Bevezetés

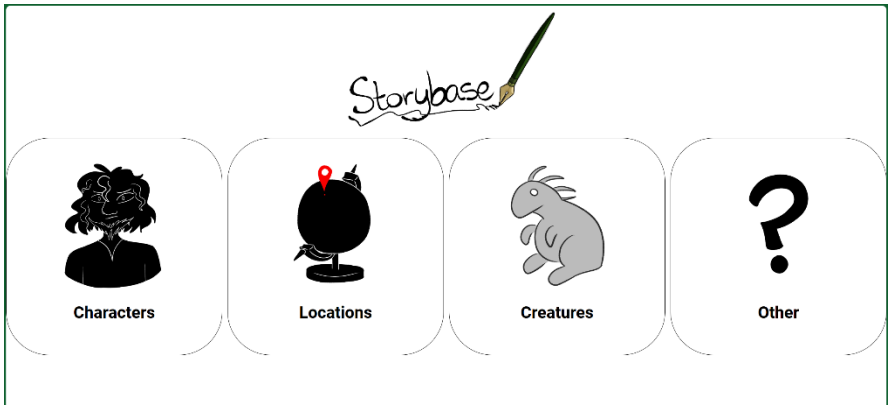
Beadandóként egy olyan weboldalt készítettem, amelynek célja, hogy hozzásegítsen a történetírás folyamatához, főképpen a világfelépítéhez. Az oldalon a történetek különböző elemeit lehet létrehozni és szerkeszteni. Jelenleg ezek három kategóriából kerülnek ki: szereplők, helyek és lények. Ezekhez az elemekhez egy kezdeti sablon tartozik, amely létrehozáskor a kategóriára nézve legfontosabb adatokat tartalmazza. Ezek az elemek bármennyi további adattal kibővíthetőek.

A három előre definiált kategórián túl egy “egyéb” kategória is található, amelyben olyan elemek hozhatóak létre, amelyekhez nem tartozik sablon, és így kezdetben adat sem. Ezeket teljes mértékben a felhasználó építi fel, ezáltal lehetőséget kínálva tulajdonképpen bármilyen történetelem létrehozására.

Frontend

Kezdőoldal (home komponens)

Az első oldal, amellyel a felhasználó találkozik, a következőképpen néz ki:



Itt a felhasználó kiválaszthatja, hogy milyen kategóriába tartozó elemet szeretne létrehozni, megtekinteni, vagy szerkeszteni.

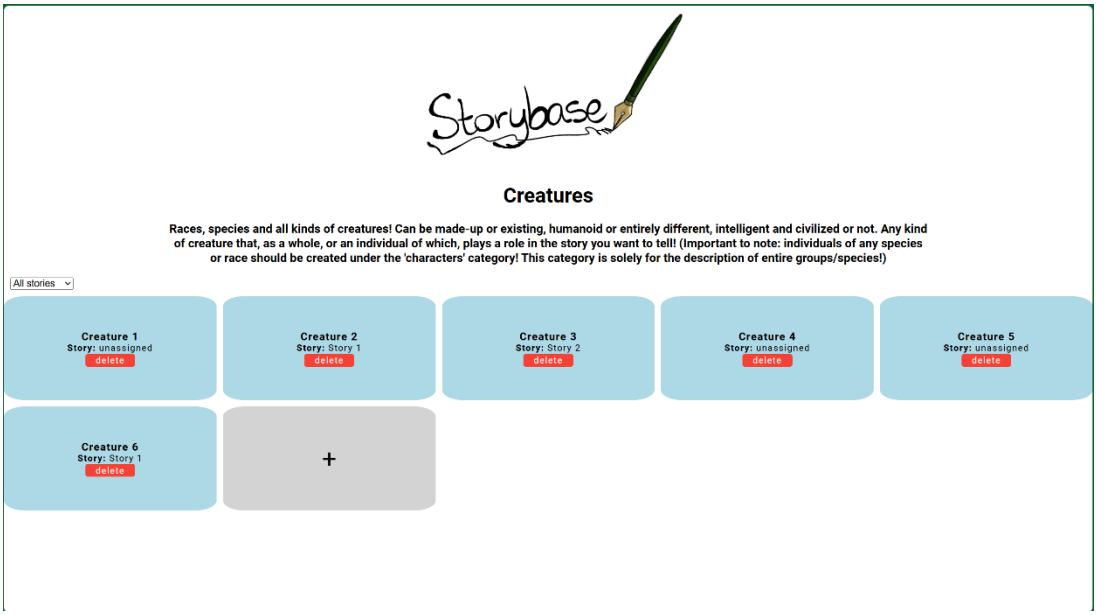
A kezdőoldalhoz a ‘/home’ útvonal és a home komponens tartozik. A weboldal logója az app komponens html-jéhez tartozik, így biztosítva, hogy mindegyik oldal tetején megjelenik. Erre kattintva bármikor visszatérhetünk erre a kezdőoldalra. A home komponens html kódja egészen rövid:

```
<mat-grid-list cols="4" rowHeight="1:1" style="box-sizing: border-box;" gutterSize="10px">
    @for (category of categories; track category){
        <mat-grid-tile mat-button (click)="onClick(category[2])">
            <div class=""tileDiv>
                <div class="iconDiv">
                    <img src="../../assets/{{category[0]}}.png"/>
                </div>
                <div class="titleDiv">
                    <h1>{{category[1]}}</h1>
                </div>
            </div>
        </mat-grid-tile>
    }
</mat-grid-list>
```

A mat-grid-list Angular materialt használtam fel, hogy felosszam a böngészőablakot négy oszlopra, és mindegyik kategóriát megjelenítsek. A kategóriák releváns adatait maga az export osztály tartalmazza: ebben található meg a felhasznált kép neve, a kategória megjelenített neve, és az átirányítási címhez használt azonosító. Bármelyik kategóriára kattintva meghívásra kerül az onClick metódus a megfelelő azonosítóval. A metódus egy egyszerű Router.navigate() hívás, és a '/kategória_azonosító' címre visz.

Elemek kilistázása (list-elements komponens)

Bármely kategóriára kattintunk, az ehhez tartozó már létező elemek kilistázásra kerülnek. A lény kategóriához tartozó oldal például a következőképpen nézhet ki (néhány létrehozott lény után):



A logo alatt a kiválasztott kategória neve látszik egy leírással arról, hogy milyen jellegű elemek kerülhetnek ide. Ezután a kilistázott elemek fölött bal oldalon select elem segítségével szűrhetjük az elemeket, hogy csak a kiválasztott történethez tartozó elemeket lássuk. Ez alapértelmezetten értelemszerűen az összes történet lesz. Végül pedig a kilistázott elemek láthatóak: a felhasználó láthatja a nevüket, és hogy melyik történethez lettek rendelve, illetve törölheti ezeket. Ha valamelyik elemre rákattintunk, szerkezthetjük azt. A lista végén mindig megjelenik a '+' elem, erre kattintva új elemet hozhatunk létre a kiválasztott kategóriában.

Az oldalhoz a '/:category' útvonal és a list-elements útvonal tartozik, ezzel biztosítva, hogy kategóriától függetlenül ugyanaz a folyamat menjen végbe, amikor kilistázásra kerülnek az elemek. Az egyetlen különbség ezek között abban áll, hogy milyen kategória szerint szűrjük az elemeket, amikor az adatbázisból lekérdezésre kerülnek.

A komponens adatainak inicializálásához az ngOnInit metódust használtam.

```
ngOnInit(): void {
  this.route.params.subscribe(
    (params) =>{
      this.type = params['category'];
      this.title = this.texts[this.type as keyof typeof this.texts][0];
      this.description = this.texts[this.type as keyof typeof this.texts][1];

      this.categoryName = this.categoryInfo[this.type as keyof typeof this.categoryInfo][0];
      this.service.GetStories(this.categoryName).subscribe((data) =>{
        this.stories = data;
      });
      this.filterElements("all");
  });
}
```

Itt először lekérdezésre kerülnek az útvonal adatai (ezesetben a kategória azonosítója), majd ennek függvényében meghatározásra kerül a megjelenítendő kategórianév, a hozzá tartozó leírás, és a kategóriaelem szerkesztéséhez szükséges útvonal azonosítója. Ez a kapott azonosító szó egyes számban (tehát szereplők esetén a kapott ‘characters’-hez a ‘character’ fog tartozni). Tulajdonképpen ennek csak annyi jelentősége van, hogy az elemek listázásánál minden elemet látunk a kategóriából, míg szerkesztésnél majd csak egyetlen eggyel dolgozunk.

A megfelelő adatok beállítása után a Service osztály GetStories metódusán keresztül lekérdezésre kerülnek az adatbázisból azok a történetek, amelyekhez tartozik elem az adott kategóriából. Ezek fogják képezni a választási lehetőségeket a történet szerinti szűrőhöz. Végül pedig a Service osztályon keresztül az adatbázistól lekérdezésre kerülnek a kategóriához tartozó elemek, amelyek a komponens elements változójába kerülnek. Ezek kilistázását a html-ben a következőképpen oldottam meg:

```
<div>
    <mat-grid-list cols="5" rowHeight="2:1" gutterSize="10px">
        @for (element of elements; track element; let  i = $index){
            <mat-grid-tile class="elementTile">
                <div>
                    <button mat-button class="elementButton" (click)="editElement(i)">
                        <h3>{{element.name}}</h3>
                        <p><b>Story:</b> {{element.story}}</p>
                    </button>
                    <div class="elementDelete">
                        <button mat-flat-button color="warn" (click)="deleteElement(i)">delete</button>
                    </div>
                </div>
            </mat-grid-tile>
        }
        <mat-grid-tile class="addTile" mat-button (click)="addElement()">+</mat-grid-tile>
    </mat-grid-list>
</div>
```

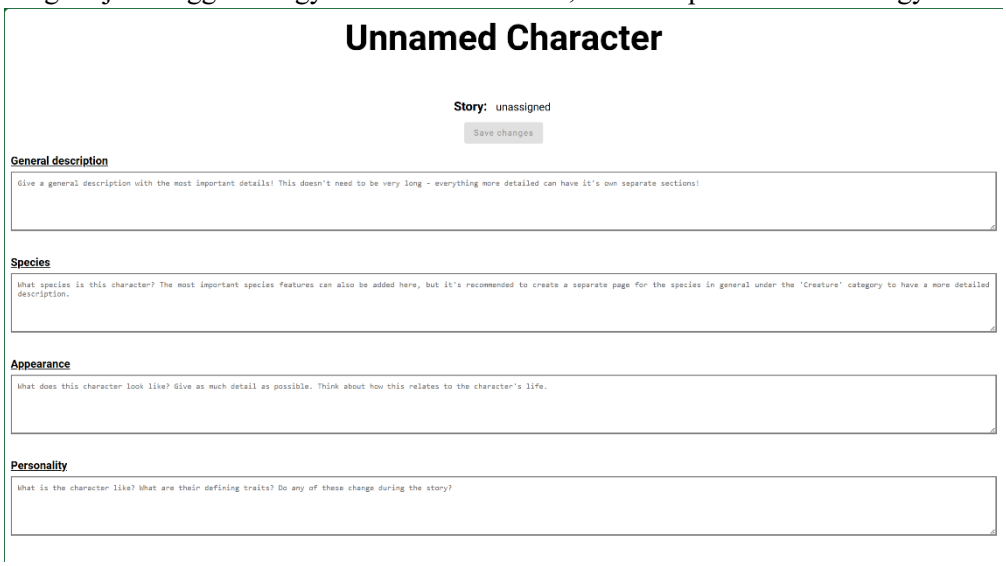
Megint a mat-grid-list segítségével lett felosztva a rendelkezésre álló hely a böngészőablakban, majd egy for ciklussal végigiterálva az elements tömbön minden mat-grid-tile elembe a megfelelő adatok kerültek: az elem neve és a történet, amelyhez tartozik, mint kattintható elem, és a két gombhoz tartozó metódus paraméterei. Ez a paraméter az elem indexe. Végül a lista végére kerül a ’+’ gomb, amellyel új elem adható hozzá a kategóriához.

Az addElement metódus létrehoz egy új elemet az adatbázisban (a Service osztály CreateElement metódusán keresztül), majd eredményként megkapja az újonnan létrehozott elem azonosítóját. Ezzel és a korábban meghatározott szerkesztési kategória-azonosítóval elnavigál a '/kategóriaazonosító/elemazonosító' címre. Az editElement ugyanerre az útvonalra navigál, azonban itt már létező elemről van szó, így a kapott index alapján az elements tömbből kiválasztható a megfelelő elem azonosítója, hogy felhasználható legyen az útvonalban.

A deleteElement először törlési kérést küld az adatbázishoz a Service osztály DeleteElement metódusán keresztül, majd törli az elemet az elements tömbből. Ekkor a megjelenített lista egyből frissül, így a böngészőben is eltűnik a törölt elem.

Új elem vagy elem szerkesztése (edit-element komponens)

Akár új elemet hozunk létre, akár szerkesztünk egy létezőt, az edit-element komponenshez érünk. Ez az elem kategóriájától függően nagyon eltérően nézhet ki, azonban példának itt van egy karakter elem:

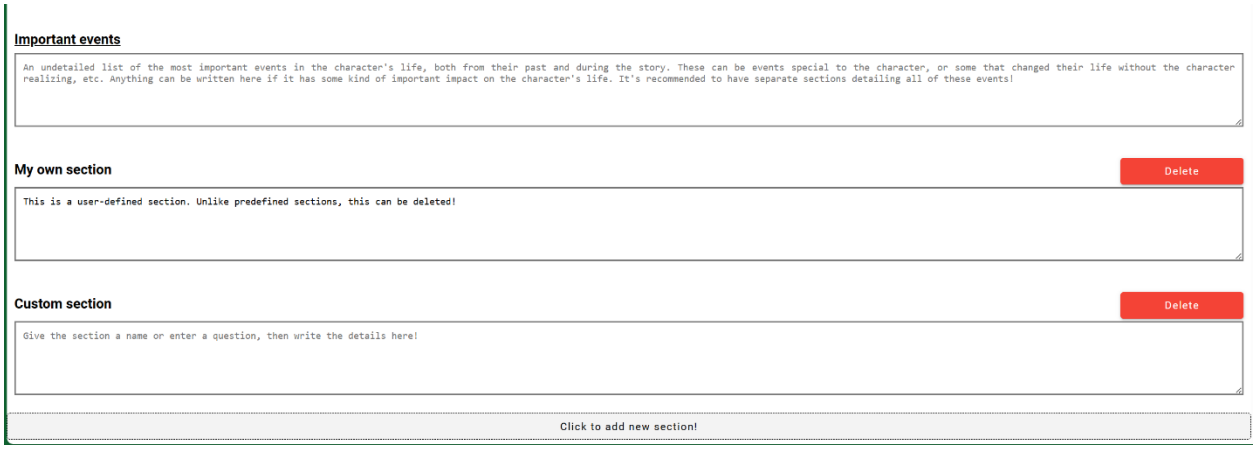


A képen egy újonnan létrehozott karakter elemnek egy részlete látszik. Ilyenkor a hozzá tartozó sablon szerint létrejön az elem, de nincsenek benne adatok, az elem névtelen, és nincs történethez rendelve. A lehetséges hozzátartozó adatok (a sablon) kategóriánként eltérő, de a szerkezet hasonló:

- az oldal tetején (a weboldal logója alatt, amely a fentebbi képen nincs feltűntetve) az elem neve látható; ez kezdetben a 'névtelen (kategória)' értéket kapja
- ez alatt a történet neve látható, amelyhez az elem rendelve lett; ez kezdetben 'unassigned', tehát nincs történethez rendelve
- a következő egy 'Változtatások mentése' gomb, amely eleinte nem kattintható, azonban ha már egyetlen változtatást is elvégzett a felhasználó, kattintható; ekkor a Service osztályon keresztül a teljes elem elküldésre kerül az adatbázisnak, hogy frissüljenek a változások
- végül pedig sorra következnek a megadható adatok cím-szövegdoboz párosokként; az elején az előre definiált (a sablonhoz tartozó) adatok vannak, utánuk pedig hozzáadható saját kiegészítés

- ❖ a szövegdobozokban placeholder-ként segédszöveg található, hogy pontosítsa, milyen jellegű adatot érdemes beleírni, illetve néhány kérdés, ami segíthet elindulni
- ❖ az előre definiált adatok címei nem változtathatóak, a felhasználó által létrehozottaké viszont igen
- ❖ a felhasználó által definiált adatok törölhetőek (az előre definiáltak nem!)
- ❖ az előre definiált adatok nevei alá vannak húzva
- ❖ egyéb (other) kategóriájú elem létrehozásakor ez csak az „Általános leírás” adatot tartalmazza; ekkor ez szerkeszthető című és törölhető

A felhasználó által definiált adatok mindig a lista végére kerülnek:



Ezek kezdetben a ’Saját részleg’ nevet kapják, de ez változtatható, a szövegdobozban pedig egy általános segédszöveg jelenik meg. Az adatlista legvégén egy gomb található, amelyre kattintva új adat adható hozzá az elemhez.

Az elem nevének, a hozzárendelt történetnek és a saját definiálású adat nevének változtatásához a kurzort a szöveg fölé kell vinni, ekkor a szöveg változtathatóvá válik.

A komponens a ’:category/:id’ útvonalhoz tartozik. Mivel az elemek mind egyetlen MongoDB kollekción belül kerülnek tárolásra, a kategória itt már nem is olyan fontos, inkább csak a weboldal-útvonal értelmezhetősége miatt van rá szükség. A komponens inicializálásakor a Service osztályon keresztül lekérdezésre kerül az id azonosítójú elem az adatbázisból a komponens element nevű változójába, majd ennek alapján beállításra kerülnek a megfelelő adatok.

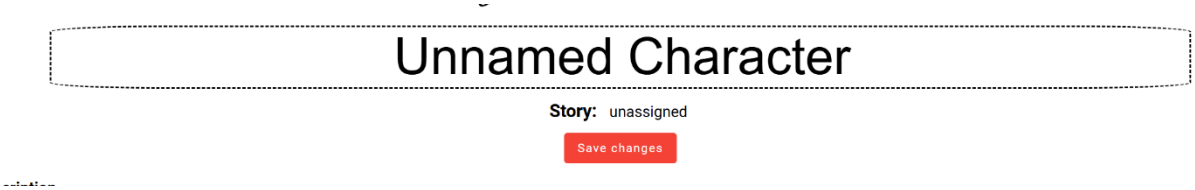
A különböző szerkeszthető elemek értékeinek követésének érdekében a FormGroup, FormArray és FormControl osztályokat használtam. A FormArray-re az adatlisták értékeinek tárolása miatt volt szükség. A FormControlok kezdőértékei mindig a betöltött elemhez tartozó, legutoljára elmentett adatok. Mindig, amikor a felhasználó szerkeszt egy szöveget, meghívásra kerül egy függvény, ami a változtatott értékkel frissíti az elemet lokálisan. Öt ilyen változtatás után, vagy ha az utolsó adatbázis-mentés több mint 5 perce készült, az alkalmazás automatikusan frissíti az elemet az adatbázisban is a Service osztályon keresztül. Ezen kívül még akkor kerül frissítésre az elem, amikor a felhasználó új adatot ad hozzá, vagy kitöröl egyet. Ekkor egyből lokálisan és az adatbázisban is frissül az elem.

Ahhoz, hogy az egyes szövegek szerkeszthetőek legyenek, amikor föléje visszük az egeret, a következő html kódot használtam:

```
<div id="mainDiv">
    <div id="title" (mouseover)="showNameInputWithDelay(true)" (mouseleave)="showNameInputWithDelay(false)">
        <h1 *ngIf="!inputShows.name">{{elementForm.get('name')?.value}}</h1>
        <input *ngIf="inputShows.name" formControlName="name" (change)="contentChange('name')">
    </div>
    <div id="storyFull">
        <div id="storyLabel"><h3>Story:</h3></div>
        <div id="story" (mouseover)="showStoryInputWithDelay(true)" (mouseleave)="showStoryInputWithDelay(false)">
            <ng-container *ngIf="!inputShows.story">{{elementForm.get('story')?.value}}</ng-container>
            <input *ngIf="inputShows.story" formControlName="story" (change)="contentChange('story')">
        </div>
    </div>
    <button mat-flat-button id="saveButton" color="warn" [disabled]="changes==0" (click)="saveButton()">Save changes</button>
</div>
```

A szöveg másképp jelenik meg, amikor nincs szerkesztés alatt, emiatt pedig nem használhattam csak inputot. Amikor nincs szerkesztés alatt a szöveg, az elem neve esetén például egy h3 elemben jelenik. Ehhez a szöveg két állapotát egy div-be csomagoltam. Ha föléje visszük az egeret, akkor kis késleltetéssel egy változó értéke igazra lesz állítva, míg ha elvisszük róla az egeret, hamisra lesz állítva. Ezután *ngIf segítségével kerül eldöntésre a változó értéke alapján, hogy melyik alak kerüljön megjelenítésre. A késleltetésre azért van szükség, mert másképp túl gyorsan reagál az alkalmazás az egérre, és folyton váltakozni kezd a két állapot között.

Egy szöveg kinézetének megváltozása és a körülötte megjelenő keret jelzi, hogy szerkeszthető:



A beírt változtatások egyből megjelennek, ha elvisszük az egeret az elem fölül.

A CategoryElement és Section osztályok

A létrehozható elemekhez a CategoryElement osztály tartozik a következő tulajdonságokkal:

- _id? : string
- category: string
- user: string
- name: string
- story: string
- sections: Section[]

Új elem létrehozásakor az _id nem kerül megadásra, mivel azt az adatbázisba való bevitelkor kapja meg. A category arra szolgál, hogy szűrhető legyen a kategória szerint listázáskor. A user változó felhasználóhoz köti az elemet. A name és story az elem neve és a történet, amihez tartozik. Végül pedig a sections tartalmazza az összes elemhez tartozó adatot.

Egy konstruktort tartalmaz, és három statikus metódust: CreateCharacter, CreateLocation, CreateCreature. A konstruktor önmagában tulajdonképpen egy ’other’ kategóriájú elemet hoz létre, míg a három metódus létrehozza az adott kategória sablonja szerint az új elemet, azaz a sections tömb kerül

másképpen kitöltésre. Mindegyik kategória tartalmazza a ’General description’ Sectiont, azonban ez ’other’ kategória esetén szerkeszthető címmel rendelkezik, és törölhető.

A Section osztály a következő tulajdonságokat tartalmazza:

- title: string
- sectionDescription: string
- content: string
- userDefined: boolean

A title változó az adat nevét tartalmazza. A sectionDescription a szövegdobozba placeholderként bekerülő szövegre vonatkozik, míg a content a ténylegesen beírt tartalomra. Ez az utóbbi új elem esetén mindig üres. A userDefined logikai érték tárolja azt, hogy az adatot a felhasználó definiálta-e, vagy az előre definiált. Ennek alapján dől el, hogy megjelenik-e a törlés gomb a név mellett, illetve hogy szerkeszthető-e az adat neve.

Backend

A Service osztály

Ez az osztály biztosítja az összeköttetést a komponensek és az adatbázis között. A Service osztály http kéréseken keresztül kommunikál a backenddel, és szükség esetén a választ a meghívó komponensnek is elküldi. Ez injektálható osztályként lett definiálva. Hat metódusa van:

- CreateElement(category : string): a kategória függvényében létrehoz egy CategoryElement elemet, majd egy post kérés segítségével elküldi a backendnek; visszakapja a létrejött elem azonosítóját
- UpdateElement(id: string, field: string, newData: any): egy post kérés segítségével kéri az adatbázist, hogy a megadott azonosítójú elem adott mezejét frissítse a megadott új adattal
- GetStories(category : string): get kérésen keresztül lekérdezi az adott kategóriában levő elemeknél szereplő történetcímeket; ez a list-elements komponensben kerül meghívásra, hogy szűrhetőek legyenek az elemek történet szerint; a történetnevek tömbjét adja vissza
- GetElements(category : string, storyFilter : string): get kérésen keresztül lekérdezi az adatbázistól az adott kategóriába és adott történethez rendelt elemeket; ez által válik lehetővé, hogy egyetlen komponenst lehessen rendelni minden elem-kilistázó útvonalhoz; ugyancsak ez végzi el a hozzárendelt történet szerinti szűrést is; visszaadja a talált elemeket
- GetElement(id : string): get kérésen keresztül lekérdezi az adott azonosítójú elemet az adatbázisból, és visszaadja azt
- DeleteElement(id : string): delete kérésen keresztül törölteti az adott azonosítójú elemet az adatbázisból

A db.js file

Ebben a file-ban történik minden adatbáziskezelési művelet. Az adatbázishoz való kapcsolódáshoz és a műveletek elvégzéséhez a NodeJS mongodb drivert használtam fel, míg a http kérések feldolgozásához és a backend applikáció futtatásához az express keretrendszert. Az applikáció csatlakozik az adatbázishoz,

elmenti a referenciát az elemeket tároló kollekcióra, feliratkozik a http kérésekre az elfogadott útvonalakon, majd figyeli az érkező kéréseket a 8080-as porton.

A csatlakozáshoz használt kód:

```
mongo.connect(url).then(
  db => {
    database = db.db("StorybaseDB");
    userCollection = database.collection("Users");
    elementsCollection = database.collection("Elements");
    console.log("Successfully connected to the database!");
  },
  error => {
    console.log(error);
    throw error;
  }
);
```

Az url változóban megadásra kerül a localhost cím (amit itt explicite meg kell adni, mint 127.0.0.1) és a port, amelyen az adatbázishoz kapcsolódjon (esetünkben 27017). A kapcsolódáshoz szükséges ugyanezen a címen valóban el is indítani az adatbázist. Sikeres kapcsolódás esetén rendelkezésre áll egy adatbázisobjektum, amin keresztül kiválaszthatjuk az éppen szükséges adatbázist, és belőle a szükséges kollekciókat.

A kérésekre való figyelés és a válaszok többnyire hasonlítanak egymásra, mivel az útvonalban levő adatok szerint határozza meg az applikáció, hogy melyik elemekkel kell dolgoznia az adatbázisnak, ezért ebben a dokumentumban ezek közül csak egyet ismertetek. Legyen ez például az elemek listázásához szükséges lekérdezés (a Service osztály GetElements metódusában kiküldött kérés):

```
router.route('/elements/:user/:category/:storyFilter').get(async (req, res) => {
    //var userID = await userCollection.findOne({username : req.params.user});
    //userID = userID['_id'];
    var user = req.params.user;
    var category = req.params.category;
    var story = req.params.storyFilter;
    var results;
    if (story == "all") results = await elementsCollection.find({category: category, user: user}).toArray();
    else results = await elementsCollection.find({category: category, story: story, user: user}).toArray();
    res.json(results);
});
```

Az express.Router() metódus által megkapunk a router változóban egy Routert. A route metódusán keresztül adhatjuk meg, hogy melyik útvonalon várunk kérést, majd azt, hogy milyen kérést várunk, és megadunk egy callback függvényt. A feljebbi példán egy get kérést vár az applikáció egy olyan útvonalról, ahol megkapja az aktuális felhasználó nevét, a kategóriát, amelyből az elemeket választani kell, és a történet nevét, ami szerint szűrni kell a találatokat. Ezek a req (request) paraméterből kiolvasásra kerülnek. Ha nem kell szűrni történet szerint (azaz minden történethez rendelt elemet látni szeretnénk), akkor csak a kategória és a felhasználó függvényében keres az applikáció. A találatokat tömbbé alakítja, majd a res.json metóduson keresztül JSON objektumként küldi vissza ezeket a találatokat.

A többi kérésfeldolgozás esetén az eljárás hasonló, csupán annyi változik, hogy a find helyett replaceOne-t vagy deleteOne-t stb. használunk. A replaceOne metódus esetén még azt is megadjuk, hogy mivel legyen frissítve az elem.

Továbbfejlesztési lehetőségek

- Bár a felhasználónév szerinti szűrés elő lett készítve, regisztrációs felület nem készült a weboldalhoz, így ez egy fontos kiegészítés lenne a jövőben
- További kategóriák vezethetőek be, például idővonalak, tárgyak, események, mítoszok, mágiarendszer stb.
- A kategóriákhoz tartozó sablonok bővíthetőek, koncepcionális szinten. Például hasznos lehet lehetőséget teremteni arra, hogy adott kategóriában referenciát lehessen tárolni egy másik elemre
- Jelenleg a felhasználók létre tudnak hozni teljesen saját felépítésű elemet, de ha még szeretnének hasonló szerkezetűt létrehozni, sajátkezűleg kell újra hozzáadni az adatokat. Érdekes lehet bevezetni, hogy a felhasználók maguk is definiálhassanak újrahasznosítható sablonokat. Az alkalmazás messzi jövőjében még akár az is elképzelhető, hogy ezeket a sablonokat megoszthassák egymás között.
- Az elemekben az adatok újrasorrendezhetőekké tehetőek, így lehetőséget teremtve a felhasználók számára, hogy a fontosabbakat vagy gyakrabban szerkesztetteket a lista elejére hozzák
- Az ’egyéb’ kategória pontosítható: ha még nem is definiálható saját sablon, legalább érdemes lehet lehetővé tenni, hogy a nem előre definiált kategóriájú elemeket a felhasználók sorolják valamilyen nem létező kategóriába, hogy szűrhetőek lehessenek