

Rétegzett alkalmazás-fejlesztés

Klasszikus háromrétegű architektúrák

- Adat, logika és megjelenítési rétegből áll.
- A háromrétegű architektúrában a felső rétegek mindig a közbenső rétegeken keresztül érik el az alsó rétegeket.

Megjelenítési réteg

- Az alkalmazás legfelső rétege.
- Fő funkciója a rétegnek, hogy lefordítsa a feladatokat az alsóbb rétegek felé és hogy a visszajövő adatokat megjelenítse a felhasználó felé.

Logikai réteg

- Alkalmazás viselkedése található meg ebben a rétegben.
- Feldolgozza a felsőbb rétegtől jövő feladatokat és döntéseket hoz az üzleti logika alapján.
- Feladata még az adatmozgatás a két réteg között.

Adat réteg

- Tárolja az információkat.
- Feladata az adatbázissal és a fájl rendszerrel való kommunikáció megvalósítása és a megfelelő adatok kigyűjtése.
- Az információt amit kinyer, azt vissza is küldi a logikai rétegnek.

Fontos kritériumok

- **Legyen**
 - o Karbantartható (maintainability)
 - Minimalizálni tudja a változtatáskor, javításkor és új funkciók bevezetésekor igényelt erőforrásokat.
 - o Újrafelhasználható (reusability)
 - Pl.: DLL-t készítünk, amit feltudunk használni másik projekthez.
 - o Skálázható (scalability)
 - Növekvő igények kiszolgálásának problémamentes kezelését jelenti.

Konverterek

- Külön adatkonvertálási réteggént lehet használni, ami a **ViewModel** és a **View** között helyezkedik el.
- Szükség akkor lehet rá, ha valamilyen speciális formázást kell végrehajtani az adatokon, amiket a **ViewModel** nem biztosít.

Felhasználói felület alkalmazása a Model-View-ViewModel tervezési mintát használva.

Model-View-ViewModel – MVVM

- A **View** tud a **ViewModel**-ről.
- A **ViewModel** tud a **Model**-ről.
- A **Model** nem ismeri a **ViewModel**-t és a **ViewModel** nem tud a **View**-ről.

ViewModel

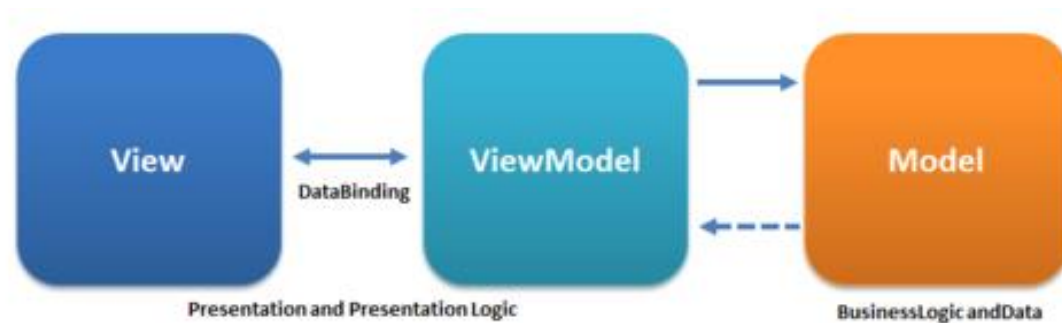
- Olyan tulajdonságokat és parancsokat implementál, amikhez a **View** adatkötést (Binding) végezhet.
- A változásértesítési eseményeken keresztül értesíti az állapotváltozások nézetét.

MVVM előnyök és hátrányok

- **Előnyök**
 - o Tesztelés lehetősége
 - o Bővíthetőség
 - o Karbantarthatóság
- **Hátrányok**
 - o A sok binding miatt bonyolult a debuggolás.
 - o Egyes esetekben nehéz megtervezni egy komplexebb ViewModel-t.

Felhasználói felület alkalmazása

1. Létrehozunk egy **Model**-t, ami tartalmazza az alkalmazás logikáját és az adatmodellt.
2. Létrehozunk egy **View**-t, ami megjeleníti az adatokat, amiket a **Model** tartalmaz.
 - a. Kezelheti a felhasználói interakciókat is.
3. Létrehozunk egy **ViewModel**-t, ami kapcsolódik a **Model**-hez és a **View**-hoz.
4. Összekötjük a **View**-t és a **ViewModel**-t adatkötés segítségével.
 - a. Az adatkötés teszi lehetővé a kommunikációt a két réteg között.



Felhasználói felület alkalmazása a Model-View-Controller tervezési mintát használva.

Model-View-Controller – MVC

- A **View** réteg frissíti a **Controller** réteget.
- A **Controller** réteg frissíti a **Model**-t.
- A **Model** réteg közvetlenül visszahat a **View** rétegre.
- Ismertebb MVC web keretrendszerek
 - o Ruby on Rails, Django, ASP.NET, Symphony

Controller

- A **Controller** felelős azért, hogy értelmezze a felhasználói interakciókat és módosítsa a **Model**-t ennek megfelelően.

MVC előnyök és hátrányok

- **Előnyök**
 - o Közös munkát lehetővé teszi a többi fejlesztő számára.
 - o Hibakeresés viszonylag könnyebb, mert több szint van.
 - o Tesztelhető minden komponens külön-külön.
- **Hátrányok**
 - o Nehezen újrahasználatóak a modellek.
 - o Fejlesztés során több technológia ismeretére lehet szükség.

Felhasználói felület alkalmazása

- Felhasználói felület elkészítésekor a felhasználói interakciókat és azok hatását a **Model**-re és a **Controller**-re kell szétválasztani.

MVC folyamat

1. Felhasználó csinál valamit a UI-on (**View**).
2. A **View** tájékoztatja a **Controller**-t a végrehajtott műveletről.
3. A **Controller** frissíti a **Model**-t.
4. A **Model** új adatokat szolgáltat.
5. **Controller** frissíti a **View**-t.

