgresql&ots=8U53LXCVk3&sig=ojHSxulz0A3t-stC8e9CpmXrltw&redir\_esc=y#v=onepage&q=postgresql&f=false](https://books.google.bg/books?hl=en&lr=&id=G8dh95j5NgcC&oi=fnd&pg=PR4&dq=postgresql&ots=8U53LXCVk3&sig=ojHSxulz0A3t-stC8e9CpmXrltw&redir_esc=y" \l "v=onepage&q=postgresql&f=false)

Active Directory 101: <https://www.serveracademy.com/blog/active-directory-101-a-step-by-step-tutorial-for-beginners/>