ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – ВАРНА

ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ

Катедра „Софтуерни и интернет технологии“

A blue and white logo

Description automatically generated with medium confidence

**ДИПЛОМЕН ПРОЕКТ**

за придобиване на ОКС „Бакалавър”

**Тема:**

**ПРОЕКТИРАНЕ И РАЗРАБОТВАНЕ НА СИСТЕМА ЗА КОНТРОЛ НА КЛИЕНТИТЕ**

**Дипломант:**  **Научен ръководител:**

Ивайло Пламенов Руменов  Доц. Венци Николов

**Специалност** Софтуер и интернет технологии

**Фак. № 19621627**

Съдържание

[I. Увод 2](#_Toc139264990)

[II. Изложение 3](#_Toc139264991)

[1. Обзор на използваните програмни средства и технологии. 3](#_Toc139264992)

[2. Оценка на пазара и конкуренцията 4](#_Toc139264993)

[3. Оценка на конкуренцията 6](#_Toc139264994)

[4. Модел на Ансов приложен за система за контрол на клиентите. 8](#_Toc139264995)

[5. Планиране на тактика за разработка на програмния продукт 13](#_Toc139264996)

[III. Проектиране 15](#_Toc139264997)

[1. Разработка на база от данни. 15](#_Toc139264998)

[2. Реализация на интерфейса за програмното приложение 24](#_Toc139264999)

[A. Обзор на нужните технологии и алгоритми 24](#_Toc139265000)

[B. Проектиране на програмния код 32](#_Toc139265001)

[C. Тестване на системата 55](#_Toc139265002)

[3. Реализация на потребителски интерфейс 58](#_Toc139265003)

[IV. Заключение 70](#_Toc139265004)

[Цитирани източници 71](#_Toc139265005)

[V. Приложения 72](#_Toc139265006)

# Увод

В дипломната работа се разглеждат в детайли всички особености за създаване на система за контрол на потока от клиенти. Започвайки от гледна точка на използваната технология, която е заложена за реализация на идеята, последвано от оценяване значимите функционалности спрямо конкуренцията на пазара. Използването на нужните критерии, които се приложими към проекта според пазарните критерии, служат за гаранция, че продукта е конкурентно способен спрямо целевия пазар. После се преминава към оценка на конкуренцията и потенциала на продукта с възможност да се развие на нашия и чужд пазар спрямо метода на Ансов[[1]](#footnote-1) за оценка на пазара. След тези нужни за доказателство за устойчивост на проекта се преминава към планирането на тактиката за разработка, която се следва за реализацията на програмния продукт, т.е. главния модел. Планът е ключов и трябва да се зададе предварително за да може да се следи времетраенето на разработката, както и пресмятането на нужните ресурси и поради това е включен в първа глава на дипломната работа.

Проектирането е следващата глава, което се разглежда в детайли. То служи за да поясни методите използвани за разработка на множеството части на проекта. Според зададения модел на работа, първо се разработва базата данни.

След което се преминава към разработка на реализация на интерфейса за програмираното приложение, или така наречения сървърен интерфейс или модел на бизнес логиката. В него се разглежда първо обзора на технологии и алгоритми, нужни за реализация на проекта и се преминава към обосновка на методите използвани в програмния код. Реализацията на потребителския интерфейс е последната под-глава от главата на проектиране, който е създаден след анализ и оценка на потребителите и съобразен с техните потребности.

# Изложение

## Обзор на използваните програмни средства и технологии.

В тази под глава се разглеждат в детайли подбраните за задачата програмни средства и технологии нужни за реализацията на проекта. Започвайки от базата от данни, която както името гласи седи като основа на проекта и нейната разработка е ключова за цялостния характер на системата. Затова е избрано да използва релационна база от данни. Поради самия характер на релационните бази от данни, които поддържа множество вписвания на данни с възможност за мащабиране в бъдещето. Конкретно е избрано да се използва Postgresql[[2]](#footnote-2), защото е свободна и отворена релационна база данни с възможности; като много потребителски достъп, гъвкави опции при разширяване и е транзакционно базирана тоест поддържа ACID[[3]](#footnote-3) (Atomicity, Consistency, Isolation, Durability).

Следващия ключов елемент за проекта е самия програмен език на който се извършва цялостната функционалност на приложението. За целта е нужен език, който да може да борави едновременно с база от данни и да изпраща нуждата информация към потребителя. Затова за изпълнението на задачата е избрано Java[[4]](#footnote-4) версия седемнадесет с Spring[[5]](#footnote-5) версия 3, като фреймуърк който спомага за по-устойчиво имплементиране на java езика. Сървъра подбран за реализацията и разгъване на задачата е Apache[[6]](#footnote-6) Tomcat[[7]](#footnote-7). Инструмента за изграждане, който е подбран за проекта, е Maven[[8]](#footnote-8) заради обвързване на всичките зависимости към програмния продукт. Зависимостите към проекта включват Spring security, за устойчива имплементация на защита на клиенти и потребители, Log4j за всички нужни логвания по проекта и за допълнителна информация при сриване или разкриване на бъгове[[9]](#footnote-9), Hibernate за динамичното закачане на java класове с базата от данни и не на последно място стои Junit Jupiter версия 5, която служи за тестване на индивидуалните части от код, както и за интеграции с Mockito. Mockito служи за моделиране на отделни нужни части за тестване. Средата за програмиране на проекта е подбрана IntelliJ, избрана поради нейната възможност за лесно и гъвкаво подреждане на проект свсичките му градивни елементи.

Като метод за визуализация, който трябва да отговаря на стратегията на проекта, която гласи че е нужен уеб базирана визуализация. За целта е избрана Vanila JavaScript[[10]](#footnote-10) съчетана с html[[11]](#footnote-11) страници. Сървърът подбран за реализация и разгъване на визуализацията е Vite версия 4.3.5, който е широко прието поради неговата стабилна екосистема за разработване на уеб приложения.

Съвкупността от тези елементи позволява гъвкавост на начина на подреждане и администриране на системата контрол на клиентите, както и устойчивост на цялостния проект. С помощта на силно типизиран език като java, когато се открие уязвимост в част от системата или даден пакет и/или библиотека то може да се отстрани с лекота без да се налага цялостна преработката. Това е поради самата методология на обектното ориентирано програмиране. С правилна енкапсулация и имплементиране на интерфейси може да се осигури програмния продукт да издържи на външния натиск от страна на потребители, клиенти и злонамерени личности.

## Оценка на пазара и конкуренцията

В тази под подглава се разглежда пазарното проучване, което е предприето преди реализацията на програмния продукт.

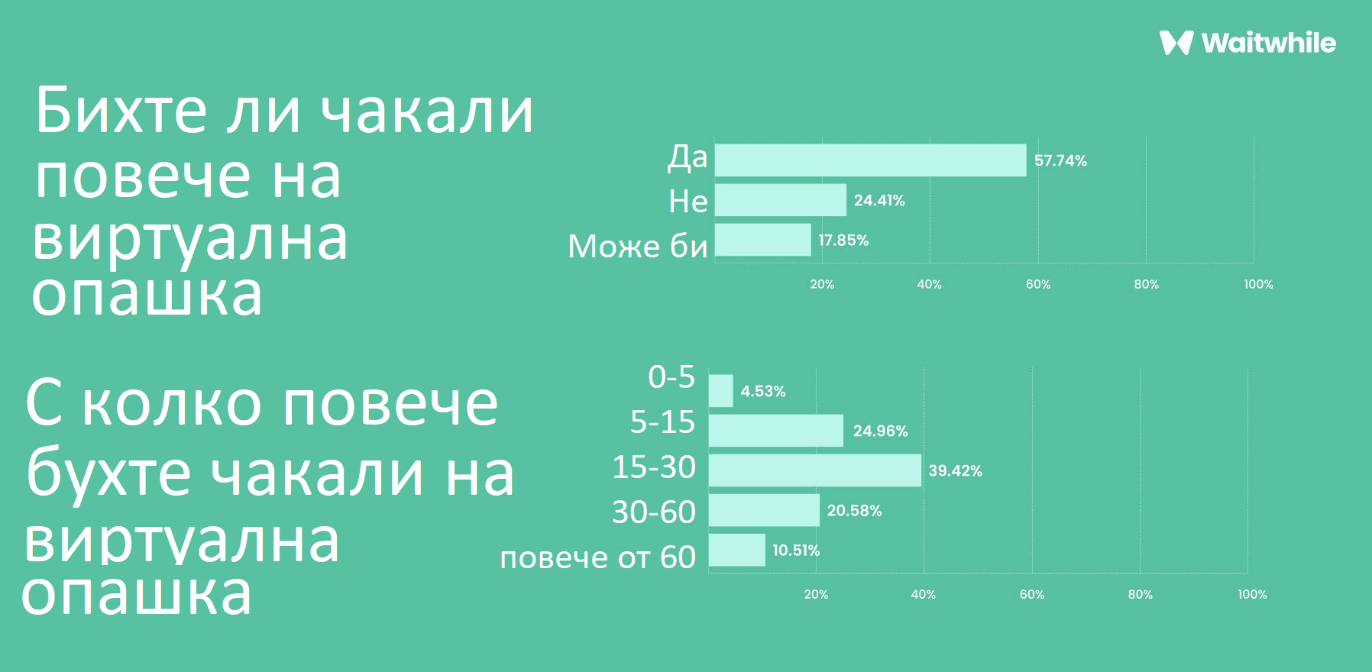
При разглеждането на конкуренцията, е добре да се ползва статистика, която е свързана със системи за контрол на клиентите, рентабилност и/или полезност. Всеизвестно е, че никой където и да е, не харесва да седи на опашка. Според годишната статистика от „*Wait while*“[[12]](#footnote-12) само в Съединените Американски щати клиенти чакат близо до 37 билиона часа чакайки на опашка всяка година. Както и според същата статистика най-често се прекарва време чакайки на опашки за базови потребности като храна и лекарства (Фиг. 1).

Демографска статистика, която е целева за проекта дава информация не толкова за най-концентрираните места, където има опашки, а там където една виртуална опашка би ускорила работния процес на заведенията, които имат малко персонал, но голям поток от клиенти. Според същата годишна анкета за чакане на опашки, хората които най-често чакат на опашки, около 70% от тях предпочитат да чакат на виртуална опашка. Това е такава опашка, на която е предоставен билет под някаква форма. Тази форма може да е физически билет или електронен, на който е отреден реда. Според статистиката хората, които чакат на виртуална опашка, са готови да чакат средно с по петнадесет минути повече от колкото ако са на физическа опашка.

Картина, която съдържа текст, екранна снимка, дизайн

Описанието е генерирано автоматично

Фиг. 1. Статистика от анкетирани къде най-често чакат на опашка.



Фиг. 2. Статистика за предимствата на виртуална опашка.

Когато се вземат тези данни в предвид можем да се оцени проекта за разработка на гъвкава система за контрол на потока от клиенти, който е нужен и предпочитан от мнозинството.

## Оценка на конкуренцията

Оценката на конкуренцията представлява разпознаването на услугите и предимствата помежду различните фирми. Анализа на фирмите конкуренти включва: вида на предлаганата услуга, под каква форма се заплаща и съответната цена за поддръжка, целевият им пазар, функционалността на продукта им, набора от клиенти. За правилно оценяване на конкурентоспособността на проекта е изведен анализ на няколко фирми конкуренти, по които се оценя програмния продукт. Всяка фирма е разгледана в детайли, след което е представена в таблица ( Фиг. 3.) за ясна визуализация.

Започвайки от „*Esii*“, която е от лидерите на инсталации на софтуера им върху различни билето - подаващи машини. Билето - подаващата машина представлява физична машина, която разпечатва хартиени билети върху което е записано номера на билета и за коя услуга или гише чака клиента. Фирмата разполага и с теглене на виртуални билети за заведението. И конкретно мобилно чрез уеб сайт за теглене на билети. Фирмата е разделила и ограничила целите демографски на: търговци на дребно, медицински сектор, публичен сектор и финансов сектор. Пазарният им дял се намира в членките от европейския съюз, Канада и Кот д'Ивоар. Събраните данни за фирмата са предоставени от собствения им уеб сайт: https://www.esii.com/en/

Преминавайки към „Q-net“, които предлагат: визуализация на опашката, организация на опашката със снимки и ключови думи, поддържат редовни клиенти с карта или ( Near-field communication) NFC[[13]](#footnote-13), обратна връзка с купувача на системата, статистика и модифициране на вече вградената логика спрямо изискванията на клиента. Главната им демографска е заложена въм финансовия сектор и конкретно с банкови клонове. Данните за дейността на фирмата са извлечени от техния уеб сайт: https://q-net.pro/index.html

Следващата фирма „Q-better“ се е откроила на пазара като предпочитана фирма за работа с опашки. Системата им поддържа: едновременно менажиране на билето -подаващи устройства и на виртуални билети, статистика, уеб запазване на часове, както и редактиране на вече създадената логика на опашката за преференцията на клиентите. Фирмата разполага с широка демография, като се започне с търговски магазини, телекомуникационни компании, медицински центрове, публичния сектор, образователни институции и банковите клонове. Клиентите са разпространени в петдесет различни държави на четири континента включвайки: Азия, Африка, Южна Америка и Европа. Всички данни по дейностите на фирмата са предоставени от техния уеб сайт: https://web.q-better.com/

Фирмата „Qudini“ е една от най-развитите, когато стане дума за дълбочина на софтуера и неговата функционалност. Софтуерът поддържа: множествена връзка с различните клонове на дадена фирма, облачно базирана сигурност, преложим софтуер на множество операционни системи, готов за употреба с предефинирани шаблони за употреба на опашката, различна имплементация на опашката спрямо клона, бизнес статистика. Фирмата разполага с потвърдили се марки на пазара като техни клиенти. Фирмата е широко разпространена, но целевия и пазар се намира в Съединените Американски щати. Клиентите им включват, но не са ограничени до: Mc Martens, Bass pro shop, Nike, Iqos Sky, Next west, Tesco, East West bank и много други. Оценява се като сериозен конкурент твърдо стъпил на пазара, според самите тях: https://www.qudini.com/

Последната фирма „Kiosk“ е съсредоточена главно в билето - подаващите машини, което е сериозен подход към това да се зачисли демография и да се предържа към нея. С лимитираната информация предоставена от фирмата може да се оцени, че главните клиенти на фирмата са банкови клонове, медицински центрове и търговски магазини. Софтуерът предлага визуализация на опашката, както и визитация върху самото билето - подаващо устройство. Оценката е формирана от данни предоставени от самите тях: https://www.viewpointfeedback.com/

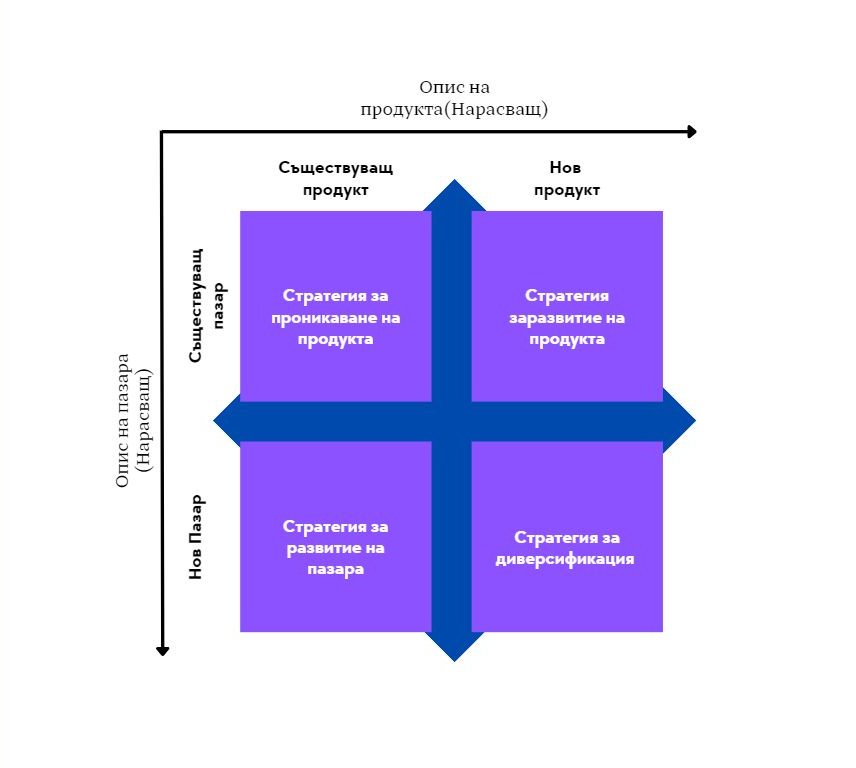
Картина, която съдържа текст, екранна снимка, Шрифт, линия

Описанието е генерирано автоматично

Фиг. 3. Таблица на конкуренцията, която изобразява различните функции, които фирмите конкуренти предлагат.

## Модел на Ансов приложен за система за контрол на клиентите.

Когато се разполага с информация конкретна за фирмите конкуренти, може да се изгради как желаният продукт изглежда на пазара, и той до колко е конкурентно способен. При проучване на конкурентно способността може да се приложи модела на Ансов. Прилагайки и употребявайки неговата матрица. Матрицата на Ансов представлява стратегията на фирмата спрямо в кой пазар се намира (Фиг. 4) и какъв продукт предлага. Разглеждат се следните точки:



Фиг. 4. Матрица на Ансов.

1. **Стратегия за проникване на продукта**

При стратегията за проникване на продукта, се следи как в съществуват пазар и съществуващ продукт и подхода на фирмата по дадените точки:

* Приход от закупува:

Предвижда се 120$ на месец за всяка инстанция или отдел на фирмата купувач, т.е. ако купувача има няколко обекта заплаща лиценз за всеки един от тях.

* Група на продукта:

Групата на продукта се унаследява от по-широките системи за масово обслужване. Така продукта като цяло попада в по-широк пазар. Това е така защото купувач търсещ начин за обслужване на повече клиенти може да не избере система с билети а неин алтернатива.

* Отклонения:

Отклонението считано за стратегически план е това, че не са нужни апарати за маркиране на билети или билето-подаващо устройство.

* Новост:

Новостите вписани за стратегията са богатият избор на видовете обслужване, т.е. клиента да притежава възможността да си нагласи системата по неговите потребности.

* Процент 20-80:

В тази стратегия се разглежда дали 80% от приходите идват от 20% от клиентите. Това е приложимо за продукта и неговите възможности. Ако заведение като увеселителен парк или верига кина решат да имплементират масово системата, може да се приложи стратегията 20-80. Като пример може да се вземе кино Арена, която е месното за България пазар верига кина. На пазара те притежават само седем кина, което по устроените тарифи означавайки по 120$ на месец за всяко кино. Водейки до заключението от 10 080$ годишен приход от една фирма.

* Икономия от мащаба:

В стратегията с фиксирана цена и план за поддръжка на системата, не е възможна икономия от мащаба.

1. **Стратегия за развитие на пазара**

Когато се разглежда стратегията за развитие на пазара се взема в предвид, че приложим съществуващ продукт на нов пазар. За правилно прилагане на подобна стратегия, се разглеждат следните точки:

* Как може да се потвърди на пазара:

Когато стратегията е потвърждаване на пазара, може чрез рекламация на множество езици. Така се преминава през социални бариери позволявайки на продукта да се развива.

* Темпото на новия пазар:

Когато се стъпва на нов пазар продукта има за задача да навлезем с темпото на новия пазар, т.е. ако е утвърдено в един пазар, като на пример англоговорещия свят, и реши да се премине към хинди говорещ пазар то има изградена инерция да настъпи към по-малък но се така конкурентен пазар.

* Брой участници:

Броя участници за дадения продукт е ключов, защото при установили се продукти те трудно се сменят. Поради естеството на система, която гласи че се ползва безвъзвратно и продължително от клиента. Самата идея за смяна на системата, възниква или от неефективноста и/или от нарастващ поток от клиенти.

* Входни бариери:

Софтуера сам по себе си няма много бариери за преминаване със съществуващият продукт на нов пазар. Софтуера се закупува и се употребява.

* Уникалното:

Уникалността като стратегия за продукта на нов пазар, е достатъчно само да се спомене нейната гъвкавост на прилагане.

1. **Стратегия за развитие на продукта**

Стратегията за развитие на продукта може да се разгледа като се реши да се надградим над избора на предефинирани шаблонно, които допълват и улесняват конфигурацията на системата. Това е само една от стратегиите, който могат да се предприемат когато има нов продукт на същия пазар. Друга подобна стратегия да се вкарат конкретни демографии към рекламата на продукта, т.е. да се преименуват готови шаблони като развити крайни продукти. Те мога да включват: медицински център, банково звено и други. Но за цялостна оценка по матрицата на Ансов трябва да се разгледат следните опорни точки за стратегията за същия пазар с нов продукт:

* Теми на пазара:

Темите на пазара могат да не са от най-стандартните на пазара. Една от тези не стандартни имплементации на системата за контрол на клиентите може да е болница за справяне с високо заразните отдели. Може да се имплементира системата в отделение за визити или за приемане на пациент.

* Отразяване на конкуренцията:

При отразяването на конкуренцията трябва да се вземе в впрочем процента заетост на конкуренцията в дела, в който желае да се подобри продукта. За продукта може да се каже че е не конкурентно способен спрямо строго специализирани програмни продукти, като на пример фирмата „Kiosk“, която са строго концентрирани върху билето-подаващите устройства.

* Ниво на границите за пробиване:

Нова граница за пробиване може да се разгледа централизиран опит за проникване на продукта скоростно или да се подходи пасивно, т.е. продукта да се наложи на пазара като заместител на по-скъпа и по-специализиран продукт.

* Иновативност:

Иновативност разглежда дали системата е достатъчно отделена от конкуренцията на сегашния пазар. Продукта с неговата гъвкавост е достатъчно отличим от конкуренцията.

* Раздялата:

С разделността на пазара може да се разгледа дали системата попада в такава демография където за момента има спад на пазара или цялостна липса на продукт. Пример: Закона за адекватно обслужване на клиенти в територията на република България. Така държавата задължава във всичките от нейните институции да има система за обработване на потока от клиенти.

* Ниво на посоката:

Разглежда се ниво на посоката като стандартите, които са нужни за да е конкурентно способна системата. Пример: Подържа ли се ред на опашките. Дали се поддържа приоритет (пререждане )на опашката. Дали е възможно прехвърлянето на часове от една дата на друга и други.

1. **Стратегия за диверсификация**

Разглеждането на диверсификацията на системата се подхваща от страна на новият пазар и новият или подобреният продукт.

* Темпо на растене:

Темпото на растене се разглежда от страна на конкурентността и от страна на продукта, т.е. водейки до един среден отчет дали решението за диверсификация е възможно или влиза на сферата на пасивите. Темп на растене не може да се оцени поради етапа на проекта до сегашния му период. Но е нужна оценка когато продукта се развие.

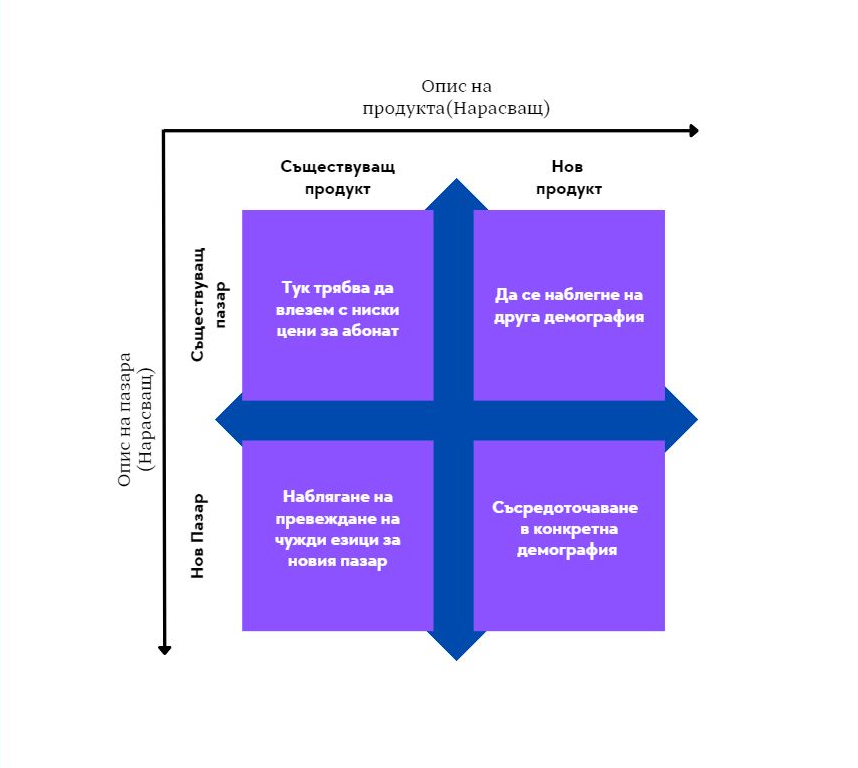
* Наличие на собствена компенсация:

Под наличие на собствена компетенция се разглежда адекватността за диверсификацията, т.е. може ли даденият отрасъл да не е подходящ за приложение на системата. Пример: Използване на системата за потвърждаване на автомобили за платени магистрали. Това е неприложимо заради липса на техника за потвърждение на регистрационни номера.

* Ниво на компетенцията:

Нивото на компанията е мярката за отклонение спрямо сходните продукти на пазара. Пример: Липсват на стандартите за незрящи клиенти на системата.

* Възможности на риска:

Възможности на риска е калкулацията на възможността за пробиване на продукта на нов пазар. Пример: Ако не се успее да се конфигурира системата за нов клиент може ли да се да се платят неустойките.

Фиг. 5. Реализацията на матрицата за конкретна ситуация

## Планиране на тактика за разработка на програмния продукт

След цялостна оценка и обзор на продукта като идея, т.е. да се реализира по адекватен, устойчив и бърз начин. За целта се предприемат следните точки:

* Реализация на база от данни според нуждите на алгоритмите за контрол на клиентите.

Базата от данни е ключова за гъвкавостта на система. А тя е организирана по такъв начин че да може да достави достатъчно функционалност на програмния код, който да се използва динамично и с висока скорост.

* Реализация на интерфейса за програмираното приложение.

Това са самите контроли, които приложени притежават, т.е. се организират контролите спрямо нуждите, които клиента има когато си настройва системата. Тук влиза и цялостната логика на системата, нейните алгоритми ( бизнес логиката) непряко свързани към крайната употреба. Реализацията на програмния интерфейс се случва като се отделят различни модули в различно слабо свързани нива, т.е. отделя се бизнес логиката от свързващите елементи. Така се гарантира че във времето един проект остава лесен за надграждане и поддръжка.

* Реализация потребителски интерфейс.

При имплементирането на интерфейса се подхожда по начин на симулиране на характеристиките, които клиента би искал да има при използването на системата. Така се гарантира най-високо ниво на съвместимост помежду потребителя и интерфейса.

Когато се следва подобен план, при който всички модул е зависим от интерфейса на следващия и са изолирани един от друг, се гарантира устойчивост и лекота при добавяне и тестване на нова функционалност. Модела, който най-добре описвайки тази стратегия е Model-view-controller къде съществува разделение помежду бизнес логиката и интерфейса на крайния потребител ( Фиг. 6.).

Картина, която съдържа текст, диаграма, скица, Шрифт

Описанието е генерирано автоматичноФиг. 6. Визуализация на метода на употребяваната стратегия за изработка на проекта.

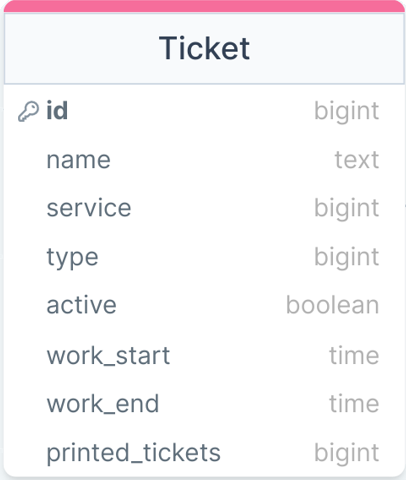
# Проектиране

Проектът се състои от три главни части: проектиране на база от данни, проектиране на интерфейса за програмираното приложение и проектиране на потребителския интерфейс.

## Разработка на база от данни.

Разработка на базата от данни започва със създаване първо на таблиците, които са от първа необходимост. Първите таблици са тези, които се явяват ядрото на цялата система. Започвайки с таблицата за билетите(Фиг. 7, Фиг. 12 таблица „Ticket“) в тази таблица са всички необходими редова за реализация на билет служещ за обслужване на клиенти. Редовете, от които е съчинена включват:

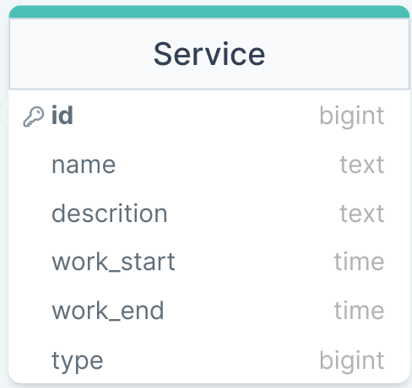
* Име( „name“) на билета, който служи за неговото наименование;
* Услуга( „service“) за която работи съответния билет и е реализирана под формата на релация от вида много към много;
* Тип на билета( „type“) които съществува поради различните възможности на имплементация на един билет. Типа на връзката много към едно;
* Активности( „active“) репрезентира дали система може да разпечатва дадения билет. Типа на реда от таблицата е булева.
* Започване на работа на билета( „work\_start“) е ред от таблицата, който репрезентира началото чрез час и минути на започвате в формат „час: минути“;
* Край на работа на билета( „work\_end“) е ред от таблица, който репрезентиран чрез час и минути на край на работата на билета в формат „час: минути“;
* Персонални билет ( „prited\_tickets“) служи за записване на всички персонални билети, който биват изтеглени от клиентите на системата.



Фиг. 7. Таблица от базата от данни отнасяща се за билетите.

След разглеждането на таблицата за билети се взема и в предвид, че даден билет е свързан с дадена услуга ( Фиг. 8, Фиг. 12 таблица „Service“). Тегленето се извършва точно по услуга. Услугата и билета са тясно свързани и подразбиране вървят винаги заедно. Но за да може да се постигни нивото на гъвкавост, което е планирано за системата, е нужна таблицата с услуги да е отделна от таблицата с билети. Когато има това разделение позволява да се комбинират множество билети към една услуга ( доказателство за работеща релация много билети една услуга фиг. 13). По този начин се осигурява гъвкавост на различен набор от варианти на системата. Редовете в таблицата включват:

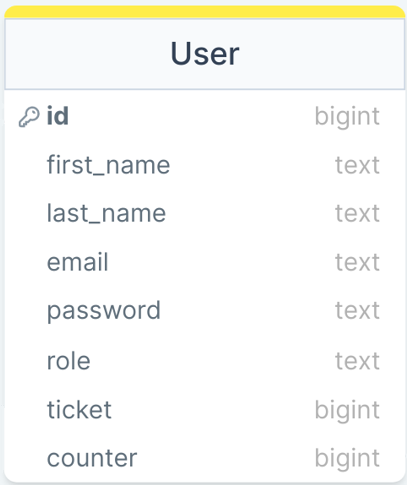
* Име ( „name“) на услугата, който служи за неговото наименование;
* Описание ( „description“) служи ча опис на дадената услуга. Вида на реда от таблицата е текст.
* Започване на работа на услугата ( „work\_start“) е ред от таблицата, който репрезентира началото чрез час и минути на започвате във формат „час: минути“. Тук добавя още едно ниво на гъвкавост когато има работно време и на билета както и на услугата.
* Край на работа на услугата ( „work\_end“) е ред от таблица, който репрезентиран чрез час и минути на край на работата на билета в формат „час: минути“.
* Тип на услугата („type“) е ред от таблицата, който служи за разглеждането на различните типове. Тези типове са възможни и да добавят допълнително усложнение към настройката на системата.



Фиг. 8. Таблицата отговаряща за услугите на системата в базата от данни.

Следващата ключова за разработката на проекта таблица, е тази на потребителя (Фиг. 9, Фиг. 12. таблица „User“). Поради естеството на таблица за потребители, таблицата служи за множествен набор от регистрации. Таблицата служи едновременно за потребители, работници, администратори и мениджъри. Поради чувствителността на данните, които таблицата съхранява, е взето в предвид и криптиране на ценната информация. Редовете на тази таблица включват:

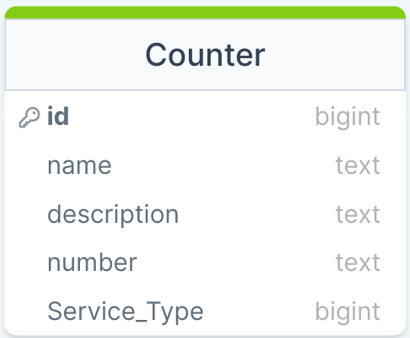
* Първото име на потребителя( „first\_name“) от типа текст;
* Фамилното име на потребителя( „last\_name“) отново от типа текст;
* Емайл адреса( „email“)на потребителя, който служи и за логване на потребителя. Реда е от тип текст;
* Паролата на потребителя( „password“), коя пристига криптирана и се декриптира от контролера. Реда е от тип текст;
* Ролята( „role“) на потребителя, която служи за различните функционалности на потребителя, както и се използва от филтъра за ограничаване на достъпа. Реда е от тип текст и когато се приеме от контролера ролята бива реализирана като енум[[14]](#footnote-14);
* Билетите които е обложил фигурират като релация от тип много към много за билета( „ticket“). Този ред служи за набор от различни статистики, от статистика за продуктивност до изкуствения интелект който решава дали някои остава на опашката.
* Гишета( „counter“) този ред съществува за определяне на потребителя на кои гишета може да се вписва, това е полезно поради различните квалификации и специалности на работниците.



Фиг. 9. Визуализация на таблицата за потребителите на системата в базата от данни.

Гишето, което представлява физическо или абстрактно такова, също така трябва да съществува в системата ( Фиг. 10, Фиг. 12. таблица „Counter“). Под физическо гише се разбира традиционната представа на гише, т.е. гише което е стационарно и клиентите пристига при него. Абстрактно гише, е гише което не спада под дефиницията на физическо. На пример едно абстрактно гише може да представлява охрана, която чекира билети, или кондуктор на влак. Идеята зад гишетата е да съществува абстрактно място където клиентите да пристигат за обслужване. Както и да се следи кой работник на кое гише работи. Това позволява и създаването на специализирани гишета, където могат само настроен набор от работница да работят. Това средство на организация, позволява на мениджърите на системата пряко да следят работниците и потока на работа. Обобщено едно гише може да служи и като стандартно гише ( физическо), както най-често се познава, или като нестандартно( абстрактно). Редове на таблица за гише включват:

* Първото име на гишето ( „name“) от типа текст;
* Описание ( „description “) който служи за кратък опис на функционалности на гишето и за неговото разположение ако това е от значение;
* Номера напишете ( „number“), който служи за допълнително нагласяне на системата. Защото може да има няколко гишета с името „каса“ и с различен номер;
* Сервизния тип ( „service\_type“) е различия, и служи за настройване на кои услуги може да обслужва гишето. Това е избрано за да се настройва по тип на услугата отколкото по самата услуга от гледна точка на бързата настройка. Защото може да има стотина услуги по един тип на услугите и така администратора му се улеснява настройването на системата.



Фиг. 10. Визуализация на таблицата за гише на системата в базата от данни.

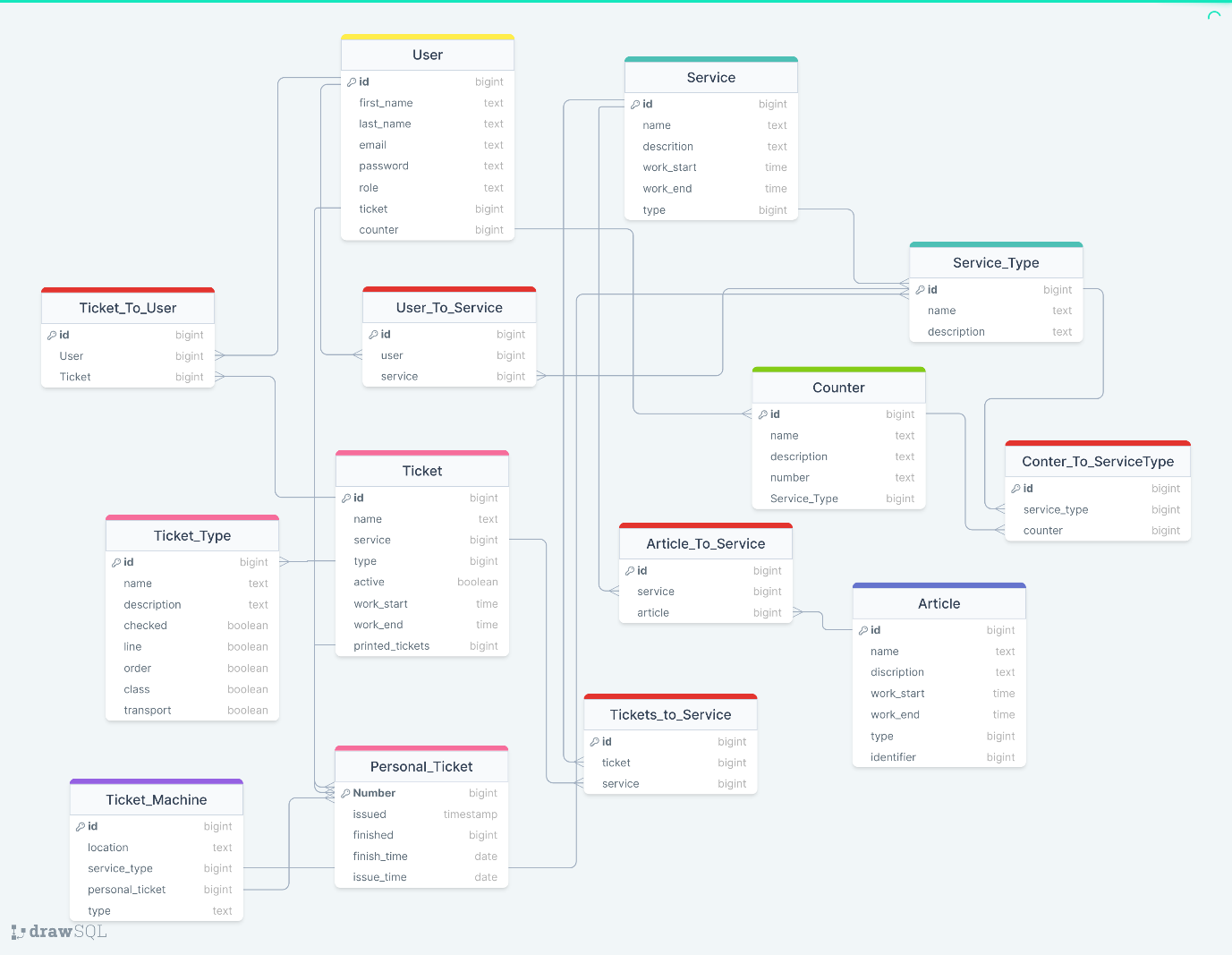
Относно главните елементи, които са полезни за системата, остава таблицата на артикулите. Тази таблица служи за добавяне на малки не обосновани предмети или услуги към даден билет( Фиг. 11, Фиг. 12. таблица „Article“). Това описание остава абстрактно, тъй като служи точно за елементи, който не са част от главните услуги. За пример може да се вземи ресторант, в който всяко ястие се регистрира като отделен артикул към услугата за предоставяне на ястия. Така клиента пристига за услугата да се нахрани и се начисляват хранителните продукти към билета на услугата. Така се показва силата на начисляване на допълнения към услуга и нейните редове включват:

* Първо е име на артикула ( „name“) от типа текст;
* Описание ( „description “), който служи за кратък опис на артикула;
* Започване на работа на артикула ( „work\_start“) е ред от таблицата, който може да служи за ограничаваме от началния часа. Формата е „час: минути“;
* Край на работа на билета ( „work\_end“) е ред от таблица, който може да служи за ограничаване на работния час на кухня на ресторант като пример. Формат на данните е „час: минути“;
* Реда за тип на артикула( „type“) служи главно за сортиране и подреждане на различните типове. Реда в таблицата е индексиран за по-бързо търсене.
* Идентификационния( „identifier“) ред е различен от реда на идентификационния на един запис в таблицата. Този ред служи само и единствено когато даден артикул има собствен номер за идентификация, често срещан пример бива бар код на продукт или номер за инвентаризация.



Фиг. 11. Таблицата за артикули на системата в базата от данни.

Относно базата от данни в нейната цялост са реализирани множество много към много връзки. Това е така за да бъде успешно реализирана гъвкавостта на приложението( Фиг. 12). Тази гъвкавост за да се докаже трябва да се реализират примерни данни, но такива данни които да симулират истинско прилагане на система за обслужване на клиенти( Фиг. 13).



Фиг. 12. База от данни и нейните релации

|  | Кино | Ресторант | Община | Влак | увеселителен парк |
| --- | --- | --- | --- | --- | --- |
| Service | | | Service | | | |
| name | Зала №3 | Хранене | Акт за Раждане | Варна-София | Въртелешка |
| description | Зала за прожекций | Сервиране на храна  до масата | Свидетелство за  новородените | ... | ... |
| work\_start | 15:30 h | 11:30 h | 8:00 h | 8:10 h | 8:20 h |
| work\_end | 23:30 h | 23:30 h | 17:00 h | 8:45 h | 22:10 h |
| type | Прожекция | Сервиране | Актове | Бърз влак | Увеселително влакче |
| Service\_Type | | | Service\_Type | | | |
| name | Прожекция | Сервиране | Актове | Бърз влак | Увеселително влакче |
| description | Прожектиране на филм | Да се вземе поръчат от  кухнята и да се занесе на клиента | Всякаккъв документ  който се връчва спрямо дадена ситуаци | Блак който е бърз | Тип развлечение  за малки и големи |
| user | Иван, Йордан... | Иванка, Петър... | Йордан, Иван... | Кодукторите и  расиерите | Иван, Йордан... |
| Articke | | | Articke | | | |
| name | Coca-Cola | Боб по селски | Свидетелство от  болницата | Място | Снимка |
| description | Газирана напитка | ... | Документ нужен  за извършване на услугата | Място на влака | Снимка на клиента  докато се забавлява |
| work\_start | 8:30 h | 10:30 h | 0:00 h | na | na |
| work\_end | 22:30 h | 12:40 h | 0:00 h | na | na |
| type | Наливна Напитка | Яхния | Свидетелство | Резервация | Сувенир |
| indentifier | barcode | barcode | idcode | number-of-seat | barcode |
| Counter | | | Counter | | | |
| name | Вход А | Маса В | Гише А | Гише А | Гише А |
| description | От дясна страна на залата | Външните маси | ... | ... | ... |
| number | 1 | 3 | 4 | 8 | 1 |
| service\_type | Зала 3 | Хранене, Напитки... | Актове | Всички | Бъртелешки, VIP |
| Ticket | | | Ticket | | | |
| name | Аватар | Маса В3 | Акт за Раждане | София-Варна | Дневен-VIP |
| description | Филм за извънземни | Билет за сервизната  на масата | Получаване акт за  новородено | ... | Дава възможната да се качваш на всички без да чакаш опашката |
| work\_start | 14:30 h | 10:30 h | 8:10 h | 8:55 h | 9:10 h |
| work\_end | 23:30 h | 17:30 h | 16:50 h | 15:00 h | 22:00 h |
| active | ✔️ | ✔️ | ✔️ | ✔️ | ✔️ |
| personal | 1500 | 3400 | 5000 | 100000 | 12000 |
| type | Entrance | Table order | Queue | Transport | VIP |
| Ticket\_Type | | | Ticket\_Type | | | |
| name | Entrance | Table order | Queue | Transport | VIP |
| description | За билети които  трябва да се проверят преди влизане | Билет, които има множество артиколи които ще се сервират на клиент | Тип билет за, които се чака на опашка за да може да се достъпи гише за обслужване | Билети който служат за транспорти нужди | Билета дава  възможности да се возиш на всички влакчета цял ден без да чакаш на опашка |
| checked | ✔️ | ✖️ | ✖️ | ✔️ | ✔️ |
| base/line | ✖️ | ✔️ | ✔️ | ✖️ | ✖️ |
| order | ✖️ | ✔️ | ✖️ | ✖️ | ✖️ |
| class | ✖️ | ✖️ | ✖️ | ✔️ | ✔️ |
| transport | ✖️ | ✖️ | ✖️ | ✔️ | ✖️ |
| Personal\_Ticket | | | Personal\_Ticket | | | |
| number | 1499 | 3399 | 4560 | 930334 | 152 |
| issued | 14:45 h | 11:35 h | 14:45 h | 9:30 h | 23:59 h |
| finished | ✔️ | ✔️ | ✔️ | ✔️ | ✔️ |
| User | | | User | | | |
| first\_name | Иван | Иванка | Ахмед | Пламен | Иван |
| last\_name | Иванов | Иванова | Ахмедов | Илиев | Асен |
| email | [Iivanov@Kino.com](mailto:Iivanov@Kino.com) | [Iivanova@Restorant.com](mailto:Iivanova@Restorant.com) | [Aahmed@varna.com](mailto:Aahmed@varna.com) | [Piliev@bdj.bg](mailto:Piliev@bdj.bg) | [Iasen@park.com](mailto:Iasen@park.com) |
| password | alabala453 | 012345678 | 12345678 | 12345679 | 12345678 |
| role | user | user | user | user | user |
| service\_type | Прожекция | Сервиране | Актове | Кондуктор | Продавачка билети |
| personal\_ticket | 540 | 10000 | 6000 | 6001 | 7005 |
| Tickete Machine | | | Tickete Machine | | | |
| type | Киоск принтер с плащане | Визуализатор | Киоск принтер | Турникет | Киоск принтер с плащане |
| location | На касата на входа | В главната кухня | Пред входа | Пред входа на станцията | Помежду влакче 1,2,3 |
| servicr\_type | Прожекция | Хранене | Административни, и др. | Преди перона | Въртелешка, Влакче, и др. |
| tickets | 940 | 540578 | 100123 | 120264 | 54003 |
| Обеснение  по рплите на машините | Тук машината служи за  принтиране на билетчета спрямо избора на клиента | В този случай  машината служи да визуализира поръчките на готвачите | В този случай машината  служи единствено за разпределение на опашката за | В този случай машината служи за предварително чекиране на входа и не допуска друг. | Тук отново машината се  ползва за закупуване директно от нея. |

Фиг. 13. Примерни данни за различни имплементации на системата за обслужване на клиенти.

## Реализация на интерфейса за програмното приложение

Реализацията на интерфейс за програмното приложение започва с концепция на какво е нужно, за да може да се изпълни бързо и лесно цялостното му проектиране. При реализация на интерфейс на програмния проект първо трябва да се обмисли цялостно от колко модула зависи технологията и те как се използват за разработката му. Второ трябва да се проектира спрямо заложените технологии и алгоритми, т.е. спазвайки всички конвенции за обектното ориентиране програмиране. Трето и последно е да се тества за да може да се потвърди функцията на програмния код.

### Обзор на нужните технологии и алгоритми

Тази под-глава е концентрирана върху теориите за алгоритмите и технологиите по, които са имплементирани. Теорията на опашките[[15]](#footnote-15), наричана още и теория на опашките за изчакване ( Queueing theory), е първия алгоритъм, който е заложен за цялостната реализация на системата. Теорията на опашките е математически алгоритъм, който изисква постъпването и обработката на сигнали да следи даден баланс. Тя се прилага за моделиране и анализиране на системи, в които се изпълняват някакви процеси със задачи. Където постъпват по-определен начин, тази задача, и се обработва и извлича от системата, така че да доведе до опростяване на процеса като цяло. Точно така алгоритъма служи за всеки индивидуален клиент най-ефективно да се справя с потока от клиенти.

Основните компоненти на теорията на опашките включват следните елементи:

* Постъпление на заявки: Това е процесът на постъпване на заявките в системата с опашка. Постъпването може да бъде случайно или според определено разпределение;
* Обработка: След като заявката постъпи в опашката, тя трябва да бъде обработена. Времето за обработка може да бъде фиксирано или случайно;
* Опашка: Заявките, които не могат да бъдат обработени веднага, се нареждат в опашка, където чакат своя ред за обработка;
* Излизане: След като заявката бъде обработена, тя напуска системата. Времето за излизане може също да бъде фиксирано или случайно.

Теорията на опашките анализира различни параметри на системата, като средното време за престой в опашката, средното време за престой в системата, производителността на системата, вероятностите за определени събития и други. Това позволява на икономисти и управители да проучват и оптимизират работата на системата с клиенти. За настройване на ефективната за работа с клиентите, опашката от клиенти трябва да се настрои спрямо спецификите на фирмата. Това е така за да се гарантира, че за всяка услуга има адекватен обслужващ персонал, т.е. чрез имплементация на теорията на опаките. Менажерът чрез методите представени от системата за контрол на клиентите предостави баланса помежду клиентите и възможността всички да бъде обслужени навреме. За да се приложи алгоритъма адекватно се налага да се разгледа как клиент би използвал системата. Така показвайки какъв формат са билетите и какви услуги на какви гишета се обслужват.

Първи пример, който е демонстриран, е за най-голямата демография за продукта( Фиг. 14). Това представлява институция предлагащи административни услуги ( общини, здравни каси, агенцията по вписванията, контрол на автомобилния транспорт и др.). Симулирайки сценария се извеждат заключението, че действията предприемани от клиента са малко. Най-често клиента идва да си свърши една работа, която му е належаща, и ако системата е претоварена би дошъл в друг ден. Демонстрацията е подкрепена от множество проучване и статистика разгледани за проекта.

Картина, която съдържа рисунка, скица, дизайн, илюстрация

Описанието е генерирано автоматично

Фиг. 14. Демонстрация на имплементация тип административни нужди

Втората демонстрация, за приложение на теорията на опашките(Фиг. 15.), е по-нестандартен, но отново е приложим към теорията на опашките. Демонстрацията представлява системата за контрол на клиентите и тя как е употребена за различни кино салони. Така в този сценарий билетите биват закупувани за различни филми, които е настройват като услуги на системата. Ролята, която е настроена като гише, играе охраната на салона, която проверява и чекира билетите в системата. В най-идеалния случай едно кино използвайки системата не би имала нужда и от проверяващ персонал а само от билето-подаващо устройство за закупуване на билет. След което те се чекира от автоматичен турникет[[16]](#footnote-16) на входа на киното.

Картина, която съдържа скица, рисунка, диаграма, дизайн

Описанието е генерирано автоматично

Фиг. 15. Демонстрация за имплементация тип кино

Последната разгледана демонстрация е пример, в който системата за контрол се употребява от заведение за хранене ( Фиг. 16). В тази демонстрация билетите са индивидуалните поръчки на клиентите, а гишето на което отиват поръчките е района ( групата от маси на един сервитьор). По този начин се спазва идеята за теорията на опашките. В тази демонстрация влизат и артикулите, споменати в предишната глава. Артикулите се използват за различните напитки и хранителни блюда.

Картина, която съдържа скица, рисунка, текст, анимирана рисунка

Описанието е генерирано автоматично

Фиг. 16. Пример за имплементация тип обект за хранене.

Елемент вложен в разработката на системата, за контрол на клиентите, е систематизиран начин за предоставяне на данни. Така може да се докаже нагледно как системата забързва работния процес на фирмата клиент. Но не всяка институция закупила системата разбира как тя работи. Затова е заложено системата сама да се оценява, спрямо статистическите извличания по време на работа. Но да разполага с множество данни за билетите и клиенти не е достатъчно да се докаже че една система е оптимизирана. Затова е избран модела на Дюпон за оценка на ръста на работа.

Моделът на Дюпон[10,11,12] (DuPont model) е финансов модел, който се използва за анализ на приходите на компанията и определяне на факторите, които влияят върху рентабилността и. Този модел е създаден от американската компания DuPont Corporation през 1920-те години и е използван от тях за вътрешни анализи и управление на предприятието. Моделът на Дюпон разделя рентабилността на компанията на три компонента: марж на печалба, оборот на активите и финансово задействане. Той изразява следната формула:

Рентабилност = (Марж на печалба) × (Оборот на активите) × (Финансово задействане)

Където:

* Марж на печалба (Profit Margin) - отразява способността на компанията да генерира печалба спрямо приходите. Тя се изчислява като деление на чистата печалба на компанията ( нетния доход) на приходите от продажби.
* Оборот на активите (Asset Turnover) - показва ефективността на компанията в използването на своите активи. Той се изчислява като деление на приходите от продажби на компанията на активите й.
* Финансово задействане (Financial Leverage) - отразява степента, в която компанията използва заемен капитал вместо собствен капитал. Той се изчислява като деление на средствата на компанията, финансирани с дълг, на собствения й капитал.

Като се анализира всеки от тези компоненти, моделът на Дюпон може да помогне на управленските екипи да идентифицират и фокусират усилията си върху областите, които могат да подобрят рентабилността на компанията.

Третият и последен метод заложен за осъществяване на система, за контрол на клиентите, е начин за предсказване на това дали човек от опашката би останал на опашката. Водейки се по извършеното проучване, статистиката гласи че множество от загубите на фирмата идват от клиенти, които не изчакват реда си на опашка. В система където може да се проследи в реално време колко е средното време на обслужване и колко души са на опашката. Тези данни са от висока стойност, тъй като може да се извлече ценна информация от тях.

Ако се изведе тази информация на менажерите те могат да формулира решения дали е достатъчен набора отворени гишета и обслужващ персонал. Или да се остави системата сама да си преценява за в момента колко гишета и колко работници да са на разположение автоматично. Благодарение на машинното обучение това вече е възможно. Машинното обучение позволява да се вгради алгоритъм, който вместо човешко същество да предвижда натоварване и разтоварване спрямо индикатори в работния процес.

За реализация на предвиждането на товара в опашката е избран алгоритъма „Дърво на решенията“ защото най-добре описва стъпките, през които човешкия мозък взема решение в подобен случай. Алгоритъмът на дървото на решенията (Decision Tree) е метод за машинно обучение, които се използва за задачите на класификация и регресия. Той представлява дървовидна структура, където всеки възел в дървото представлява тест върху някои от признаците на данните, а всеки клон от дървото отговаря на възможни резултати от този тест. Листата на дървото съответстват на класове или стойности на целевата променлива в регресията. Основните стъпки в алгоритъма на дървото на решенията включват:

* Избор на признак: В началото на алгоритъма трябва да се избере признак, който да се използва за тест в кореновия възел на дървото. Има различни методи за избор на признак, като например информационни мерки (като ентропията или при ръста на информация) или Gini[[17]](#footnote-17) коефициент.
* Разделяне на данните: Данните се разделят въз основа на стойностите на избрания признак. Всеки отделен клон в дървото отговаря на една стойност за признака.
* Рекурсивно построяване на дървото: За всеки подмножество от данни, получено след разделянето, алгоритъмът се изпълнява рекурсивно, като избира нов признак за следващия възел и продължава да разделя данните.
* Определяне на листата: Процесът на построяване на дървото продължава, докато не се достигне критерий за спиране. Например, ако всички елементи в дадено подмножество са от един и същ клас или ако са използвани всички признаци. В такъв случай се определя листо в дървото, което съдържа класовете или стойностите на целевата променлива.
* Класификация и регресия: След като дървото е построено, то може да се използва за класификация.

За целта на заданието е планирано да се разгледа алгоритъма само чрез математическите му формули без вмъкване на чужди библиотеки. Вмъкването на чужди библиотеки крие множество рисковете. Тези рискове вкупчват: спиране на поддръжка, умислено промяна на кода за да се измъкнат чувствителни данни, точност на алгоритмите. Когато е реализиран някой алгоритъм сам по себе си той може да се променя и доусъвършенстване както е подходящо в бъдещето.

Преди да се започне работа с дървото трябва да му се подберат атрибутите по които вземат решенията. Тези атрибути трябва да са свързани пряко с потребителя, тъй като целта е да се симулира крайния потребител как да постъпи в различни сценарии.

Public record Attribute() {

public static final String FINISHED = "FINISHED";

public static final String TICKETID = "TICKETID";

public static final String TIMEISSUED = "TIMEISSUED";

public static final String ACTIVETICKETS = "ACTIVETICKETS";

public static final String OPENDCOUNTERS = "OPENDCOUNTERS";

}

Всичките преценени за нужни категории атрибути биват извадени в публичен запис в константи. Избрано е да се използва запис отколкото клас тъй като едни запис е по изчистена форма на класа и компилатора работи по-бързо с него. Като и записите в java се възприемат като клас само съдържащ информация, т.е. записа( record) служи само и единствено за контейнер на стойности.

public class Node {

private String label;

private boolean isLeaf;

private List<Condition> conditions;

public Node(){

conditions = new ArrayList<>();

isLeaf = false;

}

…

Разглеждайки класа за възела в дървото, той главно служи за реализация дали дадено решение е край( leaf) или продължава( condition). След като се знае как изглежда класа за възлите може да се употреби в главния клас за реализация на дървото ( „DecisionThree“). При писането на този алгоритъм единственото правило нарушено е принципа за единствената отговорност( Single responsibility principle), като се оставя твърде много отговорности на класа за реализация на дървото. В него е заложена валидация за крайни случаи, като на пример ако извадката е грешна по някой логически принцип. Логически принцип е и заложен за ентропия на дървото, която отговаря за възможните разклонения на едно решение. В алгоритъма е и приложен класификатор на решения е вграден в класа. Това решение предприето за нарушаване на този важен принцип на обектното ориентирано програмиране, е от гледна точка неговата имплементация. При писането на подобни сложни алгоритми винаги трябва да се спазват принципите на обектното ориентирано програмиране, защото при тяхното пренебрегване води до синтезиран код без бъгове. Тези граници представляват средата в които би бил прилаган този алгоритъм. В този случай се знае каква е крайната цел и нейното приложение. Средата за прилагане на алгоритъма позволява за по-съкратено и не толкова усложнено разписване на алгоритъма. Кода в неговата цялост може да се разгледа в приложението на дипломната работата.

Картина, която съдържа текст, диаграма, Паралелен, Шрифт

Описанието е генерирано автоматично

Фиг. 17. Клас диаграма на логиката на дървото на решенията.

### Проектиране на програмния код

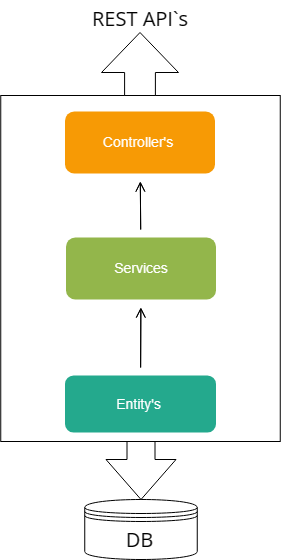
След като се знае какви са алгоритмите заложени за реализация на проекта и се знае как изглежда базата, може да се премине към реализацията на програмния код. Нужни са всички предходни стъпки да бъдат положени преди да се обмисли структурата на програмния код, защото се разглеждат крайни случаи и множество фактори. Тези крайни случай служат за демонстриране на възможните проблеми, който могат рано или късно претърпява промяна и/или цялостна преработка. Преди полагането на кода трябва да се вземе и в предвид че продукта минава през мащабиране, поддръжка, добавянето на функции и др. Затова се преминава през период на усилено планиране, където се мисли и действа с абстракции и се доказват с конкретни случаи. Това е един от най-силните методи за гарантиране на устойчивост на едно приложение.

Структурата която е използвана е от многослоен тип. Тоест имаме множество слоеве, които са слабо свързани и капсулирани за устойчива работа на приложението. Или така наречения принцип на обръщане на зависимостите, която гласи че трябва да се програмира спрямо интерфейса на слоя а не неговата имплементация. Приложението набляга на представяне на състоянието на трансфер ( така наречената Representational state transfer REST ) методология, която е приета глобално за репрезентиране на заявки и отговори във уеб екосистемата ( Фиг. 18.). Използвайки REST служа и за лесно откриване на проблеми, когато те могат да бклучват: Лесно се откриват грешки защото, когато една заявка не съответства тя връща адекватен код за грешката на проблема. Както и ако приложението премине към друг потребителски интерфейс, то директно може да се свърже към контролите от сървъра без да има нужда от пренаписване или добавяне на други контроли.

Контроли в този случай е използван spring терминологията за название на граничещ обект( border object). В класическото разглеждане на термина означаващ контролен обект, се има предвид обект с вградена бизнес логика. За целта на обяснение на системата, се назовават термините на контроли вместо граничещи обекти. И вместо контролни обекти се назовават като сервизи или сервизни обекти. Размяната на названия е често срещан проблем когато се говори за spring еко системата. Затова всички обяснения по програмния код използват spring названията.

Контролите са полезни за множество случаи. Като на пример ако приложението реши да премине към мобилен интерфейс. Новия интерфейс може да се свърже само с контролите на сървъра без да се променя нищо по бизнес логиката на приложението. Тоест REST е предназначено точно за проекти, които се нуждаят от един достъп до ресурси. Където тези ресурси са, бизнес логика, база от данни и комуникационни потоци. Самата методология подържа идеята за много слоен подход към разработката на контролерите, които се свързват към потребителския интерфейс.

Затова е прието да се подходи по три слоен метод за разработка на приложението. Където контролера извършва работата да приема заявки, както и контролера връща определени сигнали обратно. Последван от обслужващия слой, който служи за бизнес логика и извикване на базовите обекти. Тези базовите обекти служат за съдържане на данни от базата на данни. И последно е връзката към базата от данни и класовете, от който се създават различните обекти за бизнес логиката. Когато има многослоен подход може да се покрият всички точки на (Single Responsibility Principle (SRP), Open/Closed Principle ,Liskov’s Substitution Principle (LSP), Interface, Segregation Principle (ISP), Dependency Inversion Principle (DIP) - SOLID) модела.



Фиг. 18. Диаграма на приложената методология при разработване на системата контрол на клиентите

1. **Базов слой на програмния код**

Първите стъпки предприети са към базовите класове, с които се съхранява информация за различните обекти. Тези обекти са строго свързани към таблиците от базата от данни. Или така наречените „Entity“ класове. Това са най-простите класове в структура. В „Entity“ класове не е заложено да има логика, защото стратегията за разработване гласи, че бизнес логиката изпълнява тази дейност. Тази стратегия се прилага за да доведе до лесен контрол и лесна промяна при нужда. Както и покриват принципа за сегрегация на интерфейсите. Но не е забранено да има логика на това ниво, тъй като повечето стратегии при обектното ориентирано програмиране са заложели по логически начин. Като например да се състави клас от това ниво с модела на строителя[[18]](#footnote-18) ( builder pattern). Това се прави за да се използват силните страни на ООП. Метода за свързване на тези класове е строго свързан с релационната база от данни, защото при използване на java-Spring съчетанието можем да се имплементира инструмента Hibernate. Този инструмент служи за свързване на таблици от базата от данни към класове ( Фиг. 19). Това се прави за да се работи лесно с множеството обекти, които биват създадени по тази методология. Hibernate не ограничен да се използва само в съчетание с Spring, той си е независима структура от библиотеки наречени зависимост( dependency).

Тези класове и техните зависимости се попълват по подобен начин като в релационна база от данни, но вече в кода се отбелязват връзките помежду класовете с тагове. Тези тагове отново следват конвенцията на релационните бази от данни и и могат да се напишат по-следния начин:

@Table(name="Ticket")

public class Ticket {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

@ManyToMany(fetch = FetchType.LAZY,

cascade=CascadeType.ALL)

@JoinTable(

name = "favor\_tickets",

joinColumns = @JoinColumn(name = "ticket\_id"),

inverseJoinColumns = @JoinColumn(name = "favor\_id"))

private Set<Favor> favor;

@OneToMany(fetch = FetchType.LAZY,

cascade = CascadeType.ALL)

@JoinColumn(name = "ticket\_type\_id")

private Set<TicketType> type;…..

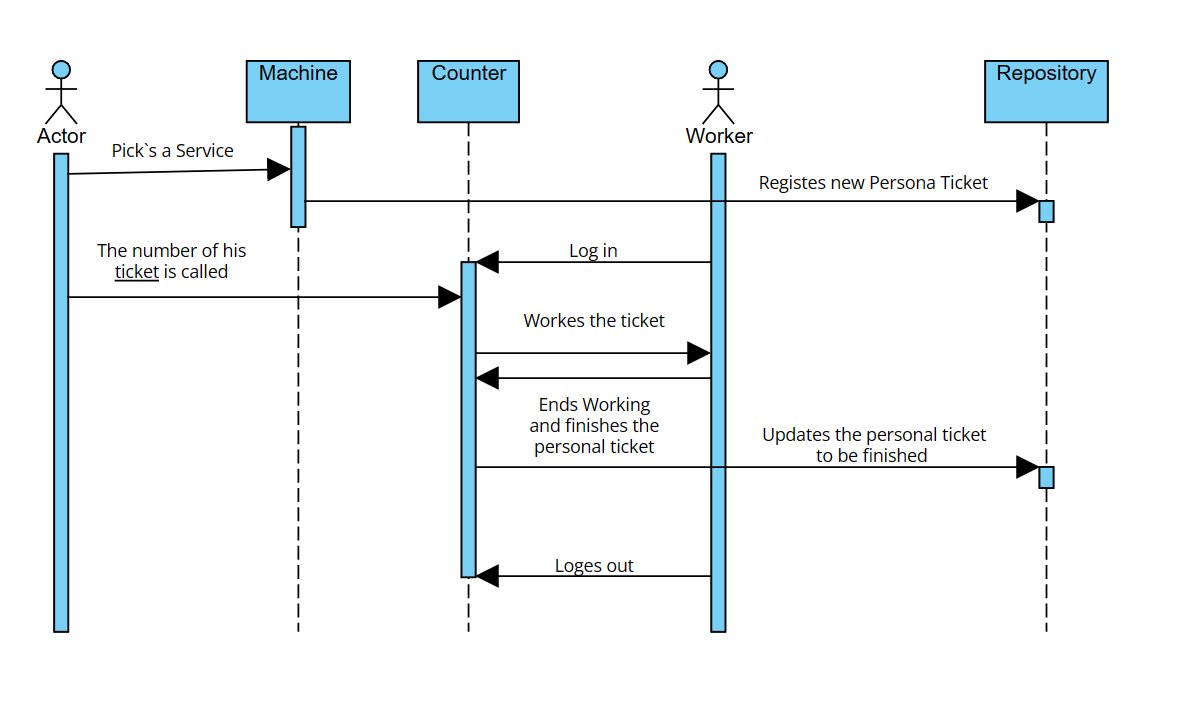
Картина, която съдържа текст, диаграма, План, Паралелен

Описанието е генерирано автоматично

Фиг. 19. Клас диаграма на базовите класове, които отговарят за съдържане на информация.

1. **Бизнес слой на програмния код**

След базовото ниво на обектите служейки за данни следва нивото на бизнес логиката. На това ниво се реализира реалната логика към опашката и как тя работи. Главната логика на задвижване на цялото приложение бъде имплементира на това ниво и тя включва: логиката за опашката ( Фиг. 20), логиката за филтриране на реда ( Фиг. 21), логиката за визуализация на чакащите и логиката за администриране на системата ( Фиг. 22). И на това ниво има връзки но те се свеждат там където биват нужни и се отбягва да има множество зависимости в едни клас(Фиг. 23).



Фиг. 20. Диаграма на последователност представляваща логиката зад обслужване на клиент чрез системата за контрол.

Относно логиката за клиента как бъде обслужван от системата следдва следната логика. Първо клиента на иституцията, която работи със ситемата, реши за коя услуга желае да бъде обслужен то тогава му се разпичата съотвентия билет за дадената услуга. Този билет представлява или физичен или виртуален и на него се разписва: дата и час на издаване, дата и час на годност, избранта услуга, хората чакащи на тази опашка. Избрано е да не му се разписва на клиента колко гишета в момента обслужват тази услуга тъй като би довело до по-висок набор от клиенти напускаяи опашката преди отреденият им ред.

Когато работник постъпи на работа след негово логване, той може да започне работа по съотвентия набор от услуги зададени за гишето. Тоест колкото услуги обслужва гишето толкова различни опашки от услуги се нареждат на него. Така реда на изпълнение на билетите не е по услуга а по ред на изтеглене на билети. Това е допостимо за голям набор от институции но някой имат така наречените приоритетни услуги, които има значение кой за какво е на опашката. Затова когато има приоритен билет той предрежда всички чакащи на опашката поради естеството на билета.

Относно метода, по пойто се викат билети на гишета е следния. Когато служителя е готов за работа с опашката той започва да извиква билети. Тук може да възникне проблем при извикването на билети, когато двама работници на еднакво настроени гишета с еднакво настроени услуги извикат билет. Затова е изградена синхронизация на извикването на билети. След като билет бива извикан от работника за да се яви на гишето той бива записан в базата от данни като извикан. Но дадения билет не е отбелязан като приключил. Един билет бива приключен след като се извика следващ. Тъй като служителя може да прицени че не е за съответната опашка или не е заплатил нужни такси. Така клиента с един билет може да се прехвърли на друга опашка. Това позволява на клиента да бъде препращан помежду гишета без да има нужда да тегли нови билети и да чака отново на опашки. Когато билет бива прехвърлен той е с предимство пред всички други ново-изтеглени билети. Това е преценено за нужно, защото се предполага че клиента си е изчакал времето за чакане и ако отиде отново най-отзад на опашката, се предполага че няма да си изчака реда би напуснал или по-лошо би създал неприятности за работниците.

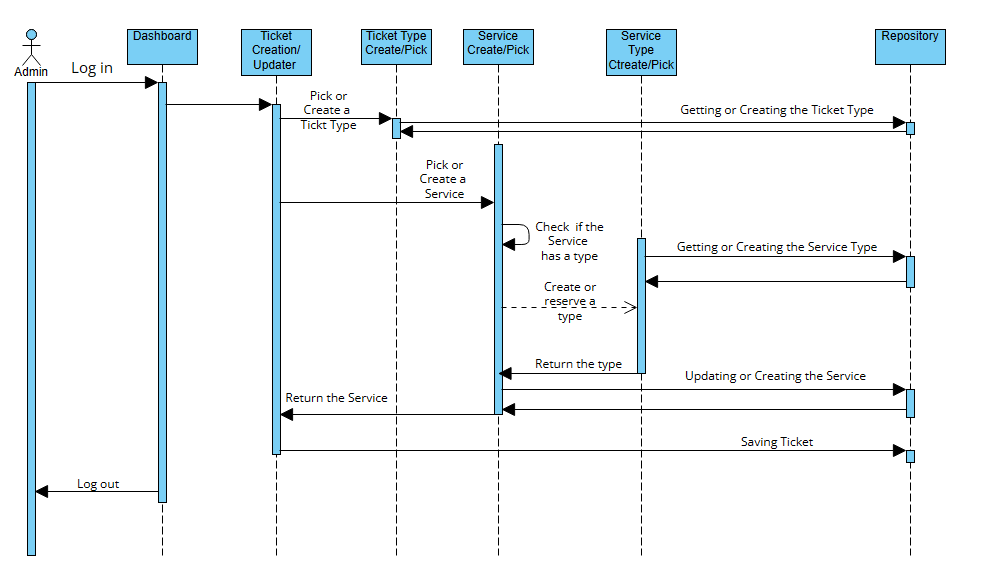
Картина, която съдържа диаграма, екранна снимка, линия, текст

Описанието е генерирано автоматично

Фиг. 21. Логиката как един билет при изтегляне и съответно извикване попада на правилното място.

Логиката относно филтрирането на системата за контрол на клиентите и техния ред се осъществява по следния начин. Когато има множество билети и множество души на опашка и те чакат за едни и същи гишета ( Фиг. 21). Тогава когато работник натисне че е готов да обслужва следващия клиент то системата първо проверява за кои услуги(Фиг. 21 „Favor“) е настроено гишето. След което за кои билети се отнасят за тези услуги, тъй като може да има билет с множество услуги, и на края проверява чрез филтрация кой билет е следващ за това гише. Имайки предвид че филтрацията е синхронизирана и няма да позволи грешка при извикването. Тук може и да се наблюдава двуслойни неврона връзка. Само когато се настроен системата по препоръчителния начин. Тоест има много услуги към много билети, и те се репрезентира като входа на неврона а персоналните билети ( Фиг, 21 „Personal Ticket“) е изхода на неврона. А метода на филтрация е следващата стъпка след неврона. Тоест когато се търси кой билет за кое гише е отново имаме множество входове с един изход. Подобно нива на сложност, която се постига с малко на брой настройки, водят до гъвкавостта на приложението. Затова може да се използва една система едновременно за кино, за ресторант и за административно звено като община и други подобни.

Най-важните аспекти, които са ключови за реализацията за системата и нейната гъвкавост, се свеждат до нивото на приоритет на билета. Модела позволява и за бъдещи допълнения, като например да се вмъкнат и резервационни билети. Това е така защото е спазен принципа на open/close при писането на модела за системата. Когато се остави адекватно място за разработка и разширяване чрез наследяване. Се поддържа подобни добавяне на модули. Но този принцип може бързо да доведе и до разширяване процеса на разработка, или така нареченото „over-engineering“. Ако вземем за пример разширяването на сегашна система и към нея добавим резервационен модел. Които представлява наследи на билета с повече функционалност. То тестовете се удвояват, защото вече имаме имплементация с един билет и кога се вкара втора имплантация трябва да се тества всички вариации с новата функционалност.



Фиг. 22. Диаграма на последователност представляваща логиката за администриране на системата за контрол на клиентите

Администрацията на системата е едно от нещата, които са заложени за потребителска свобода при използването на системата. Така клиента закупил системата може да оптимизира обслужването на клиентите си. Ако се подходи правилно при настройката може да доведе до оптимизиране на множество елемента от фирмата едновременно. Но за да гарантираме гъвкавост и висок пазарен дял, трябва системата да има множество стъпки за избор при администриране. Така се среща дилема за колко е най-малкия брой стъпки, който да гарантират едновременно леснота на администрация и гъвкавост на видовете обслужване на клиенти.

Решението предприето от страна на административната логика е тя да бъде напълно покрита, т.е. контролите, които могат да бъдат допуснати чрез заявки от потребителя, да бъдат максимално изчерпателни докато потребителския интерфейс да направи процеса на настройка на системата по-поносим и бъз за администраторите на системата. Диаграмата за последователност на фигура 22 се отнася към всички нужни заявки към сървъра за добавяне на нов вид билет. А лекотата и самото потребителско ( администраторско) преживяване да се настрой от потребителския интерфейс.

Самите връзки помежду класовете на това нива са сведени до минимално. Но когато се вземе в предвид че и главната логика на опашката е в това ниво. Класът „QueueService“( Фиг .23) отговарящ за множество от логиката в системата и е свързан с нужните за реализацията на логиката при обслужване на клиентите на опашката. Но при разглеждането на връзки с други нива на системата можем също да открием връзки за успешното осъществяване на логиката на опашката. На това ниво се намират и повечето функции, който се използват за работа с билетите и включват функции за:

* Теглене на билети по услуга;
* Визуализация на билети;
* Извикване на следващ клиент на опашката;
* Визуализация на услугите.

Тези фикции са подготвени да бъдат лесно приети от следващото ниво над тях. Тоест параметрите, който връщат, са структурирани или като базови класове или като класове данни ( payloads[[19]](#footnote-19)) за по-лесна работа от контролера. Така се спазва принципа за единствена отговорност. Защото текущото ниво зависи само от интерфейса на предишното. И своевременно класовете също така отговарят на принципа на единствената отговорност. Защото отговорностите са отделени по темата, които обслужват и администрира. Покрит е и принципа за инверсия на зависимостите, тъй като следващото ниво зависи само от интерфейса на функционалностите.

Картина, която съдържа текст, Шрифт, Паралелен, черно и бяло

Описанието е генерирано автоматично

Фиг. 23. Клас диаграма на всичките сервизи, който са употребени в разработката на системата за контрол на клиентите.

1. **Контролен слой на програмния код**

Контролното, или по-широко разпространен термин „restful“, ниво е директната връзка с потребителския интерфейс и представлява връзката, през която всеки един желаещ да комуникира със системата за контрол на клиентите би преминал. Връзката със сървъра става, след като той бива стартиран и инициализиран на даден домейн. След това за всеки контролер си има адрес ( Uniform Resource Locator-URL) на който се приемат и изпращат всички заявки към сървъра. Тези заявки спрямо адреса могат да се формулират по множествено начини, но за разработката на проекта е прието като стандарта за заявки да е пос следния начрун:

Https:// име на модула/ версия на котролера/ име на пакета/ име на контролера/ променлива ако има такава

Пост ( Post) заявките служат за регистриране на нещо ново, тоест да се добави към системата. В контекста на системата този вид заявки служат главно при администрирането на системата, там където се създават нови услуги, билети, потребители и др. При пост заявките е нужно и да има тяло ( body[[20]](#footnote-20)) на заявката и то да бъде в стандарта на JSON (JavaScript Object Natation). Този вид тяло на заявката е стандартно. JSON тяло се използва и при отговор на сървъра. Всички пост заявки от страна на администриране на системата се нуждаят от автентикация.

Гет ( Get) заявките служат за вземане на конкретна информация от сървъра и са отново стандартизирани. В проекта се използват главно за да се визуализира информация от базата относно системата. Те не се нуждаят от тяло на заявката но често получават списък от желания обект, като JSON тяло на отговора на заявката. Всички гет заявки от страна на администриране на системата се нуждаят от автентикация.

Пут ( Put) заявки и заявки за изтриване ( Delete) съществуват и при администрацията, и при обслужването на опашката на системата. Пут заявките служат за поправяне на обект или да се добави нещо към този обект. Като за пример пут заявки са използвани когато даден клиенти е обслужен и му се приключва билета. За да се оформи пут заявка най-често има тяло и член променлива в адреса на заявката. Това е широко приета практика и се спазва в проекта. Пример за пут заявка:

localhost:8080/api/v2/counter/add/ticket/:favorId/:counterid

Сигурността на системата се осъществява чрез вградените инструменти от Java spring технологията и конкретни зависимата библиотека „Spring security“. Която позволява имплементирането на множество филтри. Филтрите служат за ограничаване на достъпа и регулиране на работа на системата. Филтрите, който са осъществени за поетка включват:

* Филтър за разграничение помежду работник и администратор ( по роля);
* Филтър за JWT (JSON Web Token) автентикация;
* Настройка за предаване на информация през няколко източника (Cross-origin resource sharing CORS);

Филтрирането чрез токени е често срещани и широко използвано. В токените може да се слага множество информация, която е кодирана, и директно свързана с информацията в user базовия клас.

Когато се на напишат всичките контроли те трябва да се документират как всеки един от тях работи. Затова е изведено в детайл как всеки един от тях работи.

Употреба за всички контролери към системата

Автентикация.

Служи за всички елементи в системата свързани с административна работа.

POST /api/vi/auth/authenticate

Заявката се нуждае от тяло от вида:

email - String

password - String

Примерно тяло:

{

"email":"ivoAdmin@mail.com",

"password":"12345678"

}

Отговора от тази заявка се е съдържанието на JWT токена. Времетраенето на токена е 12 часа.

Артикул

GET /api/v2/article

Връща лист от артиколи, като отговор от заявката. Нуждае се от автентикация.

POST /api/v2/article

Позволява ти да създадеш нов артикул. Нуждае се от Json тяло с характеристиките:

name- Long - Requerd

description- String - Requerd

workStart - SQLTime HH:mm:ss format -Requerd

wrokEnd - SQlTime HH:mm:ss format -Requerd

type - String -Requerd

Примерно тяло:

{

"name": "name",

"description":"some description",

"workStart":"08:30:00",

"workEnd":"15:30:00",

"type“: „type"

}

Връща съобщение съдържащо идентификационен номер на ново създадения артикул.

PUT /api/v2/article/:articleId

Позволява да се променя артикул, който вече е бил създаден. Нужно е JSON тяло.

name- Long - Requerd

description- String - Requerd

workStart - SQLTime HH:mm:ss format -Requerd

wrokEnd - SQlTime HH:mm:ss format -Requerd

type - String -Requerd

Примерно тяло:

{

"name": "name",

"description": "some description",

"workStart":"08:30:00",

"workEnd":"15:30:00",

"type": "type"

}

Връща съобщение съдържащо идентификационен номер на артикула.

PUT /addFavor/:articleId/:favorId

Връща идентификационен номер на довявания артикул към услугата.

DELETE /api/v2/article/:articleId

Гишета ( Counter)

GET /api/v2/counter

Връща лист от гишета, като отговор от заявката. Нуждае се от автентикация.

POST api/v2/counter

Позволява да се създаде ново гише. Нуждае се от Json тяло с характеристиките:

name - String - Required

discription -String - Required

number - Integer – Required

Примерно тяло:

{

"name":"counte1",

"description":"Some description",

"number":1

}

Връща идентификационен номер на създаденото гише.

PATCH api/v2/counter/:counterId

Позволява да се променя гише, което вече е било създадено. Нужно е JSON тяло.

name - String - Required

discription -String - Required

number - Integer – Required

Примерно тяло:

{

"name":"counte1",

"description":"Some description",

"number":1

}

Връща съобщение съдържащо идентификационен номер на ново създаденото гише.

PUT api/v2/counter/add/ticket/:favorId/:counterId

За да се добави нова услуга към гишето.

DELETE /api/v2/counter/:counterId

Услуги ( Favor)

GET api/v2/favor

Връща лист от услуги, като отговор от заявката. Нуждае се от автентикация.

POST api/v2/favor

Позволява да създадеш нова услуга. Нуждае се от Json тяло с характеристиките:

name - String - Required

discription -String - Required

workStart - SQLTime HH:mm:ss format - Requerd

wrokEnd - SQlTime HH:mm:ss format - Requerd

idsOfTypeOfFavors - Integer Array - Requerd

Примерно тяло:

{

"name":"AdminWork",

"description":"some description",

"workStart":"07:00:00",

"workEnd":"19:00:00",

"idsOfTypeOfFavors":[1]

}

Връща идентификационен номер на новата услуга като съобщение.

PATCH api/v2/favor/:favorName

Позволява да се променя услугата, който вече е бил създаден. Нужно е JSON тяло.

name - String - Required

discription -String - Required

workStart - SQLTime HH:mm:ss format - Requerd

wrokEnd - SQlTime HH:mm:ss format - Requerd

idsOfTypeOfFavors - Integer Array - Requerd

Примерно тяло:

{

"name":"AdminWork",

"description":"some description",

"workStart":"07:00:00",

"workEnd":"19:00:00",

"idsOfTypeOfFavors":[1]

}

Връща съобщение съдържащо идентификационен номер на услугата.

DELETE /api/v2/favor/:favorId

Машини ( Machine)

GET api/v1/machine

Връща лист от машини, като отговор от заявката. Нуждае се от автентикация.

POST api/v1/machine

Позволява да създадеш нова машина. Нуждае се от Json тяло с характеристиките:

name - String - Required

type -String - Required

favorId -Long - Required

Примерно тяло:

{

"name": "Roller",

"type": "Gate",

"favorId":1

}

Връща идентификационен номер като съобщение на новосъздадената машина.

PATCH api/v1/machine/:machineName

Позволява да се промени машина, който вече е била създадена. Нужно е JSON тяло.

name - String - Required

type -String - Required

favorId -Long - Required

Примерно тяло:

{

"name": "Roller",

"type": "Gate",

"favorId":1

}

Връща съобщение съдържащо идентификационен номер на машината.

DELETE /api/v2/machine/:machineId

Персонален билет ( Personal Ticket)

GET api/v2/personalticket

Връща лист от персонални билети, като отговор от заявката. Нуждае се от автентикация.

GET api/v2/personalticket/:ticketId

Връща точно определен персонален билет според зададен идентификационен номер, като отговор от заявката. Нуждае се от автентикация.

POST api/v2/personalticket

Позволява да създадеш нов персонален билет. Нуждае се от Json тяло с характеристиките:

issueTime - SQLTime HH:mm:ss format - Requerd

Примерно тяло:

{

"issueTime":"12:00:00"

}

Връща съобщение съдържащо идентификационен номер на ново създадения персонален билет.

PATCH api/v2/personalticket/:personalTicketName

Позволява да се променя персонален билет, който вече е бил създаден. Нужно е JSON тяло.

name - String - Required

type -String - Required

favorId -Long - Required

Примерно тяло:

{

"name": "Roller",

"type": "Gate",

"favorId":1

}

Връща съобщение съдържащо идентификационен номер на персоналния билет.

PUT api/v2/finish/:ticketNumber

Приключва персонален билет спрямо работещия потребител.

DELETE /api/v2/personalticket/:personalTicketId

Билет ( Ticket)

GET api/v2/ticket

Връща лист от билети, като отговор от заявката. Нуждае се от автентикация.

GET api/v2/ticket/:favorId

Връща списък съдържащ всички билети от зададения идентификационен номер на услугата, който са настроени с открития по идентификационен номер на билети.

POST api/v2/ticket

Позволява да създадеш нова билети. Нуждае се от Json тяло с характеристиките:

name - String - Requerd

workStart- SQLTime HH:mm:ss format - Requerd

workEnd - SQLTime HH:mm:ss format - Requerd

favorId - Long - Requerd

typeId - Long -Requerd

Примерно тяло:

{

"name": "ticketExample",

"workStart":"07:00:01",

"workEnd":"19:00:02",

"favorId":1,

"typeId":1

}

Връща съобщение съдържащо идентификационен номер на билета.

PATCH api/v2/ticket/:ticketName

Позволява да се променя на билет, който вече е бил създаден. Нужно е JSON тяло.

name - String - Requerd

workStart- SQLTime HH:mm:ss format - Requerd

workEnd - SQLTime HH:mm:ss format - Requerd

favorId - Long - Requerd

typeId - Long -Requerd

Примерно тяло:

{

"name": "ticketExample",

"workStart":"07:00:01",

"workEnd":"19:00:02",

"favorId":1,

"typeId":1

}

Връща съобщение съдържащо идентификационен номер на билета.

PUT addFavor/:idTicket/:idFavor

Adds a favor to the ticket. Request authentication.

PUT addFavor/:idTicket/:idPersonalTicket

Adds a personal ticket to the ticket. Requirs authentication.

DELETE /api/v2/ticket/:ticketId

Тип на билета( Ticket Type)

GET api/v2/tickettype

Връща лист от типове билети, като отговор от заявката. Нуждае се от автентикация.

POST api/v2/tickettype

Позволява да създадеш нов тип билет. Нуждае се от Json тяло с характеристиките:

name - String - Requerd

description - String - Requerd

Примерно тяло:

{

"name": " Administration",

"description": "some description"

}

Връща съобщение съдържащо идентификационен номер на тип на билет.

PATCH api/v2/tickettype/:ticketTypeName

Позволява да се променя тип на билета, който вече е бил създаден. Нужно е JSON тяло.

name - String - Requerd

description - String - Requerd

Примерно тяло:

{

"name":"Administration",

"description":"some description"

}

Връща съобщение съдържащо идентификационен номер на тип на билет.

DELETE /api/v2/tickettype/:ticketTypeId

Потребител ( User)

GET /api/v1/user

Връща лист от потребители, като отговор от заявката. Нуждае се от автентикация.

POST /api/v1/user/register

Позволява да създадеш нов потребител. Нуждае се от Json тяло с характеристиките:

firstname - String - Requerd

lastname - String -Requerd

email - String - Requerd

role - String - Requerd

password- String -Requerd

Примерно тяло:

{

"firstname": "Ivo",

"lastname": "Rumenov",

"email":"ivoUser@mail.com",

"role": "USER",

"password":"12345678"

}

Връща съобщение съдържащо идентификационен номер на тип на билет.

PUT /api/v1/user/register/addCounter/:counterId

Позволява ти да добавиш ново гише за работа на потребител.

DELETE /api/v1/user/:userId

Опашка (Queue)

GET api/v1/queue/nextInLine/:counterId

Връща лист от билетите чакащи на опашка, като отговор от заявката. Нуждае се от автентикация.

PUT api/v1/queue/open/counter/:counterId

Отваря гише.

DELETE api/v1/queue/close/counter/:counterId

Затваря гише.

GET api/v1/queue/waitingForCounter/:counterId

Връща всички персонални билети за това гише.

Теглене ( Draft)

Автентикация за този контролер не е нужен.

GET api/v1/draft/favor

Връща списък със всички услуги.

Get api/v1/draft/waitingForCounter/:counterId

Връща всички активни персонални билети чакащи за обозначеното гише.

GET api/v1/draft/lastPersonalTicket

Връща последния изтеглен персонален билет.

POST api/v1/draft/:ticketId

Създава но персонален билет спрямо избора на клиент от потребителския интерфейс.

id - Long- Not Requerd

issueTime - SQLTime HH:mm:ss format - Requerd

Примерно тяло:

{

"id":123123,

"issueTime":"12:00:00"

}

Картина, която съдържа текст, Шрифт, разписка, Паралелен

Описанието е генерирано автоматично

Фиг. .24. Клас диаграма на всички контролери.

### Тестване на системата

Индивидуалното тестването на елементите от системата за контрол на клиентите и тестване в дълбочина са нужни, тъй като е заложено множествено функционалност и гъвкавост при множеството от ситуации. Затова е зададено и осигурено цялостност на тестовете и те включват: единични тестове ( unit tests), интеграционни тестване (integration tests), от край до край тестване (end-to-end testing), тестове при разгъванe (deployment testing), функционални тестове ( functional testing), тестване на производителността ( performance testing), тестове при взаимодействие с базата от данни. Тази поредица от тестове е високо ценена и употребяема в практиката при разработване на приложения.

Единичните тестове представляват такива тестове, който служат за тестване на малки части от логиката на кода, т.е. най-често се извършва от програмиста, които е написал кода първоначално. Това се прилага така защото кода, който е написал може да няма нищи общо с бизнес логиката но да играе роля при цялостната употреба на приложението. Този вид тестове са ключови, защото те трябва да се покрият в множество сценарии с множество параметри. Така гарантират че с продължението на писане на код то тези класове и функции, който се тестват, са устойчиви и на мащабируемостта. Единични тестове могат да се извършват по начин, по който да помогнат за цялостното тестване на системата. Тестовете извършени по този начин, по системата, са се вградили постепенно през всяка нова итерация в функционални и вървят заедно с интеграционните тестове. Множественото индивидуални тестове извършени върху проекта, започват от базовите класове, последвани от тестове за бизнес логиката.

Функционални тестове представляват вид тестове, които служат за до етапа на приложените промени по кода. Тоест те са променят за всеки цикъл ( спринт), през който преминава програмния продукт. Те служат за валидация на кога преди да бъде добавен към работещата версия. Както и да се измери работната скорост и успеваемост на проекта. Тук се тестват едновременно фундаментални аспекти на програмата. Както и се тестват функции на програмата, които играят роля на допълнения и не са градивна част от ядрото. Но тези тестове са нужни за цялостната завършеност на продукта. Функционалните тестове по проекта са извършени в рамките на няколко дни за да се провери как различни елементи от системата работят заедно.

Интеграционните тестове представляват тестове на индивидуалните нива, като на пример тестване само на заявки към базата от данни. Този вид тест служи за да гаранция че дадения слой от проекта може да работи самостоятелно ако му се подават правилните данни. Така се потвърждава устойчивостта на кода, който се пише на всяко едно ниво. Ако стане проблем по време на интеграционен тест, които може да се сведе до това че или не е изпратен правилен сигнал или че сигнала не е подходящ за сценария. Наблегнато е върху интеграционните тестове в проекта поради метода и идеологията, която е избрана за неговата разработка. Естеството на слоевете на програмата гарантират че има множество интеграционни тестове. За системата са извършени интеграционни тестове на нивата за: заявки към базата от данни, към сервизния слой на системата (Фиг. 23), към контролерите на системата (Фиг. 24) и към потребителския интерфейс. Този вид тестване може да покри до 97% от функционалния код на един клас. При писането на такъм вид тестване трябва да се има в предвид как едни клас и неговите функции реагират към набор от правилни и неправилни сигнали. Така се постига високо покритие на кога и неговите редове.

Край до край тестване представляват тези тестове, които тестват едно приложение от неговото начало до неговия край, или така нареченото тестване по сценарии. В този вид тестване се поглежда с една идея как крайния потребител би използвал приложението, т.е. гледат се множество сценарии, по които се употребява системата от край до край. Когато се изпълняват подобен вид тестване се разбира и как крайния потребител би имал проблем с цялостното потребителски изживяване. За тази цел е използван програмата „Postman“, която служи като инструмент за директно тестване на крайните връзки на контролера на сървъра. Поредността от действия играе голяма роля при писането на подобни тестове, тъй като се предполага че и клиента следва тази логика при използването на системата. Поредността им е заложена като фундаментална за употребата на приложението ( Фиг. 25). Видовете край до край тестове извършени по системата за контрол на клиентите са:

* Тест с гише с множество услуги

Тук се гледа система, която има еднакви услуги, който отиват на еднакви гишета, т.е. се представят нови услуги но се обслужват от същи персонал.

* Тест с множество услуги и множество гишета препокривайки се

Тук има гишета с различен набор настроени услуги който могат да бъдат обслужени но се препокриват с други гишета, който имам сходни но не и пълния набор от сходни услуги (Фиг. 26).

* Тест с множество услуги и множество гишета без препокриване

Тук по подобие на предишния тест имаме множество гишета и множество услуги но те не се покриват по никакъв начин.

Картина, която съдържа текст, екранна снимка, софтуер, Мултимедиен софтуер

Описанието е генерирано автоматично

Фиг. 25. Изображението на настройката на автоматични тестове преди тяхното стартиране в опреден ред.

Картина, която съдържа текст, екранна снимка, Шрифт, номер

Описанието е генерирано автоматично

Фиг. 26. Изображение на автоматичен тест, който включва инициализация на система с множество гишета и множество услуги тествано как обслужва с множество билети.

## Реализация на потребителски интерфейс

Разработката на потребителския интерфейс е ключова за демонстрация на успешен проект. Потребителския интерфейс е първото и последното нещо, което потребителя вижда когато борави с приложението. Тоест изисква се наблягане върху фините части на човешкото взаимодействие с програмния продукт. Когато се работи по потребителски интерфейс трябва да си отговори на няколко въпроса:

* За кого правим интерфейса;
* Каква работа ще извършва интерфейса;
* Как ще контактува човека с интерфейса;
* Кои принципи на дизайна ще са нужни за реализацията на интерфейса.

Когато се отговарят всичките тези въпроси и се приложат върху дадени персони за част от система. Започва да се изгражда потребителското изживяване (User experience-UX). Това относно как човека, като цяло се използва системата за контрол на клиентите. Употребата на приложението може да бъде в аспекта на работник или на клиенти на системата. Когато правилно се помисли как човек общува със системата това води само до по-бърза употреба на системата ( user efficiency) от всички страни. (Jerry Cao, Ben Gramillion, Matt Ellies, 2015)

Започвайки дизайна на потребителския е взето в предвид и начина на имплементиране на програмен код, т.е. разработка с JavaScript и html. Когато се има и това в предвид може да се започне да се разглеждат сценариите на интерфейса. След като си знае интерфейса на контролера как изглежда и как да го достъпи. Така се добива свобода на креативността при дизайна на потребителския интерфейс. Следва да се открие клиентите, които употребяват системата. Това е постижимо чрез употребата на персони[[21]](#footnote-21).

Първата персона, която се разглежда е тази на типичен работник на гише (Фиг. 27). При имплементирането на тази персона е приложен принцип че човека употребяващ системата е средно мотивиран да работи но с над средна грамотност с технологиите. В този случай персоната притежава динамичен интерфейс където може да вика номера на билети и ги обслужва подобаващо. Тоест от гледна точна на дизайн трябва да има ясени шрифтове на текста, както и леки за очите цветове.

Картина, която съдържа текст, компютър, човек, екранна снимка

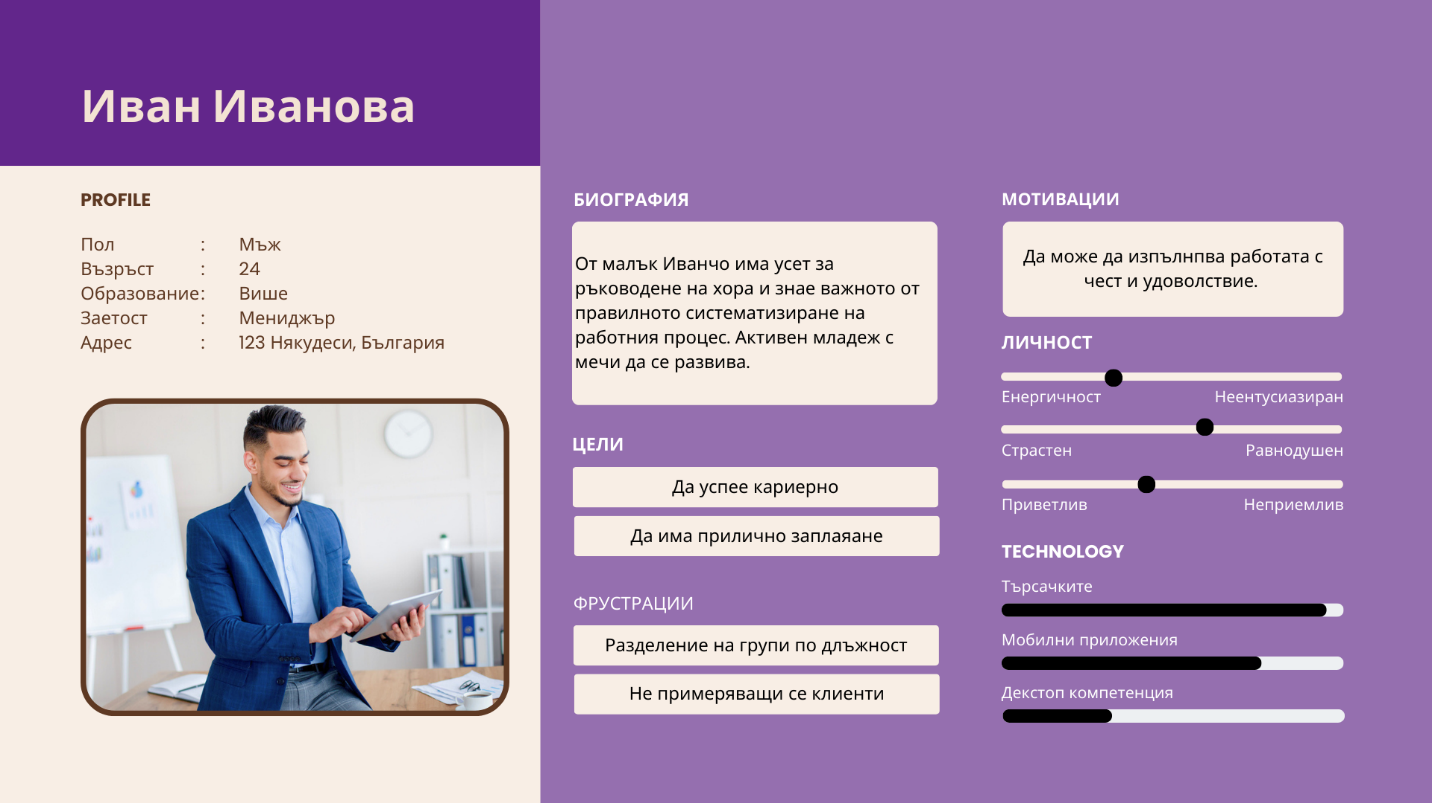
Описанието е генерирано автоматичноФиг. 27 Персона на работничка на системата за контрол на клиентите.

Картина, която съдържа текст, екранна снимка, човек, Уебсайт

Описанието е генерирано автоматичноСледващата персона, която е разгледана, е на касиер (Фиг. 28). Подобна на персоната на работник на гише тази персона представлява работник, които използва системата за викане на клиенти и съответно обложвайки ги. Но в този случай поради възрастта на служителя се вземат на предвид технологическата компетентност, която е ниска. Тук е взето и решението за опресняване на начина на извикване на клиенти към гишето.

Фиг. 28. Персона на работничка на гише тип каса.

Следващата персона, която се разглежда е от страна на оперативния персонал или така наречената организация на контрола ( Фиг. 29). Ролята на мениджъра е ключова за реализация на теорията на опашките, която гласи че трябва се постигне баланс помежду потока от работа и обслужващия персонал. Тоест преценката за персоната е критична. Тя трябва да репрезентира примерен персонал в ролята на мениджър. За това е прието тя да е на мъж в началото на своята кариера, поради нужната квалификация за изпълняването на ролята. Персоната е високо мотивирана и техническо компетентна. Тоест можем да се очаква да се възползва от статистиките в тяхната цялост и за това да се наблегне върху тяхната разновидност. Характерно за тази работна е и следенето на персонала, т.е. можем да се изведи че е желателно да се добави колко работни часове отделя един работни за даден билет и колко билета обработва за една негова смяна. От там персоната може да си редактира и до настройва опашката така че да е най-ефективна. От тази персона също може да се изведе полезния факт че може да се следи опашката от дистанция чрез мобилно устройство, т.е. потребителят на интерфейса е гъвка и четим от различен набор устройства а не само от компютър. Под гъвкав интерфейс се има в предвид, интерфейс който се мащабира и побира в рамките на множество различни среди. Тези среди могат да включват телефони, таблети, лаптопи, телевизори и др. Също така е заложено да се добави модела на Дюпон заради неговата популярност в менажирането и организацията на контрола. Така само от една персона се извеждат много изисквания за интерфейс на програмния продукт.



Фиг. 29. Персона на мениджър.

Последна персона, която е преценено че е от нужда е на оператор, който върши двойна роля. Това е точно работник тип охрана, който едновременно проверява билети и следи за реда ( Фиг. 30). Подобна е ролята на кондуктор на влак или автобус но поради пазарния дял, който цели програмния продукт, е преценено да се изработи персона за охрана заради нейната разпространеност в пазарния дял. Представяната персона на охранител е пример за чекиращо гише където работника е мобилен и трябва едновременно да проверява билети и да следи за реда. Тази ситуация показа че потребителския интерфейс трябва да бъде лесно четим и разбираем на пръв прочит. Лесното разбиране и четене на един потребителски интерфейс е изграден от: четими шрифтове, белите пространства и цветна координация. Тоест че за персона като охранителя не му е нужен много функционалност а бърза рабат. За това е подбрано когато даден работник е на гише, което чекира билети, то да сигнализира за редовен билет в зелено а за грешен, изтекъл или нередовен билет да сигнализира в червено. Метода за чекиране може да е по различни видове дали по номер на билет или баркод на билета. Тоест при получаване на билет, дали се закупува или просто изтеглен, то той отговаря за реда и контрола на опашка и след това някой като собственика или мениджъра може да погледне статистиките към дадените маркирани билети. Взето в предвид и техническата грамотност на персоната така че да е подходяща за бърза и устойчива работа.

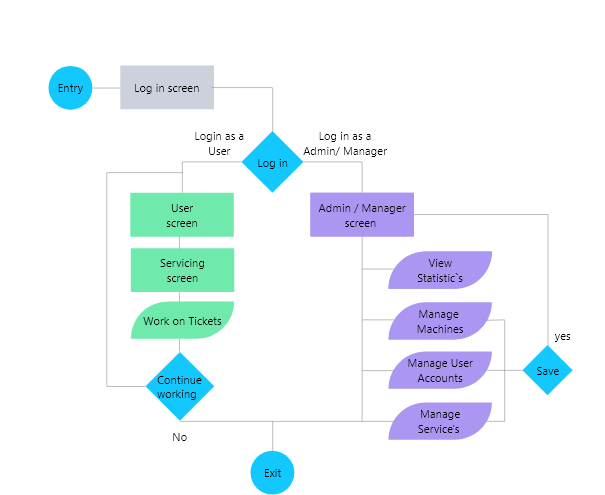


Фиг. 30. Персона представяща охрана на обект и контрол на опашката.

След вземане на предвид широката гама от персони, които са разгледани, можем да се приложат при конкретни функционалности както и тематични забележки към потребителския интерфейс. Едно от решенията, които е взето да не се разглеждат конкретни персони за клиентите на системата за контрол. Поради естеството на системата се подразбира че клиентите са от всички различни прослойки на обществото и с различни предпочитания. Тук предпочитанията не играят роля, тъй като те по един или друг начин използват системата за да достъпът услугата които желаят. Но това не е знак да се предприемат дръзки решения за дизайна от страна на клиента. А даже да се подходи по метод, които масовото общество разбират. Това включва и управление за незрящи, като звукова и допирна сигнализация. Така един метод за прилагане на системата за незрящи може да се съобщава звуково когато се извиква билет и някои билети да имат дупки в тях така че да са разчети ми чрез Брайловата[[22]](#footnote-22) азбука.

Относно дизайна на физичните билетите те включват дата и час на издаване, име на услугата, кои гишета ги обслужват и колко души има пред тях. За метода на разпечатване на билети, закупилия системата има избор да ползва какъвто си поиска принтер за разпечатването на физични билети. Системата автоматично взема зададения принтер по подразбиране за разпечатване. А за допълнителни настойки може да се достъпи чрез административния пален на системата.

От другата страна има и виртуалните билети, които са достъпни чрез уеб страница, където са изписани всички услуги на системата така че клиента да може да подбере желаната услуга и да си изтегли билет. След тегленето на билет клиента е препратен на уеб страница където изписва номера н билета и чакащите в реално време. Това позволява на клиента да може да следи реда на билетите без да бъде физически там. Когато билета бива извикан клиента бива информиран че му е реда да се яви на гишето, което го е извикало. Както и ако клиента присъства на място ако е настроено информативен дисплей, които казан на глас от информативния дисплей. Информативния дисплей представлява екран ( телевизор или уеб страница), които е настроен да показва реда и на кои гише те биват викани.

Преди осъществяването нужните уеб страници, които се разработени, първо се преглежда реда в които те са употребени. Вземайки персоните в предвид се разглежда реда по следния начин (Фиг, 31). Първо разглеждайки от страната на работния персонал и административния. След логване в системата за контрол на клиентите спрямо ролята те са прехвърляни или към административния панел или към даденото гише на което работят. Ако потребителя е клиент той бива преведен към гишето което е предназначено. От там може да обслужва различните билети настроени за гишето, ако има изтеглени такива. Но ако потребителя е администратор той е преправен към страницата за редактиране на елементите на опашката ката: гишета, услуги, артикули, машини ( ако има такива) и статистиката по системата. След като се знаем всичко това може да се премине към разглеждането на страната отговорна за теглене на билети.

Фиг. 31. Диаграма на потока за администратор или работник.

Относно потока на клиента, който служи за теглене на билет по услуга и съответно изчакване и следене на реда на чакащи за гишета (Фиг. 32). Тази част от системата е най-линейната от всичките. Тук клиента или си изчаква реда или си тръгва. Но когато се помисли по всички други елементи на системата този поток на клиента е най-лесен. Така когато е планирано това преживяване да е оптимизирано за бързина и минимална объркващия за клиента, се очаква само позитивни резултати от употребата му.

Картина, която съдържа текст, екранна снимка, диаграма, Шрифт

Описанието е генерирано автоматично

Фиг. 32. Диаграма на потока за теглене на физични и виртуални билети.

Анализа, които е извършен по персоните и те как навигират приложението, е от ценно значение при синтезирането на дизайна на потребителския интерфейс. Инструмента, който е използван за изработка на схемата на интерфейса преди да се започне неговото програмиране е Adobe Xd. Решението да се използва, това приложение за дизайн, е прието поради неговата гъвкавост и употреба на пазара и в обществото на дизайнерите. При синтезиране на схемите е започната от преценяване на цветовата гама и шрифтовете, които се използват в приложението от гледна точка на консистентност. За избора на цветове е подходящо те да са в по-студената гама, затова е избрано постелено синьо за доминиращ цвят съчетано с бяло за празните пространства и на края бебешко синьо за сигналните елементи ( Фиг. 33). Подбран е модела 60-30-10 поради неговата утвърдена репутация в сферата на потребителските интерфейси. Модела представлява гама от три цвята, като 60% от единия цвят е доминантен в мащаба на интерфейса, а 30% са използвани за предаване на промяна или различие по между елементите в интерфейса, и последно 10% са използвани за сигнални елементи там където е желателно се да привлече вниманието на потребителя. Видове сигнални елементи в един потребителски интерфейс биват бутони, прогрес барове и сходни елементи.



Фиг. 33. Гамата от цветове използвани за разработка на потребителския интерфейс.

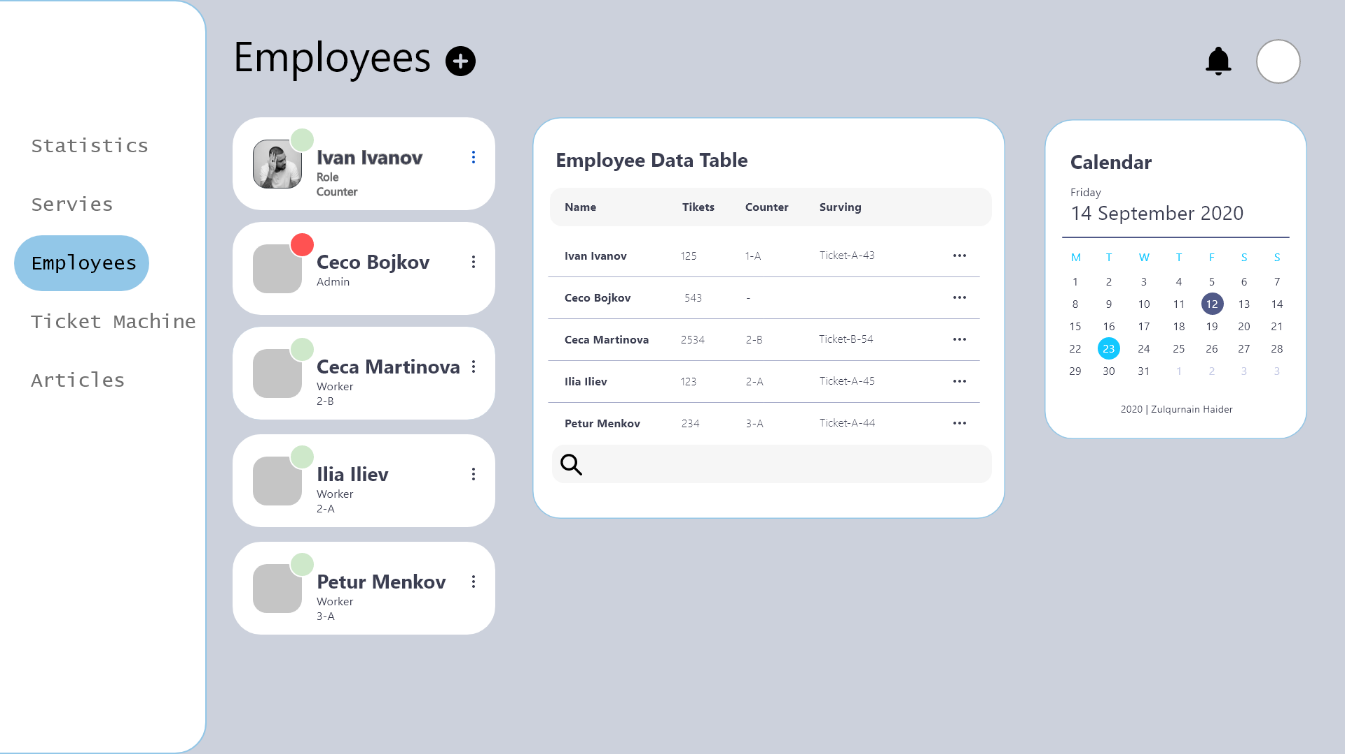
След избора на цветове се преминава към избора на подходящи шрифтове ( Фиг. 34). Първия „Nirmala UI“ се използва в чисто черен формат, като заглавия на страници и опции в менютата. Избран е зареди неговия брутализъм[[23]](#footnote-23), което води до бързо разчитане и открояване на различните символи един от друг. Следващия шрифт „Lucida console“ е комбиниран със сив цвят за да се открои от черния шрифт споменат преди, както и да се покаже допълнение в моменти където дадена информация е второстепенна към главна в черно. Последния шрифт „Bilo“ е съчетан с главния шрифт, там където е нужно да се покаже тънка разлика. Тънка разлика може да се покаже, като например различните видове статистически графики ( Фиг.35) или при изброяването на множеството работници ( Фиг. 36.). Представени са и множество от дизайна в приложната част на разработката. Там са представени идеите за цялостната визия на потребителския интерфейс. По този начин може да се стръвни как е пренесено синхронизацията на всичките уеб страници за цялостна консистентност на интерфейса.

Фиг. 34. Използваните шрифтови семейства с техния размер и цветове.

Картина, която съдържа текст, Шрифт, екранна снимка, дизайн

Описанието е генерирано автоматично

Фиг. 35. Статистика от дизайна на потребителския интерфейс.



Фиг. 36. Реализиран дизайн на страницата за побойници на системата за контрол на клиентите в Adobe Xd.

Програмния код използван за реализация на потребителския интерфейс е написан на JavaScript и е разгънат на Vite сървър, който компилира и пуска сървъра на локален източник. Страниците, който са представени чрез Adobe Xd са реализирани чрез чист html и SCC ( Cascading Style Sheets). Използван е чист html и scc позволява много подходи за дизайна на уеб страница и нисък размер на крайния продукт. Но липсва гъвкавост, която други платформи като React[[24]](#footnote-24) или Angular[[25]](#footnote-25) предлагат.

Реализацията на малкото логика, която интерфейса има, се е извършва чрез JavaScript класове и функции строго свързани към дадени действие на потребителя. Заложено е в проекта на потребителския интерфейс да има възможно най-малко логиката, тоест единствената логика да е свързана с приемане на заявки от поробителя. Както и логиката служи за обработка на съобщения от сървъра и за визуализация на информация. Всички заявки, които се извършват в приложението от и към сървъра се синхронизират за да не се изпускат никаква информация или да се покаже стара или ненужна за съответния сценарии информация.

Визуалното тестване на потребителския интерфейс е процеса, по който един потребител би виждал системата и би я употребявал на множество уеб браузери и устройства, като мобилни телефони, компютри и физически обекти. Визуалното тестване на системата се случва по следния начин. Първо за да се стартира сървъра отговорен за бизнес логиката и базата от данни. След това се настройва да симулира реална употреба на системата, като на пример административна институция. След което се стартира потребителския интерфейс и се инспектира визуално на множество различни платформи дали отговаря на изискванията, по който се води проекта. Използвайки сървър от рода на Vite позволява да се редактира един уеб потребителски интерфейс в реално време и да се опреснява след запазване на съответните промени. Това спестява презареждането на сървъра и съответно съкращава времето за разработка. Този метод работи само когато искаме да погледнем дали нещо е правения по цвят, шрифт или подравнено адекватно. Но за по-сериозни проверки, като на пример автентикация на потребител или изпращане и приемане на правилните заявки, се използват автоматични тестове.

Автоматично тестването на потребителския интерфейс се извършва чрез употребата на инструменталната библиотека на Selenium, които е широко използвана в сферата на тестване на потребителски интерфейса. Инструменталната библиотека може да бъде ползвана на множество езици, но за целта на тестване на системата за контрол на клиентите е подходящо чрез java езика. Процеса на автоматично тестване на потребителски интерфейс е сходен с този на тестване на логика в класове или фикции. След зареждане на системата и нейната настройка с писането на директен код може да проверим как един интерфейс реагира в набор от различни ситуации спестявайки време за ръчна проверка. Подобни тестове са полезни, когато има множество промени по дадени елементи но без те да си губят функциите. Така има функционалност, която не се променя а промените биват чисто визуални.

# Заключение

Основните характеристики на системата за контрол е проектирана да оптимизира начина на работа, така че да улеснява клиента и извлече максимална информация за състоянието на обслужването. Тоест по вече изградени методики в човешкото битие, като това да чакаш на опашка, да се обърне по такъв начин че да е от полза за всички. Така проекта представлява система за оптимизиране работния процес без да се нуждае дадено предприятие или институция от допълнителна работна ръка и да може да поема по-голям набор от клиенти по ефективен и бърз начин.

Особености на представеното инженерно решение се състоят в начина по който се интегрирали в проблема с изчакване на реда на опашка. Първо идеята, че хората чакат на опашки за множество услуги навежда към теорията на опашките. Потока от сигнали ( клиенти в този случай) ако бъде систематизирана, то с по-малък работен състав може да се обработят сигнали по енергийно ефективен (Bhat, 2013). Когато това се приложи в практиката се получава точно система за контрол на клиентите. Там клиентите са систематизирани по услугата, за която желаят да бъдат обслужени.

Предимства на система за контрол на опашките е строго обвързана с икономическите характеристики на едно предприятие. Тоест от малък работен персонал да се вадят повече брутни печалби. Предимство е и за менажиращия персонал, където можем да има много по малко менажери за повече обслужващ персонал. Това се постига благодарене на статистките и следенето в реално време.

Резултати от експерименти гласят че множество фирми днешно време се нуждаят от подобна система. Това се дължи на намаляващия набор от работен персонал, който е склонен да работи административни или обслужващи роли. При тестването на системата в различни сценарии може да види как един работещ персонал обслужва до 15 души чакащи за услуга по 35 минути за чакане и да няма загуби от клиенти. Както едни огромен бонус за приложението е широкия набор от приложения, който са възможни за продукта.

В последствие може да се работи върху усъвършенстването на продукта при прилагане в други нации. По този начин се разширява пазарния дял на продукта. За конкретни технологически добавки към кода може да се наблегне над добавяне на различни модули за допълнителна логика при по-специализирани системи.

# Цитирани източници

1. Bealdung. (29 12 2022 r.). *Security with Spring*. Извлечено от Baeldung: https://www.baeldung.com/security-spring.
2. Bhat, U. N. (2013). *A study of the queueing system.*
3. Cem Kaner, J. F. (1999). *Testing Computer Software.*
4. Erich Gamma, RichardHelm, Relph Johnson. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.*
5. Jerry Cao, Ben Gramillion, Matt Ellies. (2015). *UX Design Process Best Practices.*
6. Marks Pollack, Oliver Gierke, Thomas Risberg. (2012). *Spring Data.*
7. Participants, M. (2008). *The Use of Dupont Analysis.*
8. Sprila, L. (2020). *Spring Security in Action.*
9. while, W. (2022). *Consummer Survey: The start of waiting in line.*
10. Касърова, д.-р. В. (2008). Нов прочит на финансовия анализ от позициите на стойността.
11. Тодоров, д. д.-р. (2017). Доходност на бизнеса – методологични и приложни аспекти на анализа контрола.
12. Чуков, П. д.-р. (2021). Методология и методика за анализ на рентабилността на собствения капитал (интегриране на модел „Дюпон“ с модела за анализ чрез ефекта на финансовия лост).

# Приложения

1. Програмен код отговорен за изкуствения на системата за контрол на клиентите

package com.diploma.ticket.system.util.statistics.disicion.tree;

import java.util.\*;

public class DecisionTree {

private Node rootNode;

private final String targetAttribute; // target attribute is the attribute we want to predict when classifying

private String [] trainAttributes; // attributes by which the tree classifies data

private Map<String, ArrayList<Integer>> possibleAttributeValues; // values that each attribute can have

public DecisionTree(String targetAttribute, String [] trainAttributes){

this.rootNode = new Node();

// attribute that the model is trained for to predict the value --> Finished in our case

this.targetAttribute = targetAttribute;

// List with attributes that the data should be trained on

this.trainAttributes = trainAttributes;

}

public void train(Set<Passenger> data){

possibleAttributeValues = getPossibleAttributeValues(data);

long timeStart = System.currentTimeMillis();

train(this.rootNode, data, this.trainAttributes);

long duration = System.currentTimeMillis() - timeStart;

System.out.println("Duration of training: " + duration + " ms");

}

public void train(Node node, Set<Passenger> passengers, String[] attributes){

if(allPositive(passengers, this.targetAttribute)){ // if all passengers from the set finished

node.setLabel("1");

node.setLeaf(true);

return;

}

else if(allNegative(passengers, this.targetAttribute)){ // if all passengers from the set did not finish

node.setLabel("0");

node.setLeaf(true);

return;

}

else if(attributes.length == 0){

node.setLabel(String.valueOf(mcv(passengers, this.targetAttribute)));

node.setLeaf(true);

return;

}

else{

String bestAttribute = getBestSplitAttribute(passengers, attributes);

node.setLabel(bestAttribute);

ArrayList<Integer> possibleValues = possibleAttributeValues.get(bestAttribute);

for(int pv : possibleValues){

Condition c = new Condition(pv); // create new condition with condition value for each possible class value

Node child = new Node();

c.setSuccessor(child); // create new node that is connected to that condition

node.addCondition(c);

Set<Passenger> subsetPV = createSubset(passengers, bestAttribute, pv);

if(subsetPV.isEmpty()){

child.setLabel(String.valueOf(mcv(passengers, this.targetAttribute)));

child.setLeaf(true);

}

else{

String [] remainingAttributes = removeAttribute(bestAttribute, attributes);

train(child, subsetPV, remainingAttributes);

}

}

}

}

// checks if all all examples are positive/ finished

private boolean allPositive(Set<Passenger> data, String attribute){

Object [] dataArr = data.toArray();

Passenger p;

for (int i = 0; i < dataArr.length; i++){

p = (Passenger) dataArr[i];

if(p.getAttributeValue(attribute) == 0){

return false;

}

}

return true;

}

// checks if all all examples are negative/ died

private boolean allNegative(Set<Passenger> data, String attribute){

Object [] dataArr = data.toArray();

Passenger p;

for (int i = 0; i < dataArr.length; i++){

p = (Passenger) dataArr[i];

if(p.getAttributeValue(attribute) == 1){

return false;

}

}

return true;

}

// returns the most common value from a the column "attribute" of a data set "data"

private int mcv(Set<Passenger> data, String attribute){

Object[] dataArr = data.toArray();

Map<Integer, Integer> valueCount = new HashMap<>();

Passenger p;

for(int i = 0; i < dataArr.length; i++){

p = (Passenger) dataArr[i];

valueCount.merge(p.getAttributeValue(attribute), 1, Integer::sum);

}

// return value with max occurrences (= mcv)

return Collections.max(valueCount.entrySet(), Map.Entry.comparingByValue()).getKey();

}

private String getBestSplitAttribute(Set<Passenger> data, String [] attributes){

double [] gainResults = new double[attributes.length];

// calculate information gain for every attribute (that is still available)

for(int i = 0; i < attributes.length; i++){

gainResults[i] = calcInformationGain(data, attributes[i]);

}

// return attribute with max information gain

return attributes[getMaxPosition(gainResults)];

}

// returns index of the maximum

private int getMaxPosition(double[] arr){

int idx = 0;

double max = Double.MIN\_VALUE;

for(int i = 0; i < arr.length; i++){

if(arr[i] > max){

max = arr[i];

idx = i;

}

}

return idx;

}

// calculates the entropy for a set in regard to the attribute attribute

private double calcEntropy(Set<Passenger> data, String attribute){

double entropy = 0.0;

Set<Passenger> subset;

double p; // probability of event e

for(int possValue : possibleAttributeValues.get(attribute)){

subset = createSubset(data, attribute, possValue);

// if no examples for possValue, than p(v) = 0 --> entropy = 0 and thus doesn't change

if(subset.size() > 0){

p = (double) subset.size() / (double) data.size();

// entropy = - SUM (p(e) \* log2(p(e)))

entropy -= p \* log2(p);

}

}

return entropy;

}

// calculates the information gain for a set

private double calcInformationGain(Set<Passenger> data, String attribute){

double postEntropy = 0.0; // entropy that will emerge when splitting on attribute attribute

for(int possValue : possibleAttributeValues.get(attribute)){

// subset for the data with value v for attribute attribute

Set<Passenger> subset\_v = createSubset(data, attribute, possValue);

double weighting = (double) subset\_v.size() / (double) data.size();

double subsetEntropy = calcEntropy(subset\_v, this.targetAttribute);

postEntropy += weighting \* subsetEntropy;

}

double preEntropy = calcEntropy(data, this.targetAttribute); // entropy before splitting on attribute attribute

return preEntropy - postEntropy;

}

private double log2(double x){

return Math.log(x)/Math.log(2.0);

}

// creates a subset from a set with all Elements that have value as their attribute value

private Set<Passenger> createSubset(Set<Passenger> prevSet, String attribute, int value){

Set<Passenger> subset = new HashSet<>();

Object[] dataArr = prevSet.toArray();

Passenger p;

for(Object o : dataArr){

p = (Passenger) o;

if(p.getAttributeValue(attribute) == value){

subset.add(p);

}

}

return subset;

}

private String[] removeAttribute(String attribute, String [] attributes){

try{

String [] remainingAttributes = new String[attributes.length-1];

int j = 0;

for(int i = 0; i < attributes.length; i++){

if(attributes[i].equals(attribute)){

continue;

}

remainingAttributes[j] = attributes[i];

j++;

}

return remainingAttributes;

}catch (Exception e){

e.printStackTrace();

return null;

}

}

private HashMap<String, ArrayList<Integer>> getPossibleAttributeValues(Set<Passenger> data){

HashMap<String, ArrayList<Integer>> possAttValues = new HashMap<>();

for(String attribute : trainAttributes){

possAttValues.put(attribute, new ArrayList<>());

}

Object [] dataArr = data.toArray();

Passenger p;

for(Object o : dataArr){

p = (Passenger) o;

for(String attribute : trainAttributes){

if(!possAttValues.get(attribute).contains(p.getAttributeValue(attribute))){

possAttValues.get(attribute).add(p.getAttributeValue(attribute));

}

}

}

// finished values added separately

ArrayList<Integer> finishedValues = new ArrayList<>();

finishedValues.add(0);

finishedValues.add(1);

possAttValues.put(Attribute.FINISHED, finishedValues);

return possAttValues;

}

public void print(){

Queue<Node> q = new LinkedList();

Queue<Node> q2 = new LinkedList();

q.add(rootNode);

Node curr;

Node br = new Node();

br.setLabel(" | ");

List<Condition> branches;

while(!q.isEmpty() || !q2.isEmpty()){

while(!q.isEmpty()){

curr = q.poll();

branches = curr.getConditions();

System.out.print(" " + curr.getLabel());

for(Condition branch : branches){

q2.add(branch.getSuccessor());

System.out.print(branch.getCompareValue());

}

if(branches.size() != 0) q2.add(br);

}

System.out.println();

while(!q2.isEmpty()){

q.add(q2.poll());

}

}

}

// classifies a single data point

public String classify(Passenger passenger){

Node curr = rootNode;

List<Condition> branches;

while(!curr.isLeaf()) {

branches = curr.getConditions();

int compareValue = passenger.getAttributeValue(curr.getLabel());

Node next = null;

for (Condition branch : branches) {

if (branch.check(compareValue)) {

next = branch.getSuccessor();

break;

}

}

if(next == null){

return "2"; // can not classify data

}

else{

curr = next;

}

}

return curr.getLabel();

}

}

1. Програмен код отговорен за филтрирането на реда на опашката

package com.diploma.ticket.system.service;

import com.diploma.ticket.system.entity.\*;

import com.diploma.ticket.system.payload.response.NextInLineResponse;

import com.diploma.ticket.system.util.JwtUtil;

import org.apache.log4j.Logger;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.sql.Time;

import java.util.\*;

@Service

@Transactional

public class QueueService {

private final CounterService counterService;

private final FavorService favorService;

private final TicketService ticketService;

private final PersonalTicketService personalTicketService;

private final JwtUtil jwtUtil;

private final UserService userService;

private static Logger logger= Logger.getLogger(QueueService.class.getName());

@Autowired

public QueueService(

CounterService counterService,

FavorService favorService,

TicketService ticketService,

PersonalTicketService personalTicketService,

JwtUtil jwtUtil,

UserService userService

) {

this.counterService = counterService;

this.favorService = favorService;

this.ticketService = ticketService;

this.personalTicketService = personalTicketService;

this.jwtUtil = jwtUtil;

this.userService = userService;

}

public void openCounter(Long counterId, String email) {

Counter counter=

counterService.findCounter(counterId);

counter.setActive(true);

logger.info("Counter whit Id:"+counterId+" was opened");

User user=getUserFromToken(email);

//TODO Think a way to integrite to filter

}

public void closeCounter(Long counterId, String email) {

Counter counter

=counterService.findCounter(counterId);

counter.setActive(false);

logger.info("Counter whit Id:"+counterId+" was closed");

User user=getUserFromToken(email);

//TODO Think a way to integrite to filter

}

private User getUserFromToken(String userEmail){

User user=userService.findUserByEmail(userEmail);

logger.info("Get user from Token was invoked");

return user;

}

public Set<PersonalTicket> getWaithingForCounter(Long counterId){

//1. get the Counter entity

Counter counter =counterService.findCounter(counterId);

//2. get the Set of Favor types for the counter

Set<Favor> favors = counter.getFavor();

//3. get the tickets for counter

List<List<Ticket>> ticketInLine=getTicketFromFavors(favors);

//4. get the personal tickets for the counter

List<Set<PersonalTicket>> personalTickets=new ArrayList<>();

for (List<Ticket> ticketList:ticketInLine){

for (Ticket ticket:ticketList){

personalTickets.add(ticket.getPersonalTickets());

}

}

Set<PersonalTicket> waithingTickets=new HashSet<>();

SetInListIterator<PersonalTicket> nextInLineIterator=new SetInListIterator<>(personalTickets);

while(nextInLineIterator.hasNext()){

PersonalTicket next=nextInLineIterator.next();

if(next.isActive()&&next.getFinishTime()==null){

waithingTickets.add(next);

}

}

return waithingTickets;

}

public NextInLineResponse getNextInLineByCounter(

Long counterId,

String authHeader

) {

Counter counter =counterService.findCounter(counterId);

Set<Favor> favors = counter.getFavor();

List<List<Ticket>> ticketInLine=getTicketFromFavors(favors);

List<Set<PersonalTicket>> personalTickets=getPersonalTicetsFromTicket(ticketInLine);

Integer waitingInLine=-1;

PersonalTicket nextInLine=new PersonalTicket();

nextInLine.setIssueTime(new Time(23,59,59));

SetInListIterator<PersonalTicket> nextInLineIterator=new SetInListIterator<>(personalTickets);

while(nextInLineIterator.hasNext()){

PersonalTicket next=nextInLineIterator.next();

if(next.getIssueTime().before(nextInLine.getIssueTime())&&next.isActive()&&next.getFinishTime()==null){

waitingInLine++;

nextInLine=next;

}

}

if(nextInLine.getId()==null){

return null;

}

personalTicketService.setTicketToUser(authHeader,nextInLine);

NextInLineResponse response=

NextInLineResponse

.builder()

.number(nextInLine.getId())

.issueTime(nextInLine.getIssueTime())

.finishTime(nextInLine.getFinishTime())

.peopleInLine(waitingInLine)

.build();

return response;

}

private List<List<Ticket>> getTicketFromFavors(Set<Favor> favors){

List<List<Ticket>> ticketInLine=new ArrayList<>();

for(Favor favor:favors){

ticketInLine.add(ticketService.findTicketByFavor(favor.getId()));

}

logger.info("Get Ticket from Favor was invoked");

return ticketInLine;

}

private List<Set<PersonalTicket>> getPersonalTicetsFromTicket(List<List<Ticket>> ticketInLine) {

List<Set<PersonalTicket>> personalTickets = new ArrayList<>();

for (List<Ticket> ticketList : ticketInLine) {

for (Ticket ticket : ticketList) {

personalTickets.add(ticket.getPersonalTickets());

}

}

return personalTickets;

}

/\*\*

\* The class is used for the way in witch the line filters out the next ticket

\* @param <T> The set of Personal Tickets

\*/

public class SetInListIterator<T> implements Iterator<T> {

private final List<Set<T>> list;

private int currentListIndex;

private Iterator<T> currentSetIterator;

public SetInListIterator(List<Set<T>> list) {

this.list = list;

this.currentListIndex = 0;

this.currentSetIterator = list.get(0).iterator();

}

@Override

public boolean hasNext() {

if (currentSetIterator.hasNext()) {

return true;

} else if (currentListIndex < list.size() - 1) {

currentListIndex++;

currentSetIterator = list.get(currentListIndex).iterator();

return hasNext();

} else {

return false;

}

}

@Override

public T next() {

if (hasNext()) {

return currentSetIterator.next();

} else {

throw new java.util.NoSuchElementException();

}

}

public Iterator<T> getCurrent(){

return currentSetIterator;

}

}

}

1. **Програмен код отговорен за филтрирането на заявки.**

package com.diploma.ticket.system.controller;

import com.diploma.ticket.system.payload.request.TicketCreationRequest;

import com.diploma.ticket.system.payload.response.CreationResponse;

import com.diploma.ticket.system.service.TicketService;

import com.diploma.ticket.system.entity.Ticket;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.net.URI;

import java.util.List;

@RestController

@RequestMapping(path="api/v2/ticket")

public class TicketController {

private final TicketService ticketService;

@Autowired

public TicketController(TicketService ticketService){

this.ticketService=ticketService;

}

@GetMapping

public ResponseEntity<List<Ticket>> getTickets(){

List<Ticket> responseBody=ticketService.getTickets();

return ResponseEntity.ok().body(responseBody);

}

@GetMapping("/{favorId}")

public ResponseEntity<List<Ticket>> getTicketsByFavorId(

@PathVariable("favorId") Long favorId

){

List<Ticket> responseBody=ticketService.findTicketByFavor(favorId);

return ResponseEntity.ok().body(responseBody);

}

@PostMapping

public ResponseEntity<CreationResponse> registerNewTicket(

@RequestBody TicketCreationRequest ticket

){

CreationResponse responseBody=ticketService.addNewTicket(ticket);

return ResponseEntity.created(URI.create("Ticket")).body(responseBody);

}

@PatchMapping(path="{ticketId}")

public void updateTicket(

@PathVariable("ticketName")String name,

@RequestBody Ticket ticket

){

ticketService.updateTicket(name,ticket);

}

@PutMapping("addFavor/{idTicket}/{idFavor}")

public ResponseEntity<String> addFavor(

@PathVariable("idTicket") Long idTicket,

@PathVariable("idFavor") Long idFavor

){

ticketService.addFavor(idTicket,idFavor);

return ResponseEntity.ok("Favor had bean added to the Ticket successfully!");

}

@PutMapping("addPersonalTicket/{idTicket}/{idPersonalTicket}")

public ResponseEntity<String> addPersonalTicket(

@PathVariable("idTicket") Long idTicket,

@PathVariable("idPersonalTicket") Long idPersonalTicket

){

ticketService.addPersonalTicket(idTicket,idPersonalTicket);

return ResponseEntity.ok("Personal Ticket has bean added to the Ticket successfully!");

}

@DeleteMapping("/{id}")

public ResponseEntity<?> deleteATicket(

@PathVariable Long id

) {

ticketService.deleteTicket(id);

return ResponseEntity.ok().build();

}

}

package com.diploma.ticket.system.controller;

import com.diploma.ticket.system.entity.Favor;

import com.diploma.ticket.system.entity.PersonalTicket;

import com.diploma.ticket.system.payload.response.CreationResponse;

import com.diploma.ticket.system.service.FavorService;

import com.diploma.ticket.system.service.PersonalTicketService;

import com.diploma.ticket.system.service.QueueService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.net.URI;

import java.util.ArrayList;

import java.util.List;

import java.util.Set;

@RestController

@RequestMapping(path="api/v1/draft")

public class TicketDraftController {

private final QueueService queueService;

private final PersonalTicketService personalTicketService;

private final TicketController ticketService;

private final FavorService favorService;

@Autowired

public TicketDraftController(

QueueService queueService,

PersonalTicketService personalTicketService,

TicketController ticketService,

FavorService favorService) {

this.queueService = queueService;

this.personalTicketService = personalTicketService;

this.ticketService = ticketService;

this.favorService = favorService;

}

@GetMapping("/favor")

public ResponseEntity<List<Favor>> getService(){

List<Favor> responseBody=new ArrayList<>();

responseBody=favorService.getService();

return ResponseEntity.ok().body(responseBody);

}

@GetMapping("/waitingForCounter/{counterId}")

public Set<PersonalTicket> getWaitingForCounter(

@PathVariable Long counterId

){

return queueService.getWaithingForCounter(counterId);

}

@GetMapping("/lastPersonalTicket")

public Long getLastPersonalTicket(){

return personalTicketService.getLastPersonTicketId();

}

@PostMapping("/{ticketId}")

public ResponseEntity<CreationResponse> draftTicket(

@PathVariable Long ticketId,

@RequestBody PersonalTicket personalTicket

){

PersonalTicket draftedPersonalTicket

=personalTicketService.addNewPersonalTicket(personalTicket);

ticketService.addPersonalTicket(ticketId,personalTicket.getId());

CreationResponse response=

new CreationResponse(

personalTicket.getId(),

"Personal Ticket for Ticket whit id:"

+ticketId+" has bean created successfully");

return ResponseEntity.created(URI.create("PersonaTicket")).body(response);

}

}

package com.diploma.ticket.system.controller;

import com.diploma.ticket.system.payload.response.CreationResponse;

import com.diploma.ticket.system.service.PersonalTicketService;

import com.diploma.ticket.system.entity.PersonalTicket;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpHeaders;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.net.URI;

import java.util.List;

@RestController

@RequestMapping(path="api/v2/personalticket")

public class PersonalTicketController {

private final PersonalTicketService personalTicketService;

@Autowired

public PersonalTicketController(PersonalTicketService personalTicketService){

this.personalTicketService=personalTicketService;

}

@GetMapping

public ResponseEntity<List<PersonalTicket>> getTickets(){

List<PersonalTicket> responseBody=

personalTicketService.getPersonaTickets();

return ResponseEntity.ok(responseBody);

}

@GetMapping("/andTicket")

public List<PersonalTicket> findAllAndJoinTicket(){

return personalTicketService.findAllAndJoinTicket();

}

@GetMapping("/{ticketId}")

public ResponseEntity<List<PersonalTicket>> getPersonalTicketsByTicketId(

@PathVariable("ticketId") Long ticketId

){

List<PersonalTicket> responseBody=

personalTicketService.findActivePersonalTicketByTicket(ticketId);

return ResponseEntity.ok(responseBody);

}

@PostMapping

public ResponseEntity<CreationResponse> registerPersonalTicket(

@RequestBody PersonalTicket personalTicket

){

PersonalTicket personalTicketCreated

=personalTicketService.addNewPersonalTicket(personalTicket);

CreationResponse response=

new CreationResponse(personalTicketCreated.getId(),"Created successfully");

return ResponseEntity.created(URI.create("PersonaTicket")).body(response);

}

@PatchMapping(path="{personalTicketId}")

public ResponseEntity updateTicket(

@PathVariable("ticketNumber")Long number,

@RequestBody PersonalTicket personalTicket

){

personalTicketService.updatePersonalTicket(number,personalTicket);

return ResponseEntity.ok("Ticket whit number:"+number+" hase been updated");

}

@PutMapping("/finish/{ticketNumber}")

public ResponseEntity finishTicket(

@PathVariable Long ticketNumber,

@RequestHeader(HttpHeaders.AUTHORIZATION) String authHeader

){

CreationResponse responseBody=

personalTicketService.finishTicket(ticketNumber,authHeader);

return ResponseEntity.ok(responseBody);

}

@DeleteMapping("/{id}")

public ResponseEntity<?> deletePersonalTicket(

@PathVariable Long id

) {

personalTicketService.deletePersonalTicket(id);

return ResponseEntity.ok().build();

}

}

package com.diploma.ticket.system.controller;

import com.diploma.ticket.system.payload.response.CreationResponse;

import com.diploma.ticket.system.service.CounterService;

import com.diploma.ticket.system.entity.Counter;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.net.URI;

import java.util.ArrayList;

import java.util.List;

@RestController

@RequestMapping(path="api/v2/counter")

public class CounteController {

private final CounterService counterService;

@Autowired

public CounteController(CounterService counterService){

this.counterService=counterService;

}

@GetMapping

public ResponseEntity<List<Counter>> getCounter(){

List<Counter> responseBody=new ArrayList<>();

responseBody=counterService.getCounters();

return ResponseEntity.ok().body(responseBody);

}

@PostMapping

public ResponseEntity<CreationResponse> registerNewCounter(@RequestBody Counter counter){

CreationResponse response=null;

response=counterService.addNewCounter(counter);

return ResponseEntity.created(URI.create("Counter"))

.body(response);

}

@PutMapping("/add/ticket/{favorId}/{counterId}")

public ResponseEntity addFavorToCounter(

@PathVariable Long favorId,

@PathVariable Long counterId

){

CreationResponse response=null;

counterService.addNewFavor(favorId,counterId);

return ResponseEntity.ok().body(response);

}

@PatchMapping(path="{counterId}")

public ResponseEntity<String> updateCounter(

@PathVariable("counterName")String name,

@RequestBody Counter counter)

{

counterService.updateCounter(name, counter);

return ResponseEntity.ok().body("ok");

}

@DeleteMapping("/{id}")

public ResponseEntity<?> deleteCounter(

@PathVariable Long id

) {

counterService.deleteCounter(id);

return ResponseEntity.ok().build();

}

}

package com.diploma.ticket.system.controller;

import com.diploma.ticket.system.entity.PersonalTicket;

import com.diploma.ticket.system.payload.response.NextInLineResponse;

import com.diploma.ticket.system.service.QueueService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpHeaders;

import org.springframework.http.ResponseEntity;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.annotation.CurrentSecurityContext;

import org.springframework.web.bind.annotation.\*;

import java.util.Set;

@RestController

@RequestMapping(path="api/v1/queue")

public class QueueController {

private final QueueService queueService;

@Autowired

public QueueController(QueueService queueService) {

this.queueService = queueService;

}

@GetMapping("/nextInLine/{counterId}")

public ResponseEntity<NextInLineResponse> getNExtInLine(

@PathVariable Long counterId,

@RequestHeader(HttpHeaders.AUTHORIZATION) String authHeader

) {

NextInLineResponse response

=queueService.getNextInLineByCounter(counterId,authHeader);

if(response==null){

return ResponseEntity.notFound().build();

}

return ResponseEntity.ok().body(response);

}

@GetMapping("/authentication")

public Object getAuthentication(

@CurrentSecurityContext(expression = "authentication") Authentication authentication

) {

return authentication.getName();

}

@PutMapping("/open/counter/{counterId}")

public ResponseEntity<?> openCounter(

@PathVariable Long counterId,

@CurrentSecurityContext(expression = "authentication") Authentication authentication

){

queueService.openCounter(counterId,authentication.getName());

return ResponseEntity.ok("Counter whit id:"+counterId+" was opened");

}

@DeleteMapping("/close/counter/{counterId}")

public ResponseEntity<String> closeCounter(

@PathVariable Long counterId,

@CurrentSecurityContext(expression = "authentication") Authentication authentication

){

queueService.closeCounter(counterId,authentication.getName());

return ResponseEntity.ok("Counter whit id:"+counterId+"was closed");

}

}

package com.diploma.ticket.system.config;

import com.diploma.ticket.system.filter.JwtAuthFilter;

import com.diploma.ticket.system.repository.UserRepository;

import lombok.RequiredArgsConstructor;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.AuthenticationProvider;

import org.springframework.security.authentication.dao.DaoAuthenticationProvider;

import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.security.core.userdetails.UserDetailsService;

import org.springframework.security.core.userdetails.UsernameNotFoundException;

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import org.springframework.web.cors.CorsConfiguration;

import org.springframework.web.cors.CorsConfigurationSource;

import org.springframework.web.cors.UrlBasedCorsConfigurationSource;

import java.util.Arrays;

@Configuration

@EnableWebSecurity

@RequiredArgsConstructor

public class SecurityConfig {

private final JwtAuthFilter jwtAuthFilter;

@Autowired

private final UserRepository userRepository;

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception{

http

.cors().and()

.csrf().disable()

.sessionManagement()

.sessionCreationPolicy(SessionCreationPolicy.STATELESS)

.and()

.authorizeHttpRequests()

.requestMatchers("/api/vi/auth/\*\*","api/v1/draft/\*\*")

.permitAll().anyRequest().authenticated();

http

.authenticationProvider(authenticationProvider());

http

.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);

;

return http.build();

}

@Bean

public AuthenticationProvider authenticationProvider() {

final DaoAuthenticationProvider authenticationProvider=

new DaoAuthenticationProvider();

authenticationProvider.setUserDetailsService(userDetailsService());

authenticationProvider.setPasswordEncoder(passwordEncoder());

return authenticationProvider;

}

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {

return authenticationConfiguration.getAuthenticationManager();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

@Bean

public UserDetailsService userDetailsService(){

return new UserDetailsService() {

@Override

public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

return userRepository.findByEmail(email).orElseThrow();

}

};

}

}

package com.diploma.ticket.system.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.servlet.config.annotation.CorsRegistry;

import org.springframework.web.servlet.config.annotation.EnableWebMvc;

import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration

@EnableWebMvc

public class CorsConfig implements WebMvcConfigurer {

@Override

public void addCorsMappings(CorsRegistry registry) {

registry.addMapping("/\*\*")

.allowedOrigins("\*")

.allowedMethods("HEAD", "GET", "PUT", "POST", "DELETE", "PATCH");

}

}

package com.diploma.ticket.system.util;

import io.jsonwebtoken.\*;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.security.core.userdetails.UserDetails;

import org.springframework.stereotype.Component;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

@Component

public class JwtUtil {

@Value("${spring.app.jwtSecret}")

private String secretKey;

@Value("${spring.app.jwtExpirationMs}")

private int jwtExpirationMs;

public String extractUsername(String token) {

return extractClaim(token, Claims::getSubject);

}

public Date extractExpiration(String token) {

return extractClaim(token, Claims::getExpiration);

}

public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {

final Claims claims = extractAllClaims(token);

return claimsResolver.apply(claims);

}

private Claims extractAllClaims(String token) {

Claims call=null;

try {

call = Jwts.parser().setSigningKey(secretKey).parseClaimsJws(token).getBody();

} catch (ExpiredJwtException e) {

throw new RuntimeException(e);

} catch (UnsupportedJwtException e) {

throw new RuntimeException(e);

} catch (MalformedJwtException e) {

throw new RuntimeException(e);

} catch (SignatureException e) {

throw new RuntimeException(e);

} catch (IllegalArgumentException e) {

throw new RuntimeException(e);

}

return call;

}

private Boolean isTokenExpired(String token) {

return extractExpiration(token).before(new Date());

}

public String generateRefreshToken(

UserDetails userDetails

) {

return createToken(new HashMap<>(), userDetails.getUsername());

}

public String generateToken(UserDetails userDetails) {

Map<String, Object> claims = new HashMap<>();

return createToken(claims, userDetails.getUsername());

}

private String createToken(Map<String, Object> claims, String subject) {

return Jwts.builder().setClaims(claims)

.setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))

.setExpiration(new Date(System.currentTimeMillis() + jwtExpirationMs))

.signWith(SignatureAlgorithm.HS256, secretKey).compact();

}

public Boolean isTokenValid(String token, UserDetails userDetails) {

final String username = extractUsername(token);

return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));

}

}

package com.diploma.ticket.system.service;

import com.diploma.ticket.system.entity.Counter;

import com.diploma.ticket.system.entity.User;

import com.diploma.ticket.system.repository.UserRepository;

import org.apache.log4j.Logger;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import java.util.List;

import java.util.Optional;

@Service

@Transactional

public class UserService {

private final UserRepository userRepository;

private final PasswordEncoder passwordEncoder;

private final CounterService counterService;

private static Logger logger= Logger.getLogger(TicketTypeService.class.getName());

@Autowired

UserService(UserRepository userRepository, PasswordEncoder passwordEncoder, CounterService counterService){

this.userRepository=userRepository;

this.passwordEncoder = passwordEncoder;

this.counterService = counterService;

}

@Transactional

public void register(User request) {

Optional<User> user= userRepository.findByEmail(request.getEmail());

if(user.isPresent()){

logger.info("User type whit Name:"+request.getEmail()+" Exists!");

throw new IllegalArgumentException("User whit email:"+request.getEmail()+"Exists! Try a different email.");

}

request.setPassword(passwordEncoder.encode(request.getPassword()));

userRepository.save(request);

logger.info("User type whit Name:"+request.getEmail()+" has bean saved to the repository");

}

public List<User> getUsers(){

return userRepository.findAll();

}

public void addCounter(Long counterId, String email) {

User user=findUserByEmail(email);

Counter counter=counterService.findCounter(counterId);

user.addCounter(counter);

logger.info(

"Add counter method was invoked whit the CounterId:" +counterId+

" And whit the user email:"+email

);

}

public void deleteUser(Integer id) {

User user

=userRepository.findById(id).orElseThrow();

userRepository.delete(user);

logger.info("User type whit Id:"+id+" has bean saved to the repository");

}

public User findUserByEmail(String email){

logger.info("Find User by Email was invoked.");

return userRepository.findByEmail(email).orElseThrow();

}

public void updateUser(Integer id, User user) {

User userToUpdate=userRepository.findById(id).orElseThrow();

user.setId(id);

userRepository.save(user);

logger.info("User whit id:"+id+"was successfully updated and saved to the repository");

}

}

1. Метод на Ансов - е икономически метод за анализ на конкуренцията [↑](#footnote-ref-1)
2. Postgresql- известен също като Postgres, е безплатна система за управление на релационни бази данни с отворен код. [↑](#footnote-ref-2)
3. ACID- Е широка използвана методология за работа с бази от данни. [↑](#footnote-ref-3)
4. Java- Широко разпространен обектно ориентиран програмен език. [↑](#footnote-ref-4)
5. Spring- Твърдо свързани библиотеки за усилване работата на програмния език java. [↑](#footnote-ref-5)
6. Apache - Спомагателен инструмент за компилиране и пакетиране на java код. [↑](#footnote-ref-6)
7. Tomcat- Сървър за разгъване на java базиран уеб сървър. [↑](#footnote-ref-7)
8. Maven- Инструмент за обединяване и пакетиране на библиотеки използвани в проекта. [↑](#footnote-ref-8)
9. Бъгове- Вид кредитни грешки в програмния код. [↑](#footnote-ref-9)
10. Vanila JavaScript- Е функционален програмен език главно използван за уеб интерфейси. [↑](#footnote-ref-10)
11. Html- Файлов формат за визуализация на уеб страници. [↑](#footnote-ref-11)
12. Wait While- Институт за измерване на количеството от часове, който хората отделят за чакане. [↑](#footnote-ref-12)
13. NFC- Технология базирана на ниско честотни радио вълни. Главно използвани в картите за сигурност. [↑](#footnote-ref-13)
14. Енум- Вид променлива, която се използва в програмирането за да съхранява набор от константи. [↑](#footnote-ref-14)
15. Теория на опашките- Е математически алгоритъм за пресмятане на входен и изходен поток от данни. [↑](#footnote-ref-15)
16. Турникет- Вид пропускателен пункт, който пропуска един след друг индивидуално. [↑](#footnote-ref-16)
17. Gini-е мярка за разделението или разнообразието на класовете във възлите на дървото. [↑](#footnote-ref-17)
18. Строител ( Builder)- Вид логически шаблон за работа с обекти. [↑](#footnote-ref-18)
19. Payload- Клас, който служи за запис на обекти и техните данни. Те не съдържат никаква логика вътре в себе си. [↑](#footnote-ref-19)
20. Body- Така нареченото тяло е файл, който се изпраща заедно с заявката. Най-често се използва за да се прехвърлят обекти помежду потребителския интерфейс и сървъра. [↑](#footnote-ref-20)
21. Персона- Представяне на сходна личност до реална в пълен житейски характер. [↑](#footnote-ref-21)
22. Брайлова азбука-Брайлова азбука се нарича тактилна система за писане и четене, използвана от слепите и слабовиждащите хора по света. [↑](#footnote-ref-22)
23. Брутализъм- Вид дизайн. Главно използван в архитектурата. [↑](#footnote-ref-23)
24. React- Екосистема за програмиране на потребителски уеб интерфейси. [↑](#footnote-ref-24)
25. Angular- Екосистема за програмиране на потребителски уеб интерфейси. [↑](#footnote-ref-25)