



Курсов Проект ООП – Част 2

ДОКУМЕНТАЦИЯ

Явор Чамов | СИТ 1 б) | №21621577

Виктор Денчев | СИТ 1б) | №21621599

Съдържание.

1. Глава 1. Увод.	
a. Описание и идея на проекта.....	2
b. Цел и задачи на разработката.....	2
c. Структура на документацията.....	2-3
2. Глава 2. Преглед на предметната област	
a. Основни дефиниции и концепции	
i. Модели в системата.....	3
ii. Модели за пренос на данни.....	4
iii. Обекти за достъп до данни.....	4-5
iv. Приложен контекст.....	5
v. Услуги.....	6
vi. Контролерни класове.....	6-7
vii. Потребителска сесия.....	7-8
b. Алгоритми	
i. Advanced Encryption Standard (AES).....	8
ii. PBKDF2 за хеширане на парола.....	8-9
iii. Валидиране на парола.....	9
iv. Съхранение на таен ключ във файл	
v. Зареждане на таен ключ от файл	
vi. Single Sign-On (SSO)	
c. Дефиниране на проблеми и сложност на поставената задача	
i. Проблем 1: Избор на база данни	
ii. Проблем 2: Свързване на приложния слой с базата данни	
iii. Проблем 3: Дефиниране на моделите и създаване на таблицы от тях	
iv. Проблем 4: Изграждане на трислойна архитектура (Repository -> Service -> Controller)	
v. Проблем 5: Прилагане на принципа за обръщане на зависимостите (Dependency Inversion Principle)	
vi. Проблем 6: Централизирано място за смяна на изгледите	
vii. Проблем 7: Автентикация и Авторизация	
viii. Проблем 8: Тестване на кода - Unit и End-to-End тестове	
d. Подходи, методи за решаване на поставените проблеми	
e. Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)	
3. Глава 3. Проектиране	
a. Обща структура на проекта пакети, които ще се реализират.	
b. Диаграми/Блок схеми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)	
4. Глава 4. Важни моменти при реализация, алгоритми, оптимизации.	

- a. ApplicationContext клас за инстанциране на всички компоненти на приложението. Прилагане на принципа за обръщане на зависимостите
 - b. Създаване на общо базово меню с възможност за динамично добавяне на бутони в зависимост от ролята на текущо влезият потребител.
 - c. Запазване на текущо логнатият потребител в сесия с възможност за отписване.
 - d. Валидация на полета при попълване на формуляри за създаване на нови обекти.
 - e. Динамично извличане на информацията от база данни без презареждане на изгледа.
- 5. Глава 5. Работа с програмата. Изисквания за изпълнение на програмата.**
- 6. Глава 6. Тестови сценарии.**

Глава 1. Увод.

Описание и структура на проекта.

Проектът представлява разработка на информационна система за управление на складове. Основната цел на системата е да улесни процесите на съхранение и управление на складови помещения, като предоставя функционалности за множествен достъп и различни нива на управление за два типа потребители - администратори и клиенти (складови агенти и собственици). Администраторите имат правомощията да създават профили за собственици и складови агенти, докато собствениците могат да управляват складовете си, включително да отдават наем. Складовите агенти от своя страна участват в процеса на отдаване на наем и имат достъп до различни справки.

Цел и задачи на разработката.

Системата поддържа разнообразни операции, като създаване и редактиране на складове, поддържане на профили с характеристики на потребителите, както и функционалности за рейтинговане на складови агенти. В допълнение, предвидени са операции за добавяне на нови складови помещения с различни характеристики и създаване на формуляри за наем. Важна част от системата са справките, които могат да бъдат генерирани по произволен период за различни аспекти като сключени договори, налични складове за наем, история на наемателите и др. Системата също така поддържа функционалности за известия за важни събития като новопостъпили заявки за отдаване под наем, отдадени складове и изтичащи договори. Това осигурява ефективна комуникация и управление на складовата дейност, улеснявайки взаимодействието между всички участници в процеса.

Структура на документацията.

Документацията за проекта е структурирана в няколко глави, които представят последователно различни аспекти на разработката. Глава 1, "Увод", включва описание на проекта, неговата идея и целите на разработката. Глава 2, "Преглед на предметната област", представя основни дефиниции, концепции и алгоритми, които ще бъдат използвани, както и анализ на проблемите и подходите за тяхното решаване. В глава 3, "Проектиране", се разглежда общата структура на проекта и се представят диаграми и блок схеми, които илюстрират структурата и поведението на системата. Глава 4, "Реализация, тестване", се фокусира върху реализацията на класовете, включително важни моменти и кодови фрагменти, както и планирането и създаването на тестови сценарии. Заключителната глава 5, "Заключение", представя обобщение на постигнатите цели и предлага насоки за бъдещо развитие и усъвършенстване на проекта.

Глава 2. Преглед на предметната област.

Основни дефиниции и концепции.

Модели в системата.

Моделите в информационната система за управление на складове са ключови за структурирането и представянето на данните. Всеки модел описва различни аспекти и същности, които са важни за системата:

Address: Представа адресните данни. Включва информация като улица, номер, град и държава.

Agent (Складов Агент): Описва агентите, които участват в управлението и отдаването на складови помещения.

City (Град): Описва града, свързан с различни същности като адреси и складове.

Country (Държава): Модел, представящ държавата, която може да бъде асоциирана с адреси и градове.

Notification (Известие): Служи за съхранение на информация за известия, които се изпращат до потребителите на системата.

Owner (Собственик): Представа собствениците на складовите помещения.

RentalAgreement (Наемен Договор): Модел, описващ условията и детайлите на наемните договори за складови помещения.

Review (Ревю): Съдържа оценки и коментари на потребителите относно складовете или услугите.

StorageType (Тип Склад): Описва различните типове складови помещения (например хладилен склад, сух склад).

Tenant (Наемател): Представа наемателите на складовите помещения

User (Потребител): Основен модел, представящ потребителите на системата, включващ различни характеристики и идентификационни данни.

Warehouse (Склад): Съдържа информация за отделните складови помещения, включително характеристики, местоположение и състояние.

WarehouseRentalRequest (Заявка за Наем на Склад): Модел, който обработва заявки за наемане на складови помещения, включващ информация за наемателя, периода на наем и специфични изисквания.

Модели за пренос на данни.

DTO (Data Transfer Object) моделите в информационната система за управление на складове са изключително важни за ефективното прехвърляне на данни между различните слоеве на приложението.

Тези модели предоставят специализирана структура за данни, която е оптимизирана за прехвърляне, без да включва бизнес логика или информация, свързана с вътрешната реализация на системата.

Включените DTO модели са:

AddReviewDto: Използва се за събиране и прехвърляне на данни, необходими за добавяне на ново ревю. Включва информация като оценка, текст на коментара.

UserRegistrationDto: Този модел носи информацията, необходима за регистриране на нов потребител, като включва данни като име, електронна поща, парола и други лични детайли.

ViewReviewDto: Използва се за представяне на ревюта в удобен за четене формат. Включва детайли като оценка, коментар и информация за потребителя, който е написал ревюто.

WarehouseDTO: Този модел е предназначен за прехвърляне на информация за складовите помещения, включително техни характеристики, местоположение, статус и други съответни данни.

WarehouseRentalAgreementDto: Съдържа всички релевантни данни за наемни договори за склад, включително информация за наемателя, сроковете на наема, условията и други специфични детайли.

Обекти за достъп до данни.

DAO (*Data Access Object*) моделите са фундаментални за достъпа до данните в информационната система за управление на складове. Тези модели представляват интерфейсите към базата данни, които позволяват извличане, вмъкване, обновяване и изтриване на данни, свързани с различни същности в системата:

CityDao: Управлява всички операции, свързани с данните за градовете. Той позволява извличане на информация за градове, както и тяхното добавяне и обновяване в базата данни.

CountryDao: Отговаря за взаимодействието с данните за държавите. Този *DAO* модел позволява извличане и управление на информацията за различните държави, които са част от системата.

ReviewDao: Този модел управлява данните за рецензиите в системата. Чрез него се осъществява достъп до информацията за рецензиите, тяхното добавяне.

UserDao: Основен *DAO* модел за управление на потребителските данни. Той позволява операции като извличане на потребителски профили, добавяне на нови потребители и обновяване на тяхната информация.

WarehouseDao: Управлява всички аспекти на данните за складовите помещения. Този *DAO* модел позволява извличане, добавяне, обновяване и изтриване на информация за складовете в системата.

Приложен контекст. (*ApplicationContext*)

Класът *ApplicationContext* играе централна роля в информационната система, като действа като контейнер за основните услуги и компоненти. Този клас инициализира и управлява различни *DAO (Data Access Object)* обекти като *UserDao* и *ReviewDao*, които са отговорни за взаимодействието с базата данни за потребители и ревюта. Тези *DAO* обекти се използват за създаване на различни услуги, които предоставят бизнес логиката на системата - например, *UserService*, *ReviewService* и *WarehouseService*.

Услугите в *ApplicationContext* се използват за управление на потребители, складове, ревюта и други ключови функционалности. Например, *UserService* се използва за управление на потребителските профили, докато *WarehouseService* управлява информацията за складовите помещения. Също така, класът предоставя *EncryptionService* и *CredentialManagerService* за сигурност и управление на потребителските пароли и ключове за криптиране.

Освен това, *ApplicationContext* инициализира *ControllerFactory*, която създава различни контролери за уеб интерфейса на системата. Този фабричен клас свързва бизнес логиката със съответните уеб контролери, които обработват заявки от потребителите и генерират подходящите отговори. Примери за такива контролери включват *LoginController*, *RegistrationController* и *WarehouseControlPanelController*. Те улесняват взаимодействието между потребителския интерфейс и бизнес логиката на системата, като осигуряват плавно потребителски опит и ефективно управление на складовите операции.

Услуги.

Услугите в информационната система представляват ключови компоненти, които предоставят специализирана бизнес логика и функционалност.

Всяка услуга е отговорна за определена област на функционалностите в системата:

CityService: Управлява данни и операции свързани с градовете, като предоставя функционалности за извличане, добавяне.

CountryService: Отговаря за управление на информацията за държави. Тази услуга позволява манипулирането на данни за държави, като поддържа създаване.

CredentialManagerService: Предоставя функционалности за управление на удостоверения и пароли на потребителите, включително съхранение и проверка на потребителски данни за сигурност.

EncryptionService: Осигурява услуги за криптиране, които са важни за защитата на чувствителни данни в системата.

PasswordHashingService: Отговаря за хеширането на пароли, което е критичен компонент за сигурността на потребителските акаунти.

ReviewService: Управлява всички аспекти на ревютата в системата, включително добавянето, преглеждането и обработката на ревюта от потребителите.

UserService: Основната услуга за управление на потребителите, която обработва регистрация, вход в системата, управление на потребителски профили и други свързани с потребителя операции.

WarehouseService: Предоставя комплексни функционалности за управление на складовите помещения, включително създаване, обновяване, извличане и управление на информацията за складовете.

Контролерни класове.

Контролерите в информационната система за управление на складове са ключови компоненти, които улесняват взаимодействието между потребителския интерфейс и бизнес логиката. Всеки от следните контролерни класове има специфични задачи и функции:

BaseMenuController: Този контролер управлява основното меню на приложението, предоставяйки общи функционалности и навигация между различните раздели на системата.

BaseWarehouseDialogController: Този контролер се използва за управление на диалогови прозорци, които са свързани с операции за складове, като създаване и актуализация на информация за складовите помещения.

DialogAddReviewController: Този контролер обработва добавянето на рецензии от потребителите, позволявайки им да оценяват и коментират услугите предоставяни от складовете.

HomeController: Управлява началната страница на системата, предоставяйки общ преглед на различните функционалности, достъпни за потребителя.

LoginController: Отговаря за процеса на вход в системата, управлявайки удостоверяването на потребителските идентификационни данни.

MyReviewsController: Този контролер позволява на потребителите да преглеждат и управляват своите ревюта, които са публикували в системата.

ProfileController: Управлява потребителския профил, позволявайки на потребителите да променят своята информация и настройки.

RegistrationController: Отговаря за регистрацията на нови потребители в системата, обработвайки въвеждането на потребителска информация и създаването на нови акаунти.

RentalAgreementController: Този контролер управлява процесите свързани с наемните договори, включително създаване и преглед на съществуващи договори.

WarehouseControlPanelController: Осигурява управлението на панела за контрол на складовите операции, позволявайки на потребителите да извършват различни действия свързани със складовете.

WarehouseUpdateDialogController и *WarehouseCreationDialogController*: Тези контролери управляват диалоговите прозорци за създаване и актуализация на складови помещения, позволявайки на потребителите да въвеждат и променят информация за складовете.

Потребителска сесия. (*UserSession*)

Класът *UserSession* е компонент в архитектурата на информационната система, като служи за управление на потребителските сесии в приложението.

Реализиран като *Singleton*, този клас гарантира, че само една инстанция от него съществува в рамките на приложението, което осигурява централизирано управление на текущата потребителска сесия. Основната функция на *UserSession* е да поддържа информация за текущо влезлия в системата потребител, като предоставя методи за установяване на текущия потребител и излизане от системата.

Алгоритми

Advanced Encryption Standard (AES)

AES е симетричен блоков шифър. Това означава, че той шифрова данни в блокове (по стандарта 128 бита) и използва същия ключ както за шифроване, така и за дешифроване. AES поддържа дължини на ключовете от 128, 192 и 256 бита. По-голямата дължина на ключа осигурява по-висока степен на сигурност, но изисква повече време за обработка. AES е приет като стандарт от правителството на САЩ през 2001 г. и е широко използван в цял свят за защита на чувствителни данни. AES използва серия от трансформации, които се прилагат в множество рундове, за да шифрова данните. Броят на рундовете зависи от дължината на ключа - 10 рунда за 128-битов ключ, 12 рунда за 192-битов ключ и 14 рунда за 256-битов ключ. Всеки рунд включва редица стъпки - заместване на байтове, пренареждане на редове, смесване на колонии и добавяне на ключа на рунда. Процесът на дешифроване е обратен на шифроването и използва същите ключове в обратен ред.

PBKDF2 за хеширане на парола.

Класът *PasswordHashingService* използва алгоритъма PBKDF2 (Password-Based Key Derivation Function 2) за генериране на силни хешове на пароли. PBKDF2 е метод за сигурно преобразуване на парола в криптографски ключ. Алгоритъмът използва HMAC (Hash-Based Message Authentication Code) с SHA1 (Secure Hash Algorithm 1) за генериране на хешове. Salt е случайно генерирана последователност от байтове, която се добавя към паролата преди хеширане, за да предотврати атаки с готови хешове (*rainbow table attacks*). Броят на итерациите (тук 1000) определя колко пъти алгоритъмът се повтаря, което увеличава времето за генериране на хеша и устойчивостта на хеша срещу брутфорс атаки.

Методът *generateStrongPasswordHash* изпълнява следните стъпки:

1. Генериране на *Salt*: Използва *SecureRandom* за генериране на случайна *salt*.

2. Създаване на *PBEKeySpec*: Спецификацията съдържа паролата, *salt*, броя на итерациите и дължината на желанния ключ.
3. Генериране на Хеш: Използва *SecretKeyFactory* с *PBKDF2WithHmacSHA1* за генериране на хеш на паролата.
4. Форматиране на Резултата: Връща хеша като низ, който съдържа броя на итерациите, *salt* и генерирания хеш, всички кодирани в шестнайсетичен формат.

Валидация на парола

Методът *validatePassword* проверява дали дадена парола съответства на съхранения хеш.

1. Разделяне на хеша: Разделя хеша на компонентите му - итерации, *salt* и хеш.
2. Генериране на нов хеш: Използва същите итерации и *salt* за генериране на хеш от предоставената парола.
3. Сравняване на Хешовете: Проверява дали новогенерираният хеш съвпада със съхранения хеш.

Съхранение на таен ключ във файл

Методът *saveSecretKey* изпълнява следните стъпки за съхраняване на таен ключ:

1. Получаване на байтове на ключа: Използва *secretKey.getEncoded()* за получаване на байтовата репрезентация на ключа.
2. Кодиране в *Base64*: Ключът се кодира в *Base64* формат, който е подходящ за текстово представяне и съхранение.
3. Запис във файл: Байтовете се записват във файл, чието име е зададено от параметъра *filename*. Този метод позволява съхранението на ключове по безопасен начин, като избягва директното записване на байтове, които може да бъдат нечетливи или да причинят грешки при обработка в различни системи.

Зареждане на таен ключ от файл

Методът *loadSecretKey* извършва обратния процес:

1. Четене на байтове от файл: Чете байтовете, съхранени във файла.
2. Декодиране от *Base64*: Преобразува байтовете обратно от *Base64* формат.
3. Създаване на Ключа: Използва *SecretKeySpec* за създаване на обект *SecretKey* от байтовете, като указва алгоритъма, с който ключът е свързан. Този метод позволява възстановяването на ключа от файл, като

гарантира, че ключът, прочетен от файла, ще бъде съвместим с алгоритъма, с който е бил използван.

Single Sign-On (SSO)

Методът *handleSsoAction* е проектиран за обработка на действията, свързани с Single Sign-On в приложение. SSO позволява на потребителите да влязат в системата, използвайки запаметени удостоверителни данни от предишна сесия, улеснявайки процеса на вход. Използва *credentialManagerService* за зареждане на запазените удостоверителни данни. Проверява дали удостоверителните данни са налични. Ако данните са налични, опитва се да извърши вход чрез *userService*, като използва имейла и паролата от удостоверителните данни. При успешен вход, навигира към основния интерфейс на приложението. Ако входът е неуспешен, показва съобщение за грешка. Ако данните липсват или са изтекли, показва съобщение, информиращо потребителя, че трябва да извърши ръчен вход.

Дефиниране на проблеми и сложност на поставената задача

Проблем 1: Избор на база данни

Изборът на подходяща база данни е ключов елемент при проектирането на всяка информационна система, тъй като тя играе важна роля за съхранението, обработката и извличането на данни. Важността на този избор се дължи на няколко фактора:

1. **Мащабируемост и Производителност:** Базата данни трябва да може да се справя с нарастващите обеми от данни и потребителски заявки, без да компрометира производителността.
2. **Съвместимост с Технологичния Стек:** Трябва да бъде съвместима с останалите технологии, използвани в проекта, за да осигури плавна интеграция и комуникация.
3. **Сигурност и Надеждност:** Сигурността на данните е критична, особено при съхранението на чувствителна информация. Избраната база данни трябва да предлага механизми за защита на данните и гарантиране на тяхната надеждност.
4. **Скалабилност и Поддръжка:** Възможността за мащабиране на базата данни в бъдеще и лесната поддръжка са също от решаващо значение за дългосрочната устойчивост на системата.

Проблем 2: Свързване на приложния слой с базата данни

Свързването на приложния слой с базата данни е един от основните при разработката на информационни системи. Този процес включва следните ключови аспекти:

1. **Интеграция:** Ефективната интеграция между приложния слой и базата данни е критична за гладката работа на системата. Това изисква правилното избиране и конфигуриране на технологии за достъп до данни, като *ORM (Object-Relational Mapping)* инструменти или специализирани библиотеки за бази данни.
2. **Производителност и оптимизация на заявките:** За да се гарантира висока производителност, заявките към базата данни трябва да бъдат оптимизирани. Това включва ефективно управление на ресурсите, избягване на излишни заявки и оптимизиране на схемите на базата данни.
3. **Транзакционност:** Системата трябва да гарантира целостността на данните и да поддържа съответните нива на транзакционност.

Проблем 3: Дефиниране на моделите и създаване на таблици от тях

Дефинирането на модели и преобразуването им в таблиците на базата данни е фундаментален етап в разработката на информационни системи. Този процес включва следните ключови аспекти:

1. **Моделиране на Данните:** Създаването на точни и ефективни модели е критично за представянето на бизнес логиката в структурата на данните. Моделите трябва да отразяват реалните същности и отношенията между тях, като същевременно оптимизират достъпа и обработката на данни.
2. **Преобразуване в табличен формат:** След като са дефинирани, моделите трябва да бъдат преобразувани в таблици в рамките на базата данни. Това включва определяне на подходящи типове данни, ключове (първични и чужди) и ограничения, които гарантират целостността и ефективността на данните.
3. **Нормализация и оптимизация:** Нормализацията на схемата на базата данни е важна за избягване на излишни данни и за увеличаване на ефективността. Оптимизацията на схемата и заявките е необходима за поддържане на висока производителност при увеличаване на обема на данни.

Проблем 4: Изграждане на трислойна архитектура (Repository -> Service -> Controller)

Изграждането на трислойна архитектура е фундаментален аспект при разработката на устойчиви и модулни информационни системи. Тази архитектура разделя системата на три основни слоя - *Repository* (Хранилище), *Service* (Услуга) и *Controller* (Контролер), като всеки от тях има своя отделна роля и отговорности:

1. **Repository Layer** (слой хранилище): Управлява директния достъп до базата данни, включително заявки, вмъквания, обновявания и изтривания на данни. Осигуряването на ефективен и оптимизиран достъп до данните, гарантиране на целостността на данните и абстрахиране на сложността на базата данни.
2. **Service Layer** (слой услуга): Отговорности: Съдържа бизнес логиката на приложението. Осъществява обработка на данни, извлечени от хранилището, и осигурява функционалността, изисквана от потребителския интерфейс. Балансиране между логиката на приложението и достъпа до данни, осигуряване на транзакционна сигурност и управление на бизнес правила.
3. **Controller Layer** (слой контролер): Управлява потребителския вход и изход, комуникира с услугите за извличане на информация или

изпълнение на бизнес операции и решава кой изглед да бъде показан на потребителя. Манипулация на FXML елементи.

Проблем 5: Прилагане на принципа за обръщане на зависимостите (Dependency Inversion Principle)

Прилагането на принципа за обръщане на зависимостите (Dependency Inversion Principle, DIP) е ключов аспект в създаването на гъвкава и поддържаема софтуерна архитектура. Този принцип е част от SOLID принципите за обектно-ориентирано програмиране и има за цел да намали взаимозависимостите между модулите на програмата. Принципът предполага, че модулите от високо ниво не трябва да зависят директно от модулите от ниско ниво, а и двете трябва да зависят от абстракции.

Проблем 6: Централизирано място за смяна на изгледите

Създаването на централизиран механизъм за смяна на изгледите (*views*) в приложението е важен аспект при разработката на потребителски интерфейси, особено в многомодулни приложения. Централизираното място за управление на изгледите позволява единен контрол над това, как потребителският интерфейс реагира на действията на потребителя, както и лесно превключване между различните изгледи и функционалности на приложението.

Проблем 7: Автентикация и Авторизация

Разработката на надеждни системи за автентикация и авторизация е критичен компонент в създаването на сигурни информационни системи. Автентикация е процесът на установяване на идентичността на потребителя, често чрез въвеждане на потребителско име/имейл и парола. Авторизация е процесът на определяне на правата на достъп на потребителя след успешната му автентикация.

Проблем 8: Тестване на кода - Unit и End-to-End тестове

Тестването на кода е критична стъпка в разработката на софтуер, осигуряваща качество, надеждност и устойчивост на приложението. Основните категории тестове, които трябва да бъдат разгледани, са unit (модулни) тестове и end-to-end (E2E) тестове. Тестване на отделни компоненти или модули на кода в изолация от останалата част на системата. Тестове, които оценяват поведението на цялата система от начало до край, обикновено като имитират реални потребителски сценарии.

Подходи, методи за решаване на поставените проблеми

Подход към Проблем 1: Избор на база данни

За решаване на този проблем бе избран подход за използване на *MySQL* като *RDS (Relational Database Service)* от *AWS*. *AWS RDS* предлага лесна скалабилност, което улеснява управлението на растежа на базата данни. *AWS* предоставя инструменти за управление на ресурсите, като мониторинг и автоматично мащабиране. Използват се вградените функции за сигурност (*security groups*) на *AWS*, за управление на достъпа до базата по клиентско *IP*.

Подход към Проблем 2: Свързване на приложния слой с базата данни

За решаване на този проблем, приложението използва *Hibernate* като зависимост. *Hibernate* е популярна *Object-Relational Mapping (ORM)* работна рамка за *Java* приложения. Той позволява разработчиците да превръщат *Java* обекти към таблици в релационна база данни и автоматизира голяма част от работата, свързана с трансформацията на данните от обектно ориентирания в релационен модел. *Hibernate* предоставя свой собствен език за заявки, *HQL*, който е ориентиран към обекти и позволява извършването на сложни заявки над данните. Поддържа управление на транзакции, което е критично за сигурността и целостността на данните. *Hibernate* позволява конфигуриране на мапингите между *Java* класове и таблици чрез анотации или *XML* файлове.

Подход към Проблем 3: Дефиниране на моделите и създаване на таблици от тях

Използвайки *ORM* работна рамка като *Hibernate*, можем да преобразуваме нашите *Java* обекти в таблиците на релационна база данни, което улеснява управлението на данните.

Класът *BaseEntity* служи като основа за всички модели, предоставяйки универсален идентификатор (*ID*), който се генерира автоматично от базата данни. Този подход осигурява уникалност и лесна възможност за търсене на записите в базата данни.

Специфични модели като *User*, *Agent*, *Owner*, и *Tenant* представят различните роли в системата и са свързани със съответните им функционалности и взаимоотношения, като например *Agent*, който има списък с *Review* и *WarehouseRentalRequest* обекти. Това отразява реалния свят, където агентите участват в обработката на наеми и получават отзиви от клиенти.

Адресите се управляват от *Address*, *City*, и *Country* модели, като всяко *Address* е асоциирано с *City*, а всяко *City* е асоциирано с *Country*. Това позволява

структуриран подход към управлението на адресните данни и лесна навигация и обработка на свързаните данни.

Моделите *Warehouse*, *RentalAgreement*, и *StorageType* управляват логиката свързана със складовите помещения, техните наемни договори и типове съхранение. Тези модели са от решаващо значение за бизнес логиката на системата и позволяват подробно управление на складовите ресурси. Създаването на таблици от тези модели включва дефиниране на взаимоотношенията между таблиците, като например външни ключове и асоциации (*OneToMany*, *ManyToOne*), които са критични за релационната модел на данните. Всяко *OneToMany* и *ManyToOne* отношение трябва да бъде внимателно разгледано и имплементирано, за да се осигури правилната целост и производителност на системата. Крайният резултат е релационна база данни, която точно отразява бизнес модела и логиката на приложението, като по този начин осигурява стабилна и мащабируема основа за управлението на складовите операции и потребителските взаимодействия в системата.

Подход към Проблем 4: Изграждане на трислойна архитектура

В контекста на нашата информационна система за управление на складове, тази архитектура позволява ясно разделение на отговорностите и по-лесно управление на кода.

1. Слой за Достъп до Данни (DAO): Описание: *CityDao*, *CountryDao*, *ReviewDao*, *UserDao*, и *WarehouseDao* са компоненти, които отговарят директно за взаимодействието с базата данни. Те абстрахират сложността на SQL операциите и предоставят чист интерфейс за останалата част от приложението.
2. Слой за Бизнес Логика (Service): Описание: *CityService*, *CountryService*, *CredentialManagerService*, *EncryptionService*, *PasswordHashingService*, *ReviewService*, *UserService*, и *WarehouseService* обработват бизнес правилата и операциите. Този слой агрегира логиката, необходима за извършване на операции над данните, които са извлечени или ще бъдат записани в базата данни. Тук се съдържат всички преобразувания, валидации и калкулации.
3. Презентационен Слой (Controllers): Описание: *BaseMenuController*, *BaseWarehouseDialogController*, *DialogAddReviewController*, *HomeController*, *LoginController*, *MyReviewsController*, *ProfileController*, *RegistrationController*, *RentalAgreementController*, *WarehouseControlPanelController*, *WarehouseUpdateDialogController*, и

Подход към Проблем 5: Прилагане на принципа за обръщане на зависимостите (Dependency Inversion Principle)

Принципът за обръщане на зависимостите (Dependency Inversion Principle, DIP) е приложен в ApplicationContext класа на нашата информационна система с цел повишаване на гъвкавостта и намаляване на взаимната зависимост между компонентите.

Класът ApplicationContext функционира като централизиран контейнер за създаване и управление на зависимостите в системата, което е ядрото на DIP. В контекста на DIP, ApplicationContext предоставя сингълтон инстанции на услуги като UserService, EncryptionService, CredentialManagerService, CityService, CountryService, WarehouseService, и ReviewService. Всяка от тези услуги е създадена със своите зависимости, като например UserService, която зависи от UserDao и PasswordHashingService. Този подход позволява на услугите да зависят само от абстракции, а не от конкретни реализации, което улеснява бъдещи промени и подобрения.

ControllerFactory е друг ключов компонент, който използва ламбда изрази за динамично създаване на контролери с инжектираните им зависимости. Това позволява лесно замяната и тестването на различни компоненти без промяна в самите контролери. Например, LoginController и HomeController зависят от UserService и CredentialManagerService, които са предоставени като зависимости, улеснявайки така тестването и разширяването. Този дизайн гарантира, че високите нива на абстракция не зависят от детайлите на ниските нива на абстракция. Той улеснява поддръжката и разширяването на системата, тъй като изменения в базата данни, логиката за кеширане или други детайли на имплементацията могат да бъдат извършвани с минимално въздействие върху останалата част от системата. По този начин, принципът на обръщане на зависимостите допринася за създаването на по-издръжлива и модулна архитектура.

Подход към Проблем 6: Централизирано място за смяна на изгледите

Класът ViewLoaderUtil е създаден, за да отговори на тази нужда в нашата система, като предоставя унифициран интерфейс за зареждане и показване на различни изгледи на приложението. Използвайки JavaFX, ViewLoaderUtil улеснява прехода между различните екрани в приложението. Методите loadView действат като централизирани точки за зареждане на FXML файлове, които са графичните описания на потребителските интерфейси, и установяване на новия изглед в текущия прозорец или сцена. Този подход гарантира, че всяка смяна на изгледа се обработва консистентно, което намалява риска от грешки и улеснява поддръжката на кода. Методът showAlert осигурява стандартизиран начин за показване на съобщения до потребителя, като например грешки или

информационни диалози. Тази функционалност е важна за предоставяне на обратна връзка и подобряване на потребителското изживяване. Когато се случи грешка при зареждането на изгледа, `handleLoadViewException` обработва изключението и регистрира събитието, което улеснява отстраняването на проблеми. Логването на грешките е критично за бързата им диагностика и корекция.

Подход към Проблем 7: Автентикация и Авторизация

В нашата система за управление на складове, процесите на автентикация и авторизация се обработват чрез събитийно-ориентирани методи, които са част от потребителския интерфейс, управляван от JavaFX контролери.

Методът `handleLogin` се използва за проверка на потребителските данни. Потребителят въвежда своя имейл и парола, които след това се проверяват от `userService`. При успешна автентикация, `credentialManagerService` се опитва да запази удостоверителните данни за бъдеща автоматична входна сесия. В случай на грешка, потребителят е информиран и е необходимо ръчно входно действие при следващия вход.

Допълнително, методът `handleSsoAction` позволява на потребителите да влязат чрез Single Sign-On (SSO), където удостоверителните данни от предишни сесии се използват за автоматичен вход. Ако тези удостоверителни данни липсват или са изтекли, системата предоставя подходящо съобщение за грешка. Функцията `buildMenu` се грижи за авторизацията на потребителите, като проверява техната роля и предоставя съответните потребителски интерфейси. Това се реализира чрез ламбда изрази и предикати, които динамично добавят или премахват бутони от страничното меню в зависимост от ролята на потребителя - например, показване на "My Warehouses" само за собственици на складове или "My Reviews" за агенти.

Подход към Проблем 8: Тестване на кода - Unit и End-to-End тестове

В нашата система за управление на складови помещения, тестването е организирано на две основни нива: Unit тестове и End-to-End (E2E) тестове, което е видно в структурата на проектната директория. Класификацията на тестовите в поддиректории като `controllers`, `dao`, `service`, и `util` осигурява чистота и организация, улеснява намирането и поддръжката на тестове, и подчертава важността на тестването в рамките на всеки слой на приложението. За всеки компонент, Unit тестовите осигуряват детайлно покритие на функционалностите, докато E2E тестовите гарантират, че потребителските потоци са гладки и безпроблемни.

Потребителски (функционални) изисквания (права, роли, статуси, диаграми, ...) и качествени (нефункционални) изисквания (скалируемост, поддръжка, ...)

Функционални изисквания:

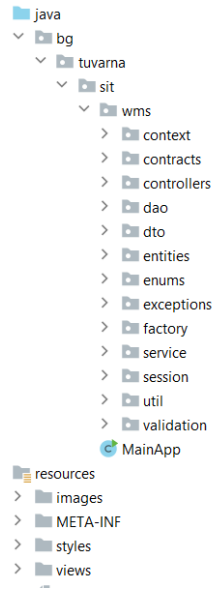
Информационната система за управление на складови помещения е проектирана да обслужва широк спектър от потребители, включително администратори, складови агенти и собственици на складове. Системата предоставя функционалности за създаване и управление на профили за всеки тип потребител, като администраторите имат възможността да създават акаунти за собственици и складови агенти. Собствениците могат да регистрират нови складови помещения, да определят параметри като размер, климатични условия и вид на стоките за съхранение, и да избират складови агенти за управление на помещенията. Складовите агенти, от своя страна, могат да отдават помещения под наем и да създават наемни формуляри с детайли за наемателя, срока на наема и цената. Системата също така позволява рейтинговане на складовите агенти и предоставя различни отчети за сключените договори, налични складове за наем и справки за активностите на агентите и собствениците.

Нефункционални изисквания:

Относно нефункционалните изисквания, системата трябва да е максимално надеждна, с висока производителност и да поддържа множествен достъп от множество потребители едновременно. Трябва да се грижи правилен формат и вид на данните посредством методи и инструменти за валидация на входния поток и действията на потребителите. Системата изисква ефективно управление на сесиите и транзакциите, гарантирайки консистентност на данните при всякакви операции. Скалабилността също е от критично значение, тъй като системата трябва да може да се адаптира към растящ брой потребители и увеличаващ обем на данни. Освен това, системата трябва да предлага интуитивен потребителски интерфейс и лесна за използване навигация, което изисква съобразяване с най-добрите практики за UX/UI дизайн.

Глава 3. Проектиране.

Обща структура на проекта пакети, които ще се реализират.



Структурата на пакетите в софтуерния проект е организирана по начин, който улеснява разбирането и поддръжката на кода. В контекста на проекта, структурата на пакетите представя ясно разделение на отговорностите и функционалностите в приложението:

context: Този пакет съдържа клас, който управлява контекста на приложението (управление на зависимости).

contracts: В този пакет могат да бъдат дефинирани интерфейси или контракти, които специфицират очакваното поведение на компонентите, което насърчава слаба свързаност и лесно тестване.

controllers: Съдържа контролерите, които служат като посредници между потребителския интерфейс и бизнес логиката, обработвайки входящи потребителски заявки и връщайки отговори.

dao (Data Access Object): Пакетът включва класове, отговорни за взаимодействие с базата данни, предоставяйки абстракция над SQL заявките.

dto (Data Transfer Object): Тук са дефинирани обекти, които се използват за пренос на данни между слоевете на приложението, улеснявайки сериализацията и десериализацията на информацията.

entities: Съдържа класове, които отговарят на бизнес модела и се картират директно върху таблиците в базата данни.

enums: Дефинира изброими типове (*enumerations*), които представят константни стойности, използвани в приложението за представяне на ограничен набор от възможни стойности, като статуси или роли.

exceptions: Този пакет може да съдържа персонализирани изключения, които улесняват обработката на грешки и повишават четливостта на кода при възникване на проблеми.

factory: Предоставя фабричен клас за създаване на инстанции на обекти, което е честа практика в дизайн шаблоните за създаване на обекти.

service: Съдържа сервизни класове, които имплементират бизнес логиката и обработката на данните, отделени от останалите компоненти.

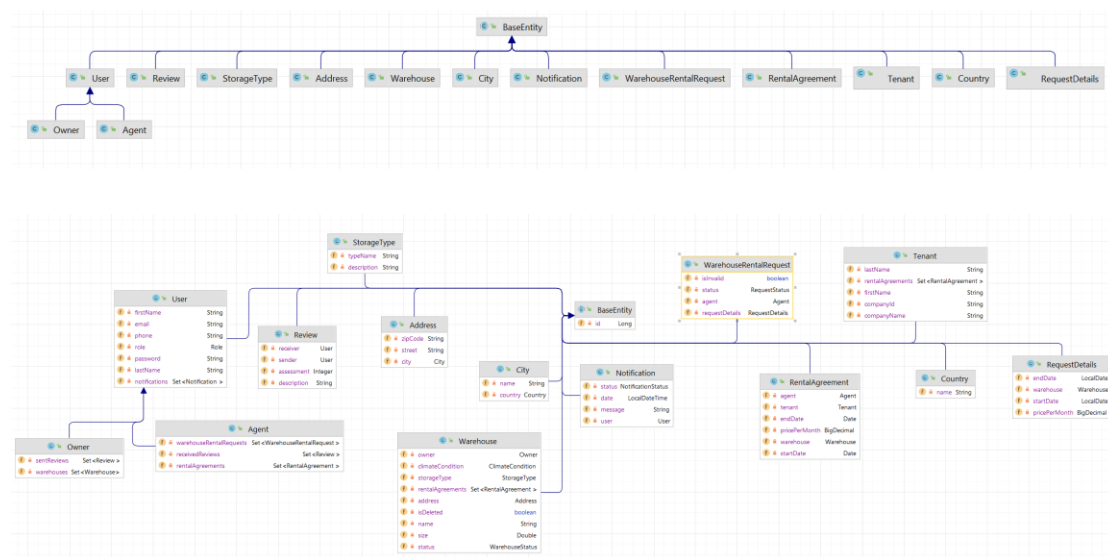
session: Този пакет може да управлява сесиите на потребителите, осигурявайки съхранение и управление на данни в текущата сесия.

util: Съдържа помощни класове, които предоставят общи функционалности, използвани от различни части на приложението.

validation: Включва класове и методи за валидация на данните,

Директорията *resources* обикновено съдържа конфигурационни файлове, XML дефиниции, шаблони, както и всички други ресурси, необходими за изпълнението на приложението, които не са Java код. Специалната папка *META-INF* е предназначена за метаданни, като например манифести и конфигурации на *JPA (Java Persistence API)*, които се използват от Java EE сървърите и контейнерите.

Диаграми/Блок схеми (на структура и поведение - по обекти, слоеве с най-важните извадки от кода)



В центъра на диаграмата е класът BaseEntity, който служи като основен родителски клас, предоставящ общо ID поле. От BaseEntity наследяват няколко други класа, които са специфични за бизнес домейна на системата:

User: Този клас моделира общата информация за потребителите на системата и е родител на Owner, Tenant и Agent. Това показва, че има различни типове потребители с различни роли и отговорности в системата.

Owner, Tenant, Agent: Тези класове представляват различни роли в системата, като всеки клас съдържа специфични полета и методи, които са уникални за своята роля.

Country и City: Тези класове управляват адресната структура и могат да бъдат свързани с други класове, като например Address, който моделира адресите на складовете.

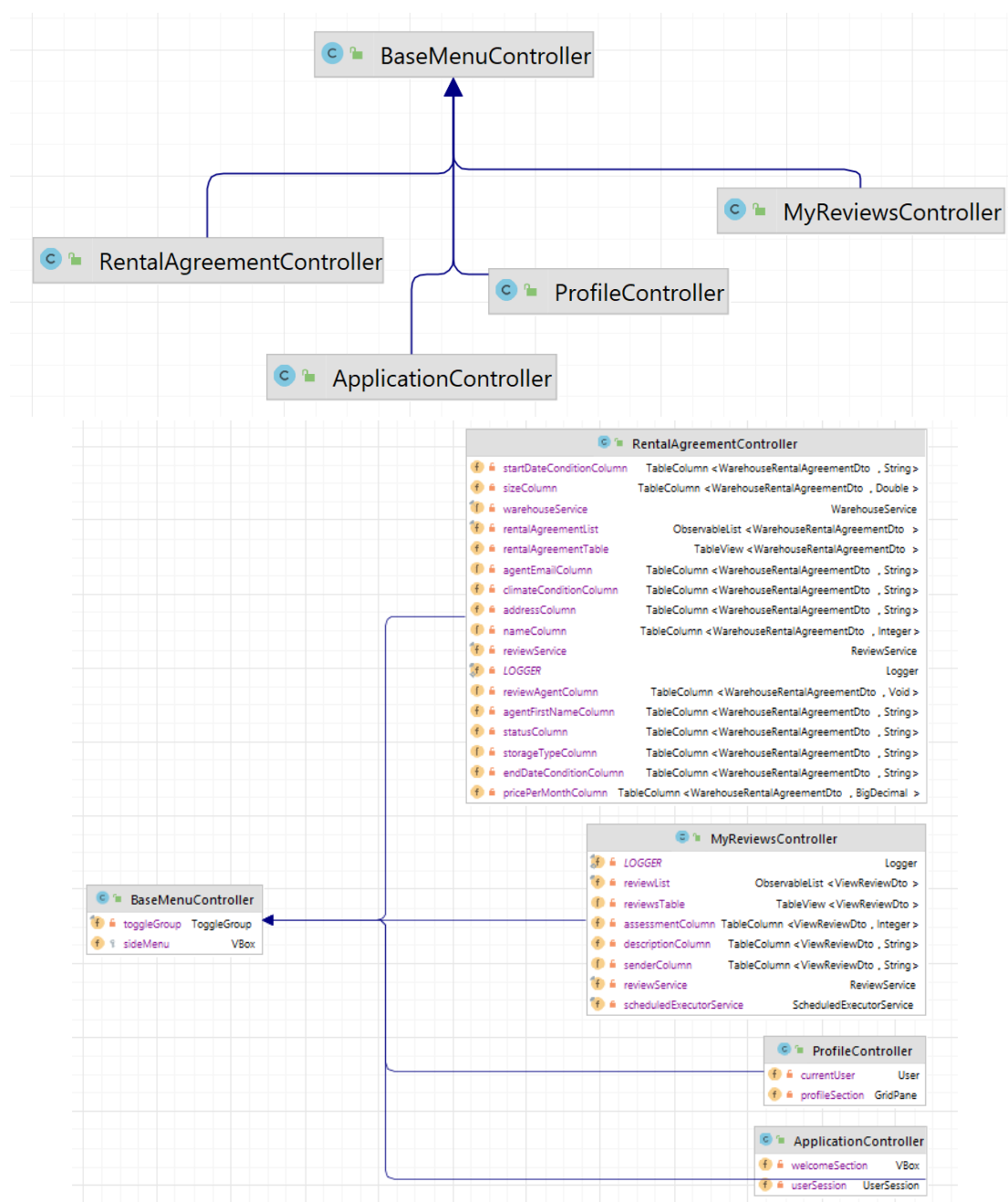
Warehouse: Съдържа информация за самите складови помещения, като размер, адрес и съхраняваните стоки, и е свързан с класовете, описващи наемните договори (RentalAgreement) и типовете съхранение (StorageType).

RentalAgreement: Представява договорите за наем на складови помещения и съдържа информация за сроковете и условията на наема.

WarehouseRentalRequest: Представява заявките за наем на складови помещения, които трябва да бъдат одобрени от собственика или управляващия агент.

Notification и Review: Тези класове могат да бъдат използвани за управление на уведомленията, които се изпращат до потребителите, и за ревюта или оценки, които потребителите могат да дават за агентите.

Тази диаграма показва, че системата е проектирана с ясно разграничение на ролите и отговорностите, което позволява лесно разширение и модификация на функционалностите.



Тази класова диаграма представлява част от структурата на JavaFX приложението, показвайки различните контролери и техните асоциации с различни UI компоненти и услуги.

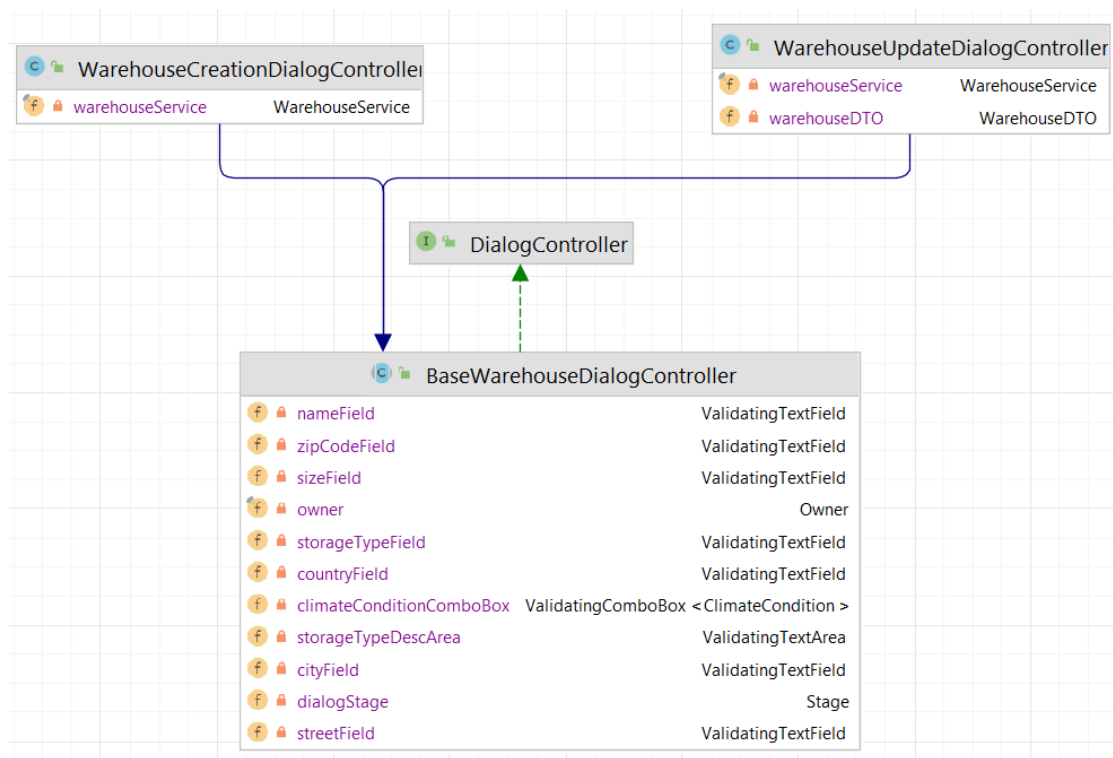
BaseMenuController: Управлява базовото меню на приложението. Съдържа **ToggleGroup**, което се използва за създаването на взаимосвързани меню бутони, и **VBox**, което е контейнер за вертикално подреждане на UI елементи в менюто.

RentalAgreementController: Управлява логиката зад визуализацията и взаимодействието със складовите наемни договори. Той съдържа редица **TableColumn** обекти, които са асоциирани с различни видове данни, за показване в таблица на графичния интерфейс. Тези колони обхващат датата на началото на наема, размера на склада, имейла на агента и други. Така се улеснява визуализацията на детайли, специфични за наемните договори, в таблица.

MyReviewsController: Отговаря за управлението на отзивите, които агентите получават. Също така включва **ObservableList**, което е използвано за съхранение и наблюдение на колекция от обекти **ReviewDto**, които се показват в **reviewsTable**. Това подпомага динамичното обновление на UI при промени в данните.

ProfileController: Управлява потребителския профил. Съдържа препратки към инстанции на **User**, което представлява текущо влезлия потребител, и **GridPane**, което се използва за организиране на UI компонентите на профила.

ApplicationController: Управлява началния потребителски интерфейс след успешна автентикация. Той включва **VBox**, за организиране на секции в приложението, и **UserSession**, за управление на сесията на потребителя.



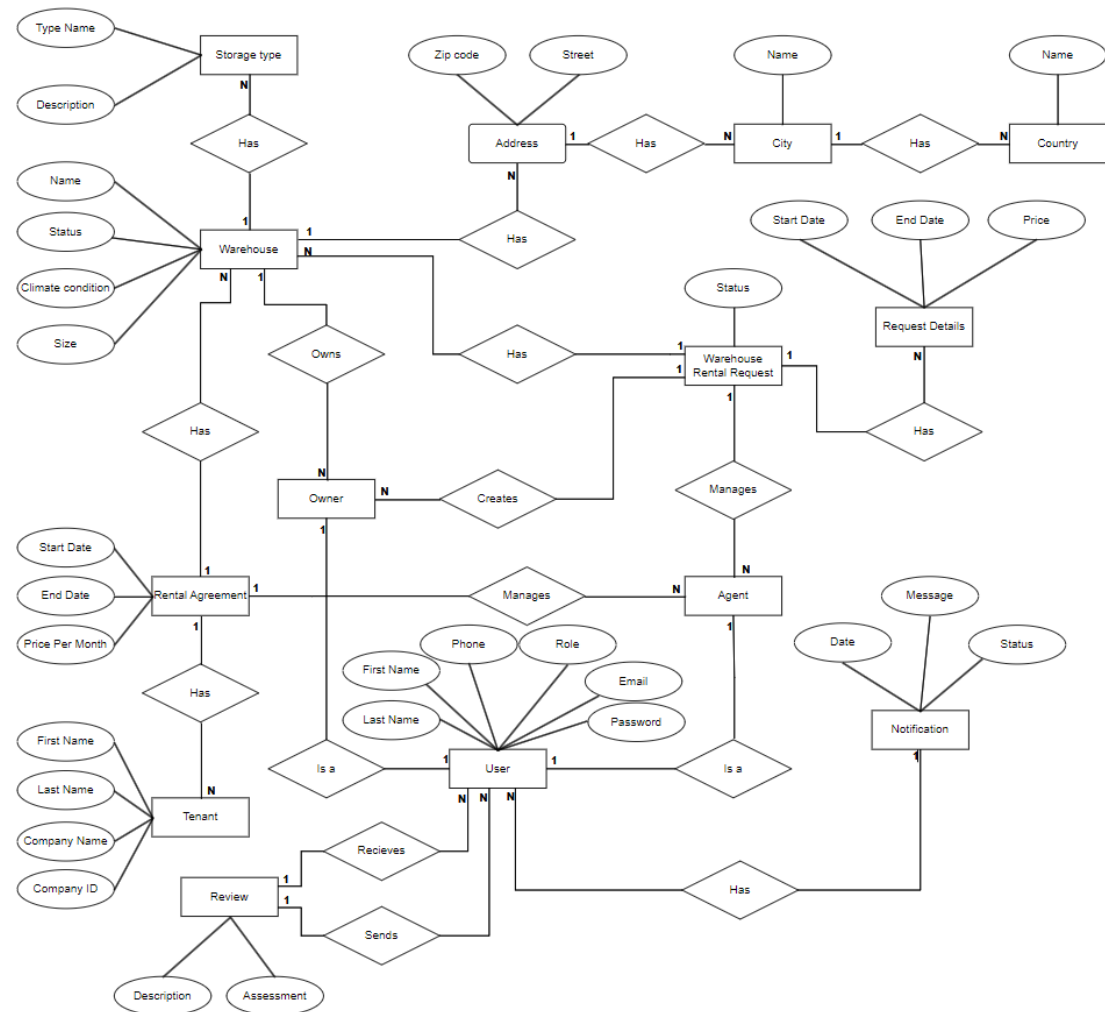
Тази класова диаграма изобразява структурата на два контролера, които са свързани със създаването и актуализирането на информацията за складове.

WarehouseCreationDialogController: Този контролер управлява логиката на потребителския интерфейс за създаването на нов склад. Той включва референция към `WarehouseService`, който е клас от сервизния слой и се използва за комуникация със задната част на приложението за обработка на бизнес логика, свързана със складовете. В контролера са дефинирани различни полета като `nameField`, `zipCodeField`, `sizeField`, които са свързани с графични компоненти в потребителския интерфейс за въвеждане на съответните данни за нов склад.

WarehouseUpdateDialogController: Този контролер е отговорен за обновяването на данните за вече съществуващ склад. Той също има връзка с `WarehouseService` за бизнес логиката, както и с `WarehouseDTO`, което е обект за пренос на данни (Data Transfer Object) и съдържа информацията, която ще бъде актуализирана.

И двата контролера наследяват *BaseWarehouseDialogController*, който предоставя общи функционалности и компоненти, които са общи както за процеса на създаване, така и за актуализиране на складове. Този базов клас съдържа редица полета за валидиране, като *ValidatingTextField* и *ValidatingComboBox*, които осигуряват валидация на входните данни от потребителя и помагат за намаляване на възможността за въвеждане на невалидни данни.

Такава структура на контролерите позволява разделение на отговорностите в приложението, повторна употреба на код и по-лесна поддръжка, като същевременно подобрява цялостната управляемост на приложението.



Глава 4. Важни моменти при реализация, алгоритми, оптимизации.

ApplicationContext клас за инстанциране на всички компоненти на приложението. Прилагане на принципа за обръщане на зависимостите.

Класът `ApplicationContext` играе ключова роля в приложението като централна точка за инициализация и предоставяне на достъп до различни услуги и обектите за достъп до данни (*Data Access Objects, DAO*). Този клас е фундаментален компонент в архитектурата на приложението, осъществявайки принципите на инверсия на контрола (*Inversion of Control, IoC*) и вмъкване на зависимости (*Dependency Injection, DI*).

```
public class ApplicationContext {

    private static final UserDao USER_DAO =
        new UserDao(JpaUtil.getEntityManagerFactory());

    private static final ReviewDao REVIEW_DAO =
        new ReviewDao(JpaUtil.getEntityManagerFactory());

    @Getter
    private static final UserService USER_SERVICE =
        new UserService(USER_DAO, new
        PasswordHashingService());

    @Getter
    private static final EncryptionService ENCRYPTION_SERVICE =
        new EncryptionService();

    @Getter
    private static final CredentialManagerService
    CREDENTIAL_MANAGER_SERVICE =
        new CredentialManagerService(ENCRYPTION_SERVICE);

    @Getter
    private static final CityService CITY_SERVICE =
        new CityService(new
        CityDAO(JpaUtil.getEntityManagerFactory()));

    @Getter
    private static final CountryService COUNTRY_SERVICE =
        new CountryService(new
        CountryDAO(JpaUtil.getEntityManagerFactory()));

    @Getter
    private static final WarehouseService WAREHOUSE_SERVICE =
        new WarehouseService(new
```

```

WarehouseDAO(JpaUtil.getEntityManagerFactory()),
                COUNTRY_SERVICE, CITY_SERVICE);

@Getter
private static final ReviewService REVIEW_SERVICE =
    new ReviewService(USER_DAO, REVIEW_DAO);

@Getter
private static final ControllerFactory CONTROLLER_FACTORY =
    createControllerFactory();

private static final ScheduledExecutorService
SCHEDULED_EXECUTOR_SERVICE =
    Executors.newSingleThreadScheduledExecutor();

private static ControllerFactory createControllerFactory() {

    ControllerFactory factory = new ControllerFactory();
    factory.addController(LoginController.class, () -> new
LoginController(USER_SERVICE, CREDENTIAL_MANAGER_SERVICE));
    factory.addController(HomeController.class, () -> new
HomeController(USER_SERVICE, CREDENTIAL_MANAGER_SERVICE));
    factory.addController(RegistrationController.class, () ->
new RegistrationController(USER_SERVICE));

    factory.addController(WarehouseControlPanelController.class,
    () -> new WarehouseControlPanelController(WAREHOUSE_SERVICE));
    factory.addController(MyReviewsController.class, () -> new
MyReviewsController(REVIEW_SERVICE,
SCHEDULED_EXECUTOR_SERVICE));
    factory.addController(RentalAgreementController.class, ()
-> new RentalAgreementController(WAREHOUSE_SERVICE,
REVIEW_SERVICE));

    return factory;
}
}

```

Създаване на общо базово меню с възможност за динамично добавяне на бутони в зависимост от ролята на текущо влезият потребител.

Методът *buildMenu*, дефиниран в класа *BaseMenuController*, представлява логиката за динамично създаване на навигационно меню, базирано на ролите на потребителите в приложението. Методът използва предикати, които са функционални интерфейси предоставящи тестова логика, за определяне на видимостта и достъпността на различни меню бутони в зависимост от ролята на текущия потребител. За всяка роля - OWNER, ADMIN и AGENT - се създават

предикати, като например *isOwnerPredicate*, които проверяват дали текущият потребител има съответната роля. Ако проверката е успешна (предикатът връща *true*), съответният бутон се добавя към менюто.

```
public class BaseMenuController {

    ...

    protected void buildMenu() {

        Predicate<User> isOwnerPredicate = user ->
        Role.OWNER.equals(user.getRole());    addToggleButtonToPane(
            createToggleButton("My Warehouses", e ->
        LoadView("/views/warehouseControlPanel.fxml", e)),
            sideMenu, isOwnerPredicate);

        addToggleButtonToPane(
            createToggleButton("My Profile", e ->
        LoadView("/views/profile.fxml", e)),
            sideMenu);

        Predicate<User> isAdminPredicate = user ->
        Role.ADMIN.equals(user.getRole());
        addToggleButtonToPane(createToggleButton("Register", e ->
        LoadView("/views/registration.fxml", e),
            Optional.of("registerButton")), sideMenu,
        isAdminPredicate);

        Predicate<User> isAgentPredicate = user ->
        Role.AGENT.equals(user.getRole());
        addToggleButtonToPane(createToggleButton("My Reviews", e -
        > LoadView("/views/reviews.fxml", e)),
            sideMenu, isAgentPredicate);

        addToggleButtonToPane(createToggleButton("My Rentals", e -
        > LoadView("/views/rental-agreements.fxml", e)),
            sideMenu, isOwnerPredicate);

        addToggleButtonToPane(createToggleButton("Logout",
        this::handleLogoutAction, Optional.of("logoutButton"),
            Optional.of("button-logout")), sideMenu);
    }

    ...
}
```

Запазване на текущо логнатият потребител в сесия с възможност за отписване.

Класът *UserSession* е реализация на шаблона за дизайн "*Singleton*", който се използва за управление на потребителската сесия в приложението. Този шаблон осигурява, че ще съществува само една инстанция на класа *UserSession* в рамките на цялото приложение, като по този начин се гарантира централизирано управление на данните за текущата потребителска сесия. Полето *currentUser* държи информация за текущо влезлия в системата потребител. Това позволява на приложението да следи кой потребител е активен в даден момент. Методът *setCurrentUser(User user)* се използва за задаване на текущия потребител, който е влязъл в системата, докато методът *logout()* осъществява изход на потребителя, като изчиства информацията за него.

@Getter

```
public class UserSession {

    private User currentUser;

    private UserSession() {
    }

    private static class Holder {
        static final UserSession INSTANCE = new UserSession();
    }

    public static UserSession getInstance() {
        return Holder.INSTANCE;
    }

    public void setCurrentUser(User user) {
        this.currentUser = user;
    }

    public void logout() {
        currentUser = null;
    }
}
```

Валидация на полета при попълване на формуляри за създаване на нови обекти.

В контекста на диалоговия прозорец, контролерът съдържа няколко полета, които са анотирани с *@FXML*. Това са *ValidatingTextField* и *ValidatingComboBox* обекти, които представляват текстови полета и падащи менюта в графичния интерфейс, които са оборудвани с функционалност за валидация на входните данни. Тези полета са свързани с различни атрибути на склада, като име, улица, град, пощенски код, страна, размер, климатични условия, тип съхранение и описание на типа съхранение.

Методът *initialize* се изпълнява автоматично от *JavaFX*, когато *FXML* файлът е зареден и се използва за начална настройка на контролера. В *initialize*, всички валидационни полета са настроени с конкретни ламбда изрази, които определят правилата за валидация на всяко поле. Например, *nameField* трябва да съдържа само букви, цифри и основни символи, докато *cityField* трябва да започва с главна буква. Това осигурява, че въведените данни от потребителя отговарят на определените формати и критерии преди да бъдат обработени или записани. Този подход осигурява силен механизъм за защита на приложението от невалидни данни, като едновременно с това подобрява потребителското изживяване чрез предоставяне на незабавна обратна връзка при въвеждането на данни.

```
public abstract class BaseWarehouseDialogController implements
DialogController {

    @FXML
    private ValidatingTextField nameField;
    @FXML
    private ValidatingTextField streetField;
    @FXML
    private ValidatingTextField cityField;
    @FXML
    private ValidatingTextField zipCodeField;
    @FXML
    private ValidatingTextField countryField;
    @FXML
    private ValidatingTextField sizeField;
    @FXML
    private ValidatingComboBox<ClimateCondition>
climateConditionComboBox;
    @FXML
    private ValidatingTextField storageTypeField;
    @FXML
    private ValidatingTextArea storageTypeDescArea;
```



```

...

@FXML
public void initialize() {

    nameField.setUp(value -> value.matches("[a-zA-Z0-9\\-
_,\\'()*\\[\\]\\#;<> ]+$"), "Warehouse name can contain only
letters, numbers and basic symbols");
    streetField.setUp(value -> value.matches("[a-zA-Z0-9
.,\\'\"#\\-]+$"), "Invalid street format");
    cityField.setUp(value -> value.matches("[A-Z][A-Za-z\\s-
]+$"), "City must start with an uppercase letter");
    zipCodeField.setUp(value -> value.matches("[0-9A-Za-z\\s-
]{3,10}$"), "Invalid zip code format");
    countryField.setUp(value -> value.matches("[A-Z][A-Za-
z\\s']+$"), "Country must start with an uppercase letter");
    sizeField.setUp(value -> value.matches("[1-
9]\\d*(\\.\\d+)?$"), "Size must be a valid number");
    climateConditionComboBox.setUp("Climate condition cannot
be empty");
    storageTypeField.setUp(value -> value.matches("[a-zA-Z0-
9\\s.,#()\\-]+$"), "Invalid storage type format");
    storageTypeDescArea.setUp(value -> value.length() <= 400,
"Description must be 400 characters maximum");

    climateConditionComboBox.getItems().setAll(ClimateCondition.va
lues());
}
...
}

```
















Динамично извличане на информацията от база данни без презареждане на изгледа.

Периодично изпълняваща се задача за обновяване или презареждане на данните в приложението. Той използва *ScheduledExecutorService*, който позволява насрочването на команди за изпълнение след определено закъснение или периодично. В конкретния контекст, методът настройва задачата да се изпълнява с нулево закъснение (0), което означава, че задачата ще стартира веднага щом методът бъде извикан. След това задачата ще се повтаря на всеки 10 секунди, както е указано от параметрите на метода *scheduleAtFixedRate*. Изпълнението в рамките на задачата се извършва във JavaFX Application Thread, благодарение на Platform.runLater. Това е необходимо, защото актуализациите

на потребителския интерфейс трябва да се извършват в този основен нишка на графичния интерфейс, за да се избегнат проблеми със синхронизацията и да се гарантира, че потребителският интерфейс се обновява правилно и консистентно. Методът `loadReviews`, който се извиква от `Platform.runLater`, е метод, който зарежда и показва отзивите в потребителския интерфейс на приложението. Чрез периодичното извикване на този метод, приложението гарантира, че потребителският интерфейс ще показва най-актуалната информация без необходимост потребителят ръчно да инициира обновяване.

```
private void startPeriodicRefresh() {  
  
    scheduledExecutorService.scheduleAtFixedRate(() ->  
        Platform.runLater(this::loadReviews), 0, 10,  
        TimeUnit.SECONDS);  
}
```

Глава 5. Работа с програмата. Изисквания за изпълнение на програмата.

 build	8.1.2024 г. 12:50	File folder	
 gradle	27.12.2023 г. 12:00	File folder	
 logs	26.11.2023 г. 11:27	File folder	
 src	27.12.2023 г. 12:00	File folder	
 .gitignore	8.1.2024 г. 16:35	Text Document	1 KB
 build.gradle	8.1.2024 г. 16:30	IntelliJ IDEA	4 KB
 credentials.txt	8.1.2024 г. 20:35	Text Document	1 KB
 credentials-test.txt	27.12.2023 г. 21:22	Text Document	1 KB
 encryption.key	27.12.2023 г. 23:19	KEY File	1 KB
 env	24.12.2023 г. 11:19	File	1 KB
 env-test	24.12.2023 г. 11:19	File	1 KB
 gradle.properties	27.12.2023 г. 12:00	PROPERTIES File	1 KB
 gradlew	27.12.2023 г. 12:00	File	9 KB
 gradlew.bat	27.12.2023 г. 12:00	Windows Batch File	3 KB
 reviews.csv	8.1.2024 г. 20:34	Microsoft Excel Co...	1 KB
 settings.gradle	27.12.2023 г. 12:00	IntelliJ IDEA	1 KB
 start.bat	8.1.2024 г. 13:20	Windows Batch File	1 KB
 start.sh	8.1.2024 г. 13:14	Shell Script	1 KB
 users.csv	8.1.2024 г. 20:32	Microsoft Excel Co...	1 KB
 warehouses.csv	8.1.2024 г. 16:30	Microsoft Excel Co...	1 KB

Ето описание на основните компоненти:

build: Директория създадена от Gradle, където се намират всички изходни файлове от процеса на компилиране и други генерирани ресурси.

gradle: Съдържа Gradle wrapper файлове, които позволяват използването на Gradle без да е нужна предварителната му инсталация.

logs: Директория, която съдържа лог файлове с информация за изпълнението на приложението.

src: Директорията за изходния код, която обикновено съдържа поддиректориите `main` и `test` за основния и тестовия код съответно.

.gitignore: Текстов файл, който указва на Git кои файлове или директории да игнорира при версионирването.

build.gradle: Основен конфигурационен файл за Gradle, където се определят зависимости, плъгини и други настройки за сглобяването на проекта.

credentials.txt, credentials-test.txt: Текстови файлове, съдържащи удостоверителни данни, използвани за автентикация в различни среди.

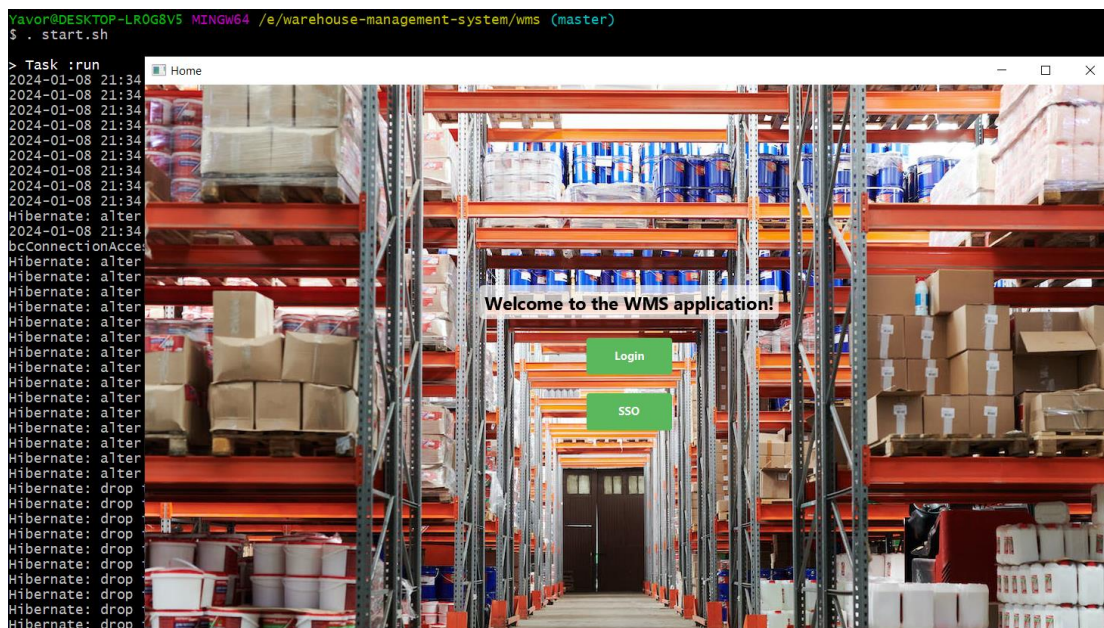
encryption.key: Файл съдържащ ключ за криптиране, който се използва от приложението за шифроване на данни.

env, env-test: Файлове, които вероятно съдържат конфигурационни променливи за различни среди на изпълнение. gradle.properties, settings.gradle: Конфигурационни файлове за Gradle, където се задават глобални пропърти и настройки за проекта.

gradlew, gradlew.bat: Изпълними скриптове за Linux и Windows, съответно, за стартиране на Gradle wrapper.

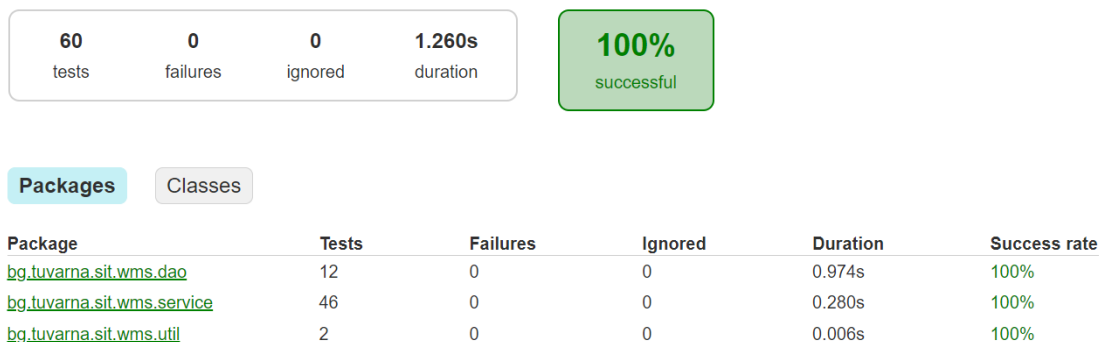
reviews.csv, users.csv, warehouses.csv: CSV файлове, които могат да съдържат данни за ревютата, потребителите и складовете, вероятно използвани от приложението за инициализиране или тестови цели.

start.bat, start.sh: Скриптове за стартиране на приложението в Windows и Unix-базирани системи.



Глава 6. Тестови сценарии.

Test Summary



Снимката показва структурата на директорията test в Java проект, която е предназначена за съхранение на тестови класове.

Тази директория е част от проект, който следва стандартната конвенция на Gradle за разпределение на изходния код и тестовете в различни директории.

В директорията test, класовете са подредени по пакети, които отразяват структурата на основния изходен код в директорията src.

bg.tuvarna.sit.wms.controllers: Съдържа тестови класове за контролерите в приложението, като например `ApplicationControllerTest`, `HomeControllerTest`, `LoginControllerTest`, и `RegistrationControllerTest`. Тези тестове имитират потребителското взаимодействие и проверяват логиката на управление на потребителския интерфейс.

bg.tuvarna.sit.wms.dao: Включва тестови класове за DAO (Data Access Object) слой, като `ReviewDaoTest`, `UserDaoTest`, и `WarehouseDAOTest`. Тестовете в този пакет се фокусират върху интеграцията с базата данни и правилното извличане и запис на данни.

bg.tuvarna.sit.wms.service: Съдържа тестове за сервизния слой на приложението, където бизнес логиката се обработва. Класовете като `CityServiceTest`, `CountryServiceTest`, и `UserServiceTest` проверяват бизнес операциите и валидациите, които сервизите извършват.

bg.tuvarna.sit.wms.util: Включва утилитни тестове като `KeyUtilTest`, които вероятно проверяват помощни инструменти, използвани от другите компоненти на системата.

resources/META-INF: Съдържа ресурсни файлове, които са свързани с тестовете, например `persistence.xml`, който може да дефинира конфигурацията на JPA (Java Persistence API) за тестовата среда.