**Probléma**

Több probléma is felmerült, melyre a szakdolgozat szeretne orvosolni. A fájdalom pontok melyek felmerültek:

Egy intézmény számára nehéz nyilvántartania a céges kapcsolataikat és lehetséges szakmai gyakorlati partner viszonyt vezetni és ezt karbantartani. Ahhoz tartozó szerződéseket kényelmesen kezelni.

A hallgatók számára nem transzparens milyen lehetőségeik vannak. Hova mehetnek szakmai gyakorlatra a képzésük alatt és melyik céggel van kapcsolatban az intézmény.

Cégeknek nincsen egy dedikált felület csatorna ahol a hallgatónak megmutathatják, milyen technológiákkal dolgoznak és jelen pillanatban van-e szabad szakmai gyakorlatra hely.

**Java nyelv**

Jav programozási nyelvet választottam a szakdolgozatom elkészítéshez. Java egy objektum orientáált nyelv xycz ben látta meg a napvilágot a Sun Microsystem zászlaja alatt. Most már az Oracle védjegye alatt van ez a nyelv.

Általánosságban enterprise és banki szoftverek fejlesztésénél szoktak kifejezetten használni és nagy vállalati körülmények között. Kiforrott szabványok és community áll körülötte így egy biztos és jó választás lehet, ha olyan alkalmazást akarunk fejleszteni, ha szeretnénk, hogy a technológia ne legyen elavult.

**Adatbázis**

Adatbázisnak a MongoDB NoSQL adatbázist választottam. Eze egy dokumentum orientált adatbázis, nevéből adódóan látszik, hogy nem SQL-re alapuló adatbázis. Előnye, hogy könnyen üzemeltethető felhőben, jól skálázható és szöveges keresésben beépített algoritmusa van, így kulcsszavas keresésre könnyedén lehet. Dinamikusabb adat struktúrát enged meg mint egy SQL-es adatbázis.

Mongo DB json formátumba tárolja az adatokat.

Ha SQL-el akarjuk össze hasonlítani az elnevezéseket, akkor egy rekordot egy dokumentumnak nevez, és table-t pedig collection-nek.

Hátránya, hogy a tranzakció kezelés még gyerekcipőben jár ennél az adatbázis kezelésnél, de nekem ezt a képességét nem kell használnom a rendszer elkészítésekor.

**Adatbázis kapcsolat**

Az adatbázissal való műveleteket és a kódot az adatbázissal való összekötésre, egy ORM-et használunk, a Spring Data Mongo-t. Az ORM vagy is az Object Relation Mapping egy olyan megoldás, amely egy nem objektum orientált környezetből (adatbázis) egy objektum orientált környezetbe (Java( alakítja át az adatokat. Ezáltal könnyeden tudunk velük dolgozni.

Hogy projektünk legyenenek a szükséges könyvtárak ezzel az alábbi függőséget kell hozzáadnunk a pom.xml-eünkbe: **spring-boot-starter-data-mongodb**. Ekkor a kezünkben lesznek azok az osztályok, amelyekkel az adatbázis műveleteket végre tudjuk hajtani.

A spring data mongo, meg könnyíti magát a kapcsolatot is, így akár 3 paraméter segítségével könnyedén rákapcsolódhatunk egy adatbázisra. Az alábbiakat kell beállítani az application.properties-be

spring.data.mongodb.uri=${MONGO\_URI:mongodb://localhost:27017/pas}

spring.data.mongodb.username=${MONGO\_USERNAME:admin}

spring.data.mongodb.password=${MONGO\_PASSWORD:123}

Annyival okosítottuk fel a rendszert, hogy adunk lehetőséget arra, hogy kívülről is jöhessenek az adatbázis kapcsolathoz szükséges paraméterek. Ezek a rendszer környezeti változói beállításával állíthatjuk.

A @Document annotáció segítségével tudjuk össze kapcsolni az osztályainkat az adatbázis collection-jében található documentumokkal. Így minden egyes Java objektumunk, egy az adatbázisban található sornak a reprezentációja.

String formátumú azonosítót generál az adatbázis.

**MongoTemplate**

Ez az osztály az amellyel tudunk könnyedén adatbázis műveleteket végre hajtani. Mi nekünk csak a save metódust kell meghívnunk és ekkor a Spring ezt átalakítja natív mongo adatbázis műveletté és vére hajtja azt.

Dependency injection segítségével tudunk elkérni egy objektum példányt a contextustól, amelyet az application.propertiesben páramét ereztünk fel, tehát ezzel az adatbázissal fogja végra hajtani a műveleteket.

**Criteria builder**

Vannak esetek amikor, amikor nem az összes adatra van szükségünk, hanem bonyolultabb lekérdezéseket kell megtennünk. Két lehetőségünk van, vagy nativ mongo query-t írunk vagy CriteriaBuilderrel javakódban tesszük össze a lekérdezést.

Szakdolgozati projekt alatt, a CriteriaBuilderes megoldást választottam, mert olvashatóbb lesz így a kód, é nem String-ekként kell queryket írni, amelyekben nagy a szintaktikai vétésnek a lehetősége.

**Maven**

Azért hogy könnyebben lehessen fejleszteni és bővíteni az alkalmazásunk könyvtárait függőségeit, built tool-okat szokták használni. Én a szakdolgozathoz a Maven-t választottam, mert az iparban ez a legtöbbet használt eszköz webes alapú java alkalmazások fejlesztésénél.

A Mavennel felkonfigurálni egy projektet úgy tudunk, hogy a minden projekthez tartozik egy leíró fájl. Ez a leíró fájl XML formátumú, és pom.xml a neve. Ebbe az xml-ben lehet leírni a projekt nevét, verzió számát és ennek a függőségeit is.

**JSF és PrimeFaces**

JSF vagyis a JavaServer Faces egyszabványos MVC alapú, Java felhasználó felületet generáló keretrendszer. Ennek a segítségével semmi, vagy kevés Javascript tudásra van szükség, UI-t készítsünk. Itt a szerveroldalon renderelődik ki maga a html és a szükséges javascriptek, miközben nekünk csak Java kódban kell fejleszteni. Komponens alapú megoldást követ, tehát a felületi elemeket újra lehet használni, és akár saját komponenseket is lehet készíteni.

Előnye, hogy csak egy programozási nyelvet kell ismerni és természetében adódóan biztonságos, mert ui kigenerálása a szerver oldalon történik. Ellenben nehéz állapot mentesre megírni egy alkalmazást vele, ellenben ez csak kimagaslóan magas felhasználó számnál lehet probléma. Általánosságban enterprise alkalmazásokat szoktak vele készíteni.

Ennek a keretrendszere épít a PrimeFaces keretrendszer, amely még plusz 100 kész komponenst szolgáltat, amelyeket a fejlesztőnek csak használni kell. Ezek a komponensek könnyedén skinezhetőek css-el akár, és könnyedén lehet velük mobile first alkalmazást is készíteni.

A **primefaces-spring-boot-starter** függőségre van nekünk itt szükségünk.

JSF xhtml el fájlok. Ezek olyan html fájlok amelyeknek kötött strukturája van. Ezeknek a fájloknak meglehet adni .xhtml vagy jsf kiterjesztést. Az xhtml a használtabb és beállítjuk az application.properties-be

jsf.default-suffix=.xhtml

jsf.facelets-suffix=.xhtml

**viewk**

Az xhtml fájlokat, a JSF a META-INF en belüli resources mappába keresi. Itt kell tárolni a szükséges view-knak a xhtml megvalósításaikat.

Az xhtml-en belül tárolom az xhtml fájlokat. Ezeket is külön részekre bontom. Az **auth** mappában találhatóak az authentikációs felületek. Az **app** mappa alatt találhatóak a tényleges applikáció felületei.

A core mappába helyeztem el a template-t és a közösen használt xhtml-eket.

**Validáció**

Az adatainkat mindekképp validálni akarjuk, vagyis ellenőrizni, hogy üzletileg megfelelőek legyenek, hogy a rendszer konzisztensen tudjon működni.

Az adatokat több rétegűen szoktuk validálni, magán a felületen, és olykor mielőtt elmentenénk az adatbázisba, és akár esetleg szolgáltatások metódus meghívása előtt.

A JSR-303 szabványnak megfelelő validációs megoldást fogjuk használni, amelyeket annotációk segítségével fogjuk felkonfigurálni, melyik beviteli mezőnek mi lehet az értéke. Hogy könnyedén tudjuk használni, a spring alapú projektünkben , a **spring-boot-starter-validation** függőségre van szükségünk

Springnek van két aját osztálya a ValidationMongoEventListener, amely figyeli a validációs eseményeket adatbázisba mentéskor. Ezt a saját magunknak kell kézzel hozzáadni.

**Kódrészlet**

**Lombok**

A fejlesztéskor sokszor kell ismétlődő kódsorokat írni. Ilyenek például a getterek setterek és a konstruktorok. Hogy ezeket egy-egy módosulásra ne kelljen mindig újra írni, ezért automatikusan generáltathatjuk ezeket a Lombok segítségével. Lombok egy olyan technológia megoldás, ahol annotációk segítségével megjelölheted, hogy mit szeretnél compile fázisban készíttetni, így ő a generált javabájtkódba legenerálja neked a szükséges kódrészleteket

**Kódrészlet**

**PrettyFilter**

Mivel a JSF xhtml kiterjesztési fájlokat használ, és alapvetően ezeknek a fájloknak az elérhetőségét kellene az URL-be beleírni, hogy láthassuk a szükséges oldalt, érdemes egy URL rewriter technológiát használni.

Ennek a segítségével beállíthatjuk, hogy milyen URL-re milyen fájlt jelenítsen nekünk, és akár paramétereket és tudunk adni egy-egy url-nek. A szükséges függőség a **prettyfaces-jsf2** melynek a 3.3.3-mas verzióját használom.

**Kódrészlet**

Hogy ezt integrálni a kódunkkal kell készítenünk egy filtert, amely minden http request-nél meg fut, és megnézi, hogy az aktuálsi url-re melyik html-fájlt kell a böngészőhöz legyártania és küldenie.

AppPrettyFilter-ben implementáljuk saját PrettyFilter osztályunkat. Erre azért van szükség, mert javakódban szándékozom konfigurálni a, hogy milyen url-ek vannak a rendszerben, és ezt inbox nincs lehetőség, ezért saját magunknak kell ezt megvalósítani.

A filter beregisztrálását a PrettyFacesConfig konfigurációs osztályra van szükség, ahol felvesszük a SpringSecurity filterei közé. Magát a requestek filterezését és ellenőrzését a SpringSecrutity hajtja végre.

**Design pattern**

A JSF egy MVC alapú keretrendszer, alapvetően itt csak a Modell és a View részt kell megírnunk, mert egy nagy kontroller kezeli a bejövő kéréseket a böngészőtől.

Service a szolgáltatások a felületet menedzselő JSF beanek és a repositoryk között, melyeknél plusz validációt végzek el. Az hogy egy osztály szolgáltatás legyen @Service annotációval kell ellátni. Hogy a validációt bekapcsoljuk metódus szinten a @Validated annotációval tudjuk megtenni. Ekkor a spring kontextusa ellenőrzi, hogy a bemenő paraméterek megfelelnek-e validációs szinten a metódusnak, és ha hibát talál a ConstraintExceptiont hibát dobna a rendszer.

Repository design patternt követ maga a Spring, erre van felkészítve a keretrendszer így én is ezt használom. A repositoryk kommunikálnak az adatbázissal, és itt tesszük össze a szükséges lekérdezéseket, mentünk vagy módosítunk adatokat az adatbázisban. Hogy egy repository legyen valami a @Repository annotációt kell rátenni a szükséges osztályra.

Generikus megoldásokkal, a CRUD (Create Read Update Delete) műveleteket könnyedén ki tudjuk vezetni ős osztályokba, és így csak egyszer kell megírni a szükséges metódusokat. Így növelni tudjuk a kód újra használhatóságát és bővíthetőségét.

**Dependecy injection**

A depenedency injection segítségével a webes java alkalmazás fejlesztésekor, automatikusan tudunk létrehozni példányokat a szükséges osztályokból. Ezeket a spring kontextusa menedzseli számunkra. Például a @Service és @Repository annotációval ellátott osztályokat könnyedén beköthetjük automatikusan az @Autowired annotáció segítéségével.

Intefaceket szokás használni, mert így az esetleges implementációkat könnyebb cserélni az alkalmazásban. Ez fajta megoldás elősegíti azt, ha SQL-es adatbázisra kellene átírnunk az alkalmazást, akkor az implementációkat átírnánk ennek megfelelően, hogy SQL adatbázisra kapcsolódnák és hozzá tartozó ORM-et használnák, akkor ugyan olyan könnyedén tudna működni az alkalmazás.

Arra figyelnünk kell azonban, hogy több implementációt nem kezel automatikusan, erre felkell készülnünk, ha ilyet szeretnénk, megvalósítani.

Használom még a @Configuration annotációt, az ilyen osztályokat alkalmazás elindulásakor létrehozza és a szükséges dolgokat megfuttatja a Spring, így ezzel feltudjuk konfigurálni a szükséges dolgokat.

**Mongo szöveges indexelés**

Mongo db-nek van egy szöveges indexelés, melyet Text indexnek nevez. Ezeknek meg lehet mondani, hogy valamilyen kulcsszót megtalál egy mezőben, akkor az hány pontot érjen. Erre könnyedén fel lehet építeni egy kereső algoritmust.

Ezt könnyedén feltudjuk konfigurálni egy-egy osztályunkon. A szükséges mezőkre rá kell tennünk a @TextIndexed annotációt és a weight

**DBref**

Sokszor akarunk olyan működést elérni, hogy egy másik dokumentumot használunk mint mező típus, de annak csak a referenciáját akarjuk bele írni a dokumentumba, nem az egész felhasználót. Ezt könnyedén megtudjuk tenni a @DbRef annotáció segítségével. A lazy paraméterrel eltudjuk érni, hogy csak akkor töltse be az egész dokumentumot az adabázisból, ha hivatkozunk rá és használni akarjuk valamilyen adatát.

**Security**

Manapság már minden rendszerben vannak felhasználók és nekik az adataikat védelme érdekében különböző biztonsági intézkedéseket kell tenni.

A Spring ökoszitsztémának van egy saját security megvalósítása, mellyel könnyedén lehet authentikációs folyamatokat végre hajtani, audit információkat eltárolni, és a szerepköröket is könnyedén lehet vele menedzselni.

Hogy tudjuk használni ezt a megvalósítást, a **spring-boot-starter-security** dependenciát kell használnunk

**UserDetails**

A Spring, hogy tudja használni ami osztály megoldásunkat, implementálnunk kell UserDetails osztály, melyben a szükséges metódusokat meg kell valósítanunk. Azért van szükség erre, mert maga az authentikációs flow ezt az interface-t használja.

Számomra getUsername a getPassword és a getAuthorities metódus a fontos, ezeket valósítjuk meg. A többi implementálandó jelen pillanatban nem értelmezhető nálunk, mert nincs szükség admin elfogadásra, és nem járhat le a jelszava egy felhasznlónak.

**Authentikáció**

A Spring authentikációs motorját használom a szakdolgozatomban, ezért a implementáltam a UserDetailsService osztályt.

A spring authorizációs motorja a loadUserByUsername metódust hívja meg, és az alapján szerzi meg a szükséges felhasználót username alapján, amely az én esetemben az emailcím. Ezután hasonlítja össze a jelszavat, az eltárolt titkosított jelszóval.

Amikor a felhasználó belép a rendszerben az aktuális sessionbe eltárolodik, ezt a JSESSIONID sütibe tárolja el a böngészőben.

**Audit**

CustomAuditorAware osztály az AuditorAware osztály implementálásával be tudjuk állítani, hogy eltudjuk tárolni ki hozott létre egy dokumentumot, és ki módosította utoljára.

Ekkor csak a @CreatedBy vagy a @LastModifiedBy annotációt kell rátenni a kívánt mezőre, melynek természetesen felhasználó típusúnak kell lennie.

Ezek mellett eltudjuk tárolni a @CreatedDate és a @LastModifiedDate annotáció segítségével a létrehozás és utolsó módosítás dátumát is.

Hogy működjön Mongo adatbázissal az auditáció, a main metódust tartalmazó osztályunkra szükséges a @EnableMongoAuditin annotációt rá tenni, és referenciaként megadni egy bean-nevet, melyet implementálni kell.

Ezt az AuditorConfig osztályban valósítottam meg.

**SecurityContextHolder**

Az aktuális felhasználót tudjuk elkérni ettől az osztálytól. Ha nincsen authentication objektuma, akkor nem történt meg még az authentikáció.

Tehát ezt az osztályt lehet használni, ha egy sessionon belül van felhasználó, akkor ezen keresztül elérhetjük az adatait.

**Security felkonfigurálása**

Ahozz hogy letilthassunk url-eket, hogy csak bizonyos jogosultsággal lehessen látni, vagy esetleg már csak bejelentkezett felhasználoknak, a WebSecurityConfigurerAdapter osztályból kell leszármaznunk, és ezzel egy @Configuration annotácóval kell ellátni az osztályunkat.

A configure metódust kell felülírnink A Configurációt egy builder segítségével tudjuk megtenni, ahol megtudjuk mondani mely urlt engedünk meg bejelentkezés nélkül használni.

Az antMatchers("/").authenticated() –el azt mondjuk meg, hogy authentikált felhasználó láthatja csak ezt az oldalt. Az admin oldalakat csak adminisztrátoroknak szabad látniuk, ezt a megkötést az alábbi sorral tesszük meg: antMatchers("/admin/\*\*").hasRole("ADMIN")

**Statikus resourcok**

Statikus resourcoket be kell állítanunk, hogy engedje betölteni nekünk a Spring securitynek meg kell mondani ezeket ne akarja letiltani feleslegesen.

**Templatek**

A core mappába készítettem el a fő xhtml tempaltet, amelyet minden xhtml oldal definiálni fog, ekkor könnyedén lehet egységes kinézetet varázsolni az oldalaknak, és megtudjuk kimélni magunkat a kódismétléstől.

A template ben definiálni kell ui:insert tageket majd a valódi oldalaknak implementálnia azokat. A ui:composition tagnél a template paraméterrel adhatjuk mely, hogy melyik xhtml-et akarjuk template ként használni

**CSRF**

Cross Site Request Forgin támadások ellen CSRF token használunk, mellyel megtudjuk védeni az alkalmazásunkat, hogy más weboldalak ne tudják ellopni a kérésünket. Minden egyes requestnél szükségünk van a CSRF tokent elküldeni, hogy a szerver betudja azonosítani, hogy jóhelyre küldi az adatokat.

Szerencsére a Spring securityvel könnyen tudjuk ezt menedzselni. Hogy elküldjük minden egyes requestben a megfelelő CSRF tokent, ahhoz minden egyes JSF form-ba bele kell tenni, egy hidden inputot amelyben tárolni fogjuk a csrf tokent.

Szükséges osztály amelyre szükségünk

**BCrypt**

Jelszavakat BCryptenrycptáljuk. Jelen pillanatban ez az elfogadott a springben az SHA256 már depreceated megoldásnak jelöli a spring.

**@Named**

A @Named annnotáció segítségével tudjuk megmondani a JSF-nek, hogy ez egy nevesített osztály. Itt a webserver motorja szükség esetén létrehozza nekünk a példányokat.

Egyedinek kell lenni a nevének.

**@ViewScoped**

A ViewScoped annotáció segítésével megtudjuk mondani, hogy a képernyőt menedzselő backend bean addig éljen, amíg a felhasználónál a képernyő aktív.

**Anonym user**

Amikor még nem lépünkbe a rendszerbe anonym felhasználóként kezel minket a spring, és ekkor egy String értékként eltárolja az anonymousUser értéket. Tehát ha a SecurityContextHolder-ben, ez az érték akkor nincsen még bejelentkezve a felhasználó.

A LoggedInUserBean session scopu, tehát addig él amíg a Htpp Session él. Ha azt szeretnénk, hogy egy managedbeanónk, session scopu legyen, a @SessionScoped annotációt kell használni. Arra figylni kell, egy ilyen bean nek implementálni kell a Serializable interfacet.

**Nyelvesítés**

Resource bundlek segítségével kulcs és érték pár alapján tudjuk nyelvesíteni a felületet. Ezt egyszerűen megtudjuk csinálni, két különböző property fált kell létrehozni, ugyan azzal a prefix-el és post fiként kell megadni a nyelvet. Esetemben most az en és a hu-t.

A loadbundle tag segítségével be tudjuk tölteni ezeket a property fájlokat az oldalainkra. Ezután a kulcs segítségével a JSF a megfelelő értéket rendereli ki a képernyőre, és azt választja amilyen nyelvet van beállítva az aktuális sessionhöz.

**Metódus szintű jogosultság kezelés**

Metúdosknál és egész bean-eknél eltudjuk dönteni, hogy kihasználhassa azokat, ezzel letudjuk védeni, hogy például egy nem admin szintű felhasználó véletlenül se tudjon új szerződést felvenni a rendszerben. Erre a @PreAuthorize metódust kell használnunk, és ezen belül megtudjuk mondani minek kell teljesülni. Megtudjuk nézni, hogy authentikált-e a felhasználó vagy anonym, vagy hogy milyen szerepkörrel rendelkezik. Erre a hasRole metódust kell használnunk az annotáció értékének, és a szerepkörnek a ROLE\_ prefix nélküli nevét.

Szöveges konstansok

Szöveges konstansokat, a CoreConstants van gyüjtőm. Ez egy final osztály privat konstuktorral. Ez azért van így, hogy ezt az osztályt ne lehessen inicializálni, benne csak a konstnasokat tárolom.

Konstansokat ildomos egy helyen gyűjteni, mert Stringeket beleírni a kódba nem szép megoldás és nagy a hiba lehetőség is ha váltotani kell a kódot.

**Mycompanies**

A Cégeim oldalon képes menedzselni a cégeit. Újat felvenni, módosítani, keresni és törölni ezeket.

Ezen a képernyőn az adatok nem Lazy megoldással jönnek a táblázatba, mivel feltehetőleg egy felhasználónak nem lesz 1000 céges profilja, így ez nem fogja terhelni a rendszert.

**Jövőbeni lehetőségek**

A rendszert jövőben érdemes lehetne összekötni az aktuális tanulmányi rendszerrel, PTE esetében ez a Neptun. Cégeknek is lehet API-t biztosítani, hogy rákapcsolódhassanak a rendszerre, és egyéb szolgáltatásokat nyújtani számukra, mint a hallgatók számának lekérdezése.

A rendszeren belül lehetne a hallgatóknak a gyakorlati helyre való jelentkezést lebonyolítani, és akár szükséges dokumentumokat 1-1 jelentkezéshez automatikusan kitölthetné rendszer.

Különböző rétegeket jobban szét lehetne választani, és modularizálni őket, ezáltal az elkészített kódokat újra lehetne használni.