**Szegedi Tudományegyetem**

**Informatikai Intézet**

# SZAKDOLGOZAT

**Urban Roland**

**2020**

**Szegedi Tudományegyetem**
**Informatikai Intézet**

# CRM rendszer megvalósítása Spring Boot keretrendszerrel

Szakdolgozat

Készítette:
**Urban Roland**
gazdaságinformatikus
hallgató

Témavezető:
**Dr. Bodnár Péter**
adjuntus

Szeged
2020

# 1. Feladatkiírás

A feladat egy CRM rendszer megtervezése és implementálása. A webes alkalmazásban tárolt adatok egy tetszőlegesen választott adatbázisban legyenek elhelyezve. A projekt tetszőleges nyelven elkészíthető.

A rendszerrel szemben támasztott főbb követelmények az alábbiak:

- Képesnek kell lennie a hibajegyek feltöltésére, tárolására és annak megjelenítésére.
- Meg kell tudnia különböztetni a hibajegyeket státuszuk szerint.
- Lehessen regisztrálni, amivel személyre szabott adatok érhetőek el.
- A hibajegy prioritásának kezelése.
- A hiba továbbítása a megfelelő csoport számára.
- Keresés a hibák és az ügyfelek közt
- Statisztikai kimutatások készítése.
- Képes legyen kezelni a hibajegyekhez rendelt adatokat:
- A hibabejegyzés életciklusának kezelése, nyomon követése.
- Támogassa a felhasználók kijelentkezését a rendszerből.

# 2. Tartalmi összefoglaló

**A téma megnevezése:**

CRM webes alkalmazás fejlesztése

**A megadott feladat megfogalmazása:**

A cél az volt, hogy egy működő CRM rendszert készítsek a megadott szempontok alapján. A webalkalmazásban használt adatokat egy adatbázisban kellett eltárolni. Tetszőlegesen választható volt, hogy milyen programozási nyelven implementálom.

**A megoldási mód:**

A szakdolgozat készítése során kipróbálhattam magam egy komolyabb projektmunkában, kiderült, hogy mennyire sikerül alkalmaznom az egyetemen megszerzett tudást, valamint, hogy el tudok-e sajátítani eddig számomra nemismert technológiákat is. Úgy gondolom, hogy ezt is teljes mértékben megvalósítottam.

**Alkalmazott eszközök, módszerek:**

Sikerült elsajátítanom az általam korábban nem ismert technológiákat backend részen Spring, JPA, Hibernate, frontend thymelaf technológiát.

Módszereknél külön kiemelném az MVC architektúrát.

**Elért eredmények:**

Úgy gondolom, hogy a dolgozatban kitűzött célomat sikerült elérnem, hiszen a Spring Boot segítségével létrehoztam a magam által megtervezett CRM rendszert. Sikerült egy áttekinthető, és könnyedén tovább fejleszthető alkalmazást elkészítenem.
Mindent összevetve elmondható, hogy hasznosnak érzem az alkalmazás elkészítését, mivel implementációja során rengeteg, számomra új ismeretet szerezhettem, és alkalmazhattam. Munkám eredményeképpen pedig egy átfogó képet kaptam a webalkalmazások fejlesztésről.

**Kulcsszavak:**

- Spring
- Spring Boot
- Java
- JPA
- CRM

# Tartalomjegyzék

# 3. Bevezetés

Az Internet a megjelenése óta folyamatosan fejlődik, és a fejlődés máig szinte semmit sem veszített kezdeti ütemében. A fejlődésnek értelemszerűen ára van és ennek az összegnek a túlnyomó részét különböz vállalkozások milliói teremtik elő azáltal, hogy befektetnek olyan megoldásokba, amelyek az interneten bárki által elérhetők. Ezek a befektetések jórészt valamilyen weben elérhető alkalmazást jelentenek, ebből kifolyólag azt mondhatjuk, hogy jelenleg az Internet és ezen belül a web fejlődésének az a kulcsa, hogy óriási piaci igény alakul ki a webalkalmazások iránt. Manapság az Internet mára egy világméretű hálózattá nőte ki magát, amit több millió ember használ. Ahogy növekedett, úgy jelentek meg újabb szabványok, technológiák és ajánlások. Napjainkban, nem csak szükséges, hanem elengedhetetlen, a használat mind a hétköznapi felhasználóknak, mind a vállalatoknak is.

A web-es technológiák már régóta érdekelnek így amikor a szakdolgozati témaválasztásra került a sor, számomra egyértelmű volt, hogy olyat jelölök meg ami része ennek a nagy rendszernek. Mindig is fontos volt, hogy olyan technológiákat ismerjek meg, melyek napjainkban is helytállnak és a jövőben is helyt fognak. A Szegedi Tudományegyetem folytatott tanulmányaim alatt megismerkedtem rengeteg ilyennel, de ezek közül kiemelném a Java-t, melyet a leggyakrabban használtam nem csak az egyes kurzusokhoz, hanem hobbi projektekhez is.

A dolgozat megírásának további pozitívuma lehet számomra, hogy nagyobb rálátásom lett az alkalmazás-tervezésre, illetve a fejlesztésre, így bízom benne, hogy ezen ismeretek birtokában nagyobb lehetőségem lesz e területen elhelyezkedni Szakdolgozatom témája egy CRM rendszer megtervezése és implementálása. A dolgozat első részében magáról a CRM rendszerről írok, majd a feladat megoldása során felhasznált technológiákat ismertetem és részletezem. Ezután a specifikáció ismertetése következik, ahol írni fogok többek között arról, hogy milyen tulajdonságokkal rendelkezik az alkalmazás. Mindezek után a tervezési folyamatról írok itt fogom részletezni az adatbázist is, valamint saját ábrákon keresztül mutatom be a programot. Végül pedig az implementációs folyamat részletezése következik, ahol csomagról csomagra haladva részletezni fogom az osztályokat és a köztük lévő kapcsolatot.

# 4. Ügyfélkapcsolat-menedzsment

## 4.1 A CRM általános jellemzői

Az ügyfélkapcsolat-menedzsment (Customer Relationship Management, CRM) tulajdonképpen új ügyfélkapcsolatok létrehozásával, ezek fenntartásával, valamint az ügyfelekről szóló információk felhalmozásával és megőrzésével foglalkozik.

Egy CRM rendszer segítségével a saját ügyfelekről egy információs adatbázist lehet létrehozni, amelybe a szervezethez tartozó minden felhasználó rögzíthet új értékesítési lehetőségeket, feladatokat. Ha az ügyfelekkel kapcsolatos minden információ egy helyen van, rendszerezetten tárolva, akkor a rendszert használók könnyebben, szervezettebben és hatékonyabban működhetnek együtt az ügyfelekkel és egymással is.


4.1 ábra
CRM rendszer összetétele
(Forrás [3])

A legtöbb CRM rendszer építőelemei:

- **Ügyfelek**: lehetnek természetes személyek vagy kapcsolattartók abban az esetben, ha az ügyfél egy szervezetet jelent.
- **Adatlapok**: ezeken kerülnek tárolásra az egyes ügyfelekhez köthető értékesítési lehetőségek, segítségükkel követhető és mérhető az üzletkötés folyamata.
- **Teendők**: az ügyfelekkel történt kapcsolattartási események (e-mail küldés, telefonhívás stb.) feljegyzései, valamint a jövőbeni feladatok, emlékeztetők, amelyek az adatlapokon kerülnek rögzítésre.

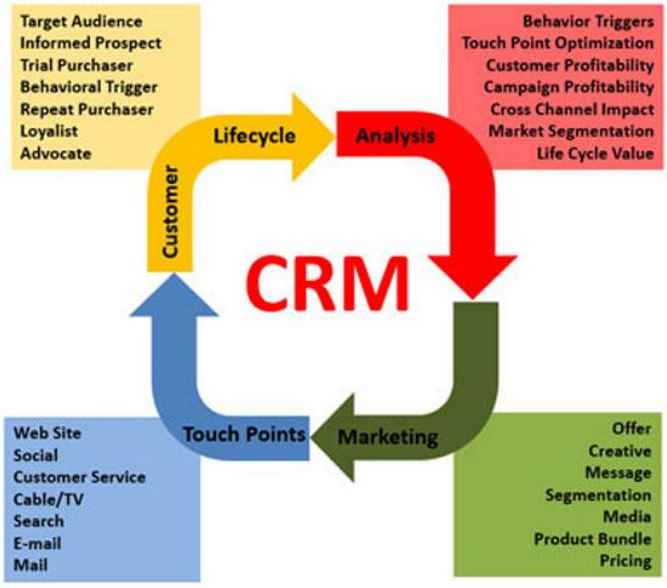
Ezekből létrejön egy adathalmaz, ami lehetővé teszi többek között egy személyre való keresést, láthatók a hozzá kapcsolódó múltbeli értékesítések és jövőbeni lehetőségek, valamint az összes vele történt interakció.

Az alábbi területekre alkalmazható ez a rendszer:

- call centerek fejlesztése, átszervezése, telepítése, létesítése
- e-business
- értékesítési módszerek
- tréningek, gyakorlati támogatás
- ügyfél és piac szegmentáció

- értékesítési hatékonyság, stratégia, csatornák
- ügyfélelégedettség mérése

### 4.2 A CRM rendszer körforgása




4.2. ábra
CRM körfolyamata
(Forrás[4])

Mint már fentebb említettem a CRM alkalmazások központjában az ügyfelekről begyűjtött adatok állnak, hiszen ezek alapján tud a vállalkozás feléjük nyitni. Ezen adatok az ügyfelek általános adataira, termék vagy szolgáltatásokra, a velük megkötött megállapodásokra és az általuk előnyösnek vélt szállítási és egyéb szolgáltatásokra vonatkozhatnak. Ezek természetesen függenek a vállalkozás típusától, így egy banknak fontos lehet, hogy az ügyfél mennyit utazik külföldre, míg a szakácsnak nem biztos, hogy ez az információ releváns lenne. Az adatokat először be kell gyűjteni a klienstől, majd ezeket analizálni kell. Amennyiben ez a folyamat sikeresen megtörtént úgy a cég már könnyen el tudja dönteni, hogy az adott ügyfélnek mire van szüksége, így könnyebben bevonzhatja az ügyfelet, és elkezdheti a kapcsolatépítést. Ezután a kapcsolatot fenn kell tartania, amihez még több adatra van szüksége az ügyfélről. Ekkor újra elkezdődik az információszerzés, melyet ismét analizálás követ és az eljárás egy körforgásként működhet tovább.

# 5. Felhasznált technológiák

A projekt elkészítésénél használt különböző komponensek és technológiák a következők:

A rendszert az Spring Tool Suite(STS)nevű nyílt forráskódú, platformfüggetlen szoftverkeretrendszer segítségével fejlesztettem, Java programozási nyelven.

A projektet az Apache Maven plugin segítségével hoztam létre különböző modulokká felosztva és a Maven segítségével épül fel az alkalmazás.

A webalkalmazást a Spring Boot keretrendszer használatával fejlesztettem.

Adattárolásra PostgreSQL relációs adatbázist használtam.

A webalkalmazást Apache Tomcat szerveren fejlesztettem, teszteltem.

A webes felület a Bootstrap front-end keretrendszerrel, Bootstrap icons betűtípus és ikonkészlettel valósítottam meg.

A projekt hosztolásához a Herokut választottam, lényegében ez egy platform szolgáltatás (PaaS) webalkalmazásokhoz.

A különböző diagramokat High-chart api segítségével jelenítettem meg.

A relációs adatok kezelésére JPA használtam.

A projektet készítése során az Git verziókövető rendszert használtam

Az alkalmazás teszteléséhez JUnit5 használtam

## 5.1 Spring és Spring Boot

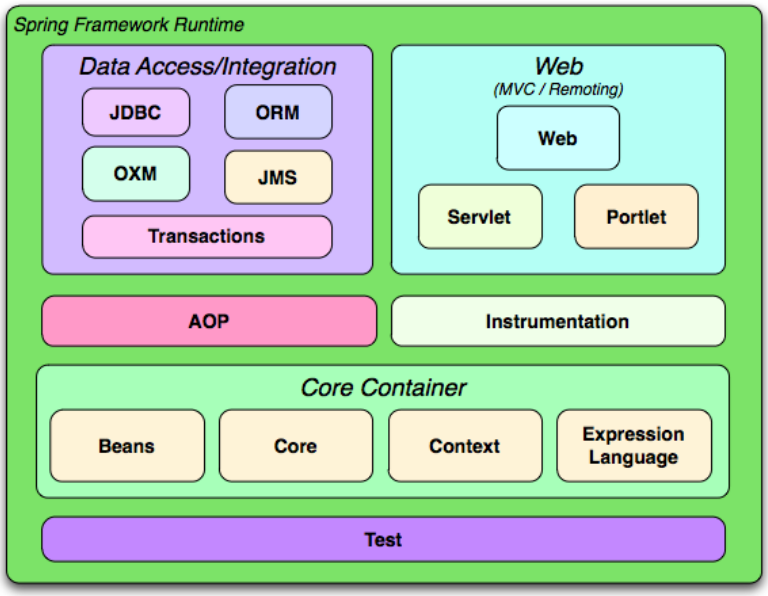
Ahhoz, hogy a teljes projektet egy nagy egészként tudjam kezelni, szükségszerű volt egy keretrendszerre használata. Ebbe a rendszerbe elhelyeztem magát az alkalmazást, így lehetőséget kaptam arra, hogy különböző szolgáltatásokat vegyek igénybe, amelyek megkönnyítették és felgyorsították számomra a fejlesztést.

Először a Spring-et mutatom be, ezután tovább haladva a Spring Bootot is, hiszen az alkalmazásomat ennek a keretrendszernek a használatával valósítottam meg.

A Spring egy nyílt forráskódú keretrendszer, amely az IOC (Inversion of Control, fordított vezérlés) tervezési mintára alapszik Java nyelven. A Spring gyakorlatilag egy IOC konténer, az IOC a Dependency Injection (függőség injektálás) támogatásával végzi el. Szoftverfejlesztés során az egyes elemek működése gyakran függ más elemek által nyújtott szolgáltatásoktól. Jellemzően tudniuk kell, hogy mely komponensekkel kommunikáljanak, hogy azok hol találhatóak, és hogy miként kommunikáljanak velük. Ha egy ilyen szolgáltatás valamilyen módon megváltozik, akkor azt valószínűleg elég sok helyen át kell írni. A szokásos módszer a kód struktúrálására az, hogy a szolgáltatások

elérésének módját az alapvető logika részeként implementáljuk. Egy másik metódus, hogy elkerüljük az előzővel járó problémákat, hogy csak függőségeket deklarálunk a szükséges szolgáltatásokhoz, és valamilyen módon egy külső kódrészlet felel majd ezek megtalálásáért és inicializálásáért. Ez engedélyezi, hogy a szolgáltatások megváltozása esetén az ezeket használó kódrészleteket ne kelljen módosítani. Ezt a típusú módszert hívjuk Dependency Injection-nek.

A Spring egyetemes megoldásokat kínál a fejlesztés során felmerülő problémákra, a gyors és eredményes alkalmazásfejlesztésre, illetve rétegelt alkalmazások készítésére. A Spring rétegelt összetételű, ami azt jelenti, hogy a különböző fajtájú feladatok elvégzésére különböző modulok vannak. Ezek a modulok hat kategóriába sorolhatók (3. ábra). Ilyen például az adatbáziskezelés, webes szolgáltatások, tesztelési megoldások, amelyek mind külön modulokban találhatók.
Ennek a modularitásnak az a jelentősége, hogy nem kell az alkalmazást teljes mértékben a Spring keretrendszertől függővé tenni, nem kell hozzákapcsolni magunkat. Szabadon




5.1. ábra
Spring modulok csoportosítása
(Forrás[4])

dönthetünk arról, hogy mely modulokat használjuk, ami konkrétan kell, illetve, ha valamelyik modul nem megfelelő a számunkra, akkor egyszerűen egy másik keretrendszer könyvtárát használjuk a Spring által kínált megoldás helyett.

Tény az, hogy a Spring megkönnyíti a fejlesztés folyamatát a modulok megalkotásával, de maga a szoftver eltérő beállításait, mint például a Maven függőségek

hozzáadását és az XML fájlok konfigurálását a fejlesztőnek kell kiviteleznie. Erre a problémára kínál megoldást a Spring Boot.

Ez a keretrendszer a Spring család egyik tagja, amely olyan plusz funkciókkal rendelkezik a Springhez képest, amelyekkel még eredményesebbé, hatékonyabbá és könnyebé teszi a fejlesztést. Az a célja, hogy a szoftverfejlesztők ne a program konfigurálásával foglalkozzanak, hanem koncentráljanak a megírandó kódra, és hogy azt az időt, amit a beállítások módosításával töltenének, azt az üzleti logikára és az implementálására fordíthassák.

### 5.2 Maven

Az Apache Maven egy csomagoló keretrendszer Java projektekhez. Egyik legfontosabb feladata, hogy automatizál folyamatokat. A Maven egyrészről leírja, hogy miként is épül fel a projekt, másrészről pedig meghatározza a projekt függőségeit(dependency) más moduloktól, vagy külső függvénykönyvtáraktól, amely csomagokat a build folyamat során automatikusan le is tölt. Mind ezen információk egy xml fájlban vannak eltárolva, melynek neve pom.xml. A külső package-ket dinamikusan tölti le egy, vagy több repository-ból, ez általában a központi Maven repository. A letöltött csomagokat a Maven helyi gyorsítótárban tárolja.

A már említett pom.xml file (pom = Project Object Model) tartalmazza az összes olyan adatot, ami elengedhetetlen egy program buildeléséhez. Ez többnyire magában foglalja a projekt nevét, tulajdonosát és a függőségek listáját. A kiterjedtebb projekteket általában szétszedik különálló modulokká, ahol minden modulhoz külön tartozik egy pom fájl. Ilyenkor annak a projektnek a pom.xml fájlja, ami magában foglalja a modulokat, lesz a gyökere az alkalmazásnak, ennek segítségével könnyedén lehet buildelni az összes modult. Az én projektemnél csak egy modult használtam.

### 5.3 PostgreSQL

A PostgreSQL egy nyílt-forráskódú relációs adatbázis-kezelő rendszer (RDBMS). A relációs adatbázis az adatokat táblákba rendezi, szerkezete: az adatok azonos összetételű sorokként kerülnek be egy táblába, ez a rekord. A felépítést, azaz a megőrzendő adatokat az oszlopok határozzák meg, ez pedig a mező. A tábla így egy egyedtípust reprezentál. Ha egy mezőt elsődleges kulcsnak jelölünk ki, akkor annak értéke egyértelműen meghatározza a rekordot, nem ismétlődhet. Ezt kiaknázva megadhatjuk ezt a mezőt egy másik táblában idegen kulcsnak jelölve, majd lekérdezéseknél egyesíthetjük a táblákat a kulcsértékek alapján. Minden relációs adatbázis-kezelő rendszer, így a PostgreSQL is az

SQL (Structured Query Language, azaz struktúrált lekérdező nyelv) nyelvet használja az adatbázis létrehozására, módosítására és az adatok lekérdezésére. A PostgreSQL biztosít grafikus felületet az adatbázisok kezelésére. Én a munkám során parancssort és a pgAdmin-t használtam, ami egy PostgreSQL adatbázisok kezelésére írt grafikus felületet biztosító eszköz.

### 5.4 JPA (Java Persistence API)

Mint ahogy fentebb is említettem a JPA a relációs adatokat kezeli. egy keretrendszer a Java programozási nyelvhez. Lényegében az adatok automatizálásának áramlása az adatbázisból a programba és vissza.

A kulcsszó itt a perzisztencia (tartós fennmaradás). Ezt olyan adatokra használjuk, mely túléli az őt létrehozó folyamatot. A Java perzisztenciát kicsit másképpen határozzuk meg, ebben az esetben arról van szó, hogy a tárolás a Java programozási nyelv segítségével történik. Java-ban az adatok perzisztálására különböző módok is léteznek, többek között ilyen a:

- szerializációság,
- JCA,
- XML,
- JDBC adatbázisok.

A fejlesztés során azért döntöttem a JPA mellet, mert nagyban megkönnyítette az implementálást azáltal, hogy automatizált bizonyos folyamatokat, illetve az annotációk segítségével egyszerűbben meg lehet mondani, hogy milyen attribútumok legyenek elérhetőek a későbbiekben.

### 5.5 GIT

A GIT [12] egy nyílt-forráskódú verziókövető rendszer (version control system, VCS). A verziókövető rendszerek kezelik a fájlokat és mappákat, valamint eltárolják a rajtuk megvalósított változtatásokat. Ez nagymértékben megkönnyíti a csapatban végzett fejlesztési munkákat, mivel nyomon követhetjük a saját, illetve a mások által végzett módosításokat, hiba esetén akármikor visszatölthetjük a rendszer egy korábbi állapotát. Bármilyen hálózaton üzemeltethetünk GIT szervert, ennek okán egymástól távol is eredményesen dolgozhatunk. Az, hogy másfajta fejlesztők egymástól távol képesek ugyanazt az adathalmazt kezelni és módosítani, jelentősen megkönnyíti az együttműködést. Mivel a munka verziókezelt, nem kell attól tartani, hogy valami súlyos

hibát, vagy minőségromlást okoz változtatásunk a rendszerben, ez esetben csupán vissza kell töltenünk egy régebbi verziót.

Némely verziókövető rendszerek egyúttal szoftver konfigurációs rendszerek is (software configuration management system, SCM). Ezeket a rendszereket arra tervezték, hogy tudják kezelni a forráskódot. Efféle például, hogy felismerik és képesek kezelni a különböző programnyelveket, valamint rendelkeznek szoftver fejlesztést segítő beépített eszközökkel is. A GIT azonban nem tartozik ezek közé, hanem egy általános verziókövető rendszer, amellyel bármilyen adathalmazt kezelhetünk. A verziókövető rendszer központja az ún. repository, ami a központilag tárolt adathalmazt jelenti. Ez legtöbbször fájlrendszer formájában tárolja az adatokat, faszerkezetbe van rendezve a fájlok és mappák hierarchiája. Akármennyi kliens kapcsolódhat a repository-hoz, olvashatja és írhatja a fájlokat. Eddig nem túl sok mindenben tér el egy fájlszervertől, amiben különbözik az az, hogy a repository nyilvántartja az összes változtatást, emlékszik a fájlok minden verziójára. A verziókezelő rendszerek másik alapvető eleme a munkapéldány (working copy), ez a verziókezelt fájlok lokális másolata az egyes kliensek gépén. A munkapéldányt a kliens először letölti a központi repository-ból, majd azon végzi el a változtatásait, fejlesztéseit, ezután hozzáadja a lokális tárolóhoz majd végül visszatölti azt a központi-ba, amennyiben nem történt olyan változás, ami ütközik egy másik kliens változtatásával. Ha az utóbbi megtörténik akkor először fel kell oldani a konfliktusokat, majd ezután válik lehetővé a feltöltés. Ennek megoldásában is segítenek az GIT eszközök. Szoftverfejlesztésnél legtöbbször van egy törzs (trunk), ami a szoftver fő verzióit tartalmazza, továbbá vannak a másmilyen ágak (branch), amelyeken fejlesztik a módosításokat, új funkciókat hoznak létre és csak utána vezetik ezeket át a fő vonalra.

### 5.6 JUnit

A JUnit egy szabad forráskódú egységteszt-keretrendszer. Könnyű a használhatósága továbbá lehetőséget biztosít arra, hogy a tesztjeinket automatikussan futtasuk a kód egészére, vagy egy részére vonatkozóan, továbbá a teszteredmények megjelenítése mellett strukturált riportok készítése is lehetséges. A rendszer a külön bevezet egy azonosítást azokra a metódusokra, amiket tesztelni kell, mindezt annotációk segítségével valósítja meg vagyis ebből kifolyólag rendelkezik egy annotációfeldolgozóval. A teszt-vezérelt fejlesztés (TDD, Test-Driven Development) lényeges eleme és az xUnit egységteszt-keretrendszer család tagja. Segítségével javíthatjuk a programjaink minőségét, amivel időt takarítunk meg.

# 6.Specifikáció

A hibajegyek elbírálási folyamata során a rendszerben a vele kapcsolatba kerülő felhasználók különböző szerepkörökben különböző feladatokat látnak el. Ezek a szerepkörök határozzák meg az alkalmazás funkcionalitását. Most az egyes szerepkörök részletezése következik:

Vezető – Ezzel a jogosultsággal rendelkező felhasználó az alap hozzáféréseken kívül eléri a kimutatásokat, különböző statisztikákat, illetve módisítani tudja a többi felhasználó jogkörét, valamint akár ki is tudja őket törölni a rendszerből.

Ügyintéző – Alapértelemzetten minden új felhasználónak ez lesz a szerepköre. Elsődleges feladatuk a hibajegyek rögzítése, továbbítása, módosítása, lezárása, törlése.

Fejlesztő – Ezzel a szerepkörrel rendelkező felhasználók csak a számukra továbbított hibajegyeket látják. Elsődleges feladatuk ezeket megoldani. A módosítás és a törlésen kívül továbbítani is tudják a hibajegyeket.

Tesztelő – Ezzel a szerepkörrel rendelkező felhasználók csak a számukra továbbított hibajegyeket látják. Egy- egy specifikus hiba megoldása a feladatuk. A módosítás és a törlésen kívül továbbítani is tudják a hibajegyeket.

Szerelő– Ezzel a szerepkörrel rendelkező felhasználók csak a számukra továbbított hibajegyeket látják. Olyan hibajegyeket kapnak, amelyt a más szerepkörrel rendelkezők nem tudnak megoldani. A módosítás és a törlésen kívül továbbítani is tudják a hibajegyeket.

| | Regisztrálás | Bejelentkezés | Új hibajegy rögzítése | Hibajegy továbbítása, módosítása, törlése | Új ügyfél hozzáadása | Ügyfél adatainak módosítása, ügyfél törlése | Statisztikák kimutatások | Fiók aktiválása | Felhasználónév, jelszó módosítás | Keresés az ügyfelek között | Keresés a hibajegyek között | Kijelentkezés |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Vezető | x | x | x | x | x | x | x | x | x | x | x | x |
| Ügyintéző | x | x | x | x | x | x | | x | x | x | x | x |
| Fejlesztő | x | x | | x | | | | x | x | | | x |
| Tesztelő | x | x | | x | | | | x | x | | | x |
| Szerelő | x | x | | x | | | | x | x | | | x |

6.1. ábra
Szerep-funkció mátrix

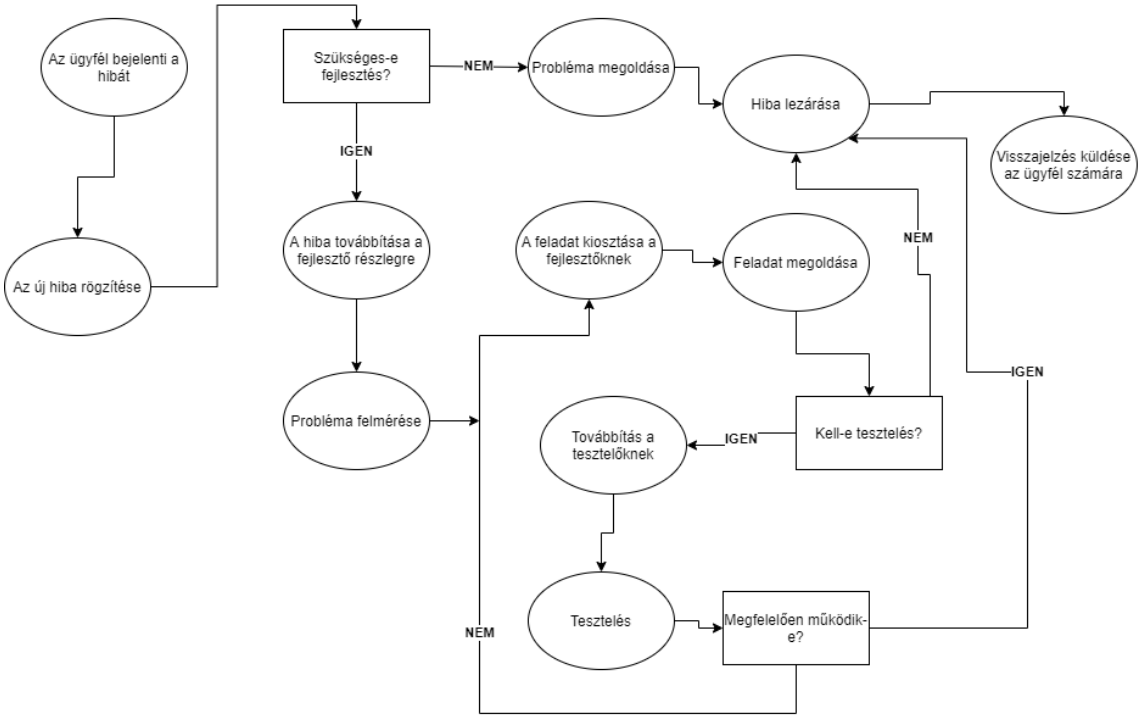
Egy hibajegy lehetséges állapotai a következők:

- Nyitott
- Zárt
- Folyamatban
- Felfüggesztett

A továbbított hibabejegyzések a következő sorrendben lesznek kezelve a továbbiakban:

- Kritikus hiba
- Gyorsfelmérés
- Extra fontos
- Fontos
- Normál

Az 6.2. ábra szemlélteti a hiba életciklusát, segítségével könnyebben megérthető, hogy a rendszer miként is kezeli a hibajegyeket.

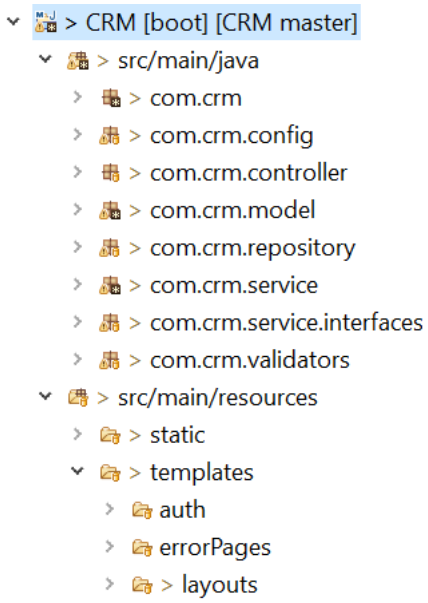



6.2. ábra
A hiba életciklusa

# 7.Tervezés

## 7.1 Modulok

A projekt Maven modulokba vannak szervezve. Az alkalmazás moduláris felosztása a következő:

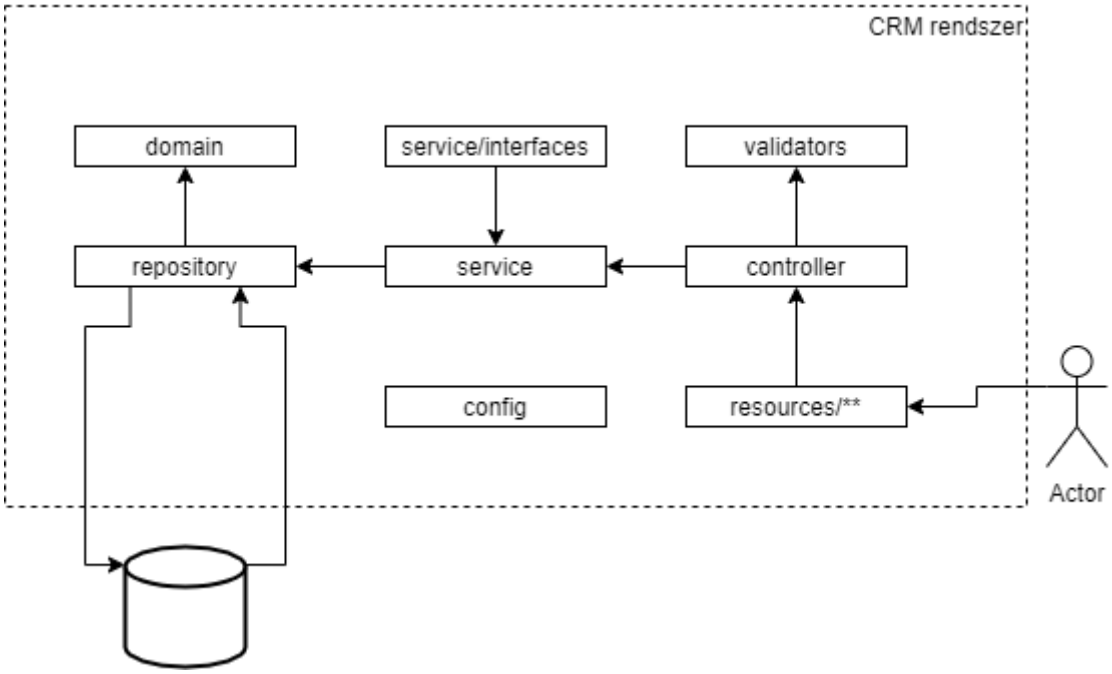



7.1. ábra
alkalmazás csomag szerkezete

- config: Ez a modul felelős a biztonságért, itt található a Spring Security. Alapvetően két osztályt tartalmaz. WebConfig nézet konfiguráció a saját bejelentkező oldal útvonalának megadása. Illetve a SecurityConf ahol a jogosultságok kezelését valamint a jelszó titkosító algoritmus megadása a feladata.
- controller: Ez a réteg felelős a webes felület (font-end) és háttérrendszer (back-end) kommunikációjához. Szintén két osztályból áll ErrorPageController itt a különböző hibaoldalakat kezelni. HomeController lényegében bizonyos end point-okra különböző nézeteket ad vissza. Pontosabban a klienstől érkező kéréseket elkapja, majd meghatározza, hogy milyen szolgáltatásokat kell végrehajtania és visszaadnia. Hozzáfér a service modulhoz is interfészeken keresztűl.
- model: Ezen modul tartalmazza a pojo-kat(plain old java object), más szóval az alap java osztályokat. A tárolandó adatszerkezeteket reprezentálják. Általánoságban elmondható, hogy ezek az osztályok rendelkeznek getter/setter

metódus párral. Ezekre azért van szükség, mert elsősorban az ORM-ek(Object Relational Modeling, pl: Hibernate, JPA) masszívan használják őket.

- repository: Azért felelős, hogy a kód képes legyen kommunikálni a enttites-kkel(pojo). Az adatbázisból objektumokat fog várni, ezek a repository-k nem osztályok, hanem interfészek és mindegyik kiterjeszti a CrudRepository-t. Ennek köszönhetően már előre meg vannak írva bizonyos metódusok, ami gyorsabb fejlesztést tesz lehetővé.
- service/interfaces: Ez a modul tartalmazza azon interfészeket, amelyek a controller felé biztosítandó szolgáltatásokat írják le.
- service: Réteg feleős az üzleti logika megvalósításáért. Pontosabban a service/interfaces package-ben megírt metódusok megvalósítása a feladata. Hozzáfér a model, repository modulokhoz, valamint itt elérhető egyéb funkciók is (pl: email küldés).
- validators: Ezen modul tartalmazza az űrlapok validálásához szükséges logikát. Ezt a modult a controller fogja használni.
- resources/templates: Ebben a rétegben találhatóak a html fájlok. Amiket a felhasználó elérhet amennyiben megfelelő jogosultsággal rendelkezik.
- resources/templates/auth: Itt található a bejelentkezési oldal.
- resources/templates/errorPages: Ebben a modulban vannak a különböző hibaoldalak.
- resources/templates/layouts: Itt találhatóak a fragmentek. Ezekre azért van szükség mert egy adott oldalt többször is felhasználunk, így csak egyszer kell megírni.
- resources/static: Ebben a modulban vannak a statikus fájlok. Mint pl: javascript, css, illetve a képek.

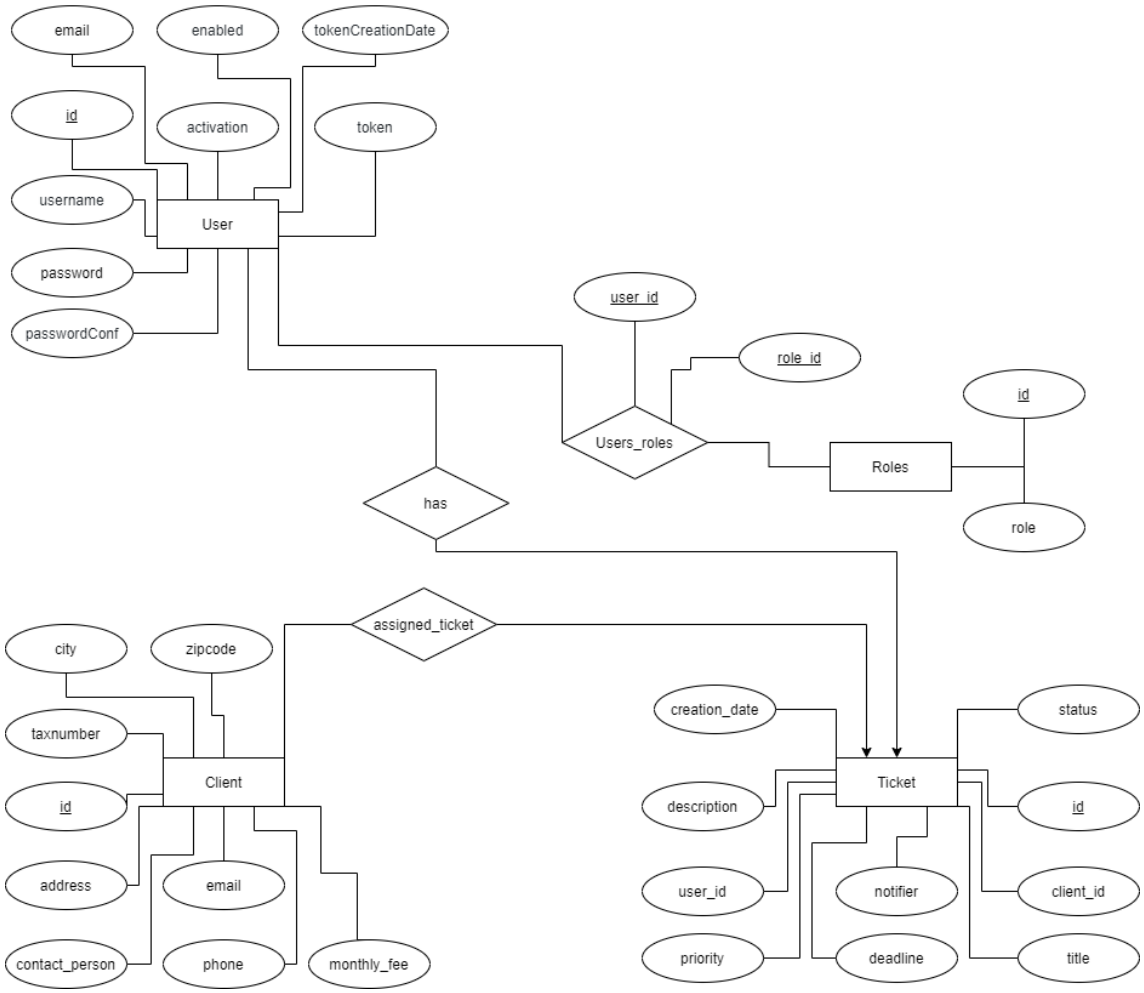



7.2. ábra
Moduláris felépítés

## 7.2 Adatbázis tervezés

A tervezés folyamán egyik legfontosabb feladat az adatbázis megtervezése. Ezt az egyed kapcsolat diagrammal szemléltetem. Ez az ábra a tárolandó adatok és köztük lévő kapcsolatok grafikus ábrázolására szolgál. Ez a modell a fejlesztés során többször is módosításra került, ezáltal jelentős átdolgozási munkákat okozva.

Az adattárolási modell végleges EK modelljét szemlélteti a 7.3. ábra




7.3. ábra
EK modell

### 7.2.1 Felhasználó

Látható, hogy az adattárolás három fő egyede a felhasználó (User) a hibajegy (Ticket) és az ügyfél (Client). A felhasználó egyed kulcsa a felhasználó azonosítója (id), ezért annak egyedinek kell lennie a rendszerben, valamint fontos megemlíteni a jelszót, amelyet titkosítva tárol el az adatbázis (password). Ezenkívül tartalmazza az alapvető adatokat, mint felhasználónév, email, aktivációs kód(activation), a profil aktív-e(enabled), token ami lehetővé teszi, hogy a felhasználó módosítja jelszavát egy bizonyos ideig, (tokenCreationDate) előbbi változónak a létrehozási ideje.

### 7.2.2 Hibajegy

A hibajegy egyednek egy egyedi azonosító lesz a kulcs tulajdonsága (id). Tárolásra kerül továbbá a státusza (status), az ügyfél azonosítója(client_id), a hibajegy címe (title), határideje (deadline), aki bejelentette a hibajegyet (notifier), prioritása

(priority), felhasználó, aki legutóbb módosította a hibajegyet (user_id), a hiba részletes leírása (description), a létrehozási dátuma (creation_date).

### 7.2.3. Ügyfél

Az ügyfél egyednek egy egyedi azonosító lesz a kulcs tulajdonsága (id). Továbbá tárolásra kerül az az összeg, amit havonta fizet (monthly_fee), telefonszáma (phone), email címe (email), az a személy neve, akivel a kapcsolatot fel lehet venni (contact_person), az ügyfél pontos címe(address), város (*city*), irányítószám(zipcode), valamint az adó azonosítója (taxnumber).

### 7.2.4. További egyedek

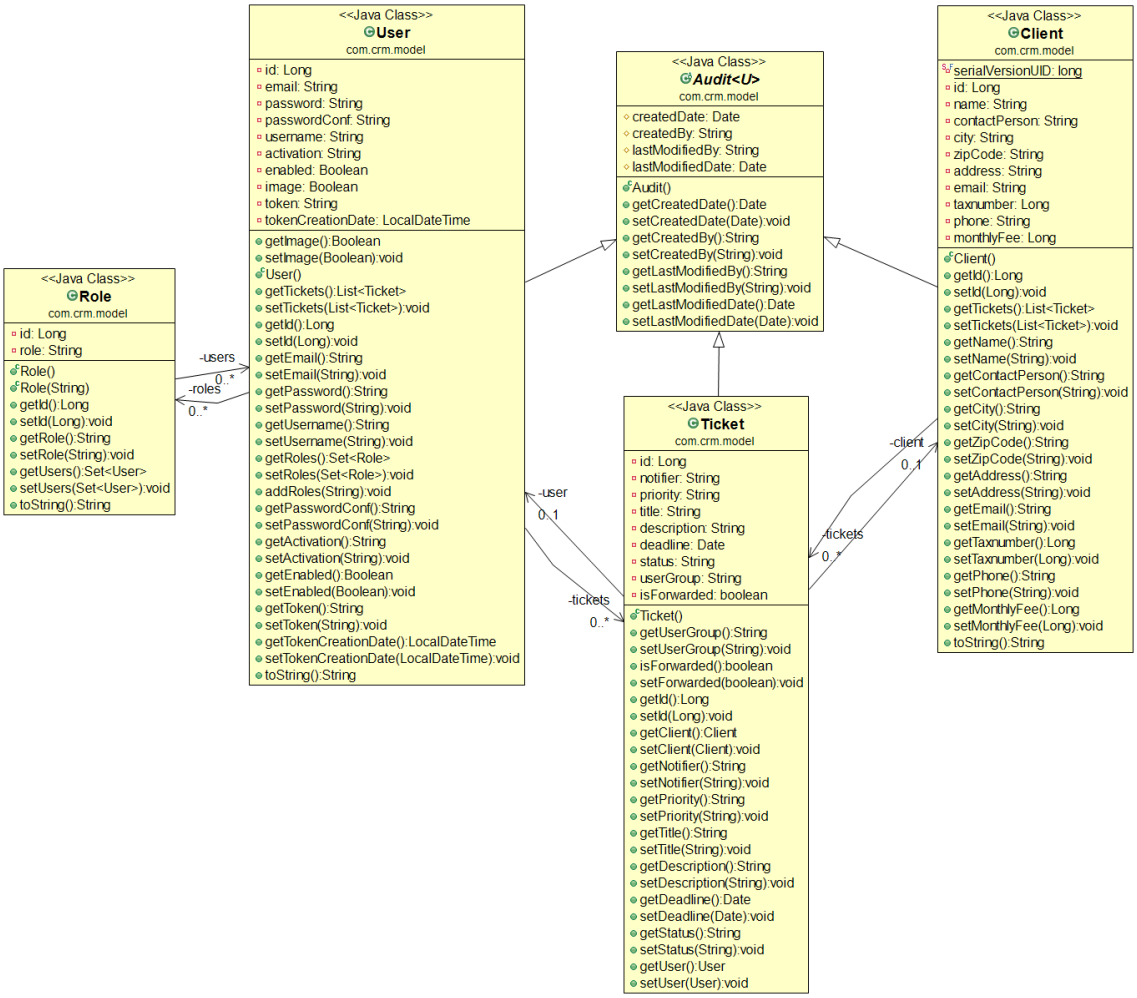
A Roles tábla tartalmazza a jogosultságokat, amivel minden felhasználó rendelkezik. A Users_roles egy összekötő tábla, ami a user és roles táblákat kapcsolja össze. Alapvetően két attribútuma van a felhasználó azonosítója és a szerepkör azonosítója. Erre az átláthatóság, illetve a könnyebb kezelés miatt van szükség.

### 7.2.5 A felhasználók és a hibajegyek közötti kapcsolat

Mivel egy felhasználó rögzíthet több hibajegyet is és egy hibajegyet egyszerre csak egy felhasználó rögzíthet így a köztük lévő kapcsolat egy a többhöz.

## 7.3 Osztályok tervezése

A 7.3. ábra szemlélteti a pojo osztályokat és a köztük lévő kapcsolatot.

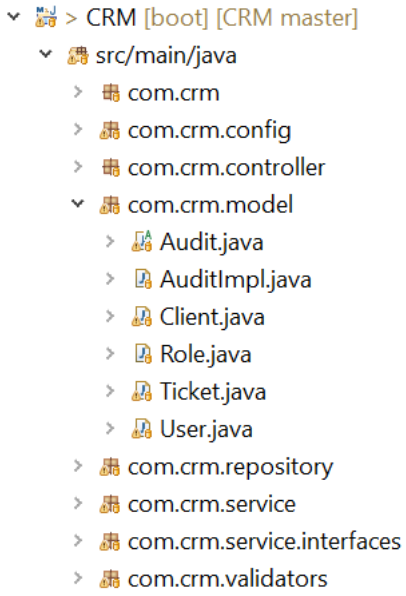



7.3. ábra
POJO osztálydiagram

# 8. Implementáció

## 8.1. Entitások

A projekt implementációját ennél a modulnál kezdtem, segítségemre volt a JPA. Az entitások úgy kerültek kialakításra, hogy ahány tábla készült az adatbázisban, annyi entitás is lett előállítva. Ezeket az osztályok a model csomagban helyezkednek el.




8.1. ábra
A model csomag

Az alábbi táblázat összefoglalja, hogy mely entitások lettek létrehozva (táblázat 8.2).

| osztálynév | leképezés |
|---|---|
| User | USERS |
| Ticket | TICKET |
| Role | ROLES |
| Client | CLIENT |
| users_roles | USERS_ROLES |
| Audit | Nincs |
| AuditImpl | Nincs |

8.2 táblázat – Létrehozott entitások

Az entitások szerkezete nagyban hasonlít egymásra, eltérések, a tulajdonságok nevei és típusai között van.

Egy entitás általános szerkezete a következő:

- Az osztálydefiníció előtt meg kell adni az @Entity annotációt, ezzel jelezve a keretrendszer számára, hogy ez egy JPA entitás. Valamint opcionális karaméterként egy @Table annotációt, amivel a tábla nevét tudjuk megadni.
- Az entitásban meg kell adni az adott tábla mezőit privát adattagokkal, rájuk pedig egy @Column annotációt rakni, amelyben meg lehet adni, hogy mely mezőre mutasson, valamint, hogy lehet-e null az értéke. Az azonosító adattagnál(ID) pedig ki kell tenni az @Id valamint a @GeneratedValue(strategy = GenerationType.IDENTITY) annotációkat ezzel jelezvén, hogy ez lesz az entitás azonosítója.
- Az adattagok felett meg lehet adni annotációk segítségével a kapcsolatok típusait is @ManyToMany(több a többhöz), @OneToOne(egy az egyhez), @ManyToOne(több az egyhez)
- Minden adattaghoz írni kell egy publikus getter, setter metódust

A fenti táblázatban látható két osztály, amelynek nincs leképezése. Ez azért van mert ezek amolyan segéd osztályok technikailag a model csomaghoz tartoznak. Az Audit tartalmaz olyan változókat, amelyeket több osztályban használunk fel (létrehozó, módosítási dátum…), így ezt a legtöbb osztály használni fogja. Tulajdonképpen itt azt történt, hogy kiszerveztem néhány gyakori változót egy osztályba. Az AuditImpl szerepe, hogy aki módosította akár az ügyfél akár a hibajegy adatait annak a neve automatikusan ellegyen tárolva. Ezekre az osztályokra azért volt szükség, mert így követni tudunk bizonyos tevékenységeket.

### 8.2 Adathozzáférési interfész (DAO)

A projektben a komponensek konfigurációját annotációkkal valósítottam meg, mivel számomra ez tűnt a legegyszerűbbnek és egyben a leghatékonyabbnak is.
Az interfészek megtervezése során az egyedek alapján választottam szét a funkciókat. Ennek értelmében négy interfészt hoztam létre a repository csomagjába, ezek: a felhasználók adatainak kezelését végző UserRepository, a hibajegyekkel kapcsolatos adatmozgatási funkciókat leíró TicketRepository, az ügyfelekkel kapcsolatos adatok kezelésére ClientRepository és a szerepköröket végző RoleRepository.

#### 8.2.1 DAO interfészek és metódusok

Először a DAO-k interfészeit nézzük meg közelebbről majd pedig az implementációit (8.3.táblázat)

| Interfész | Implementáció |
|---|---|
| ClientRepository | ClientServiceImpl |
| RoleRepository | Nincs |
| TicketRepository | TicketServiceImpl |
| UserRepository | UserServiceImpl |

8.3.táblázat - DAO interfészek és implementációk

Ami feltűnhet, hogy a RoleRepository-nak nincs implementációja ez azért van mert a UserServiceImpl-ben van megvalósítva. Valamint az interface és az implementáció között be lett iktatva egy másik interface ami növeli a projekt átláthatóságát. Az is látható, hogy egy elnevezési konvenciót alkalmaztam az a teljes projekten belül érvényesül. A névadási konvenció a következő: az osztálynevek eleje valamilyen entitásra utal majd, hogy mely réteghez tartozik, valamint, hogy interfészről vagy implementációról van-e szó. Továbbá fontos még megjegyezni, hogy nem került felírásra a CrudRepository metódusai, viszont azok is használatban vannak.
Az alábbiakban nézzük Repository-k metódusait, valamint azok faladatait.

**ClientRepository**

Ebben az interface-ben a CrudRepository által nyújtott metódusokat használtam fel.

**RoleRepository** (8.4. táblázat)

| metódus | leírás |
|---|---|
| findByRole(String) | visszaad egy szerepkört |
| findByRoleName(String) | visszaad egy Role-nak egy azonosítóját |

8.4.táblázat - RoleRepository metódusai és leírásai

**TicketRepository** (8.5.táblázat)

| metódus | leírás |
|---|---|
| findAll() | visszaadja az összes hibajegyet |
| ticketGroupedByStatus() | státusz alapján csoportosítja a hibajegyeket |
| ticketsGroupedByPriority() | prioritás alapján csoportosítja a hibajegyeket |
| ticketGroupedByMonths() | hónap alapján csoportosítja a hibajegyeket |

| | |
|---|---|
| deleteTicketByClient_ID() | azokat a hibajegyeket törli ki, akik egy bizonyos ügyfélhez tartoztak |
| updateTicketGroup() | módosítja a hibajegy továbbítási státuszát és csoportokat |

8.5. táblázat – TicketRepository metódusai és leírásai

**UserRepository** (8.6.táblázat)

| metódus | leírás |
|---|---|
| findByEmail(String) | megkeresi a paraméterül adott email címet (ha létezik) |
| allEmail() | visszaadja az összes email címet |
| findAll() | visszaadja az összes felhasználót |
| updateUserRole(Long,Long) | módosítja a felhasználó szerepkörét |
| deleteUsers_Roles(Long) | törli a felhasználó szerepkörét |
| findByActivation() | megkeresi a felhasználót aktivációs kód alapján |
| findByToken() | megkeresi a felhasználót token alapján |

8.6. táblázat – UserRepository metódusai és leírásai

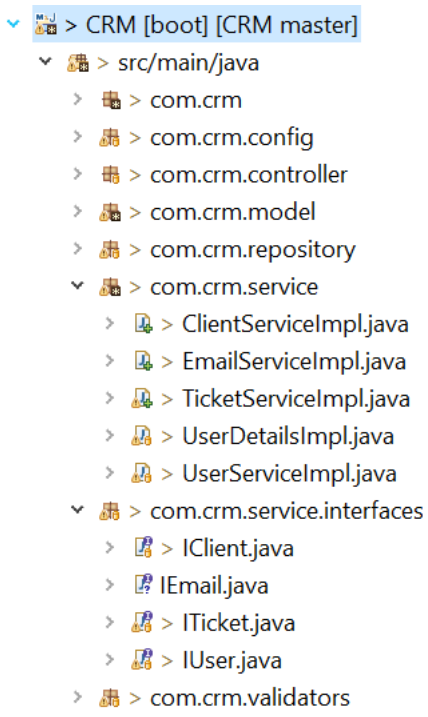
### 8.3 Service réteg

Az alkalmazást az MVC tervezési minta alapján hoztam létre, annyi kiegészítéssel, hogy a Model és a Controller réteg közé készítettem egy Service és egy Configuration réteget is.

A Service réteg elsődleges feladata az üzleti logika megvalósítása. A Service-ben definiálva van a Repository @Autowired jelzéssel, ami a dependency injection egy típusa. Pontosabban konstruktor alapú DI valósít meg. A Service osztályban az osztálydefiníciók előtt be kell szúrni a @Service annotációt.

Ennek a rétegnek az az előnye, hogy „tehermentesíti” a controller-t. Valamint átláthatóbb, jobban strukturált és biztonságosabb lesz tőle maga az alkalmazás

Minden táblának van egy külön Service részlege kivéve a Role osztályt, ezek a service csomagban találhatóak meg. Valamint a strukturáltság miatt a Service osztályokban megvalósított metódusokat kigyűjtöttem külön-külön egy interface-be. Ez megtalálható a service/interfaces csomagban.




8.7 ábra
A service csomag

Service osztályok részletesen (8.8. táblázat)

| Interface | Implementáció |
|---|---|
| IUser | UserServiceImpl |
| ITicket | TicketServiceImpl |
| IClient | ClientServiceImpl |
| IEmail | EmailServiceImpl |
| UserDetails | UserDetailsImpl |

8.8. táblázat – Service interfacek és implementációik

Akár csak, mint a Repository rétegnél, itt is bemutatásra kerül, hogy milyen metódusokat tartalmaznak az osztályok. Mivel a Repository rétegnél már bemutatásra kerültek a metódusok, nem részletezném, itt már csak azt mutatom meg, hogy melyik metódus mely korábbi metódusra hivatkozik.

A UserRepository műveleteit az IUser kezeli (8.9. táblázat)

A IUser metódusai a következők:

| metódus | DAO művelet |
|---|---|
| registerUser | userRepository.save(User) |

| | |
|---|---|
| save | userRepository.save (User) |
| findByEmail | userRepository. findByEmail(String) |
| userActivation | userRepository. userActivation(String) |
| forgotPassword | userRepository .forgotPassword(String) |
| resetPassword | userRepository.resetPassword(String,String) |
| updateUserPassword | userRepository.updateUserPassword(User,User) |
| updateUserNameAndEmail | userRepository.save (User) |
| findById | userRepository.findById(Long) |
| findAll | userRepository findAll() |
| updateUserRole | userRepository .updateUserRole(Long,Long) |
| findByRoleName | userRepository .findByRoleName(String) |
| deleteUsers_Roles | userRepository .deleteUsers_Roles(Long) |
| deleteById | userRepository.deleteById(Long) |
| addPhoto | userRepository.save (User) |

8.9. táblázat – IUser metódusai és azok metódushívásai

A TicketRepository műveleteit az ITicket kezeli (8.10. táblázat)

A ITicket metódusai a következők:

| metódus | DAO művelet |
|---|---|
| findAll | ticketRepository.findAll() |
| findbyId | ticketRepository.findById(Long) |
| saveTicket | ticketRepository.saveTicket(Ticket,User,Client) |
| updateTicket | ticketRepository.updateTicket(Ticket,Ticket) |
| updateTicketGroup | ticketRepository.save(Long,String) |
| deleteById | ticketRepository.deleteById(Long) |
| deleteTicketByClient_ID | ticketRepository.deleteTicketByClient_ID(Long) |
| ticketsGroupedByStatus() | ticketRepository.ticketsGroupedByStatus() |
| ticketsGroupedByPriority() | ticketRepository.ticketsGroupedByPriority() |
| ticketGroupedByMonths() | ticketRepository.ticketGroupedByMonths() |

8.10. táblázat – ITicket metódusai és azok metódushívásai

A ClientRepository műveleteit az IClient kezeli (8.11. táblázat)

A IClient metódusai a következők:

| metódus | DAO művelet |
|---|---|
| saveClient | clientRepository. saveClient() |
| findAll | clientRepository.findAll() |

8.11. táblázat – IClient metódusai és azok metódushívásai

A UserDetails egy speciális osztály, amelyet a Spring Security használ. Ide kapcsolódik a bejelentkezéssi funkciók kezelése, jelszó titkosítás, felhasználó akíválása.
Ezt az osztály maga a keretrendszer adja így neki nincs külön Reposyitory rétege.
Viszont a service rétegben egészítettem ki a saját implementációval.
A UserDetails műveleteit az UserDetailsImpl kezeli (8.12. táblázat)
A UserDetailsImpl metódusai a következők:

| metódus | leírás |
|---|---|
| getAuthorities() | itt lehet megadni a saját szerepköröket a security számára |
| getPassword() | a felhasználó jelszava |
| getUserEmail() | a felhasználó email címe |
| getUsername() | a felhasználó felhasználóneve |
| getName() | a felhasználó felhasználóneve |
| getId() | a felhasználó azonosítója |
| getRole() | a felhasználó szerepköre |
| isAccountNonExpired() | a felhasználó fiókja lejárt-e |
| isAccountNonLocked() | a felhasználó fiókja lezárt-e |
| isCredentialsNonExpired() | a felhasználó jelszava lejárt-e |
| isEnabled() | a felhasználó fiókja aktiválva lett-e |

8.12. táblázat – UserDetailsImpl metódusai és azok metódushívásai

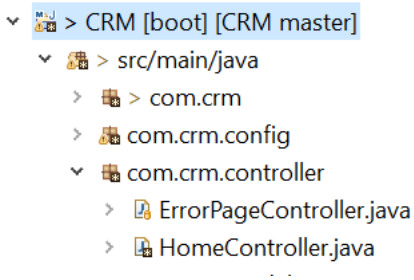
### 8.4 Controller réteg

Az adattárolási modell és a felület közötti réteg az MVC architektúrában. Alapvető feladata a bejövő kérések kezelése és a megfelelő válaszok visszaadása. Ezen osztályok felépítése, szintén hasonlítanak egymáshoz. Jellemzőik:

- @Controller annotáció használata az osztálydefiníció előtt
- Ezekbe az osztályokba be kell kötni a megfelelő Service-eket, @Autowired jelzéssel ellátni

- Valamint a service eljárásokat kezelő metódusokat a @RequestMapping jelzéssel láttam el.

A projekteben két Controller osztály található a Home és az ErrorPage utóbbi a hibaoldalakat kezeli, pl: nincs ilyen oldal vagy belső hiba történt, előbbi pedig minden mást.




8.13 ábra
A controller csomag

Az alábbi képen látható két metódus az ügyfél adatainak módosításáért felelős. Nézzük ezt meg közelebbről.

```
1
2  @RequestMapping(value = "/editClient/{id}",method = RequestMethod.GET)
3               public String editClient(@PathVariable Long id, Model model){
4
5              model.addAttribute("editClient", clientService.findById(id) );
6                       return "editClient";
7             }
7
9     @RequestMapping(value = "/editClient/{id}",method = RequestMethod.POST)
10      public String updateClient(@ModelAttribute("editClient")
11      @Validated Client editClient, BindingResult bindingResult) {
12
13    Client editedClient = clientService.findById(editClient.getId());
14
15          clientValidator.validate(editClient, bindingResult);
16                  if (bindingResult.hasErrors()) {
17                      return "editClient";
18                  }
19
20          clientService.updateClient(editedClient, editClient);
21
22           return "redirect:/listClient?clientUpdateSuccess";
23            }
24
```

8.14 ábra
A controller osztály egy részlete

Az első metódus a /editClient/{id} url-re hivatkozik. Itt az id egy dinamikus változó, mindig az, amit a user kiválaszt. Paraméterben ezt a változót eltároljuk, illetve egy model objektumot is megadunk ez az adatok továbbadásáért felelős. A törzsében megkeressük azt az ügyfelet, akit módosítani szeretnénk a korábban megadott id alapján. Ezt a model segítségével továbbadjuk a frontend számára. Ott az editClient néven tudunk majd rá hivatkozni.
A második metódus az adatok mentéséért felelős. Az url itt is ugyan az, csak a get helyett post-ot használunk. Paraméterül megkapja a korábban megkeresett ügyfelet, akit már módosítottak. Ezt egy @Validated annotációval látjuk el, ami azt jelenti, hogy itt történik az adatok validálása. Illetve található még egy BindingResult változó, ami a hibaüzeneteket fogja tárolni. Majd megkeressük a paraméterül megadott ügyfelet id alapján. Ezután az adatok validálása történik meg, erre egy saját osztályt is létrehoztam, ha hiba van maradok az oldalon és fontend-en meghívódnak a hibaüzenetek. Amennyiben minden adat megfelelő úgy mentésre kerül az ügyfél, átnavigálunk egy oldalra, ahol a user erről visszajelzést kap.

### 8.5 View réteg

Az MVC architektúrában a view réteggel kommunikál a felhasználó. A felhasználói interakció valamilyen eseményeket generál, amikre a Controller hívásokat intéz.
A projektben ez a réteg a templates csomagban van implementálva, ahol .html fájlok találhatóak. Ezek kiegészítve a Thymeleaf segítségével.
A Thymeleaf egy szerver oldali Java-s template engine (view rétegben dinamikusan renderelhetünk tartalmat) mind webes, mind standalone környezetben.
A Thymeleaf fő célja, hogy "natural template"-eket írhassunk. Ennek lényege, hogy például a HTML tartalmat a böngésző helyesen meg tudja jeleníteni még fejlesztés közben is
Többféle modulokkal rendelkezik, melyek között ott van például a Spring is, de ezen felül további modulokkal is rendelkezik, illetve saját magunk is írhatunk konkrét megvalósításokat, mely rendkívül rugalmassá teszi ezt a template engine-t. A Thymeleaf-et helyettesítheti a JSP, JSF.

### 8.6 Validators

Az adatok validálására egy sajátos megoldást alkalmaztam. A model osztályokban az annotációk helyett. Alapvetően ez nem része az MCV modellnek ezért ezt egy külön csomagban helyeztem el. Nézzük ezt meg közelebbről.

```java

@Component
public class UserDataValidator implements Validator {

        private UserServiceImpl userService;

        @Autowired
        public void setUserService(UserServiceImpl userService) {
                this.userService = userService;
        }

        @Override
        public boolean supports(Class<?> clazz) {
                return User.class.equals(clazz);
        }

        @Override
    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace
           (errors, "username", "usernameCannotBeEmpty", "Kötlező kitölteni!");
                ValidationUtils.rejectIfEmptyOrWhitespace
           (errors, "email", "emailCannotBeEmpty", "Kötlező kitölteni!");

                User user = (User) target;

      if(user.getUsername().length()<5) {
       errors.rejectValue("username","usernameMustBeLongerThan5Characters",
       "A felhasználónévnek legalább 5 karaktert kell tartalmaznia!");
                }

      for(User u:userService.findAll()) {
       if(u.getEmail().equals(user.getEmail())) {
          errors.rejectValue  ("email", "emailAlreadyExists",
                                        "Ez az email már foglalt!");
        }
       }

       if (matchEmail(user.getEmail()) == false) {
        errors.rejectValue
        ("email", "emailIsNotValid", "Nem megfelelő az email formátuma!");
        }

      }

        public Boolean matchEmail(String content) {
                Pattern p = Pattern.compile(
       "^[a-zA-Z0-9_!#$%&'*+/=?`{|}~^.-]+@[a-zA-Z0-9.-]+$",
        Pattern.DOTALL | Pattern.CASE_INSENSITIVE);
                if (p.matcher(content).find()) {return true;
                 }else {
                         return false;
                }
        }
}
```

8.15 ábra
A validators csomag egy osztálya

Ez az osztály a user objektumot validálja, pontosabban a felhasználónevet és az email-t. A profil módosításánál kerül elő. Az osztályt elláttam egy stereotype (@Component) annotációval, ami automatikusan regisztrálja a Spring bean-ek között. Az osztály kiterjeszt egy Validator nevezetű interface-t(ezt egy külső függőségkét tudjuk hozzáadni). Ekkor két metódust kell felülírni. A supports metódusban megadjuk, hogy mely enity osztályt szeretnénk validálni, a validate-ben pedig maga az adatok validálása történik. Ezeken kívül található még egy segédfüggvény, ami az email formátumát ellenőrzi. Illetve a user osztály service rétege is meghívódik. Erre azért van szükség, hogy ellenőrizni tudjuk, hogy a megadott email már foglalt-e.

# 9. Tesztelés

Az alkalmazást egységtesztek által teszteltem. Unit-teszt (egységteszt) a metódusokat teszteli. A rendszernek köszönhetően lehetőségünk van megvizsgálni, hogy egy adott metódus visszaadott értéke vagy a metódus hatása megegyezik-e az elvárttal, amennyiben igen úgy a teszt sikeres, hogy ha nem akkor sikertelen. Fontos, hogy magának a unit-tesztnek nem lehet mellékhatása a rendszere nézve. A tesztek az src/test/java/ útvonalon elérhetőek. Itt a Service réteget teszteltem. A teszteket JUnit és Mockito segítségével készítettem el. A Mockito, lehetővé teszi a mock objektumok használatát. A mock objektum egy szimulált objektum, ami lemásolja a valós objektum viselkedését.

A következőkben az egységtesztjeimet szeretném bemutatni:

- Az osztálydefiníciók előtt, egy @ExtendWith(MockitoExtension.class) annotációt használtam.
- A használni kívánt spring-es függőséget és komponenseket, például service ezeket el kell látni @InjectMocks annotációval, továbbá a benne használt spring-es függőségeket is be kell itt kötni, @Mock annotációval ellátva.
- Létrehoztam mindegyikben egy init metódust, amire @BeforeEach annotációt helyeztem, ami azt jelzi, hogy ez a metódus a teszt lefutása előtt fog lefutni. Itt lehet megadni, hogy egy bizonyos metódus hívásakor, milyen objektummal térjen vissza a meghívott metódus, majd a tesztben. Valamint itt érdemes engedélyezni a mock objektumokat.
- Az összes unit tesztben csináltam, egy olyan metódust, ami @Test annotációval van ellátva. Itt történik a tényleges tesztelés, és itt ellenőrizzük az elvárt eredményeket az Assert valamint verify utasításokkal.

Nézzünk meg egy teszt esetet közelebbről:

```
1           @Test
2           public void testfindById() {
3                    User u = new User();
4                    u.setId(1L);
5                    u.setEmail(„test@gmail.com”);
6
7                    Mockito.when(userRepository.
8                   findById(Mockito.anyLong())).
9                   thenReturn(Optional.of(u));
10
11                   User user=userServiceImpl.findById(1L);
12                   assertThat(user.getId()).isEqualTo(1L);
13          }
```

9.1 ábra
Egy teszteset

A 9.1 ábrán látható teszteset ellenőrzi, hogy meg-e tudunk találni egy felhasználót az azonosítója alapján. Ehhez szükség van több dologra is. Mivel ez a metódus a service rétegben van implementálva így az osztályba be kell húzni ezt a réteget és el kell látni a fent említett @InjectMocks annotációval, továbbá szükség van a repository rétegre is, ezt is be kell húzni és ellátni @Mock annotációval. Utóbbi annotációval a repository működését utánozzuk előbbi pedig feloldja a függőségeket. Alapvetően a teszteseteknek nincsennek visszatérési értékeik tehát mindig void típusú függvények lesznek. Először manuálisan létre kell hozni egy User objektumot, majd feltölteni teszt adatokkal. Itt nagyon fontos, hogy az azonosítóját meg kell adni különben a teszt el fog bukni. Ezután

meg kell hívni a Mockito csomagból a when() metódust. Ez gyakorlatilag annyit csinál, hogy ha meghívódik a repository findById() metódusa akkor adja vissza a megtalált objektumot, jelen esetben a korábban létrehozott User-t. Mind ezek után következik, hogy meghívjuk a service rétegnek ugyan ezt a metódusát ami visszaadja a megtalált objektumot. Végül következik az assertThat utasítás, amiben ellenőrzésre kerül a metódus. Vagyis, ha a korábban megtalált objektumnak az azonosítója nem a paraméterben megadott érték ebben az esetben 1, akkor hibát fog visszaadni.

# 10. Irodalomjegyzék


[1] https://www.soulware.hu/szakmai-reszletek/a-crm-reszletesebben/?gclid=Cj0KCQjwka_1BRCPARIsAMlUmErp3rORyrxHa-gWCO4X_QDosv66Lzhs_bacdrquNuRyX57OUhmBs4aAsMcEALw_wcB

[2] https://hu.wikipedia.org/wiki/CRM

[3] https://www.perfectviewcrm.com/what-is-crm/

[4] https://www.resonantanalytics.com/services/customer-relationship-management/

[5 ]https://spring.io/projects/spring-boot

[6] https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch01s02.html

[7] https://hu.wikipedia.org/wiki/A_f%C3%BCgg%C5%91s%C3%A9g_befecskendez%C3%A9se

[8] https://hu.wikipedia.org/wiki/Apache_Maven

[9] http://maven.apache.org/index.html

[10] https://hu.wikipedia.org/wiki/PostgreSQL

[11] https://www.postgresql.org/

[12] https://hu.wikipedia.org/wiki/Git

[13] https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

[14] https://hu.wikipedia.org/wiki/Java_Persistence_API#Java_Persistence_Query_Language

[15] https://www.thymeleaf.org/

[16] https://hu.wikipedia.org/wiki/Tesztvez%C3%A9relt_fejleszt%C3%A9s#xUnit_keretrendszerek

# 11. Nyilatkozat

Alulírott ……..…… szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet …….. Tanszékén készítettem, …….……. diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Dátum

Aláírás