

# Национална олимпиада по информационни технологии

**Тема:** Полуавтоматично проверяване на тестове със  
затворен отговор

**Автор:**

Християн Валерий Генчев, ЕГН: 9901066320, гр. София, жк.  
“Студентски град”, ТЕЛ: 0876565850, [hristiangelchev99@gmail.com](mailto:hristiangelchev99@gmail.com), ТУЕС -  
ТУ, 12 клас

**Ръководител:**

Красимир                      Росенов                      Чонов,                      0892238162,  
[krasimir.chonov@nokia.com](mailto:krasimir.chonov@nokia.com), Софтуерен архитект

# Резюме

## Увод

В днешния развиващ се свят, с всеки изминал миг, бумът на технологиите променя всяка сфера на обществото. С години в образованието и училищата рядко биват интегрирани нови научни открития или постижения. Средностатистическите училища нямат възможност да поддържат технологична база, която да ограмотява учениците. Много нови открития са в сферата на автоматизацията, но малка част от тях биват интегрирана в училищата. Слабата интеграция е в резултат на това, че те нямат паричната възможност за това или пък въобще нямат нужната техническа грамотност, за да го направят. В свят, където учителят има много важна роля в изграждане на бъдещето общество, както морално, така и научно, той трябва да е запознат с най-новите тенденции, както и да се опитва да ги налага, ако те са по-добри и водят до положителни промени.

Проверяването на бланки е трудоемка задача, недотолкова сложна, колкото отегчаваща и натоварваща. Всеки учител или преподавател се е сблъсквал с това, да проверява огромен брой контролни и знае, че е натоварващо и отнема ненужно много време.

Това ме провокира да създам просто и удобно приложение, което спестява много време, лесно е за употреба и не изисква професионални технически познания.

Линк към приложението: <https://evening-sierra-80012.herokuapp.com/>

Линк към repository: <https://github.com/hris11/AutomaticTestEvaluation>

## 1.1. Цели

### 1.1.1. Изисквания към приложението

Функционалните изисквания към уеб приложението са следните:

- потребителски профил
  - създаване на нов потребител
    - имейл
    - парола
    - име
    - фамилия
  - влизане
  - излизане
- създаване на бланка посредством меню, което има следните възможности:
  - създаване на универсална бланка
    - поле за име
    - поле за номер
    - поле за клас
    - поле за група
  - създаване на бланка за определен клас
    - дали имената на учениците да се визуализират
    - дали номерата на учениците да се визуализират
- възможност потребител да създаде бланка, без да е влязъл в или регистрирал профил
- след като потребителя е влязъл в профил той може да:
  - излезе от него
  - създаде нов клас
  - промени минало-създадени класове
    - възможност да се
      - трийт
      - добавят и премахват ученици

- принтират и добавят тестове
- резултати от тестове и радар диаграми
- търсене на резултатите въз основа на имейл и име на клас
- автоматично проверяване на бланките след качване на снимката в web приложението

## 1.1.2. Преглед на съществуващи решения

### 1.1.2.1. Eduleaf (<https://eduleaf.com/#>)

Eduleaf е уеб и мобилно приложение, което предоставя голям набор от средства, с които учители и преподаватели могат лесно и удобно да оценяват тестове и контролни работи.



Приложението позволява пълно и обстойно изследване на бланката като се нуждае от сканиране или снимка. Тестовите се проверяват автоматично, а отворените въпроси се оценяват ръчно от преподавателите. Предоставя се двупосочна връзка между учениците и преподавателите.

### 1.1.2.2. Exam Reader (<https://bebyaz.com/ExamReader>)

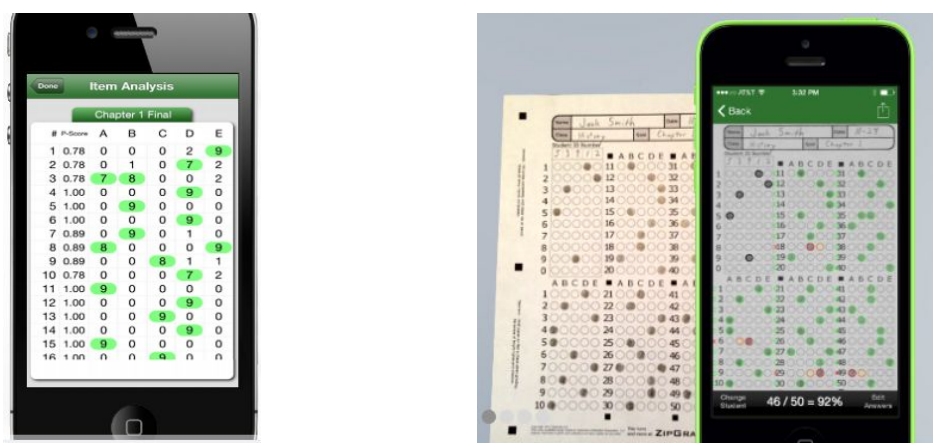
Exam Reader е мобилно приложение за android и iOS, което е пригодно да разпознава различни бланки като очаква потребителя да зададе определени параметри, по които програмата да проверява верността на верните отговори. Приложението може да проверява от 5 до 200 отговора. При спазване на зададените



условия успеваемостта е 100%. То също предлага възможност за изготвяне на бланка. ExamReader Cloud предоставя възможност на учителя да следи резултатите на неговите ученици и класове.

### 1.1.2.3. Zipgrade (<https://www.zipgrade.com>)

ZipGrade е приложение за телефон или таблет, което превръща устройството в машина за оптично класифициране, подобна на скантрон. Приложението обработва безплатни за сваляне листове за отговори в няколко размера. Предоставя обратна връзка към студентите, като обработват изходни тестове, викторини и оценки за оформяне, веднага щом завършат. Приложението предлага готови бланки, както и възможност за създаването на нови. Всеки потребител има лимит от 100 сканирания на месец като за безкраен брой сканирания потребителя трябва да заплати цена от \$6.99 за 1 година.



*Снимки от приложението*

## 1.2. Основни етапи в реализирането на проекта

Първоначално при създаването на приложението бе създадена клиентската страна. Това бе нужно, защото се нуждаех от модел на бланки, които всъщност алгоритъма да проверява. Създаването на немалка част от *front-end-a* доведе до изясняването на това, каква да е структурата на приложно-програмния интерфейс. След като знаех какви са основните заявки от които се нуждаех, както и структурата на базата, започнах с имплементирането на сървърната страна. Относно сигурността, първоначално добавих хеширане на паролите във базата, а по-късно в развитието на проекта добавих програмната рамка *Apache shiro*, която се грижеше за сесиите на потребителите. Успоредно с *back-end-a* развивах и функционалностите на *front-end-a*.

### **1.3. Ниво на сложност на проекта - основни проблеми при реализацията на поставените цели**

Субективността на този въпрос е много голяма, тъй като лично аз няма как да оценя проекта обективно и безпристрастно.

Проблеми срещнах със специфични грешки и конфигурации на различните библиотеки. Един от по-големите проблеми, който срещнах бе с изпращането на файлове към сървъра посредством *react* и *fetch API-то*. Бе нужно да се промени структурата на досегашната заявка, за да потдържа изпращането на *FormData*.

Имах проблеми със *set up-ването* на проекта под различни операционни системи породени от грешки в зависимостите на машината.

По време на разработката се натъкнах на много проблеми, който в последствие реших.

## 1.4. Логическо и функционално описание на решението

### 1.4.1. Структура на приложно-програмния интерфейс

Програмният интерфейс е набор от крайни точки, към които се изпращат заявки с цел получаване на информация или създаване на ресурси. Той работи на следния път “/rest/” и поддържа следните заявки:

#### 1.4.1.1. GET заявки - използват се, за да се получи дадена информация

- [/users/classes/{class\\_id}/blanks](#) - връща всички бланки на определен клас, чийто идентификационен номер е подаден в пътя
- [/users/classes/blanks/{blank\\_id}](#) - връща бланка, чийто идентификационен номер е подаден в пътя
- [/users/classes/blanks](#) - връща всички съществуващи бланки
- [/user/{user\\_id}/classes](#) - връща всички класове на потребител, чийто идентификационен номер е подаден в пътя
- [/user/classes/{class\\_id}](#) - връща клас, чийто идентификационен номер е подаден в пътя
- [/students/all/{class\\_id}](#) - връща всички ученици от определен клас, чийто идентификационен номер е подаден в пътя
- [/result/{blank\\_id}/marks](#) - връща всички оценки от определена бланка, чийто идентификационен номер е подаден в пътя
- [/result/{blank\\_id}](#) - връща списък на учениците с техните имена и оценки от определена бланка, чийто идентификационен номер е подаден в пътя
- [/result/{blank\\_id}/{student\\_id}](#) - връща оценката на определен ученик за определена бланка, чийто идентификационен номер е подаден в пътя
- [/files/materials/{blank\\_id}](#) - връща всички имена на материалите, които са публикувани за определена бланка
- [/files/{material\\_id}](#) - позволява на потребителя да свали материала, който желае

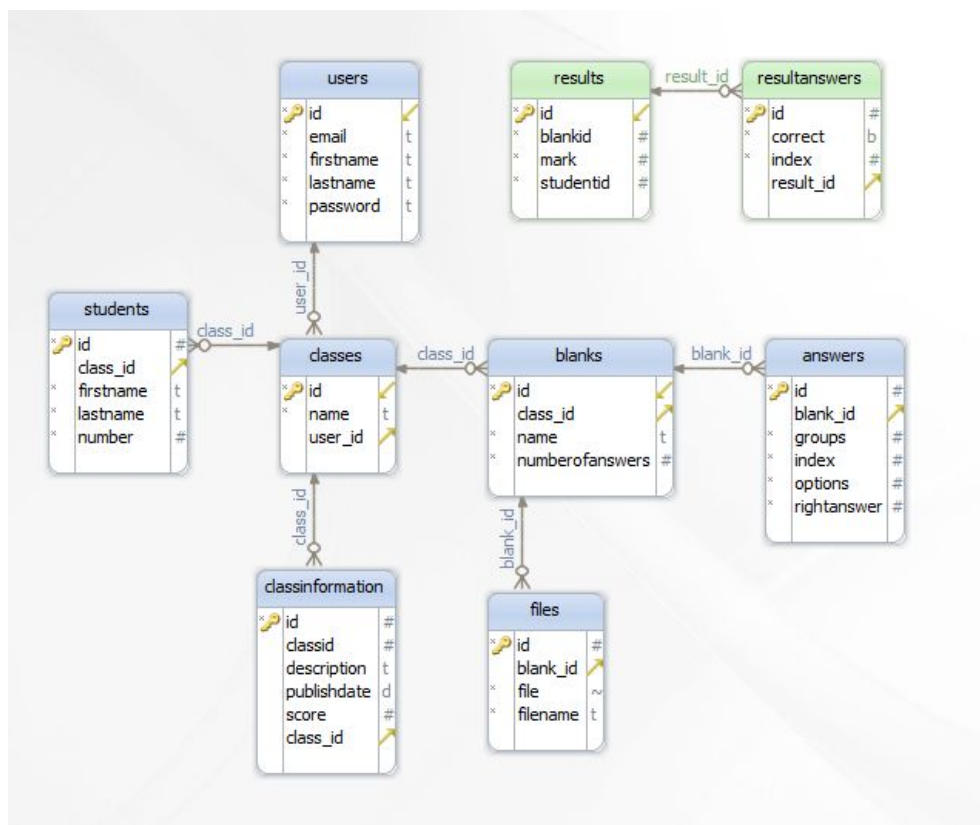


#### 1.4.1.2. POST заявки - използват се, за да се създаде нов ресурс

- [/user/classes](#) - получава класовете на даден потребител посредством получен имейл
- [/user/classes/new](#) - създава нов клас в базата
- [/image/upload](#) - праща снимка към алгоритъма за проверяване като тя не се запазва никъде
- [/auth/login](#) - проверява дали потребителя е въвел правилни данни за вписване
- [/auth/logout](#) - изчиства сесията на потребителя
- [/auth/register](#) - приема заявката, когато бъде създаден нов профил
- [/users/classes/{class\\_id}/blanks](#) - създава нова бланка за определен клас, чийто идентификационен номер е подаден в пътя
- [/students/new](#) - създава нов ученик и връща списък с настоящите
- [/files/upload/{blank\\_id}](#) - добавя нова група от материали за определена бланка

Необходимо е да се отбележи и акцентира, че всяка заявка за промяна на ресурс(всички заявки, различни от GET и тези за вписване и регистрация) трябва да бъде удостоверена преди да бъде изпълнена(потребителят трябва да е влязъл в профил и да е създал сесия).

### 1.4.2. Диаграма на базата данни



### 1.4.3. Структура на базата данни

В процеса на разработка на приложението бяха създадени 9 (девет) таблици, съдържащи информация за потребителите, техните класове, информация за класа, ученици (които са част от клас), бланки, отговори за бланките, резултати от бланки и резултатите на всеки въпрос.

Разработените таблици са следните:

- *users* - съдържа информация за потребителя
- *classes* - съдържа информация за всеки клас
- *classinformation* - съдържа информация за класа
- *students* - съдържа информация за всеки ученик
- *blanks* - съдържа информация за всички бланки
- *answers* - съдържа информация за всеки въпрос поотделно

- *results* - съдържа информация за резултатите на всеки ученик и всеки тест, който са направили
- *resultanswers* - съдържа информация за всеки въпрос от резултата
- *files* - съдържа материалите за определена бланка

#### 1.4.5.1. Схема на таблица “users”

Таблицата “users” съдържа информация за потребителите (таблица 5.1).

users		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор
<u>email</u>	character varying (50) not null	имейл адрес на потребителя
<u>firstname</u>	character varying (30) not null	собствено име на потребителя
<u>lastname</u>	character varying (30) not null	фамилно име на потребителя
<u>password</u>	character varying (30) not null	хеширана парола на потребителя

таблица 5.1

#### 1.4.5.2. Схема на таблица “classes”

Таблицата “classes” съдържа информацията за всеки клас (таблица 5.2). Тя има отношение много към едно с таблицата “users” (таблица 5.1) и едно към много с таблиците “classinformation” (таблица 5.3), “students” (таблица 5.4) и “blanks” (таблица 5.5).

classes
---------

Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор
<u>name</u>	character varying (30) not null	име на класа
<u>user_id</u>	integer, foreign key references users (id)	връзка с таблицата “users”

таблица 5.2

#### 1.4.5.3. Схема на таблица “classinformation”

Таблицата “classinformation” съдържа информация за всеки клас като тази информация се подава от учителя и се води като оценка на класа (таблица 5.3). Тя има връзка много към едно с таблицата “classes” (таблица 5.2).

classinformation		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор
<u>description</u>	character varying (255) not null	кратко описание на класа
<u>score</u>	integer not null	оценка на класа спрямо, максимум приет от учителя
<u>class_id</u>	integer, foreign key references classes(id)	връзка с таблицата “classes”

таблица 5.3

#### 1.4.5.4. Схемa на таблица “students”

Таблицата “students” съдържа информация за всеки ученик, който е член на даден клас. Тя има връзка много към едно с таблицата “classes”(таблица 5.2)

students		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор
<u>firstname</u>	character varying (50) not null	собствено име на ученика
<u>lastname</u>	character varying (50) not null	фамилно име на ученика
<u>number</u>	integer not null	номер на ученика в класа
<u>class_id</u>	integer, foreign key references classes(id)	връзка с таблицата “classes”

таблица 5.4

#### 1.4.5.5. Схемa на таблица “blanks”

Таблицата “blanks” съдържа информация за всяка бланка, която същевременно е направена и сочи само към един клас. Тя има връзка много към едно с таблицата “classes” (таблица 5.2) и едно към много с таблицата “answers” (таблица 5.6).

blanks		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор

<u>name</u>	character varaying (255) not null	име на бланката
<u>numberofanswrs</u>	integer not null	брой въпроси, съдържащи се в бланката
<u>class_id</u>	integer, foreign key references classes(id)	връзка с таблицата “classes”

таблица 5.5

#### 1.4.5.6. Схема на таблица “answers”

Таблицата “answers” съдържа информация за всеки отговор, който има определен номер и бланка, към която сочи. Тя има връзка много към едно с таблицата “blanks” (таблица 5.5).

answers		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор
<u>groups</u>	integer not null	идентификатор на групата на въпроса
<u>index</u>	integer not null	номера на въпроса в бланката
<u>options</u>	integer not null	броя на възможностите за отговор
<u>rightanswer</u>	integer not null	номера на верния вариант

<u>blank_id</u>	integer, foreign key references blanks(id)	връзка с таблицата “blanks”
-----------------	---	--------------------------------

таблица 5.6

#### 1.4.5.7. Схема на таблица “files”

Таблицата “files” съдържа материалите за всяка бланка. Тя има връзка много към едно с таблицата “blanks” (фиг. 5.5)

resultanswers		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key not null	уникален идентификатор
<u>file</u>	bytea not null	съдържание на материала
<u>filename</u>	character varying (255) not null	име на материала
<u>blank_id</u>	integer, foreign key references result (id)	връзка с таблицата “blanks”

таблица 5.7

#### 1.4.5.8. Схема на таблица “results”

Таблицата “results” съдържа информация за резултатите на всеки един ученик като съхранява кой е той и за коя бланка е резултата (таблица 5.8). Има връзка едно към много с таблицата “resultanswers” (таблица 5.9).

answers		
Поле	Зависимост	Описание
<u>id</u>	integer, primary key	уникален идентификатор

	<b>not null</b>	
<u>blankid</u>	<b>integer not null</b>	<i>идентификационен номер, който показва за коя бланка се отнася резултата</i>
<u>studentid</u>	<b>integer not null</b>	<i>идентификационен номер, който показва за кой ученик се отнася резултата</i>
<u>mark</u>	<b>integer not null</b>	<i>оценката на ученика за тази бланка</i>

таблица 5.8

#### 1.4.5.9. Схема на таблица “resultanswers”

Таблицата “resultanswers” съдържа информация за всеки един проверен въпрос (таблица 5.9). Тя има връзка много към едно с таблицата “results” (таблица 5.8).

resultanswers		
Поле	Зависимост	Описание
<u>id</u>	<b>integer, primary key not null</b>	<i>уникален идентификатор</i>
<u>correct</u>	<b>boolean not null</b>	<i>показва дали на въпроса е отговорено правилно</i>
<u>index</u>	<b>integer not null</b>	<i>номера на въпроса в бланката</i>
<u>result_id</u>	<b>integer, foreign key references result (id)</b>	<i>връзка с таблицата “result”</i>

таблица 5.9



## 1.4.4. Реализация на сървърна страна - Back-End

### 1.4.4.1. Създаване на модели - entities

Приложението използва обектно релационен мапър - *hibernate*, който използва *java* класове, за създаване на таблици и техните схеми. Това се получава с помощта на анотацията *@Entity*, поставена преди дефиницията на класа и добавянето на звено за съхранение в *persistence.xml*, наречено *Hibernate persistense provider*, в което се добавя пълния път към модела.

Приложението разполага с 9 (девет) модела, които репрезентират таблици от базата:

- *User.java* - репрезентира таблицата *users*
- *Class.java* - репрезентира таблицата *classes*
- *ClassInformation.java* - репрезентира таблицата *classinformation*
- *Student.java* - репрезентира таблицата *students*
- *Blank.java* - репрезентира таблицата *blanks*
- *Answer.java* - репрезентира таблицата *answers*
- *Result.java* - репрезентира таблицата *results*
- *ResultAnswer.java* - репрезентира таблицата *resultanswers*
- *File.java* - репрезентира таблицата *files*

Всички тези модели репрезентират схемите на техните таблици или това каква е структурата на всеки ред от определена таблица.

Приложението съдържа и 1 (един) модел, който служи за взимане на резултат от базата и не репрезентира таблица - *BlankMarks.java*. Този модел има анотацията *@Embeddable*.

#### **User.java**

Този клас репрезентира схемата на таблицата *users*, която се грижи за съхраняването на входните данни на потребителя.

```

@Entity
@Table(name = "users")
public class User implements Serializable {
    ...
}
User.java

```

фиг. 1.4.4.1

За да може *hibernate* да разпознае, че дадения клас репрезентира някоя таблица, то той трябва да се анулира с ануацията *@Entity*. Ануацията *@Table* оказва, че това е основната таблица за определено *entity* и ако параметърът *name* не бъде попълнен, то на таблицата се задава служебно име по подразбиране. Всеки един от моделите имплементира интерфейса - *Serializable*, за да окаже, че класът може да се сериализира и десериализира. В този интерфейс няма методи и той служи само за указание.

Всяка таблица съдържа уникален идентификатор, като в модела това се регламентира с ануацията *@Id* и начин за генериране на това число.

```

public class User implements Serializable {
    @JsonIgnore
    @Id
    @GeneratedValue(generator="increment")
    @GenericGenerator(name="increment", strategy = "increment")
    private Integer id;
    ...
}
User.java

```

фиг. 1.4.4.2

Ануацията *@JsonIgnore* указва, че когато един *JSON* обект трябва да бъде превърнат в *Java* обект от този тип, то полето *id* няма да бъде прочетено. *@GeneratedValue* и *@GenericGenerator* указват начинът, по който стойността се генерира и се присвоява на полето.

Класът съдържа полетата *email* (имейл) и *password* (парола), които заедно са уникалният ключ, с който потребителът може да влиза в профила си.

```

public class User implements Serializable {
    ...
    @Column(nullable = false)
    private String email;
    ...
    @Column(nullable = false, length = 32)
    private String password;
    ...
}
User.java

```

фиг. 1.4.4.3

Анотацията *@Column* указва, че това поле от класа ще е колона от таблицата, а стойностите указват следните ограничения - *nullable = false*, т.е. полето не може да бъде със стойност *null* (в схемата това се репрезентира с *not null*) и *length* (дължина), което ограничава дължината на полето (в схемата това се репрезентира със стойността в скобите на типа данни - *character varying* (максимална дължина). Ако дължината на полето не бъде променена, то нейната първоначална стойност е 255 символа.

Между таблиците *users* и *classes* има връзка едно към много, което означава, че един потребител може да има много класове. Това се реализира със създаването на поле в модела, чийто тип е лист от другия модел. Това поле се анотира с анотацията *@OneToMany*, която обозначава, че това поле ще има връзка с другата таблица, а анотацията *@JoinColumn* указва как ще се казва колоната, чрез която ще са свързани (фиг. 1.4.4.4). В първата анотация има поле *cascade*, което указва как ще се правят операции върху данните, които имат връзка с тази таблица. (Пример: когато трябва да се изтрие един потребител, то *CascadeType.ALL* ще позволи да се изтрие този запис с всички негови прилежащи записи от други таблици).

```

public class User implements Serializable {
    ...
    @OneToMany(cascade = {CascadeType.ALL})
    @JoinColumn(name = "user_id")
    private List<Class> classes;
}

```

```
...
}
User.java
```

фиг. 1.4.4.4

За да се получи пълната между двете таблици, то в модела на класа (Class.java) трябва да има поле с името, зададено от `@JoinColumn` (фиг. 1.4.4.5).

```
public class Class implements Serializable {
...
@Column(name = "user_id")
@JsonIgnore
private Integer userId;
...
}
Class.java
```

фиг. 1.4.4.5

Всички останали модели са реализирани по подобен начин като в някои от тях има специфични за колоната анотации. Например в класа `ClassInformation.java` има поле, което съхранява датата и има за стойност времето, когато е създаден записа (фиг. 1.4.4.6).

```
public class ClassInformation implements Serializable {
...
@Temporal(TemporalType.DATE)
@CreationTimestamp
@Column(nullable = false)
private Date publishDate;
...
}
ClassInformation.java
```

фиг. 1.4.4.6

Класовете на моделите имплементират *Serializable* интерфейс, който оказва, че те могат да се серилизират.

Всеки един от моделите съдържа празен конструктор (фиг. 1.4.4.7), който служи за инициализацията на таблиците от `persistense-unit`, предоставен от *hibernate* - *HibernatePersistenseProvider*.

```
public class User implements Serializable {
    ...
    public User() {}
    ...
}
```

фиг. 1.4.4.7

Моделите съдържат също и методи за взимане и запазване на стойности за всяко едно от полетата си, също са променени методите *equals()* и *hashCode()*.

#### 1.4.4.2. Връзка с базата и манипулация на данните

Обработката на данните в базата и връзката с нея става посредством класове (хранилища), които изпълняват методи за промяна или взимане на данни. Всяко от тези хранилища имплементира интерфейс с основните заявки за промяна на определен ред от базата - *insert*, *delete*, *update* (фиг. 1.4.4.8).

```
public interface RepositoryInterface<T> {
    public void insert(T t);
    public void delete(T t);
    public void update(T t);
}
```

фиг. 1.4.4.8

Всяко хранилище обработва определена таблица и имплементира интерфейса (фиг. 1.4.4.8) с модела, с който работи (фиг. 1.4.4.9).

```
public class UserRepository implements RepositoryInterface<User> {
    ...
}
UserRepository.java
```

фиг. 1.4.4.9

Достъпът до базата се осъществява чрез *entity manager*, предоставен от *javax.persistence.EntityManager* и всеки клас я има като инстанция (фиг. 1.4.4.10).

```
import javax.persistence.EntityManager;
public class UserRepository implements RepositoryInterface<User>{
```

```

private EntityManager entityManager;
@Inject
public UserRepository(EntityManager entityManager) {
    this.entityManager = entityManager;
}
...
}
UserRepository.java

```

фиг. 1.4.4.10

Конфигурацията на *entityManager* е направена в класът *DbModule.java*, където се имплементира как приложението да се свързва с базата. Това става посредством имплементация на методите *provideEntityManagerFactory()* (фиг. 1.4.4.11) и *provideEntityManager(...)* (фиг. 1.4.4.12).

```

@Provides
@Singleton
public EntityManagerFactory provideEntityManagerFactory() {
    ...
}

```

фиг. 1.4.4.11

```

@Provides
public EntityManager provideEntityManager(EntityManagerFactory
entityManagerFactory) {
    ...
}

```

фиг. 1.4.4.12

Обработката на данни става посредством описаните хранилища, в които се имплементират нужните методи за манипулация. Тези методи се викат от сервиси, в които е концентрирана бизнес логиката на приложението и проверките на входните данни.

```

@Override
public void insert(User user) {
    this.entityManager.getTransaction().begin();
    entityManager.persist(user);
    this.entityManager.getTransaction().commit();
}

```

```

}

@Override
public void delete(User user) {
    this.entityManager.getTransaction().begin();
    entityManager.remove(user);
    this.entityManager.getTransaction().commit();
}

@Override
public void update(User user) {
    this.entityManager.getTransaction().begin();
    entityManager.merge(user);
    this.entityManager.getTransaction().commit();
}

```

фиг. 1.4.4.13

На фигура 1.4.4.13 е показана имплементацията на всеки един от главните методи, които са записани в началния интерфейс, който всяко хранилище имплементира.

При разработката на приложението функционалността на базата се използва винаги, когато може с максимален потенциал, тоест взимането на специфични данни става посредством *query* заявки, които се възползват от езика и възможностите на базата.

```

public class ResultRepository implements RepositoryInterface<Result>
{
    ...
    public BlankMarks getBlankmarks(Integer blankId) {
        BlankMarks blankMarks;
        blankMarks = entityManager.createQuery(
            "SELECT NEW
genchev.hristian.automatictestevaluation.OutputModels.BlankMarks(" +
            " COUNT(CASE WHEN r.mark >= 2 AND r.mark < 3 then 2
end) as mark2," +
            ... +
            "FROM Result r WHERE blankid = :blank_id",
BlankMarks.class)
            .setParameter("blank_id", blankId)
            .getSingleResult();
    }
}

```

```

        return blankMarks;
    }
}

```

фиг. 1.4.4.14

Фигура 1.4.4.14 представява пример за възможностите, които базата предоставя чрез използване на *query-ma*. Всеки метод от хранилищата извършва промени в рамките на транзакция, което се постига чрез добавянето на *@Transactional* анотация пред метода на контролера. Транзакциите ни предпазват от нежелана промяна на информация, когато приложението спре или не работи правилно. В хранилищата почти няма натоварваща логика и всяка такава е изместена в сървисите.

### 1.4.4.3. REST контролери - крайни точки

Приложението предоставя възможност на потребителя да манипулира данните или да използва функционалностите му посредством крайни точки, описани в контролерите му. Тези заявки могат да бъдат *GET*, *POST*, *DELETE* и други. Приложението има 7 (седем) контролера, които се състоят в интерфейс (фиг. 1.4.4.15), съдържащ всички методи и клас (фиг 1.4.4.16), в който се имплементират.

```

public interface ResultREST {
    Result getStudentResult(Integer blankId, Integer studentId);

    List<StudentMark> getBlankResult(Integer blankId);

    BlankMarks getBlankMarks(Integer blankId);
}

```

фиг. 1.4.4.15

```

@Path("result")
public class ResultRESTImpl implements ResultREST {
    ...
}

```

фиг. 1.4.4.16



```

public class ResultRESTImpl implements ResultREST {
    private ResultServiceImpl resultService;
    @Inject
    public ResultRESTImpl(ResultServiceImpl resultService) {
        this.resultService = resultService;
    }
    ...
}

```

*фиг. 1.4.4.17*

На фигури 1.4.4.17 и 1.4.4.18 е показана имплементацията на един от методите в контролера за резултати. Всеки един контролер се анутира с анотацията *@Path* (фиг. 1.4.4.16), която указва пътят, на който ще се достъпва. Всичката логика е изместена в сървизи, които се добавят като полета на класа (фиг. 1.4.4.15) и се инициализират в конструкторите им с анотация *@Inject*, което позволява да се анулира грижата за неговата инициализация. Всеки контролер има връзка с един или повече сервизи и когато заявка трябва да бъде обработена, то това се случва в тях.

```

public class ResultRESTImpl implements ResultREST {
    ...
    @GET
    @Path("{blank_id}/marks")
    @Produces(MediaType.APPLICATION_JSON)
    @Override
    public BlankMarks getBlankMarks(@PathParam("blank_id") Integer
blankId) {
        return resultService.getBlankMarks(blankId);
    }
}

```

*фиг. 1.4.4.18*

Методите в контролерите (фиг. 1.4.4.18) се анутират с анотация, която указва какъв е техния тип - *@GET*, *@POST*, *@DELETE* и др. Те могат да имат анотацията *@Path*, като това указва на кой път в приложението се намират. Ако тази анотация за път не бъде добавена, то пътят на метода ще бъде същия като този на контролера.

Когато някоя заявка трябва да връща или/и приема определена информация, то на метода се добавят анотициите *@Produces* или/и *@Consumes* и в тях записват нужните хедъри, които дават сведение за това от какъв тип ще е информацията.

#### 1.4.4.4. Сервизи и бизнес логика (Services)

Сервизите са частта от приложението, в която се изпълнява бизнес логиката и проверките за валидност на данните. Те служат като медиатор между контролерите и хранилищата. Приложението се състои от 7 (седем) сървиса като всеки от тях се състои от интерфейс (фиг. 1.4.4.19), в който са методите и клас (фиг. 1.4.4.20), в който се имплементират.

```
public interface ResultService {  
    Result getStudentResult(Integer blankId, Integer studentId);  
  
    List<StudentMark> getBlankResult(Integer blankId);  
  
    BlankMarks getBlankMarks(Integer blankId);  
}
```

фиг. 1.4.4.19

```
public class ResultServiceImpl implements ResultService {  
    ...  
}
```

фиг. 1.4.4.20

Всеки сървиз има връзка с едно или повече хранилища в зависимост от това какви данни трябва да вземе или създаде (фиг. 1.4.4.21).

```
public class ResultServiceImpl implements ResultService {  
    private ResultRepository resultRepository;  
    private StudentRepository studentRepository;  
  
    @Inject  
    public ResultServiceImpl(ResultRepository resultRepository,  
        StudentRepository studentRepository) {  
        this.resultRepository = resultRepository;  
    }  
}
```

```

        this.studentRepository = studentRepository;
    }
}

```

фиг. 1.4.4.21

#### 1.4.4.5. Реализация на сигурност

Приложението използва библиотеката *Shiro*, която предоставя възможност да бъдат генерират токени, които се използват за създаване на потребителски сесии и сигурност на заявките.

Когато потребител влезе в профила си то за него се генерира токен, който му служи като ключ, за следващо функциониране на приложението.

```

public Response login(LoginUser loginUser) {
    ...
    int status = 401;
    Subject currentUser = SecurityUtils.getSubject();
    if (!currentUser.isAuthenticated()) {
        UsernamePasswordToken token = new
UsernamePasswordToken(loginUser.getEmail(), loginUser.getPassword());

        try {
            currentUser.login(token);
            status = 200;
        } ...
    }
    return Response.status(status).build();
}

```

фиг. 1.4.4.22

Вътрешно библиотеката *shiro* изпраща бисквитка с име *JSESSIONID* и стойност - генерирания *token*. Това нещо се случва след като методът *login(token)* се изпълни успешно и бъде оторизиран. Проверката на входните данни на потребителя се изпълнява в метода от фигура 1.4.4.23, който се изпълнява от метода *login()*.

```

@Override

```

```

protected AuthenticationInfo
doGetAuthenticationInfo(AuthenticationToken token) throws
AuthenticationException {
    UsernamePasswordToken upToken = (UsernamePasswordToken) token;

    LoginUser user = new LoginUser(upToken.getUsername(), new
String(upToken.getPassword()));
    if (!userService.authLoginUser(user)) {
        throw new IncorrectCredentialsException();
    }
    SimpleAuthenticationInfo authn = new
SimpleAuthenticationInfo(token.getPrincipal(),
token.getCredentials(), getName());

    return authn;
}

```

фиг. 1.4.4.23

В опциите за конфигуриране на библиотеката (фиг. 1.4.4.24) се добавят адресите, за които се очаква да бъде подаден *token*-а, за да може заявката да се уторизира и да се изпълни.

```

@Override
protected void configureShiroWeb() {
    try {
        bindRealm().to(CustomShiroRealm.class);
    } catch (SecurityException e) {
        addError(e);
    }

    this.addFilterChain("/rest/auth/**", ANON);
    this.addFilterChain("/rest/**", ANON);
    this.addFilterChain("/rest/users/classes/*/blanks", AUTHC);
    ...
}

```

фиг. 1.4.4.24

Методът *addFilterChain(a, b)*, получава два параметъра, от които *a* е релативния адрес, а *b* е опцията дали заявката трябва да съдържа *token*.

Всички пароли, които се съхраняват в приложението, са хеширани. Това се получава посредством сервиз (фиг. 1.4.4.25), който се грижи за тяхното хеширане. Методът за хеширане е *MD5*.

```
public String encryptPassword(String password) {
    String encrypted = null;
    try {
        byte[] bytesOfMessage = password.getBytes("UTF-8");
        MessageDigest md = MessageDigest.getInstance("MD5");
        byte[] thedigest = md.digest(bytesOfMessage);
        BigInteger bi = new BigInteger(1, thedigest);
        encrypted = bi.toString(16);
    } catch (UnsupportedEncodingException ex) {

        Logger.getLogger(SecurityService.class.getName()).log(Level.SEVERE,
        null, ex);
    } catch (NoSuchAlgorithmException ex) {

        Logger.getLogger(SecurityService.class.getName()).log(Level.SEVERE,
        null, ex);
    }

    return encrypted;
}
```

фиг. 1.4.4.25

#### 1.4.4.6. Реализация на алгоритъм за проверяване на бланки

Реализацията на алгоритъма за проверяване на бланки се реализира в 3 (три) отделни класа. Първият от тях е крайната точка (фиг. 1.4.4.26), на която се изпраща снимковия материал (*REST endpoint*).

@POST

```

@Path("upload")
@Consumes(MediaType.MULTIPART_FORM_DATA)
public void uploadImage(@FormDataTypeParam("file") InputStream
inputStream, @FormDataTypeParam("file") FormDataContentDisposition
fileDetail) throws Exception {
    imageService.uploadImage(inputStream, fileDetail);
}

```

фиг. 1.4.4.26

След като снимката бъде приета от крайната точка на приложно-програмния интерфейс, тя се изпраща към сървиз (фиг. 1.4.4.27), който се грижи за нейната обработка.

```

public class ImageService {
    private BlankService blankService;
    private ResultRepository resultRepository;

    @Inject
    public ImageService(BlankService blankService, ResultRepository
resultRepository) {
        this.blankService = blankService;
        this.resultRepository = resultRepository;
    }
    ...
}

```

фиг. 1.4.4.27

Сервизът има 1 (един) главен метод, който се извиква, когато се обработва получено изображение. Тъй като полученото изображение е от *InputStream* тип, то трябва да бъде превърнато в матрица (фиг. 1.4.4.28).

```

byte bytes[] = streamToByteArray(inputStream, fileDetail);
Mat src = Imgcodecs.imdecode(new MatOfByte(bytes),
Imgcodecs.CV_LOAD_IMAGE_UNCHANGED);
Core.rotate(src, src, Core.ROTATE_90_CLOCKWISE); //ROTATE_180 or
ROTATE_90_COUNTERCLOCKWISE

```

фиг. 1.4.4.28

Получената матрица се преобразува до *BufferedImage* и се подава на метод, който се опитва да прочете дали в изображението има *QR* код (фиг. 1.4.4.29). Ако има такъв,

приложението продължава напред, а ако няма - методът прекратява работа и връща съобщение за грешка.

```
BufferedImage bf = Mat2BufferedImage(src);  
String qrValue = "";  
try {  
    qrValue = readQr(bf);  
} catch ...
```

фиг. 1.4.4.29

Ако *QR* кодът е прочетен успешно, то информацията от него се извлича. Тя се състои в уникалния идентификатор на бланката и уникалния идентификатор на ученика.

След това на матрицата на изображението се прави копие, което минава през алгоритъм, който го прави абсолютно черно-бяло (изображението има само 2 (два) вида пиксели - абсолютно бели и абсолютно черни). Това служи за проверка дали кръгчето е запълнено или не.

```
Mat gray = new Mat();  
Imgproc.cvtColor(src, gray, Imgproc.COLOR_BGR2GRAY);  
Imgproc.medianBlur(gray, gray, 5);  
Mat circles = new Mat();  
  
Imgproc.HoughCircles(gray, circles, Imgproc.HOUGH_GRADIENT, 1.0,  
    (double) gray.rows() / 128, // change this value to detect  
    circles with different distances to each other  
    100.0, 30.0, 20, 70); // change the last two parameters  
// (min_radius & max_radius) to detect larger circles
```

фиг. 1.4.4.30

Представеният фрагмент от код (фиг. 1.4.4.30) се грижи за намирането на всички окръжности по бланката. Матрицата на изображението се трансформира от цветна в сива и се прилага филтъра *median blur*, който спомага за по-прецизното разчитане на изображението. След като бъде създадена новата матрица, на нея се прилага методът *Imgproc.HoughCircles(...)*, който се грижи за намирането на всички кръгове според определени параметри (максимален и минимален възможен радиус, раздалеченост и

други). Причината, поради която се задават максимален и минимален радиус, е защото ако няма ограничение, то алгоритъмът засича и неща, които в действителност не са окръжности или такива, които не формират пълна окръжност (букви и цифри).

```
List<Point> orderedPoints = getAllRowsOrdered(circles,  
blank.getNumberOfAnswers());
```

*фиг. 1.4.4.31*

Тъй като методът за намиране на окръжности връща намерените кръгове в разбъркан ред (алгоритъмът няма логичен и лесен за разбиране метод на намиране), то трябва всички намерени координати да бъдат подредени по определен ред. Затова се грижи методът (фиг. 1.4.4.31), чиято имплементация е на фигура 1.4.4.32.

```
public static List<Point> getAllRowsOrdered(Mat circles, Integer  
numberOfAnswers) {  
    List<Point> allOrdered = new ArrayList<>();  
    Point top = determineTop(circles);  
    double rad = circles.get(0, 0)[2];  
    for (int i = 0; i < numberOfAnswers; i++) {  
        allOrdered.addAll(orderRow(new Point(top.x, top.y - rad * 3 /  
4), new Point(top.x, top.y + rad * 3 / 4), circles));  
        /* трябва да се намери следващите по  
големина y*/  
        for (int x = 0; x < circles.cols(); x++) {  
            double[] c = circles.get(0, x);  
            Point center = new Point(c[0], c[1]);  
            double radius = c[2];  
  
            if (center.y >= top.y + (2 * radius) && center.y <= top.y  
+ (4 * radius)) {  
                top = center.clone();  
                break;  
            }  
        }  
    }  
    return allOrdered;  
}
```



фиг. 1.4.4.32

Методът намира най-горния ред от кръгове и вика метода *orderRow(...)* (фиг. 1.4.4.32) с параметри, които обозначават границите на реда. *OrderRow(...)* връща списък от подредените кръгчета на този ред. Така един вид, ред по ред, всички кръгове биват подредени, както следва да са в бланката.

```
private static List<Point> orderRow(Point left, Point right, Mat
circles) {
    double smallerY = left.y;
    double biggerY = right.y;
    List<Point> pointsOnRow = new ArrayList<>();
    if (left.y >= right.y) {
        smallerY = right.y;
        biggerY = left.y;
    }
    for (int x = 0; x < circles.cols(); x++) {
        double[] c = circles.get(0, x);
        Point center = new Point(c[0], c[1]);
        double radius = c[2] / 2;
        if (center.y >= (smallerY - radius) && center.y <= (biggerY +
radius)) {
            pointsOnRow.add(center.clone());
        }
    }
    pointsOnRow = sortRow(pointsOnRow);
    return pointsOnRow;
}
```

фиг. 1.4.4.33

След като координатите на окръжностите се подредят, то те се сравняват с броя, който е записан в базата. Ако те не съвпадна, то методът приключва своята работа и връща съобщение за грешка.

Ако броят намерени окръжности съвпада с този, който е записан в базата, то намерените координати, които са центрове на кръговете се итерират и се оценяват дали са маркирани правилно. Когато се проверява верността на даден отговор, то се проверява и

дали в бланката не са маркирани два или повече отговора на определен въпрос. Ако бъде открита такава нередност, то целият отговор се счита за грешно отговорен.

След като бланката бъде проверена, се формира оценка по формулата (фиг. 1.4.4.34)

$$2 + 4 * \frac{\text{брой получени точки}}{\text{максимален брой точки}}$$

фиг. 1.4.4.34

След като оценката бъде пресметната, то в базата, посредством хранилище, се прави запис на получения резултат (фиг. 1.4.4.35).

```
Double mark = 2.0 + (((double) numberOfCorrect / (double)
blank.getNumberOfAnswers() * 4));
result.setAnswers(resultAnswers);
result.setMark(mark);
result.setBlankId(blankId);
result.setStudentId(studentId);
resultRepository.insert(result);
```

фиг. 1.4.4.35

## 1.4.5. Реализация на клиентска страна - *Front-End*

### 1.4.5.1. React компонент

Компонентите контролират определена част от екрана. Логиката на един компонент се намира в клас (фиг. 1.4.5.1), който се състои от методи и променливи.

```
class LoginComponent extends Component {
  constructor(props){
    super(props);
    ...
  }
  ...
  render() {
    ...
  }
}
export default LoginComponent;
```

фиг. 1.4.5.1

Всеки компонент непременно трябва да има метода *render()*, който се репрезентира това, което ще показва. По време на цикъла на живот на всеки компонент се предлагат методи като *componentDidMount()*, *componentWillMount()* и други, които се викат в специфичен момент. Всеки компонент има и незадължителен конструктор, който ако не бъде имплементиран се използва подразбиращ се такъв. В конструктора може да се инициализира променливата *state*, която пази актуалното състояние на компонента.

#### 1.4.5.2. Node package manager(npm) - зависимости на клиентската страна

Зависимостите на клиентската страна са описани в *package.json* (фиг. 1.4.5.2).

```
{
  "name": "automatic-test-evaluation",
  "version": "0.1.0",
  "private": true,
  "proxy": "http://localhost:8080/",
  "dependencies": {
    "material-ui": "^0.20.0",
    "react": "^16.2.0",
    "react-cookie": "^2.1.2",
    "react-d3-radar": "^0.2.6",
    "react-dom": "^16.2.0",
    "react-qr-component": "^0.7.3",
    "react-scripts": "1.0.14",
    "react-tap-event-plugin": "^3.0.2"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject",
    "heroku-prebuild": "ln -s ../node_modules
automatic-test-evaluation/node_modules"
  }
}
```

фиг. 1.4.5.2

В този документ са описани името на проекта, версия, *proxy* (пренасочва заявки към специфициран адрес), зависимости към други модули и др.

### 1.4.5.3. Fetch приложно-програмен интерфейс(API)

*Fetch* е най-модерният начин за изпращане на *AJAX* заявки към сървър. Той предлага обща дефиниция на обектите за *Request* и *Response*.

```
const post = (url, headers, body, callback, fileUpload) => {
  let head = new Headers();
  if (headers) {
    Object.keys(headers).forEach(function (key) {
      head.append(key, headers[key]);
    });
  } else if (!fileUpload) {
    head.append("Content-Type", "application/json");
  }
  const options = {
    method: 'POST',
    body: fileUpload ? body : JSON.stringify(body),
    headers: head,
    mode: 'cors',
    credentials: 'same-origin'
  };
  fetch(url, options)
    .then(callback)
    .catch(function (error) {
      console.error(error);
    });
};
```

фиг. 1.4.5.3

На фигура 1.4.5.3 е представен фрагмент от код, който се използва в приложението. Това е *Fetch POST* заявка, която се извиква всеки път, когато клиентът трябва да изпрати заявка от тип *POST*. Всички заявки (*GET*, *POST*, *DELETE* и други) са отделени в един файл - *RestCalls.js*, на който се прави инстанция и се вика нужния метод. Всяка заявка се

изпълнява асинхронно, поради което се подава *callback* метод, който се изпълнява, когато заявката приключи.

#### 1.4.5.4. Използване на външни компоненти

В приложението се използват някои готови библиотеки, които предлагат външни компоненти с готова функционалност. Такива компоненти в приложението са тези, предоставени от *material-ui* (всички бутони, менюта, спинери, полета за въвеждане и други компоненти от потребителския интерфейс), *react-d3-diagram* (радарната диаграма, с която се представят резултатите на определена бланка) и *react-qrcode-component* (генерира *QR* код посредством даден параметър). В документацията на тези модули е описано как те се използват, какви параметри могат да получават и как се добавят (*import* (фиг. 1.4.5.4)).

```
import {CircularProgress, FlatButton, Paper, RaisedButton, Step,
StepLabel, Stepper} from "material-ui";
```

фиг. 1.4.5.4

#### 1.4.5.5. Реализация на главно меню

Главното меню на приложението е осъществено чрез *material-ui* компонента `<Tabs/>` (фиг. 1.4.5.5)

```
render() {
  return (
    <div className="navigation-main">
      <MuiThemeProvider>
        <Tabs
          value={this.state.value}
          onChange={(value) => this.handleChange(value)}
        >
          <Tab
            {/* заглавна част */}
            ...
          >
            {/* тяло */}
            ...
          </Tab>
        </Tabs>
      </MuiThemeProvider>
    </div>
  );
}
```

```

        <Tab>
            ...
        </Tab>
    </MuiThemeProvider>
</div>
);
}

```

фиг. 1.4.5.5

Всеки *Tab* разполага със заглавна част, която се визуализира в менюто и може да съдържа заглавие, икона и други. Разполага и с тяло, което се визуализира под менюто, когато заглавната му част бива избрана.

#### 1.4.5.6. Реализация на вписване и създаване на профил

Компонентите, които се грижат за създаване на нов профил и влизане в съществуващ, са разграничени в 3 (три) отделни класа:

- компонент, който се грижи за вписване (има сходна структура като фиг. 1.4.5.6)
- компонент, който се грижи за регистриране (фиг. 1.4.5.6)
- компонент, който се грижи за смяната между вписване и регистриране

```

class RegisterComponent extends Component {
    // конструктор
    ...
    // методи за валидация
    ...
    // методи, които връщат съобщения за грешка
    ...
    // метод за изпращане на информация
    ...
    // render() метод
}

```

фиг. 3.2.6

Компонентът, който се грижи за смяната на вписване и регистриране е много обемист, но в общи линии работи по следния начин. Когато компонента се зареди за

първи път, то той зарежда компонента, който е за вписване(фиг. 1.4.5.7). Под него зарежда и бутон(фиг. 1.4.5.8), който предоставя възможност потребителя да избере да се регистрира ако в момента му се предоставя възможност за вписване (и обратното, когато се натисне)

```
componentWillMount() {  
  let loginScreen = [];  
  loginScreen.push(<Login  
    logged={this.props.logged}  
    login={(email) => this.props.login(email)}  
    setMail={(mail) => this.props.setMail(mail)}  
    key="LoginFirstMount"  
    parentContext={this}  
    appContext={this.props.parentContext}/>);  
  ...  
});  
}
```

*фиг. 1.4.5.7*

```
render() {  
  return (  
    ...  
    <RaisedButton  
      label={this.state.buttonLabel}  
      primary={true}  
      style={style}  
      onClick={(event) => this.handleClick(event)}  
    />  
    ...  
  );  
}
```

*фиг. 1.4.5.8*

Когато потребител иска да създаде нов профил или иска да влезе в съществуващ, то полетата, които попълва се проверяват от набор методи за проверка на валидност и ако не изпълняват дадените условия(фиг. 1.4.5.9), то тогава потребителя не може да изпълни заявката(бутон е блокиран(фиг. 1.4.5.9) и не може да се натисне)

```

submitStateChecker() {
  if (this.state.firstNameSubmitState === true &&
      this.state.lastNameSubmitState === true &&
      this.state.emailSubmitState === true &&
      this.state.passwordSubmitState === true) {
    return true;
  } else {
    return false;
  }
}
...
<RaisedButton
  ...
  disabled={!this.handleSubmitButtonState()}
/>

```

фиг. 1.4.5.9

### 1.4.5.7. Създаване на бланка

Проектирането на бланка се състои в няколко стъпки, които потребителят трябва да завърши, за да може да я създаде в завършен вид. Стъпките се състоят избор на атрибутите на теста (бланката) и структурирането му.

Тази функционалност е разработена посредством компонент от *material-ui* - `<Stepper/>`. Всяка информация се пази в главния (родителски) компонент и всяка промяна в някое *deme* се осъществява чрез *callback* методи, които променят *state-a*.

За избора на атрибутите на бланката се грижи компонент (фиг. 1.4.5.10), който е разделен на подкомпоненти за отделните менюта.

```

<div className="menu-bar">
  <div className="default-options-bar">
    <div>
      Изберете колко опции да имат
      отговорите по подразбиране
    </div>
  </div>
</div>

```



```

    </div>
    <DropDownMenu
      ...
    >
      ...
    </DropDownMenu>
  </div>
  <BlankMenuToggles
    ...
  />
  <BlankMenuSlider
    ...
  />
</div>

```

фиг. 1.4.5.10

Менюто, което се грижи за заглавните параграфи (*име, номер в клас* и други) има различна структура в зависимост дали структурирането на бланка е извикано от главното меню или за определен клас.

```

handleDisplayRight() {
  if (this.props.parentProps.navCall === true) {
    return (<div>
      <Toggle
        className="toggleMargins"
        label="Поле за група"
        ...
      />
      <Toggle
        className="toggleMargins"
        label="Поле за клас"
        ...
      />
    </div>);
  } else {
    return (
      <Toggle
        className="toggleMargins"

```

```

        label="Номер на учениците от
списъка"
        ...
    />
    );
}
}

```

фиг. 1.4.5.11

Това се постига чрез използването на методи (фиг. 1.4.5.11), които връщат различен резултат в зависимост от това от къде е извикан компонента.

Когато потребителят избере атрибутите на теста, пред него се появява компонент (фиг. 1.4.5.12), който се грижи за групиране на въпросите, избиране на правилен отговор, промяна на въпрос и промяна на определена група.

```

<div>
  {this.handleDisplayGroupBar()}
  <Divider/>
  <ol className="hr-group-list">
    {elements.map((elem) => elem)}
  </ol>
</div>

```

фиг. 1.4.5.12

Отново компонента предлага различни резултати в зависимост от къде е извикан (фиг. 1.4.5.13).

```

handleDisplayGroupBar() {
  if (this.props.navCall === true) {
    return null;
  } else {
    return (
      <GroupsBar
        ...
      />
    );
  }
}

```

```
}
```

фиг. 1.4.5.13

Данните на този компонент отново се пазят в родителския *state* и когато потребител ги променя те се променят там.

Когато потребителя завърши стриктурата на бланкта, то пред него се визуализира именно тя. За нейната визуализация се грижат група от компоненти с родител - PrintPage.js (фиг. 1.4.5.14). Той получава нужната му информация от компонента, от който е извикан. Това позволява неговото преизползване без да се минава през стъпките за структуриране.

```
<div className="blank-for-print">
  <div className="site-header">this blank is created with automatic
test evaluation</div>
  <div className="main-header">
    <div className="content-headers">
      <div className="headers-floating" >
        <div className="header-titles" >
          <div className="name-header">{nameHeader[0]}</div>
          <div
className="class-header">{classHeader[0]}</div>
          <div
className="number-header">{numberHeader[0]}</div>
          <div
className="group-header">{groupHeader[0]}</div>
          </div>
          <div className="header-fields">
            <div
className="header-field">{nameHeader[1]}</div>
            <div
className="header-field">{classHeader[1]}</div>
            <div
className="header-field">{numberHeader[1]}</div>
            <div
className="header-field">{groupHeader[1]}</div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

```

        <div className="header-qr">
            {this.getQr()}
        </div>
    </div>

    </div>
</div>
<hr className="test-header-divider"/>
<div className="blank-lines-block">
    <OptionsHandler
        sliderValue={this.props.sliderValue}

eachAnswerNumberOfOptions={this.props.eachAnswerNumberOfOptions}
    />
</div>
<hr className="test-header-divider"/>
</div>

```

фиг. 1.4.5.14

### 1.4.5.8. Преференции и управление на профил

Функционалността на профилната страница е постигната в компонента AccountManager.js. Структурата му се състои от едно меню от бутони - фигура 1.4.5.15 и екран, на който се визуализират различни компоненти - фигура 1.4.5.16.

```

<div className="account-navigation">
    {/*
    navigation:
    -   logout
    -   create class
    -   compare classes
    -   etc
    */}
    <RaisedButton
        ...
    />
    ...

```

```
</div>
```

фиг. 1.4.5.15

```
<div className="account-workplace">  
  <Paper zDepth={2}>  
    <div className={displayClass}>  
      {this.getContent()}  
    </div>  
  </Paper>  
</div>
```

фиг. 1.4.5.16

Методът *getContent()* (фиг. 1.4.5.16) връща различни компоненти в зависимост от това, какво иска потребителя. Методът разчита на променлива от *state-a* на програмата, чиято стойност указва кой компонент ще се визуализира. Предоставени са методи, които променят тази променлива и се подават на компоненти, които наследяват главния. От там те могат да бъдат извикани и съответно да променят тази променлива, което води и до промяна на видимата част от екрана.

## 1.5. Реализация - избор на програмен език и библиотеки

### 1.5.1. JavaScript

JavaScript® (или JS) е лек, интерпретиран, обектно-ориентиран език с first-class функции и е известен най-вече като скриптов език за уеб страници, но се използва и в много не-браузърни среди. Той е прототипно-базиран, мулти-парадигматичен скриптов език, който е динамичен и поддържа обектно-ориентирани, императивни и функционални стилове на програмиране. JavaScript се изпълнява от клиентската страна на мрежата, което може да се използва за проектиране/програмиране на начина, по който уеб страниците се държат при настъпване на събитие. JavaScript е мощен скриптов език, широко използван за контролиране на поведението на уеб страниците. JavaScript е динамичен скриптов език, който поддържа прототипно изграждане на обект. JS е най-често използвания език за разработване на front-end на едно уеб (web) приложение (в комбинация с HTML и CSS), което ни дава една голяма база, от която можем да черпим информация за възникнали проблеми.

#### 1.5.1.1. React.js

React е декларативна, ефективна и гъвкава JavaScript програмна рамка за изграждане на потребителски интерфейси. Тя се поддържа от Facebook, Instagram и общност от отделни разработчици и корпорации. React позволява да се създават големи уеб приложения, които използват данни и могат да се променят във времето без презареждане на страницата. Тя има за цел главно да осигури скорост, простота и мащабируемост. React е само потребителският интерфейс в приложението. Това съответства на View в шаблона Model-View- Controller (MVC) и може да се използва в комбинация с други библиотеки. React.js е една от най-използваните библиотеки за потребителски интерфейс, поради което обществото, което я разработва е голямо. При

възникнал проблем има подробна и пълна информация, която може да се намери в основните източници StackOverflow и Reactiflux chat.

#### **1.5.1.1.1. Material-UI (<http://www.material-ui.com/>)**

Material-ui е npm пакет, собственост на Google, който предоставя лесна възможност за написване на потребителски интерфейс чрез вече готви react компоненти. Съставните компоненти на material-ui работят в изолация. Те са самоподдържащи се и е нужно да се добавят към компонента, в който се използват. Сами добавят стиловете, които използват, когато се визуализират. Те не разчитат на глобални стилове като normalize.css, въпреки че material-ui предоставя опционален компонент за рестартиране. Предоставена е голяма гама от функционални компоненти, с които лесно може да се разработи функционален и красив потребителски интерфейс.

#### **1.5.1.1.2. react-d3-radar (<https://www.npmjs.com/package/react-d3-radar>)**

Това е компонент за радарна диаграма, който визуализира множество набори от данни. Покриване с курсора и визуализиране на допълнителни данни се поддържа чрез d3-voronoi.

#### **1.5.1.1.3. react-qr-component**

(<https://www.npmjs.com/package/react-qr-component>)

React компонент, който генерира QR кодове.

### **1.5.2. Java**

Java е обектно ориентиран програмен език. Той е един от най- използваните програмни езици, поради което има много решения на проблеми и разнообразни библиотеки. Голямото общество, което програмира на Java ни предлага възможност за решаване на определен проблем в сайтове като GitHub и StackOverflow. Java лесно се

интегрира към различни платформи, защото се нуждае само от JVM - Java virtual machine. Java е език от високо ниво, който премахва необходимостта от собственоръчно заделяне и освобождаване на памет. Наличието на множество библиотеки и досегашния опит предопределиха изборът на Java.

#### **1.5.2.1. Jersey**

Jersey е програмна рамка за разработване на RESTful уеб приложения, изпозващи Java. Jersey е повече от JAX-RS имплементация. Тя предоставя свое собствено API (приложно-програмен интерфейс), което надгражда JAX-RS инструментариума с допълнителни функции и помощни програми, които опростяват разработката на приложенията. Jersey също така предлага и многобройни SPI разширения, така че разработчиците могат да я разширят, за да отговарят най-добре на техните потребности.

#### **1.5.2.2. Hibernate**

Hibernate ORM е обектно релационен преобразовател, който позволява съпоставяне на object-relational връзка за програмния език на Java. Той осигурява програмна рамка за преобразуване на обектно-ориентиран модел (Java class) в релационна база данни. Основната функция на Hibernate е преобразуване на Java класове до таблици (SQL Tables) и преобразуване на Java типове данни до SQL типове.

#### **1.5.2.3. Guice persist**

Guice Persist предоставя абстракция за работа с информация и данни в Guice приложения. Той работи в нормално компютърно или сървърно Java приложение, в обикновена Servlet среда или дори в контейнер с Java EE. Когато се създава нов обект с анотация `@Inject`, не е необходима допълнителната грижа за всички негови зависимости.



#### 1.5.2.4. Jackson

Jackson parser е библиотека, която позволява превръщането на JSON в Java обект, както и обратното. С Jackson се обезпечава необходимостта от преработката на новопостъпили и изпратени данни.

#### 1.5.2.5. Apache Shiro™

Apache Shiro™ е мощна и лесна за използване Java програмна рамка за защита, която извършва удостоверяване, оторизация, криптография и управление на сесиите. С Shiro лесно могат да се защитят всякакви приложения - от най-малки мобилни приложения до най-големите уеб и корпоративни приложения.

#### 1.5.2.6. OpenCV и ZXing

OpenCV и ZXing са библиотеки, които се използват в настоящата обработка на бланките. ZXing е библиотека с отворен код, имплементирана на Java, която предоставя възможност за разчитане на 1D/2D баркодове. Приложението я използва, за да разчете стойностите в баркода на всяка бланка. OpenCV (Open Source Computer Vision Library) има интерфейси за C++, C, Python, Java и поддържа Windows, Linux, Mac OS, iOS и Android операционни системи. OpenCV е проектирана за изчислителна ефективност и със силен фокус върху приложения в реално време. Написана в оптимизиран C / C++ език, библиотеката може да се възползва от многоядрената обработка. Приложението използва OpenCV за обработка на бланките като намиране на кръгове и манипулации за постигане на необходим ефект.

### 1.5.3. PostgreSQL

PostgreSQL е обект-релационна база данни с отворен код. Тя има повече от 15 години активно развитие и доказана архитектура, която ѝ е спечелила добра репутация за надеждност, целостта на данните и коректността. Базата работи на всички основни операционни системи, включително Linux, UNIX (AIX, BSD, HP-UX, MacOS, Solaris) и

Windows. Тя е напълно съвместим с ACID, има пълна поддръжка за foreign keys, joins, views, triggers и stored procedures.

## 1.6. Описание на приложението

### 1.6.1. Нужни зависимости

За да може приложението да заработи, то се нуждае от следните зависимости:

- *Nodejs* пакет (включващ npm) - <https://nodejs.org/en/>
- *JDK* (Java SE development kit) - задаване на *JAVA\_HOME* променлива в *\$PATH* (инсталира се от сайта на oracle - <http://www.oracle.com/technetwork/java/javase/downloads/index.html>)
- *Maven* (нуждае се от *JDK*) - задаване на *MAVEN\_HOME* променлива в *\$PATH* (инсталира се от сайта на apache maven <https://maven.apache.org/download.cgi>)
- *Heroku* (за стартиране на локален *tomcat server*) - <https://devcenter.heroku.com/articles/heroku-cli>
- *PostgreSQL* (версия >= 9.4) - <https://www.postgresql.org/download/>

Забележка: В зависимост от операционната система, тези зависимости се инсталират по различни начини. Важно е да ги има и да са точни версии и пакети. В противен случай приложението може да не работи правилно.

Задължително се добавя системна променлива -

`DATABASE_URL=postgres://<user>:<pass>@localhost:<server_port>/<dbname>`

*User* и *pass* са входните данни на потребителя направени прдварително, *server\_port* е портът, на който работи *postgres service-a* и *dbname* е името на предварително направена база. Системната променлива *PORT* се добавя (под *Linux* дистрибуции), защото когато приложението бъде качено на сървър, то се пуска на порта запазен в конфигурациите му. Когато приложението бъде пуснато локално в *procfile-a* пише, на кой порт ще работи приловението. Там може да се постави *\$PORT* ако тази системна променлива съществува (под *Windows* на мястото на *\$PORT* се записва 8080).

В някои дистрибуции са нужни и допълнителни конфигурации.

## 1.6.2. Стартиране на приложението

### 1.6.2.1. Стартиране на клиентска страна - Front-End

За стартирането на клиентската страна е нужно конзолно да се влезе в потдиректорията *automatic-test-evaluation*, която се намира в главната директория на проекта. След като директория бъде отворена, то при първо стартиране се пише командата ***npm install***, което ще инсталира нужните зависимости специфицирани в *package.json* файла. След това се изпълнява командата ***npm start***, което ще стартира сървър *CAMO* с *front-end-a*.

### 1.6.2.2. Стартиране на цялото приложение - Front-End + Back-End

За стартиране на цялото приложение е нужно да се навигира конзолно до директорията на проекта и да се влезе в нея. След това се изпълнява командата ***mvn clean install***, което създава нов *build* на проекта. След това се стартира командата ***heroku local web***, която ще стартира *tomcat* сървър със *build-натия* проект локално.

В *README.md* има допълнителна информация за това как се стартира приложението.

## 1.6.3. Ръководство на потребителя

Когато нов потребител влезе в приложението, пред него има меню - фигура 1.6.3.1.

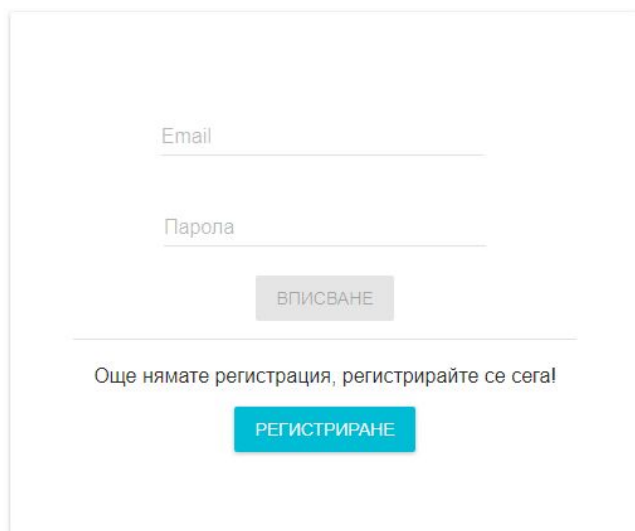


фиг. 1.6.3.1

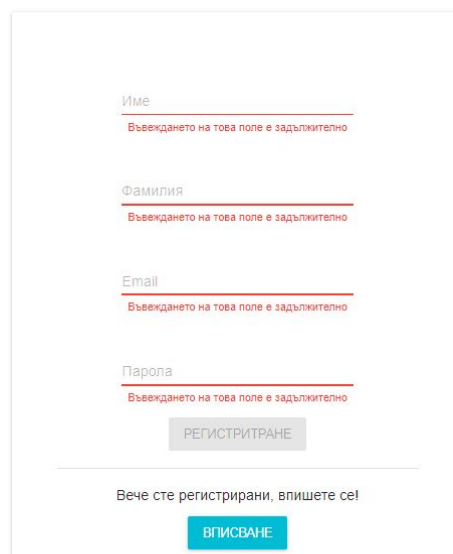
Менюто предоставя на потребителя набор от възможности: *начало*, *качване на снимка*, *резултати*, *генериране на бланка* и *вписване / профил*.

### 1.6.3.1. Създаване на нов профил или влизане в съществуващ

Когато влезе потребител, който няма регистрация или не е вписан, той може създаде нов профил или да влезе във вече съществуващ като натисне компонента “Вписване” от главното меню (фиг. 1.6.3.1).



фиг. 1.6.3.2



фиг. 1.6.3.3

За да се регистрира (фиг. 1.6.3.3), потребителят трябва да попълни следните полета:

- Име (до 30 (тридесет) символа и не трябва да съдържа интервал)
- Фамилия (до 30 (тридесет) символа и не трябва да съдържа интервал)
- Email (до 50 (петдесет) символа и трябва да е в правилен формат за имейл. Имейлът се използва за следващо вписване)
- Парола (трябва да е над 8 (осем) символа и да съдържа главна и малка буква)

Компонентът предоставя избор дали потребителя да се регистрира или да влезе (фиг. 1.6.3.2) в своя профил.

### 1.6.3.2. Създаване на универсална бланка

На потребителя се предоставя възможност да създаде универсална бланка като не е нужно да има регистрация и да е вписан. Когато натисне компонента озаглавен

“Генериране на бланка” от главното меню (фиг. 1.6.3.1), пред него се визуализира следния набор от възможности (фиг. 1.6.3.4), които структурират бланката.

1 Изберете какви атрибути да има теста

2 Структура на теста

НАЗАД НАПРЕД

Изберете колко опции да имат отговорите по подразбиране

4

Поле за име ☒ Поле за група ☒  
Поле за номер ☒ Поле за клас ☒

Чрез плъзгане изберете от колко въпроса да се състои бланката (минимален брой - 1, максимален брой - 60)

Избран брой: 10

НАЗАД НАПРЕД

фиг. 1.6.3.4

Той може да избере от колко въпроса ще се състои бланката, по колко възможните отговора ще имат тези въпроси първоначално (първоначалните зададени стойности са 10 (десет) въпроса с по 4 (четири) възможни отговора). Също така може да избере дали в бланката да има полета за име, номер, клас и група. След като приключи с тази стъпка от конфигурацията, той трябва да натисне бутона “Напред”, който ще го отведе до следващата стъпка от конфигурацията.

1 Изберете какви атрибути да има теста

2 Структура на теста

НАЗАД ЗАВЪРШВАНЕ

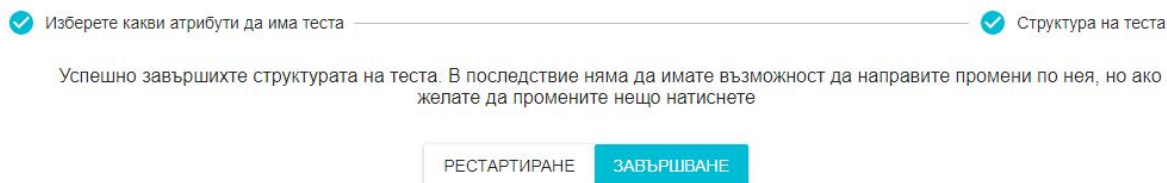
1. ☐ ☐ ☐ ☐ 4

2. ☐ ☐ ☐ ☐ 4

фиг. 1.6.3.5

В тази стъпка (фиг. 1.6.3.5) той може да променя стойността на броя възможни отговори за всеки въпрос като може да зададе стойност от 2 (два) до 7 (седем). При

натискане на бутона “Назад” той ще се върне на предната стъпка като отново ще има възможност да промени нещо, което е забравил или иска да промени. При натискане на бутона “Напред” на екрана на потребителя се появява последната стъпка (фиг. 1.6.3.6) от структурирането на бланката.



фиг. 1.6.3.6

На тази стъпка потребителят трябва да даде своето съгласие, ако е завършил структурата и не иска да прави повече промени. Тъй като в следствие потребителят няма да има възможност отново да променя структурата на бланката, то му се предоставя възможност с натискането на бутона “Рестартиране” да се върне назад и да направи нужните промени.

След натискане на бутона “Завършване” на екрана на потребителя се визуализира завършената бланка, която има следния вид - фигура 1.6.3.7.

this blank is created with automatic test evaluation

Име: \_\_\_\_\_

Клас: \_\_\_\_\_

Номер: \_\_\_\_\_

Група: \_\_\_\_\_



---

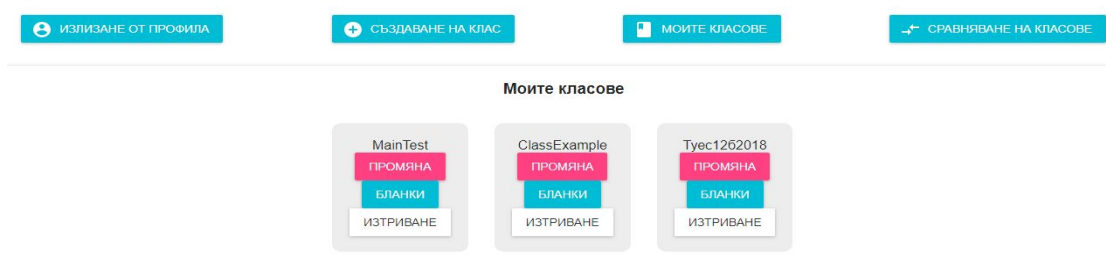
1. ☐ ☐ ☐ ☐ ☐
2. ☐ ☐ ☐ ☐ ☐
3. ☐ ☐ ☐ ☐ ☐
4. ☐ ☐ ☐ ☐ ☐
5. ☐ ☐ ☐ ☐ ☐
6. ☐ ☐ ☐ ☐ ☐
7. ☐ ☐ ☐ ☐ ☐
8. ☐ ☐ ☐ ☐ ☐
9. ☐ ☐ ☐ ☐ ☐
10. ☐ ☐ ☐ ☐ ☐

фиг. 1.6.3.7

За да принтира изготвената бланка, потребителят трябва да натисне клавишната комбинация *Ctrl + P* и на екрана му ще се появи меню, предоставено от *browser-a*, с което той може да направи допълнителни конфигурации за принтирането (*например брой копия*).

### 1.6.3.3. Какви предимства предоставя наличието на профил и как да се използват

Когато един потребител е създал своя профил и се е вписал в него, то на главното меню (фиг. 1.6.3.1) вече се е появил компонент, озаглавен “Профил”. Когато потребителят го натисне, пред него се визуализира следния прозорец -



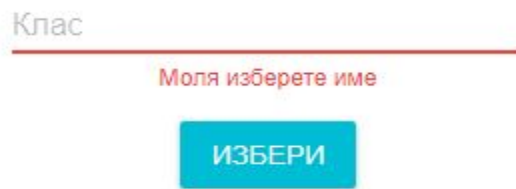
фиг. 1.6.3.8

фигура 1.6.3.8.

При първоначалното зареждане на този компонент, в него е визуализирано подменю, което има следните възможности: *излизане от профила*, *създаване на клас*, *моите класове* и *сравняване на класове*, а под тях са визуализирани вече създадените класове на потребителя, подредени в каталог по три на ред.

При натискане на бутона *излизане от профила*, потребителят бива изведен до началния екран - компонента за влизане.

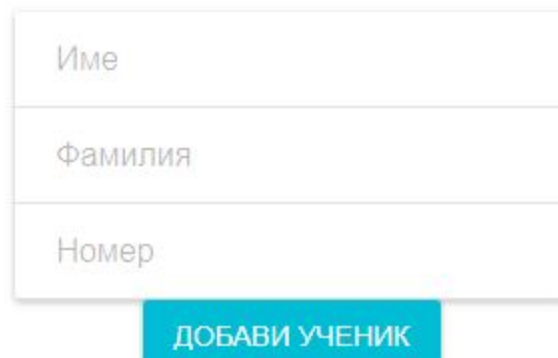
При натискане на бутона за *създаване на клас* на екрана на потребителя излиза следното поле (фиг. 1.6.3.9), в което потребителят трябва да напише как иска да се казва неговия клас.



фиг. 1.6.3.9

Забележка: *името на класа трябва да е уникално спрямо другите класове на потребителя.*

След като потребителят избере име и натисне бутона *избери*, пред него се открива възможност да добавя нови ученици (фиг. 1.6.3.10) към класа, а от лявата страна се визуализират всички ученици, които са част от класа.

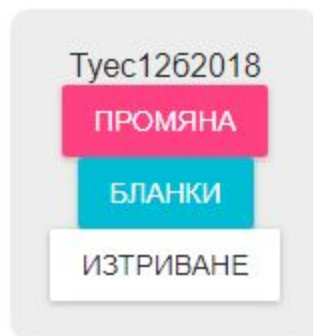


фиг. 1.6.3.10

При натискане на бутона *моите класове* от подменюто на екрана на потребителя се представя каталога с неговите класове.

Пример за това как изглежда един елемент от каталога с класовете - фигура 1.6.3.11.





фиг. 1.6.3.11

Всеки клас предоставя следните възможности:

- *промяна* - предоставя възможност за промяна на класа - добавяне на нови ученици, промяна на стари или изтриването им.
- *бланки* - на екрана на потребителя се появява екран с текущите бланки за този клас, възможност да ги принтира и да види резултатите за тази бланка, както и възможност да създава нови бланки.
- *изтриване* - предоставя възможност на потребителя да изтрие избран от него клас.

#### 1.6.3.4. Създаване на бланка за определен клас (предоставя възможност за автоматично проверяване)

Създаването на бланка за нов клас става посредством натискане на бутона *бланки* (върху някой клас от каталога (фиг. 1.6.3.7)) и след това под списъка от текущите бланки излиза панел, който приканва потребителят да избере име на бъдещата бланка. След натискане на бутона *избиране*, пред потребителят се визуализира следния набор от възможности, с които той да структурира бланката си:

1. Избира какви полета да съдържа бланката - фигура 1.6.3.12

1 Изберете какви атрибути да има теста

2 Структура на теста

НАЗАД

НАПРЕД

Изберете колко опции да имат отговорите по подразбиране

4 ▾

Имена на учениците от списъка

Номер на учениците от списъка

Чрез плъзгане изберете от колко въпроса да се състои бланката (минимален брой - 1, максимален брой - 60)

---

Избран брой: 10

НАЗАД

НАПРЕД

фиг. 1.6.3.12

2. Потребителят трябва да избере кой е верният отговор за всеки въпрос (първоначално верният отговор на всеки въпрос е първият) като има възможност да промени броя на възможните отговори за всеки отговор (фиг. 1.6.3.13).

1.

☒
☐
☐
☐

4 ▾

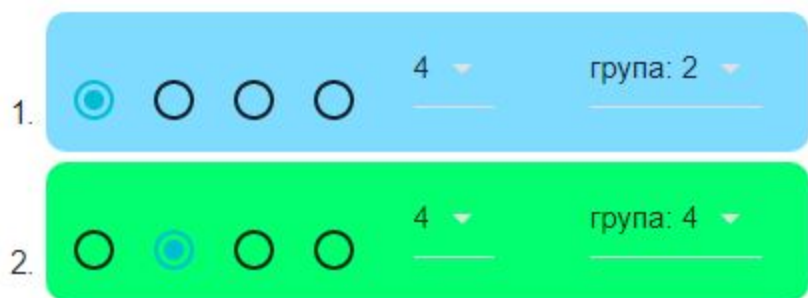
2.

☐
☒
☐
☐

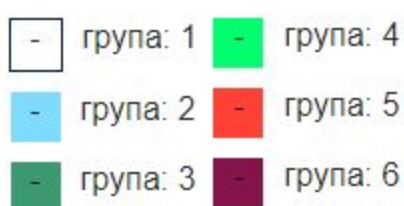
4 ▾

фиг. 1.6.3.13

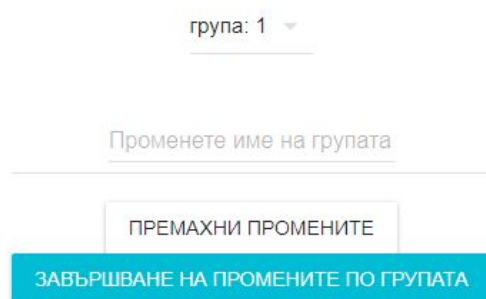
Той може да групира различни въпроси посредством спускащо се меню, което има всеки въпрос. Промяната на група на даден въпрос ще смени подсветката му (фиг. 1.6.3.14) като всички въпроси от една група са с еднаква подсветка.



фиг. 1.6.3.14



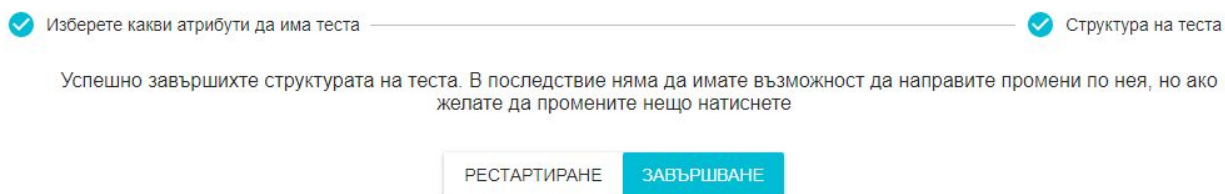
фиг. 1.6.3.15



фиг. 1.6.3.16

Потребителят може да променя имената на групите (фиг. 1.6.3.16). Най-отгоре на тази стъпка от конфигурацията на структурата има легенда (фиг. 1.6.3.15) на групите (група и съответната ѝ подсветка) и полето за промяна (фиг. 1.6.3.16) на името на дадена група.

3. След като потребителят завърши промените по структурата и потвърди, че вече е направил всички нужни промени по структурата, пред него се визуализира прозорец, от който той може да се върне в началото на структурирането или да завърши подготовката - фигура 1.6.3.17.



фиг. 1.6.3.17

След като потребителят натисне бутона завършване, пред него се визуализират бланки за всеки ученик от класа с неговото име и номер (фиг. 1.6.3.18).

this blank is created with automatic test evaluation

Име: Ivan Dimitrov  
Номер: 2



---

1. ☐ ☐ ☐ ☐ ☐

2. ☐ ☐ ☐ ☐ ☐

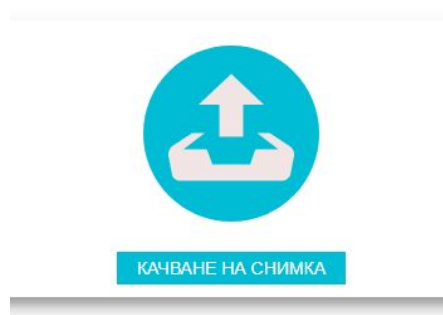
3. ☐ ☐ ☐ ☐ ☐

фиг. 1.6.3.18

Потребителят трябва да натисне клавишната комбинация *Ctrl + P*, за да принтира бланките за всеки ученик (Пример: Ако в класа има 3-ма ученика, то когато се направи нова бланка за тях и трябва да се принтира, ще излязат 3 (три) бланки, като на всяка от тях ще има имената и номерата на учениците).

### 1.6.3.5. Качване на снимка и автоматично проверяване

Когато бланката бъде попълнена и учителят иска да се възползва от възможността за автоматично проверяване, той трябва да избере опцията качване на снимка от главното меню (фиг. 1.6.3.1). Пред него ще се покаже следния прозорец - фигура 1.6.3.19.



#### фиг. 1.6.3.19

Той трябва да натисне кръглия бутон и да избере съответната снимка на бланката, която иска да бъде проверена. След като е избрал снимката, трябва да натисне бутона *качване на снимка* и на екрана ще му бъде върнат резултат дали бланката е проверена успешно или има някакъв проблем в обработката ѝ.

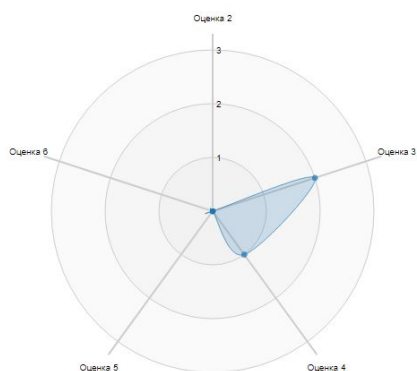
*Забележка:* За да може бланката да бъде обработена, трябва:

- снимката да бъде снимана хоризонтално
- да не бъде размазана
- бланката да е снимана в цял размер
- да се чете QR кода
- да няма драстични светлосенки
- не трябва да има тъмни петна върху самите отговори

#### 1.6.3.6. Проверяване на резултатите и добавяне на материали

Всеки потребител, който притежава профил и собствени класове, може да провери резултатите от всяка бланка. Той трябва да избере от *главно меню* (фиг. 1.6.3.1) -> *профил*, след което в подменюто да избере *моите класове* и от каталога с класове да натисне бутона *бланки* на определен клас.

Пред него ще се появят набор от възможности за бланките, като една от тях е бутонът *резултати*. След натискането му, пред потребителят се визуализират настоящите резултати от бланката. В лявата половина на екрана се появява диаграма (фиг. 1.6.3.20), на която са показани общия брой оценки на класа и тяхното разпределение, а от другата страна - оценките на всички ученици от класа (фиг. 1.6.3.21). Под тях се визуализират материалите на бланката (фиг. 1.6.3.22).



фиг. 1.6.3.20

Pesho	3
Ivan	4
gogo	3

фиг. 1.6.3.21

Readme.txt	СВАЛЯНЕ
Readme.txt	СВАЛЯНЕ
Readme.txt	СВАЛЯНЕ

фиг. 1.6.3.22

Собственика на бланката може да качи материали за нея като това става посредством *бутона качване на материали*, който всяка бланка притежава

Всеки потребител може провери резултатите на всяка бланка без да има създаден профил. Това се случва посредством натискане на опцията *резултати* от главното меню (фиг. 1.6.3.1). Пред него ще се визуализира списък на всички съществуващи бланки, както и възможност да види резултати за всяка една от тях, ако има такива. При натискане на бутона *резултати* пред него ще бъде представена диаграма на общия резултат, собствените резултати на всеки ученик, попълвал бланката (фиг. 1.6.3.21), както и материалите за бланката качени от собственика ѝ.



## 1.7 Заключение

Използването на информационните и комуникационни технологии във всички области на живота гарантира в максимална степен превръщането на иновативни идеи в нови продукти и услуги, създаващи устойчив и приобщаващ растеж, качествени работни места и помагат за разрешаването на възникналите предизвикателства пред обществото. За постигане на желания успех усилията трябва да са насочени към: иновации; цифровизиране на обществото; образование, обучение и учене през целия живот [Европа 2020, 10, с. 14-15].

Използването на методи, позволяващи количествено измерване на знанията водят до по-голяма обективност в крайния резултат. Дидактичните тестове като такива притежават и еталон, пълен и правилен начин за изпълнение на дадена дейност по всички операции с посочване на най-съществените, които отразяват същността и съдържанието на проверката (Беспалко, 1982). Чрез пооперационно съпоставяне на отговора на тестираните с еталона се стига до извода за качеството на изпълнения тест, т.е. елиминира се интервала от колебанията в оценката от различни преподаватели на една и съща работа. Освен това самите обучаеми могат да ги използват като техника на самодиагностика. Колкото по-добре те си представят очакваните резултати, толкова по-високо ще бъде равнището на тяхната познавателна дейност.

Разработеното приложение за полуавтоматично оценяване на учебните тестове в настоящото дипломно изследване се отнася за задачи с множествени отговори.

Предимства на тестовите задачи с избираем отговор:

- многовариантност – могат да измерват широк спектър от постижения – от познаване на факти до анализ и интерпретация на информация, правене на изводи, решаване на проблеми и умения за вземане на решения;



- ефективност – позволяват да се покрие напълно учебното съдържание и да се постигне съответствие с целите на измерването;
- всяка тестова задача проверява дадена цел;
- контролиране на трудността – нивото на трудност на теста може да се променя (увеличава или намалява) чрез промяна на степента на близост
- на отделните алтернативи на задачата;
- тестираният оперира с напълно определена структурирана ситуация;
- точно и бързо оценяване – базира се на предварителна експертна оценка;
- позитивна мотивация;
- надеждност - точността на тестовите балове на съдържателно валидни тестове позволява резултатите да се обобщават и използват за прогнозиране на постиженията на същата група лица върху други подобни тестове или на постиженията на подобна група лица върху същия тест;
- диагностициране – изборът на дистракторите (неправилните отговори) може да бъде използван за анализ на типичните грешки както на отделни изпитвани лица, така и на групи;
- намаляване на вероятността за налучкване на верния отговор;
- независими са от такива външни фактори като правопис, краснопис и т.н.;
- сравнително бърза проверка;
- оценката е обективна и не зависи от проверяващия;
- осигуряват съпоставимост на резултатите от обучението както между изследваните лица, така и при едно и също изследвано лице в различни периоди от време;
- възможност за статистически анализ – подобряване на техните качества.

Недостатъци на тестовите задачи с избираем отговор:

- трудоемкост – писането на тестови задачи с избираем отговор е трудна задача, която отнема много време;
- повечето от тях измерват фактически знания без съобразяване с таксономията на когнитивните цели;

- стимулират трупане на знания за факти;
- позволяват налучкване на верния отговор, т. е. някои от проверяваните могат да узнаят верния отговор сред дистракторите, но не и да го конструират самостоятелно;
- количеството на задачите от този тип в един тест е относително голямо;
- не измерват способности за излагане на мнение, прояви на находчивост и творчество.

## ЛИТЕРАТУРА

[Европа 2020, 10] Стратегия за интелигентен, устойчив и приобщаващ растеж – Европа 2020 (2010), Брюксел, 3.3.2010, [http://ec.europa.eu/eu2020/pdf/1\\_BG\\_ACT\\_part1\\_v1.pdf](http://ec.europa.eu/eu2020/pdf/1_BG_ACT_part1_v1.pdf).

Фрагменти от код и библиотеки, които трябва да се цитират:

- <https://shiro.apache.org/10-minute-tutorial.html>
- <http://answers.opencv.org/question/24578/select-only-gray-pixels-in-color-image/>
- <https://stackoverflow.com/questions/415953/how-can-i-generate-an-md5-hash>
- [https://www.w3schools.com/js/js\\_cookies.asp](https://www.w3schools.com/js/js_cookies.asp)
- <https://designshack.net/articles/css/5-simple-and-practical-css-list-styles-you-can-copy-and-paste/>
- <https://tympanus.net/codrops/2015/09/15/styling-customizing-file-inputs-smart-way/>
- <https://stackoverflow.com/questions/46155/how-to-validate-email-address-in-javascript>
- Цитат на модул за *QR* код Copyright (c) 2004-2010 by Internet Systems Consortium, Inc. ("ISC") Copyright (c) 1995-2003 by Internet Software Consortium

# Съдържание

<b>1.1. Цели</b>	<b>3</b>
1.1.1. Изисквания към приложението	3
1.1.2. Преглед на съществуващи решения	4
1.1.2.1. Eduleaf ( <a href="https://eduleaf.com/#">https://eduleaf.com/#</a> )	4
1.1.2.2. Exam Reader ( <a href="https://bebyaz.com/ExamReader">https://bebyaz.com/ExamReader</a> )	4
1.1.2.3. Zipgrade ( <a href="https://www.zipgrade.com">https://www.zipgrade.com</a> )	5
<b>1.2. Основни етапи в реализирането на проекта</b>	<b>6</b>
<b>1.3. Ниво на сложност на проекта - основни проблеми при реализацията на поставените цели</b>	<b>7</b>
<b>1.4. Логическо и функционално описание на решението</b>	<b>8</b>
1.4.1. Структура на приложно-програмния интерфейс	8
1.4.1.1. GET заявки - използват се, за да се получи дадена информация	8
1.4.1.2. POST заявки - използват се, за да се създаде нов ресурс	9
1.4.2. Диаграма на базата данни	10
1.4.3. Структура на базата данни	10
1.4.5.1. Схема на таблица “users”	11
1.4.5.2. Схема на таблица “classes”	11
1.4.5.3. Схема на таблица “classinformation”	12
1.4.5.4. Схема на таблица “students”	13
1.4.5.5. Схема на таблица “blanks”	13
1.4.5.6. Схема на таблица “answers”	14
1.4.5.7. Схема на таблица “files”	15
1.4.5.8. Схема на таблица “results”	15
1.4.5.9. Схема на таблица “resultanswers”	16
1.4.4. Реализация на сървърна страна - Back-End	17
1.4.4.1. Създаване на модели - entities	17
User.java	17
1.4.4.2. Връзка с базата и манипулация на данните	21
1.4.4.3. REST контролери - крайни точки	24
1.4.4.4. Сервиси и бизнес логика (Services)	26
1.4.4.5. Реализация на сигурност	27
1.4.4.6. Реализация на алгоритъм за проверяване на бланки	29

1.4.5. Реализация на клиентска страна - Front-End	34
1.4.5.1. React компонент	34
1.4.5.2. Node package manager(npm) - зависимости на клиентската страна	35
1.4.5.3. Fetch приложно-програмен интерфейс(API)	36
1.4.5.4. Използване на външни компоненти	37
1.4.5.5. Реализация на главно меню	37
1.4.5.6. Реализация на вписване и създаване на профил	38
1.4.5.7. Създаване на бланка	40
1.4.5.8. Преференции и управление на профил	44
<b>1.5. Реализация - избор на програмен език и библиотеки</b>	<b>46</b>
1.5.1. JavaScript	46
1.5.1.1. React.js	46
1.5.1.1.1. Material-UI ( <a href="http://www.material-ui.com/">http://www.material-ui.com/</a> )	47
1.5.1.1.2. react-d3-radar ( <a href="https://www.npmjs.com/package/react-d3-radar">https://www.npmjs.com/package/react-d3-radar</a> )	47
1.5.1.1.3. react-qr-component ( <a href="https://www.npmjs.com/package/react-qr-component">https://www.npmjs.com/package/react-qr-component</a> )	47
1.5.2. Java	47
1.5.2.1. Jersey	48
1.5.2.2. Hibernate	48
1.5.2.3. Guice persist	48
1.5.2.4. Jackson	48
1.5.2.5. Apache Shiro™	49
1.5.2.6. OpenCV и ZXing	49
1.5.3. PostgreSQL	49
<b>1.6. Описание на приложението</b>	<b>50</b>
1.6.1. Нужни зависимости	50
1.6.2. Стартиране на приложението	51
1.6.2.1. Стартиране на клиентска страна - Front-End	51
1.6.2.2. Стартиране на цялото приложение - Front-End + Back-End	51
1.6.3. Ръководство на потребителя	51
1.6.3.1. Създаване на нов профил или влизане в съществуващ	51
1.6.3.2. Създаване на универсална бланка	52
1.6.3.3. Какви предимства предоставя наличието на профил и как да се използват	55
1.6.3.4. Създаване на бланка за определен клас (предоставя възможност за автоматично проверяване)	57
1.6.3.5. Качване на снимка и автоматично проверяване	60
1.6.3.6. Проверяване на резултатите и добавяне на материали	61

