Modern iOS alkalmazás fejlesztése

Önálló laboratórium 2 – Dokumentáció

Készítette: Dálnoky Bertalan András – IKLE6K
Konzulens: Dr. Blázovics László
Tanszék: Automatizálási és Alkalmazott Informatikai Tanszék

Ötlet bemutatása

A projekt alapötlete egy idő és feladat menedzsmentet segítő alkalmazás létrehozása *iOS* platformon. Az alkalmazás ezt olyan funkciók megvalósításával, és kombinálásával kívánja megvalósítani, mint a naptár és *Kanban* tábla, ami a feladatok vizuális követését teszi lehetővé. Az alkalmazásnak az egyik fő célja, hogy a felhasználóknak minél több szabadságot adjon a feladataik követésében, és a folyamataik személyre szabhatóságában.

A főbb funkcionális elvárások a következőek. A felhasználó legyen képes létrehozni, szerkeszteni és törölni feladatokat, valamint a létrehozott feladatokat napokra lebontva legyen képes megtekinteni és kezelni egy, a *Kanban* tábláknak megfelelő nézet segítségével. Ezen felül legyen képes keresni már létrehozott feladatok között, valamint szűrők segítségével szűkíteni a megtekintett feladatokat. Az aktuális naphoz a feladatai mellett a felhasználó legyen képes megtekinteni egy vizuális indikátort, ami a napi feladataival való haladását szemlélteti.

A feladatoknak legyen címe, leírása, kezdési időpontja, befejezési időpontja, valamint legyen lehetőség lépéseket hozzáadni a feladatokhoz. Ezeken kívül minden feladat rendelkezzen egy prioritással és kategóriával, valamint minden feladat tartozzon egy *Kanban* tábla szerinti oszlopba. A prioritások, kategóriák és oszlopok esetében az alkalmazás nyújtson alapértelmezett értékeket, azonban a felhasználó legyen képes ezeket szerkeszteni, törölni, illetve új értéket felvenni.

Az alkalmazás mellett a felhasználó legyen képes az aktuális naphoz tartozó feladatai, illetve a napi haladása megtekintésére az alkalmazáson kívül is, *Widget* komponenseken keresztül.

Az alkalmazás egyik célja, hogy az *iOS* platformon kívül, hogy az *Apple* ökoszisztéma többi lényegesebb platformján is elérhető legyen a jövőben és a felhasználó ugyanazokat az adatokat érje el minden eszközéről. Ezért az applikáció már ennek a célnak a figyelembevételével *iCloud* adat szinkronizációval lett kialakítva.

Ez a dokumentáció ennek az alkalmazásnak az implementációját tárgyalja.

Felhasznált technológiák

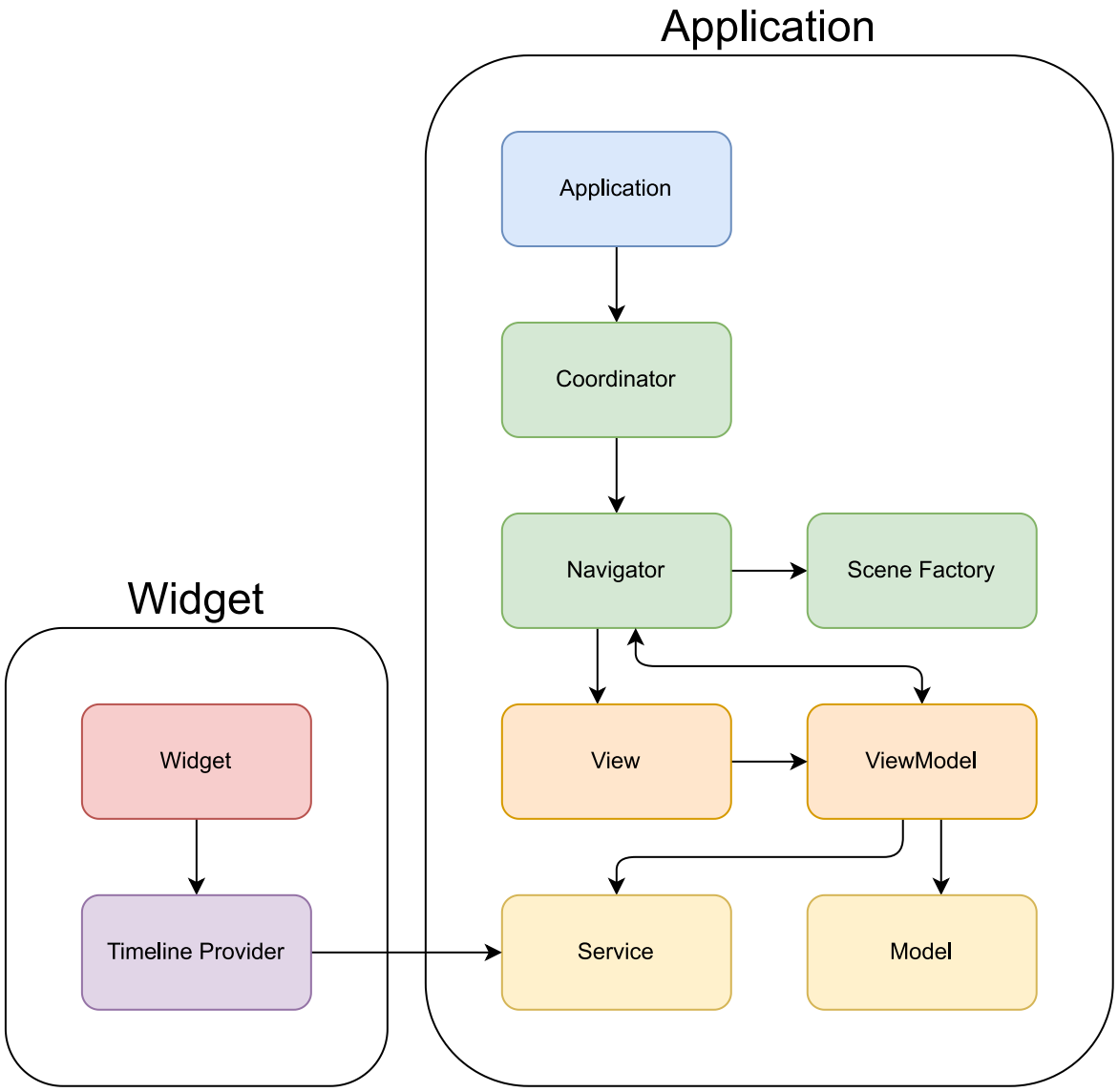
Mivel ez a dokumentáció nem a felhasznált technológiákat helyezi előtérbe ezért ezen technológiák csupán felsorolás és említés szinten kerülnek bemutatásra. Ennek megfelelően az alkalmazás az alábbi keretrendszerekre, könyvtárakra és eszközökre támaszkodik.

- *SwiftUI* – a felhasználói felület keretrendszere
- *Combine* – aszinkron események kezelését végző keretrendszer
- *Core Data* – az adatok perzisztens tárolását kezelő keretrendszer
- *iCloud* – az adatok eszközök közötti megosztását támogató felhő alapú adattár
- *Swiftlint* – konzisztens kódstílust kényszerítését segítő eszköz
- *Resolver* – függőség injektálást támogató eszköz
- *SwiftGen* – típusbiztos szövegek generálására alkalmas eszköz
- *CocoaPods* – csomagkezelő eszköz

Megvalósítás

Architektúra

Az alkalmazás a *Model-View-ViewModel* (*MVVM*) architektúrára épül a *Coordinator* mintával kiegészítve. A teljes architektúrát vizualizáló ábra az alábbi képen látható.



Az *MVVM* architektúrában a *Model* réteg az alkalmazásban szereplő logikailag összetartozó adatok egységbe foglalásáért felelős. A *View* réteg az adatok megfelelő megjelenítéséért és a felhasználók által végzett interakciók továbbításáért felelős. A *ViewModel* réteg tárolja az alkalmazás képernyőinek állapotát, és kezeli a felhasználói interakciókat.

A *Coordinator* minta lényege az, hogy a képernyők közötti navigáció elemeit egy külön réteg kezelje és ne a *View* vagy a *ViewModel* rétegnek legyen a felelőssége.

Az alkalmazásban jelen van az a *MVVM* architektúrában gyakran alkalmazott megoldás, miszerint a nagyobb műveleteket, például hálózati kommunikációt végző kódrészletek ki vannak szervezve egy *Service* elnevezésű rétegbe, ezzel tovább csökkentve a *ViewModel* réteg felelősségeit. Ezen felül a *Service* réteg felelős az alkalmazáshoz készített *Widget* komponenseknek szükséges adatok biztosításáért is.

Navigációs réteg

A navigációs rétegnek négy fő komponense van, melyek a képernyők megjelenítéséért, és létrehozásukért, konfigurációjukért felelősek. Ezek a komponensek a *Coordinator*, a *Navigator*, a *SceneFactory* objektumok és a *Screen* elnevezésű felsorolt típus.

A *Coordinator* felelőssége a navigáció egységbe foglalása az alkalmazás számára, aminek részeként ez a komponens hozza létre a *Navigator* és *SceneFactory* példányokat. Ezen felül a Coordinator feladata megjeleníteni a Navigator objektumot a képernyőn.

A *Navigator* egy felhasználói felület nélküli objektum, ami megvalósítja a *View* protokollt, így képes megjeleníteni bármelyik olyan képernyőt, amit definiáltunk a *Screen* típusban, valamint váltani is tud közöttük. Referenciája van az alkalmazás *SceneFactory* egyedére, amitől az egyes képernyőket és a mögöttük álló objektumokat kapja felkonfigurálva az adott *Screen* értéke alapján.

A Screen egy normális felsorolt típus, ami definiálja milyen képernyők lehetnek az alkalmazásban. Azt is itt tudjuk definiálni, hogy az egyes képernyőknek milyen paraméterekre van szükségük a megjelenéshez. Ezt a Swift nyelv asszociált értékein keresztül tudjuk megadni.

A képernyők és az állapotukat tároló *ViewModel* objektumok közös elnevezése *Scene*. Miután a *SceneFactory* konfigurálja és a *Navigator* megjeleníti őket, már képesek fogadni a felhasználó interakcióit, és kezelni azokat. Mint látható itt jelenik meg a hagyományos *MVVM* minta, amihez képest csak a *ViewModel* tartalmaz egy további függőséget, mégpedig az alkalmazás *Navigator* példányára. Erre azért van szükség, hogy a *ViewModel* tudja jelezni amikor elnavigáltak az adott képernyőről, és egy másikat kell megjeleníteni.

Service réteg

Ebben a rétegben az adatok perzisztenciájával kapcsolatos kódrészletek találhatóak meg, aminek megfelelően ebben a rétegben jelenik meg a *Core Data* használata. Mivel kifejezetten ellenjavallott létrehozni több olyan osztályt, amiknek a *Core Data* objektumaihoz közvetlen hozzáférése van, ezért egy kifejezetten nagy méretű *DataService* elnevezésű osztály van ebben a rétegben. Mivel azonban egy ilyen méretű és ennyi felelősséggel rendelkező osztályt nagyon nehéz lenne karban tartani, ezért az egyes entitások és bizonyos aspektusok mentén hét különálló osztály kiegészítésre lett darabolva.

A *DataService* az egész szolgáltatás konfigurálását végzi, illetve definiál alapvető műveleteket, amiket a kiegészítések használni tudnak.

A *DataService+Task, DataService+Category, DataService+Priority*, illetve a *DataService+TaskColumn* kiegészítések a nevüknek megfelelő entitások kezeléséért felelősek. Ezekhez tartozik az entitások létrehozása, törlése, illetve adataik megváltoztatása.

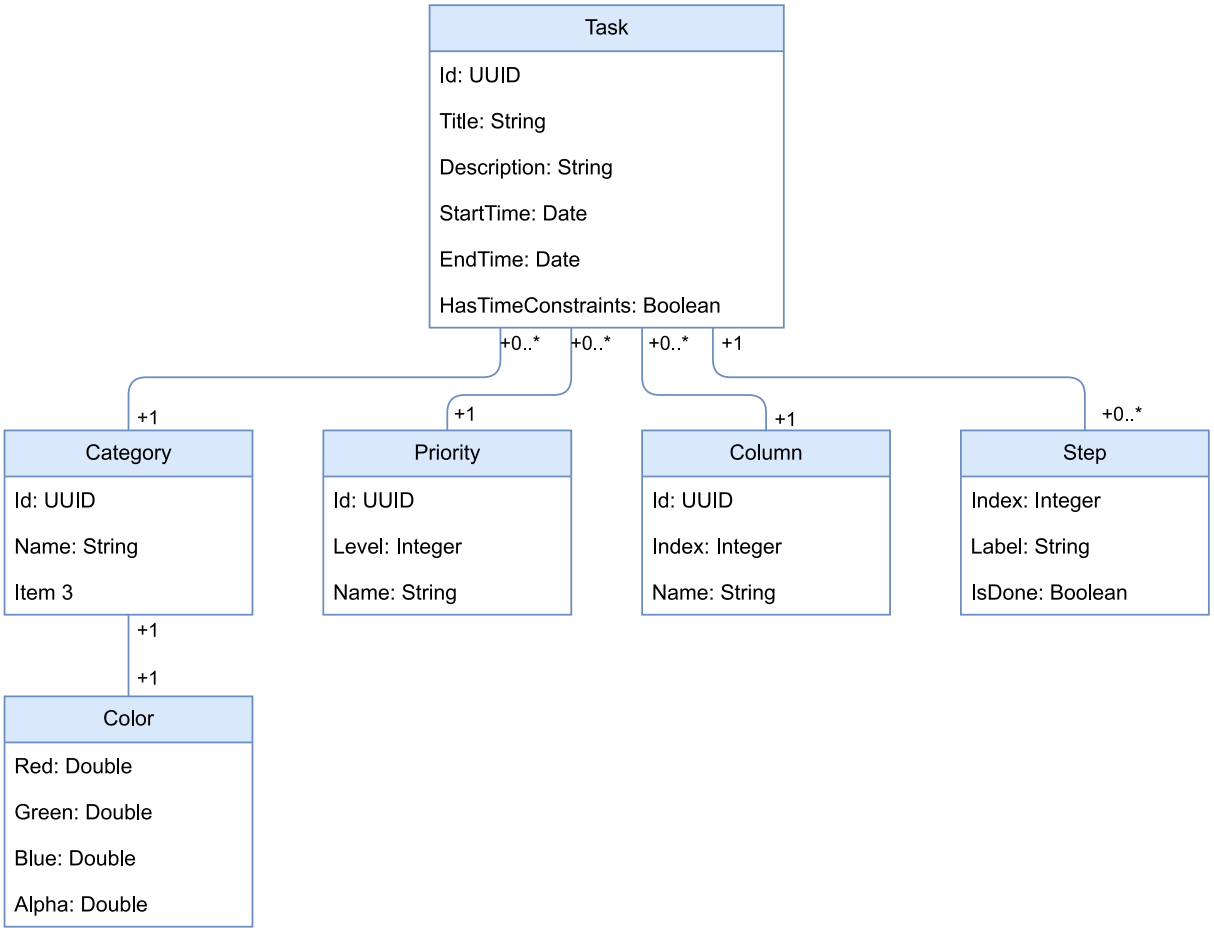
A *DataService+Fetch* kiegészítés az összes adat lekérdezést foglalja magába, a *DataService+Widget* pedig a *Widget* komponensek adatait szolgáltatja.

Model réteg

Ez a réteg, mint már említve volt az alkalmazásban szereplő logikailag összetartozó adatok egységbe foglalásáért és az ezeken az egységeken végzett műveletekért felelős. Az alkalmazásban hat ilyen egység fordul elő.

A *Core Data* használata érdekében ezek az egységek eredetileg egy a *Core Data* által használt adat modellben lettek létrehozva, amiből a keretrendszer számunkra nem látható osztályokat generál. Mivel ez a generált kód nem mindig teljesen megfelelő számunkra, és meglehetősen nehéz az alapvető használata, ebben a projektben a könnyebb kezelés érdekében az első generálás után ki lett kapcsolva a kódgenerálás, és ezután manuálisan lettek módosítva az entitások.

A *Task* entitás az alkalmazásban kezelhető feladatok adatait fogja egységbe. A *Priority*, a *Category* a *TaskColumn* és a *TaskStep* rendre a feladatokhoz tartozó prioritást, kategóriát, oszlopot, illetve a feladatokhoz adható lépéseket reprezentálják. A *ColorComponents* egy kategóriához tartozó színt reprezentál *RGBA* formátumban, amire azért van szükség mivel a *Core Data* nem képes a *SwiftUI* által használt *Color* objektumok elmentésére. Az entitások adatait és kapcsolataikat az alábbi diagram ábrázolja.



Mivel az alkalmazáson belül meglehetősen gyakran előfordul, hogy egyszerűbb könnyű struktúrákat használni ezeknek az adatoknak a használatára, mint a *Core Data* által létrehozott osztályokat, ezért minden entitáshoz készült kiegészítésként egy *Data Transfer Object* (*DTO*). Ezek az eredeti osztályoknak a könnyített struktúra verziója. Ezen felül minden osztályhoz készült egy *create* elnevezésű metódus, ami a megfelelő *DTO* objektumból készíti el a Core Data által felhasználható osztály egy példányát.

View réteg

A *View* réteg az alkalmazás helyes megjelenítéséért felelős. A *SwiftUI* lehetőségeinek köszönhetően rengeteg újra felhasználható önálló komponens készült az alkalmazáshoz, ezzel csökkentve a duplikált kódrészleteket.

Annak érdekében, hogy az olyan erőforrásokat, mint a képernyőn megjelenő statikus szövegek és betűtípusok típusbiztosan lehessen használni az alkalmazás fejlesztése során, a *SwiftGen* elnevezésű eszközt használja az alkalmazás.

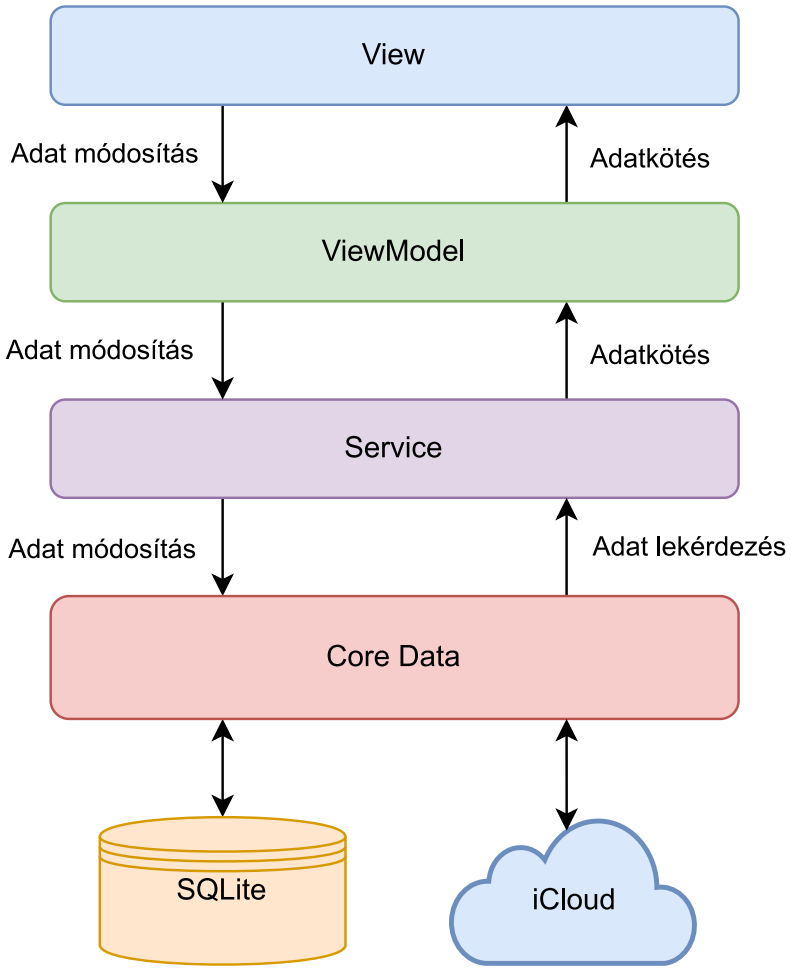
Ez az eszköz a megadott erősforrásokhoz tud generálni *Swift* kódot a beépített vagy az általunk megadott *stencil* típusú sablon fájlok alapján. Annak érdekében, hogy olvasható és lényegre törő kódot generáljon az eszköz az alkalmazáshoz készültek egyedi *stencil* sablon fájlok.

A View rétegből az aktuális napi haladást megjelenítő nézetet lehet kiemelni, mivel ez az alkalmazás fő képernyőjén helyezkedik el, és a felhasználó egy pillantás után tudja belőle, hogy hol tart az aznapi teendőivel. A nézet az alábbi képeken látható különböző állapotokban.

ViewModel réteg

A *ViewModel* az egyes képernyők állapotait tartalmazó réteg. A *ViewModel* objektumok a különböző *Service* osztályok által publikált adat forrásokra iratkoznak fel. A *ViewModel* réteg használja az *Apple* által újonnan kiadott *@Observable* makrót, ami a korábbi *ObservableObject* protokollt és az *@Published* tulajdonság csomagolót váltja fel. Használata rengeteg kód írását teszi feleslegessé, és az *Apple* nyilatkozata alapján még teljesítmény javulást is várhatunk tőle.

A *ViewModel* adatainak változására a *View* réteg iratkozik fel, amivel kialakul egy olyan lineáris adatfolyam, amiben a rétegek felelőssége nagyon szépen elválik egymástól, valamint a *Single source of truth* elv is megvalósul. A teljes vizualizált adatfolyamot az alábbi kép ábrázolja.



Widget komponensek

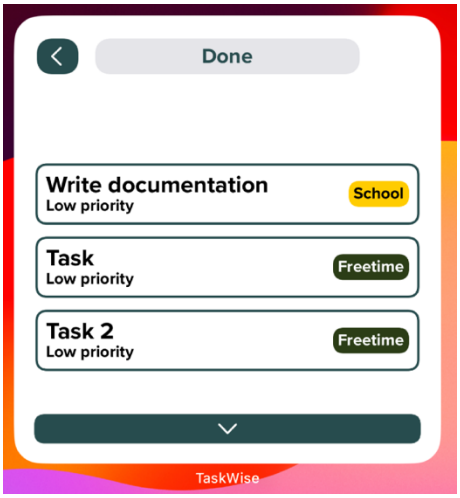
Mint már említve volt az alkalmazás mellé készült két *Widget* komponens is. Az *Apple* platformokon használt *Widget* komponenseknek három fő része van, amik a *View*, az *Entry* és az *TimelineProvider*.

Ezekből a *View* a megjelenítésért felelős, az *Entry* a megjelenített adatokat hívatott egységbe fogni, az *TimelineProvider* pedig az adatok a fő alkalmazástól való lekérdezéséért és azok *Entry* példánnyá való alakításáért felelős. Az alkalmazásban használt *TimelineProvider* leszármazottak a fő alkalmazás *Service* rétegével vannak összekötve *async-await* mintát használó metódusok segítségével.

Az kisebbik *Widget* a már említett főképernyőn elhelyezkedő haladás jelző nézet kicsinyített és egyszerűsített változata. Ezt a felhasználók lezárt képernyő mellet is megtudják tekinteni. Ez a *Widget* az alábbi képeken látható különböző állapotokban.

A *iOS* 17 verziótól kezdve lehet létrehozni interaktív *Widget* komponenseket is, amit az alkalmazás ki is használ. A nagyobb *Widget* az aktuális nap feladatait jeleníti meg a *Kanban* táblának megfelelő nézetben, amin a nyilak megnyomásával navigálhatunk az oszlopok között vagy adott oszlopban a feladatok között.

Ez a listás Widget az alkalmazásban való Deeplink technológiát is használ, vagyis a Widget komponensen való érintéskor az alkalmazást az érintésnek megfelelő elemnél nyitja meg. Ez a *Widget* az alábbi képen látható.



Továbbfejlesztési lehetőségek

Mint már említve volt az alkalmazás további platformokon is elfog készülni a jövőben. Ezen kívül számos felhasználói felületet és felhasználói élményt javító fejlesztés, illetve fenntarthatóságot érintő fejlesztés is tervben van az alkalmazás számára. A jelenleg tervezett további fejlesztéseket az alábbi felsorolás foglalja össze, ami az idő és a fejlesztés előrehaladtával természetesen bővülhet.

- Képernyőbe integrált szín választó komponens
- Jobb felhasználói élményt nyújtó naptár komponens
- A feladatok ismételhetőségére való lehetőség
- *Apple Watch* alkalmazás
- *Apple Watch Watchface Widget* komponens
- *Apple Watch SmartStack Widget* komponens
- Értesítések, jelzések az alkalmazásból
- *iPad* specifikus felhasználói felület
- *MacOS* specifikus felhasználói felület
- *iPad* és *MacOS* specifikus extra nagy méretű *Widget* komponens
- Egység tesztek
- Automatizált felület tesztek