Съдържание

Увод

Съществува голям потенциал на пазара на труда, който е малко използван в днешно време. И този потенциал е младата работна ръка. Много млади хора търсят начин да изкарат пари, но нямат възможност да се ангажират с работа на пълен работен ден. След провеждане на запитване към много наши връстници, открихме, че желанието наистина съществува и много биха работили, дори и за по-малко пари. В същото време съществуват и много хора, които биха наели работна ръка за всякаква еднократна битова работа.

Възможностите са безкрайни - събиране на боклук, подреждане, чистене, помощ по двора на къща, преместване на различни видове товари. Големият плюс за работодателите е евтината работна ръка. Както голям процент от хората започнаха да си търсят място за преспиване в AirBnB поради възможността да си намериш по-евтини нощувки, така и съществува същата възможност, когато става въпрос за неспециализирана битова работа. Плюсът за младите работници е бързите пари. Според мен е добре един млад човек, в рамките на няколко дена, да може да изкара нужните му пари, които най-често са неголяма сума, но такава възможност все още не съществува. Около 20 процента от населението е от млади хора на възраст между 16 и 24 години. Повечето са ученици или студенти, които не могат да си позволят да работят ежедневно на пълен работен ден. Това е и идеята на нашето приложение. Желаем да бъдем нужната връзка между младите, амбициозни работници и работодателите, които да могат да се възползват от по-евтината работна ръка. По този начин ще стимулираме младото поколение да стане част от работещото общество от ранна възраст и едновременно с това ще спестим пари за работодателите.

Целта на дипломната работа е разработването на основните backend фун-

кционалности подсигуряващи приложение за създаване на обяви за работа чрез потребителски акаунти. Всеки потребител трябва да има възможността да заяви желание да извърши определената работа, а от своя страна всеки работодател да може да оценява извършената работа.

Дипломната работа трябва да бъде разработена използвайки TDD методология, а API-то да бъде RESTful.

Глава 1

Проучване

На пазара в момента съществува огромен потенциал за този тип приложения, тъй като бизнес моделът, който искаме да постигнем, е доказано ефективен. Пример за това е Airbnb, който в рамките на няколко години стана едно от най-успешните приложения, набавяйки си милиони потребители от целия свят.

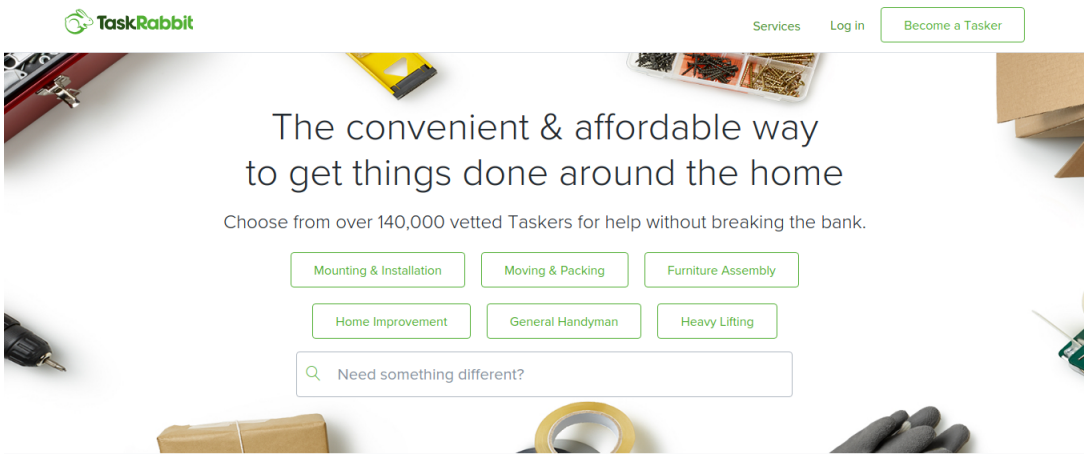
Разглеждайки съществуващи такива продукти на пазара откриваме, че има недостиг на възможности човек да намери битова работа, чрез която да си изкара пари. Всички съществуващи вече продукти страдат от лош бизнес модел и несигурна за потребителя система.

1.1 Съществуващи продукти с подобни цели

1.1.1 TaskRabbit

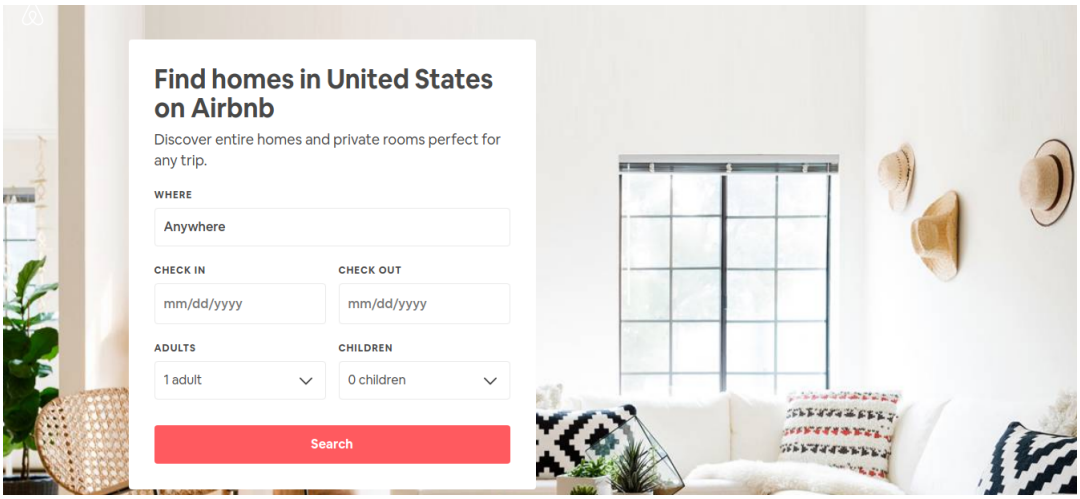
TaskRabbit е уеб и мобилно приложение, което свързва фрийланс работници с локални работодатели, позволяващо на хората да намират незабавна помощ с битова работа като готвене, местене и доставки. TaskRabbit е най-известното приложение в тази сфера, но въпреки това страда от доста недостатъци.

Услугата не е достъпна за България и регистрирането е сложен процес, който изисква прекалено много време и струва пари. Приложението също не е достатъчно гъвкаво за различните типове работа.



Фигура 1.1: TaskRabbit - Титулна страница

1.1.2 Airbnb



Фигура 1.2: Airbnb - Титулна страница

Airbnb е услуга, която предлага наемането и даването под наем на апартаменти. Специалното за тяхната услуга е, че всеки може да даде под наем своя стая или апартамент, което позволява за повече гъвкавост и от двете страни на услугата. За наемащите спестява пари, тъй като цените на апартаменти и стаи са драстично по-ниски отколкото хотелски такива. За даващите под наем е лесен начин за изкарване на пари. Всеки, който има апартамент или стая, която не използва, може да си осигури доход чрез техните услуги.

Airbnb бързо добива популярност поради иновацията и възможностите, които предоставя на клиентите си.

Причината да съм добавил Airbnb към този списък е поради общото между функционалността на тяхната услуга и нашата идея. Много от идеите на проекта са вдъхновени от техните услуги, тъй като и при тях трябва да се покрие размяната на сумите, осигуряването на връзката между клиентите и създаването на обявите.

1.2 Развойни среди

1.2.1 Eclipse

Фигура 1.3: Eclipse Лого

Eclipse е среда за разработване на софтуер, написана на Java. Използва плъгини за да предостави функционалност на различни езици. Възможно е да се пише на C, Python, както и езици като LaTeX за документиране. Дълго време тази среда бива основа в разработката на Java софтуер, но е изместен от IntelliJ, който разглеждаме във втора глава.

1.2.2 NetBeans

Netbeans е интегрирана среда за разработка, на която може да се пише на Java, JavaScript, PHP, Python, Ruby, Groovy, C, C++, Scala, Clojure и други езици. За удължаване на функционалността на средате, клиент може да инсталира модули създадени от трето лице изработчик. Средата е създадена с цел да предлага интелигентен и бърз начин за писане на код. За жалост обаче

Фигура 1.4: Netbeans Лого

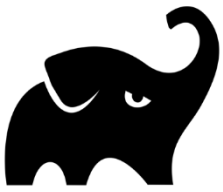
страда от по-малка общност отколкото Eclipse или IntelliJ, което ограничава възможността от плъгини, които могат да бъдат инсталирани. Netbeans има способността да работи на повечето съвременни операционни системи.

1.2.3 Maven

Maven е технология за автоматизиране на проектите като ги сглобява. Използва се основно за Java проекти. Тя покрива две части от компилирането на софтуер. Първата е начинът, по който се компилират и второто е как да се описват зависимостите на проект. Maven динамично изтегля пакети, които са дефинирани във файла му и ги изтегля през интернет, което улеснява процеса за играждане на цялостния продукт, елиминирайки нуждата от ръчно изтегляне на пакети и добавяне към проекта.

1.2.4 Gradle

Gradle е също технология за автоматизиране на проектите, но е по-ефективна, по-бърза и по-лесна за използване от Maven. За този проект именно нея съм предпочел от двете поради изброените причини. Всички фунцкионалности, които Maven предлага са покрити и от Gradle.



Фигура 1.5: Gradle Лого

Глава 2

Основен продукт

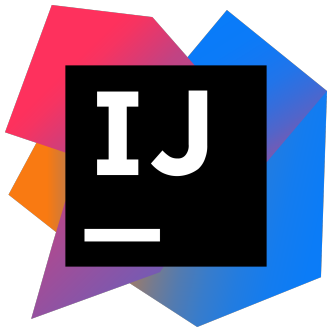
2.1 Изисквания към продукта

Продуктът трябва да осъществява връзката между работници, които желаят еднодневна, бърза работа и работодатели, които имат нужда от помощ с всякакъв вид битова дейност. Начинът, по който това трябва да се осъществи, е чрез обяви, които могат да се качват в приложението от работодателите. Те трябва да съдържат нужната информация за работата като град, предлаганата сума, нужният брой работници и описание на дейността, която трябва да се извърши. Потенциалните работниците трябва да имат възможност да изберат от всички обяви най-желаната и удобна за тях. За да се улесни търсенето, трябва да може да се филтрира чрез всичките полета в обявата, което би улеснило откриването на най-подходящата за тях работа. След като направят своят избор те трябва да заявят желание за участие чрез подаване на заявление. Работодателят ще има възможност да избере предпочитания от него работник или работници. За да може той да направи най-добрия избор, той ще има достъп до обратните връзки от предишни техни работодатели. Приложението дава възможност на работодателя да отбележи кога е стартирало реалното изпълнение на дейността и кои от приетите от него работници са се явили. Ако работник е приет от работодател, но не се е явил на посочените от него място и час за извършване на дейността, това би понижило неговия рейтинг. Работодателят ще има възможността да отбележи края на дейността, след което да остави своята обратна връзка за работата на всеки един от

работниците.

2.2 Използвани технологии

2.2.1 IntelliJ



Фигура 2.1: IntelliJ Лого

IntelliJ е среда за разработка на софтуер с Java създадена от JetBrains. Използвайки тази среда е улеснено създаването на проекта заради характеристики, като завършеност на кода чрез анализ на контекста, навигация на кода, която позволява да се направи клас или декларация в кода директно, да го преработи и да предостави възможности за поправяне на несъответствия чрез предложения. Според статията на Андрей Соинцев[4], IntelliJ е по-добра среда за разработка на Java софтуерни продукти от Eclipse възлагайки следните точки:

- Повече възможности при дебъгване При дебъгване на софтуер, IntelliJ възлага много повече нужна информация и откриването на стойностите на желаните параметри е лесно и интуитивно.
- Интелигентен autocomplete IntelliJ има способността да разбира от контекста, в който кодът се намира и успява да спомага в процеса на писането му. Autocomplete е основната услуга, която разграничава развойна среда от текстов редактор и поради тази причина доброто му осъществяване е от голямо значение.

- Големи възможности при рефакториране Рефакторирането е част от всяка развойна среда, но IntelliJ отново успява да го осъществи по "интелигентен"начин. Средата успява да определи контекста на това, което програмистът се опитва да рефакторира и улеснява работата му значително. Аз лично съм се възползвал от тези услуги множество пъти по време на изработване на дипломнта работа.

2.2.2 Kotlin

Фигура 2.2: Kotlin Лого

Котлин е език за програмиране, който излиза за публично използване през 2012 година, но добива популярност през 2016 с излизането на Котлин 1.0. Езикът е създаден от JetBrains(създателите и на IntelliJ) и е създаден да работи изцяло с и вместо Java, използвайки основните библютеки на предшественика си. Плюсовете за използване на Kotlin вместо Java са много и Магнус Винтер излага част от тях в статията си "Why you should tottaly switch to Kotlin"[5]:

- Съвместимост с Java

  Kotlin позволява съвсем свободното писане на Java код съвместно с него. Той може да замести или да обогати вече написан Java код. Това значи, че всички възможности и архитектури, които Java предоставя, са достъпни и от Kotlin.

- Познат синтаксис

  Kotlin е подобен по синтаксис на повечетео обектно-ориентирани езици. Поради тази причина научването и използването му е тривиално. Kotlin все пак принася и новости като val и var декларторите за променливи, но те също единствено спомагат на работата с езика.

- Интерполация на текст

  Езикът предоставя лесна обработка на променливи в текст.

- Извод за типа променлива

  За разлика от предшественика си, Kotlin има способността сам да прави изводи за типа променливи, които се създават и използват.

- Аргументи по подразбиране

  При избиране на аргументите, езикът предлага задаването на стойността им по подразбиране, елиминрайки нуждата от създаването на множествно методи.

- Именувани аргументи

  Заедно с аргументите по подразбиране се елеминира нуждата от използването на билдер модела.

- When функцията

  Switch case е заменен с много по-удобната when функция.

- Data класът

  В Котлин е възможно създаването на клас от тип data. Това автоматично създава toString(), equals(), hashCode() и copy(), което отстранява кода, който иначе би се налагало да се пише на Java.

- Extension функции

  Котлин има възможността да се пишат extension функции към някой клас. По този наичн би могло да се добавят нужни функции към стари Java библиотеки.

- Обезопасяване от null

  В Java при инстанцирането на обект се задава неговият клас. Но това не осигурява, че стойноста на обекта или променливата няма да бъде null. Котлин отстранява този проблем като прави разлика между обекти, които биха могли да бъдат null и обекти, които не трябва да могат. Това

се прави чрез изписването на името на класа и добавянето на "?" в края на класа. Пр. val age: Int?

2.2.3 Spring

Фигура 2.3: Spring Лого

Spring е технологична рамка, която улеснява свързването между множество компоненти чрез IoC(Inversion of Control) и Dependency Injection. Inversion of Control е принцип в програмирането, който позволява обръщането на последователността на изпълнението на отделни заявки. Използвайки IoC е възможно да се раздели програмния код на отделни компоненти, които изпълняват своите заявки спрямо дефинираната последователност на кода. Това позволява лесното разширяване на програмата, когато има нужда от това.

Dependency Injection е принцип, който се използва за снабдяването на отделните компоненти с техните зависимости. Spring е най-известната и използвана технологична рамка за Java проекти. В контекста на уеб апликации Spring е перфектната технология за създаването на бек енда на Jobche. Използвайки различните технологии, които Spring предоставя, създаването на системата става лесно и интуитивно.

Spring MVC

Spring MVC(Model-View-Controller) е технологична рамка и част от Spring, която предоставя лесно имплементиране на Model-View-Controller модела, чиято

цел е цялостното ограничаване между различните компоненти в програмния код. Това е основната технологична рамка, която поема грижата за заявките, тяхното изпълнение и връщането на съответния отговор.

Spring Data

Spring Data е модел за лесно работене с бази данни. Технологията предоставя множестно компоненти за работене с бази данни, но основният, който се използва в проекта е Spring Data JPA. Spring Data JPA е програмна рамка за създаване и изпълняване на команди върху хранилища създадени на принципа JPA.

Spring Security

Spring Security е технологична рамка за защитаване на проекти създадени със Spring. Тя позволява контрол върху заверяването и упълномощаване на всяка заявка. В проекта е използван за създаване на Basic Authentication на всяка заявка.

Spring Boot

Spring Boot не е технологична рамка, а начин за изработване на Spring проекти. Чрез него лесно можем да стартираме Spring проект, тъй като ни предоставя вграден сървлет. Има множество библиотеки, които Spring Boot предоставя за лесното започване и създаване на софтуер чрез тази технологична рамка. На https://start.spring.io/ може автоматинчо да се създаде Spring Boot проект като имаме опцията да изберем зависимостите, които желаем да използваме.

2.2.4 JUnit5

JUnit е технологична рамка за създаване на тестове върху отделни компоненти и класове, използвайки езици от семейството на Java. В проекта това е основната използвана технология за структурирането и създаването на тестове. Презентацията на Филип Хауер е основната ми мотивация за използването на именно тези технологии за създаването на тестове[3]. В презентацията си

Фигура 2.4: JUnit5 Лого

той представя JUnit5 като най-добрата технологична рамка за писане на тестове с Kotlin. Поради разликите на Kotlin с Java възникват проблеми при използването на по-стари версии на JUnit, тъй като е написан с идеята да се използва в тандем с Java. Но JUnit5 е написан взимайки предвид концепции от Kotlin. Поради тази причина JUnit5 е перфектният избор за създаване на тестове в този проект.

AssertJ

AssertJ е библиотека за създаване на проверки за Java. Библиотеката е най-често използвана и най-добре пригодена за използване заедно с JUnit. В презентацията си Филип Хауер набляга върху огромния брой възможни проверки, които AssertJ предоставя. Именно поради тези причина аз избрах нея за библиотеката за създаване на проверки в проекта.

MockK

MockK е библиотека за създаване на Mock обекти при писането на тестови код. Мокването е важна концепция в света на тестовете и избора на правилна библиотека играе важна роля за цялостното създаване на тестове. Изборът ми на MockK като библиотека за мокване в проекта идва пак от презентацията на Филип Хауер.

2.2.5 Swagger

Swagger е технологична рамка, която помага на програмистта да създаде дизайна и документацията на своя RESTful уеб продукт. Добрата документация на бекенда е важна за лесното й използване от клиентски приложения. Тъй

Фигура 2.5: Swagger Лого

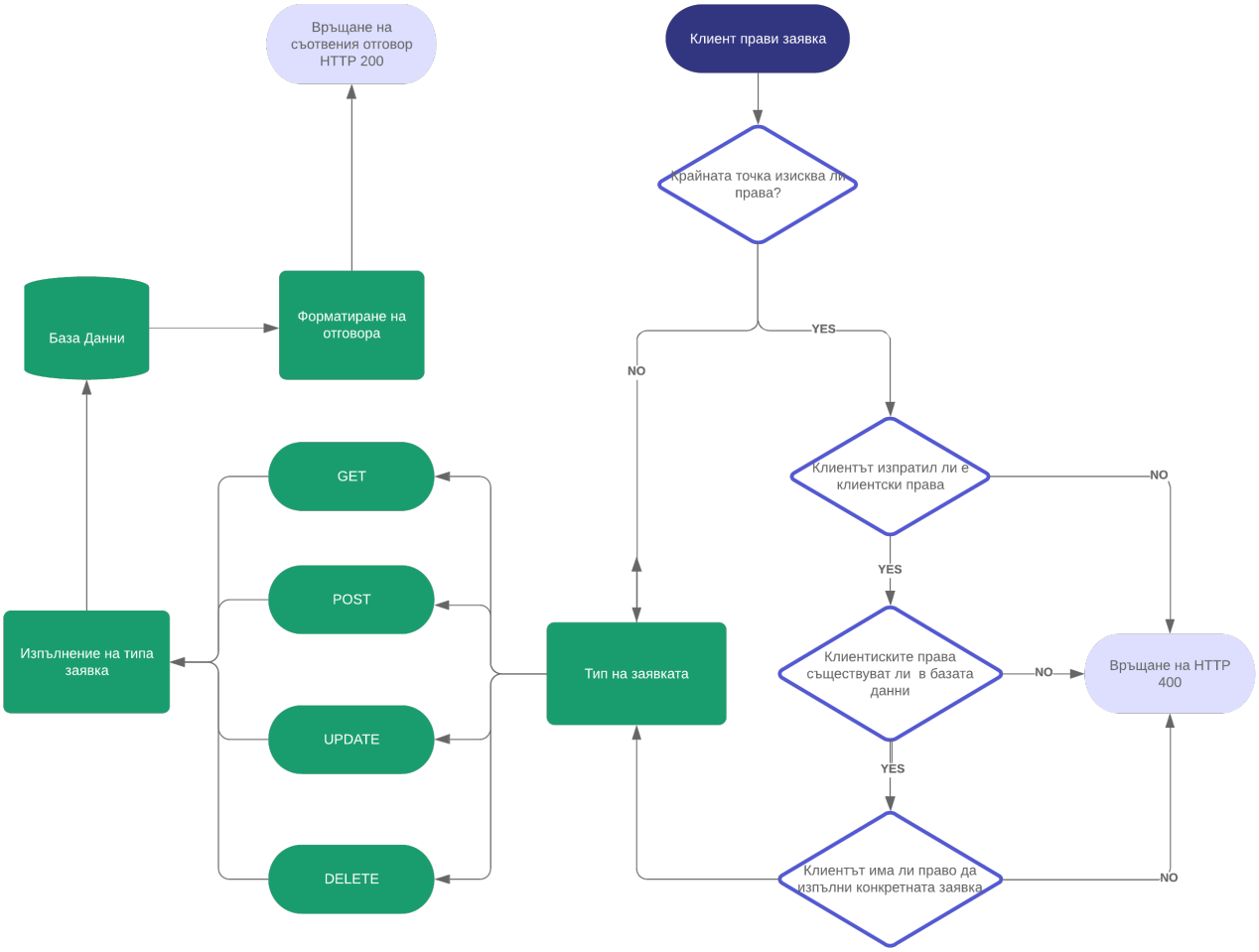
като бекенда е използван от андроид приложение, Swagger много спомага за лесното документиране на всяка крайна точка на бек енд-а. За целта е използван специфично Swagger UI за Spring приложение, имплементирано от Springfox.

2.3 Основен алгоритъм

При изпълняване на заявка алгоритъмът на приложението първо проверява дали крайната точка изисква наличието на права. Ако тя изисква права се проверява дали клиентът има нужните права за изпълнението на заявката. Тъй като не може всеки клиент да прави промени по ресурси, които не му принадлежат, се правят проверки за собствеността на ресурсите, които клиентът желае да манипулира. Заявката се изпълнява ако клиентът, който я създава, съвпада със собственика на ресурсите. Алгоритъмът е показан и с блок схема на фигура 2.6.

2.4 База данни

Базата данни, която е използвана за проекта е PostgreSQL. Причината за този избор е, че PostgreSQL е релационна база данни. В проекта има много връзки между различните модели, които са създадени. Поради тази причина NoSQL база данни като MongoDB не би била добър избор. Специфицирайки връзките между различните модели можем да изберем и действията, които да



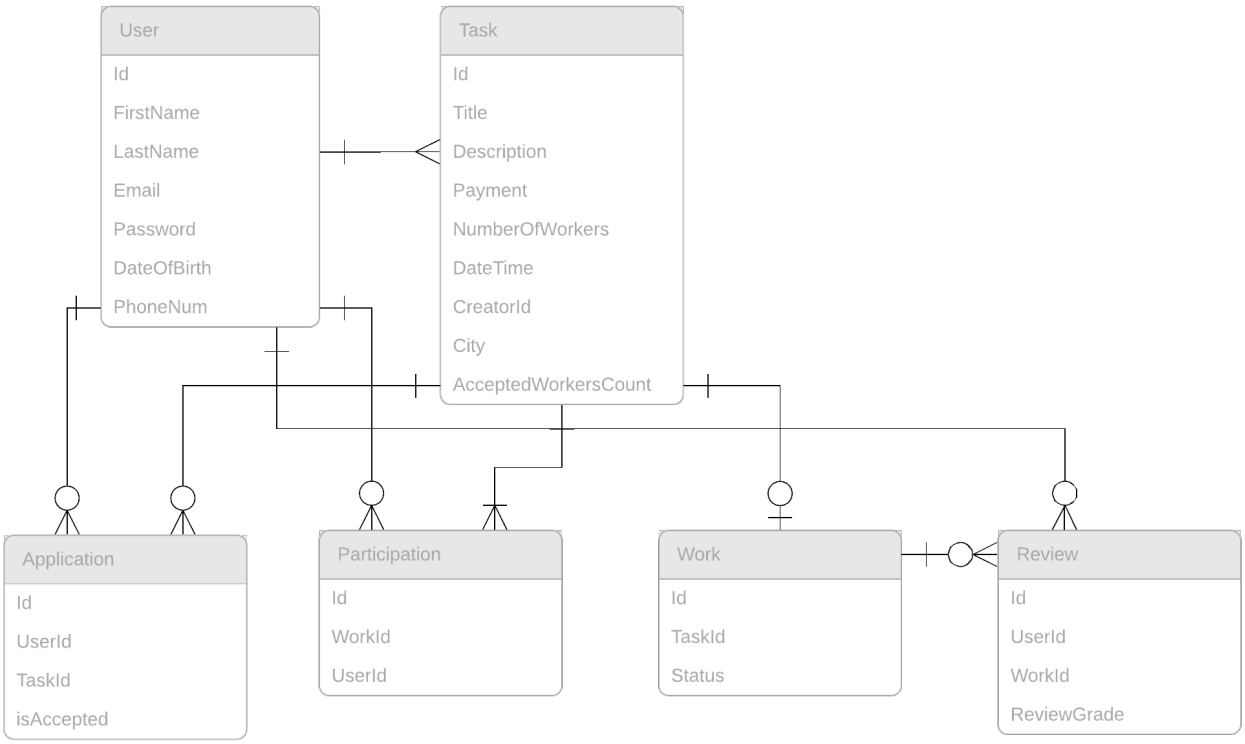
Фигура 2.6: Основен алгоритъм

се извършват при манипулация на свързаните към тях модели. При изтриване на User е желателно да се изтрият и Task-овете, които са създадени от него. PostgreSQL е перфектният избор за тази цел.

Фигура 2.7: PostgreSQL Лого

На фигура 2.8 са показани всичките таблици и връзките между тях. Показаните таблици са:

- User - Информацията за потребителя
- Task - Обява за работа
- Application - Заявление за участие в конкретна работа
- Work - Започната работа
- Participation - Участие на конкретен потребител в конкретна започната работа
- Review - Обратна връзка от работодател към работник, оценка на работника за някоя работа



Фигура 2.8: Схема на базата данни

Глава 3

Реализация

3.1 Функционална част

3.1.1 Сигурност

За осъществяване на сигурността е използван Spring Security, което позволява защитаването на крайните точки на API-то, в този случай, с Basic Auth. Spring Security действа като филтър, който се изпълнява върху крайните точки, които са опоменати като нуждаещи се от защита. Тези точки са:

- "/application/**"
- "/task/**"
- "/work/**"
- "/review/**"
- Всички действия върху "/user/**" освен "/user/login" и POST "/user"

За да се конфигурира защитата на крайните точки, се създава клас, който да разширява класа WebSecurityConfigurerAdapter и да имплементира два метода, обозначаващи крайните точки, които да бъдат защитени и клас, дефиниращ извличането на информацията за потребител от базата данни.

Във фигура 3.1 са дефинирани крайните точки, които да бъдат защитени. "WHITELISTED_URLS" е списък от крайни точки, които не изискват защита,

```
@Throws(Exception::class)
public override fun configure(builder: AuthenticationManagerBuilder) {
    builder
            .userDetailsService(userDetailsService)
            .passwordEncoder(passwordEncoder())

}

override fun configure(http: HttpSecurity) {
    http.run { this: HttpSecurity
        authorizeRequests()
            .antMatchers(HttpMethod.POST,  ...antPatterns: "/users").permitAll()
            .antMatchers(HttpMethod.POST,  ...antPatterns: "/users/login").permitAll()
            .antMatchers(*WHITELISTED_URLS).permitAll()
            .anyRequest().authenticated()
            .and()
            .httpBasic()
         ^run csrf().disable()
    }
}
```

Фигура 3.1:

а чрез “anyRequest().authenticated()” определяме всяка останала крайна точка да изисква защита. В първия метод на фигура 3.1 userDetailsService е обект от тип PostgreUserDetailsService, който дефинира извличането на данните от базата и проверката на правата на конкретния потребител. Имплементацията му е показана във фигура 3.2

```
@Component
class PostgreUserDetailsService(val userRepository: UserRepository) : UserDetailsService {
    override fun loadUserByUsername(email: String): UserDetails {
        val user : User?  = userRepository.findByEmail(email)

        if (user == null) {
            throw UserNotFoundException()
        }

        return User(user.email, user.password, listOf(SimpleGrantedAuthority( role: "USER")))
    }
}
```

Фигура 3.2:

3.1.2 Модели

Моделите са начин за изобразяване на таблиците в базите данни чрез абстракции. Това се случва чрез създаване на клас, анотиран с “Entity”, предоставна

```
@MappedSuperclass
@EntityListeners(AuditingEntityListener::class)
@JsonIgnoreProperties(value = ["createdAt", "updatedAt"], allowGetters = true)
abstract class BaseEntity(
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "id", nullable = false, updatable = false)
        val id: Long = 1,

        @Temporal(TemporalType.TIMESTAMP)
        @Column(name = "created_at", nullable = false, updatable = false)
        @CreatedDate
        val createdAt: Date = Date(),

        @Temporal(TemporalType.TIMESTAMP)
        @Column(name = "updated_at", nullable = false)
        @LastModifiedDate
        val updatedAt: Date = Date()) : Serializable
```

Фигура 3.3: BaseEntity

от Spring Data JPA. Класът трябва да има полета, представляващи колона от таблицата, с която искаме да работим. Пр. Класът User трябва да има поле “firstName”. По този начин при запазване на обекта в хранилището ще се добави нов ред в таблицата “User” с попълнените полета. В проекта има шест такива модела. Всеки модел удължава клас на име “BaseEntity”, който съдържа полета “Id”, “createdAt” и “updatedAt”, както е показано на фигура 3.3

Чрез анотацията “@MappedSuperclass”’ дефинираме класа “BaseEntity” като супер клас, на който полетата биват наследени от останалите модели. Тоест полетата, които съществуват в него като “Id” ще бъдат наследени и от останалите модели.

“@JsonIgnoreProperties” казва на Spring да не десериализира подадените полета. Целта на това е да не се връщат стойностите им, тъй като няма нужда да се използват от клиентите. Съществуването на тези полета е единствено за удобството на бек енд-а.

Във всяка таблица трябва да съществува първичен ключ, който уникално да идентифицира всеки ред от таблицата. Това се случва чрез анотацията “Id”, която оказва полете, над което е поставено като първичен ключ. Освен това искаме уникалния идентификатор да бъде атоматично генериран при вмъква-

нето на нов ред в таблицата. Това се случва чрез анотацията “GeneratedValue”, което подавайки му типа на генериране като “Identity” добавя нов ред с уникален, генериран идентификатор. Чрез анотацията “Column” можем да променяме параметри за съответната колона. В контекста на класа е използва за преименуването на колоната.

Всеки следващ модел в документацията ще наследява този клас, тоест ще има тези полета в таблицата си.

User модел

User модела представлява абстракцията на User таблицата в PostgreSQL базата данни. Таблицата притежава следните полета:

- firstName - Първото име на потребителя
- lastName - Фамилията на потребителя
- email - Електронният мейл на потребителя
- password - Паролата на потребителя
- dateOfBirth - Датата на раждане на потребителя
- phoneNum - Мобилният телефон на потребителя

Чрез анотацията “Table” използвана във фигура 3.4 специфично казваме как желаем таблицата да бъде именувана. Ако не предоставим тази информация Spring Data ще кръсти таблицата спрямо името на класа. Важна роля в класа играят полетата анотирани с “OneToMany”. Това е една от анотациите предоставени от Spring Data, която дефинира връзката между класа и типа на полетата, върху който е поставен. User таблицата има връзка “OneToMany” с таблицата Task. Това означава, че множество редове от типа Task могат да принадлежат на един User, но един Task може да бъде притежаван от само един User. Тези полета обаче няма да бъдат добавени като стойности в таблицата User. Те съществуват единствено за улеснение на програмиста. При желание да извлечем всички Task-ове, които принадлежат на конкретен User, Spring Data автоматично ще извлече и създадените от него съответни Task-ове. В анотацията можем също да изберем какво да се случва със създадените

```
@Entity
@Table(name = "users")
@JsonIgnoreProperties( ...value: "tasks", "applications", "participations")
data class User(
        var firstName: String,

        var lastName: String,

        var email: String,

        var password: String,

        var dateOfBirth: String,

        var phoneNum: String,

        @OneToMany(cascade = [CascadeType.ALL], mappedBy = "creator", fetch = FetchType.LAZY)
        var tasks: List<Task> = emptyList(),

        @OneToMany(cascade = [CascadeType.ALL], mappedBy = "user", fetch = FetchType.LAZY)
        var applications: List<Application> = emptyList(),

        @OneToMany(cascade = [CascadeType.ALL], mappedBy = "user", fetch = FetchType.LAZY)
        var participations: List<Participation> = emptyList(),

        @OneToMany(cascade = [CascadeType.ALL], mappedBy = "user", fetch = FetchType.LAZY)
        var reviews: List<Review> = emptyList()) : BaseEntity() {
```

Фигура 3.4:

от потребител ресурси при неговото манипулиране. Това се случва чрез дефинирата на “CascadeType.ALL”. Това ще извърши всички действия, които се извършват върху потребител и върху неговите Task-ове. Примерно при изтриване на User ще бъдат изтрите и неговите Task-ове, които е създал. Тъй като е възможно един ред от таблица да притежава голям брой редове от друга таблица може да е неефикасно да се извличат всеки път ресурсите, които са създаден от него. Поради тази причина можем специфично да кажем на Spring Data как желаем данните да бъдат извличани. Имаме избор между “FetchType.LAZY” и “FetchType.EAGER”. При първият начин данните ще бъдат извлечени единствено когато се извика Get метода на полето, а при вторият стойностите винаги ще бъдат извличани заедно с обекта. В конкретния случай съм избрал типът на извличане да бъде “LAZY”. В анотацията е попълнен също параметър на име “mappedBy”. Той показва какво е съответното име на полето в класа върху който е поставена анотацията. Например в класа Task има поле на име “creator”. По този начин също показваме на кого принадлежи връзката. В този случай връзката към Task принадлежи на User.

Task модел

Потрбителите имат възможност за създаване на обяви за работа, които са достъпни до всички останали потребители. Това е начинът, по който работодателите намират работници и работниците намират работа, която им харесва. На фигура 3.5 е показан модела на Task. Новите анотации в този клас са “ManyToOne” и “OneToOne”.

- ManyToOne

  Тази анотация представлява абстракция на връзката много към едно като в този случай многото са Task-овете а едното е потребител. Тук FetchType сме дефинирали като “EAGER” тъй като един Task винаги ще има един единсвен създател, което означава, че извличането му не би било тежка операция.

- OneToOne

  Тази анотация дефинира връзката едно към едно. Повече информация за тази анотация ще има в описанието на Work модела.

Полетата в Task модела са следните:

- Title - името на обявата.
- Description - описание на работата, която ще се извършва.
- Payment - сумата, която всеки работник ще бъде заплатен.
- NumberOfWorkers - Броя работници, които са нужни за работата.
- DateTime - Времето, когато работата ще се извършва.
- Creator - Идентификационния номер на създателя на обявата.
- City - Градът, в който ще се извършва работата.
- AcceptedWorkersCount - Броя работници, които вече са приети за работата.

```
@Entity
@Table(name = "tasks")
@JsonIgnoreProperties( ...value: "applications", "work")
data class Task(
        @NotNull
        var title: String,

        @NotNull
        var description: String,

        @NotNull
        var payment: Int,

        @NotNull
        @Size(min = 1, message = "The number of workers needed cannot be less than 1")
        var numberOfWorkers: Int,

        @NotNull
        var dateTime: LocalDateTime,

        @NotNull
        @ManyToOne(fetch = FetchType.EAGER)
        var creator: User,

        @NotNull
        var city: String,

        var acceptedWorkersCount: Int = 0,

        @OneToMany(cascade = [CascadeType.ALL], mappedBy = "task", fetch = FetchType.LAZY)
        var applications: List<Application> = emptyList(),

        @OneToOne(fetch = FetchType.LAZY, mappedBy = "task", cascade = [CascadeType.ALL])
        var work: Work? = null) : BaseEntity() {
```

Фигура 3.5:

Application модел

Application представлява заявление за участие на работа. При желание работник да участва в определена работа, той трябва да създаде заявление. Работодателят има достъп до тези заявления, чрез които преценява кои са най-подходящите работници за неговата работа.

Новото в този модел, показан на фигура 3.6, е полето “uniqueConstraints” в анотацията “Table”. Всеки Application има информация за обявата, на която принадлежи и работника, който я е създал. Искаме да не може един работник да създава повече от едно заявление за една и съща работа. Именно за това служи полето “uniquerContraints” в анотацията “Table”. “UniqueContraints” очаква списък от полета в таблицата, които не могат да бъдат повтаряни. В

този случай желаем да не се повтарят “user_id” и “task_id”. Тъй като сме обявили полетата в модела от тип “ManyToOne” Spring Data автоматично ще ги запише като конкатинира “_id” към името на полето. При опит за повтаряне на двойката полета приложението връща HTTP 400.

```
@Entity
@Table(name = "applications", uniqueConstraints = [UniqueConstraint(columnNames = ["user_id", "task_id"])])
data class Application(

        @ManyToOne(optional = false, fetch = FetchType.EAGER)
        val user: User,

        @ManyToOne(optional = false, fetch = FetchType.EAGER)
        val task: Task?,

        var accepted: Boolean = false) : BaseEntity() {

    override fun toString(): String {
        return ""
    }
}
```

Фигура 3.6:

Work модел

Новото в Work модела е анотацията Enumerated, на която стойност е EnumType.STRING. Тази анотация служи за да можем да изберем по какъв начин да се запазва ENUM стойността. Това поле показва какъв е статус на работата.

Възможните статуси са:

- IN_PROGRESS
- ENDED

Ако изберем EnumType.STRING, стойността ще бъде запазена като String. Ако изберем EnumType.ORDINAL, стойността ще бъде запазена като число, обозначаващо позицията на стойността в Enum класа.

Останалите модели са структурирани на същия принцип.

3.1.3 Основна функционалност

Основната функционалност е създадена чрез модела MVC(Model-View-Controller) на Spring. MVC е архитектурен шаблон за дизайн в програмирането, основан

```
@Entity
@Table(name = "works")
@JsonIgnoreProperties( ...value: "participations")
data class Work(

        @OneToOne(fetch = FetchType.EAGER)
        val task: Task,

        @OneToMany(fetch = FetchType.LAZY, mappedBy = "work", cascade = [CascadeType.REMOVE])
        var participations: List<Participation> = emptyList(),

        @Enumerated(EnumType.STRING)
        var status: WorkStatus = WorkStatus.IN_PROGRESS,

        @OneToMany(fetch = FetchType.LAZY, mappedBy = "work", cascade = [CascadeType.REMOVE])
        var reviews: List<Review> = emptyList()) : BaseEntity() {
```

Фигура 3.7:

на разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение. В конкретния случай, тъй като няма графичен интерфейс в проекта, не е използвана View частта от шаблона.

Клиентът контактува единствнео с контролер, който отговаря за работенето с модела, който от своя страна прави промени или извлича данни от базата данни спрямо бизнес логика.

Контролери

Контролерът е един вид диригента на цялата система. Той определя кои функции трябва да бъдат изпълнени от модела, когато клиент направя заявка към приложението. Контролерът съдържа информация за всички крайни точки, които са достъпни до клиента.

На фигура 3.8 е показан контролера за потребители и един метод от него. В Spring MVC за да дефинираме клас като контролер го анотираме с анотацията @RestController (Или @Controller ако API-то няма да е RESTful). Анотирайки го казваме на Spring, че този клас ще бъде Компонент, който трябва да бъде регистриран и този Компонент ще служи работата на контролер. Важно е всеки клас, който желаем да използваме като Компонент да бъде анотиран с един от следните анотации:

- Controller(RestController)
- Service

```
@Api(value = "User Operations", description = "All operations for users")
@RestController
@RequestMapping( ...value: "/users")
class UserController(val userService: UserService,
                     val applicationService: ApplicationService,
                     val converters: Converters = Converters()) {

    @PostMapping
    @ApiOperation(value = "Create a new user",
            httpMethod = "POST")
    @ApiResponses(ApiResponse(code = 201, message = "Created", response = UserResponse::class))
    fun create(@RequestBody userRegister: UserRegisterBody): ResponseEntity<UserResponse> {
        return ResponseEntity(userService.create(userRegister), HttpStatus.CREATED)
    }
```

Фигура 3.8:

- Repository
- Component

Всеки един от тези анотации има за основа анотацията Component. Анотирайки го с Component казваме на Spring, че желаем да бъде създаден Bean от този тип клас. Създаването на Bean е за да можем да се възползваме от DI(Dependency Injection) и IoC(Inversion of Control), които бяха коментирани в глава 2.

Анотирайки контролера с RequestMapping описваме линк, на който крайните точки на този контролер ще бъдат достъпни.

Метода “create” е анотиран с PostMapping за да дефинираме, че този метод може да бъде достъпен чрез POST заявка. На анотацията не е подадена стойност, следователно ако направим POST заявка на URL “/users” ще извикаме този метод.

Анотациите започващи с името “Api” ще бъдат коментирани по-късно в главата

В проекта има 5 контролера:

- UserController
- TaskController
- ApplicationController
- WorkController

- ReviewController

В аргументите поставяме обект, който очакваме. Spring MVC автоматично десериализира изпратения JSON в обекта, който сме поставили като аргумент. В този случай метода във фиугра 3.8 очаква обект от типа UseBody. Всеки POST метод в контролер очаква Body клас, който представлява абстракция на JSON-а, който се очаква. В този случай очакваме JSON с полета:

- firstName
- lastName
- email
- password
- dateOfBirth
- phoneNum

За да кажем, че обекта който е в аргументите го очакваме като JSON в тялото на заявката го анотираме с анотацията RequestBody.

При изпращане на заявка без такъв JSON или с грешен такъв се връща HTTP 400 Bad Request.

Всеки метод в контролера връща обект от тип ResponseEntity. Обекта позволява за пълен контрол върху HTTP отговора. В него дефинираме тялото на отговора като първи параметър и HTTP статуса, който се връща като втори.

За създаването на тялото на HTTP отговора контролера извиква конкретен метод от услуга, който извършва всичката бизнес логика и мнаипулира данните в базата данни.

Това, което всъщност се връща е обект от типа UserResponse. Тъй като искаме да имаме контрол върху това, което връщаме на потребителя искаме да имаме специални класове, които действат като отговор на заявките. Повечето от моделите имат съответен Response клас. На фигура 3.9 е показан Response класа на потребител. Тъй като паролата е чувствителна информация, не желаем да я връщаме като отговор на заявка.

```
data class UserResponse(val id: Long?,
                        val firstName: String?,
                        val lastName: String?,
                        val dateOfBirth: DateOfBirth?,
                        val phoneNum: String?,
                        val reviews: List<ReviewResponse>? = emptyList())
```

Фигура 3.9:

Услугата, която ползваме сме сложили в конструктура на класа и в този случай услугата се казва “UserService”. Spring автоматично ще намери Bean-а и ще го инжектира в класа.

Услуги

Услугите са Model частта от MVC шаблона. Те изпълняват всичката бизнес логика, когато клиент направи заявка към приложението. Не желаем услугата да има каквото и да било знание за контролера, който го извиква. Една то целите на MVC шаблона е всеки компонент да бъде колкото се може по-отделен от останалите компоненти. Това помага не само за по-доброто преизползване и дебъгване на приложението, но и по-лесното писане на тестове върху конкретния компонент. Всеки контролер има съответен клас от типа “услуга”. На фигура 3.10 е показан класа UserService заедно с метода create от него.

В конструктура на класа се очаква да се подадат четири обекта:

- UserRepository - класа, който представлява австракция за манипулиране на базата данни (Повече информация по-надолу в главата)
- PasswordEncoder - обект за кодиране на парола на потребител. Никога не искаме да пазим паролата на потребител в чист вид.
- AuthenticationDetails - Клас, създаден от мен, за лесното извличане на имейла на потребителя, който прави заявката.
- Converters - Клас, който съдържа удължителни функции за превръщане на клас от тип Entity в тип Response.

Spring авотматично ще ни ги осигури при стартиране на приложението.

```
@Service
class UserService(val userRepository: UserRepository,
                  val passwordEncoder: PasswordEncoder,
                  val authenticationDetails: AuthenticationDetails,
                  val converters: Converters = Converters()) {

    fun create(userRegister: UserBody): UserResponse {
        if (userRepository.existsByEmail(userRegister.email)) {
            throw EmailExistsException()
        }

        if (userRepository.existsByPhoneNum(userRegister.phoneNum)) {
            throw PhoneNumberExistsException()
        }

        val dateOfBirth : DateOfBirth = userRegister.dateOfBirth

        val userDTO = User(userRegister.firstName,
                userRegister.lastName,
                userRegister.email,
                passwordEncoder.encode(userRegister.password),
                dateOfBirth.toString(),
                userRegister.phoneNum
        )

        val savedUser : User = userRepository.save(userDTO)
        with(converters) { this: Converters
            return savedUser.response
        }
    }
```

Фигура 3.10:

На фигура 3.11 е показан класа AuthenticationDetails. В него има един метод на име SecurityContextHolder, който ни предоставя имейла, подаден при аутентикацията. Причината да се нуждаем от този клас е за да можем да получим повече информация за потребителя, който прави заявката, когато имаме нужда от това.

Converters, показан във фигура 3.12 е класа, който съдържа всички удължителни функции за Entity класовете за превръщане в Response класове. При промяна на Response клас единствено трябва да се промени удължаващата функция в този клас. Тъй като тези методи се използват на множествно мес-

```
@Component
class AuthenticationDetails {
    fun getEmail(): String {
        return SecurityContextHolder.getContext().authentication.name
    }
}
```

Фигура 3.11:

та искаме дефиницията на превръщането от Entity в Response да бъде централизирано. Ако променим нещо по Response клас трябва да актуализираме единствено в Converters класа.

Хранилище

Хранилищата са интерфейси, предоставени от Spring Data JPA, за манипулиране и извличане на данните от PostgreSQL базата. За всяка таблица искаме да имаме по едно хранилище, което ще се грижи за действията по нея. Ще разберем как работят те като разгледаме UserRepository на фигура 3.13.

Създавайки интерфейс, който наследява JpaRepository, Spring автоматично ни позволява да използваме вече дефинирани методи. Част от тях са:

- save - запозва подадения обект от тип User в таблицата
- findById - намира ред от таблицата спрямо Id-то, което сме му подали и ни го връща като обект от тип Optional<User>
- deleteById - изтрива ред в таблицата спрямо подаденото Id

Интерфейсът, който наследяваме трябва да получи типа на обектите, които ще бъдат запазени в базата данни и типа на Id-то, което те ще имат. В нашия случай желаем да запазваме User обекти, на които Id-то ще бъде от тип Long.

Често обаче имаме нужда от по-специфични методи за манипулиране на данните. В този случай имаме нужда от метод, който връща Потребител спрямо неговият имейл адрес. Spring предоставя възможността да изграждаме такива методи единствено чрез името, което слагаме на метода. В класа е

```
@Component
class Converters {
    val Work.response : WorkResponse
        get() = WorkResponse(id, task.response,
                participations.map { it.userResponse }, createdAt, status)

    val Task.response : TaskResponse
        get() = TaskResponse(id, title, description, payment,
                numberOfWorkers, dateTime, city, creator.id, acceptedWorkersCount)

    val User.response : UserResponse
        get() = UserResponse(id, firstName, lastName,
                toDateOfBirth(dateOfBirth), phoneNum, reviews.map { it.response })

    val Application.response : ApplicationResponse
        get() = ApplicationResponse(id, user.response, task?.response, accepted)

    val Review.response : ReviewResponse
        get() = ReviewResponse(id, work.id, reviewGrade)

    val Participation.userResponse : UserResponse
        get() = UserResponse(id, user.firstName, user.lastName,
                toDateOfBirth(user.dateOfBirth), user.phoneNum, user.reviews.map { it.response })

    fun toDateOfBirth(date: String): DateOfBirth {
        val values : List<String> = date.split( ...delimiters: "-")
        return DateOfBirth(values.get(0).toInt(), values.get(1).toInt(), values.get(2).toInt())
    }
}
```

Фигура 3.12:

създаден един метод на име findByEmail. Спрямо името на метода Spring автоматично разбира каква е нашата цел. В документацията на Spring Data JPA [2] може да се намери повече информация.

Добавените от нас методи са:

- findByEmail Връща потребител спрямо неговият имейл адрес.
- existsByEmail Връща boolean стойност спрямо това дали съществува потребител с подадения имейл адрес
- deleteByEmail Премахва потребител от таблицата спрямо неговия имейл адрес
- getOneByEmail - намира потребител от таблицата спрямо неговия имейл адрес. Разликата между getOneByEmail и findByEmail е начинът, по който данните се извличат от базата. FindByEmail извлича потребител EAGERLY, докато getOneByEmail го извлича LAZILY.

```
@Repository
interface UserRepository : JpaRepository<User, Long> {
    fun findByEmail(email: String): User?
    fun existsByEmail(email: String): Boolean
    fun deleteByEmail(email: String)
    fun getOneByEmail(email: String): User
    fun existsByPhoneNum(phoneNum: String): Boolean
}
```

Фигура 3.13:

- existsByPhoneNum Връща boolean спрямо това дали потребител с подадения телефонен номер съществува.

Останалите хранилища са направени по същия модел.

Едно от изискванията, които имаме към приложението е да може да се филтрира спрямо полетата на Task. За да изпълним това изискване трябва да се възползваме от QueryBuilder предоставен от Spring Data.

```
@Repository
interface TaskRepository : JpaRepository<Task, Long>, CustomTaskRepository {
    fun findAllByCreatorId(pageable: Pageable, creatorId: Long?): Page<Task>
}

interface CustomTaskRepository {
    fun findAll(pageable: Pageable,
                title: String?,
                paymentStart: Int?,
                paymentEnd: Int?,
                numWStart: Int?,
                numWEnd: Int?,
                dateStart: LocalDateTime?,
                dateEnd: LocalDateTime?,
                city: String?): List<Task>
}
```

Фигура 3.14:

За тази цел трябва да добавим нов метод, на който искаме да добавим тяло. Започваме със създаването на interface, който дефинира метод с всичките полета в един Task, както е показано на фигура 3.14. След това, за да може

TaskRepository да използва метода искаме да наследява създадения от нас interface. Последната стъпка е да създадем клас, който да наследява създадения от нас interface и да попълним метода.

```
@Repository
class CustomTaskRepositoryImpl(val em: EntityManager) : CustomTaskRepository {
    override fun findAll(pageable: Pageable, title: String?,
                         paymentStart: Int?, paymentEnd: Int?,
                         numWStart: Int?, numWEnd: Int?,
                         dateStart: LocalDateTime?, dateEnd: LocalDateTime?,
                         city: String?): List<Task> {
        val cb : CriteriaBuilder! = em.criteriaBuilder
        val cq : CriteriaQuery<Task!>! = cb.createQuery(Task::class.java)

        val task : Root<Task!>! = cq.from(Task::class.java)
        val predicates : MutableList<Predicate> = mutableListOf<Predicate>()

        if (title != null) {
            predicates.add(cb.like(task.get("title"), pattern: "%" + title + "%"))
        }
```

Фигура 3.15:

На фигура 3.15 е показана имплементацията на създадения от нас клас. Методът приема полетата на един Task и спрямо това дали са null ги добавя в списък от Predicate. За да създадем персонализирания query искаме първо да използваме предоставения от Spring EntityManager. Той е централния обект, който комуникира с базата данни и чрез него извличаме обект от тип CriteriaBuilder. Така съставяме Query, на което подаваме резултатния клас (в този случай Task) и класа спрямо който ще извършваме Query-то(отново Task). След като всичките нужни предикати са добавени ги добавяме към Query-то(cq) и го изпълняваме. Резултатния списък е всички Task-ове, които отговарят на филтъра.

Причината да извличаме данните чрез QueryBuilder е поради натоварване. Възможно е първо да се извлиекат всички Task-ове от базата данни и след това да се филтрира спрямо полетата, но в случая, че имаме голям брой Task-ове тяхното извличане би било бавен и тежък процес.

Swagger

За имплементирането на Swagger е нужно създаването на един Configuration клас.

```
@Configuration
@EnableSwagger2
class SwaggerConfig {
    @Bean
    fun api(): Docket {
        return Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage( basePackage: "bg.elsys.jobche"))
                .paths(PathSelectors.any())
                .build()
                .apiInfo(apiInfo())
                .securitySchemes(listOf(BasicAuth( name: "basicAuth")))
    }

    fun apiInfo(): ApiInfo {
        return ApiInfo( title: "Jobche",
                description: "Application for creating ads for easy and fast work",
                version: "BETA",
                termsOfServiceUrl: "None",
                Contact( name: "Radoslav Hubenov",  url: "https://github.com/rrhubenov",  email: "rrhubenov@gmail.com"),
                license: "None",
                licenseUrl: "None",
                emptyList())
    }
}
```

Фигура 3.16:

Този клас е показан на фигура 3.16. Анотираме класа с Configuration и EnableSwagger2. Както е показано трябва да добавим един метод, който връща Bean от тип Docket. Причината за това е, че Swagger има нужда от конфигурации, които да настроим. Като например в кой пакет се намират контролерите. В този случай сме посочили основния пакет на приложението “bg.elsys.jobche”. Swagger автоматично ще намери всички класове анотирани със Spring анотация Controller и ще ги добави към документацията.

В много случаи обаче желаем да добавим повече информация за конкретната крайна точка. Това се случва чрез анотации започващи с Api. Както показахме на фигура 3.8 чрез тях можем да добавим наши обяснения към документацията. Като пример функцията на специфичната крайна точка, какво връща, какво очаква и тн.

3.2 Тестова част

Много важна част от създаването на проекта са тестовете. Старал съм се да спазвам TDD(Test Driven Development) принципи при създаването му. TDD - изработка чрез тестове, спрямо дефиницията в уикипедия, е метод за разработка на софтуер, при който се спазва следният работен цикъл: първо се пишат тестови варианти (test cases), които да покрият изискванията за новия изходен код, а чак след това се пише програмния код така, че да покрива тези тестове. Кент Бек, който се счита за създател на този метод, заявява през 2003, че TDD цели един опростен дизайн[6].

За спазване и разработка чрез TDD съм използвал презентацията на Uncle Bob[1]. В него той дефинира три правила за използването на TDD:

1. Не се пише фунцкионален код, освен ако е с цел да се накара тест да мине.

2. Пишеш само толкова тестов код, колкото е нужно за да се провали.

3. Пише се единствено толкова функционален код, колкото е необходимо за да мине теста.

Тези три правила те вкарват в един цикъл на изработване на тестове и писане на функционален код. Пише се част от тест, пише се част от функционалната част. Предимствата на TDD са особено подходящи за нови програмисти. Използвайки тази методология програмиста успява да си представи архитектурата на проекта, който иска да създаде. Кара го да мисли за това какво функционалния код трябва да прави, отколкото КАК да го прави. Въпреки това писането на тестове не е лесна работа. През целия проект писането на тестове отне 80 процента от цялото време. Средно за всеки ред функционален код е написан 2 реда тестове. Те също имат архитектура, която трябва да се спазва и поради тази причина писането им не бе лесна задачка.

3.2.1 Unit тестове

Unit тест представлява тест върху един компонент. Компонент в този случай е клас. В проекта за всеки контролер и всяка услуга има по един Unit Test клас,

който тества всичките методи и изпълнява test cases. Както се споменава в глава 2 за писането на тестове е използван JUnit5, mockK и assertJ.

Тестове върху контролери

При писане на Unit тестове желаем да тестваме отделните компоненти изцяло отделени от останалите или от зависимостите си. В контекста на UserController искаме да бъде тестван без да се притесняваме дали UserService, или която и да е друга зависимост, работят правилно. Тестът трябва да бъде изцяло съсредоточен върху методите и функционалността единствено на компонента, който тестваме.

```
class UserControllerTest: BaseUnitTest() {

    companion object {
        val userResponse : UserResponse  = DefaultValues.creatorUserResponse()
        val userBody : UserBody  = DefaultValues.creatorUserBody()
    }

    private val userService: UserService = mockk()
    private val applicationService: ApplicationService = mockk()

    private val controller = UserController(userService, applicationService)

    @Nested
    inner class create() {
        @Test
        fun `create should return valid user response`() {
            every { userService.create(userBody) } returns userResponse

            val result : ResponseEntity<UserResponse>  = controller.create(userBody)

            assertThat(result.body).isEqualTo(userResponse)
        }
    }
```

Фигура 3.17:

На фигура 3.17 е показан класа UserControllerTest и един тестов метод от него. Всеки UnitTest наследява класа BaseUnitTest, който осигурява анотацията @ExtendWith(MockKExtension::class). Това е нужно за да бъде регистриран MockK като осигурител на Мок обектите. Мок обектите служат като заместител на зависимостите, които някой клас използва. В този слу-

чай мокваме двете услуги, които UserController класа използва - UserService и ApplicationService. След като са мокнати, за да няма грешка в тестове, трябва да контролираме какво се случва при изпълнението на техните методи. За това служи и метода every. Казваме му при изпълнението на метода create с подаден аргумент userBody върни обект userResponse. По този начин осигуряваме, че независимо дали зависимостите работят правилно или грешно, единствено тестваме метода от компонента дали се изпълнява правилно.

Тестове върху услуги

Тестовете на услугите се извършват по същия принцип като тестовете на контролерите. Мокваме зависимостите на класа и очакваме, че той ще върне отоговра, който очакваме.

3.2.2 Integration тестове

Integration тестовете служат за тестване на цялостен процес на софтуера. Симулира се цялостна заявка към бекенда, и се очаква че ще се върне отговора, който очакваме.

```
@Test
fun `creating a user with an already existing phone number should return 400`() {
    val invalidUserBody = UserBody(FIRST_NAME, LAST_NAME,  email: "Random@asd.com", PASSWORD, DATE_OF_BIRTH, PHONE_NUM)
    val invalidRegisterResponse : ResponseEntity<UserResponse!>! =
            restTemplate.postForEntity(REGISTER_URL, invalidUserBody, UserResponse::class.java)

    assertThat(invalidRegisterResponse.statusCode).isEqualTo(HttpStatus.CONFLICT)
}
```

Фигура 3.18:

На фигура 3.18 е показан един тест от UserIntegrationTest класа. Важно е името на метода да обозначава колкото се може по-ясно каква е целта на теста. Поради тази причина използваме функционалността на Kotlin за именуване с празно място между думите чрез обграждане със знака ‘. В теста при опит за създаване на потребител, чийто телефонен номер вече регистриран отговора да е от тип HTTP 400. За създаване на заявката използваме обект от тип TestRestTemplate. Той ни позволява да изпълняваме различен тип заявки към дефиниран линк и да получим обратно отговор като му предоставяме в какъв

обект да десериализира JSON отговора. В този случай единствено искаме да знаем, че статус кода, който е върнат е 400.

Глава 4

Ръководство

4.1 Инсталиране на продукта

За инсталиране на продукта може да се клонира хранилището на проекта в GitHub.

Линк: https://github.com/Jobche/Jobche-BE

За да се клонира първо трябва да е инсталиран git клиента за терминал и да се изпълни командата:

“git clone https://github.com/Jobche/Jobche-BE”

Това ще клонира целия проект в текущата директория.

4.2 Използване на продукта

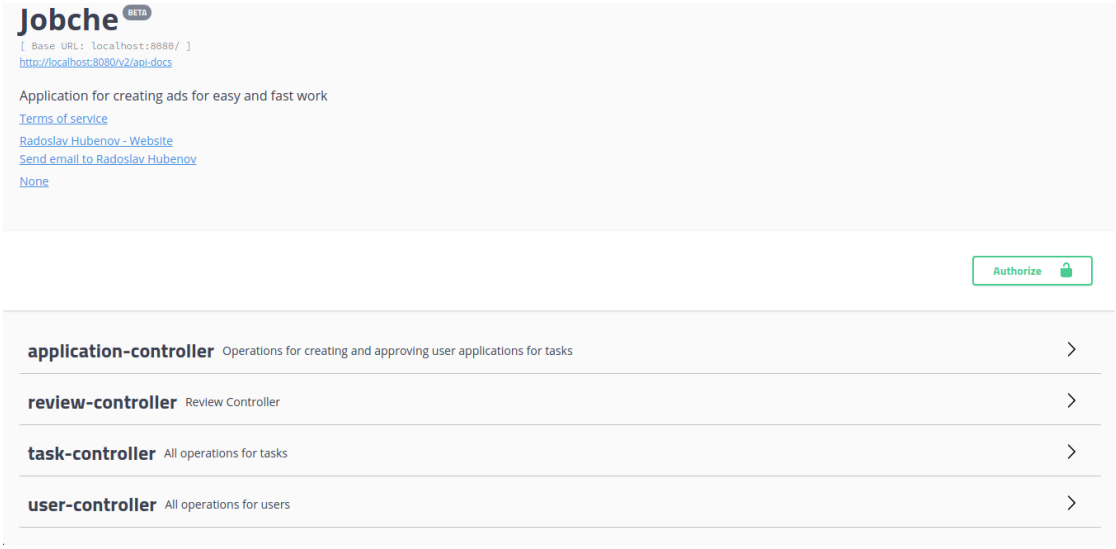
4.2.1 Стартиране

За стартиране на продукта има няколко варианта, но най-лесните са:

- Ако имате инсталиран IntelliJ IDE просто отворете проекта в него и натиснете зелената стрелка горе-вдясно на прозореца. Това ще стартира проекта.
- През конзолата може да се стартира като се влезне в директорията на проекта и се напише “./gradlew bootRun”. Това ще изпълни gradle Task, който стартира целия проект.

4.2.2 Използване

При стартиране на проекта на локална машина може да се достъпи на URL:localhost:8080. Това е основния линк, от който могат да се достъпят крайните точки (Примерно localhost:8080/users/create е линка за създаване на потребител). Но най лесният начин за използване ако желаете единствено да го пробвате е чрез Swagger. На линк "localhost:8080/swagger-ui.html" може да се достъпи Swagger интерфейса на проекта.



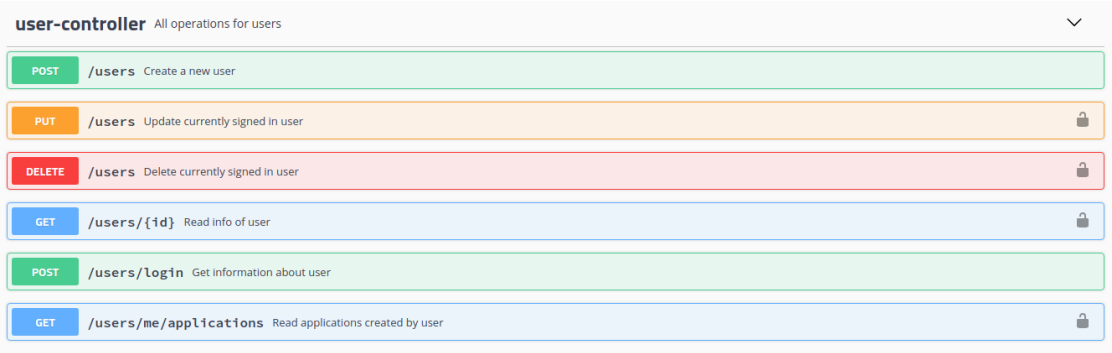
Фигура 4.1:

На фигура 4.3 е показан част от интерфейса на Swagger. През него могат да се изпълняват заявки директно към продукта, но основната цел е за документиране на крайните точки.

Когато се натисне върху един от контролерите чекмеджето се отваря за да покаже всяка крайна точка, която този контролер съдържа.

От тук клиент може да открие всичката му необходима информацип за достъпните крайни точки. Описана е функцията, която те извършват, както и JSON тялото, което очакват и което връщат.

Отваряйки крайна точка виждаме полетата, които се изискват, както и възможните отговори, които можем да получим. Позволение на това всеки, който желае да използва API-то за да изработи приложение спрямо него може лесно да разбере всичката нужна информация.



Фигура 4.2:

Фигура 4.3:

Показан ни е обекта, който крайната точка очаква да получи както и кои от тях са задължителни. Всяко поле, което има звезда над името си е задължително.

Натискайки Try it out и след това Execute се изпълнява заявка с подаденото JSON тяло, на която заявка се връща и отговора от сървъра.

Заключение

Проектът успява да покрие основните функционалности на приложение за работа, но има много детайли, които биха могли да бъдат добавени. Процесът за създаване на обява за работа, осигуряване на плащанията между клиентите и качеството на свършената работа са неща, които не са обект на текущия проект, но могат да бъдат покрити в последствие. Приложението е в предпазарен етап на разработка.

За да бъде предложено на пазара е нужно да бъдат добавени следните функционалности:

- По-добра защита
- Login с фейсбук
- Чат система
- Начин за осъществяване на транзакции
- Система за нотификации
- По-добри опции за описание на работата

Използвана литература

[1] Uncle Bob. The Three Laws of TDD (Featuring Kotlin). URL: https://www.youtube.com/watch?v=qkblc5WRn-U&t=3416s.

[2] Oliver Gierke et al. Spring Data JPA - Reference Documentation. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.core-concepts.

[3] Philipp Hauer. Best Practices for Unit Testing in Kotlin. URL: https://www.youtube.com/watch?v=RX_g65J14H0.

[4] Andrei Solntsev. Intellij vs. Eclipse: Why IDEA is Better. URL: https://dzone.com/articles/why-idea-better-eclipse.

[5] Magnus Vinther. Why you should totally switch to Kotlin. URL: https://medium.com/@magnus.chatt/why-you-should-totally-switch-to-kotlin-c7bbde9e10d5.

[6] Wikipedia. Test Driven Development. URL: https://en.wikipedia.org/wiki/Test-driven_development.