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.