XXII. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)
Kolozsvár, 2019. május 23–26.

Bookta

Online könyvesboltok bányászására és csoportos rendelésre alkalmas szoftverrendszer

Szerzők:

Döngölő Zsolt
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év
Takács Kálmán
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

Kiss Anna, szoftverfejlesztő
Codespring
Ráduly Sándor, szoftverfejlesztő
Codespring
Sulyok Csaba, doktorandusz
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

CODESPRING
OUR CODE YOUR SUCCESS

Kivonat

A Bookta alkalmazás célja, hogy online könyvesboltokból nyerje ki a rendelkezésre álló könyvek információit adatbányászat (ezen belül web scraping) segítségével, majd a felhasználók elé tárja ezeket egy mobil alkalmazás keretében.

A rendszer lehetővé teszi, hogy a felhasználó rákeressen szűrők segítségevel egy adott könyvre, értékeljen egy könyvet, kosarába helyezze, kedvencekhez adja, valamint csoportos rendelést hozzon létre. A csoportos rendelésbe történő felhasználók meghívása egy linken keresztül történik, amely minden rendelésnél egyedi, és a rá hivatkozó személyt az alkalmazásba irányítja. Ez esetben szabadon böngészhetjük a csatlakozott felhasználók kosarait. Továbbá a rendszer alkalmas a könyvek árainak, kiszállítási idejének összehasonlítására különböző üzletekben. Így egy egységes platformon szabadon kereshetünk a webáruházakban levő könyvek között, és kiválaszthatjuk a számunkra legelőnyösebb üzletet, amelyben egyénileg vagy csoportosan vásárolhatunk.

Jelen dolgozat bemutatja a Bookta nevű szoftverrendszer architektúráját, működését és megvalósításának részleteit, utóbbin belül a fejlesztés során felhasznált technológiákat, eszközöket és módszereket.

Tartalomjegyzék

Bevezető

A Bookta alkalmazás célja, hogy megkönnyítse az online platformokon történő csoportos könyvrendeléseket. Ennek érdekében összegyűjti a nagyobb online felületekről *(Bookline*[1]*, Libri*[2] *stb.)* a könyvek információit, amit adatbányászat, ezen belül web scraping segítségével valósít meg.

A szoftverrendszer három komponensből áll, ez két kliens alkalmazást, valamint egy központi szervert foglal magába. Az első kliens feladata, hogy kibányássza az említett *webshop*-okból a könyvek információt és ezeket felküldje a szervernek, ami megfelelő módon lementi az adatbázisba ezeket az adatokat, az üzlet adataival együtt. A fent említett információkat a felhasználók megtekinthetik egy Android alkalmazáson belül, amely lehetőséget nyújt számos műveletre a könyvekkel. Mivel előfordulhat, hogy egy bizonyos könyv példánya több üzletben is árusítva van, így ezeket felismeri a rendszer és összehasonlíthatóak a különböző platformokon levő árak. Továbbá összetett szűrőkkel, és keresővel böngészhetőek a termékek több kritérium alapján, amelyek hozzáadhatóak a kedvencekhez, kosárba helyezhetőek.

Az alkalmazás legfőbb funkcionalitását tekintve kezdeményezhetünk csoportos rendeléseket is. A csoportos rendelésbe meghívhatóak más regisztrált felhasználók is egy egyedi linken keresztül. Ezt a linket az alkalmazás generálja a csoportos rendelés létrejöttekor, és egy gomb segítségével vágólapra másolhatjuk. Ha a linket a felhasználó egy Android készüléken nyitja meg, a rendszer a Bookta alkalmazásba irányítja, ahol automatikusan csatlakozik a csoportos rendeléshez. Ugyanitt megtekinthető mások kosara, illetve a felhasználó saját kosarát szerkesztheti. A csoportos rendelés üzlet-orientált, így például ha egy rendelés a Bookline-ra volt készítve, akkor a meghívott felhasználók csak olyan könyveket adhatnak hozzá a kosarukhoz, amelyek elérhetőek a Bookline-on. A rendeléseket le lehet zárni, amely hatására a rendelés indítója egy e-mailben fog értesülni a megrendelni kívánt könyvek linkjeiről.

A Bookta projekt fejlesztése 2018 októberében vette kezdetét a Codespring mentorprogram, illetve a csoportos projekt keretein belül. A fejlesztés kezdeti szakaszában az egyetem részéről segítségünkre voltak Ballán Dávid-Lajos, Tat Alexandra-Mária, Fogarasi Norbert és Nagy Róbert. Szeretnénk köszönetet nyilvánítani a hozzájáruló diákoknak, valamint a Codespring részéről szakmai segítőinknek, Kiss Annának és Ráduly Sándornak, illetve témavezető tanárunknak, Sulyok Csabának.

A dolgozat hátralevő része a következőképpen van struktúrálva: az 1. fejezet bemutatja a Bookta alkalmazás szerepköreit a hozzájuk tartozó funkcionalitásokkal, valamint a három komponens egymással való kommunikációját. A 2., 3. és 4. fejezetekben a három komponens

[1] https://www.bookline.hu
[2] https://www.libri.hu

megvalósítása kerül leírásra. Szó esik a komponensek belső felépítéséről, bemutatva a felhasznált technológiákat. Az 5. fejezetben érintjük az eszközöket, amelyek hozzájárultak az alkalmazás létrejöttéhez, valamint a 6. fejezet bemutatja az alkalmazás működését.

1. A Bookta alkalmazás

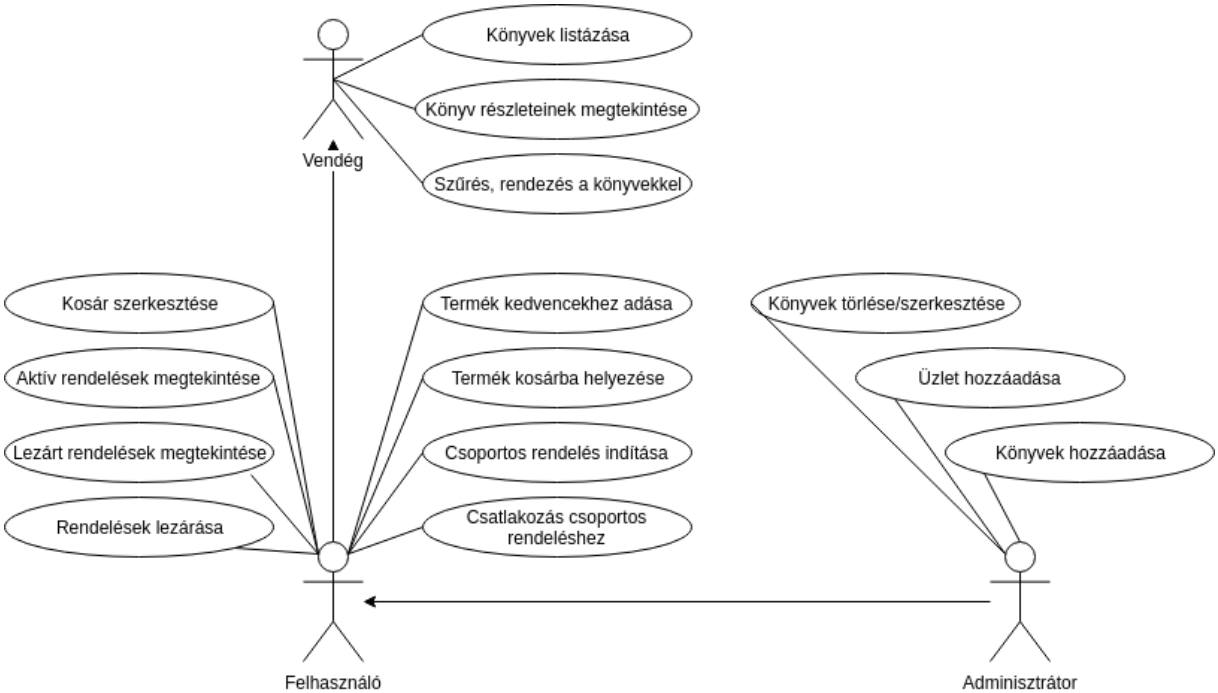
A Bookta projekt megkönnyíti a csoportos rendeléseket könyvek esetében, valamint nyújt egy egységes platformot, ahol több üzlet által árusított könyveket megtekinthetünk. Ebben a fejezetben az alkalmazás szerepkörei és az azokhoz tartozó funkcionalitások, illetve a rendszer teljes architektúrája kerül bemutatásra.

1.1. Szerepkörök és funkcionalitások

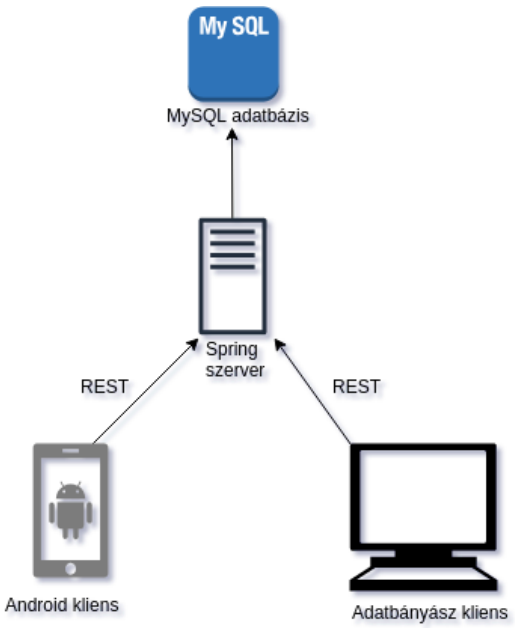
Az alkalmazás felhasználói három kategóriába sorolhatóak (ld. 1. ábra): vendég, felhasználó és adminisztrátor.

Vendég szerepkörrel rendelkezik minden be nem jelentkezett felhasználó. Ebben a szerepkörben levő felhasználónak lehetősége van kilistázni az elérhető könyveket, megtekinteni azokról az információkat, illetve szűrni, keresni közöttük. A vendég nem jogosult a termékekkel történő egyéb műveletek elvégzésére.

Sikeres regisztráció és bejelentkezés után a vendég **felhasználói** jogot kap. A vendég összes funkcionalitása mellett ez a szerepkör rendelkezik saját kosárral, illetve kedvencek listájával. Ennek köszönhetően a megnézett könyveket behelyezheti kedvencei közé, vagy saját kosarába. Ezek mellett indíthat csoportos rendelést is, amelyben szintén lesz egy kosara. Megoszthatja további felhasználókkal a csoportos rendelést. Minden felhasználó karbantarthatja saját kosarait, egy-egy rendelés indítója pedig lezárhatja ezt. Ezen kívül megtekinthetőek a lezárt, illetve aktív csoportos rendelések is.



1. ábra. A rendszer szerepkörei és a hozzuk tartozó funkcionalitások



2. ábra. A rendszer architektúrája és a komponensek közötti kommunikáció

Az **adminisztrátor** szerepköre rendelkezik a fent említett összes funkcionalitással, viszont joga van törölni, szerkeszteni, vagy hozzáadni új könyvet az alkalmazáshoz.

1.2. Architektúra

A Bookta szoftverrendszer három fő komponensből tevődik össze (ld. 2. ábra): egy mobilalkalmazásból, egy adatbányász kliensből illetve egy központi szerverből. A kliensek REST API-n keresztül kommunikálnak a szerverrel.

A szerver esetében az API réteg (2.4. fejezet) komponensei fogadják a hálózati kéréseket. Ezután a szolgáltatási réteg (2.3. fejezet) komponenseivel elvégeztetik az üzleti logikáért felelős műveleteket. Az adatok tárolására a szerver relációs adatbázist használ és az adathozzáférési rétegen (2.2. fejezet) keresztül valósul meg a kommunikáció. Az API réteg a kapott választ DTO (Data Transfer Object) objektumokba alakítja és JSON formátumban továbbítja a kliensnek.

Az Android kliens alkalmazásban a szerverrel történő kommunikáció az adatelérési rétegen (3.2. fejezet) történik meg. Ennek a rétegnek a komponensei a válaszul kapott DTO-kat átalakítják modellekké. A prezentációs rétegre (3.1. fejezet) az adatok modell objektumok formájában érkeznek. A nézetmodell értesíti a kapott válaszról a nézetet, amely megjeleníti az információkat.

Az adatbányász kliens (4. fejezet) időszakosan begyűjt információkat külső online könyvesboltokból, és ugyanazon REST API-n keresztül JSON formátumban felküldi a szervernek. A szerver ezeket az adatokat elmenti az adatbázisba, ahonnan kiszolgálja az Android klienseket.

2. Szerver

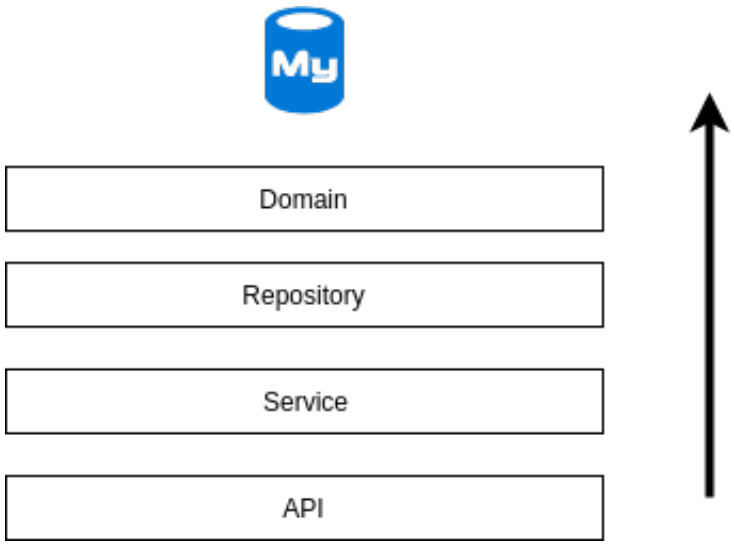
A szerver a Spring [6] több fontosabb projekjét is használja, melyeknek egyszerű és gyors konfigurációját a Spring Boot [36] biztosítja. A Spring egy IoC-t (Inversion of Control) megvalósító keretrendszer, amely számos modullal rendelkezik. Az IoC konténer menedzseli a beaneknek nevezett központi entitásokat és a közöttük lévő függőségeket DI [26] (Dependency Injection) tervezési minta alapján kezeli. Ezen technikák alkalmazása által csökken a szoros függőség a szoftver egyes részei között, elkülönül a beanek mendezsmentje a program többi részétől. Az interfész alapú kommunikáció által az egyes rétegek függetlenek a többi réteg implementációjától. Ez átlátható, könnyen karbantartható és cserélhető kódot eredményez.

Az alkalmazás a többrétegű architektúra [29] alapján van felépítve. Szolgáltatásai REST API-n [35] keresztül vannak publikálva. Négy fontosabb réteget különíthetünk el az alkalmazáson belül: az adatmodell, az adathozzáférési réteg, a szolgáltatási réteg és az API (ld. 3. ábra).

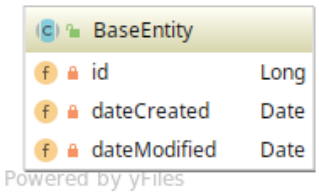
A fejezet következő részeiben bemutatásra kerülnek az egyes rétegeken felhasznált minták, valamint azok megvalósításában használt technológiák. Ezen kívül sor kerül majd a hibakezelés és a biztonság részleteinek leírására is.

2.1. Adatmodell

Az adatokat reprezentáló osztályok a `domain` csomagban találhatóak. Minden entitás a `BaseEntity` osztályból származik, amely kiemeli ezek közös tulajdonságait (ld. 4. ábra). Ezek az osztályok betartják a *Java Bean* definiciót, azaz rendelkeznek paraméter nélküli konstruktorral, publikus getter/setter (adattag lekérő, illetve beállító) metódusokkal, valamint



3. ábra. A rétegek közötti kommunikációs diagramm. Minden réteg egy külön Java csomagnak felel meg, és függősége a hierarchiában felette lévő csomagnak.

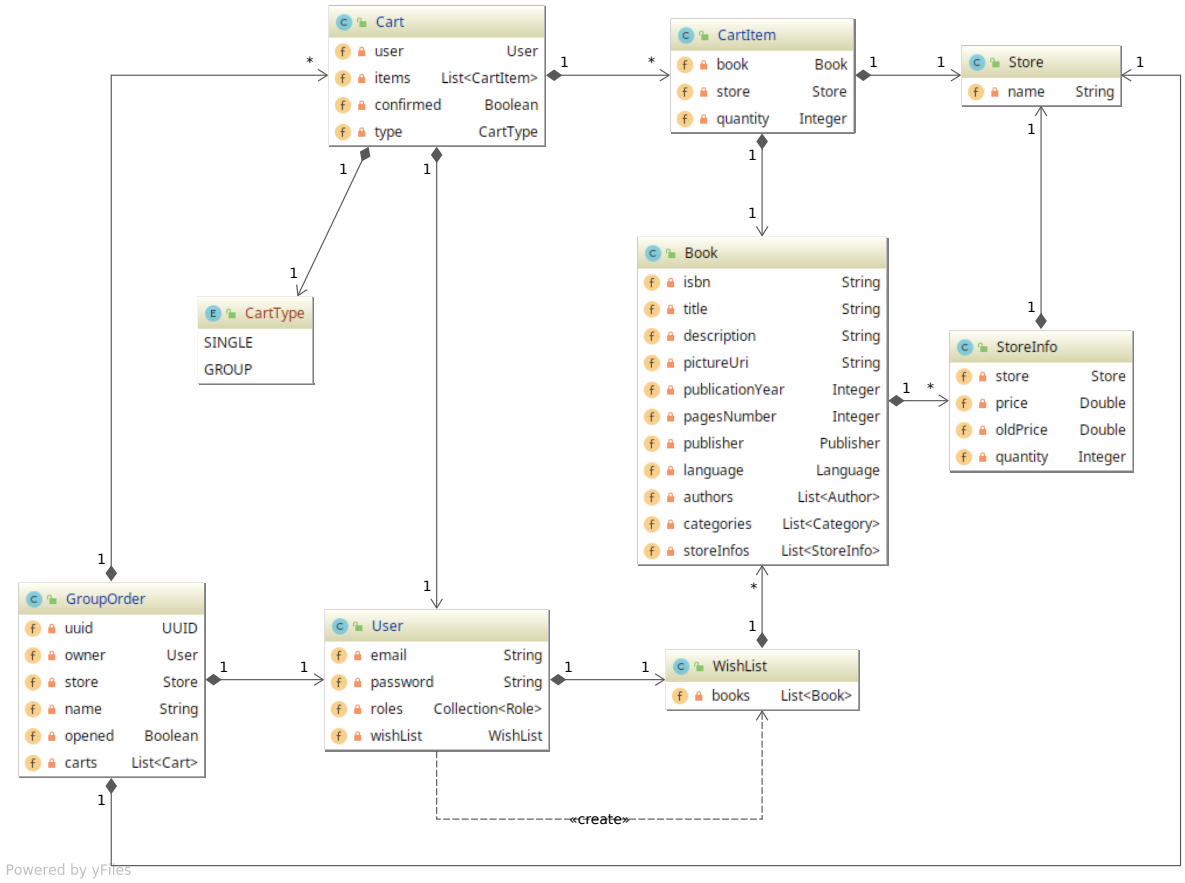


4. ábra. A `BaseEntity` osztály. Az `id` adattag az adatbázis elsődleges kulcsa, míg a `dateCreated` és `dateModified` adattagok a létrehozás, illetve utolsó módosítás dátumai.

implementálják a `Serializable` interfészt (a `BaseEntity`-től öröklik). Ezek az osztályok vannak leképezve relációs adatbázismodellként, és alkotják az adatbázisséma tábláit.

Mint a 5. ábra is mutatja, a hierarchia középpontjában a `Book` osztály található, amely tartalmaz minden információt egy a rendszerben ábrázolt könyvről. A `Book` osztály egy a többhöz kapcsolatban áll a `StoreInfo` osztállyal, amely az egyes könyvek különböző üzletekben elérhető árát tartalmazza. Ezen kívül jól elkülönített entitások reprezentálják a rendszeren belül tárolt kategóriákat, üzleteket, szerzőket, nyelveket és kiadókat, sorban a `Category`, `Store`, `Author`, `Language`, `Pubisher` osztályok.

A `User` osztály reprezentálja az alkalmazás felhasználóit, és több a többhöz kapcsolatban



5. ábra. A rendszer központi entitásai (a `model` csomag egy része)

áll a `Role` entitással, amely a felhasználókhoz tulajdonított jogokat jelképezi. Ezenkívül minden `User` objektumhoz tartozik egy `WishList` objektum is, amely az adott felhasználó kívánságlistára helyezett könyveit tartalmazza.

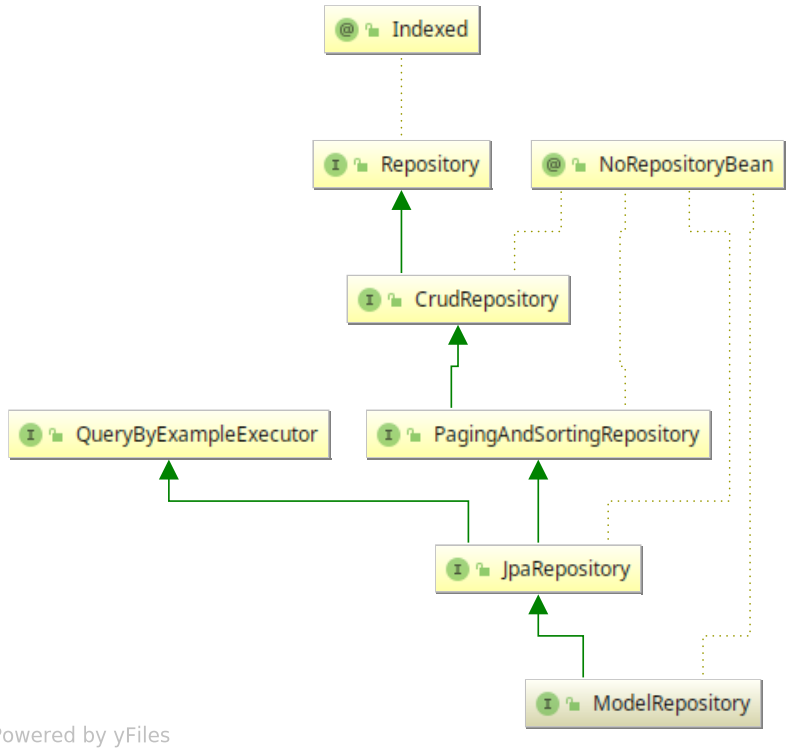
A másik jelentős entitás a rendszerben a `Cart` osztály, amely egy adott felhasználó vásárlói kosarát reprezentálja. Tartalmaz egy `type` adattagot, ami megadja, hogy az adott `Cart` objektum egy csoportos rendeléshez tartozó kosarat, vagy a felhasználó saját kosarát jelenti. A `Cart` több a többhöz kapcsolatban áll a `CartItem` entitással, ami beazonosít egy bizonyos könyvet egy bizonyos üzleten belül, valamint tartalmazza a rendelni kívánt darabszámot. A `GroupOrder` osztály is ilyen `Cart` objektumokat tartalmaz, és ezáltal jelképezi a csoportos rendelést. Ez az entitás a résztvevő felhasználók kosarain kívül tartalmaz még egy `UUID` típusú azonosítót, amelyen kereszül majd csatlakozni tudnak új felhasználók a csoportos rendelésbe. A rendelést kezdeményező felhasználó véglegesíteni tudja a rendelést, amit az `opened` adattag igazra való átállításával ér el.

2.2. Adathozzáférés

Az adatok hozzáféréséért felelős osztályok a `repository` csomagban találhatóak. Ez a csomag minden jelentősebb modell objektumhoz tartalmaz egy adathozzáférési osztályt (például `BookRepository`), amely az adott entitással kapcsolatos adatmanipulációs műveleteket hivatott végrehajtani. A Spring Data JPA [25] modul biztosítja az adatok egyszerű elérését, egy absztrakciós szintet emelve a JPA [14] (Java Persistence API) fölé. A JPA szerepe az adatok egységes kezelése az adattároló természetétől eltekintve. A Bookta alkalmazás MySQL adatbázist használ az adatok tárolásához, viszont bármilyen más adatbázist is könnyedén lehetne integrálni a Spring Data-nak köszönhetően. Az adatok relációs adatbázisba való leképezését a Hibernate [17] ORM keretrendszer végzi, ami a Spring Data JPA alapértelmezetten bekonfigurált JPA implementációja.

Az adathozzáférési osztályok központjában a `JpaRepository` beépített interfész áll, amely tartalmazza a leggyakoribb adatmanipulációs műveletek nagyrészét. Egy entitás esetén úgy lesznek elérhetőek ezek a műveletek, hogy származtatunk egy interfészt az előbb említett interfészből, megadva két típusparamétert: az entitás típusát, valamint az egyedi azonosító mezőjének a típusát. Az azonosító minden esetben `long` típusú, ezért be van vezetve egy `ModelRepository` interfész is, amely csak az entitás típusát kéri paraméterül. Minden specifikus adathozzáférési osztály a `ModelRepository` osztályból származik, ezáltal alapértelmezetten tartalmazzák a `JpaRepository` és annak összes ősinterfészének metódusait (az interfészek öröklődéséről bővebben ld. 6. ábra)

A lekérések dinamikusan generálódnak az alkalmazás indulásakor, valamint futási időben



6. ábra. A hierarchia tetején a `Repository` interfész található, amely jelzi a Spring Data számára az adathozzáférési osztályok jelenlétét, és indexeli azokat a rendszerben. A `CrudRepository` interfész tartalmazza a CRUD (create, read, update, delete) műveleteket, ezeket a `PagingAndSortingRepository` interfész egészíti ki egyszerű szűrő és rendező műveletekkel, valamint a `JpaRepository` interfész biztosítja a JPA specifikus műveleteket.

paramétereződnek és hajtódnak végre. A Spring Data JPA lehetőséget ad saját lekérések generálására is, natív SQL kód alapján, JPQL alapján, vagy az adathozzáférési interfészek metódusneveinek elnevezése alapján. A Bookta projekt nem használ natív SQL vagy JPQL kódot, minden lekérés metódusnév alapján generálódik.

Példaként a `BookRepository` `findByIsbn` metódusa, amely dinamikusan van generálva a neve alapján, és egy könyvet hivatott lekérni annak ISBN mezője alapján:

```
Optional<Book> findByIsbn(String isbn);
```

A Spring Data tartalmaz más interfészeket is lekérések végrehajtásához, ilyen például a `JpaSpecificationExecutor` interfész, amely a bonyolultabb szűrő és rendező műveletek végrehajtására alkalmas. Míg a `PagingAndSortingRepository` csak rendezésre és oldalankénti lekérésre alkalmas (`Sort`, és `Pageable` objektumok által), addig az előbb említett interfész képes komplex szűrő műveletek végrehajtására is, egy `Specification` objektumon keresztül.

```java
@Override
@Transactional(readOnly = true)
public User findUserByEmail(String email) {
    return userRepository.findByEmail(email)
            .orElseThrow(itemNotFoundException("user.notFound").supplier());
}
```

1. kódrészlet. A `UserService` egyik metódusának implementációja

2.3. Üzleti logika

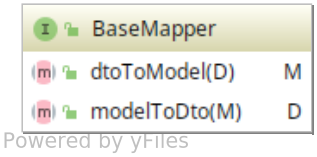
Az üzleti logikát megvalósító komponensek a `service` csomagban találhatóak. A legfontosabb modell objektumokhoz tartozik egy-egy üzleti logikát definiáló interfész és egy ahhoz tartozó implementáció. Ezeket az interfészeket implementáló osztályok az `impl` alcsomagban találhatóak a `service` csomagon belül. Ezen osztályok mindegyike rendelkezik egy `@Service` annotációval, amely által a Spring által menedzselt beanek közé tartozik. Az implementációt tartalmazó osztályok az API rétegen az interfész típusa alapján vannak injektálva. Ezáltal magasfokú cserélhetőség érhető el az üzleti logikát tartalmazó osztályok implementációját illetően.

Fontosabb feladataik a beérkező paraméterek helyességének ellenőrzése, fellépő kivételek lekezelése, valamint rétegspecifikus kivételek továbbdobása adott esetben. Ezen kívül itt történik a tranzakciók kezelése, a különböző üzleti logikát tartalmazó kódrészletek végrehajtása, valamint az adathozzáférési réteg metódusainak továbbhívása, és válasz küldése az őt hívó rétegek felé. Példaként a `UserService` interfészt implementáló bean `findUserByEmail` metódusa (ld. 1. ábra). A metódus indulásakor *readonly* tranzakciót hírdet, továbbhív az adathozzáférési rétegre, hiba esetén specifikus kivételt dob, helyes eredmény esetén pedig visszatéríti az ereményt reprezentáló `User` objektumot.

2.4. API

A szerver szolgáltatásai REST API-n keresztül vannak publikálva, HTTP kérések által. Az API megvalósításában a Spring Web MVC [8] modul segít. Az `api` csomag tartalmazza a legfontosabb modell osztályok kontroller osztályait (például `BookApi`). Ezen osztályok `@RestController` és `@RequestMapping` annotációval rendelkeznek. A Web MVC központi entitása, a `DispatcherServlet`, fogadja a kéréseket és a megfelelő kontroller osztályok felé továbbítja a `RequestMapping` annotáció `value` paramétere alapján.

A kontroller osztályok metódusai különböző hálózati kérésekhez vannak társítva HTTP metódus és útvonal alapján, a `@GetMapping`, `@PostMapping`, stb. annotációk által. Ezek a metódusok hajtódnak végre az adott kérés beérkezésekor és ezek visszatérítési értéke fog



7. ábra. A `BaseMapper` interfész. Metódusai biztosítják a DTO-ból való átalakítást modellbe, illetve fordítva

bekerülni a HTTP válasz törzsébe. Lehetőség van a HTTP állapotkód beállítására is, a `@ResponseStatus` annotáció által, ami alapértelmezetten 200 OK sikeres végrehajtás esetén. A metódusok paramétereket a `@RequestBody` annotáció segítségével tudnak fogadni, valamint a kérés paramétereit is magkapják a `@RequestParam` annotáció által. (példa egy kontroller metódusra: 2. ábra)

A DTO [21] (Data Transfer Object) tervezési minta alpján a kérés illetve válasz objektumok az erőforrások egy specifikus alakját tartalmazzák. Ezáltal csökken a kommunikáció során fellépő adatátvitel mérete. A transzfer objektumok az `api` csomagon belül a `dto` csomagban helyezkednek el. A beérkező DTO-k validálását Java Bean Validation [10] annotációk által végzi a keretrendszer (például: `@NotNull`, `@Email`, `@Min`, `@Size`).

A transzfer és modell objektumok közötti átalakításra mapper objektumokat használunk. A mapper objektumok az `api` csomagon belül a `mapper` csomagban találhatóak. A mapper osztályok közös őse a `BaseMapper` interfész. Ez az interfész határozza meg a két metódust amit minden mapper osztálynak implementálnia kell (ld. 7. ábra).

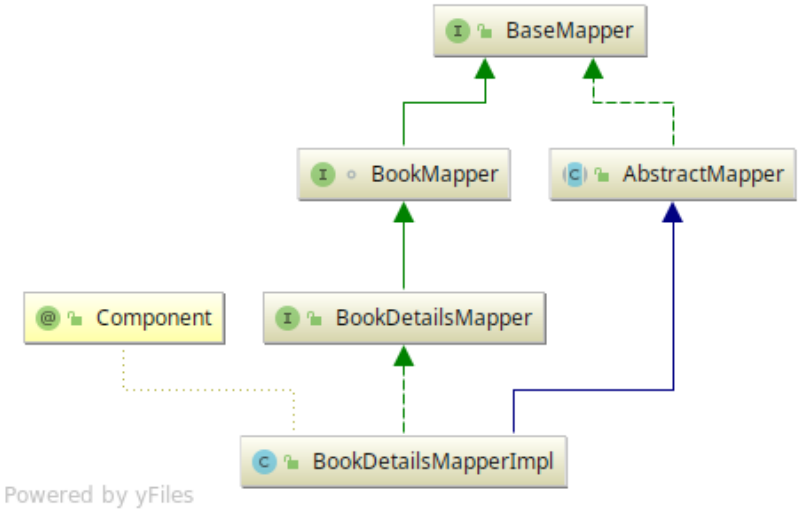
Az `AbstractMapper` absztrakt osztály a `BaseMapper` interfész parciális implementációját tartalmazza. Ez az osztály a *modelmapper* könyvtárat használja. Két absztrakt metódussal rendelkezik:

```
protected abstract Class<M> getModelClass();
protected abstract Class<D> getDtoClass();
```

Tartalmaz egy `ModelMapper` objektumot, amelyen keresztül megvalósítja a `BaseMapper` két metódusának implementációját. Ez az implementáció a DTO és modell azonos nevű valamint típusú adattagjait képes leképezni egyik oldalról a másikra. A különböző típusok konverziójának definiálására az `AbstractMapper` osztály tartalmaz egy metódust, amelyet a származtató osztályok a konstruktoraikban hívhatnak meg. A metódus fejléce:

```
protected <F, T> void addMapping(Class<F> from, Class<T> to,
                                 Function<F, T> converter)
```

Az összes mapper osztály az `AbstractMapper` osztályból származik, így az adattagok másolása, átalakítása automatikusan megtörténik, kevesebb kódot és kevesebb hibalehetőséget eredményezve. (példa a `BookDetailsMapper` öröklődési diagrammja: 8. ábra)



8. ábra. A BookDetailsMapper öröklődési diagrammja

Az API dokumentációja a Swagger2 [33] specifikáció Sprigfox [31] implementációja által lett megvalósítva. A dokumentáció elérhető a */swagger-ui.html* címen, valamint JSON formátumban a *v2/api-docs* címen. Ez a dokumentáció automatiukusan frissül minden változás esetén, és naprakész (vizuális) reprezentációt ad az API-n belül elérhető endpointok valamint azok jogosultágait, paramétereit illetően. A dokumentáció teljes értékű kliensként szolgál, amelybe a felhasználók be is tudnak jelentkezni és a felületről kéréseket intézni a szerverhez. Főként a fejlesztés során hasznos, megkönnyíti a fejlesztők közötti kommunikcáiót, valamint egyszerű tesztelési lehetőségeket biztosít. Integrációja a rendszerbe a Spring-nek köszönhetően nagyon egyszerű. Megvalósításában a `SwaggerConfig` konfigurációs osztály játszik központi szerepet. Az osztály rendelkezik az `@EnableSwagger2` annotációval, beállítja a leírást, dokumentáció nevet, valamint verziót, ezenkívúl biztosítja a bejelentkezési adatokat, amely OAuth 2.0 prtokoll által történik.

2.5. Hibakezelés

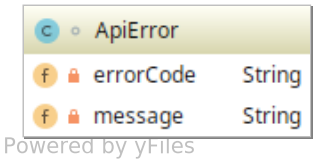
A hibák kezeléséért a `RestExceptionHandler` osztály felelős. Ez az osztály rendelkezik egy `@RestControllerAdvice` annotációval, amely Spring 4.3-tól lett bevezetve, és a `@ControllerAdvice` és `@ResponseBody` annotációkat együttesen helyettesíti. Az osztály metódusai `@ExceptionHandler` annotációval rendelkeznek, amelyek megkapnak egy kivétel osztályt paraméterként, így egy bizonyos típusú kivétel egységesen kezelődik le, kódismétlés nélkül. Minden ilyen metódus felelős egy bizonyos kivételtípus lekezeléséért. A háttérben `@ResponseBody`-val annotált metódusok visszatérési értéke fog belekerülni a válasz törzsébe, követve ugyanazt a szerializációs mechanizmust mint a többi kontroller, ebben az esetben JSON. Így lehetőség adódik plusz információk átadására a HTTP állapotkódon kívül.

```
@RestController
@RequestMapping(value = "stores")
public class StoreApi {
    ...

    @AdminRole
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public StoreDetailsDto createStore(@Valid @RequestBody StoreDto storeDto)

    ...
}
```

2. kódrészlet. A `StoreApi` egyik metódusa. A metódus a /stores -ra történő POST kérésekre hívódik meg, paraméte az előre validált `StoreDto`-t, sikeres lefutás estén 201 CREATED-et és a létrehozott `Store` objektum egy másik DTO-ját fogja visszatéríteni a HTTP válaszban. Valamint a metódus csak adminisztrátori jogokkal elérhető.



9. ábra. Az `errorCode` adattag szöveges típusú rendszer szinten egyedi hibakód (például `user.notFound`, ami lehetőséget ad kliens oldali nemzetköziesítésre. A `message` adattag pedig egy, a szerver által előre definiált konfigurációs fájlból kinyert angol nyelvű hibaüzenet. Ez akár a szerver újraindítása nékül is megváltoztatható.

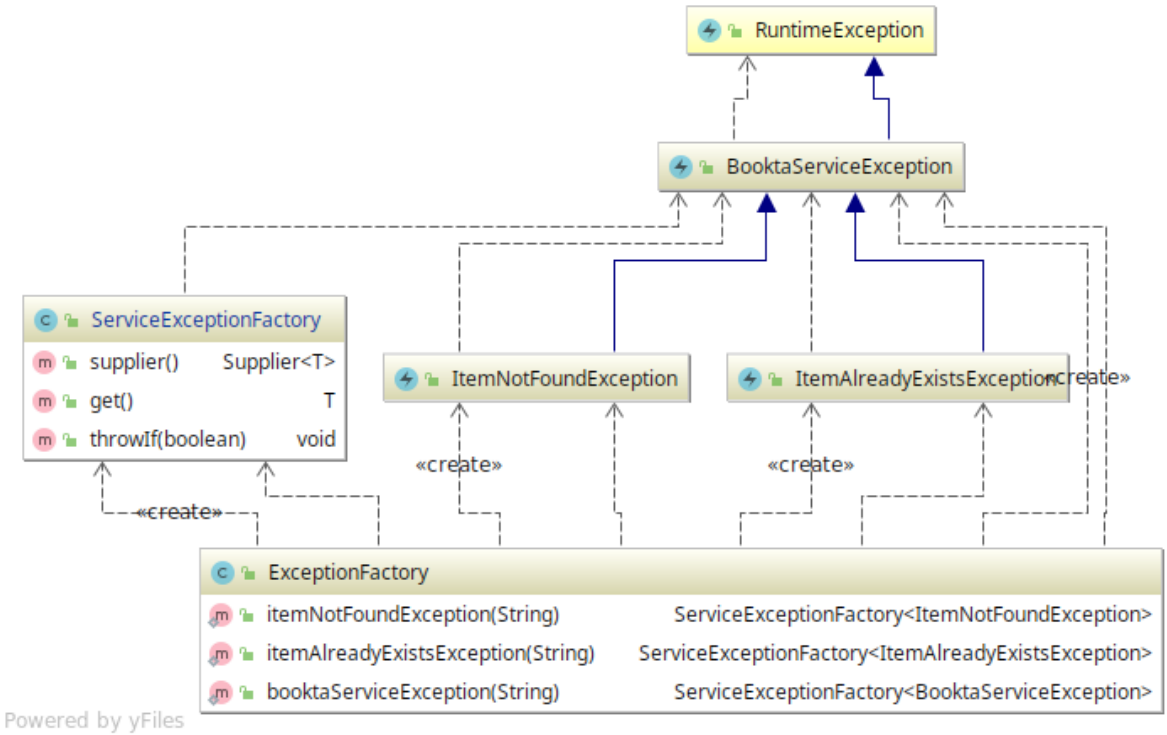
Minden kivételkezelő metódus visszatérítési értéke egy `ApiError` (ld. 9. ábra) objektum, amelybe a kivételekből kinyert információ lesz belecsomagolva. Ez egységessé teszi a hibakezelést kliens oldalon is. Példaként az `ItemNotFoundException`-t lekezelő metódus fejléce:

```
@ExceptionHandler({ItemNotFoundException.class})
@ResponseStatus(HttpStatus.NOT_FOUND)
public ApiError handleItemNotFoundException(ItemNotFoundException e)
```

A rendszerben három egyedi kivételtípus van definiálva: `ItemNotFoundException`, `ItemAlreadyExistsException` valamint egy általános `BooktaServiceException`. Ezek a kivételek az üzleti logika rétegén váltódnak ki többnyire. Példa kivétel kiváltására:

```
itemAlreadyExistsException("user.alreadyExists")
    .throwIf(userRepository.existsByEmail(user.getEmail()));
```

A kivételek példányosítására, feltételes kiváltódására egy factory osztály van létrehozva, a `ServiceExceptionFactory`. Ez az osztály rendelkezik egy típusparaméterrel, amely a



10. ábra. A kivételek osztálydiagrammja

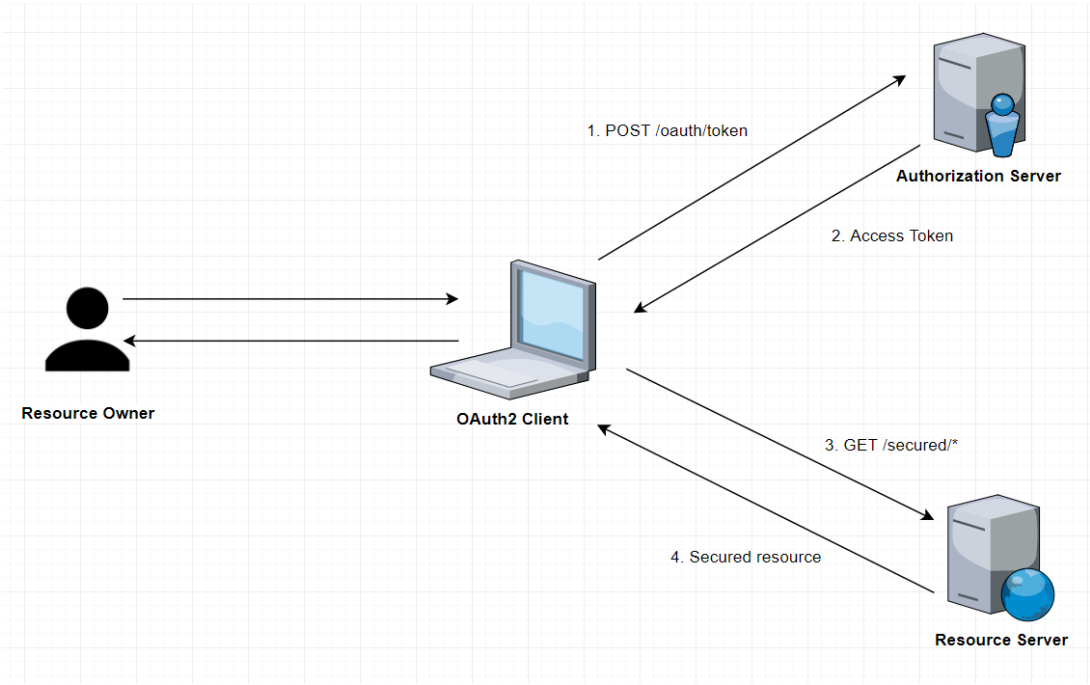
kivétel típusát adja meg. A különböző kivételek factory objektumai az `ExceptionFactory` statikus metódusain keresztül érhetőek el (ld. 10. ábra). Segítségükkel egyszerű, átlátható és központosított hibakezelés érhető el.

2.6. Biztonság

Az alkalmazásban a hitelesítés és engedélyezés (authentication és authorization) OAuth 2.0 [4] protokoll szerint történik. A Spring Security OAuth 2 [23] biztosítja az OAuth 2.0 implementációt a Spring Security [30] keretrendszerhez. Az OAuth 2.0 protokoll négy fontos szerepet különböztet meg: az erőforrás birtokosa (a felhasználó), a kliens, az erőforrás szerver és az autorizációs szerver. A felhasználók a kliens alkalmazásokon keresztül kérnek access tokeneket az autorizációs szervertől. Az autorizációs szerver egy access tokent biztosít, amelyet a kliens minden további kérés fejlécében meg kell jelenítsen majd, így biztosítva a hozzáférést az erőforrás szerver védett adataihoz. A protokoll négy különböző módszert különböztet meg az access tokenek megszerzését illetően. Ebből a Bookta alkalmazás a jelszó alapút használja (ld 11. ábra[3]). Az autentikációs adatok tárolása JWT [5] (JSON Web Token) tokeneken alapszik. Ez egy kis méretű, digitálisan aláírt önleíró struktúra, amely lecsökkenti az adatbázis műveletek számát és biztosítja az állapotmentes kommunikációt a kliensekkel.

A felhasználót a rendszerben a `User` osztály reprezentálja. A felhasználók egy

[3]Forrás: `https://dzone.com/storage/temp/8259301-1.png`



11. ábra. Az OAuth 2.0 protokoll jelszó alapú flow-ja

`UserDetailsService` objektumon keresztül lesznek elérhetőek a Spring Security számára, amit a`WebSecurityConfigurerAdapter` beépített interfész `SecurityConfig` implementációja állít be.

Az alkalmazásban az autorizációs szervert az `AuthorizationServerConfig` konfigurációs osztály jelképezi, amely a `AuthorizationServerConfigurerAdapter` osztály leszármazottja. Ebben az osztályban definiáljuk a tokenek tárolási mechanizmusát (ami jelen esetben egy `JwtTokenStore` objektum), a jelszó titkosítót (`bcrypt`), és a kliensek tárolásának módját. A kliensek memóriában tárolódnak egy `ClientDetailsService` objektumban, ahol többek közt eltárolódik a kliens azonosítója, a titkos jelszava, valamint az access és refresh tokenek lejárási ideje.

Az erőforrás szervert a `ResourceServerConfig` konfigurációs osztály valósítja meg, amely a `ResourceServerConfigurerAdapter` osztály leszármazottja. Fontosabb tulajdonságai az egyedi erőforrás azonosító beállítása, valamint a különböző alapbeállítások erőforrás elérési útvonal minták alapján. Az erőforrásokhoz szükséges jogosultságok megadása nem globálisan történik az előbbi osztály által, hanem a Spring Method Security-t használva a metódusok szintjén. Két annotáció létezik a rendszerben, amelyek segítségével beállítható az adott endpointhoz szükséges jogosultság: `@AdminRole` és `@UserRole`, amelyek a beépített `@PreAuthorize` annotáció előre paraméterezett változatai.

Az `AdminRole` osztály parciális definiciója:

```
@PreAuthorize("hasRole('ADMIN')")
public @interface AdminRole {
}
```

3. Android kliens alkalmazás

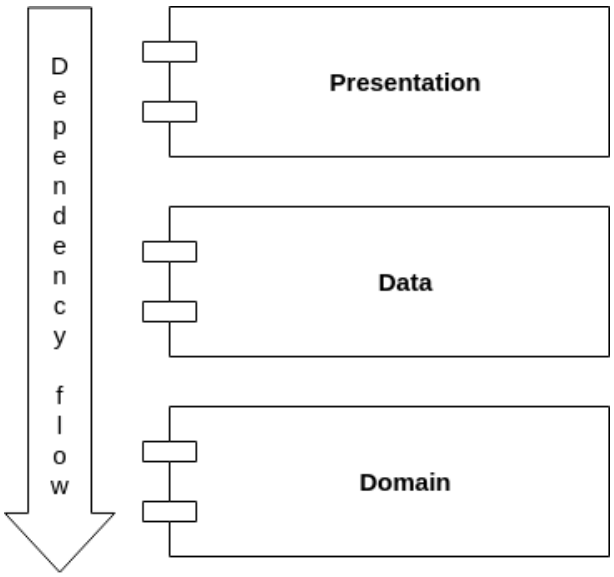
A mobilalkalmazás Kotlin [16] nyelvben íródott, amely 2011-ben került nyilvánosságra. 2017-ben a Google bejelentette, hogy a Java mellett a Kotlint is elfogadja, mint Android platformra történő fejlesztésre hivatalos programnyelv.

Az Android alkalmazás három fő modulból tevődik össze, melyeknek belső felépítését a 12. ábra illusztrálja. A hierarchia legalsó rétegén a `domain` modul helyezkedik el, amely független a másik két rétegtől. A domain modulban találhatóak a központi entitások, valamint az interfészek, amelyekből kiolvashatóak a rendszer funkcionalitásai (use case-ek), de a műveletek implementációi itt nem szerepelnek. Ez azért hasznos, mert ha változik a konkrét implementáció, akkor csak hozzá kell adni az újat és a többi réteg változatlanul használhatja az interfészeket anélkül, hogy valamit módosítottunk volna. A domain feletti réteg a `data`, amely implementálja a `domain` modulban levő interfészeket, és valósítja meg a szerverrel történő kommunikációt. Az API kliens a Retrofit könyvtár segítségével van implementálva (ld. 3.2. fejezet). A legfelső réteg felelős az alkalmazás külalakjáért, itt találhatóak a vizuális elemek, valamint ezen keresztül lép a felhasználó interakcióba az alkalmazással. A `presentation` modulban lép fel a dependency injection használata (ld. 3.5. fejezet), ahova a `domain` réteg interfészei injektálódnak az adathozzáférési rétegen definiált implementációkkal.

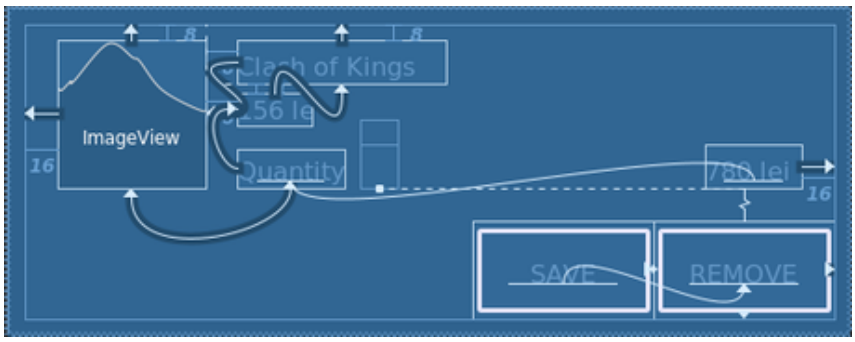
3.1. Prezentációs réteg

3.1.1. Felhasználói felület

A Bookta alkalmazás felhasználói felülete az MVVM tervezési mintát követi, amely a 3.1.2. fejezetben van részletesebben bemutatva. Az Android alkalmazások alapvető komponensei



12. ábra. Az Android alkalmazás szerkezeti diagrammja



13. ábra. Egy listaelem *ConstraintLayout*tal megvalósítva, ahol egymáshoz viszonyítva helyezkednek el az elemek

az *Activity*-k, amelyekhez egy-egy *XML* állomány társul. Az XML állomány tartalmazza a képernyőn megjelenő vizuális elemeket (menük, gombok, képek stb.), ezek méreteit és elhelyezkedését a képernyőn. Mivel az Android operációs rendszert használó mobileszközök képernyőmérete nagyon változó, ennek érdekében az alkalmazás fejlesztése során hangsúlyt kap a *responsive design* kialakítása, erre a megoldást a *ConstraintLayout* (13. ábra) használata jelenti.

Ennek a layoutnak a jelentősége, hogy minden elemnek kötelező módon rendelkeznie kell legalább két kapcsolattal (constraint), ami azt jelenti, hogy egy másik elem valamely oldalához van hozzárendelve. Ez azt eredményezi, hogy nem abszolút pozíciót kapnak az elemek, hanem egymáshoz viszonyítva jelennek meg a képernyőn. Ha megfelelő kapcsolatok jöttek létre, akkor elkerülhető az elemek összekavarodása vagy kicsúszása a felhasználói felületen, legyen az eszköz mérete kisebb, esetleg nagyobb, mint a tesztelő eszköz.

A Bookta felhasználói felületének központjában a `MainActivity` áll, amely elsősorban egy `NavigationDrawer`t tartalmaz, ami elősegíti a `Fragment`ek közötti navigálást. A fragment az activity-hez hasonló, kisebb logikai egység, amely a nézet egy önálló részét tartalmazza. Azért előnyös ezek használata, mert sok esetben nincs szükség az egész képernyő újrarajzolására, csak valamely része változik, így ezeket a részeket képviselik a fragmentek. Ez történik a `MainActivity` esetében is, amely tartalmazza többek között a `SearchFragment`-et, `CartFragment`-et, stb. Ezek pedig az említett `NavigationDrawer` segítségével váltogathtóak, amely tulajdonképpen az alkalmazás főmenüjeként szolgál. Ebből a menüből navigálhatunk további két Activity-re, amelyek a felhasználók bejelentkezését illetve regisztrációt biztosítják.

Az alkalmazás egyik legfontosabb nézete a `SearchFragment`, ahol könyvek listája található. Mivel a lista sok elemet tartalmaz, a `ListView` helyett `RecyclerView` [27] segítségével valósul meg. A `RecyclerView`, ahogyan a neve is mutatja, oly módon gyorsítja a folyamatot, hogy a képernyőről legörgetett listaelemet nem törli ki a memóriából, hanem újrahasznosítja úgy, hogy a régi komponens az újonnan érkező elem tartalmát kapja meg. Ezekről az elemekről eljuthatunk a `BookDetailActivity`-re, ahol megtekinthetünk minden

információt a könyvekről, és műveleteket végezhetünk velük (ld. 6. fejezet).

A megfelelő felhasználói élmény kialakítása érdekében alkalmazva lettek a Google által előírt *Material Design* [19] princípiumok, amelyek elősegítik a felhasználói felület megértését és könnyű használatát a felhasználók számára. A *Material Design* a Google által lett bevezetve annak érdekében, hogy egységesítse ezeket a princípiumokat és a fejlesztők számára megkönnyítse a felhasználói felületek tervezését. A Bookta projekten például *Floating Action Button* irányelvet követtük, ugyanis ezzel a gombbal érjük el minden képernyő nézetben a legfontosabb funkciókat. Például a `BookDetailActivity`-ben Floating Action Button segítségével helyezhetjük kosárba az illető könyvet, a `GroupOrderActivity`-ben új csoportos rendelést indíthatunk, illetve a főoldalon a szűrés funkció ezzel a gombbal érhető el. További design princípiumok is jelen vannak az alkalmazásban, ilyen például a `NavigationDrawer` jelenléte vagy éppen az elsődleges és másodlagos színek kiválasztása, amelyek minden képernyő nézeten használva vannak.

3.1.2. MVVM

A Bookta alkalmazás prezentációs rétege az MVVM (Model-View-ViewModel) [2] architekturális mintát használja. A ViewModel-ek (nézetmodell) olyan osztályok, amelyek felhasználói felülethez kapcsolódó adatokat tárolnak, megszabadítva ettől az Activity-ket. A nézetmodell bevezetése könnyebbé teszi a tesztelhetőséget, és átláthatóbbá válik a kód.

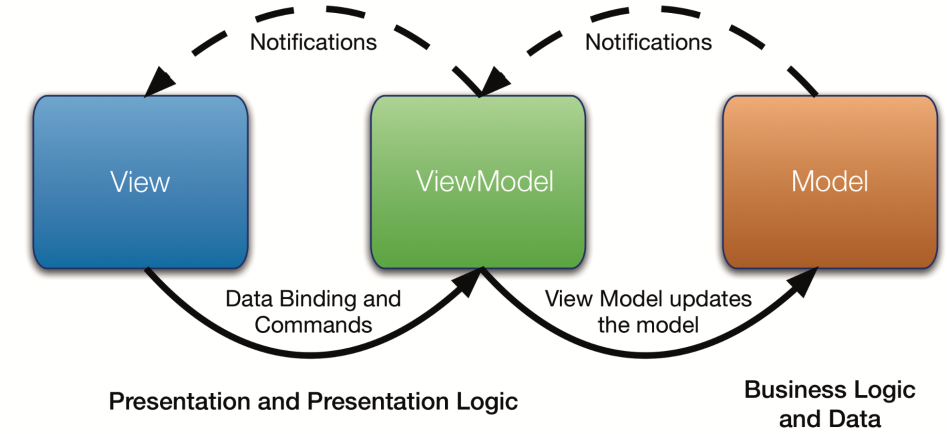
Minden központi entitásnak megfelelő use case-hez létrejön egy nézetmodell is (például `CartViewModel`). A nézetmodellekben található az adatok lekéréséért, kezeléséért, és a view-kon történő megjelenítésért felelős logika. A Bookta esetében *Activity-k, Fragment-ek illetve DialogFragment-ek* használják a nézetmodelleket. Ha a felhasználó interakcióba lép a nézettel, meghívódik a nézetmodell megfelelő metódusa, amely kérést küld a modellek fele. Ezután a view figyel a változásokra, és ha megérkezett a válasz értesül a nézetmodell megfigyelt attribútumain keresztül és megjeleníti az adatokat, ahogyan a 14. ábra [4] bemutatja.

3.1.3. Data Binding

A Data Binding [1] egy Android könyvtár, amely segít összekötni az ui komponenseket az adatmodellel. Ez a könyvtár lehetőséget ad, hogy elkerülhető legyen a boilerplate[5] kód írása. Ez azt jelenti, hogy e könyvtár bevezetésének következtében továbbá nincs szükség az Android környezetben megszokott `findViewById` metódusra, vagy a nézet manuális változtatására, ha változik az adat. A `findViewById` metódus abban segített, hogy összeköthetőek legyenek az

[4] Forrás: https://brainbeanapps.com/best-practices/designing-the-architecture-of-your-mobile-product-4-patters-to-choose-among-in-2018/

[5] Ismétlődő kódrészlet, amely nehezebbé teszi a kód megértését



14. ábra. Az MVVM tervezési minta működésének diagrammja

XML layout fájlokban definiált nézet elemek a Java vagy Kotlin attribútumokkal, és ezt meg kellett ismételni minden adattag esetében, ami redundáns kódot eredmenyézett. A modellek adatának változása esetén pedig manuálisan kellett átállítani a megfelelő nézet elem tartalmát is, ami szintén hosszabb, nehezen átlátható kódot eredményez.

A Data Binding esetén ezek másképpen vannak megoldva. Az előző fejezetben említett nézetmodellek tárolják és kezelik a modelleket osztályváltozók formájában. Ezek az attríbutumok *ObservableField* típusúak lesznek. Lehetőség van az XML fájlokban változókat deklarálni, így a layout fájlban is létrehozhatóak a nézetmodellek. Ez a következő módon valósítható meg:

```
<variable
    name="viewModel"
    type="edu.codespring.bookta.android.app.viewmodel.BookDetailViewModel" />
```

A példány létrehozása után hivatkozhatunk a nézetmodellben `ObservableField`-ként deklarált változókra, és itt történik meg a nézet összekötése a modellekkel. A hivatkozás előtt egy „@” jel jelzi, hogy az adott nézet elem figyeljen a nézetmodellben levő field változásaira és változás esetén frissítse a nézetet. Ugyanez elvégezhető fordítva, például egy szövegdoboz esetében az is megvalósítható, hogy a szövegdobozba történő változások frissítsék a modellt magát, ezesetben a „@=”-t használjuk hivatkozáskor. Mindkét esetre példa a 3. kódrészletben található.

E könyvtár segítségével kevesebb és átláthatóbb kóddal végezhető el a modell és az XML nézet összekötése, anélkül, hogy bármelyiket is manuálisan szükség legyen frissíteni.

3.2. Adatelérési réteg: Szerverrel történő kommunikáció

A szerverrel történő kommunikáció REST API-n keresztül valósul meg és az adatelérési rétegen van implementálva. A HTTP kérések fejlécének és törzsének összeállítását és a

```
<TextView
    android:text="@{viewModel.bookTitle}" />

<EditText
    android:text="@={viewModel.author}" />
```

3. kódrészlet. Ebben az esetben ha a `bookTitle` attribtútum megváltozik a nézetmodellben valami hatására, akkor a nézeten azonnal látható a változás. Valamint ha az `EditText`-be írunk szöveget, akkor a `author` nevű attribútum fogja követni a változásokat és frissül a beírt szövegre.

válasz parse-olását a *Retrofit* [28] könyvtár biztosítja. A könyvtár megkönnyíti az API kliens implementálását Android környezetben, és lehetővé teszi hogy Kotlin interfészek segítségével definiáljuk a kéréseket metódusok formájában (ld. lenti kódrészlet).

```
@GET("/books/{id}")
fun getBook(@Path("id") id: Long): Single<BookDetailDTO>
```

A metódusokat felannotálhatjuk annak függvényében, hogy milyen típusú kérést (`@GET`, `@POST`, `@DELETE`, stb.) szeretnénk küldeni és milyen formátumban (@FormUrlEncoded, @Json, stb.). Az alapértelmezett formátum a *JSON*, ezt a formátumot használja az adatok küldésére és fogadására a Bookta projekt. A kapott JSON adatoknak a Kotlin objektumba való deszerializációját, illetve objektumok JSON formátumba való átalakítását a *Gson* könyvtár végzi el. Ez a Retrofit Builder osztályán keresztül konfigurálható, ahol a *cél url*-t is meg kell adni, amelyre a kérést küldjük. A Bookta projektben a `NetworkModule` képviseli ezt az osztályt, ahol emellett a központi entitásoknak megfelelő Retrofit interfészekből példányok jönnek létre. Például a `RetrofitBookRestClient` interfészben találhatóak a könyvekkel kapcsolatos kérések. Ezeknek az interfészeknek az implementációi automatikusan generálódnak le a könyvtár segítségével.

A kérések előfeldolgozását, küldését és fogadását az `OkHttpClient` végzi egy interceptor segítségével. A kimenő kérések először az interceptorhoz jutnak, ami hozzácsatolja a felhasználó tokenjét (ha bejelentkezett a felhasználó) a kérés fejlécéhez, így a szerver be tudja azonosítani a kérés küldőjét, és eldöntheti, hogy jogosult-e az adott műveletre vagy sem. A vendég felhasználó funkcionalitásai elérhetőek token nélkül, viszont a további műveletek csak bejelentkezéssel és a token használatával hajthatóak végre.

3.3. Domain réteg

A domain réteg tartalmazza a központi entitásokat és az ezeket kezelő interfészeket, amelyekkel a prezentációs réteg műveleteket hajt végre. A központi entitások POJO[6]-knak

[6]Plain Old Java Object

felelnek meg, amelyek Kotlinban a következőképpen néznek ki:

```
data class Author(var id: Long, var name: String)
```

A fenti kódrészletben található deklaráció rendelkezik két adattaggal, ezeknek megfelelő getterekkel, setterekkel, illetve felülírja a `toString`, `hashCode` és `equals` metódusokat. Ezek elérése végett a kódban szereplő `data` kulcsszó használható.

Az Android kliens adatmodellje hasonló a szerveren definiált adatmodellel (ld. 2.1. fejezet). A központi entitások implementálják a `Parcelable` interfészt, amely arra szolgál, hogy az osztályokból létrejött objektumok átadhatóak legyenek fragmentek vagy activity-k között.

3.4. Adattárolás

A 3.2. fejezetben említett tokent el kell tárolnia az alkalmazásnak, hogy könnyen és gyorsan elérhető legyen bármely kérés küldése előtt. Ezért, ha sikeres volt a bejelentkezés, akkor az alkalmazás a *shared preferences*-be menti el a tokent. Ez a mentés `PreferenceHelper` osztály segítségével hajtódik végre, és az interceptorok mindig ellenőrzik, hogy van-e érvényes token a shared preferences-ben vagy sem. Ez a token meghatározott időn belül érvényét veszti, vagy kijelentkezés esetében törlődik.

3.5. Dependency injection

A dependency injection egy tervezési minta, amely alapján egy osztályban levő függőségek példányosítását egy külső rendszer vagy könyvtár végzi el. Ez a minta segít megteremteni egy könnyebben átlátható, újra felhasználható, illetve tesztelhető kódot.

Az Android Software Development Kit [32] alapértelmezetten nem támogatja a dependency injectiont, viszont vannak különböző könyvtárak, amelyek lehetőséget adnak a DI használatára. Ilyen könyvtár a *Dagger* [7], amelyet a Bookta projekt is használ.

A Dagger függőségként van jelen a projektben és annotációkkal szolgál, valamint használja az `android.javax` csomagban található annotációkat is. Egyik ilyen jelentős annotáció az `@Inject`, amely segítségével osztályváltozókat tudunk injektálni. Injektáláskor a függőségek példányosítását és beállítását a könyvtár automatikusan végzi el. Ennek megvalósulása érdekében létre kell hozni egy `@Module` annotációval ellátott osztályt. Ebben az osztályban végezzük el azokat a példányosításokat, amelyeket injektálni szeretnénk a későbbiekben. A példányokat visszatérítő metódusokat a `@Provides` annotációval annotáljuk fel, valamint ha azt szeretnénk, hogy csak egyetlen példány létezzen az adott osztályból, akkor használhatjuk a *@Singleton* annotációt is (ld. 4. kódrészlet). A presentation rétegen található egy `@Component` nevű annotációval ellátott `AppComponent` interfész. Amelyik osztály használni szeretné az

```
@Provides
@Singleton
fun createBookRestClient(retrofitBookRestClient: RetrofitBookRestClient):
    BookRestClient =
        BookRestClientImpl(retrofitBookRestClient)
```

4. kódrészlet. Az adatelérési rétegen található *@Module* annotációval ellátott *ApiModule* osztályban végzett példányosítás, amelyet presentation rétegen injektálunk

`@Inject` annotációt, annak meg kell hívnia az `AppComponent` interfész megfelelő metódusát, amelynek a saját magára mutató referenciát adja át.

3.6. RxJava

Az RxJava [22] könyvtár a szerver fele irányuló aszinkron kéréseket, illetve a szálak kezelését teszi lehetővé. Retrofit kérések során visszatérítünk egy `io.reactivex` csomagban levő `Observable` típusú adatfolyamot. Ennek az eredményére fel tudunk iratkozni, vagyis tudunk figyelni rá a `subscribe` metódus segítségével. Ezt a figyelést a nézetmodellekben tesszük meg. A `subscribe` metódus két callback függvényt vár paraméterként: az első akkor hívódik meg, ha a kérés sikeres eredménnyel tért vissza, a második pedig ha a kérés valamilyen hiba következtében nem tért vissza helyes eredménnyel (például kapcsolati probléma). Ezekben a callback függvényekben frissítjük a nézetmodell attribútumait, ha sikeres volt a lekérés, amiről rögtön értesülnek az activity-k vagy fragmentek. Hiba esetén egy kivételt kapunk válaszul, amelynek törzsében megtalálható a megfelelő hibaüzenet. Ezt a hibaüzenetet felhasználóbarát módon kiírjuk a képernyőre.

4. Scraper

A Python3-ban íródott scraper a Bookta projekt harmadik komponense, amely az adatbázis könyvekkel való feltöltéséért felel. Ezt web scraping [20]nek nevezett technika alkalmazása által oldja meg, amely megvalósításában a Scrapy [15] keretrendszer segített. A scraper jelenleg két online könyvesboltból tölti le a könyveket és küldi el azokat a központi szervernek. Mindez HTTP kérések küldése és a válasz HTML állományok feldolgozása által van megvalósítva. A HTML állományokból az adatok kinyerése XPath [13] (XML Path Language) azonosítók alapján történik. Az XPath segítségével XML dokumentumok egyes részeit lehet egyszerűen beazonosítani, amely az XML dokumentum fa csomópontjainak bizonyos szempontok szerinti kiválasztásával valósítható meg.

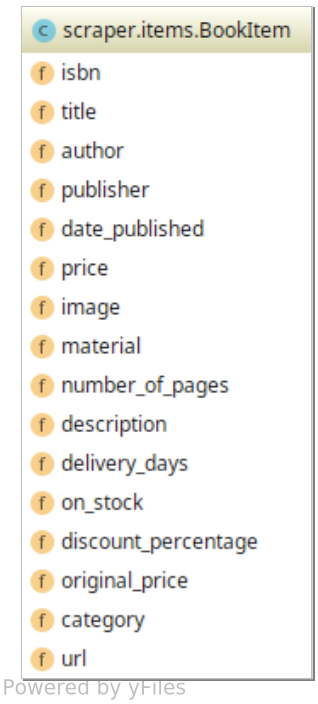
A projekt felépítése előre meghatározott a Scrapy keretrendszer által. A különböző üzleteket bejáró pókok (spider) a `spiders` csomagban találhatóak. Ezek egy-egy Python állomány által vannak képviselve és külön-külön futtathatóak. Az alkalmazásban található `service` csomag a szerverrel való kommunikációért felel. Ez REST hívásokon keresztül történik, követve a szerver által megszabott hitelesítési és engedélyezési szabályzatot (ld. 2.6 fejezet).

A két pókot reprezentáló osztály, a `BooklineSpider` és a `LibriSpider` felelősek a bookline, illetve libri online könyvesboltok könyveinek begyűjtésért. Ez a két osztály a `BaseSpider` leszármazottjai, amely tartalmazza az adatgyűjtéshez szükséges logikát, valamint lehetőséget ad annak bővítésére. A pókok általános felépítése egyszerűvé teszi hogy új boltokból letöltő pókokat hozzáadjunk a rendszerhez, illetve könnyen tudjuk módosítani a már meglévőket, ha az oldalaik szerkezete esetleg változna. Mindkét pókhoz tartozik egy-egy, az `xpath` csomagon belül elhelyezkedő konfigurációs fájl is, amely az adott weboldalhoz tartozó XPath azonosítókat tartalmazzák. Példa XPath azonosítóra: *//*[@id = 'book']/@data-price* - a libri könyvesboltban egy, könyv részleteinek oldalán az árat tartalmazó attribútum értékét adja meg. Az XPath jelentése: a HTML oldalon bárhol elhelyezkedő, bármilyen "book" id-jú *tag*, "data-price" attribútumának értéke. Az ehhez a példához tartozó HTML forráskód egy része ld. 5. kódrészlet.

A pókok betöltik az adott weboldal kezdő URL-jét, majd XPath alapján beazonosítják az

```
<html>
  <!--...-->
    <div id="book" class="product-page" data-id="2725757"
        data-name="A győztesek törvényei" data-price="2872">
    </div>
  <!--...-->
</html>
```

5. kódrészlet. Kódrészlet a libri oldalról



15. ábra. A BookItem osztály adattagjai

oldalon található elemeket, és addig navigálnak, amíg egy könyv részleteit tartalmazó HTML oldalra nem érnek. Mindezt HTTP kérések segítségével és a Scrapy-ben definiált callback mechanizmus által érik el. A könyvek tulajdonságait kinyerve egy `BookItem` (ld. 15. ábra) objektumot építenek fel, amely különböző pipeline-nak nevezett folyamatokon megy keresztül. Jelentősebb pipeline osztályok:

- `FieldFixerPipeline`, amely a `BookItem` objektum megfelelő mezőin végez különböző műveleteket. Például kitörli a whitespace-eket, szöveget számmá alkít, vagy logikai értéket határoz meg bizonyos szöveg alapján.
- `RestPipeline`, amely a service csomag `BooktaService` osztály metódusait meghívva elküldi a könyveket a szervernek. Ez a "/books" endpointra történő BookItem objektum JSON reprezentációját tartalamzó PATCH kérés által történik.
- `JsonPipeline`, ami kimenti lokálisan JSON formátumban a könyveket, ami megkönnyíti a szerver tesztelését, mivel nem kell minden adatbázis változás után újra kinbányászni azokat.

A scrapert kezdetleges kísérletek során mintegy 2000 könyv begyűjtésére alkalmaztuk. Jelenlegi állapotában lokális gépen fut, manuális indításra. Tervben van a komponens folyamatos integrációjának, valamint kitelepítésének a megvalósítása (ld. 5.3. fejezet), ami által a scraper periodikusan futna docker konténerben egy külső szerveren.

5. Felhasznált eszközök

Ebben a fejezetben a fejlesztés során használt fontosabb eszközök kerülnek bemutatásra, amelyek elősegítették és felgyorsították a projekt kialakulásának folyamatát.

5.1. Build és függőségkezelő eszközök: gradle és pip

A függőségek mendezsmentjét és a build folyamat kezelését mobil és szerver oldalon is a Gradle [34] biztosítja. Szerver oldalon a `java` plugin játszik fő szerepet, míg Android oldalon a `kotlin` és `android` pluginok. A Gradle a függőségeket egy központi repository-ból tölti le egy lokális könyvtárba, majd azokat felhasználja a build folyamán.

A scraper, mivel Pythonban íródott, nem szükséges különféle build eszköz használata, csupán a PIP [24] függőségmendedszment eszközre. A PIP a központi tárolóból tölti le a függőségeket a projekthez, amelyeket globálisan vagy virtuális környezeben tárol.

5.2. Verziókövető: git

Verziókövetéshez a csapat Gitet [18] használ, amely egy osztott verziókövető rendszer. Az alkalmazás három komponensét különböző Git repository-kban tároltuk. Branch-elési stratégiának a *git flow*-t [11] használjuk.

5.3. Folyamatos integráció és kitelepítés: GitLab CI és Docker

A forráskód külső verziója GitLab [3] által hosztolt tárolókon él. A kód tárolása mellett folyamatos integrációt és folyamatos kitelepítést is biztosít. Ezekhez a GitLab Docker [12] konténertechnológiát használ. A GitLab lehetőséget ad különböző folyamatok végrehajtására minden kódbázis változás esetén. Ezek a folyamatok elemezhetik, tesztelhetik, fordíthatják a kódot vagy bármilyen más műveletet végrehajthatnak rajta. Ezek a folyamatok sorba rendezhetőek, és hierarchia alakítható ki közöttük. Folyamatos integráció pipeline-ok jelenleg csak a mobil és a szerver komponensekhez vannak definiálva.

A mobilalkalmazás esetén a kompilálási folyamat fut le minden új push után, így elkerülhető a nem fordítható kód bekerülése a `develop` ágba.

A szerver esetén első fázisként lefut egy tesztelési folyamat, s ha ez sikeres volt, akkor lefut a build folyamat is. A `master` és `develop` ágakra végrehajtódik a deploy folyamat, amely az előző folyamat által készített futtatható állományból egy docker image-t hoz létre, valamint kitelepíti a GitLab konténer registry-jébe.

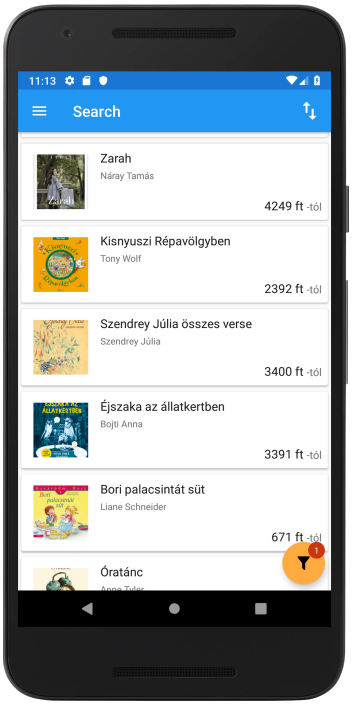
A kitelepített konténer elindítását a docker-compose [9] végzi. Segítségével egyszerre több mikroszervízt tudunk elindítani, melyek jelen esetben a *bookta-api* és a *bookta-database*.

6. Az Android kliens alkalmazás használata

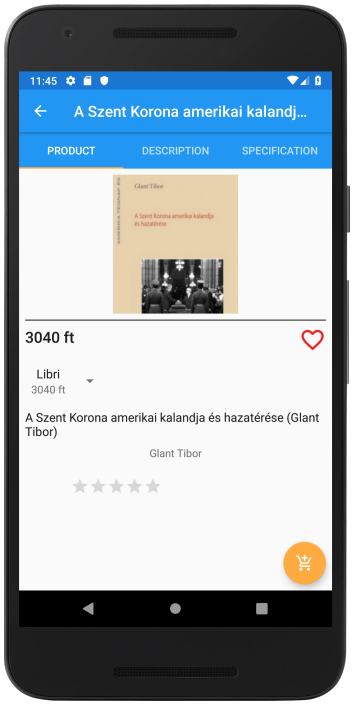
Ez a fejezet ismerteti az Android mobilalkalmazás funkcionalitásait, illetve ezek működését képernyőfotókkal és magyarázattal alátámasztva.

Az alkalmazás megnyitásakor a könyvek listája jelenik meg a felhasználónak (16. ábra). A képernyő jobb felső sarkában található egy ikon, amely segítségével a listában szereplő könyveket tudjuk rendezni ár vagy megjelenési dátum szerint. Ugyanezen nézeten található egy szűrő gomb a jobb alsó sarokban. Erre kattintva egy új nézet jelenik meg, amelyen összetett szűrést állíthatunk több tényező (pl. szerző, kategória, ár stb.) szerint. A szűrési feltételeket checkbox segítségével választhatjuk ki, ezekből egyszerre több is megadható. Ezután a beállított szűrést alkalmazhatjuk vagy törölhetjük a felső sávon lévő gombokkal. A szűrés alkalmazása után visszatérünk a lista nézethez, ahol már az általunk kiválasztott könyvek szerepelnek, és amelyeket szintén rendezhetünk.

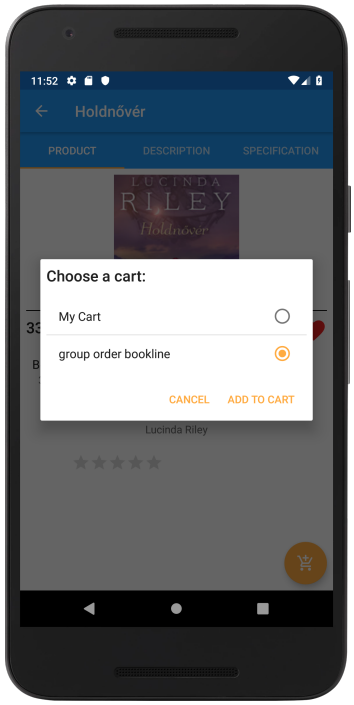
A listában szereplő könyvekre kattintva egy új nézet nyílik meg, amely az adott könyv részletes információit jeleníti meg (17. ábra). Ezen a nézeten adhatjuk hozzá a könyvet a kedvencekhez, valamint helyezhetjük kosárba. A kosárba helyezéskor az alkalmazás megkérdi, hogy melyik kosarunkba szeretnénk helyezni. Lehetőségünk van a saját kosarunkba tenni a terméket a *My Cart* (Saját kosár) opciót használva, illetve, ha csatlakozva vagyunk csoportos rendelésekhez, akkor azok között is választhatunk nevük alapján (18. ábra). Az alkalmazás főmenüje a képernyő bal oldaláról húzható be, de elérhetjük a menü gombon keresztül is.



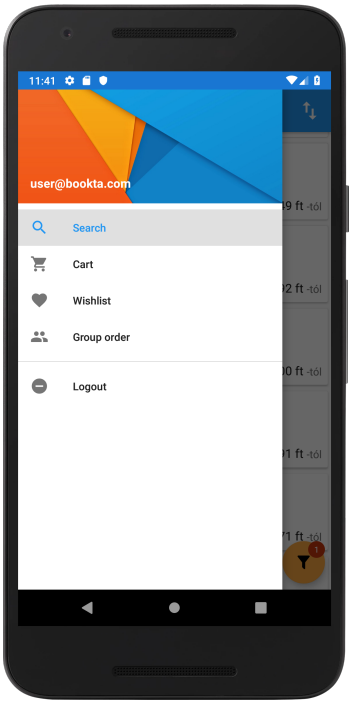
16. ábra. Könyvek listája (főoldal)



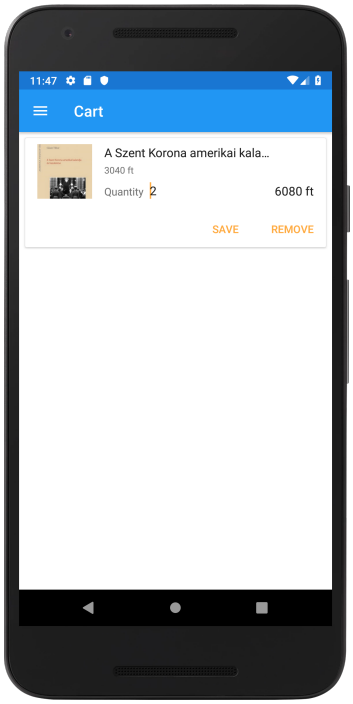
17. ábra. Könyv részleteinek megtekintése



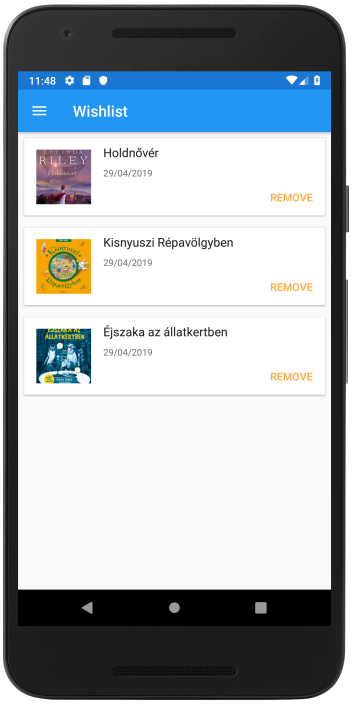
18. ábra. Könyv hozzáadása a kosárhoz



19. ábra. Főmenü



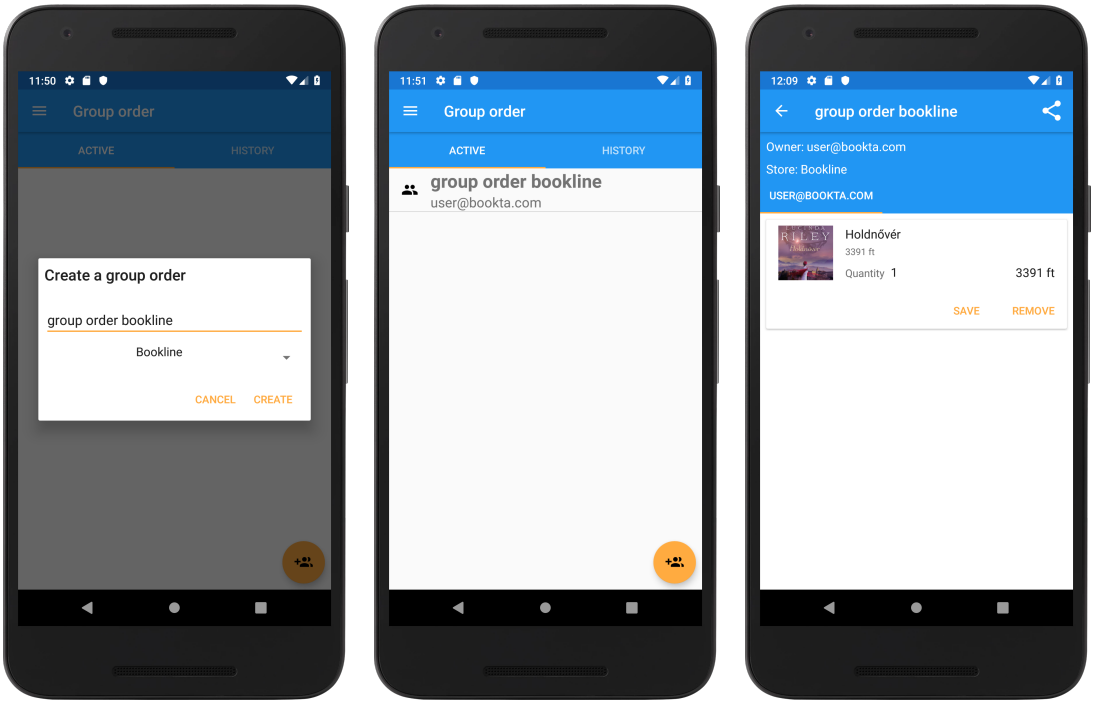
20. ábra. Saját kosár



21. ábra. Kedvencek

Bejelentkezett felhasználó esetében a menü a 19. ábrán látható módon néz ki. Az első menüpont a *Search* (Keresés), amely be lett mutatva a fentiekben. A keresés funkció működik ékezetek használata nélkül is és minden betű leütése után a legtalalóbb termékeket jeleníti meg. A második menüpontban a felhasználó saját kosarát tekintheti meg. A kosárban szerkeszthető a termékek mennyisége, illetve kitörölhető egy termék a kosárból (20. ábra). A következő elem a menüben a *Wishlist* (Kedvencek), amelyben a kedvencekhez adott könyvek listája jelenik meg (21. ábra). Lehetőség van egyes könyvek a kedvencekből való eltávolítására, illetve rákattintani a könyvre, ennek hatására a könyv részletes leírása nyílik meg. Ezen a nézeten is lehetőség van egy könyvet eltávolítani a kedvencek közül. A fő funkcionalitás a *Group Order* (Csoportos rendelés) menüpontból érhető el. Erre a menüpontra kattintva egy lista jelenik meg azokkal a csoportos rendelésekkel, amelyekhez a felhasználó csatlakozva van, és még nem lett lezárva. A lezárt rendelések is megtekinthetőek a *History* (Előzmények) tabra kattintva. Ha új csoportos rendelést szeretnénk indítani, azt megtehetjük a jobb alsó sarokban levő gomb segítségével. Ahhoz, hogy sikeres legyen a rendelés elindítása, kell adnunk egy nevet a rendelésnek és ki kell választanunk, hogy melyik üzlettől szeretnénk rendelni (22. ábra). Amint létrehoztuk a rendelést, megjelenik a saját listánkban, mint aktív csoportos rendelés, ahogyan a 23. ábrán láthatjuk. Ha erre az elemre rákattintunk, akkor eljutunk a csoportos rendelés részletes nézetéhez. Itt minden információt megtekinthetünk a rendelésről, többek között a kezdeményező nevét vagy a csatlakozott felhasználókat. A rendeléshez csatlakozott felhasználók megtekinthetik egymás kosarát, de csak a sajátjukat szerkeszthetik (24. ábra).

A csoportos rendelés létrehozása után a felhasználók csatlakozása a következő módon



22. ábra. Csoportos rendelés készítése

23. ábra. Aktív csoportos rendelések listája

24. ábra. Csoportos rendelés részletei

történik: a rendelés részletes nézetében a jobb felső sarokban található egy gomb, amelyet megnyomva az alkalmazás a vágólapra másolja a rendelés egyedi linkjét. Ha ezt a linket Android készülékről nyitjuk meg, akkor bevezet a Bookta alkalmazásba és automatikusan csatlakoztatja a felhasználót az adott rendeléshez. Ezután az újonnan érkezett felhasználó is adhat hozzá termékeket ehhez a rendeléshez, viszont csak abból az üzletből, amelyhez a rendelés kapcsolódik. A kosárhoz való hozzáadás a fennebb bemutatott módon történik. A rendelés lezárása utána az kezdeményező egy e-mailben értesül a megrendelni kívánt könyvek információiról, beleértve a linket amelyen keresztül elérheti.

A menüben található *Logout* (Kijelentkezés) gombbal ki tudunk jelentkezni az alkalmazásból. Ennek következtében csak a *Search* menüpont lesz elérhető, illetve megjelenik a *Login* (Bejelentkezés) és a *Register* (Regisztráció) lehetőség is. A regisztráció az e-mail cím és egy erős[7] jelszó megadásával történik. Sikeres regisztráció esetén a felhasználó azonnal be tud jelentkezni .

[7]Tartalmaznia kell legalább egy kisbetűt, nagybetűt, illetve számjegyet

Következtetések és továbbfejlesztési lehetőségek

Jelen dolgozatban bemutattuk a Bookta alkalmazás egy prototípusát, a felhasznált technológiákkal és a megvalósítás részleteivel együtt. Jelenleg az alkalmazás online könyvrendelő platformokról gyűjt össze könyveket, valamint lehetőséget ad a felhasználónak, hogy egy egységes felületen böngésszenek közöttük. A könyveket a kedvencek listájához lehet adni, illetve minden felhasználó rendelkezik saját kosárral és indíthat csoportos rendelést amelyet megoszthat más felhasználókkal. A kosárba helyezett könyvek linkjeit megkapja a rendelés indítója e-mail formájában.

Mind a kliens, mind a szerver követi a *clean architecture* mintákat, melyek a kódot könnyen bővíthetővé és átláthatóvá teszik. Ennek köszönhetően az alkalmazás továbbfejleszthető és skálázható újabb funkcionalitásokkal rövid idő alatt. A fejlesztés során felmerültek a következő továbbfejlesztési lehetőségek:

- elkészíteni egy webes platformot, amely menedzser felületként szolgálna; ez kisebb könyves üzleteknek lenne megfelelő hely termékeik megosztására és árusítására a Bookta alkalmazáson keresztül;
- kibővíteni az alkalmazást a *store manager* szerepkörrel, aki a webes felületen vinné fel saját üzletének a könyveit, így minden rendelés lezárásakor ő is értesülne a megrendelt termékekről;
- kibővíteni az Android kliens alkalmazást egy rendeléskor vagy regisztrációkor kitöltendő űrlappal, amellyel támogatnánk a direkt rendelést kisebb könyvesboltokból;
- a regisztrációt lehetővé tenni egyéb OAuth2-t támogató platformokon keresztül (pl. Facebook, Google);
- az Android alkalmazás többnyelvűsítése;
- grafikonok készítése az árak változásáról egy adott terméknél az idő függvényében;
- értesítést küldeni a felhasználónak, ha a kedvenc könyvei közül valamelyiknek változott az ára;
- kibővíteni az alkalmazást más, csoportosan rendelhető termékekkel (pl. sportruházat).

Hivatkozások

[1] *Android Architecture Components: DataBinding - Dependent Properties*. 2019. URL: https://medium.com/halcyon-mobile/android-architecture-components-databinding-dependent-properties-6e8eba6c8b13 (utolsó elérés dátuma: 2019. ápr. 30.)

[2] *Android Architecture Patterns Part 3: Model-View-ViewModel*. 2019. URL: https://medium.com/upday-devs/android-architecture-patterns-part-3-model-view-viewmodel-e7eeee76b73b (utolsó elérés dátuma: 2019. ápr. 30.)

[3] J. van Baarsen. *GitLab Cookbook*. EBL-Schweitzer. Packt Publishing, 2014. ISBN: 9781783986859. URL: https://books.google.ro/books?id=ANIGBgAAQBAJ.

[4] C. Bihis. *Mastering OAuth 2.0*. Packt Publishing, 2015. ISBN: 9781784392307. URL: https://books.google.ro/books?id=HjflCwAAQBAJ.

[5] G. Blokdyk. *Json Web Token*. CreateSpace Independent Publishing Platform, 2018. ISBN: 9781985676466. URL: https://books.google.ro/books?id=cy9rtAEACAAJ.

[6] I. Cosmina et al. *Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools*. Apress, 2017. ISBN: 9781484228081. URL: https://books.google.ro/books?id=N-U5DwAAQBAJ.

[7] *Dagger: dependency injection framework*. 2019. URL: https://google.github.io/dagger/ (utolsó elérés dátuma: 2019. ápr. 30.)

[8] M. Deinum et al. *Pro Spring MVC: With Web Flow*. Books for professionals by professionals. Apress, 2012. ISBN: 9781430241560. URL: https://books.google.ro/books?id=fzpgOZLyWpwC.

[9] *Docker compose: tool for defining and running multi-container Docker applications*. 2019. URL: https://docs.docker.com/compose/ (utolsó elérés dátuma: 2019. máj. 1.)

[10] Steve Peterson Emmanuel Bernard. *JSR 303: Bean Validation*. 2009.

[11] *Git flow*. 2019. URL: https://nvie.com/posts/a-successful-git-branching-model/ (utolsó elérés dátuma: 2019. máj. 1.)

[12] S. Goasguen. *Docker Cookbook: Solutions and Examples for Building Distributed Applications*. O'Reilly Media, 2015. ISBN: 9781491919774. URL: https://books.google.ro/books?id=C8XeCgAAQBAJ.

[13] M. Kay. *XPath 2.0 Programmer's Reference*. Programmer to programmer. Wiley, 2004. ISBN: 9780764569104. URL: https://books.google.ro/books?id=Xw3%5C_tzEJVEwC.

[14] M. Keith, M. Schincariol, és J. Keith. *Pro JPA 2: Mastering the JavaTM Persistence API*. Books for professionals by professionals. Apress, 2011. ISBN: 9781430219576. URL: https://books.google.ro/books?id=UHobxmgB714C.

[15] D. Kouzis-Loukas. *Learning Scrapy*. Packt Publishing, 2016. ISBN: 9781784390914. URL: https://books.google.ro/books?id=EF8dDAAAQBAJ.

[16] A. Leiva és Lean Publishing. *Kotlin for Android Developers: Learn Kotlin the Easy Way While Developing an Android App*. CreateSpace Independent Publishing Platform, 2016. ISBN: 9781530075614. URL: https://books.google.ro/books?id=cdT1jwEACAAJ.

[17] J. Linwood és D. Minter. *Beginning Hibernate*. Apresspod Series. Apress, 2010. ISBN: 9781430228509. URL: https://books.google.ro/books?id=CW8mLHrLWnUC.

[18] J. Loeliger. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, 2009. ISBN: 9780596551391. URL: https://books.google.ro/books?id=e9FsGUHjR5sC.

[19] K. Mew. *Learning Material Design*. Packt Publishing, 2015. ISBN: 9781785288715. URL: https://books.google.ro/books?id=tyDlCwAAQBAJ.

[20] R. Mitchell. *Web Scraping with Python: Collecting Data from the Modern Web*. O'Reilly Media, 2015. ISBN: 9781491910252. URL: https://books.google.ro/books?id=7z%5C_fCQAAQBAJ.

[21] P.B. Monday. *Web Service Patterns: Java Edition*. Books for professionals by professionals. Apress, 2008. ISBN: 9781430207764. URL: https://books.google.ro/books?id=8RWcvRgPTV4C.

[22] I. Morgillo. *RxJava Essentials*. Community experience distilled. Packt Publishing, 2015. ISBN: 9781784393571. URL: https://books.google.ro/books?id=cuiuCQAAQBAJ.

[23] A.E. Nascimento. *OAuth 2.0 Cookbook: Protect your web applications using Spring Security*. Packt Publishing, 2017. ISBN: 9781788290630. URL: https://books.google.ro/books?id=OhtKDwAAQBAJ.

[24] *Pip: the package installer for Python*. 2019. URL: https://pypi.org/project/pip/ (utolsó elérés dátuma: 2019. máj. 1.)

[25] M. Pollack et al. *Spring Data: Modern Data Access for Enterprise Java*. O'Reilly Media, 2012. ISBN: 9781449331887. URL: https://books.google.ro/books?id=DeTO4xbC-eoC.

[26] D. Rajput. *Spring 5 Design Patterns: Master efficient application development with patterns such as proxy, singleton, the template method, and more*. Packt Publishing, 2017. ISBN: 9781788299596. URL: https://books.google.ro/books?id=ExlKDwAAQBAJ.

[27] *RecyclerView. Android API Reference.* 2019. URL: https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html (utolsó elérés dátuma: 2019. ápr. 30.)

[28] *Retrofit: A type-safe HTTP client for Android and Java*. 2019. URL: https://square.github.io/retrofit/ (utolsó elérés dátuma: 2019. ápr. 30.)

[29] M. Richards. *Software Architecture Patterns: Understanding Common Architecture Patterns and when to Use Them*. O'Reilly Media, 2015. ISBN: 9781491924242. URL: https://books.google.ro/books?id=ZLYtuwEACAAJ.

[30] C. Scarioni. *Pro Spring Security*. Expert's voice in Spring. Apress, 2013. ISBN: 9781430248194. URL: https://books.google.ro/books?id=4CtJE6fVergC.

[31] *Springfox Reference Documentation*. 2019. URL: https://springfox.github.io/springfox/docs/current/ (utolsó elérés dátuma: 2019. máj. 1.)

[32] J. Steele és N. To. *The Android Developer's Cookbook: Building Applications with the Android SDK*. Developer's Library. Pearson Education, 2010. ISBN: 9780132464567. URL: https://books.google.ro/books?id=Y4JR2yI2Fo0C.

[33] *Swagger 2.0 Specification*. 2019. URL: https://swagger.io/docs/specification/2-0/basic-structure/ (utolsó elérés dátuma: 2019. máj. 1.)

[34] B. Varanasi. *Introducing Gradle*. Apress, 2015. ISBN: 9781484210314. URL: https://books.google.ro/books?id=%5C_kJECwAAQBAJ.

[35] B. Varanasi és S. Belida. *Spring REST*. Apress, 2015. ISBN: 9781484208236. URL: https://books.google.ro/books?id=2GInCgAAQBAJ.

[36] C. Walls. *Spring Boot in Action*. Manning Publications, 2016. ISBN: 9781617292545. URL: https://books.google.ro/books?id=9CiPrgEACAAJ.