Haladási napló – Diploma tervezés 2

Október 3.

Sajnos az Xcode-dal problémák adódtak, ezért előbb azt kellett újra telepíteni, de mivel ez sem oldotta meg a problémát, ezért újra telepítettem a gépet a macOS Sonoma frissítés után, úgy tűnik, ez jelenleg orvosolta a problémát.

Emiatt a képernyőkkel nem igazán tudtam haladni ezen a héten, viszont elkezdtem beleásni magam abba, hogy milyen lehetőségek vannak chat alkalmazások/funkciók megvalósítására, és kettő relevánsabb videót is találtam:

<https://www.youtube.com/watch?v=6v4fmg9iRSU>: régebbi videó (2020-as), storyboardos alkalamzást csinál benne, de bemutatja a MessageKit library alapjait, ami biztosítja az alapokat egy chat funkciót megvalósító alkalmazáshoz, viszont a csatorna sok érdekességet bemutat Swift fejelsztésben, pl. push notification használattal kapcsolatban is van videó

<https://www.youtube.com/watch?v=Zz9XQy8PRpQ>: friss videó, SwiftUI-ban építenek chat alkalmazást Firebase backendhez – a saját alkalmazásom szempontjából ez a relevánsabb találat, de ezen a csatornán is sok érdekes dolog van, leginkább az alkalmazás esztétikája szempontjából, illetve design tutorialok is vannak (Figma, Sketch)

Október 10.

Implementálásra került a chatnézet alapja. Ez az alap módosításra, illetve bővítésre kerül majd az üzleti logika implementálása során. A chat nézet alapja a következő komponensekből tevődik össze:

* létrehozásra került egy „ChatTitleRow” nevű osztály, amely a chat nézetben a beszélgetés headerjéül fog szolgálni. Ez az komponens megjeleníti a beszélgetésben részt vevő másik fél profilképét (jelenleg system image-ek vannak beégetve az adott helyeken), nevét, és amennyiben rendelkezik felhasználónévvel, akkor abban az esetben azt is. A profilkép fix mérettel, lekerekítve jelenik meg (frame; width: 50, height: 50) (cornerRadius: 50)
* létrehozásra került az üzenetbuborék megjelenítéséért felelős komponens („MessageBubble”), amely implementálásához szükség volt egy Model osztály létrehozására („ChatMessage”), ebben az üzenetek a következő property-kkel rendelkeznek: id (azért, hogy ForEach függvénnyel bejárható legyen az ilyen elemeket tartalmazó tömb), az üzenet, egy „received” boolean property, amely azt hivatott jelezni, hogy az adott felhasználó kapta-e az üzenetet (true esetben igen, false esetben nem) és egy „timestamp” mező. A „MessageBubble” példány eltárol egy ChatMessage példányt, majd attól függően rendezi bal vagy jobb oldalra, illetve színezi szürkére vagy narancssárgára a buborékot, hogy az adott user kapta, illetve küldte az üzenetet. Az üzenetbuborékokat 25-ös cornerRadius értékkel láttam el. A buborékra való tap gesztúrát követően a buborék alatt megjelenik az üzenet timestamp-je.
* A chat nézethez létrehoztam egy külön TextField komponenst is. Ehhez első körben létrehoztam egy „CustomTextField” osztályt. (Ennek az oka az volt, hogy ha a későbbiekben szükség volna még egyedi TextField osztályokra, akkor ezt a komponenst fel lehessen használni.) Ebben az osztályban eltárolásra kerül property-ként a placeholder szöveg, illetve @Binding változóként a begépelt szöveg (azért @Binding változó, hogy külső komponensből is módosítható legyen a tartalma), továbbá eltárolásra került két függvény típusú változó, az egyik az onEditingChanged esemény handlerje (boolean -> void típussal), a másik az onCommit esemény handlerje (void -> void típussal). Az osztály body property-jében elhelyeztem egy ZStack elemet, amelyben üres szöveg string esetén megjelenítésre kerül a placeholder, amelyre egy TextField került, amelybe a user beviheti a szöveget.
* A chatben használt bevitel kezelésére szolgáló komponensben felhasználtam az előző pontban leírt CustomTextField osztályt, ahol egyelőre csak a szöveges paraméterek lettek megadva a konstruktorban (üzleti logika hiányában még nem adtam értékül event handler függvényeket). A TextField széleit lekerekítettem és adtam neki egy bordert is, amely megjelenítéséhez egy workaround megoldást alkalmaztam: overlay modifiert alkalmaztam, amelyben értékül adtam egy RoundedRectangle elemmel. Elhelyezésre került egy gomb is, amely jelen pillanatban funkciót még nem lát el.
* Végül az elkészült komponenseket elhelyeztem a „ChatConversationView” komponensben, amely a teljes, beszélgetéshez tartozó képernyőt leírja. Jelen pillanatban dummy adatok szolgálnak a partner adataiként és az üzenetek adataiként is. A képernyő kialakításához létrehoztam a megfelelő VStack struktúrát, amelyekben elhelyeztem a korábban leírt elemeket. Az üzenetbuborékokat egy ScrollView-ban helyeztem el, így azok görgethetővé váltak.

Október 17.

Carousel view-hoz tutorial: <https://www.youtube.com/watch?v=6gAqdamj6tg>

Az adott szálláshoz tartozó képernyő implementálása következett, amelyre akkor jut el a felhasználó, ha a szállás listából kiválasztja az adott szállást. Ennek a képernyőnek az egyik fontos és talán legösszetettebb eleme a carousel, amelyben megjelenítésre kerülnek az adott szálláshoz feltöltött képek.

A carousel elemnek létrehoztam egy külön View komponenst, amelynek a CarouselView nevet adtam. Ez egy generikus komponens, amely rendelkezik egy View típusú „Contenttel”, valamint egy „Itemmel” és egy „ID-val”. Az „Item” megvalósítja a RandomAccessCollection protoclt, az ID pedig a Hashable és Equatable protocollokat. Property-ként a következő adatok vannak eltárolva:

* „content”, amely egy olyan @escaping closure, ami az „Item.Element” és egy CGSize paraméter alapján ad vissza „Contentet”
* „id” „KeyPath<Item.Element, ID>” típussal
* illetve a kártyákhoz tartozó property-k mint „spacing” és „cardPadding” „CGFloat” típussal (default értékek 30 és 80)
* items „Item” típussal
* illetve egy @Binding Int típusú „index”, amely az aktuálisan kiválaszott Item.Element (kártya) indexe.

Ezeket az értékeket a konstruktoron keresztül lehet megadni az carousel objektumnak.

Következő lépésként felvettem az „AccommDetailsView” objektumban a carousel-hez szükséges property-ket (egyelőre dummy elemekkel, ezek később majd a backend válaszból lesznek kinyerve): egy @State index property-t, amely az aktuálisan kiválasztott kártya indexét tartalmazza, illetve egy kép típust tartalmazó tömböt. A konstruktorban megadtam az indexet, adtam egy 100-as paddinget a kártyáknak, az items-nek megadtam a képeket tartalmazó tömböt, illetve megadtam az id-t szintén a tömbből. A content closure-nek pedig egy olyan closure-t adtam meg, amely egy „Image” objektummal tér vissza, amely megjeleníti az Items tömbből az aktuális indexű elemet. Ennek az Image elemnek adtam modifier-eket:

* resizeable
* aspectRatio(contentMode: .fill)
* frame(width: cardSize.width, height: cardSize.height) (a cardSize a CGSize paraméter neve a closure-ben)
* clipShape(RoundedRectangle(cornerRadius: 5, style: .continous))

A CarouselView elemet is eláttam modifierekkel: „padding(.horizontal, -15)” és „padding(.vertical)”.

A CarouselView body-jában elhelyeztem egy GeometryReader objektumot, amelynek a célja az volt, hogy ne a SwiftUI határozza meg automatikusan, hogy az adott View-nak mekkora helyre van szüksége, illetve, hogy az hol helyezkedjen el a képernyőn, hanem azt kicsit szabadabban lehessen meghatározni. Ebben lekértem GeometryReader által visszaadott méretet (size), majd ez alapján meghatároztam a kártya számára rendelkezésre álló szélesség értéket: size.width – (cardPadding – spacing). Ezután elhelyeztem egy LazyHStack elemet a body-ban, amelybe felvettem az elemeket az items tömbből a megadott spacing értékkel. Az elemek megjelenítését a „content” függvénnyel valósítottam meg, ahol az aktuális elemet adtam meg az „Item.Element” típusú paraméternek és egy „CGSize(width: size.width – cardPadding, height: size.height)” értéket a CGSize típusúnak. A content függvény által visszaadott értéket is elláttam a megfelelő modifierekkel.

A gesztúra (lapozás) kezeléshez is felvettem a szükséges property-ket a CarouselView objektumban:

* @GestureState translation: CGFloat – ez adja meg, hogy felhasználó által kiváltott lapozás során, amíg az ujja a képernyőn van, aktuálisan mennyivel tolódott el az adott elem (0-val inicializálva)
* @State offset: CGFloat – hogy mennyivel tolódtak el az elemek a teljes tömbben (0-val inicializálva)
* @State lastStoredOffset: CGFloat – adott drag gesztúra végén mennyivel volt eltolódva a tömb (0-val inicializálva)
* @State currentIndex: Int – tömb aktuálisan kiválasztott elemének az indexe (0-val inicializálva)

Ezután a LazyHStack elemhez hozzáadtam egy „.gesture” modifiert, amelyben megadtam egy „DragGesture” objektumot 5-ös minimumDistance értékkel, amellyel megadtam a minimális húzandó távolságot, amelyet el kell érni, hogy a Drag Gesture sikeres legyen. Az „.updating” modifierrel megadtam, hogy a @GestureState property-t módosítsa az aktuálisan fennálló gesztúra állapot translation.width értékével. Az „.onChanged” és „.onEnd” modifierekhez létrehoztam külön eseménykezelő függvényeket:

* Az onChanged eseménykezelőben módosítottam az offset értéket olyan módon, hogy az aktuális DragGesture.Value.translation.width értéket hozzáadom a lastStoredOffset értékhez, és ezt adom értékül az offsetnek.
* Az onEnd eseménykezelőben az offset és a cardWidth értékek alapján meghatároztam egy \_index értéket a következő képelettel: (offset / cardWidth).rounded(), majd értékül adtam neki a következő két függvény értékét: max(-CGFloat(item.count - 1), \_index) és min(\_index, 0). Erre azért volt szükség, hogy az index értéke ne legyen kisebb 0-nál, és ne legyen nagyobb a tömb utolsó elemének az indexénél, továbbá azért negatív irányban történt az értékek vizsgálata, mert a tömböt „előrefele” lapozást balra draggel lehet elérni, amely miatt az offset értéke annál negatívabb, minél nagyobb indexű elemét vizsgáljuk a tömbnek. Az így kapott \_index értéknek a -1-szeresét adtam oda a currentIndex @State változónak. Ezután a „withAnimation” függvény meghívásával (.easInOut(duration: 0.25) paraméter megadással) módosítottam az offset értékét: offset = (cardWidth \* \_index) + extraSapce, ahol az extraSpace értéke: (cardPadding / 2) – spacing. Végül az így kapott offset értéket adom értékül a lastStoredOffset változónak. Ezutóbbira azért volt szükség, hogy a kártyák közül minden elem középen legyen, amikor „ki van választva”.

Annak érdekében, hogy kezdetben is középen legyen az első kártya, a GeometryReader .onAppear meodifierében is értékeül adom a korábban kiszámolt extraSpace értéket az offset és lastStoredOffset változóknak.

A scrollozhatóságot limitáltam az első és utolsó kártyán a következő módon: létrehoztam egy limitScroll függvényt, amely az első kártya és extraSpace-nél (azaz a kezdeti középre tolásnál) nagyobb offset esetén (index == 0 && offset > extraScpace) az új offsetnek csak a negyedét adja az offset értékhez, illetve az utolsó kártya és negatív (jobbra történő) drag esetén csak a drag érték felét vonja ki az offset értékéből. Ennek a függvénynek az értéket adom értékül a LazyHStack „.offset” modifierének.

Ezek mellett az AccommDetailsView képernyőre kitettem a „Földrajzi helyzet” label mellé egy kis kék tooltipet, amelyre kattintva a felhasználó egy alert formájában tajékoztatást kap arról, hogy a földrajzi koordinátákra tappolva azok megnyitásra kerülnek a Térképek alkalmazásban, és egy gombostű formájában megtekintheti, hogy pontosan hol találhtó az aktuálisan megtekintett szállás a térképen. Ehhez a földrajzi koordinátákat tartalmazó Text objektumon elhelyeztem egy „onTapGesture” modifiert, amely meghív egy általam definiált „openLocationOnMap” callback függvényt a koordinátákkal. Ez a függvény megfelelő típusú értékeket hoz létre a koordinátákból, majd ezek alapján létrehoz egy 10 km-es sugarú régiót a meghatározott pont körül, amelyet megjelenít a Térképek alkalmazásban. Ehhez a funkcióhoz beimportáltam a MapKit könyvtárat a swift fájlban.

Egy kisebb refaktoráláson esett át a BookingViewForm képernyő: lehetővé tettem, hogy a keresés indításakor egyszerre több fizetési mód is kiválasztható legyen. Ehhez módosítani kellett az „OtherNeedsListItemSelectedView” osztályt: ahhoz, hogy általánosabb felhasználású legyen és ezt a neve is tükrözze, átneveztem „ListItemSelectedView”-ra és átmozgattam a utils mappába. Ezt a változást le kellett követni több helyen is a kódban, illetve létrehoztam egy ListItem osztályt, amely átvette az eddigi „OtherNeed” osztály szerepét a lista létrehozásban. Az „OtherNeed” osztályból pedig eltávolítottam az „is\_selected”property-t, amelyre kizárólag a lista létrehozás miatt volt eddig szükség. Ezen kívül megjelenik a szállás részletei oldalon is, hogy milyen elfogadott fizetési módok állnak rendelkezésre az adott szálláson.

November 11.

Létrehoztam a kifejezetten a foglalási flow-hoz tartozó utolsó képernyőt. Ezen a képernyőn adhatja meg a user a foglalásainak az adatait, illetve a képernyő alján kap egy összefoglalást a foglalás adataival, amelyek az általa megadott adatok, illetőleg a szálláshoz tartozó releváns adatok alapján kerülnek meghatározásra.

A formban minden főbb résznek külön Section-t hoztam létre. Az első Sectionben elhelyeztem két DatePicker elemet, amelyeknek megadtam két @State property-t, amelyekben eltárolom a kiválasztott dátumokat, és amelyeket a keresési feltételekben megadott dátumokkal inicializálok a konstruktorban. Ezen kívül megadtam, hogy az adott napnál korábban nem lehet foglalni, illetve, hogy a foglalás végét jelző nap nem lehet korábban, mint a foglalás kezdetét jelző nap.

Ezt követően megadtam egy Picker elemet, amelyben a szobához tartozó maximálisan megadott vendégszám függvényében választhat a felhasználó a [1, maximális vendégszám] intervallumból egy értéket. A következő Sectionben lévő Picker használatával a felhasználó választhat egy fizetési módot. Az ezutáni Sectionben lehetőség van kiválasztani, hogy amennyiben az lehetséges, úgy ellenszolgáltatás igénybevételével is lehet fizetni. Ez a Section csak abban az esetben jelenik meg, amennyiben lehetséges az adott szállásnál ellenszolgáltatást igénybe venni, és ha a felhasználó ezt elfogadja, úgy egy kedvezményes ár fog megjelenni majd az összegzésben. A következő Sectionben lehetőség van megadni egy tetszőleges megjegyzést a foglaláshoz. Ezt egy TextEditor elemmel oldottam meg, amelyben elhelyeztem egy placeholder szöveget is szürke színnel, ha untouched a TextEditor. Amikor a user belekattint az editorba, akkor a placeholder eltűnik és a betűszín az elsődleges betűszínre változik.

Az oldal alján pedig található egy összegzés az foglalás adataival. Az itt szereplő értékek változnak a form értékek változásainak megfelelően. Az összefoglalásban a felhasználó ellenőrizheti a tartózkodás kezdetének és végének az időpontját, a kiválasztott fizetési módot, az árat (amelynél az ellenszolgáltatás elfogadása esetén az alacsonyabb érték jelenik meg), azt, hogy elfogadta-e az ellenszolgáltatást, az érkező személyek számát, illetve amennyiben írt megjegyzést a foglaláshoz, akkor abban az esetben az is megjelenik az összegzésben.

Az oldal legalján pedig található egy gomb, amely megnyomásával leadhatja a foglalást a megadott paraméterekkel.

November 14.

Létrehoztam a „másik” felhasználóhoz tartozó profil nézetet, amely szinte teljes mértékben analóg az adott felhasználóhoz tartozó profilnézettel, annyi különbségtől eltekintve, hogy ezen az oldalon egyetlen gomb található, amely az adott felhasználóval való csevegéseket nyitja meg. Az szállás részletei oldalra elhelyeztem egy gombot, amelyet megnyomva az adott szállást meghirdető felhasználó profil oldalára navigál az alkalmazás.

Ezeken felül felmértem, hogy milyen backend fejlesztések szükségesek még ahhoz, hogy megkezdődhessen a kliens és a backend alkalmazások összekötése.

November 15.

Több probléma is felmerült a MySQL adatbázissal, többek között számos alkalommal előfordult, hogy az alkalmazás bezárása után is tovább futott a háttérben a MySQL folyamat, és az alkalmazás újbóli elindítását ez akadályozta, mivel foglalta a default esetre beállított portot. Ezen felül is voltak vele problémák, emiatt jobbnak találtam lecserélni az adatbázis technológiát, amelyet a backend alkalmazás használ, és beállítottam a PostgreSQL-t mint adatbázis megoldást. A konfiguráció elvégzése után módosítottam az adatbázis inicializálását végző SQL szkriptet tartalmazó fájlt, illetve a spring alkalmazás konfigurációját végző application.yaml fájlt, a függőségeket kezelő pom.xml-t, illetve a docket-compose.yml-t, amely az adatbázis docker image-ben történő futtatásáért felelős.

November 18.

Implementálásra került backend oldalon a szállások listájának a visszaadása a megadott keresési feltételek alapján. A megadott keresési feltételek az „AccommodationRequest” objektumban kerülnek felküldésre a backendnek. A „RoomController” osztályban létrehoztam egy „@GetMapping” annotációval ellátott „getRoomsBasedOnSearchConditions” nevű metódust, amelynek a visszatérési értéke „ResponseEntity<List<RoomResponse>>”. Ez a metódus visszatér egy „ResponseEntity.ok(roomService. getRoomsBasedOnSearchConditions(accReq))” objektummal.

A controller objektum módosítása után módosítottam a service objetumot is, mégpedig olyan módon, hogy létrehoztam benne az előbb felhasznált metódust: ennek a visszatérési értéke „List<RoomResponse>”, paraméterének a típusa szinten „AccommodationRequest” és visszatérési értéke „roomMapper.toResponseList(roomRepository.findRoomsBasedOnSearchConditions(accReq))”.

A már meglévő repository osztályt ki kellett bővítenem egy „RoomRepositoryCustom” interface-szel. Erre azért volt szükség, mert a keresési feltételek alapján való adatbázis lekérés meglehetősen komplex művelet ebben az esetben, nem volt elegendő a Spring áltaé kigenerált repository függvények használata, illetve a „@Query” annotációban megadni az SQL lekérést. Ennek a megvalósításához CriteriaBuilder használatára volt szükség, ehhez pedig létrehoztam az említett Custom repository interface-t, majd implementáltam azt a „RoomRepositoryCustomImpl” osztályban.

Ebben az osztályban létrehoztam „@PersistenceContext” annotációval egy EntityManager property-t, majd felüldefiniáltam az interface-ben megadott query függvényt, amely az adatbázis lekérést hajtja végre. Ebben létrehoztam egy CriteriaBuilder objektumot, amely segítségével a lekérés paramétereit definiálom, illetve egy CriteriaQuery<Room> objektumot, amelyben magához a lekéréshez tartozó objektumot tárolom el. Létrehoztam továbbá a „Root<Room>” objektumot, amely az entitás referenciáját tarolja, illetve egy „List<Predicate>” objektumot, amelyben eltárolom a különböző feltételeket.

Az első ilyen feltétel az annak a meghatározása volt, hogy csak olyan Room objektumok kerüljenek bele a query-be, amelyek az adott városban találhatóak, majd a kérésben szereplő fizetési módok alapján szűrök a query-ben. Ezt követően ellenőriztem, hogy az adott keresési feltételek jelen vannak-e a kérésben (ár kategória, tulajdonos értékelése), illetve, hogy az egyéb szoba tulajdonságok megfeleljenek a keresési körülményeknek. Ezután kikötöm azt a feltételt, hogy csak azok a szállások kerülhetnek bele a keresés eredményébe, amelyek nem az adott felhasználóhoz tartoznak, azaz ne ő adta ki a szállásokat.

Végül egy subquery használatával kiszűröm a keresési eredményekből azokat a szállásokat, amelyek a keresési feltételekben megszabott időintervallumban foglaltak. Ehhez létrehoztam egy „public Predicate occupiedRoomsSubquery(CriteriaBuilder cb, CriteriaQuery<?> query, Root<Room> root, LocalDate startDate, LocalDate endDate)” karakterisztikájú metódust, amely létrehoz egy „Subquery<Long> subquery = query.subquery(Long.class)” lokális változót, illetve egy „Root<Booking> subroot = subquery.from(Booking.class)” lokális változót. Ezek után a subquery-ben lekérem azokat a Booking objektumokat (foglalásokat), amelyeknek a vége később van, mint a keresési feltételekben megadott kezdő dátum, és amelyeknek kezdete megelőzi a keresési feltételekben megadott befejező időpontot. Az ilyenformán megkapott Booking objektumokat ezután olyan módon használom fel, hogy csak azok a Room objektumok szerepelhetnek a lekérésben, amelyekhez nem tartozik a subquery által visszaadott Booking objektum.

<https://www.baeldung.com/spring-data-derived-queries>

<https://www.baeldung.com/jpa-many-to-many>

<https://www.baeldung.com/spring-data-jpa-query>

<https://www.baeldung.com/jpa-and-or-criteria-predicates>

<https://www.baeldung.com/spring-data-criteria-queries>

<https://stackoverflow.com/questions/19977130/joining-two-table-entities-in-spring-data-jpa>

<https://www.baeldung.com/spring-jpa-joining-tables>

<https://stackoverflow.com/questions/72162260/query-many-to-many-jpa>

<https://evonsdesigns.medium.com/spring-jpa-one-to-many-query-examples-281078bc457b>

<https://stackoverflow.com/questions/29145958/jpa-criteria-builder-onetomany-restrictions>

<https://stackoverflow.com/questions/48431944/jpa-criteria-api-filter-subentities/48441456#48441456>

Intervallum fedés vizsgálat: <https://stackoverflow.com/questions/3269434/whats-the-most-efficient-way-to-test-if-two-ranges-overlap>

Insertnél enum típus hiba megoldása: <https://stackoverflow.com/questions/65478350/error-column-is-of-type-json-but-expression-is-of-type-character-varying-in-hib> - fel kell annotálni "@ColumnTransformer(write = "?::<enum\_type>")" annotációval.

November 22.

Implementálásra kerültek a Booking táblához tarotó komponensek. Ebben a fejlesztésben elkészült a BookingController osztály, amely a „/api/booking” végponthoz tartozó működéseket valósítja meg. Itt a „/api/booking/owner/{id}” végpont egy GET hívásra visszaadja azon BookingResponse-ok listáját, amelyek az adott id-jú felhasználó által meghirdetett szobákhoz tartoznak. Ugyanez a működés meg lett valósítva a bérlő oldalon is, csak ebben az esetben a „/api/booking/tenant/{id}” végponton keresztüli GET kérésre. Illetve megvalósításra került a „/api/booking” végponton való POST hívás, amely kérésre létrehozásra kerül egy foglalás objektum. A controller ezekhez a végpont hívásokhoz tartozó függvényeket implementálja, és mindegyik függvény a BookingService objektumon keresztül végzi el a működését. A BookingService interface-ben felsoroltam az implementálandó függvényeket, ezek a „createBooking”, „getAllBookingsForTenant”, „getAllBookingsForOwner” és „deleteBooking”. Ez utóbbihoz egyelőre nem tartozik végpont, de a későbbiekben megvalósításra kerülhet ilyen működés, emiatt létrehozásra került.

A BookingServiceImpl osztályban kerülnek implementálásra az előbb felsorolt függvények az interface-ben. A „createBooking” függvényben először ellenőrzésre kerül, hogy létezik-e a foglalást leadó felhasználó, illetve a lefoglalandó szálláshely. Abban az esetben, ha nem léteznek, egy „EntityNotFoundException” kivétel kerül eldobásra. Majd ellenőrzésre kerül, hogy a foglalást leadó személy és a szálláshely tulajdonosa nem ugyanaz a személy-e. Amennyiben igen, úgy egy „BookingOwnRoomException” kerül eldobásra. (Ez az eset hipotetikusan fordulhat csak elő, mert foglálaskor a szállás találat listában csak olyan szállások jelenhetnek meg, amelyek nem az adott felhasználó által vannak meghirdetve.) Ezután lekérésre kerül az adott szobához minden foglalás, és ha az adott időintervallumban van már leadott foglalás az adott szobára, egy „BookingAnAlreadyBookedRoomException” dobódik (ez is hipotetikus eset, mert csak a találat listában csak olyan szállások jelenhetnek meg, amelyek az adott időintervallumban nem foglaltak). Ha egyik kivétel sem váltódott ki, akkor a BookingRequest objektum átalakításra kerül egy Booking (entitás) objektummá, hozzárendelődik a foglaló User és a foglalt szoba Room objektum, majd mentésre kerül. A „getAllBookingsForTenant” és „getAllBookingsForOwner” függvényekben csupán meghívásra kerülnek a BookingRepository interface megfelelő query-ket végrehajtó függvényei.

A BookingRepository interface-ben definiáltam a „List<Booking> findByRoomId(Long id)”, „List<Booking> findByTenantId(Long id)” és „List<Booking> findByOwnerId(Long id)” függvényeket. Az előbbi kettőnél elegendő volt az automatikusan a név alapján kikövetkeztett működés, az utóbbinál azonban definiálni kellett egy query-t. Ebben megadtam, hogy a booking tábla minden attribútuma kerüljön kiválasztásra, egy room táblával való join után, és minden olyan rekord kerüljön kiválasztásra, ahol a room táblában az owner\_id attribútum megegyezik a függvényparaméterben megadott id-val.

November 25.

Létrehoztam a Leaderboard táblához tartozó komponenseket. Itt is első lépésként létrehoztam a LeaderboardController osztályt, amelyben felsoroltam a végpontokhoz tartozó működéseket megvalósító függvényeket. A „/api/leaderboard?role={value}” végponthoz tartozó GET kéréseket a „getLeaderboardScores” függvény kezeli. Ha a „value” értéke „tenant”, akkor a LeaderboardService objektum „getScores” függvényét „LeaderboardRole.TENANT” értékkel hívja meg, ha pedig a „value” értéke „owner”, akkor a „getLeaderboardScores” függvényt „LeaderboardRole.OWNER” értékkel hívja meg. Mivel csak ez a két érték lehetséges, ezért, ha a „value” értéke eltér ezektől, akkor egy „CouchsurfingRuntimeException” kivétel dobódik el. A „/api/leaderboard?role={value}” végponthoz intézett POST kéréseket pedig a „createLeaderboardScore” függvény kezeli. Ha a „value” értéke „tenant”, akkor a „LeaderboardService” osztály „createTenantScore” függvénye hívódik meg a kérés törzsében megadott objektummal, ha pedig a „value” értéke „owner”, akkor a „createOwnerScore” függvény hívódik meg. A kivétel eldobása az előzővel analóg módon történik.

A „LeaderboardService” interface-ben létrehoztam a „List<LeaderboardProjection> getScores(LeaderboardRole role)”, „Long createTenantScore(LeaderboardRequest lbReq)” és „Long createOwnerScore(LeaderboardRequest lbReq)” függvényeket. Ezeket implementálom a „LeaderboardServiceImpl” osztályban. A „getScores” függvény meghívja a „LeaderboardRepository” objektum „getTenantScore” vagy „getOwnerScore” függvényét a megadott „role” paraméter szerint. A „createTenantScore” függvényben ellenőrzésre kerül, hogy a kérésben megadott „tenantScore” mező nem null-e, mert amennyiben igen, úgy egy „CouchsurfingRuntimeException” kivétel dobódik. Ezután ellenőrzésre kerül, hogy az értékelő és az értékelt felhasználók léteznek-e, ellenkező esetben „EntityNotFoundException” kivétel dobódik, majd ellenőrzésre kerül, hogy létezik-e olyan foglalás, ahol az értékelt felhasználó mint bérlő, az értékelő felhasználó pedig mint kiadó van jelen. Ha nincs ilyen, akkor „CanNotRateUserWithoutBookingHistoryException” kivétel dobódik. Ha egyik kivételes esetbe sem futott bele a működés, akkor a kérés alapján létrehozásra kerül az entitás objektum, majd mentésre kerül az adatbázisban. A „createOwnerScore” függvény működése szinte teljes mértékben analóg a „createTenantScore” függvény működésével, csak annyi különbség van, hogy itt az kerül ellenőrzésre, hogy a request objektum „hostScore” mezője null-e, illetve, hogy létezik-e olyan Booking objektum, ahol az értékelt felhasználó mint kiadó, az értékelő felhasználó pedig mint bérlő van jelen.

A „LeaderboardRepository” interface-ben két függvényt vettem fel, ezek a „getOwnerScore” és „getTenantScore” függvények. A „getOwnerScore” függvény visszaadja annak a query-nek az eredményét, amelyben egy join-t hajtok végre a leaderboard és couchsurfing\_user táblák között, és lekérem a leaderboard táblából a „user\_id”, az „AVG(leaderboard.host\_score)” mint „score” értékeket, a couchsurfing\_user táblából pedig a „full\_name” és „user\_name” attribútumokat. A query-t szűrtem azokra az esetekre, ahol a „host\_score” attribútum nem NULL és nem 0 értékű, csoportosítottam a „full\_name”, „user\_id” és „user\_name” attribútumok alapján és a „score” alapján rendeztem csökkenő sorrendbe. A „getTenantScore” függvényhez tartozó query majdnem teljes mértékben analóg volt az imént leírt lekéréssel, annyi különbséggel, hogy ebben az esetben a leaderboard táblából a „tenant\_score” attribútum került lekérésre a „host\_score” helyett minden olyan helyen, ahol az előző esetben a „host\_score” volt használva. A két függvény visszatérési értékének a típusa „List<LeaderboardScoreProjection>”, ehhez pedig létrehoztam egy „LeaderboardScorePorjection” interface-t. Erre azért volt szükség, mert az adatbázisban tárolt objektumoknak megfelelő „Leaderboard” entitás mezői és a visszaadni kívánt response objektum mezői között eltérések voltak, emiatt a mapper objektum nem tudtam megfelelően feldolgozni a query eredményét és leképezni a visszaadni kívánt objektumot.

<https://www.baeldung.com/spring-request-param>

<https://www.baeldung.com/jpa-queries-custom-result-with-aggregation-functions> - data projection

<https://stackoverflow.com/questions/2421388/using-group-by-on-multiple-columns>

<https://postgrespro.com/docs/postgresql/9.4/functions-comparison>

<https://stackoverflow.com/questions/46083329/no-converter-found-capable-of-converting-from-type-to-type> - projection

November 27.

WebSocket oldalak:

<https://spring.io/guides/gs/messaging-stomp-websocket/> - WebSocket alapok

<http://stomp.github.io/stomp-specification-1.2.html#Abstract>

<https://www.baeldung.com/websockets-spring> - WebSocket alapok

<https://www.baeldung.com/spring-websockets-sendtouser> - @SentToUser annotáció

<https://www.youtube.com/watch?v=TywlS9iAZCM>

<https://gabrielpulga.medium.com/building-a-real-time-chat-application-with-spring-boot-and-websocket-4ad1c68ee217>

<https://medium.com/@AlexanderObregon/using-sendto-and-sendtouser-in-spring-websocket-for-message-routing-f503fd74047f> - @SentTo és @SendToUser annotációk

<https://www.baeldung.com/spring-websockets-send-message-to-user> - message to specific user

<https://stackoverflow.com/questions/38323193/what-is-setapplicationdestinationprefixes-being-used-for>

<https://www.baeldung.com/spring-data-jpa-pagination-sorting> - pagination

Elkészült a Chat táblához tartozó funkcionálitás. Első lépésként felvettem a projekt struktúráját összefoglaó pom.xml fájlba egy új dependenciát: ez a „spring-boot-starter-websocket” dependencia, majd létrehoztam egy „WebSocketConfig” nevű osztályt, amely WebScoket protokollon keresztül történő kommunikáció konfigurációjáért felelős. Ez az osztály implementálja a WebScoketMessageBroker interface-t, illetve elláttam a „@Configuration” és „@EnableWebSocketMessageBroker” annotációkkal. Az osztályban az interface két függvényét írtam felül, az egyik a „configureMessageBroker” függvény, amellyel először engedélyeztem egy in-memory message brokert, amely a klienseknek való üzenettovábbításban játszik szerepet, amely kliensek a „/”-lal prefixált útvonalakra vannak feliratkozva. Majd ugyanebben a függvényben beállítottam a „/app” prefixet, amellyel azt állítottam be, hogy a kliens által STOMP protokollon keresztül küldött üzenetek akkor kerülnek feldolgozásra a controllerben a „@MessageMapping” annotációval ellátott metódusok által, ha a kliens azokat valamilyen „/app/...” útvonalra küldi. A másik metódus, amit felüldefiniáltam pedig a „registerStompEndpoints” metódus volt, amelyben regsiztráltam a „/chat” végpontot mint STOMP végpontot, amelyre csatlakozhat kliens. Ezután létrehoztam a funkcióhoz tartozó controller osztályt, amely a „ChatController” nevet kapta. Ezt az osztályt elláttam a „@Slf4j”, „@Controller” és „@RequiredArgsConstructor” annotációkkal. Az osztály rendelkezik egy „ChatService” típusú és egy „SimpMessageTemplate” típusú property-vel, az előbbi felelős a funkcióhoz tartozó üzleti logikai funkcionalitások megvalósításáért, az utóbbi pedig az üzenetek kliensek felé továbbításáért felelős.

A controller osztályban létrehoztam egy „getMessagesForConversation” nevű metódust, amely egy „ResponseEntity<List<ChatMessageResponse>>” objektummal tér vissza, amennyiben a kliens egy GET kérést küld a „/api/chat?currentUserId={value}1&partnerUserID={value2}” végpontra és query paraméterekként megadja a saját felhasználói azonosítóját, illetve annak a felhasználónak a felhasználói azonosítóját, akivel a csevegést folytatja. Ekkor a „chatService” objektum „getMessagesForConversation” függvénye hívódik meg a query paraméterként megadott felhasználói azonosító értékekkel. A service objektum metódusának a hívása sikeresen lefut, akkor controller osztály metódusa visszatér a „ResponseEntity” objektummal.

Az osztályban felvettem ezen kívül egy „createMessage” nevű metódust, amely akkor hívódik meg, ha a felhasználó a „/api/chat” végpontot egy POST kérésben hívja meg, és egy „ResponseEntity<ChatMessageResponse>” objektummal tér vissza. Ekkor a kliens a kérés törzsében elküld egy „ChatMessageRequest” típusú objektumot. A metódus először meghívja a „chatService” objektum „createChatMessage” metódusát a kérés body-jában elküldött objektummal, majd a metódushívás eredményét eltárolja egy lokális változóban. Ezt követően a „SimpMessageTemplate” típusú objektum „convertAndSend” metódusával elküldi a metódushívás eredményét a „/chat/{fromUserId}/{toUserId}” útvonalra, amely útvonalon található az a topic, amelyre a kliens feliratkozik a WebSocket kapcsolat megnyitásakor. (A WebSocket kapcsolat megnyitásakor minden kliens feliratkozik egy olyan topicra, amelyben a beszélgetésben részt vevő másik felhasználó azonosítója az első ID az útvonalon, a saját felhasználói azonosítója pedig a másodikként szereplő ID az útvonalban.) Végül egy ResponseEntity-ben elküldöm annak a kliensnek is a „chatService” objektum metódushívásának az eredményét, amelyik a HTTP kérést elküldte a backend felé.

A „ChatService” interface-ben létrehoztam egy „createChatMessage” nevű metódust, amely egy „ChatMessageRequest” típusú objektumot vár paraméterként és egy „ChatMessageResponse” típusú objektummal tér vissza, illetve „getMessagesForConversation” nevű metódust, amely két Long típusú paramétert vár – az egyik a küldő felhasználó azonosítója, a másik a fogadó felhasználó azonosítója – és egy „List<ChatMessageResponse>” objektummal tér vissza. Ezt a két metódust implementáltam a „ChatServiceImpl” osztályban. A „createChatMessage” függvény implementációjakor először megpróbáltam létrehozni a megadott felhasználói azonosítók alapján a küldő és fogadó „User”entitásokat. Amennyiben nem léteznek a megadott felhasználói ID-k által azonosított felhasználók, úgy egy-egy „EntityNotFoundException” kerül eldobásra a megfelelő tájékoztató üzenettel. Ezt követően a „ChatMessageRequest” objektumból a „ChatMapper” típusú objektum segítségével létrehoztam egy „Chat” példányt, amely a „chat” tábla ORM-jének felel meg, majd az így létrehozott objektumnak a „messageTime” property-jére beállítottam az éppen aktuális időt, illetve a „userFrom” és „userTo” property-k értékének is beállítottam a korábban létrehozott „User” entitás objektumokat. Ezt követően a „Chat” entitás objektumot a „ChatRepository” típusú property-vel elmentettem az adatbázisba, majd pedig az entitás objektum alapján a „ChatMapper” típusú objektum által létrehozott „ChatMessageResponse” objektummal visszatértem a metódusból. A „getMessagesForConversation” metódusban is ellenőriztem, hogy a megadott felhasználói azonosítókkal rendelkező felhasználók valóban léteznek-e, ha pedig nem, akkor „EntityNotFoundException” kivételt dobtam el. A következő lépésként létrehoztam egy „Pageable” típusú lokális változót, amely az 50 legfrissebb üzenet lekérésért felelős, azaz a 0. oldalt kéri el, amelynek a mérete 50 és a „message\_time” oszlop szerint van csökkenő sorrendbe rendezve. Ezután meghívtam a „ChatRepository” objektum „findMessagesForConversation” metódusát a paraméterben megadott felhasználói ID-kkal, illetve az imént létrehozott „Pageable” objektummal. Az így kapott „List<Chat>” típusú eredményt a „ChatMapper” segítségével „List<ChatMessageResponse>” típusúvá alakítom, majd visszatérek vele.

A „ChatRepository” interface-ben egyedül a „findMessagesForConversation” nevű metódust vettem fel, amely paraméterben várja az a HTTP kérést küldő felhasználó és a beszélgető partnerének a felhasználó azonosítóit, illetve egy „Pageable” típusú objektumot, amellyel meg lehet adni, hogy mennyi üzenetet szeretnénk visszakapni a lekérésből. Itt a „@Query” annotációban megadtam a lekérdezést, amelyben definiáltam, hogy minden attribútum értékét le szeretném kérni, és csak azokat a rekordokat szeretném megkapni a lekérdezésben, ahol a két felhasználói azonosító szerepel, akár küldő-fogadó, akár fogadó-küldő kapcsolatban.

Ezen kívül elkészültek a „ChatMessageRequest” és „ChatMessageResponse” objektumok, illetve a „Chat” Entity objektum, ami a „chat” tábla ORM leképezése.