**JEGYZŐKÖNYV**

Mobil programozási alapok

Féléves feladat

Teendőlista- és feladatkezelő alkalmazás

Készítette: **Ilyó-Kovács Levente**

Neptunkód: **NLFUA8**

Dátum: **2024. 12. 07.**

# Bevezetés:

A projektem egy egyszerű, modern mobilalkalmazás, amely a felhasználók napi rutinjának menedzselésére készült. Az app lényege, hogy egy felhasználóbarát és gyors módot biztosítson a feladatok rögzítésére, a teljesítések nyomon követésére, valamint a már nem szükséges elemek eltávolítására. A fejlesztés során a React Native technológiára építettem, melyhez az Expo-t használtam a fejlesztési folyamat felgyorsítása érdekében. Az adatok tárolását és szinkronizálását egy valós idejű adatbázis, a Firebase Firestore végzi, amely stabil és megbízható alapot nyújtott. Az alkalmazás minden részlete arra lett optimalizálva, hogy a felhasználók számára egyszerű legyen a kezelhetőség, miközben a funkcionalitás megmaradjon letisztult és átlátható formában.

A screenshot of a phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a phone

Description automatically generated

# Az alkalmazás funkciói

 **Regisztráció**: A felhasználók képesek új fiókot létrehozni, mely a firebase authentikációját használja így biztonságosan eltárolhatóak az adatok.

 **Belépés**: A felhasználók be tudnak lépni a már regisztrált fiókjukba, így az teendőlistájuk automatikusan szinkronizálásra kerül.

 **Teendők hozzáadása**: A felhasználók képesek új teendőket hozzáadni a listához. A teendőket az alkalmazás Firebase Firestore adatbázisába menti.

 **Teendők státuszának szerkesztése:** A hozzáadott teendők a felhasználói felületen megjelennek, státuszukat egy checkbox jelzi. Amennyiben a checkbox bepipált, a teendő késznek tekinthető.

 **Teendők törlése**: A felhasználók törölhetik a teendőket a listából. A törlés az adatbázisból is eltávolítja a teendőket, biztosítva az adatok valós idejű frissítését.

 **Felhasználóbarát felület**: Az alkalmazás egyszerű, könnyen navigálható és modern felhasználói felülettel rendelkezik, amely segíti a felhasználókat a könyveik kezelésében. A Material UI szerű esztétikai megjelenés is hozzájárul a kellemes használati élményhez.

 **Valós idejű adatkezelés**: • Az alkalmazás a Firebase Firestore adatbázis segítségével biztosítja az adatok azonnali szinkronizációját, így a teendők listája automatikusan frissül minden új feladat hozzáadása vagy meglévő módosítása esetén.

# A TodoItem komponens funkciói

*export* interface TodoItemProps {

  createdAt?: Timestamp;

  completedAt?: Timestamp | string;

  docId: string;

  todo: string;

  isCompleted: boolean;

  ownerId: string;

}

A TodoItemProps egy TypeScript interfész, amely egy teendő adatainak típusait definiálja. Tartalmazza:

* **createdAt és completedAt**: opcionális időbélyegek (dátum/idő).
* **docId**: a dokumentum egyedi azonosítója.
* **todo**: a teendő szöveges leírása.
* **isCompleted**: a feladat állapota (kész-e vagy sem).
* **ownerId**: a felhasználó azonosítója.

Ez az interfész biztosítja az adatok típusbiztonságát és az egységes Firestore-integrációt.

* const *checkAsCompleted* = *async* ()=>{
* *try* {
* const *index* = *tasks.findIndex*((*task*)=> *task.docId* === *docId*);
* const *updatedTasks* =[...*tasks*];
* *updatedTasks*[*index*]*.isCompleted* =!*updatedTasks*[*index*]*.isCompleted*;
* *setTasks*(*updatedTasks*);
* *setCompleted*(!*completed*);
* *await updateTodoItem*(*docId*,!*isCompleted*);
* } *catch* (*error*: *any*){
* *Alert.alert*("Valami hiba történt.", *error.message*);
* }
* };

Ez a checkAsCompleted egy aszinkron függvény, amely egy teendő státuszának (kész vagy nem kész) váltására szolgál.

**Lépések:**

1. **Feladat keresése:** A findIndex segítségével megkeresi a megfelelő teendőt az tasks tömbben az docId alapján.
2. **Státusz frissítése:** Egy új tömbmásolatot hoz létre (updatedTasks), majd frissíti az adott teendő isCompleted értékét az ellenkezőjére.
3. **Állapotfrissítés:** Frissíti a helyi állapotot (setTasks és setCompleted).
4. **Adatbázis szinkronizálása:** A updateTodoItem hívással frissíti a változást a távoli adatbázisban.
5. **Hiba kezelése:** Ha valami hiba történik, egy figyelmeztetés jelenik meg az Alert segítségével.

Ez egy valós idejű frissítést biztosító funkció, amely kezeli a lokális és távoli adatok szinkronját.

  const *deleteMyTodo* = *async* ()=>{

*try* {

const *updatedTasks* = *tasks.filter*((*t*)=> *t.docId* !== *docId*);

*setTasks*(*updatedTasks*);

*await deleteMyTodoItem*(*docId*);

} *catch* (*error*: *any*){

*Alert.alert*("Valami hiba történt.", *error.message*);

}

};

A deleteMyTodo egy aszinkron függvény, amely egy teendő törléséért felel.

**Lépések:**

1. **Teendő eltávolítása a listából:** A filter metódussal létrehoz egy új tömböt, amelyből a megadott docId-hez tartozó teendőt eltávolítja.
2. **Állapot frissítése:** Frissíti a lokális tasks állapotot a módosított tömbbel (setTasks).
3. **Adatbázis szinkronizálása:** Meghívja a deleteMyTodoItem függvényt, hogy eltávolítsa az adott teendőt a távoli adatbázisból.
4. **Hiba kezelés:** Ha hiba történik, egy figyelmeztetés jelenik meg az Alert segítségével.

Ez a funkció a lokális és távoli adatok törlését is kezeli, egyszerűen és hatékonyan.

# ****Állapotkezelés (State Management)****:

A programban statekezelés a useState hook segítségével valósul meg, amely a komponensek állapotainak kezelésére szolgál. Ez lehetővé teszi a dinamikus működést, mivel az állapotváltozások automatikusan újrarenderelik a komponenst.

**Példák a statekezelésre:**

1. **Teendők listája:**
   * A tasks state tárolja a felhasználó teendőit. Ha egy új teendő kerül hozzáadásra vagy egy meglévő törlésre kerül, a setTasks segítségével frissítik a listát, amely azonnal megjelenik a felhasználói felületen.
2. **Feladat állapota:**
   * A completed state nyomon követi, hogy egy adott feladat be van-e fejezve. Ha a felhasználó megjelöl egy teendőt késznek, a setCompleted frissíti az állapotot, és a változás vizuálisan is láthatóvá válik (pl. egy checkbox kipipálása).
3. **Adatbetöltés állapota:**
   * A loading state mutatja, hogy egy aszinkron művelet (pl. teendők törlése vagy frissítése) éppen zajlik-e. Ez lehetőséget ad arra, hogy a felhasználónak betöltési jelzőt mutasson, például egy spinner ikon formájában.
4. **Felhasználói adatok:**
   * Az email és password state-ek tárolják a felhasználó által bevitt adatokat, például a bejelentkezés vagy regisztráció során. Ezek segítségével történik az űrlap adatainak kezelése és validálása.

**Miért fontos?**

A statekezelés teszi lehetővé, hogy az alkalmazás dinamikusan reagáljon a felhasználói műveletekre. Például, ha egy feladatot teljesítettnek jelölnek, a state azonnali frissítésével a változás láthatóvá válik anélkül, hogy az oldalt újratöltenénk. Ez javítja a felhasználói élményt és egyszerűsíti a fejlesztést.

# Kontextus létrehozása az állapot megosztásához

A DataContext egy React kontextus, amely lehetővé teszi a tasks (teendők listája) és a setTasks (állapotfrissítő függvény) megosztását az alkalmazás különböző komponensei között. Ez a megoldás megkönnyíti az állapot kezelését, mivel nincs szükség arra, hogy minden szülő-komponens propokon keresztül továbbadja az állapotot a gyermek-komponenseknek.  
Az alapértelmezett érték egy üres teendőlista (tasks: []) és egy üres setTasks függvény. Az utóbbi egy üres funkcionalitásként van definiálva, hogy elkerüljük a hibákat a komponensek kezdeti állapotában, amíg a kontextust valódi értékekkel fel nem töltjük.

*import* { createContext } *from* "react";

*import* { TodoItemProps } *from* "../todo/TodoItem";

interface DataContextProps {

  tasks: TodoItemProps[];

  setTasks: React*.*Dispatch<React*.*SetStateAction<any[]>>;

}

*export* const *DataContext* = *createContext*<*DataContextProps*>({

*setTasks*:()=>{},

*tasks*:[],

});

# Teendők rendezése teljesítési állapot szerint

A sortItemsByCompletion egy segédfüggvény, amely egy teendők listáját rendezi azok isCompleted állapota alapján. A kész feladatok a lista végére kerülnek, míg a még nem teljesített feladatok előre sorolódnak.

* **Működési elv:** Az arr.sort metódust használja, ahol összehasonlítja a lista elemeinek isCompleted értékét. Ha egy elem teljesített (isCompleted: true), és a másik nem, akkor az előbbi hátrébb kerül a listában.
* **Használat:** Ez a függvény hasznos, ha egy teendőlista áttekinthetőbbé tételére van szükség, például ha a felhasználó a még elvégzendő feladatokra akar fókuszálni.

Ez a megközelítés együtt lehetővé teszi az állapot egyszerű megosztását és a lista dinamikus rendezését, javítva a felhasználói élményt.

*export* function *sortItemsByCompletion*(arr: any[]) {

*return* arr*.sort*((a, b) => {

    if (a*.*isCompleted && !b*.*isCompleted) {

*return* 1;

    } else if (!a*.*isCompleted && b*.*isCompleted) {

*return* -1;

    } else {

*return* 0;

    }

  });

}

# ****Felhasználói Felület (UI)****:

Az alkalmazás felhasználói felülete (UI) modern, egyszerű és felhasználóbarát, amely a React Native komponenseire és stílusára épít. A főképernyőn a felhasználót egy üdvözlő szöveg fogadja, amelyet a következő kód valósít meg:

<Text *style*={styles.greeting}>

  Helló, <Text *style*={{ *color:* "teal" *}}*>{user?.displayName || "user"}</Text>

</Text>

Itt a Text komponens stílusa italikus és vastag, hogy barátságos és kiemelkedő legyen. A név dinamikusan jelenik meg a felhasználói adatok alapján. Az új teendők hozzáadása egy sorban elrendezett mezőből és gombból áll, amelyet a TextInput és a TouchableOpacity komponensek kezelnek:

<View style={styles.row}>

  <TextInput

    style={styles.input}

    placeholder="Adjunk hozzá valami újat..."

    onEndEditing={addToList}

    returnKeyType="done"

    onChangeText={setTodo}

    defaultValue={todo}

  />

  <TouchableOpacity

    style={styles.button}

    onPress={addToList}

    disabled={loading}

  >

    {loading ? (

      <ActivityIndicator animating={loading} color={"white"} />

    ) : (

      <MaterialIcons name="add" size={30} color="white" />

    )}

  </TouchableOpacity>

</View>

, hogy a felhasználók könnyedén hozzáadhassanak új teendőket. Az input mező jól látható, a fehér háttér és a finom árnyék segítségével vizuálisan elkülönül. A hozzáadás gomb egy "+" ikont tartalmaz a MaterialIcons csomagból, amely betöltéskor automatikusan egy ActivityIndicator-re vált, ezzel visszajelzést adva a felhasználónak. A teendők listázása egy FlatList komponenssel történik, amely dinamikusan rendereli az elemeket, és üres lista esetén egy Empty komponenst jelenít meg:

<FlatList

*data*={sortedTodos}

*renderItem*={({ *item* *})* => <TodoItem *data*={item} *key*={item.docId} />}

  ListEmptyComponent={Empty}

  contentContainerStyle={styles.content}

  ItemSeparatorComponent={() => <View *style*={{ *height:* *10* *}}* />}

  showsVerticalScrollIndicator={false}

/>

A lista a sortItemsByCompletion függvénnyel van rendezve, amely előresorolja a még el nem végzett teendőket, így azonnal a fontosabb elemeket láthatjuk. A megjelenés reszponzív, a szélességek százalékos arányú megadása biztosítja, hogy a felület különböző eszközméreteken is jól nézzen ki. A gombok és mezők lekerekített szélei (borderRadius) és a vizuális árnyékok (shadowOffset, elevation) modern, kártyaszerű megjelenést kölcsönöznek az alkalmazásnak, amely letisztultságot és könnyedséget sugároz a felhasználók számára.

Az alkalmazás stílusa modern és felhasználóbarát, amely a vizuális egyszerűséget és funkcionalitást egyensúlyban tartja. A reszponzív elrendezést a flexbox alapú stílusok biztosítják, például a feladatok és beviteli mezők egy sorba helyezésével. Az árnyékok és lekerekített sarkok kiemelik az interaktív elemeket, mint például a gombokat és mezőket, így azok vizuálisan megkülönböztethetők és könnyen használhatók.

A színek, például a teal és a fehér kombinációja, letisztult megjelenést ad, miközben a halvány szürke szövegszín a visszafogott, mégis olvasható tartalom megjelenítését támogatja. Az alkalmazás külön figyelmet fordít arra, hogy a listák, gombok és mezők közötti terek megfelelőek legyenek, így a felület nem zsúfolt, de nem is túl szellős. Az egész felület úgy lett kialakítva, hogy vizuális hierarchiát teremtsen, ahol a legfontosabb elemek (pl. beviteli mező, lista) azonnal észlelhetők, és könnyen elérhetők a felhasználók számára.

# ****Alkalmazás indítása****

* Az alkalmazás indításához szükséges egy terminálban futtatni az npm i && npx **expo start** parancsot.
* Ez a folyamat letölti a függőségeket ás elindítja az Expo fejlesztői szervert, amely egy QR-kódot jelenít meg a terminálban vagy egy megnyíló böngészőben.
* A generált QR-kód beolvasásához egy eszközön telepített **Expo Go** alkalmazás használata szükséges, amely lehetővé teszi az alkalmazás futtatását közvetlenül a készüléken.

