

ETDK-dolgozat

Baksa Andrea

Szabó Adrienn-Barbara

XXVIII. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)

Informatika II.: innovatív számítástechnikai termékek, alkalmazások szekció

Kolozsvár, 2025. május 15–18.

StoryGraph

Közösségi történetírás és nemlineáris narratívák vizuális kezelése



Szerzők:

Baksa Andrea

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Szabó Adrienn-Barbara

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

dr. Simon Károly, egyetemi adjunktus,

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Both Bence, szoftverfejlesztő,

Codespring

Zsebe Ákos, szoftverfejlesztő,

Codespring



2025. MÁJUS 15-18.

KOLOZSVÁR

XXVIII. ERDÉLYI TUDOMÁNYOS DIÁKKÖRI KONFERENCIA REÁL- ÉS HUMÁNTUDOMÁNYOK

CONFERINȚA ȘTIINȚIFICĂ STUDENTEASCĂ DIN TRANSILVANIA, EDIȚIA A 28-A
28TH TRANSILVANIA STUDENTS' SCIENTIFIC CONFERENCE
ȘTIINȚE REALE ȘI UMANISTE // FORMAL AND EMPIRICAL SCIENCES

StoryGraph: Közösségi történetírás és nemlineáris narratívák vizuális kezelése

Baksa Andrea, Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Szabó Adrienn-Barbara, Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

dr. Simon Károly, egyetemi adjunktus,
Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar
Both Bence, szoftverfejlesztő,
Codespring
Zsebe Ákos, szoftverfejlesztő,
Codespring

Az írói és olvasói közösségekben egyre nagyobb az igény a közösségi történetírásra, valamint nemlineáris történetek kezelésére. Az átlagos szerkesztői és olvasói platformok lineáris tartalmak kezelésére épülnek, nem adnak lehetőséget alternatív történetvezetési formákra. Az adatmennyiség növekedésével a szerkesztés és olvasás is egyre bonyolultabbá válik. Egy idősíkokban vagy cselekményszálakban elágazó, több szerkesztő által szerkesztett történet hatékony kezelése egy jól megtervezett és könnyen használható rendszert igényel.

A StoryGraph projekt keretén belül egy olyan alkalmazás került megvalósításra, amely ezeket az igényeket kiszolgálja. Az alkalmazáson belül a felhasználónak lehetősége van több szálú, fastruktúrájú történetek vizuális kezelésére, valamint más szerkesztőkkel való hatékony együttműködésre. Az olvasók számára elérhetőek a történetek szerkezetét ábrázoló nézetek, amelyek javítják az olvasási élményt és megkönnyítik a nemlineáris történetek fejezetei közötti navigációt.

A kezelőfelület a felhasználó számára elérhető Android és iOS platformokon, valamint egy weboldalon keresztül, melyeket egy központi szerver szolgál ki. A dolgozat további része bemutatja ezek felépítését, valamint részletezi az alkalmazás működését és funkcionalitását.

Tartalomjegyzék

Bevezető	1
1. A StoryGraph projekt	2
1.1. Funkcionalitások	2
1.2. Rendszer architektúra	3
2. Szerver	5
2.1. Funkcionalitások	5
2.1.1. REST API	5
2.1.2. Adatbázis-kezelés és modellek	6
2.1.3. Bejelentkezés és autorizáció	7
2.1.4. Képfeltöltés és médiakezelés	8
2.2. Architektúra	9
2.3. Alkalmazott technológiák	10
3. Mobil- és webalkalmazás	11
3.1. Alkalmazás architektúra	11
3.2. Funkcionalitások	12
3.2.1. Bejelentkezés	12
3.2.2. Főoldal	13
3.2.3. Saját sztorik	13
3.2.4. Közös szerkesztés és sztoribeállítások	14
3.2.5. Szerkesztő	14
3.2.6. Olvasói nézet	16
3.2.7. Mentett sztorik	16
3.2.8. Beállítások	16
3.3. Flutter	21
3.3.1. A Dart nyelv	22
3.3.2. Flutter belső architektúra	22
3.3.3. Widgetek és UI komponensek	23
3.3.4. Konfiguráció	23
3.4. BLoC	23

3.4.1. BLoC osztályok	24
3.4.2. BLoC-alapú widgetek	24
3.5. Hálózati réteg	24
3.5.1. Dio	25
3.5.2. Retrofit	25
3.6. Dependency injection	26
3.7. Nyelvi támogatás	26
3.8. Témázás	27
3.9. Globális hibakezelés	28
4. DevOps	29
4.1. Gitlab CI/CD, Docker	29
4.2. Kubernetes, Helm	29
5. Fejlesztési eszközök és módszerek	30
5.1. Scrum	30
5.2. Git és GitLab	30
5.3. Figma	31
Következtetések és továbbfejlesztési lehetőségek	32

Bevezető

A StoryGraph projekt célja egy olyan alkalmazás megalkotása volt, amely az írói és olvasói közösségek számára nyújt megoldást a közösségi történetírás lehetőségének biztosítására, valamint a nemlineáris narratívák hatékony kezelésének lehetővé tételére. Az alkalmazás lehetőséget ad a felhasználóknak, hogy egy platformon közösen dolgozzanak egy-egy történeten, miközben a történet szálai rugalmasan, több irányban fejlődhetnek, így nemcsak az írási, hanem az olvasási élményt is gazdagítja a megoldás.

"A történetmesélés mindig is alapvető része volt az emberi kifejezésnek. Már az első barlangrajzok és az ókori civilizációk epikus költeményei is lenyűgözték és szórakoztatták a közönséget. Azonban, ahogy a társadalom fejlődik, úgy a történetmesélés módja is változik." [1]

Bár a nemlineáris történetmesélés nem új keletű dolog, a modern kiadók és írók világában egyre felkapottabbá és népszerűbbé válik - új lehetőségeket nyit a történetek gazdagítására, és a kreativitás kibontakoztatására. Ugyanakkor a nemlinearitás ugyancsak megjelenik társszerzőknél, vagy nagyobb közösség általi írói kollaborációk során, hiszen a különböző forrásokból érkező tartalmak gyakran nem egy egységes idővonal mentén szerveződnek, hanem párhuzamosan, eltérő nézőpontokból vagy eseménysorrendekben. Ezeknek a tartalmaknak a következetes és jól strukturált kezelése állandó kihívást jelent, ebben segíthet egy olyan rendszer, amely képes támogatni az eltérő narratív szálak logikus összekapcsolását és vizuális átláthatóságát. Ezért született meg a StoryGraph.

A projekt alapötlete 2024-ben született meg Kolozsváron, a Codespring¹ mentorprogram keretében. A fejlesztés a programhoz tartozó nyári szakmai gyakorlat során indult el, ahol a dolgozat szerzői elkezdtek kidolgozni az alkalmazás alapjait. Később, az egyetemi csoportos projekt tantárgy keretében, a platform újabb funkciókkal bővült és további fejlesztési fázisokon ment keresztül, melyekben a szerzőkön kívül Kántor Rolland-Róbert, Kovács Kristóf, Poszet Kinga és Rigo Kristóf-Richard is közreműködtek. A szerzők ezúton szeretnék kifejezni köszönetüket mentoraiknak és szakmai koordinátoruknak, akik támogatásukkal, szakértelmükkel és folyamatos iránymutatásukkal meghatározó szerepet játszottak a projekt sikeres megvalósításában.

¹<https://edu.codespring.ro/>

1. A StoryGraph projekt

A dolgozat a továbbiakban részletesen bemutatja a StoryGraph projekt felépítését, főbb komponenseit, az alkalmazás fejlesztésének folyamatát, valamint annak különböző felhasználási lehetőségeit.

1.1. Funkcionalitások

A felhasználási módok két főbb csoportba sorolhatóak. Az első kategóriába tartoznak a történetek létrehozására és szerkesztésére vonatkozó funkcionálisok. A második csoport a már megírt történetek megjelenítésével és mentésével kapcsolatos lehetőségeket foglalja magában.

Az íráshoz kapcsolódó funkciók a következők:

- Történetek létrehozása, törlése és módosítása: a felhasználók létrehozhatják és törölhetik saját történeteiket, valamint különböző beállításokat adhatnak meg. Például meghatározhatják, hogy ki olvashatja el, illetve a megtekintés ingyenes legyen-e vagy sem.
- Fejezetek létrehozása, szerkesztése és törlése: bármely saját történethez új fejezetek adhatóak hozzá, amelyek az alkalmazáson belül szerkeszthetők és törölhetők.
- Szerkesztők meghívása, letiltása és feloldása: a közösségi történetírást támogatva lehetőség van társszerkesztők meghívására, akik kizárólag az előző pontban említett, fejezetekhez kapcsolódó műveletek végrehajtására jogosultak. A főszerkesztő jogosult megvonni a szerkesztési jogokat a meghívott szerkesztőktől, illetve szükség esetén visszaadni azokat.

Az olvasáshoz kapcsolódó funkciók a következők:

- Sztorik kategóriák szerinti listázása és keresése, cím vagy író alapján: a főoldalon láthatóak a már létrehozott sztorik különféle kategóriák szerint csoportosítva. A felhasználók rákereshetnek egy történetre cím vagy szerkesztő alapján.
- Fejezetek megjelenítése gráfnézetben: egy történet kiválasztásakor a hozzá tartozó fejezetek gráfstruktúrában kerülnek megjelenítésre. Az egymást követő részek egy fastruktúrát ábrázolnak, így elősegítve az olvasónak a könnyebb átláthatóságot.

- Történetek mentése: a rendszer lehetőséget biztosít arra, hogy a felhasználó könyvjelzőkkel lássa el a számára fontos tartalmakat, ezzel megkönnyítve azok későbbi elérését.
- Olvasási haladás nyomon követése: a rendszer nyomon követi az olvasás előrehaladását, és eltárolja az elolvasott fejezeteket, ezzel támogatva a felhasználót abban, hogy később könnyen folytathassa az olvasást ott, ahol korábban abbahagyta.

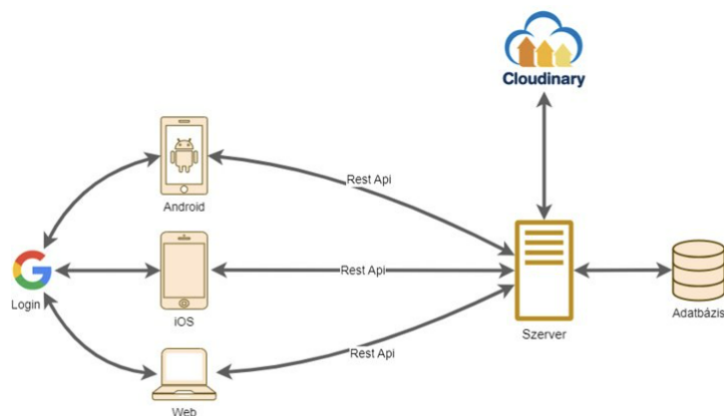
A fent említett funkcionalitásokon kívül más, felhasználói élményt növelő elemek is rendelkezésre állnak:

- Nyelv beállítás: az alkalmazás magyar és angol nyelven is elérhető.
- Témázás: a felhasználók szabadon kiválaszthatják a számukra megfelelő megjelenési módot, mivel az alkalmazás támogatja mind a világos, mind a sötét témát.

1.2. Rendszer architektúra

A StoryGraph célja, hogy platformfüggetlen módon egységes és kényelmes felhasználói élményt nyújtson. Ehhez hozzájárul egy egyszerű és praktikus bejelentkezési folyamat.

Ennek érdekében a belépés Google-alapú hitelesítéssel történik, amelyet a felhasználók megtehetnek Android, iOS vagy Webes felületről. A három eszköz támogatása indokolt, ugyanis az írók számára sokkal egyszerűbb a tartalomgyártás laptopról. Ezzel ellentétben az olvasók szempontjából praktikusabbak a kisebb, hordozható eszközök, mint például a telefon vagy táblagép. Ezek lehetővé teszik a kényelmes olvasást bárholnan, bármikor.



1. ábra. Architektúra

A már bejelentkezett eszközök Rest API segítségével kommunikálnak a szerverrel. A szerver az információkat az adatbázisba menti el, míg a képeket a Cloudinary felhőalapú tároló tárolja el.

Az architektúra diagram megtekinthető az 1. ábrán.

2. Szerver

A StoryGraph projektben a szerver tölti be a központi szerepet: kezeli az üzleti logikát, biztosítja az adatok perzisztens tárolását, valamint felel a kliensekkel való kommunikációért.

2.1. Funkcionalitások

Az alábbi fejezet a szerver funkionalitásainak megvalósítását ismerteti, egyaránt kitérve a működési és technikai részletekre.

2.1.1. REST API

A szerver elérése REST API²-n keresztül történik, melynek implementációja a Spring keretrendszerrel valósult meg.

A szükséges funkionalitás biztosítása érdekében a függőségek közé bekerült az *org.springframework.boot:spring-boot-starter-web* csomag. A vezérlő osztályok `@RestController` annotációval vannak ellátva, amely lehetővé teszi a HTTP-kérések megfelelő kezelését. Az osztályok metódusai a különböző `@Mapping` annotációknak köszönhetően képesek a meghatározott útvonalakról érkező kéréseket fogadni, feldolgozni és válaszolni. Az 1 kódrészletben egy olyan metódus látható, amely a fejezetek lekérését valósítja meg. Ez a metódus a `/api/chapters/12` útvonalon érkező GET kérést fogadja, és válaszként a 12-es azonosítóval rendelkező fejezet szövegét küldi vissza. Az előbb említett útvonalon (`/api/chapters/{chapterId}`) egy DELETE kérés is elérhető, amely a megadott azonosítójú fejezet törlésére szolgál.

A biztonság és a rétegek közötti szétválasztás érdekében az adatok Data Transfer Object (DTO)³ típusú objektumokon keresztül érkeznek be, illetve távoznak a rendszerből.

```
@GetMapping(value = "{chapterId}")
public ResponseEntity<ChapterResponseDto> chapterDetailsForReaders(
    @PathVariable("chapterId") String chapterId
) {
    return new ResponseEntity<>(chapterService.getById(chapterId), HttpStatus.OK);
}
```

1. kódrészlet. `GetMapping`

²<https://restfulapi.net/>

³<https://www.baeldung.com/java-dto-pattern>

2.1.2. Adatbázis-kezelés és modellek

A StoryGraph projekt elágazó történetek olvasására és írására alkalmas platform. Olyan adatbázisra van szükség, amely hatékonyan képes nagy mennyiségű szöveges tartalmat tárolni. Emellett elengedhetetlen, hogy az egyre inkább elágazó fejezetek közötti kapcsolatok egyszerű és strukturált módon rögzíthetők legyenek, amely gráfok segítségével könnyen és hatékonyan megoldható. Mindezen követelmények alapján az ArangoDB [2] adatbázis került kiválasztásra, mivel ez egy multimodell adatbázis, amely támogatja a gráf- és dokumentum-alapú megközelítést.

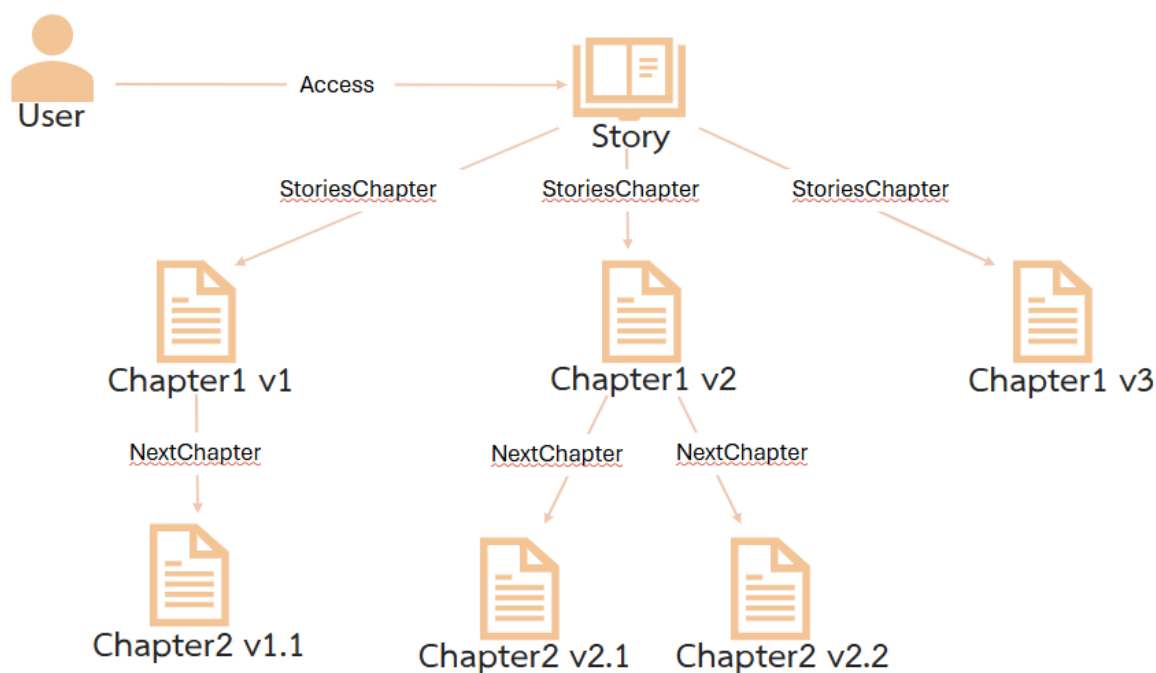
Az adatbázis két függőség segítségével került integrálásra, amelyek a Maven központi tárolójában elérhetőek:

- `com.arangodb:arangodb-spring-data`: biztosítja az automatikus leképezést Java osztályok és adatbázis-dokumentumok között, valamint lehetővé teszi az adatbázisműveletek generálását.
- `com.arangodb:arangodb-spring-boot-starter`: ez a modul a Spring Boot automatikus konfigurációs mechanizmusát használja ki, ezáltal leegyszerűsíti az adatbázis konfigurációt.

Az ArangoDB gráfalapú struktúrája két típusra épül: csomópontokra (node), amelyek az entitásokat reprezentálják, valamint élekre (edge), amelyek a csomópontok közötti kapcsolatokat írják le.

Az adatbázisban 3 féle csomópont létezik. A User (felhasználó) entitás tartalmazza a felhasználó nevét és e-mail címét. A Story (sztori) csomópontban a történet címe, rövid leírás, borítókép, keletkezési dátum, valamint utolsó módosítás dátuma kerül tárolásra. A Chapter (fejezet) entitás a fejezet szövegét, létrehozásának dátumát, és utolsó módosításának dátumát tartalmazza.

A csomópontokat 3 típusú éllel lehet összekötni. Az Access (hozzáférés) él a felhasználókat köti a sztorikhoz. Az élben a kiindulópont és a végpont azonosítóin kívül bizonyos jogosultságok vannak eltárolva, mint például a tulajdonos, szerkesztő, mentett történet. A StoriesChapter (sztorik fejezetei) él a sztorik és az első fejezetek között van. Ezzel az éllel tudja a felhasználó lekérni egy történet első fejezeteit. A NextChapter (következő fejezet) él két fejezet között van. Ez az él határozza meg a fejezetek közötti sorrendet.



2. ábra. Adatmodell

2.1.3. Bejelentkezés és autorizáció

A rendszer biztonsága és megbízhatósága érdekében a bejelentkezés/regisztrálás és a hitelesítés a Spring Security által támogatott JWT (JSON Web Token) tokenkezelésre épül.

Bejelentkezési vagy regisztrálási kérés érkezése esetén a szerver egy, a bejelentkezni kívánt felhasználó adatait tartalmazó adatátviteli objektumot kap, melyből kinyer egy felhasználónevet, egy email címet, valamint egy profilképet tartalmazó URL-t.

A bejelentkezés logikáját az AuthService végzi. Amennyiben a felhasználó még egyszer sem volt bejelentkezve, a szerver elmenti az érkezett adatokat az adatbázisba. Minden bejelentkezés esetén a szerver generál egy JWT token (2. kódrészlet), amelyet visszaküld a válaszban. A bejelentkezéshez használt endpointon kívüli további endpointok eléréséhez a felhasználónak szüksége van egy Bearer tokenre, amely a kérés fejlécéhez csatolódik. Ennek jelenlétét és érvényességét a rendszer a szerveroldalon ellenőrzi. A JWT formátum lehetővé teszi, hogy a szerver állapotmentesen működjön, mivel a bejelentkezett felhasználóval kapcsolatos információkat így nem szükséges tárolni. A token alapján könnyedén kinyerhető az aktuális felhasználó (3. kódrészlet).

A JWT tokenek generálásához és validálásához szükség volt az *org.springframework.boot:spring-boot-starter-security*, valamint az *org.springframework.boot:spring-boot-starter-oauth2-resource-server* függőségek behúzására,

amelyek biztosítják a biztonsági infrastruktúrát és a tokenkezeléshez szükséges eszközöket. Az autentikációhoz, valamint a kéreşfeldolgozó filter működéséhez továbbá elengedhetetlen volt az *org.springframework.boot:spring-boot-starter-web* használata.

```
@Override
public String generateJwtToken(User user) {
    JwtClaimsSet claims = JwtClaimsSet.builder()
        .issuedAt(Instant.now())
        .subject(user.getUserName())
        .claim("email", user.getEmail())
        .build();
    return jwtEncoder.encode(JwtEncoderParameters.from(claims)).getTokenValue();
}
```

2. kódrészlet. JWT tokengenerálás

```
public Optional<User> findUser(String authorizationHeader) {
    String token = authorizationHeader.replace("Bearer ", "");
    Jwt jwt = jwtDecoder.decode(token);
    String email = jwt.getClaims().get("email").toString();
    return userRepository.findByEmail(email);
}
```

3. kódrészlet. Felhasználó megkeresése Bearer token alapján

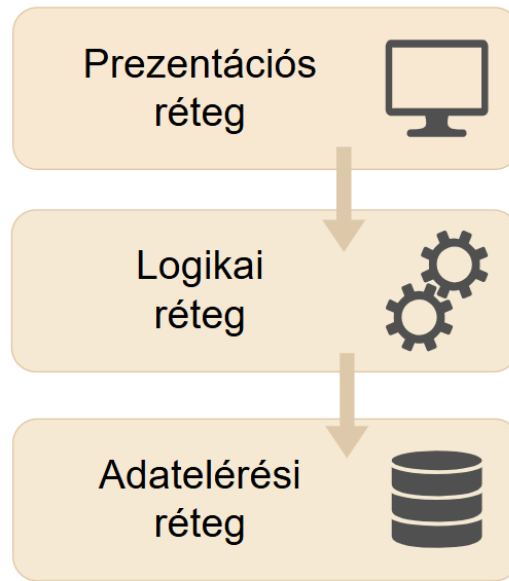
2.1.4. Képfeltöltés és médiakezelés

A képek hatékony megőrzéséhez szükség van egy külső, képek tárolására alkalmas szolgáltatásra. Ilyen céllal az alkalmazásban a Cloudinary került integrálásra.

A Cloudinary egy olyan felhőalapú platform, amely lehetőséget biztosít képek tárolására, kezelésére és elérésére. A szolgáltatás használatához mindössze egy regisztráció szükséges. A képek feltöltése történhet közvetlenül a böngészőből, valamint rendelkezésre állnak különböző nyelvű SDK-k is, amelyek megkönnyítik a képek programból való feltöltését és kezelését.

A projektben a Java SDK⁴ került felhasználásra, amely egyszerű hozzáférést biztosít a Cloudinary feltöltési API-jához. A szükséges csomag elérhető a Maven központi tárolójában, *com.cloudinary:cloudinary-http44* néven. A Cloudinary objektum konfigurálásához meg kell adni a szolgáltató által biztosított azonosítókat: *apiKey*, *apiSecret*, valamint a *cloudName* paramétereket. Ezek a hitelesítési adatok biztosítják a kapcsolatot az alkalmazás és a Cloudinary

⁴https://cloudinary.com/documentation/java_integration



3. ábra. Három rétegű architektúra

tároló között. Mentéskor a képek Base64 formátumú karakterláncként, REST API-n keresztül érkeznek a frontend felől. A feldolgozás során ezek a karakterláncok először dekódolásra kerülnek byte tömbbé, majd a `clouinary.uploader()` metódus segítségével feltöltődnek a felhőalapú tárhelyre. Sikeres mentés esetén a `uploadResult.get("secure_url")` hívással elérhetővé válik a feltöltött kép URL-je, amely az adatbázisban kerül eltárolásra.

2.2. Architektúra

A szerver a háromrétegű architektúra alapján épül fel. Ebben a struktúrában a magasabb szintek csakis az alattuk lévő rétegektől függenek, ezzel biztosítva a rendszer átláthatóságát és könnyebb karbantarthatóságát.

Az alsó réteg az adatelérési réteg, amely közvetlen kapcsolatban áll az adatbázissal. Ez a réteg felelős az adatok lekérdezéséért és módosításáért.

Középen található a logikai réteg, amely a rendszer üzleti logikáját valósítja meg. Itt történik a különböző adatműveletek feldolgozása, validálása és az ezekhez kapcsolódó szabályok kezelése.

A legfelső réteg a prezentációs réteg, amely a kliens és a szerver közötti kommunikációért felel. Ezen keresztül érkeznek a kérések a felhasználói felületről.

2.3. Alkalmazott technológiák

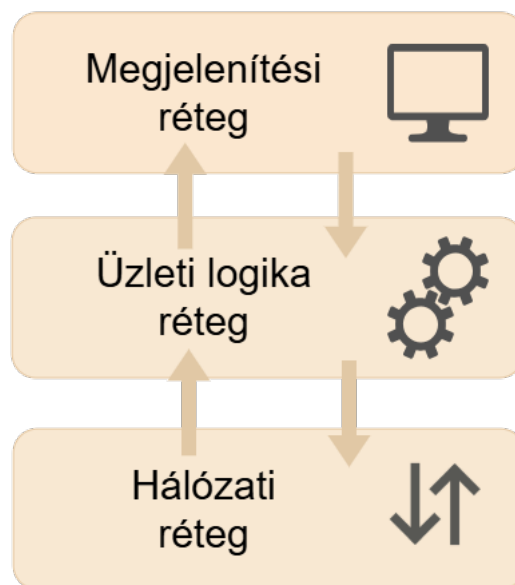
A szerver a Spring [13] keretrendszer segítségével, Java nyelvben készült. A Spring egyik legnagyobb előnye, hogy leveszi az infrastruktúra kezelésének terhét a fejlesztő válláról, így az alkalmazás üzleti logikájára tud koncentrálni. A Spring lehetővé teszi, hogy az alkalmazás egyszerű Java objektumokra, az úgynevezett Plain Old Java Objects-re (POJO-kra) épüljön. Ezekhez az objektumokhoz könnyen hozzárendelhetők különféle szolgáltatások, anélkül hogy a fejlesztőnek alacsony szintű API-kat kellene közvetlenül kezelnie.

3. Mobil- és webalkalmazás

A StoryGraph rendszerhez a felhasználó egy többplatformos (cross-platform) alkalmazáson keresztül tud hozzáférni, mely elérhető Android és iOS operációs rendszerekkel rendelkező mobileszközökön, valamint webes felületen keresztül is. Ez a fejezet bemutatja az alkalmazás rétegzett architektúráját, különböző funkcióit, valamint a fejlesztéshez használt technológiákat és eszközöket.

3.1. Alkalmazás architektúra

A StoryGraph mobil- és webalkalmazásának szerkezete egy háromrétegű architektúrán alapul, ahol minden réteg jól meghatározott szerepet tölt be. A rétegek közötti kommunikáció adott osztályokon és azok metódusain keresztül történik. Ezt a szerkezetet szemlélteti a 4. ábra.



4. ábra. A mobil- és webalkalmazás többrétegű architektúrája

A legfelső réteg a megjelenítési réteg, ami az alkalmazás elemeinek megjelenítésére fókuszál. Magába foglalja azokat a widget osztályokat, melyeket az alkalmazás kirajzol a képernyőre. A Flutter widgetekről részletesebben a 3.3.3 fejezetben esik szó.

A második réteg, az üzleti logikai réteg, az alkalmazás kliensoldali logikájával foglalkozik. Ide tartoznak a BLoC osztályok, valamint az ezek által használt különböző segédfüggvények. Ez a köztes réteg az első rétegből érkező eseményekre reagálva állapotokat generál, így

biztosítva a kommunikációt a megjelenítési réteggel. Emellett kapcsolatban áll a legalsó réteggel is, mivel annak visszaadott adatait is feldolgozza. A BLoC technológia részletezve van a 3.4 fejezetben.

Az utolsó, és egyben legalsó rétege az architektúrának, a hálózati réteg, azokat az API osztályokat tartalmazza, amelyek a Dio és a Retrofit technológiák segítségével kerültek megvalósításra. Ezek az osztályok közvetlenül létesítenek kapcsolatot a StoryGraph szerverével HTTP alapú kérések útján. A réteg DTO-kat (Data Transfer Object), vagyis adatátviteli objektumokat használ az adatok küldésére és fogadására. A Dio technológia részletesebb bemutatása a 3.5.1 fejezetben, a Retrofit részletesebb leírása pedig a 3.5.2 fejezetben található.

3.2. Funkcionalitások

A StoryGraph mobilalkalmazás számos funkcionáliszt kínál a felhasználó számára, lehetővé téve a történetek könnyű kezelését, valamint más felhasználókkal létesített kollaborációt. Ez a fejezet részletesen bemutatja a Flutter alapú alkalmazás által kínált funkciókat, valamint ezek működésének technikai hátterét.

3.2.1. Bejelentkezés

Az alkalmazás legelső funkcionálisza, ami egy új felhasználót fogad, a bejelentkezés, mely az 5. ábrán is látható kezdőoldalon végezhető el. A bejelentkezés a Google ikonnal jelölt gomb megnyomásának hatására történik, már létező Google-fiókkal lehetséges.

A kliensoldali bejelentkezési logika három lépésből áll.

Az első lépés a bejelentkezés indítása. A felhasználó bejelentkezése a Google Sign-In⁵ csomag segítségével történik. Ekkor a rendszer egy felugró ablakban megkéri a felhasználót, hogy jelentkezzen be a Google-fiókjába.

A második lépés a felhasználói adatok kinyerése. A sikeres bejelentkezés után a rendszer hozzáfér a felhasználó adataihoz. Az adatok közül az alkalmazás kinyeri a Google-fiók felhasználónevét, email címét és profilképét, amelyeket elküld a szervernek, várva a szerver válaszát a kérésre, amely sikeres bejelentkezés esetén tartalmaz egy generált JWT token.

Az utolsó lépés a felhasználói adatok tárolása és felhasználása. A szerver által visszaküldött token, valamint a kinyert felhasználói adatokat az alkalmazás a lokális tárolóba menti, és

⁵https://pub.dev/packages/google_sign_in

annak segítségével tárolja. Ez a `SharedPreferences`⁶ osztály segítségével valósul meg. A 3.5.1. fejezetben említett `AuthorizationInterceptor` az előzőleg elmentett token értékét a későbbiekben innen, a lokális tárhelyből olvassa ki, és automatikusan hozzáfűzi azt a megfelelő HTTP-hívások fejlécéhez, mielőtt azokat az adott endpointokra továbbítaná.

Ezáltal a bejelentkezési logika által a felhasználó kényelmesen és biztonságosan jelentkezhet be a Google-fiókjával, miközben a rendszer biztosítja az autentikációt a JWT token segítségével. Sikeres bejelentkezés esetében egy köztes betöltőképernyő (splash screen) megjelenése után a felhasználó az alkalmazás főoldalára kerül, ahol a saját adatai (név és profilkép) meg is jelennek.

3.2.2. Főoldal

Bejelentkezés után a felhasználók a főoldalra jutnak (6a. ábra), ahol egy reszponzív navigációs sáv segíti a tájékozódást. A sáv a képernyő méretétől függően a bal oldalon vagy az alsó részen jelenik meg. A navigáció segítségével a felhasználók egyszerűen navigálhatnak a főoldal, a mentett történetek (3.2.7. fejezet) és a saját történetek (3.2.3. fejezet) között. Az oldal tetején egy információs sáv található, amely megjeleníti a felhasználó nevét és profilképét. Ez alatt helyezkedik el a keresősáv, ahol cím vagy szerző alapján lehet történetek között keresni. A képernyő maradék részében a már létező történetek jelennek meg, tematikus kategóriák szerint csoportosítva, melyek a következők: "Popular" (Népszerű), "Best sellers" (Best-seller), "Recent" (Legújabb), valamint "Private" (Privát).

3.2.3. Saját sztorik

A navigációs sáv harmadik ikonjának megnyomásával a "My Stories" oldal tekinthető meg. Ezen az oldalon lehetőség van új sztorikat létrehozni, valamint a már meglévőket módosítani. A főoldalhoz hasonlóan ezen az oldalon is kategóriák szerint csoportosulnak a történetek, ahogyan ez a 6c. ábrán is látható. Az első sáv első eleme egy plusz gomb, amivel új könyvek hozhatók létre. A gomb megnyomására egy form (űrlap) jelenik meg a felhasználó előtt, amivel megadhatja az újonnan létrehozni kívánt sztori címét, szerzőjét, valamint borítóképét. A form leadásával a felhasználó egyenesen a szerkesztőhöz kerül, mely részletesebben a 3.2.5. fejezetben van bemutatva.

A "My Stories" oldalon az első sávban ugyancsak megjelennek a felhasználó által már

⁶https://pub.dev/packages/shared_preferences

létrehozott történetek. A második sávban azok a sztorik sorakoznak, ahol a felhasználó aktív szerkesztőként van jelen. Az oldal legalján, az utolsó sávban az előbb említett két kategóriába tartozó könyvek jelennek meg, a legutóbbi módosítás dátuma szerint csökkenő sorrendben.

3.2.4. Közös szerkesztés és sztoribeállítások

A közösségi történetírás támogatása érdekében a felhasználók meghívhatnak másokat szerkesztőnek a saját történeteikhez.

A 7. ábrán egy sztori beállítási oldala látható, amely a szerkesztői nézetből (3.2.5. fejezet), a "More" gomb megnyomásával érhető el. Ezen az oldalon megjelenik a történet létrehozója, a szerkesztők listája, valamint — ha a történet privát — azoknak a felhasználóknak a listája is, akik megtekinthetik azt. Az "Add New" gombok kizárólag a tulajdonos számára érhetők el; más felhasználók számára ezek nem jelennek meg. A szerkesztők neve mellett elhelyezett törlés ikon segítségével a tulajdonos visszavonhatja az adott szerkesztő jogosultságait, a kis plusz ikonra kattintva pedig újra engedélyezheti azokat. A meghívott szerkesztők nemcsak új fejezetek létrehozására jogosultak, hanem a meglévő fejezetek szerkesztésére is.

A "Private Story" kapcsoló segítségével a saját történetek priváttá tehetők. Privát történetet kizárólag a meghívott felhasználók olvashatnak el. Jelenleg még nincs lehetőség meghívók küldésére, így a privát történetek egyelőre mások számára nem elérhetők.

A "For Subscribers" kapcsoló hasonlóan működik a privát sztori beállításhoz, azonban ez a funkció sem került még megvalósításra.

A beállítások oldal legalján egy "Delete" gomb található, amely kizárólag a történet létrehozója számára elérhető, és a könyv végleges törlésére szolgál.

3.2.5. Szerkesztő

A felhasználónak két lehetősége van sztorik szerkesztésére: struktúra alapú szerkesztés (8a. ábra), valamint tartalom alapú szerkesztés (8b. ábra). Egy sztori szerkesztői nézete a "My Stories" oldalról (3.2.3. fejezet) válik elérhetővé, az adott történet borítóképére való kattintással. Ekkor a felhasználó számára megjelenik a sztori belső struktúrája egy interaktív gráfnézeten keresztül. Fa struktúra segítségével az alkalmazás megjeleníti a különböző fejezeteket, a köztük levő kapcsolatokat pedig halvány vonalakkal szemlélteti. A gráfnézet a Flutter graphview ⁷ nevű csomagjának segítségével készül el, a felépített gráfot pedig egy InteractiveViewer widget

⁷<https://pub.dev/packages/graphview>

tartalmazza, amely lehetőséget ad a felhasználónak különböző interakciókra, mint például a nagyítás és kicsinyítés, vagy a tartalom mozgatása a megadott határokon belül. Ebben a gráfnézetben a felhasználónak lehetősége nyílik változtatni a sztori struktúráján új fejezetek hozzáadásával.

A megjelenített gráfnak két fajta csomópontja van. Az egyik, mely az adott fejezet borítóképét tartalmazza, a "Chapter node", vagy fejezet csomópont. A másik, mellyel új fejezetek hozhatók létre, az "Add New node", vagyis a hozzáadó csomópont. Minden fejezet tartalmaz egy saját hozzáadó csomópontot, mellyel utána következő fejezetek adhatók hozzá. Az első fejezetek egy közös hozzáadó csomópontból származnak le - ezzel van lehetősége a felhasználónak párhuzamos első fejezetek létrehozására.

A fejezet tartalmának szerkesztőjét az egyes fejezetek borítóképére való kattintásával lehet elérni. A felhasználó ebben az esetben egy Rich Text Editor-hoz kerül, ahol változtathatja a fejezet szövegét, valamint címét. A változtatásokat a "Save" gombra való kattintással lehet elmenteni. A fejezet ugyancsak ebből a szerkesztőből kerülhet törlésre, a jobb felső sarokban található menüpontból. Törlés esetén a felhasználó visszakerül a sztori szerkesztői gráfnézetére, ahol a frissült struktúra fog megjeleníteni.

A Rich Text formátum segítségével a felhasználónak lehetősége nyílik a tartalom vizuális formázására, mint például félkövér vagy dőlt betűk használata, listák beszúrása, esetleg a szöveg átszínezése vagy átméretezése. Ez a formátum a Plain Text (egyszerű szöveg) formátumhoz képest egy sokkal gazdagabb esztétikai élményt nyújt a felhasználó számára.[11] Az alkalmazás a szerkesztőt a Flutter Quill⁸ csomag segítségével építi fel és konfigurálja. A csomag által kínált szerkesztő Delta⁹ struktúrában tárolja és kezeli a tartalmat. Ez egy JSON-alapú struktúra, mely a sima szöveg helyett műveletsorozatokat ír le, valamint attribútumokat rendel ezekhez. A struktúrára példa a 4. kódrészletben található.

```
[
  {"insert": "A szöveg "},
  {"insert": "félkövér", "attributes": {"bold": true}},
  {"insert": " formátumot tartalmaz.\n"}
]
```

4. kódrészlet. Delta struktúra

⁸https://pub.dev/packages/flutter_quill

⁹Delta dokumentáció: <https://quilljs.com/docs/delta>

3.2.6. Olvasói nézet

Olvasói szempontból két helyről lehet hozzáférni a sztorik tartalmához: a főoldalról és a "Bookmark" oldalról (3.2.7. fejezet), ahol a felhasználó által lementett és megjelölt történetek találhatók. Az olvasói nézet mindkét esetben a sztori borítóképére való kattintással jelenik meg.

A felhasználót két helyzet fogadhatja az olvasói gráfnézet megnyitásakor. Ha még ezelőtt sosem nyitotta meg a könyv egyik fejezetének tartalmát sem, akkor megjelenik az oldalon a sztori belső struktúrája, egy hasonló gráfstruktúrával, mint amit a 3.2.5. fejezetben említett szerkesztő tartalmaz, azzal a különbséggel, hogy itt csupán fejezet csomópontok találhatók, és a fejezetekre való kattintás nem a szerkesztőt, hanem a fejezet olvasói nézetét jeleníti meg. Abban az esetben, ha a felhasználó már meglátogatott legalább egy fejezetet, a gráfnézet felett a már elolvasott fejezetek görgethető listája fog megjelenni. Ez a funkcionalitás segít a felhasználónak az olvasási haladás nyomon követésében, és lehetővé teszi, hogy a legutoljára elolvasott fejezetek könnyen elérhetők legyenek számára. Az elolvasott fejezetek frissítése az olvasói nézetből való kilépéskor történik.

Az olvasásra megnyitott fejezetek végén ugyancsak található egy görgethető sáv a közvetlenül az adott fejezet után következő tartalmakkal, így a felhasználó könnyen kiválaszthatja azt a történetszálat, amin tovább szeretne menni. Ha olvasás után vissza szeretne térni a teljes gráfnézetre, meglehet ezt az oldal alján található "Full Graph View" gomb segítségével.

Az olvasói nézeteket a 9. ábra, valamint a 10. ábra szemlélteti.

3.2.7. Mentett sztorik

A navigációs sáv második ikonjának megnyomásával a felhasználók a mentett történetek oldalára jutnak, amit szemléltet a 6b. ábra. Egy történet az olvasói nézetben (3.2.6. fejezet) menthető el a jobb felső sarokban található ikonnal. Mentéskor az adatbázisban a felhasználó és a történet közötti kapcsolat egy logikai adattagja igazra áll, amely jelzi, hogy az adott történet mentésre került-e vagy sem.

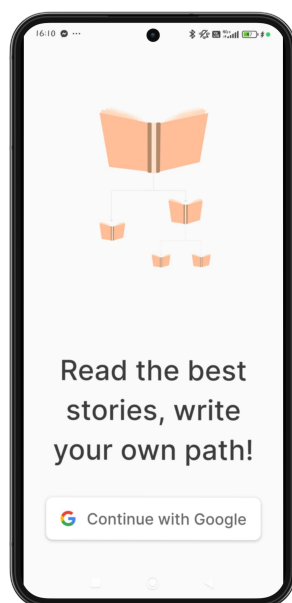
3.2.8. Beállítások

A StoryGraph alkalmazás lehetőséget biztosít a felhasználók számára, hogy a platform kinézetét saját igényeik szerint testreszabhassák. Az ehhez tartozó funkcionalitásokat a bejelentkezett felhasználó az alkalmazás jobb felső sarkában megjelenített profilképére kattintva

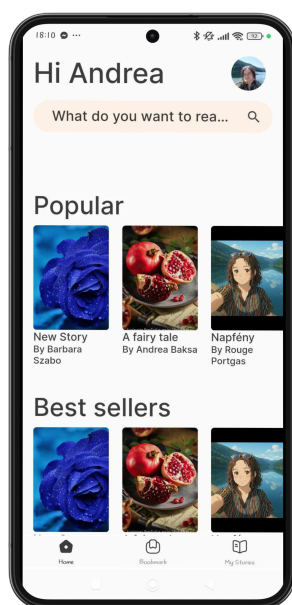
érheti el. A profilképre kattintás után egy új oldal jelenik meg, amely a beállításokat, valamint a kijelentkezéshez használható "Sign Out" gombot tartalmazza. A felhasználónak két lehetősége van preferencia szerint testreszabni az alkalmazást:

- Nyelvi beállítások: lehetőséget adnak választani a StoryGraph által támogatott két nyelv, az angol és a magyar nyelv között. A nemzetköziesítésről további részletek a 3.7. fejezetben találhatóak.
- Téma beállítások: lehetőséget adnak választani világos, valamint sötét téma között az alkalmazás számára. A témázásról részletesebben a 3.8. fejezet beszél.

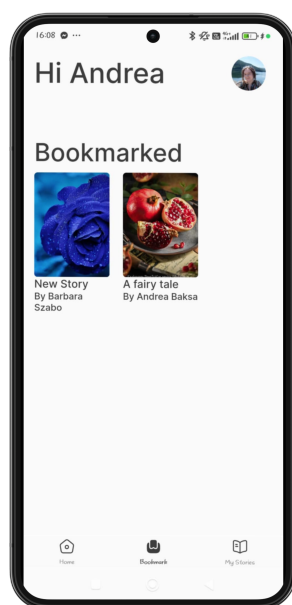
Az alkalmazás beállításai elmentődnek a lokális tárolóba, éppen ezért az alkalmazás elhagyása után sem vesznek el. A beállításokat kezelő oldalt a 11. ábra szemlélteti.



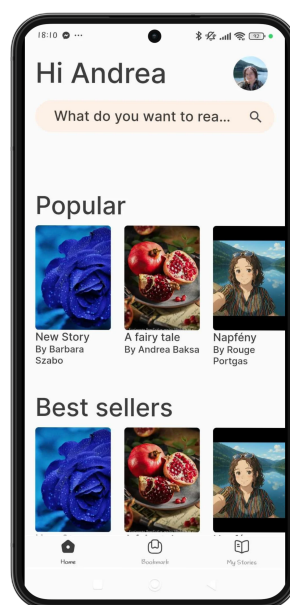
5. ábra. Bejelentkezési oldal és a StoryGraph szlogen



(a) Főoldal

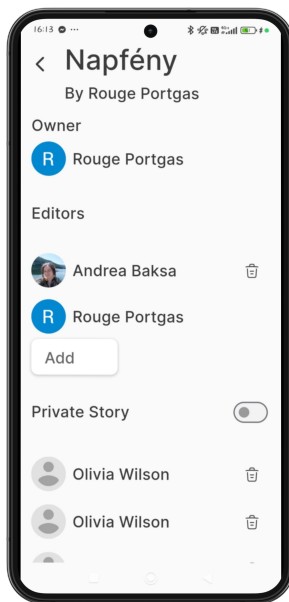


(b) Mentettek

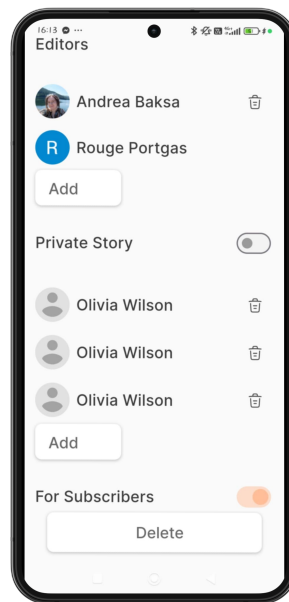


(c) Saját sztorik

6. ábra. A főmenüből elérhető oldalak

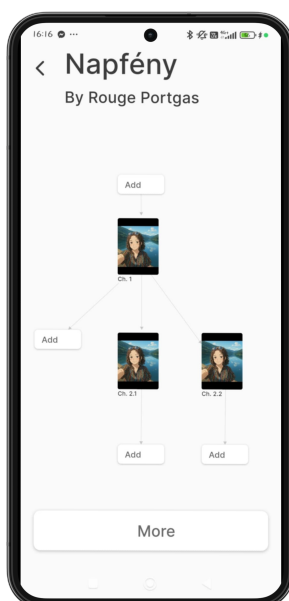


(a)

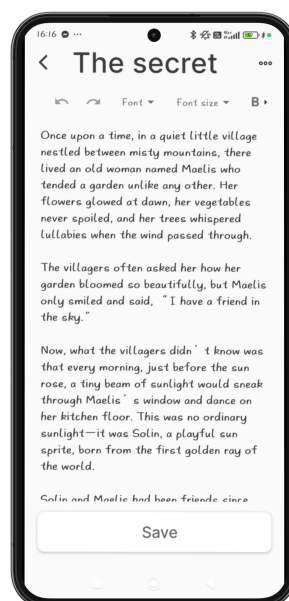


(b)

7. ábra. A sztori beállításait tartalmazó oldal

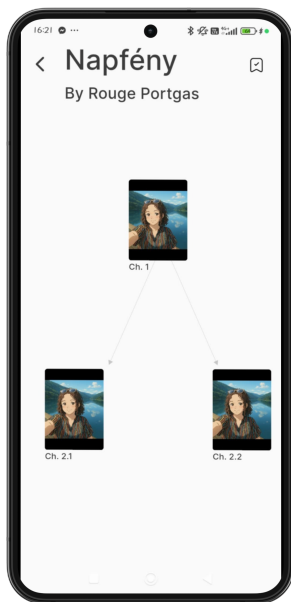


(a) Struktúra szerkesztő

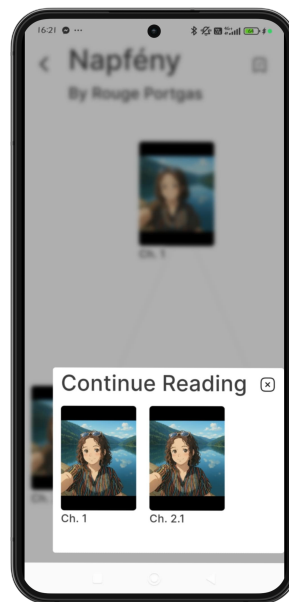


(b) Tartalom szerkesztő

8. ábra. Sztori szerkesztési lehetőségek



(a) Gráfnézet olvasóknak

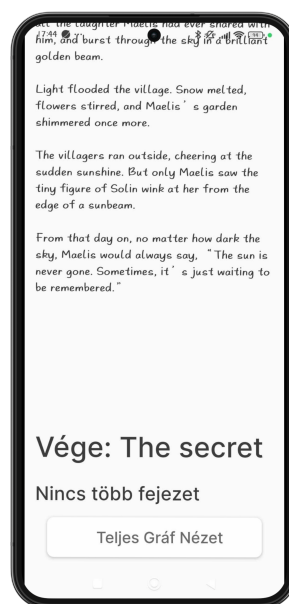


(b) Olvasási előzmények

9. ábra. Szerkezeti olvasói nézet

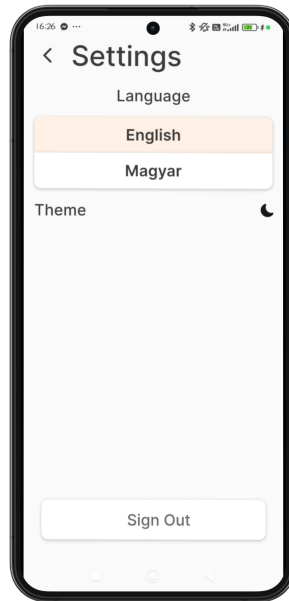


(a)

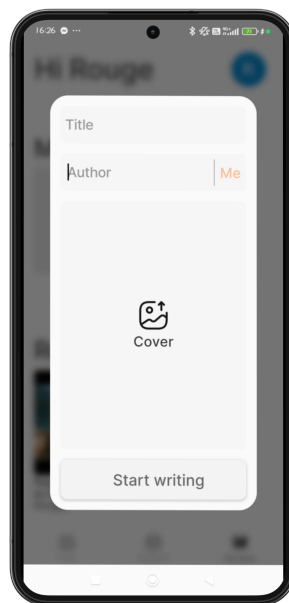


(b)

10. ábra. Tartalmi olvasói nézet



11. ábra. Nyelvi és témázási beállítások az alkalmazásban



12. ábra. Új sztori létrehozásakor megjelenő form

3.3. Flutter

A StoryGraph mobilalkalmazás a Flutter [7] nyílt forráskódú, cross-platform UI-toolkit segítségével készült. Ez a fejlesztői eszköz lehetővé teszi, hogy az alkalmazás egyetlen közös kódbázison alapuljon, miközben egységes megjelenést biztosít az alkalmazás által támogatott platformokon. A Flutter célja, hogy natív kód írása nélkül is gyorsan, több platformon is futtatható alkalmazásokat lehessen készíteni. A StoryGraph ennek köszönhetően futtatható

iOS-en, Androidon, valamint webes környezetben is.

3.3.1. A Dart nyelv

A Dart [5] egy objektumorientált, típusos programozási nyelv, amelyet a Google fejlesztett ki, és amelyet elsődlegesen kliensoldali alkalmazások fejlesztésére terveztek.

A Dart a Flutter (3.3. fejezet) alapjául szolgál. A nyelv szintaxisa hasonló más magas szintű nyelvek szintaxisához, mint például a Java vagy JavaScript. A Flutter szoros integrációban működik a Dart nyelvvel, amelynek egyik kulcsfontosságú komponense a Dart Virtual Machine. Ez a komponens biztosítja az alkalmazások futtatásának alapját, valamint lehetőséget a hot reload mechanizmus használatára, amely dinamikus kódfordítással teszi lehetővé a valós idejű változtatásokat a fejlesztés alatt.

3.3.2. Flutter belső architektúra

Architektúra szempontjából a Flutter egy többrétegű, rugalmasan bővíthető rendszer. A keretrendszer különálló, egymásra épülő könyvtárakból áll, ahol egyik réteg sincs kiváltságos helyzetben a másikkal szemben – minden komponens lecserélhető vagy kiegészíthető. [3]

A Flutter motor (Flutter Engine) a rendszer központi eleme, amely C++ nyelven íródott, és gondoskodik a grafikus felületek rendereléséről, a szövegkezelésről, az I/O műveletekről, a bővítmények kezeléséről, valamint a Dart futtatókörnyezet működtetéséről.

A motorhoz a `dart:ui` könyvtáron keresztül fér hozzá a magasabb szintű Dart kód, amely Dart osztályokba csomagolja a C++-os funkcionalitást. Erre épül a Flutter framework.

A framework több rétegből áll:

- Alapréteg: animációkat, festést és gesztuskezelést biztosít.
- Renderelési réteg: gondoskodik az elrendezésről és a megjelenítendő objektumok fa struktúrában való kezeléséről.
- Widget réteg: absztrakciót nyújt a UI elemekre, valamint itt jelenik meg a reaktív szemlélet.
- Material és Cupertino könyvtárak: platform-specifikus komponenseket (Android és iOS dizájnelemek) biztosítanak.

Ezek mellett a legfelsőbb szinten helyezkednek el a pluginek és csomagok, amelyek kiegészítő funkcionalitásokat nyújtanak.

3.3.3. Widgetek és UI komponensek

A Flutter widgetek a felhasználói felület alapvető építőelemei. Hierarchikus struktúrába szerveződnek, és egymásba ágyazva építik fel az alkalmazás felhasználói felületét. Egy adott widget `build` függvénye az, ami leírja, hogy a widget milyen további widgetekből és komponensekből áll.

A widgeteknek két fő típusa van: az állapot nélküli widgetek (`StatelessWidget` [15]), amelyek nem tárolnak belső állapotot, és az állapottal rendelkező widgetek (`StatefulWidget` [14]), amelyek képesek kezelni és frissíteni az eltárolt állapotot. Az állapot nélküli widgetek statikus elemeket jelenítenek meg, és nem változtatják meg tulajdonságaikat vagy viselkedésüket futás közben, míg az állapottal rendelkező widgetek dinamikusan reagálnak a felhasználói interakciókra - képesek állapotot eltárolni, valamint annak változására reagálni.

3.3.4. Konfiguráció

A StoryGraph projekt frontendjének konfigurációs beállításait a projekt gyökerében található `pubspec.yaml` fájl tartalmazza. Ez az állomány határozza meg többek között az alkalmazás nevét, verziószámát, valamint azt a Flutter SDK-verziót, amelyre a fejlesztés épül. Ebben a fájlban található meg az alkalmazás összes külső függősége is, melyeket a Dart csomagkezelője (`pub`) kezel. A függőségek felsorolása után a fájlban szerepel az alkalmazás ikonjára, valamint az egyéb erőforrásokra vonatkozó konfiguráció is, mint a képek, animációk és a nemzetköziesítéshez használt fordításokat tartalmazó állományok forrásmappája.

A további, környezeti változókkal megvalósított konfigurációkhoz a Flutter `envied`¹⁰ csomagja került felhasználásra.

3.4. BLoC

A Flutter BLoC [12] (Business Logic Component) egy népszerű állapotkezelési megoldás a Flutter alkalmazások számára. Segítségével könnyen kezelhetők az alkalmazás különféle állapotai, legyen szó egyszerű vagy összetettebb interakciókról. A könyvtár jól dokumentált, számos példával kiegészítve, és széles körben elterjedt a Flutter közösségben.

¹⁰<https://pub.dev/packages/envied>

3.4.1. BLoC osztályok

A BLoC ¹¹ tervezési minta lényege, hogy elválassza az üzleti logikát a megjelenítési rétegtől. A szétválasztás események (event) és állapotok (state) segítségével történik. A felhasználói interakciók eseményként kerülnek továbbításra a BLoC felé, amely feldolgozza azokat, majd állapotokat bocsát ki a felhasználói felületet frissítése érdekében. Az alkalmazásban a BLoC minta a flutter_bloc csomag felhasználásával került implementálásra. Ez a csomag egy Bloc<Event, State> [4] osztályt biztosít, amely a fent ismertetett működési logikát valósítja meg. Az alkalmazás minden egyes oldala, illetve a jelentősebb logikai műveleteket végző widgetjei önálló BLoC osztályt kaptak, saját esemény- és állapotdefiníciókkal.

3.4.2. BLoC-alapú widgetek

A flutter_bloc csomag widgeteket [6] is biztosít, amelyek elősegítik az állapotfigyelést és az interakciók kezelését. A BlocBuilder komponens az állapotok közötti váltást kezeli, amikor egy új állapot érkezik, a BlocBuilder annak megfelelően hoz létre egy új widgetet. A BlocProvider widget egy új bloc létrehozására szolgál, amelyet az alatta elhelyezkedő bármely widget elérhet. A BlocListener működése hasonlít a BlocBuilder-hez, azonban nem a widgetek újraépítését végzi, hanem az állapotváltozásokra adott egyszeri, mellékhatás-jellegű műveletek kezelésére szolgál.

3.5. Hálózati réteg

A StoryGraph mobilalkalmazás a rendszer szerveréhez csatlakozva végzi el az adatok tartós tárolását, valamint a legtöbb funkció működéséhez is a backend szolgáltatásait használja. Az alkalmazás hálózati kommunikációját egy különálló hálózati réteg biztosítja, amelyen keresztül az applikáció eléri a szerverrel való kapcsolatot. Ez a réteg a dio¹² és retrofit¹³ csomagok segítségével épül fel, amelyek megkönnyítik a HTTP-kérések kezelését. A réteget alkotó osztályok felelősek a kérések elküldéséért és a válaszok feldolgozásáért. A fejezet célja a hálózati réteget megvalósító osztályok bemutatása, valamint azoknak a külső csomagoknak az ismertetése, amelyek elengedhetetlenek a hálózati kommunikáció kialakításához és működtetéséhez.

¹¹<https://pub.dev/documentation/bloc/latest/index.html>

¹²<https://pub.dev/packages/dio>

¹³<https://pub.dev/packages/retrofit>

3.5.1. Dio

A StoryGraph mobilalkalmazás a szerverrel való adatcserét REST API-n keresztül valósítja meg, amely a HTTP protokollt használja. Ehhez az alkalmazás egy fejlett HTTP kliens könyvtárat alkalmaz, a dio-t. A dio a Dart beépített http csomagjára épül, ugyanakkor számos hasznos többletfunkcióval rendelkezik: testre szabható beállítások, automatikus hibakezelés, interceptorok és időkorlátok.

A HTTP-kérések lebonyolításához minden esetben egy Dio példányra van szükség. A StoryGraph alkalmazásban ezeknek a példányoknak az előállítását egy külön erre a célra létrehozott factory osztály végzi, amely különböző konfigurációkat kínál statikus metódusokon keresztül. A létrehozott példányok eltérhetnek például abban, hogy milyen interceptorokat használnak.

A fejlesztés során két fajta interceptor használatát igényelte az alkalmazás:

- **LogInterceptor:** a fejlesztés során a HTTP kérések és válaszok naplózásában játszott szerepet.
- **AuthorizationInterceptor:** a LogInterceptor-al ellentétben egy saját, fejlesztés során létrehozott interceptor, mely az Interceptor osztályból van származtatva. Az `onRequest` függvény felülírásával minden HTTP kérés esetén automatikusan hozzáadja a kéréshez az adott kliens JWT tokenjét egy hitelesítési fejléccel, amennyiben a token jelen van. A token jelenlétének jelentőségét a 3.2.1. fejezet, valamint a 2.1.3. fejezet tárgyalja bővebben.

3.5.2. Retrofit

Az alkalmazásban az endpointok elérésében egy másik csomag is nagy szerepet játszik, a retrofit. Ez a csomag a dio könyvtárra épül, és lehetővé teszi, hogy az API hívások annotációk segítségével, deklaratív módon legyenek definiálva, csökkentve ezzel a boilerplate kód mennyiségét, és növelve az olvashatóságot. A csomag segítségével az API hívások interfészek formájában hozhatók létre, amik a háttérben automatikusan implementálásra kerülnek - a retrofit automatikusan legenerálja a szükséges háttérkódot a HTTP kérések és válaszok feldolgozására. Az API osztályok a `@RestApi()` annotációval, a függvényfejlécek pedig további annotációkkal vannak ellátva. Egy ilyen osztály függvényfejlécére mutat egy példát az 5. kódrészlet.

```
@GET("/api/chapters/{chapterId}")  
Future<Chapter> getChapter(@Path("chapterId") String chapterId);
```

5. kódrészlet. Fejezetet lekérő API hívás definíciója

A Retrofit a szerverrel történő kommunikáció során JSON formátumot használ az adatok küldésére és fogadására. Az adatok megfelelő átalakításához (szerializálásához és deszerializálásához) a rendszer a `json_serializable`¹⁴ csomag szolgáltatásaira épül. Ez a csomag lehetővé teszi, hogy az adatátviteli objektumokhoz automatikusan generálódjanak azok az átalakító függvények, amelyek a Dart objektumokat JSON-ná alakítják, illetve fordítva. Az adatküldéshez és fogadáshoz a rendszer olyan adatátviteli objektumokat alkalmaz, amelyek a `json_annotation`¹⁵ megfelelő annotációival vannak ellátva. A függvények implementációját ugyancsak a rendszer generálja ki az előbb említett `json_serializable` csomag segítségével.

3.6. Dependency injection

A StoryGraph mobilalkalmazásban a függőségek kezelését a `get_it`¹⁶ nevű csomag végzi, amely a Service Locator tervezési minta alapján működik. Ez lehetővé teszi, hogy az alkalmazás - különböző pontokról - központilag regisztrált példányokat érjen el.

Az alkalmazás indításakor létrejön a `GetIt` osztály egy példánya, majd egy aszinkron inicializáló függvény segítségével regisztrálódnak a szükséges szolgáltatások: adatkezelők, a hálózati réteg komponensei, valamint a különböző állapotkezelő objektumok. Ezek az objektumok az alkalmazás futása során bárholn is elérhetők a `GetIt` osztály létrehozott példányán keresztül anélkül, hogy külön példányosításra lenne szükség belőlük. A StoryGraph alkalmazás esetében a rendszer az objektumokat lazy singleton, vagy singleton életciklussal regisztrálja.

3.7. Nyelvi támogatás

Az applikáció több nyelven is elérhető, ami a Flutter `easy_localization`¹⁷ csomag segítségével valósult meg. A rendszer kulcsszavai és mondatai nyelvenként külön JSON fájlokban lettek definiálva. Ezek a fájlok alapján a terminálban futtatott `flutter pub`

¹⁴https://pub.dev/packages/json_serializable

¹⁵https://pub.dev/packages/json_annotation

¹⁶https://pub.dev/packages/get_it

¹⁷https://pub.dev/packages/easy_localization

`run easy_localization:generate -f keys -o locale_keys.g.dart` parancs automatikusan legenerálja a szükséges összesítő fájlokat. A rendszer gyökerében (a MaterialApp widget-ben) beállításra kerültek a szükséges kulcsok, így a LocaleKeys osztályon keresztül bármely alwidgetben egyszerűen elérhetők az előre definiált kulcsszavak. Az ily módon megadott szöveg automatikusan a rendszerben beállított nyelvhez fog igazodni.

3.8. Témázás

Az alkalmazásban a felhasználónak lehetősége nyílik választani sötét és világos téma között, valamint kedve szerint bármikor váltogatni ezeket. A két téma közötti váltást egy külön erre a célra készített komponens, a ThemeManager osztály kezeli. A felhasználói élmény növelése érdekében az aktuálisan kiválasztott téma állapota tartósan mentésre kerül a készülék lokális tárolójában, így az alkalmazás újraindításkor is megőrzi a felhasználó preferenciáját.

A ThemeManager a ChangeNotifier osztályból származik, így lehetőséget biztosít arra, hogy a felhasználói felület különböző komponensei automatikusan reagáljanak a témaváltásra. Az osztály egy belső logikai változó segítségével tárolja, hogy éppen melyik téma aktív, valamint egy _currentTheme nevű példányban tartalmazza az aktuális témához tartozó stílus- és vizuális elemeket, amelyeket a UiThemedAssets osztály definiál. A témaváltást a toggleTheme() metódus végzi, amely megfordítja az eltárolt logikai változó értékét, ennek megfelelően frissíti a stíluselemeket tartalmazó UiThemedAssets példányt, elmenti az új beállítást, majd értesíti az összes érintett UI-komponenst a változásról.

Az aktuális téma megjelenéséért felelős UiThemedAssets osztály tartalmazza az alkalmazáshoz szükséges színeket, ikonokat és egyéb grafikai erőforrásokat. Ez az osztály két gyári konstruktort kínál, amelyek külön példányokat hoznak létre a világos és sötét témához tartozó beállításokkal.

Az alkalmazás egészében a Provider¹⁸ csomag segítségével valósul meg a témakezeléshez szükséges állapotkezelés: a ThemeManager példány egy ChangeNotifierProvider-en keresztül válik elérhetővé a widget-fában, ezáltal a felhasználói felület bármely része hozzáférhet az aktuális témaelemekhez és automatikusan frissülhet, ha a téma megváltozik.

¹⁸<https://pub.dev/packages/provider>

3.9. Globális hibakezelés

Az alkalmazás globális hibakezelését egy központi komponens, a `StoryGraphErrorHandler` nevű osztály valósítja meg. Ez a komponens lehetővé teszi, hogy a futásidejű hibák egységes módon jelenjenek meg a felhasználói felületen. A hibák megjelenítésének módja teljes mértékben testre szabott: a rendszer minden hiba esetén egy dialógusablakban értesíti a felhasználót, amely az alkalmazás aktuális témájának megfelelő stílusában jelenik meg.

A hibakezelő konfigurálása során regisztrálásra kerül egy hiba-visszahívó (callback) a `StoryGraphErrorHandler.onError` mezőben, amelyet később bárholnan el lehet érni az alkalmazásból. Ez a visszahívó automatikusan megjelenít egy `AlertDialog` típusú figyelmeztető ablakot. Az ablakban egy középre igazított cím és a hiba részletes leírása is megjelenik, egyetlen „OK” gomb kíséretében, amely bezárja az értesítést.

A globális hibakezelés segítségével az alkalmazás bármely pontján, közvetlenül vagy közvetetten is meghívható a hibamegjelenítő logika, és az az egész alkalmazáson keresztül egységes szerkezeti és kinézeti szempontból is.

4. DevOps

A fejlesztés során fontos mérföldkő volt az alkalmazás kitelepítésének megvalósítása. A kitelepítési folyamat részleges automatizálása megkönnyítette a fejlesztési folyamatot, valamint a szerver kitelepítése ugyancsak előnyt jelentett a kliensoldali fejlesztés során, mivel ez eltörölte a lokális szerverkörnyezet szükségességét.

4.1. Gitlab CI/CD, Docker

A frontend kitelepítésének automatizálásában a GitLab pipeline[8] segített. A pipeline futása során először egy build folyamat megy végbe, majd a sikeres buildet követően Docker segítségével létrejön egy image, amely a Container Registry-be töltődik fel.

A backend oldalon a GitLab pipeline kizárólag buildelésre és a kódminőség ellenőrzésére szolgált. Az image-ek létrehozása itt manuálisan, szintén Docker segítségével történik, azonban ezek az image-ek is a GitLab Container Registry-ben kerülnek tárolásra.

4.2. Kubernetes, Helm

A StoryGraph komponensei (adatbázis, szerver, valamint webalkalmazás) egy meglévő Kubernetes [10] klaszterre kerültek kitelepítésre. A Kubernetes egy nyílt forráskódú konténer orkesztrációs platform, amely automatikusan kezeli a konténerek telepítését, skálázását, frissítését és visszaállítását. Az alkalmazás különböző komponensei önálló konténerekben futnak, amelyeket a Kubernetes podokként kezel. A podokat szolgáltatásokon (Services) keresztül lehet elérni, így biztosított a komponensek közötti hálózati kommunikáció.

A Helm [9] a Kubernetes-alkalmazások csomagkezelője, amely sablonalapú YAML fájlok segítségével segíti elő az erőforrások deklaratív leírását és verziózását. A StoryGraph üzembe helyezése során a Helm chartok lehetővé tették a fejlesztési, tesztelési és éles környezetek gyors, konzisztens és automatizált telepítését. A StoryGraph alkalmazás három fő komponensből épül fel: az API, a web frontend és az ArangoDB adatbázis. Minden komponens saját Helm chartot tartalmaz, amely biztosítja a könnyű telepítést és frissítést. A Helm chartok a values.yaml fájlokban definiált konfigurációk szerint telepítik az alkalmazás különböző komponenseit.

5. Fejlesztési eszközök és módszerek

A hatékony és produktív fejlesztés érdekében a StoryGraph projekt során számos fejlesztési eszköz és módszer használatára sor került. Az alkalmazott fejlesztési módszerek segítették a projekt megfelelő ütemezését, a fejlesztői csapat közötti hatékony kommunikációt, valamint a folyamatos visszajelzés biztosításában is nagy szerepet játszottak. Más eszközök elősegítették az alkalmazás kompakt és egységes kinézetének megtervezési folyamatát.

5.1. Scrum

A fejlesztés során a csapatmunka a Scrum¹⁹ módszer elvei alapján szerveződött. A munkafolyamat kéthetes sprintekre tagolódott, és a rendszeres napi megbeszélések segítették a csapat sikeres előrehaladását. A daily standupok, ahogy a neve is sugallja, állva történtek és nagyjából 10 percet vettek igénybe. A megbeszélések célja az volt, hogy minden csapattag beszámoljon arról, min dolgozott előző nap, mit tervez aznapra, illetve jelezze, ha bármilyen elakadásba ütközött, hogy segítséget kaphasson a többiektől. Minden sprint a feladatok meghatározásával és megbecslésével kezdődött. A becslés a Planning Poker²⁰ alkalmazás segítségével zajlott. A sprintek demóval záródtak, ahol bemutatásra kerültek az adott időszakban elkészült funkcionálisok.

5.2. Git és GitLab

A könnyű nyomon követhetőség és az egyszerűbb csapatmunka érdekében az alkalmazás fejlesztése a Git osztott verziókövető rendszer segítségével történt. A projekthez két külön repository került létrehozásra: egy a backend, egy pedig a frontend számára. A GitLab webes felülete jelentős mértékben megkönnyítette a fejlesztési folyamatokat, mivel lehetőséget biztosít az issue board használatára is. A sprintek során a feladatok (taskok) ezen a felületen kerültek rögzítésre. Egy feladat megvalósítása során új branch készült. Egy feladat elkészülésekor merge request készült, amely jelezte a csapattagok számára az ellenőrzés szükségességét. A módosítások csak az ellenőrzést követően kerültek beolvasztásra a fő ágba.

¹⁹<https://www.atlassian.com/agile/scrum>

²⁰<https://planningpokeronline.com/>

5.3. Figma

A StoryGraph alkalmazás fejlesztése során a felhasználói felület és a felhasználói élmény tervezése kulcsfontosságú szerepet játszott a sikeres megvalósításban. A Figma volt a választott eszköz, amely lehetővé tette a felhasználói felületek vizuális megtervezését. A Figma gazdag eszközkészlete felgyorsította a tervezési folyamatot, miközben fenntartotta a dizájn minőségét és konzisztenciáját. A fejlesztés első fázisában a világos téma által meghatározott komponenskinézetek készültek el, a későbbi fázisokban pedig az adott tervek mintájára a sötét téma is megszületett. A Figma eszközkészlete tartalmazott olyan hasznos funkciókat is, mint a virtuális vonalzó és a pixelrács, amelyek lehetővé tették a pixelpontos tervezést. Ezen eszközök segítségével biztosítva volt, hogy minden vizuális elem precízen illeszkedjen a kívánt méretekhez és helyekhez. A pixelpontos tervezés lehetősége segítette a fejlesztői csapatot abban, hogy az alkalmazás minden egyes képernyője és annak elemei összhangban legyenek egymással és a dizájn irányelvekkel.

Következtetések és továbbfejlesztési lehetőségek

A StoryGraph projekt keretein belül sikeresen megvalósult egy olyan alkalmazás, amely a piacra való bevezetéshez megfelelő stabilitással és felhasználóbarát felülettel rendelkezik. A projekt kezdetén kitűzött célok megvalósultak, valamint plusz funkcionálisok is sikeresen beépültek az alkalmazásba. A nemlineáris történetkezelés és a közösségi történetírás terén a projekt jelentős előrelépést jelenthet.

Mindezek mellett, az alkalmazás minőségének további növelése érdekében számos új fejlesztési lehetőség is felmerült.

Egy fontos fejlesztési lehetőség a későbbiekre nézve a különböző típusú belső interakciók bevezetése. A felhasználói élmény növelése érdekében fontos lenne sztorikkal, valamint más felhasználókkal való további kapcsolódási pontok elérhetővé tétele. Ilyen új lehetőség lenne sztorik vagy fejezetek alatti kommentszekció létrehozása, vagy egy egyszerűbb olvasó-író közötti visszajelzési/értékelési rendszer megvalósítása. Új statisztikák bevezetése és használata további lehetőségeket biztosítana a sztorik kategorizálására is.

Ezek mellett további fejlesztési lehetőségként szolgálhatnak még a következők:

- Felhasználók közötti üzenetküldés, valamint értesítések lehetősége a jobb felhasználói élmény érdekében
- Sztorik tartalmának importálása és exportálása a még könnyebb kezelhetőség érdekében
- Olvasási haladás monitorizálásának továbbfejlesztése az olvasási kényelem fenntartásának érdekében

Hivatkozások

- [1] Joshua Adamns. *Unconventional Narratives: Nonlinear Storytelling in Modern Publishing*. 2025. URL: <https://www.huntsvilleindependent.com/post/unconventional-narratives-nonlinear-storytelling-in-modern-publishing> (elérés dátuma: 2025. ápr. 25.)
- [2] *ArangoDB*. URL: <https://docs.arangodb.com/3.12/about-arangodb/> (elérés dátuma: 2025. ápr. 26.)
- [3] *Architectural Overview*. URL: <https://docs.flutter.dev/resources/architectural-overview> (elérés dátuma: 2025. ápr. 25.)
- [4] *Bloc*. URL: <https://pub.dev/documentation/bloc/latest/bloc/Bloc-class.html> (elérés dátuma: 2025. ápr. 26.)
- [5] Dart Team. *Dart Documentation*. URL: <https://dart.dev> (elérés dátuma: 2025. ápr. 25.)
- [6] *Flutter Bloc Concepts*. URL: <https://bloclibrary.dev/flutter-bloc-concepts/> (elérés dátuma: 2025. ápr. 26.)
- [7] Flutter Team. *Flutter Documentation*. URL: <https://docs.flutter.dev/> (elérés dátuma: 2025. ápr. 25.)
- [8] *GitLab CI/CD pipelines*. URL: <https://docs.gitlab.com/ci/pipelines/> (elérés dátuma: 2025. ápr. 25.)
- [9] Red Hat. *What is Helm?* URL: <https://www.redhat.com/en/topics/devops/what-is-helm> (elérés dátuma: 2025. ápr. 25.)
- [10] Red Hat. *What is Kubernetes?* URL: <https://www.redhat.com/en/topics/containers/what-is-kubernetes> (elérés dátuma: 2025. ápr. 25.)
- [11] Miracle Onyenma. *What is Rich Text?* URL: <https://buttercms.com/blog/what-is-rich-text/> (elérés dátuma: 2025. ápr. 24.)
- [12] Ana Polo. *Flutter Bloc for Beginners*. 2022. URL: <https://medium.com/flutter-community/flutter-bloc-for-beginners-839e22adb9f5> (elérés dátuma: 2025. ápr. 25.)
- [13] *Spring Boot*. URL: <https://spring.io/guides/gs/spring-boot> (elérés dátuma: 2025. ápr. 26.)
- [14] *StatefulWidget class*. URL: <https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html> (elérés dátuma: 2025. ápr. 25.)
- [15] *StatelessWidget class*. URL: <https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html> (elérés dátuma: 2025. ápr. 25.)