

Teendők listája

<input type="checkbox"/> "Témebejelentő" header?	1
<input type="checkbox"/> Közérthető leírás alatt ezt érti, vagy ez már túl specifikus?	3
<input type="checkbox"/> Irodalomjegyzékben jó a citation?	4
<input type="checkbox"/> Telefonra ha meg lesz akkor az jöhet ide	5
<input type="checkbox"/> Content elrendezése nagy page break nélkül	6
<input type="checkbox"/> Ez a kép nem felel meg a margónak, az baj? Legalább átlátható	11
<input type="checkbox"/> Szekvencia diagram egy példa requestnél	21
<input type="checkbox"/> database uml	25
<input type="checkbox"/> iwacontext uml	25
<input type="checkbox"/> screenshot, hogy lefutott az összes teszt	30
<input type="checkbox"/> Kell egy összefoglaló	34
<input type="checkbox"/> Mobil optimalizált weblap?	34



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSinFORMATIKAI

TANSZÉK

Időpont foglaló webes alkalmazás

Témavezető:

Dr. Menyhárt László Gábor
adjunktus

Szerző:

Andi Péter
programtervező informatikus BSc

Budapest, 2021

"Témebejelentő"
header?

A szakdolgozat célja egy időpont foglaló webes alkalmazás létrehozása. Az alkalmazásban vállalkozók (pl.: edzők, magán tanárok) szabad időpontokat hirdethetnek, melyeket ügyfelek lefoglalhatnak. Ez az alkalmazás lehetővé teszi, hogy az egyéni-, kis- és középvállalkozók egyszerűen tudják egyeztetni ügyfeleikkel a munkáikat. Továbbá, a szoftver számon tartja a múltbeli foglaltidőpontokat, melyek így lekérdezhetők, így például a vállalkozó számlázás során egyszerűen meg tudja állapítani, hogy az adott hónapra hány alkalmat vett igénybe egy kliens.

A program két különálló részből áll, egy webes frontendből, amit Javascript-el és hasonló modern technológiákkal valósítok meg és egy backend API-ból melyet C# ASP.NET-ben kivitelezek. A frontend a backenddel http requestekkel kommunikál, a backend pedig egy adatbázist használ az adatok tárolására. A dolgozatomban rámutatok ennek az architektúrának az előnyeire és hátrányaira egy monolitikus MVC alapú webes alkalmazással szemben.

Tartalomjegyzék

1. Bevezetés	3
1.1. Motiváció	3
1.2. Megvalósítandó alkalmazás leírása	3
1.3. Kedvhozó az architektúrához	4
2. Felhasználói dokumentáció	5
2.1. Rendszerkövetelmények	5
2.2. Telepítés	5
2.2.1. Telepítés Dockerrel	5
2.2.2. Telepítés Docker nélkül	8
2.3. Funkciók leírása	8
2.4. Használat	10
2.4.1. Ügyfeleknek	10
2.4.2. Vállalkozóknak	10
3. Fejlesztői dokumentáció	11
3.1. Tervezés	11
3.1.1. Probléma leírása	11
3.1.2. Felhasználói esetek	11
3.1.3. REST API vs MVC architektúra	13
3.1.4. Clean Architecture Backenden	15
3.1.5. Adatbázis - Entity Framework	25
3.1.6. Funkcionális Frontend	25
3.2. Megvalósítás	28
3.2.1. Fejlesztési környezet	28
3.2.2. Fejlesztési döntések	28

3.2.3.	Fejlesztés közben felmerült problémák	28
3.3.	DevOps	28
3.3.1.	CI/CD	28
3.3.2.	Docker	29
3.4.	Tesztelés	30
3.4.1.	Unit tesztek	31
3.4.2.	Integrációs tesztek	32
3.4.3.	Manuális tesztek	33
4.	Összegzés	34
4.1.	További fejlesztői lehetőségek	34
	Irodalomjegyzék	35
	Ábrajegyzék	36

1. fejezet

Bevezetés

1.1. Motiváció

Szakdolgozatom célja egy időpont foglaló webes alkalmazás létrehozása. A motivációt unokatestvérem adta, aki személyi edzőként dolgozik. A munkájához elengedhetetlen, hogy időpontot egyeztessen ügyfeivel. Ezt üzenetváltásokkal tette, viszont, ha valaki lemondott egy időpontot, akkor utána arra a szabad időpontra más ügyfelet körülményes volt találni a platform miatt. Arról nem is beszélve, hogy hónap végén a számlakiállításához így nem volt egy konkrét listája, amit egyszerűen be tudott volna vinni a számlázó rendszerébe.

Én programozásban mindig is webes alkalmazások fejlesztését élveztem a legjobban, így amikor felvetette az ötletet, hogy lehetne egy időpont foglaló alkalmazást csinálni, le is csaptam rá. Ezzel nem csak az egész eddigi összes webes tudásomat tesztelhetem és fejleszthetem, hanem segíthetek is unokatestvéremnek, aki nagyon sokat segített rajtam is.

1.2. Megvalósítandó alkalmazás leírása

Közérthető leírás alatt ezt érti, vagy ez már túl specifikus?

Az alkalmazásnak két fő felhasználói köre van, a vállalkozók és az ügyfelek. Az ügyfelek tudnak a vállalkozók között böngészni, egyes vállalkozók időpontjait megnézni, szűrni és lefoglalni. Megnézhetik a lefoglalt időpontjaikat, melyeket lemondhatnak.

A vállalkozók létrehozhatnak kategóriákat (pl.: személyi edzés, angol korrepetálás), melynek megadhatnak árat, maximum résztvevő számot és hogy publikus-e az esemény, vagy csak megadott ügyfelek láthatják. Ez azért fontos, mert például unokatestvérem hétvégére csak családtagoknak vagy közeli ismerősöknek tartott edzéseket, az alkalmazásban ezért kell tudni szabályozni a láthatóságát a kategóriáknak. A vállalkozók időpont hirdetésnél választhatnak egy kategóriát és kezdő és vég időpontot, esetleg módosíthatják a résztvevő limitet. A kategóriákat, időpontokat és vállalkozói profilt lehet szerkeszteni. A vállalkozó le tudja kérdezni, kategóriákra és időtartamra szűrhetően, hogy egy ügyfél melyik kategóriából hány időpontot foglalt, ezek mennyibe kerültek összesen és generálhat egy pdf formátumú számlát.

1.3. Kedvhozó az architektúrához

A dolgozatomban nem csak a programra koncentráltam, hanem, hogy a mögöttes architektúra és kód minőségi és bővíthető legyen.

Irodalomjegyzékben jó a citation?

A backendem Uncle Bob Clean Architecture[1] elvén alapuló objektum orientált kód. Ezzel moduláris, elkülönített hatáskörű osztályokból áll a REST API-om, mellyel a Dependency Inversion Principle miatt egyszerűen és hatékonyan unit- és integrációs tesztelhető az alkalmazás.

A frontendemen React.js-t¹ használom Typescript-el, e miatt erős fordítási idejű garanciát kapok, hogy a kódom helyes. Továbbá a Typescript erős típusrendszere miatt a megjelenítés mögött funkcionális paradigmájú kód van. Ez azt jelenti, hogy nincs destruktív értékadás, összeg típusokkal és egy saját aszinkron Result monád típus miatt nem kivételeket kezelek, hanem típus szintű konstrukciókkal garantálom, hogy minden hiba megfelelően le legyen kezelve és programozói hibából ne lehessen inkonzisztens állapotban levő adathoz hozzáférni.

¹React.js - <https://reactjs.org/>

2. fejezet

Felhasználói dokumentáció

2.1. Rendszerkövetelmények

Szerver oldalon: Windows 10 vagy Linux operációs rendszer, 2GB RAM, legalább 5GB tárhely az adatoknak, port nyitási lehetőség.

Telefonra ha
meg lesz akkor
az jöhet ide

Kliens oldalon: Legalább Chrome 90, Firefox 88, Edge 90, ezek mind asztali számítógépen, legalább 1280x720-as képernyő felbontással.

2.2. Telepítés

Az alkalmazást legegyszerűbben Docker² segítségével lehet telepíteni. Van lehetőség Docker nélkül is, viszont az több konfigurációval és üzemeltetéssel jár.

2.2.1. Telepítés Dockerrel

A Dockeres telepítéshez szükséges a Docker³ és Docker Compose⁴ telepítése.

A fő mappában megtalálható *docker-compse.yml* fájlban találhatók meg a konténerek konfigurációi. Három konténerből áll, egy MariaDB ⁵ adatbázisból, a backend REST API-ból és a frontendből. A yml fájlban a következő konfigurációs lehetőségek a 2.1 táblázatban találhatók.

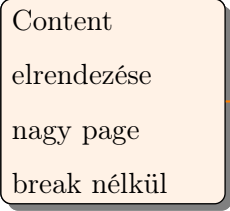
²<https://www.docker.com/>

³<https://www.docker.com/get-started>

⁴<https://docs.docker.com/compose/install/>

⁵<https://mariadb.org/>

Az alap beállításokkal a `http://localhost:8100`-on érhető el az alkalmazás. HTTPS-t, tűzfalat érdemes bekonfigurálni egy Reverse Proxy[2]-val, például Nginx-el.



Konfigurációs változó	Alap érték	Megjegyzés
db		
MYSQL_ROOT_PASSWORD	kebab	MariaDB adatbázis root felhasználójának jelszava
volumes	./db_data:/var/lib/mysql	MariaDB adatbázis perzisztens tárolása a lokális db_data mappában
ports	3306:3306	A konténer 3306-as portja a hoston az 3306-es portra forwardolása
backend		
IWA_CorsAllowUrls	http://127.0.0.1:8100	Vesszővel elválasztva az engedélyezett publikus frontend url-ek. Pl.: https://andipeter.me
IWA_MYSQL_HOST	db	MariaDB adatbázis host neve
IWA_MYSQL_PORT	3306	MariaDB adatbázis portja
IWA_MYSQL_DB	iwa	MariaDB adatbázison belül használandó adatbázis
IWA_MYSQL_USER	root	MariaDB adatbázis felhasználója
IWA_MYSQL_PASS	kebab	MariaDB adatbázis felhasználójának jelszava
volumes	./avatars:/app/AvatarData	A profilképek perzisztens tárolása a lokális avatars mappában
ports	5000:80	A konténer 80-as portja a hoston az 5000-es portra forwardolása
frontend		
API_URL	http://127.0.0.1:5000	A backend publikus elérési url-je. Pl.: https://andipeter.me/api/
ports	8100:80	A konténer 80-as portja a hoston az 8100-es portra forwardolása

2.1. táblázat. Konfigurációs változók beállításai

Az alkalmazást ez után a *docker-compose up* paranccsal indíthatjuk el. Első futtatásra ez eltarthat pár percig, mert a Dockernek le kell töltenie a megfelelő alap konténereket az internetről és utána létre kell hozni ezeket a konténereket a forráskódból.

További docker-compose parancsok a docker-compose dokumentációjábanban⁶ található.

2.2.2. Telepítés Docker nélkül

Az alkalmazás futtatásához szükség lesz egy MariaDB szerverre és azon belül egy *iwa* nevű adatbázisra. Az alkalmazás Linux és Windows rendszereken is futhat, ehhez a megfelelő backend fájl futtatása szükséges.

A backend futtatása parancssorból az *IWA_Backend.API* futtatható fájlal lehet. A konfigurációja az *appsettings.json* fájlban található, kitöltése a 2.1 táblázat alapján történik. A backend így a host 80-as portján fog futni. Ezt a `-urls=http://localhost:5001/` konzoli argumentummal lehet megváltoztatni, ebben az esetben az 5001-es porton futna az alkalmazás.

A frontend statikus HTML, JS és CSS fájlokból áll, ezt például Apache⁷ vagy Nginx⁸ szerverekkel, vagy más hasonló webhost szolgáltatásokkal lehet kitelepíteni. A frontend konfigurációja a mappájában a *config.js* fájlban történik, kitöltési útmutató a 2.1 táblázatban található.

2.3. Funkciók leírása

Az alkalmazásban lehet regisztrálni ügyfél vagy vállalkozóként. Az oldalt lehet bejelentkezve vagy bejelentkezés nélkül böngészni.

Kategóriák

A vállalkozók létrehozhatnak kategóriákat. A kategória effektíve egy időpont típus, például személyi edzés. A kategóriák megegyeszerűsítik az új időpontok

⁶<https://docs.docker.com/compose/reference/>

⁷<https://httpd.apache.org/>

⁸<https://www.nginx.com/>

létrehozását, mert a különböző időpontok közti azonos adatokat enkapszulálják, az időpontnál így csak az időpont specifikus adatokat kell megadni. Egy kategóriának lehet egy leírása, ára, ajánlott max résztvevő száma és láthatósága. Az ajánlott max résztvevőszám azt jelenti, hogy egy új időpont létrehozásánál alapból ez a szám lesz a max résztvevők mezőben, viszont ettől el lehet térni időpontról időpontra, például egy csoportos edzésre a Margit szigeten többen jöhetnek mint a Hősök tereire.

Egy kategória láthatósága a következőt jelenti. Ha nyílt egy esemény, akkor bárki láthatja, bárki jelentkezhet rá. Ha egy esemény nem nyílt, akkor csak azok az emberek láthatják és jelentkezhetnek rá, akik engedélyezett résztvevőként fel lettek véve a kategóriára. Ennek az a szerepe, hogy például egy Családi edzésre hétvégén ne tudjon mindenki jelentkezni, csak az előre felvett családtagok. Vagy például egy kedvezményes árazású időpontnak más lehet a kategóriája.

Kategóriákat nem lehet törölni, abból az okból, hogy akkor az összes hozzá tartozó időpont is törlődne, ezzel múltbeli időpontok adatai elvesznének.

Időpontok

Új időpont hirdetésénél az időponthoz kell választani egy kategóriát, kezdő és vég időpontot. Opcionálisan meg lehet változtatni a max résztvevő számot. Van lehetőség alapból felvenni ügyfeleket az időpontra, például ha a vállalkozó már előre leegyeztetett egy időpontot de még nem írta ki az alkalmazáson, akkor az ügyfélnek nem kell bejelentkeznie és lefoglalni az időpontot.

Időpont szerkesztésnél a vállalkozónak van lehetősége változtatni egy időpont összes értékén. A kategórián például azért változtathat, mert Angol óra helyett Német órát tartott az ügyfélnek, vagy Páros edzés helyett Személyi Edzést, mert közbe jött valami. Lehet az időpontra jelentkezett felhasználókat is módosítani, lejelentkeztetni és felvenni ügyfeleket, akár az időpont után is. Például valaki lemondott egy edzést és beugrott helyette valaki más, a nap végén pedig így helyesen tudja adminisztrálni ezt a vállalkozó.

Számlázás

A számlázás funkciónál a vállalkozók adott felhasználók lefoglalt időpontjaiból tudnak számlát generálni egy időszakra, például Április 1 és 30 között. Ez

a számla jelenlegi formájában nem minősül NAV által elfogadott számlának, viszont a vállalkozónak nagyon jó segítség, hogy a saját számlázó szoftverébe (pl.: számlázz.hu) miről írjon számlát. Az alkalmazásba azért se került online fizetési lehetőség vagy számlázz.hu integráció, mert a valóságban az időpontokon kívül mást is tartalmazni szokott a számla (pl.: edzőterem bérlet, edzésterv) és ezekre akkor ezen felül egy külön számlát kéne kiállítania a vállalkozónak.

2.4. Használat

Regisztráció, bejelentkezés

2.4.1. Ügyfeleknek

Felhasználói leírás, hogy lehet böngészni a vállalkozókat, hogy lehet szűrni az időpontokat, hogy lehet lefoglalni időpontot, foglalt időpontokat hol lehet megnézni, hogy lehet lemondani időpontot, saját profilt megnézni és szerkeszteni

2.4.2. Vállalkozóknak

vállalkozói oldal, kategória létrehozás, szerkesztés, megtekintés

Időpont létrehozás, megtekintés, szerkesztés, törlés

profil szerkesztése, profilkép frissítése

számlázás, szűrés, számla letöltése

3. fejezet

Fejlesztői dokumentáció

3.1. Tervezés

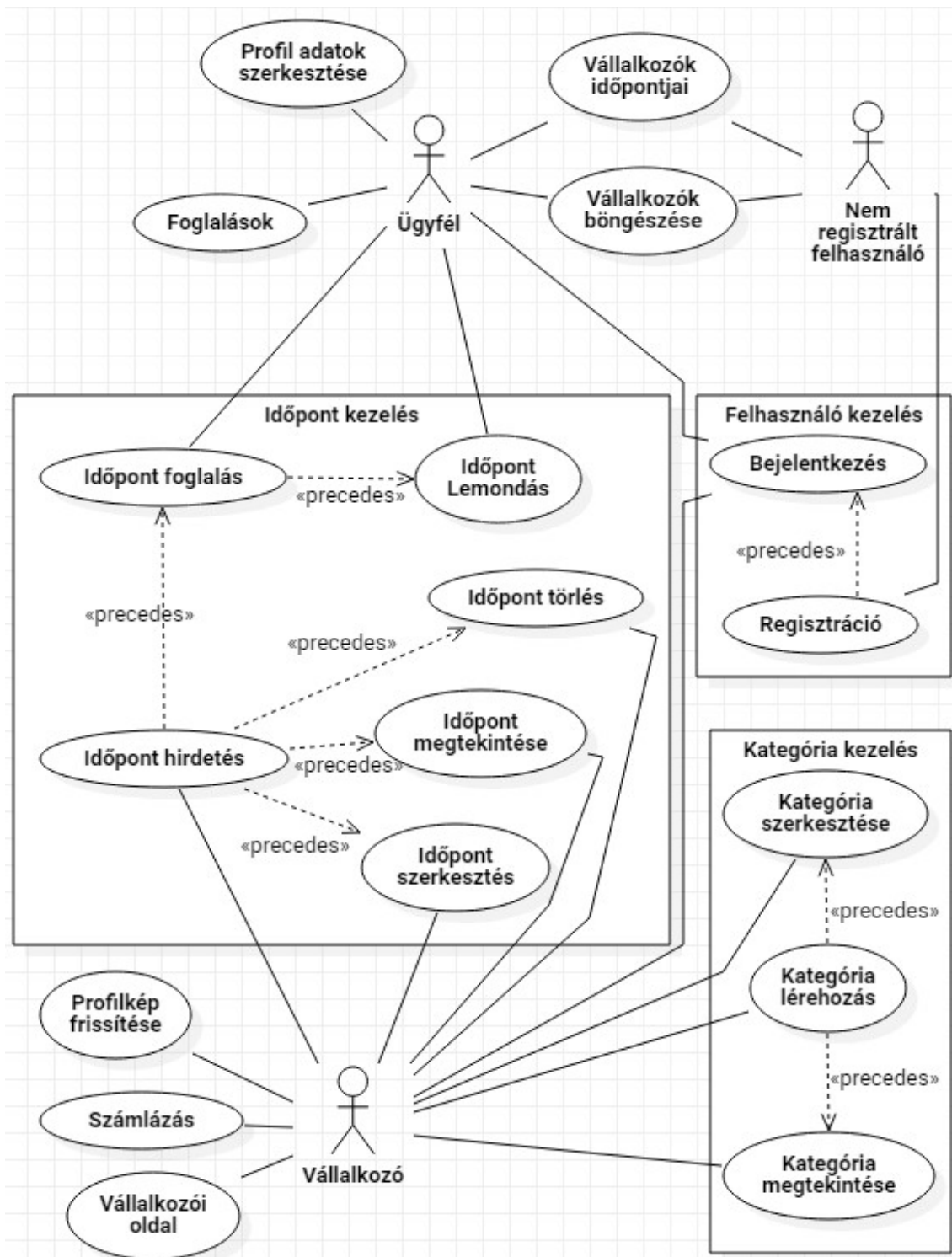
3.1.1. Probléma leírása

Implementáció szemponjából fontosabb elemeket kiemelni felsorolásba? A felhasználói résznél már nagyjából a funkciók specifikálva lettek

3.1.2. Felhasználói esetek

Ez a kép nem felel meg a margónak, az baj? Legalább átlátható

A felhasználói esetek a következőképpen néznek ki. A vállalkozó egyben ügyfél is (hogy esetleg más vállalkozók időpontjaira tudjon jelentkezni), az ügyfelek összes funkcióját tudják használni, ezt nem jelöltem a diagrammban, hogy átlátható maradjon.



3.1. ábra. Felhasználói esetek

	Leírás	Kód
GIVEN	Nincs bejelentkezve	asd
WHEN	Bejelentkezéshez kötött oldalt nyitna meg	
THEN	Visszairányítódik a főoldalra	
GIVEN	asd	asd
WHEN	asd	
THEN	asd	
GIVEN	asd	asd
WHEN	asd	
THEN	asd	
GIVEN	asd	asd
WHEN	asd	
THEN	asd	
GIVEN	asd	asd
WHEN	asd	
THEN	asd	
GIVEN	asd	asd
WHEN	asd	
THEN	asd	

3.1.3. REST API vs MVC architektúra

Webes alkalmazások körében régebben elterjedt volt a Modell-View-Controller architektúra (röviden MVC). Röviden ez azt jelenti, hogy a felhasználó akcióira a Controller réteg eldönti, hogy az állapotot (Modellt) hogy kell frissíteni, ez után pedig egy nézetet (View-t) ad vissza a felhasználónak. A gyakorlatban ez szerver oldali renderelést jelent, például a felhasználó elküld egy űrlapot a szervernek, az feldolgozza és egy szerver által renderelt HTML fájlt küld vissza a felhasználó böngészőjének.

Ennek a megközelítésnek vannak előnyei, többek között, hogy az alkalmazásnak egy kódbázisa van, egyszerűbb egy új funkciót implementálni, kevesebb technológiát is elég ismerni. Hátránya viszont, hogy dinamikus felhasználói felületet nehéz benne építeni, más alkalmazásokba, például mobil alkalmazásba, nem lehet integrálni.

Ezekre nyújt megoldást, ha a logikát egy REST API (Representational state transfer, Application Programming Interface) valósítja meg backenden, a megjelenítésért pedig egy másik program felel frontend-en. A REST egy interfész leíró struktúra, legtöbb esetben HTTP protokoll alapú kommunikációt ír le, melyben JSON formátumú adattal lehet kommunikálni.

Mivel az API-t így programatikusan tudjuk elérni, ezért más alkalmazásokkal egyszerűen képes kommunikálni. Így lehet például web-ről, telefonos- vagy asztali alkalmazásból elérni a biznisz logikát, ezért csak a megjelenítést kell variálni platformok között.

A REST API állapot mentes, ami azt jelenti, hogy a szerver nem függ valamilyen kontextustól, csak a kérésben szereplő adattal elég dolgoznia. Ez lehetővé teszi, hogy a backend több szerveren horizontálisan egy load balancer (terheléelosztó) segítségével legyen skálázva. Egy ilyen rendszerben az egymást követő kérések akár különböző backend példányokhoz futhatnak be, az alkalmazás ugyan olyan pontosan működik.

A programozható felület lehetővé teszi, hogy a szerver ne teljes oldalakat küldjön vissza válasznak, hanem csak adatot. Ez a rugalmasság lehetővé teszi, hogy a frontend dinamikus legyen. Például az én alkalmazásomban egy új időpont hirdetésénél a böngésző tesz egy kérést a szerver felé, ami visszaadja a létrejött időpont adatát és a frontend azt az egy időpontot beilleszti a jelenleg megjelenített időpontok közé, nem kell a teljes oldalt az összes időponttal újra tölteni.

A hátránya ennek az architektúrának, hogy a backend és frontend teljesen különálló, akár más programozási nyelvekben vannak írva, más eszközökkel kell fejleszteni őket, így nagyobb a projekt komplexitása. Vállalati környezetben ez előny lehet, mert külön csapatokra szét lehet osztani a frontend és backend fejlesztést. További nehézség lehet, hogy az backendet és a frontendet össze kell kötni, ez az integráció nem olyan triviális, mint egy monolitikus MVC alkalmazásban, ahol egyből a modell adatát bele lehet renderelni HTML tagek közé. Továbbá, mivel

az API így egy külön álló alkalmazás, amit bárhol lehet lekérdezni, fontos biztonsági lépésekkel le kell védeni, hogy jogosulatlan adathoz ne lehessen hozzáférni, szennyezett adattal ne lehessen elrontani az alkalmazást.

3.1.4. Clean Architecture Backenden

Clean Architecture

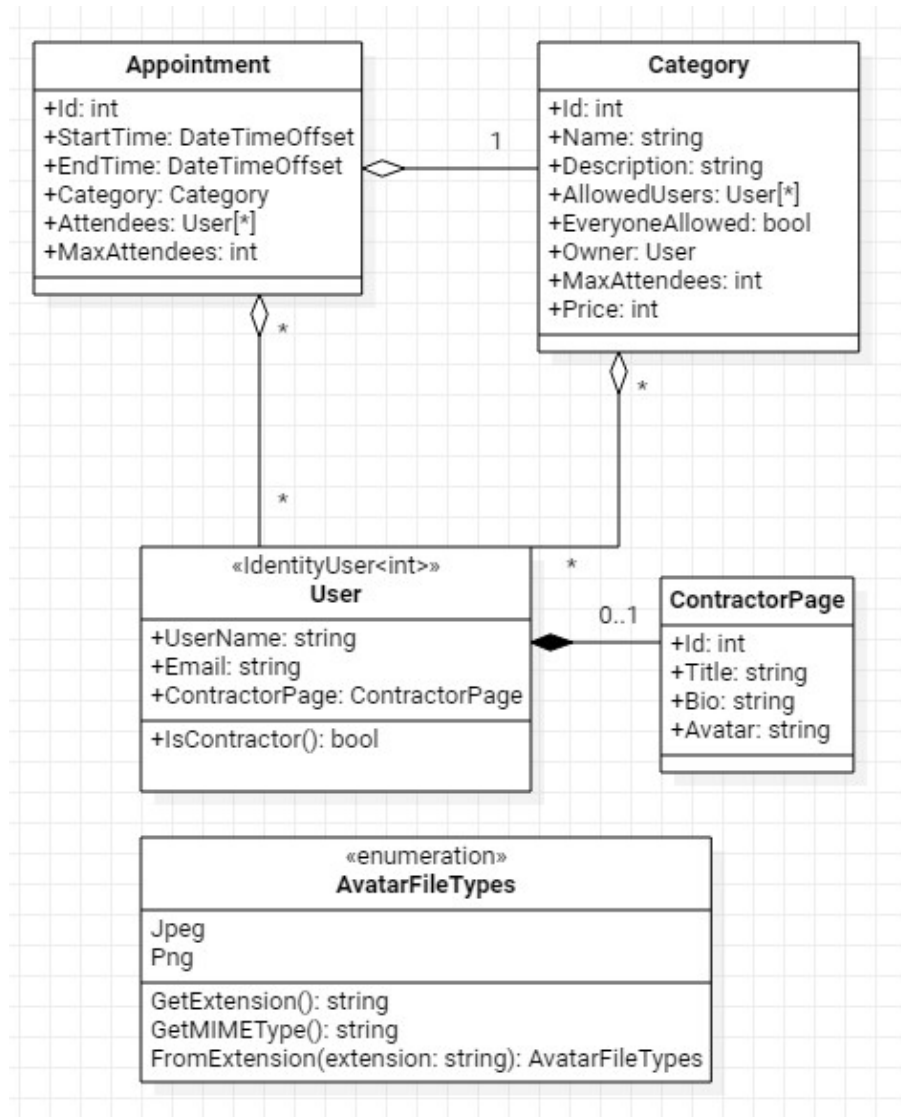
Uncle Bob Clean Architecture[1]-jének a lényege, hogy az alkalmazás különböző rétegei minél kevésbé függjenek egymástól. Ő négy réteget definiál: entitások, felhasználói esetek, kontrollerek és külső szolgáltatások. Az én alkalmazásomban az entitások az alkalmazás belső reprezentációs adattagjai. A felhasználói esetek a logika osztályokban vannak, minden egyes függvény a logika osztályban egy felhasználói esetet fed le. A kontrollerek az ASP.NET-es kontrollerek. A külső szolgáltatások pedig az adatbázis kezeléssel foglalkozó repository-k és majd a jövőben az email küldő szolgáltatás.

A különböző rétegek csak egymás interfészeitől, nem implementációitól függenek. Így például a logikában nincsenek SQL lekérdezések, a controller nem tud fájlokat megnyitni. Ezt a függőségi befecskenkezés elvével (Dependency inversion principle) valósítom meg. Ez azt jelenti, hogy egy osztály ha valami más rétegre hivatkozna, pl.: logika egy repository-ra, akkor a logika osztály konstruktora csak a repository egy interfészét várja, mert a logika szempontjából csak az a lényeg, hogy le tudjon kérdezni adatot, az nem, hogy az konkrétan hogy történik.

Ez lehetővé teszi, hogy a különböző szolgáltatásokat egyszerűen lehessen refaktorálni. Mivel minden interfészekkel dolgozik, ezért ha az adatbázis elérést Entity Framework-ről lecserélném általam írt SQL lekérdezésekre, akkor csak a repository implementációt kell megváltoztatnom és betartani az interfészt és ugyan úgy működik az alkalmazás.

Entitások

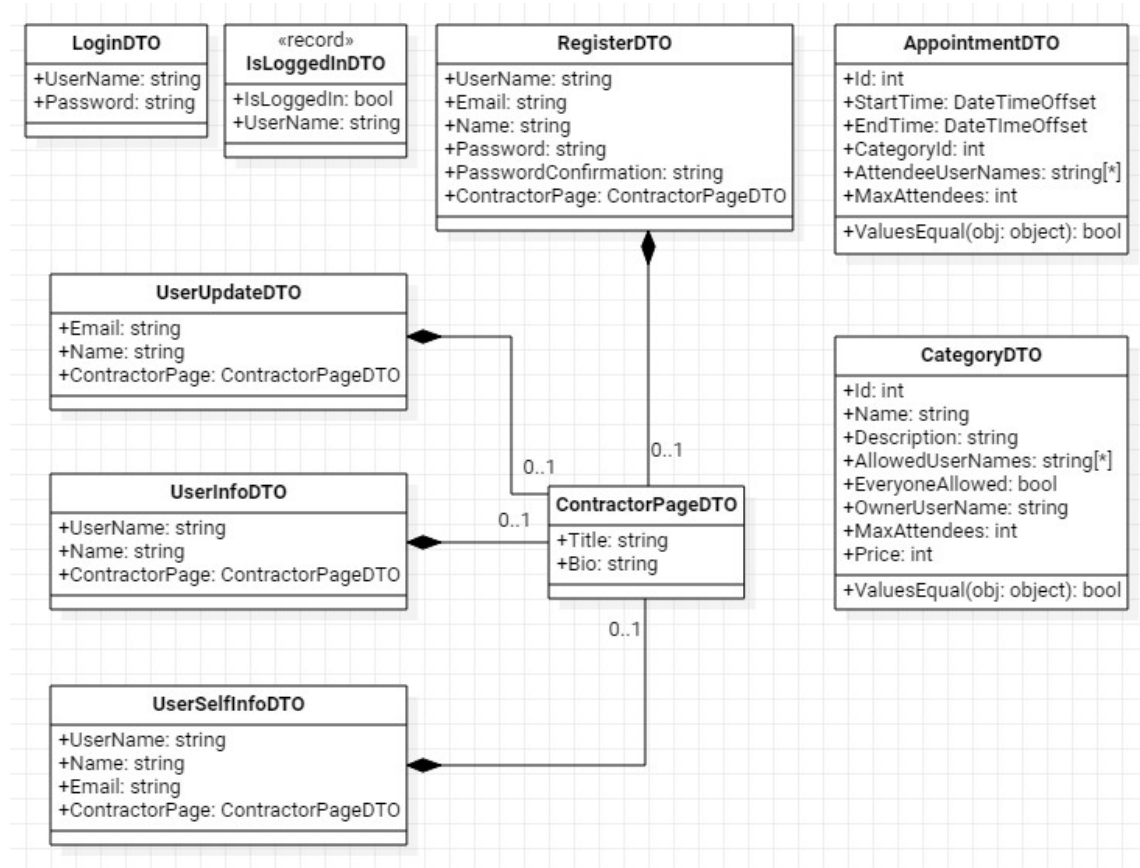
Az entitásaim a következőféleképpen néznek ki:



3.2. ábra. Entitások UML diagrammja

DTOk

Az entitások mellett DTO⁹-kat is használtam, a REST API ezekkel az adatszerkezetekkel kommunikál.



3.3. ábra. DTO-k UML diagrammja

⁹Data Transfer Object - Adatátviteli objektum

Logika

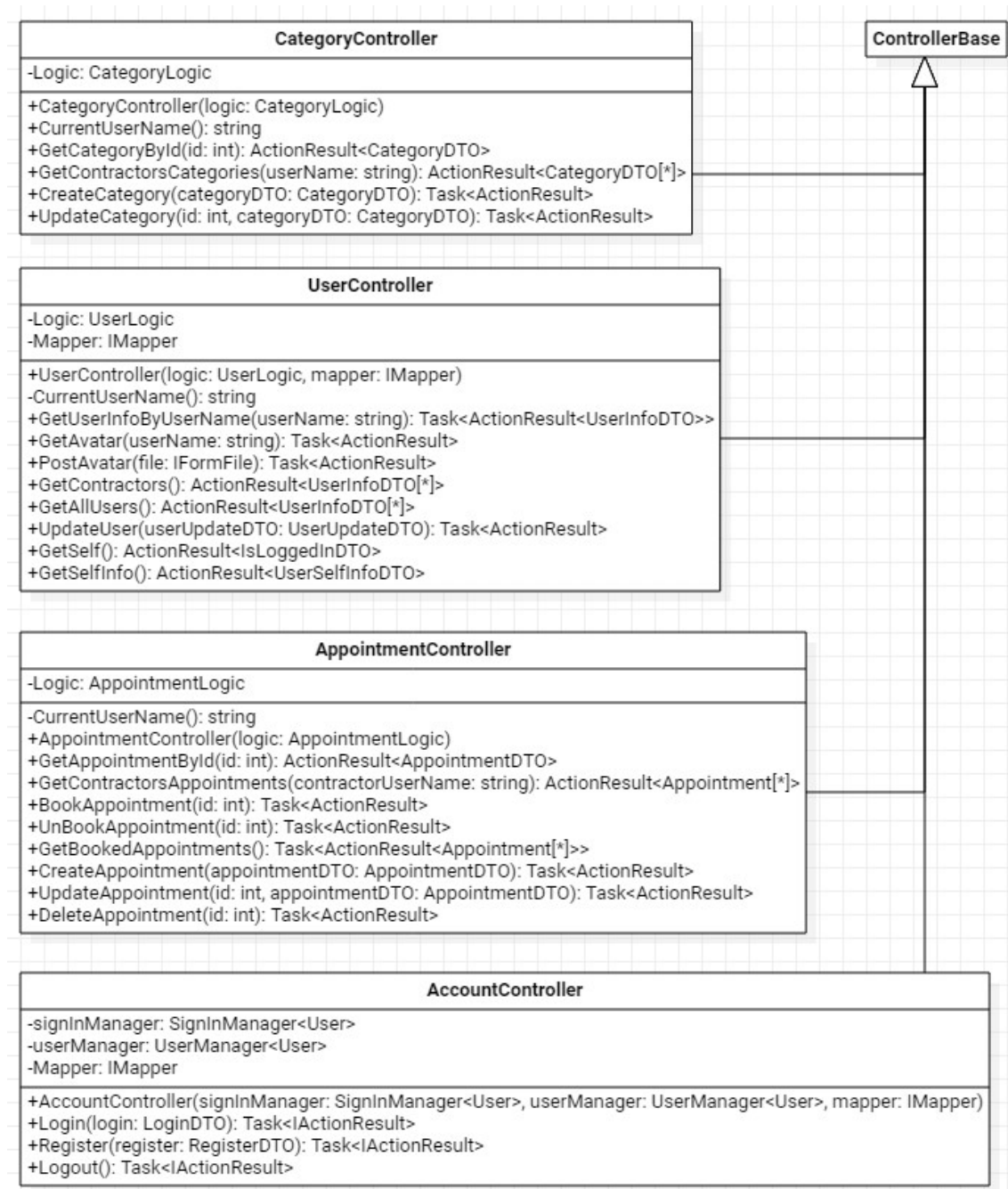
A logikát megvalósító osztályaimat entitásonként különítettem el, azaz az egy fajta entitással dolgozó felhasználói esetek tipikusan egy osztályba kerültek. A logika osztályban lehet először látni a függőségi befecskendezés elvét. A logika osztályok csak a releváns repository-k interfészeit kapják meg.



3.4. ábra. Logika osztályok UML diagrammja

Kontrollerek

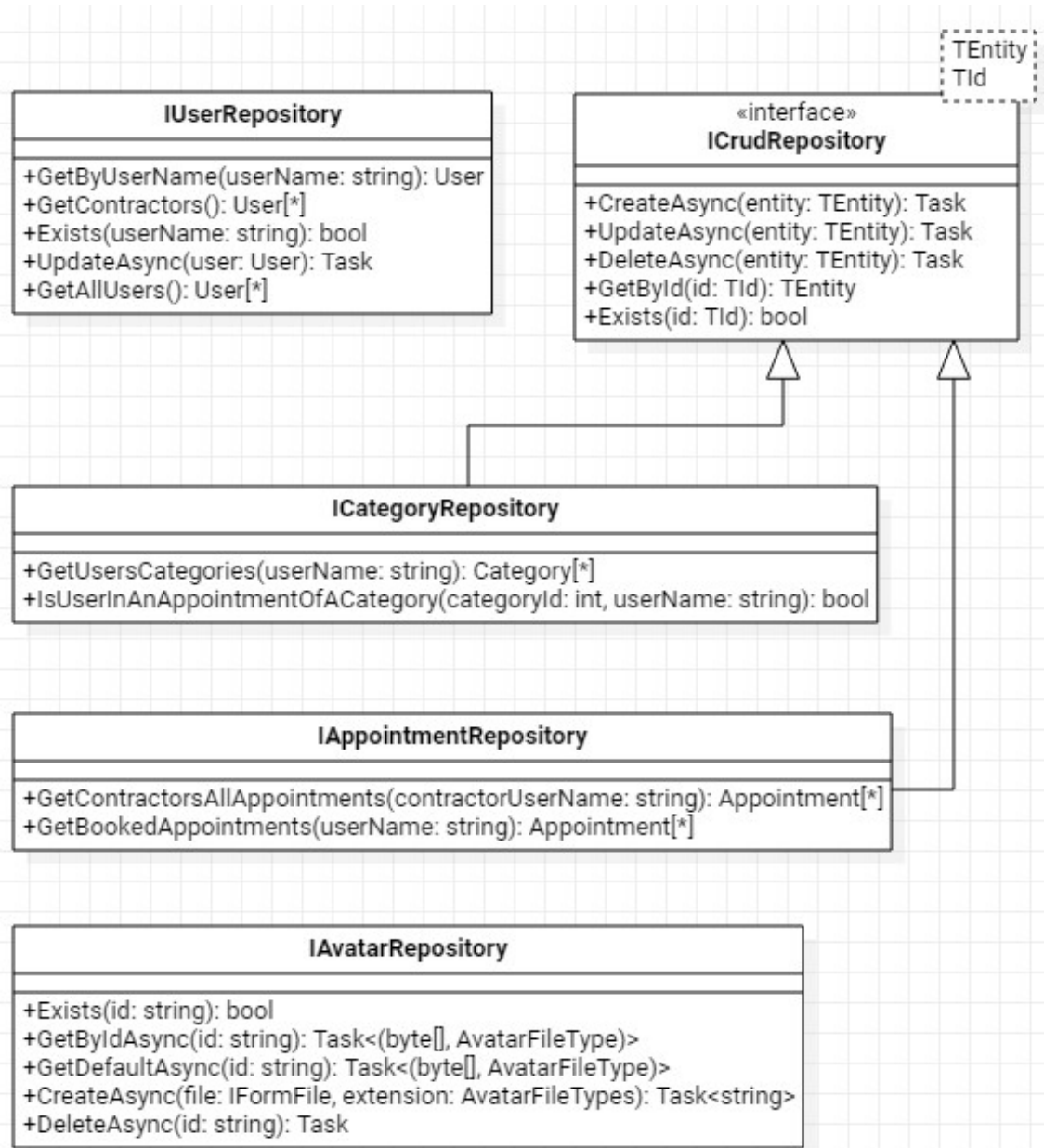
A kontrollerek ugyan azokat a felhasználói eseteket fedik le, mint a logika osztályok. A különbség, hogy a bejövő HTTP kéréseket kezelik, alakítják át a logikának megfelelő adatra, utána meghívják a logika egy függvényét, majd a visszakapott belső reprezentációs adatot mappelik DTO-vá.



3.5. ábra. Controller-ek UML diagrammja

Repository-k

Az adat elérő repository interfészek az alábbi 3.6 ábrán láthatók. A repository megvalósítások nem képezik részét a diagrammnak, mert nem térnek el érdemben a megvalósított interfészekről, csak megvalósítják azt.

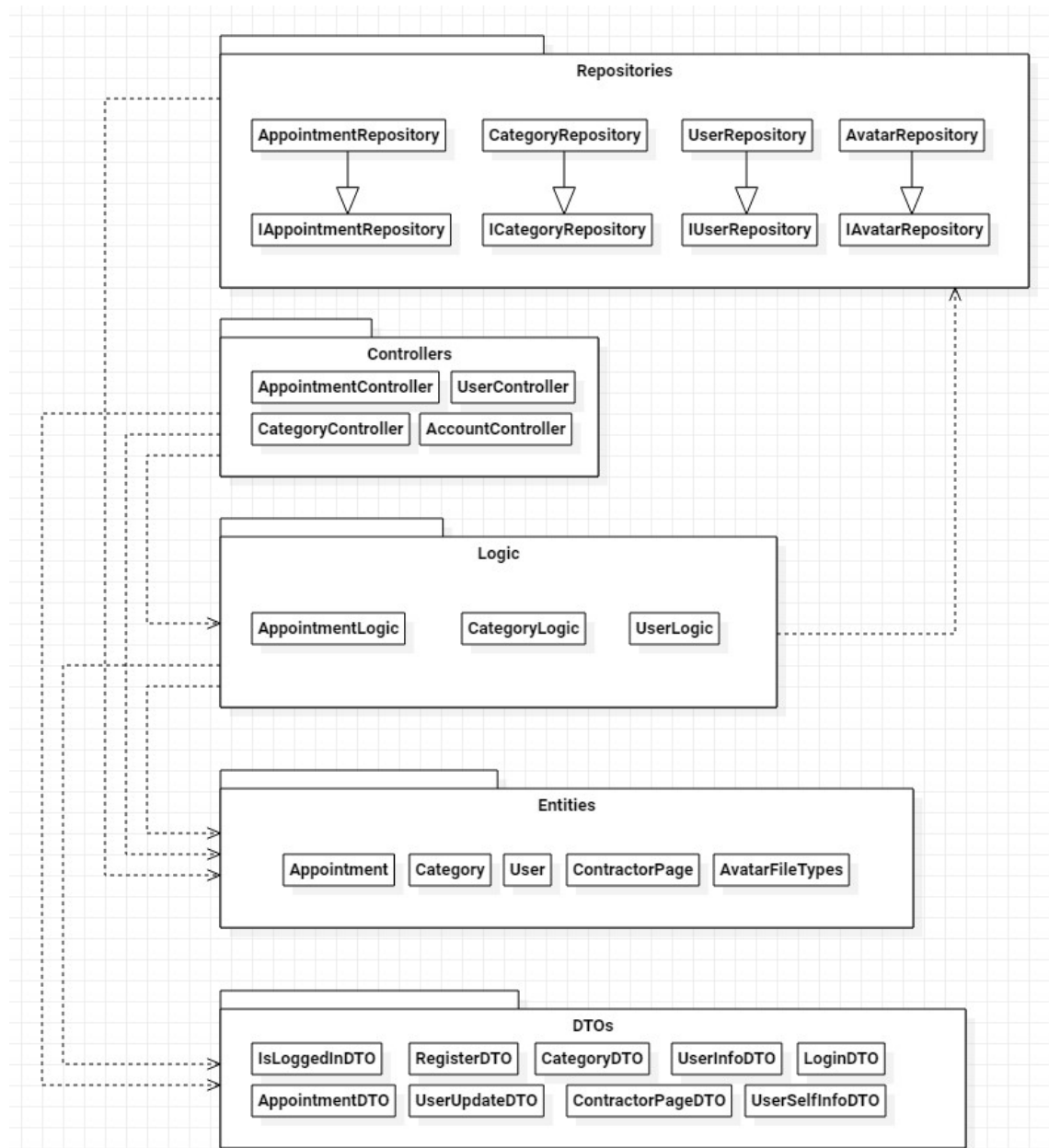


3.6. ábra. Repository interfészek UML diagrammja

Program komponensek interakciója

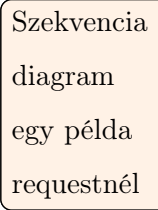
Mint látható, az entitásokon kívül az osztályok nem tartalmaznak állapot tárolásra szolgáló adattagokat. Ez a REST API állapotfüggetlensége miatt van. Így például a logika meg repository osztályok csak azért vannak osztályba szervezve, hogy ugyan azokat a befecskendezett függőségeket használják, hogy ne kelljen minden metódusuknál paraméterként megadni őket. Ettől funkcionális érzetű a kód, viszont ez a unit tesztelésnél hasznos, amit a 3.4.1 részben tárgyalok.

A program komponensei a következő módon függenek egymástól:



3.7. ábra. Program komponenseit összeítő UML diagramm

Szekvencia
diagram
egy példa
requestnél



REST API végpontok

A REST API végpontok a következőképpen néznek ki:

Account	
POST	/Account/Login
POST	/Account/Register
POST	/Account/Logout
Appointment	
GET	/Appointment/{id}
PUT	/Appointment/{id}
DELETE	/Appointment/{id}
GET	/Appointment/Contractor {contractorUserName}
POST	/Appointment/{id}/Book
POST	/Appointment/{id}/UnBook
GET	/Appointment/Booked
POST	/Appointment

3.8. ábra. REST API végpontok

Category		▼
GET	/Category/{id}	
PUT	/Category/{id}	
DELETE	/Category/{id}	
GET	/Category/Contractor/{userName}	
POST	/Category	
User		▼
GET	/User/Info/{userName}	
GET	/User/Avatar/{userName}	
POST	/User/Avatar	
GET	/User/Contractors	
GET	/User/All	
PUT	/User	
GET	/User/Self	
GET	/User/SelfInfo	

3.9. ábra. REST API végpontok folytatása

3.1.5. Adatbázis - Entity Framework

Az Entity Framework¹⁰ (továbbiakban EF) egy Microsoft által fejlesztett könyvtár a .NET keretrendszerrel, egy ORM¹¹ keretrendszer, mely C# osztályokat fordít adatbázis elemekre és vissza. Code first módon elég a C# osztályokat definiálni és az EF létrehozza az SQL táblákat és kapcsolatokat, a lekérdezéseket CX-ban LINQ¹² segítségével lehet végezni.

Azonban EF-el sem triviális az adatbázis kezelés, több a többhöz kapcsolatokat (pl.: egy ügyfél több időpontra is jelentkezhet és egy időpontra több ügyfél is jelentkezhet) elég sok manuális konfigurálással kell létrehozni, erről bővebben a megvalósítás 3.2.3 részében írok.

Az adatbázis terve körülbelül egyezik az előző részben definiált entitásokkal, a következőképpen néz ki:

database uml

Az EF-ben a DbContext osztály biztosítja az adat elérést és definíciót. A DbContext virtual DbSet propertyjei lesznek azok az értékek, amiket az EF használni fog. Az OnModelCreating metódusban lehet testre szabni, hogy hogyan is generálja le az objektumok között a kapcsolatokat az EF. A backend DbContext-je az IWaContext a következőképpen néz ki:

iwacontext uml

3.1.6. Funkcionális Frontend

React és Typescript

Az alkalmazás frontendjét React.js¹³ keretrendszerrel Typescriptben valósítom meg. A Typescript kibővíti a Javascript nyelvet és típusellenőrzést biztosít fordítási időben. A Typescript fájlokat a fordító Javascript fájlokká fordítja, közben szintaktikai és szemantikai elemzéseket hajt végre.

¹⁰<https://docs.microsoft.com/en-us/ef/>

¹¹Object-Relational Mapping - Objektum-Reláció fordítás

¹²<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/>

[linq/](#)

¹³<https://reactjs.org/>

A React keretrendszer alapjára, hogy komponensekből épül fel a felhasználói felület. Ezeket a komponenseket lehet újra felhasználhatóra tervezni, így kód duplikációt elkerülni. Meg van a lehetőség őket egymásba ágyazni, egymás között a komponenseknek kommunikálni, így komplex rendszereket lehet építeni relatíve kis építőelemekből.

A React a nevét a reaktivitásból kapta, a lényege, hogy dinamikusán változó felhasználói felületeket lehessen létrehozni, elkerülve a régi statikus, bármilyen változtatás után újratöltést igénylő oldalakat. Ezt egy virtuális DOM-al éri el, nem a böngészőre hagyja az oldal szerkezet kezelését, hanem javascriptben kódban csinálja. Ennek az előnye, ha bármilyen érték változik és frissíteni kell a DOM elemeket, akkor a React el tudja dönteni, hogy konkrétan melyik elemeket kell újrarajzolni és csak azokat változtatja meg a böngésző DOM-jában, ezáltal nagyon gyors és hatékony.

React-ben régen osztály komponensekkel lehetett dolgozni, de újabban a funkcionális komponensek egyre több támogatást és funkciót kaptak, most már ez az ajánlott módja a React-ben való fejlesztésnek. A funkcionális komponensek lényege, hogy tiszta, mellékhatásmentes függvényekkel írjuk le a komponenseinket, melyek bemenetként kaphatnak bármilyen értéket és kimenetként a kirajzolandó komponens adják vissza. Mivel ezek mellékhatásmentes függvények, ezért ha nem változik a bemenetük, akkor nem változik a kimenetük se, ezért a React nagyon effektíven tudja eldönteni, hogy állapot változásnál melyik komponenseket kell újra rajzolni és melyikeket nem.

Ennek ellenére valahogy mégis le kell kezelni például állapotok változását, külső API hívásokat, console-ra írást, melyeket tiszta környezetben nem tehetnénk meg. Erre adnak választ a React Hook-ok. Ezek a 'kampók' 'belekapaszkodnak' egy funkcionális komponensbe és a programozók oldalán egy funkcionális interfészt biztosít ilyen típusú dinamizmus kezelésére. A *useState* hook például egy állapotot és egy állapot módosító függvényt biztosít nekünk. Egy gomb *onClick* eseményében ha meghívjuk az állapot módosító függvényt, akkor a React a háttérben elvégzi nekünk az értékadást, mi csak azt vesszük észre, hogy az állapotunk megváltozott. Mivel a React kezébe adjuk mutálható változó értékek kezelését, ezért effektív tud maradni a keretrendszer, az előbb leírt feltételes újra rajzolásokat hatékonyan tudja kezelni.

Async Result Monád típus

Hibakezelésre imperatív programozási nyelvekben hagyományosan kivételeket használnak. Typescriptben is meg van a lehetőség kivételek dobására és elkapására. Viszont, mivel funkcionális komponensekkel dolgozok és fejlesztés közben párszor nem lekezelt kivételek miatt inkonzisztens állapotba került a UI, ezért a Haskellben *Either* és Rust-ban *Result* néven ismert monádhoz fordultam. Röviden ezeknek az a lényegük, hogy típus szinten kezeljük le a hibákat, amikor egy függvény alkalmazás lánc végén akarjuk használni az értéket, akkor muszáj megvizsgálnunk, hogy hibába ütköztünk-e vagy lefutott az összes számítás és felhasználhatjuk az értéket.

A Rust mintájára egy *Result* nevű unió típust[3] hoztam létre, ami egyszerre vagy egy *Ok* vagy egy *Err* osztályt tartalmazhat. Az *Ok* és *Err* is egy-egy értéket tartalmaznak, az *Ok*-ban szereplő érték egy jó értéket szimbolizál, amit utána továbbra is lehet használni, az *Err* pedig valamilyen hibát tartalmaz (pl.: *string*, *Exception*, saját osztály), mellyel nem folytatódik tovább a számítás.

A *Result* egy *Funktor* típus, definiálva van rá egy *map* függvény. Hogy ha a *Result* *Ok*, akkor alkalmazza rá a függvényt, ha *Err* akkor pedig az *Err*-t adja vissza. Ezen felül a *Result* egy *Monád* típus, az *andThen* függvény alkalmazza az *Ok*-ban levő értékre a függvényt ami egy *Result* típust ad vissza, vagy ha *Err* érték a *Result*, akkor visszaadja az *Err*-t. Az *sideEffect* függvénnyel mellékhatásosságot lehet elérni, a *Result* marad ugyan az, viszont az *Ok*, akkor az értékre lehet mellékhatásos függvényeket hívni. Ez azért hasznos, mert a Javascriptet nem tisztán funkcionálisra tervezték és például a böngésző API-jával való kommunikációhoz hasznos lehet.

Result-ból adatot kinyerni a *match* függvénnyel lehet, ami paraméterül kap egy *Ok* esetén és egy *Err* esetén lefutó függvényt, a *result* értéke szerint a megfelelő futtatja le.

Kivételt dobó függvényeket be lehet csomagolni, hogy *Result*-ot adjanak vissza a *fromThrowable* függvénnyel. Ha egy függvény nem dob kivételt, de mégis szeretnénk, hogy *Result*-ba csomagolt visszatérítési értéke legyen, akkor a *fromSafe* függvénnyel tehetjük meg.

JavaScriptben és ezáltal Typescriptben az aszinkron függvényeknek muszáj *Promise* típussal visszatérniük. Ez megnehezíti a dolgunkat, mert például *Result* típust így nem tudunk visszaadni. Ezt a problémát oldja meg a *ResultPromise*,

ami lényegében egy Result-ba csomagolt Promise. Ugyan azokkal a függvényekkel rendelkezik mint a Result (*map*, *andThen*, *sideEffect*), viszont Promise-okkal dolgozik a háttérben. Ha egy Promise Rejected állapotba kerülne, akkor a ResultPromise-on belüli Result Err lesz.

A result függvények amik Async végződésűek ugyan azon az elven alapulnak, mint a megféleőkik (pl.: *map*, *mapAsync*) csak aszinkron függvényekkel dolgoznak.

Összességében, a result és ResultPromise típusokkal és függvények kompozíciójával sok hibakezelő boilerplate kód kerülhető el és típusbiztos lehet a kód.

Komponensek

3.2. Megvalósítás

3.2.1. Fejlesztési környezet

Rider, vs code, visual studio, docker
dotnet, nuget packages dotnet ef add migrations
snowpack, node.js, yarn, react, npm packages
???

3.2.2. Fejlesztési döntések

exceptions, dto-entity mappers
c# records, LINQ,

3.2.3. Fejlesztés közben felmerült problémák

EF many-to-many, ef virtual classes nullable, sqlite inmemory test parallelization

3.3. DevOps

3.3.1. CI/CD

Github actions, cd dockerrel(?)

3.3.2. Docker

A Docker egy konténerizációs technológia, amely megkönnyíti az alkalmazások kihelyezését. Az alkalmazások 'konténerekbe' csomagolódnak minden függőségükkel, ezen konténereket utána egy egységként lehet futtatni, nem kell a felhasználó rendszerére egyesével a program futásához felállítani a környezetet.

A Docker konténereket úgy találták ki, hogy eldobhatók legyenek, azaz ki lehessen őket törölni és újra futtatni és ugyan úgy működjenek. Az adatok perzisztálását így *volume*-okkal oldhatjuk meg, jelen esetben az adatbázis és a profilképeket a host gép lokális mappáiban tároljuk.

Egy konténer adatait egy Dockerfile nevű fájlban definiálhatjuk. Itt megadhatjuk, hogy milyen más image-ből szeretnénk kiindulni (pl.: ubuntu, debian, node.js, dotnet) és utána, hogy azt hogy szeretnénk testre szabni. Az én Dockerfile-jaim használnak *builder*-eket, ami azt jelenti, hogy az elején definiálom, hogy a forráskód milyen környezetben legyen lefordítva, utána pedig egy másik környezetet definiálok, amiben azt a lefordított kódot futattom. Ennek az az előnye, hogy a végeleges futtatható konténerben nincsenek fordításhoz szükséges függőségek, így kisebb a konténer mérete.

Konténerek optimalizálása

A konténereimben Alpine Linux¹⁴-ot használok. Az Alpine egy minimális Linux disztribúció, melynek a konténerekre optimált változata <3MB. Ezzel szemben egy Ubuntu vagy Debian alapú konténer csak a disztribúciók miatt több száz MB is lehet.

Egy Docker konténer buildelése közben a Docker megjegyzi, hogy az egyes utasításokat ugyan ilyen környezetben végrehajtotta-e már, ha igen, akkor az elcache-elt értéket használja. Ez azt jelenti, hogy ha a konténerünk elején telepítünk fordításhoz szükséges programokat, akkor a következő buildnél nem fogja még egyszer feltelepíteni őket nulláról, hanem használja az előző build cache-ét. Ezt a cache-t a COPY felrúghatja, amikor a fájlrendszerünkről másolunk a konténerbe fájlokat. Ezek a fájlok változhatnak builek között, így nem biztos, hogy lehet cache-elni. A Docker ezeket a fájlokat hash-eli és összehasonlítja, ha nem változtak akkor használja

¹⁴<https://alpinelinux.org/>

a cache-t. Ez a gyakorlatban azért jó, mert például a frontendél a sok npm csomag sokáig települ. Ha a konténerbe bemásolom csak a *package.json* és *yarn.lock* fájlokat és hívom meg a *yarn install* parancsot, akkor ha a következő konténer buildnél nem változnak a dependency-k, akkor lehet cache-ből betölteni azokat. Ez által fejlesztés közben gyorsítható a buildelés folyamata.

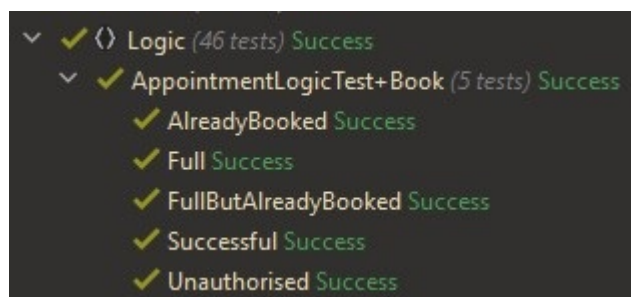
Docker Compose

A Docker Compose programmal konténereket lehet okersztrálni, egy konfigurációs fájlban eltárolni, hogy ezek környezeti változókat, fájlokat, portokat használhatnak, egymást hogy éri el. Mivel egy fájlban található ez a konfiguráció, egyszerűsíti egy teljes rendszer feltelepítési és üzemeltetési komplexitását.

Az alkalmazásban így például a MariaDB adatbázist deklaráltam egy konténernek, megadtam, hogy a backend azt érje el, így az adatbázis telepítése és üzemeltetése is nagyon egyszerű.

3.4. Tesztelés

Mind az integrációs, mind a unit teszteknel a következő struktúrát használtam. Létrehoztam egy teszt osztályt a tesztelendő osztálynak (logika, repository, mapper). Ezen belül létrehoztam egy osztályt a függvény / metódus nevével, azon belül pedig a függvények nevei az adott esetet írják le. Például, sikeres, nem engedélyezett felhasználó, már lefoglalt időpont.



3.10. ábra. Teszt struktúra

screenshot,
hogy lefutott
az összes teszt

3.4.1. Unit tesztek

Unit tesztelésnél a backend logika, mapper és repository implementációk függvényeit teszteltem white box módon.

Tesztelésnél sokat segített a függőségi befecskendezés és az úgynevezett mock-olás. Mockolásánál létrehozunk egy hamis interfész implementációt, ez után különböző szabályokkal megadhatjuk, hogy melyik függvények milyen paraméterekre milyen értékeket adjanak vissza.

Például az AppointmentLogic CreateAppointment metódusában meghívom a CategoryRepository GetById metódusát és az AppointmentRepository CreateAsync metódusát. Ha mockolom a CategoryRepository-t, akkor megmondhatom, hogy ha a GetById-t 10-re hívják meg, akkor adjon vissza egy konkrét kategóriát amit előtte definiálok. Az AppointmentRepository-nál pedig a mockolás miatt ellenőrizni tudom, hogy sikeres futás után meghívta-e a logika pontosan egyszer a CreateAsync metódust.

Ezzel a unit tesztjeim futásai közben nem kell futnia az adatbázisnak, mert a repository-k nem az adatbázisban dolgoznak, hanem a mockolt objektummal váltottam ki a működésüket. Viszont, a logika szempontjából ugyanúgy egy repository-n keresztül dolgozunk, a logika teljesen jól tesztelhető, nem függ külső szolgáltatásoktól.

A repository-k teszteléséhez hasonlóan mockolok, viszont az EF DbContext-jét kell mockolni, azon keresztül érhető el az adatbázis. Erre a MockDbContextBuilder osztály szolgál, ami létrehoz egy mock DbContext-et a szolgáltatott adatokkal. Így azt lehet szimulálni, hogy azok az adatok vannak az adatbázisban, lehet tesztelni a repository helyességét, hogy megfelelően kérdezi le vagy módosítja őket.

A kontrollereket nem unit tesztelem, mert csak átalakítják a bemenetet, meghívják a logika egy függvényét és DTO-vá alakítják a kimenetet. A kontrollereket integrációs tesztelés során tesztelem, mert ha az integrációs tesztek átmennek, akkor az azt jelenti, hogy a kontrollerek is jók.

Az unit tesztek a következő struktúrájúak:

1. **Arrange:** A teszthez szükséges adatok előkészítése. Itt általában a mock objektumok előkészítése történik, vagy a várt eredmény definiálása.

2. **Act:** A tesztelendő műveletek végrehajtása, a visszatért érték eltárolása egy változóban. Kezelt kivételek esetén azt ellenőrzöm, hogy kiváltódott-e a megfelelő kivétel.
3. **Assert:** Ellenőrzés, hogy a művelet megfelelően hajtott-e végre. Ellenőrzöm a visszatérési érték helyességét és azt, hogy a mock objektum megfelelő metódusai megfelelően meg lettek-e hívva vagy nem.

3.4.2. Integrációs tesztek

A backendem integrációs tesztelve is lett. Ez azt jelenti, hogy az api-t futtatom és nem egyesével az egységeit tesztelem mint unit tesztelésnél, hanem hogy az egész rendszer a bejövő kéréstől a kiadott válaszig jól működik-e.

Ahhoz, hogy az integrációs tesztek gyorsak és külső szolgáltatás függetlenek legyenek SQLite¹⁵ inmemory adatbázist használok. Ez egy teljes értékű SQL szerverrel ér fel, ami az eszköz memóriájában fut. Így az integrációs futhatnak Coninuous Integration közben, nem kell egy MardiaDB szervert futtatni a felhőben.

Az integrációs tesztekhez egy előre elkészített adat csomagot töltök be mindig az adatbázisba, ezt az IntegrationTestData osztály tartalmazza.

A TestWebApplicationFactory egy gyár tervmintát alkalmazó osztály, ami ez előbb említett SQLite inmemory adatbázissal konfigurál fel és készít el egy alkalmazás példányt.

Az IntegrationTestBase osztály az ősosztálya az összes integrációs teszt osztálynak, tartalmaz egy TestWebApplicationFactory-t és pár segédfüggvényt, hogy egyszerűbb legyen megírni a teszteseteket.

Az integrációs tesztek a következő struktúrájúak:

1. **Arrange:** A teszthez szükséges adatok előkészítése. Ez lehet egy új DTO létrehozás, amit majd egy POST request-el elküldök az API-nak, vagy az adatbázisból érték kiolvasás, hogy a végén össze lehessen hasonlítani, hogy egyezik-e a visszatért értékkel.

¹⁵<https://www.sqlite.org>

2. **Act:** A tesztelendő műveletek végrehajtása. Egy HTTP klienssel HTTP kéréseket küldök az API-nak, ha kell egy DTO-val, vagy előtte még egy bejelentkező kéréssel.
3. **Assert:** Ellenőrzés, hogy a műveletek megfelelően hajtottak-e végre. Itt vizsgálom a HTTP státusz kód helyességét, a visszatért JSON adatot beolvasom DTO-ként és ellenőrzöm, hogy megfelelő-e.

3.4.3. Manuális tesztek

frontend tesztelés

4. fejezet

Összegzés

Kell egy
összefoglaló

4.1. További fejlesztői lehetőségek

- email - confirmation, password reset, időpont változás
- mobil alkalmazások

Mobil
optimalizált
weblap?

- mobil optimalizált weblap (?)
- Adószám, számlázási infó, hogy lehessen rendes valid számlát kiállítani
- Számlázz.hu integráció
- Lemondás X időn (24 órán) belül nem lehetséges
- emailt használni username helyett kb mindenhol

Irodalomjegyzék

- [1] Robert C. Martin (Uncle Bob). *The Clean Architecture*. Aug. 2012. URL: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html> (visited on 04/14/2021).
- [2] *NGINX Reverse Proxy*. URL: <https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/> (visited on 04/25/2021).
- [3] *Union Types*. URL: <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types> (visited on 04/25/2021).

Ábrák jegyzéke

3.1. Felhasználói esetek	12
3.2. Entitások UML diagrammja	16
3.3. DTO-k UML diagrammja	17
3.4. Logika osztályok UML diagrammja	18
3.5. Controller-ek UML diagrammja	19
3.6. Repository interfészek UML diagrammja	20
3.7. Program komponenseit összeítő UML diagramm	21
3.8. REST API végpontok	23
3.9. REST API végpontok folytatása	24
3.10. Teszt struktúra	30