Szakdolgozat

Miskolci Egyetem

JavaScript alapú frontend technológiák összehasonlítása

Készítette:

Zajáros Tamás

Mérnökinformatikus BSc

Témavezetõ:

Piller Imre

Miskolc, 2017

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Zajáros Tamás (I0VATO) mérnökinformatikus jelölt részére.

A szakdolgozat tárgyköre: frontend fejlesztés, JavaScript keretrendszerek

A szakdolgozat címe: JavaScript alapú frontend technológiák összehasonlítása

A feladat részletezése:

A keretrendszerek struktúrájának, mechanizmusainak elemzése. Az azonos problémákra adott megoldások komplexitásának vizsgálata.

Az elérhető JavaScript keretrendszerek (például *ReactJS*, *AngularJS*, *Vue.js*) összehasonlítása. Példaprogramok írása, amelyeken szignifikáns különbség mutatkozik a keretrendszerek megoldásai között.

Témavezetõ: Piller Imre (egyetemi tanársegéd)

A feladat kiadásának ideje:

. . . . . .
szakfelelõs

Eredetiségi Nyilatkozat

Alulírott .....; Neptun-kód: .....
a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzõs .....
szakos hallgatója ezennel büntetõjogi és fegyelmi felelõsségem tudatában nyilatkozom és aláírásommal igazolom, hogy .....
címû szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézõjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezetõ(k)

2. A feladat kidolgozását ellenõriztem:

témavezetõ (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezetõ(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezetõ(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelõs

6. A szakdolgozat osztályzata

a témavezetõ javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

2. fejezet

JavaScript technológiák és keretrendszerek

2.1. MVC keretrendszer

Az MVC (model-view-controller, modell-nézet-vezérlő) egy jól használható módszer arra, hogy hogyan válasszuk szét a felhasználói felületet és az alkalmazás logikát. Az elsődleges cél az, hogy a felhasználói felület megjelenítéséért felelős kódot teljesen elkülönítsük. Ezáltal annak módosítása, kiegészítése nem vonja maga után az alkalmazás logikát megtestesítő kód módosítását, vagy megismétlését.

A módszer lényege az, hogy a hagyományos eljárás alapú programok adatbevitel-adatfeldolgozás-eredmény megjelenítése feladatokat leképezzék a grafikus felhasználói felülettel rendelkező programokban:

adatbevitel – adatfeldolgozás - eredmény megjelenítése

controller – model - view

A vezérlő dolgozza fel a felhasználói adatbevitelt, függvényhívásokká képezi le azokat. Ezek fogják előidézni az adatok módosítását, törlését, vagy a nézetek megváltozását.

Például ha a felhasználó kiválasztja a menü egyik elemét, akkor egy vezérlő fogja meghatározni, hogy ennek hatására mi is történjen. A modell reprezentálja az alkalmazás logikát, feladata az adatok kezelésével kapcsolatos feladatok elvégzése. Ez az egység felelős pl. egy számla áfa tartalmának és végösszegének kiszámolásáért. A modell tudja, hogy melyik vevő fogja kifizetni a számlát, és ha szükséges, azt is, hogy éppen születésnapja van-e ennek a vevőnek.

A nézet feladata a felhasználói felület megjelenítése. Ez az űrlapokat, táblázatokat, linkeket, gombokat, szövegeket jelent. Ezek az elemek megjelenhetnek egy szabványos HTML oldalon.

2.2. Miért kell a kliens oldalra MVC?

A szerver oldalról nézve: A logika mozgatása a klienshez a renderelést lassítja. Többféle probléma van, amit nem lehet szerver oldalon megoldani. Minden fejlesztő meg akarja találni a legjobb egyensúlyt a szerver és a kliens között. Ha nincs meg az egyensúly, akkor marad a logika és a kód duplikálás.

Pl. csinálunk egy form validációt a kliens oldalon, mert az gyors és szépen néz ki, de ugyanazt a procedúrát meg kell valósítanunk a szerveren biztonsági okokból. Az összes kódot a szerver oldalra rakhatod és használhatsz jQuery-t az egyszerű kölcsönhatásokhoz. A kliens oldali MVC szükségessé akkor válik, amikor nem csak HTML-el dolgozol, de adatok vannak az oldaladon.

Példa: Ha ki akarod listázni az összes felhasználót az oldalon ajax lapozással. Minden alkalommal, amikor a Következő oldal gombra kattintassz egy ajax kérés megy a backend-hez az oldal számával és egy HTML jön vissza.

Viszont, ha meg kell jelenítened egy számlálót a felhasználókhoz az oldalon: 10 aktív felhasználó, 2 tiltott felhasználó? A szerver képes renderelni ezt is, de ha a számlálót egy navigációs sávon akarod elhelyezni, az azt jelenti, hogy a szervernek ki kell cserélnie minden oldalt, hogy a számláló megfelelően működjön.

Valahogyan kezelnünk kell a sok adatot, amik a szervertől jönnek: különféle felhasználó információk, cikk gyűjtemények, komment gyűjtemények. Egy olyan eszközre van szükség, ami frissíti a részeket a weboldalon az állapotuk szerint.

CRUD tevékenységek: amikor kezelnünk kell az adatokat a kliens oldalon, természetes, hogy az összes kölcsönhatást a kliensnél készítjük és csak szinkronizáljuk a szerverrel REST felületen keresztül. A felhasználó hozzáadása és törlése nem frissíti az oldalt, mert mi frissíteni tudjuk az oldalt az aktuális adatállapotai szerint.

Erre valóak az MVC (Model View Controller) keretrendszerek.

2.3. Mi a különbség a JavaScript és ECMAScript között?

A JavaScript-et Brendan Eich találta fel 1995-ben, és 1997-ben lett ECMA szabvány. A szabvány hivatalos neve ECMA-262, az ECMAScript pedig a hivatalos neve a nyelvnek.

Év	Név	Leírás
1997	ECMAScript 1	Első kiadás
1998	ECMAScript 2	Csak szerkesztőségi változtatások
1999	ECMAScript 3	Hagyományos kifejezések és try/catch (hibakezelés) hozzáadva
-	ECMAScript 4	Sosem jelent meg
2009	ECMAScript 5	„Szigorú mód" és JSON támogatás hozzáadva
2011	ECMAScript 5.1	Szerkesztőségi változtatások
2015	ECMAScript 6	Osztályok és modulok hozzáadva
2016	ECMAScript 7	Exponenciális operátor (**) és Array.prototype.includes hozzáadva

A JavaScript-et a Netscape nevű cég fejlesztette, az első böngésző, ami futtatni tudta a JavaScript-et a Netscape 2 volt 1996-ban. A Netscape után a Mozilla cége folytatta a fejlesztést a Firefox nevű böngészőhöz. A JavaScript verziószámok 1.0-tól 1.8-ig vannak számozva.

Az ECMAScript az Ecma International cég által lett kifejlesztve, miután feltalálták a JavaScript-et. Az első kiadása 1997-ben jelent meg, a verziószámai 1-től 7-ig vannak sorszámozva. A JScript-et a Microsoft fejlesztette ki 1996-ban, mint egy kompatibilis JavaScript nyelvet az Internet Explorer böngészőjükhöz. A JScript verziószámai 1.0-tól 9.0-ig mennek.

2.3.1. AngularJS

Előnyei:

- Web-es, natív mobil, és asztali alkalmazások fejlesztésére egyaránt használható.
- Előnyei közé tartozik még a kódgeneráció, a kód szétválasztás és az, hogy univerzális.
- Template-eket, Angular CLI-t és Integrált Fejlesztői Környezeteket, úgynevezett IDE-ket használ.
- Van lehetőség benne tesztelésre, animációk használatára.
- Készíthetünk vele megközelíthető alkalmazásokat az ARIA engedélyezett komponensekkel, a beépített a11y teszt infrastruktúrával és a fejlesztői útmutatók alapján.

- Nincs szükség megfigyelő függvények használatára, az Angular analizálja az oldal DOM-ját és felépíti a kötéseket (binds) az Angular specifikus elemek attribútumai alapján. Ez kevesebb kódírást eredményez, a kód tisztább, könnyebb megérteni és kevesebb hiba merül fel.
- Az Angular közvetlenül módosítja a DOM-ot az oldalon, ahelyett, hogy belső HTML kódot adna hozzá, ami lassabb lenne.
- Az adatkötés nem csak minden vezérlőre, vagy értékváltoztatásra érvényesül, hanem a JavaScript kód végrehajtás egyes pontjain is, így nagymértékben növeli a teljesítményt, mint egy egyedüli Modell/Nézet frissítés, ami lecserél többszáz adatváltoztató eseményt.
- Többféle módon meg lehet oldani ugyanazokat a problémákat.
- A Google által támogatott, és nagyszerű fejlesztői közösséggel rendelkezik.
- Támogatott továbbá az IntelliJ IDEA és a Visual Studio .NET IDE-k által.

Hátrányai:

- Nagy és komplikált, több módon meg lehet oldani egy-egy feladatot és nehéz megállapítani, hogy az egyes feladatokhoz melyik a legjobb megoldás.
- Komplex életciklusa van: a fordítás és a linkelés nem újulnak meg, némely esetekben ez zavaró lehet például: rekurzió a fordításban, vagy ütközések a direktívák között.
- Ahogy az idő halad előre, lehet hogy el kell dobni egy meglévő implementációt és egy újat kell létrehozni egy más megközelítésből nézve.
- Több mint 2000 néző késleltetheti az UI (User Interface) megjelenését a böngészőben. Ez limitálja az alkalmazásban használt form-ok, különösen a nagyobb adatlisták és adatrácsok komplexitását.
- Hosszú munkára vár az a projekt, amiben az alkalmazás meghaladja közepes méretű, mérsékelt nehézségű limitet.

2.3.2. Angular 2

- TypeScript alapú
- Komponens alapú
- Több platformon elérhető
- Jobb teljesítmény és gyorsaság
- Router erősítés

2.3.3. Angular 4

- Router ParamMap: Eddig a route paraméterek egy egyszerű kulcs-érték objektum struktúrában voltak tárolva, tehát a hagyományos JavaScript szintaktikával: (paramterObject['parameter-name']) lehetett elérni őket.
- Animációk: A függvényekhez szükséges animációk az Angular v4 megjelenéséig a @angular/core modul által voltak biztosítva. Ha egy alkalmazás animációk használata nélkül készült, az animációkért felelős része a kódnak akkor is hozzátartozott az alkalmazáshoz.
- ngIf direktíva használható „else" ággal is

2.3.4. React Js

- Deklaratív, interaktív UI-k készítését könnyíti meg.
- Egyszerű nézetek tervezése után az adatok változtatása esetén a React frissíti és rendereli a megfelelő komponenseket.
- Komponens alapú: komponensei menedzselik a saját állapotaikat, emiatt bonyolult UI-k tervezésére alkalmas
- A React-ot elég egyszer megtanulni, és azután bárhol lehet azt használni, új funkciókat lehet beépíteni az alkalmazásba anélkül, hogy újraírnánk a meglévő kódot.
- Szerveren is lehet futtatni Node-t használva. Mobil alkalmazásokhoz pedig React Native technológiát használ.

2.3.5. TypeScript

- A TypeScript egy ingyenes és nyilt forráskódú programozási nyelv, amit a Microsoft fejlesztett ki és tart karban.
- Egy szigorú szintaktikai kiterjesztése (superset) a JavaScript-nek és hozzáad választható statikus írást a nyelvhez.
- Anders Hejlsberg, dolgozott a TypeScript fejlesztésén, aki a C# programnyelv vezető fejlesztője és a Delphi-nek és a Turbo Pascal-nak a létrehozója.
- Arra használják, hogy fejlesszenek JavaScript alkalmazásokat a kliens- vagy a szerver oldali (Node.js) végrehajtásra.
- A TypeScript arra lett tervezve, hogy nagy alkalmazásokat fejlesszenek és fordítsanak JavaScript-re. Mivel a TypeScript a JavaScript superset-je, így a JavaScript programok teljesen érvényes TypeScript programok.

- A TypeScript ugyanazzal a JavaScript kóddal kezdődik és végződik, amit több millió fejlesztő ismer ma.
- Létező JavaScript kódot használ, együtt dolgozik népszerű JavaScript könyvtárakkal és a JavaScript-ből hívja meg a TypeScript kódot.
- Egy letisztult, egyszerű JavaScript kódra fordít, amely bármelyik böngészőben képes futni, akár Node.js-ben, vagy bármilyen más JavaScript motorban, ami támogatja az ECMAScript 3-at, vagy annak újabb verzióit.

További funkciói

- statikus ellenőrzés (static checking)
- kód újraírás (code refactoring),
- választható típusok
- típus öröklődés
- aszinkron függvények és dekorátorok

2.3.6. CoffeeScript

- Egy nyelv, ami JavaScript kódra fordít. Egy próbálkozás arra, hogy a JavaScript jó részeire koncentráljunk.
- A lefordított kód olvasható, és tart afelé, hogy legalább olyan gyors, vagy gyorsabb legyen, mint a kézzel írt JavaScript.
- Használhatunk bármilyen létező JavaScript könyvtárat CoffeeScript-ből és fordítva.

2.4. JavaScript implementációk

ECMAScript motorok: olyan programok, amik végrehajtanak olyan forráskódokat, amik az ECMAScript nyelvi standardjaiban, például JavaScript-ben íródtak.

Carakan: Egy JavaScript motor, amit az Opera Software ASA fejleszt, a 10.50-es verziószámú Opera webböngészőben volt, amíg nem váltottak a V8-ra az Opera 15 verziójával, ami 2013-ban jelent meg.

Chakra (JScript9): A JScript motor az Internet Explorer-ben volt használatos. Először a MIX 10-ben lett bemutatva.

Chakra: JavaScript motor, amit a Microsoft Edge-ben használnak.

SpiderMonkey: Egy JavaScript motor a Mozilla Gecko alkalmazásaiban, beleértve a Firefox-ot. Tartalmazza az IonMonkey fordítót és az OdinMonkey optimalizációs modult.

JavaScriptCore: Egy JavaScript értelmező és JIT (just-in-time compilation), a KJS fejlesztése. A WebKit projekt kereteiben belül használatos olyan alkalmazásokban, mint a Safari.

Tamarin: Egy ActionScript és ECMAScript motor, amit az Adobe Flash használ.

V8: JavaScript motor, amit a Google Chrome, a Node.js és a V8.NET használ.

Nashorn: JavaScript motor, az Oracle Java Development Kit (JDK) használja a 8-as verziótól.

3. fejezet

Mintaalkalmazás specifikáció

Egy MMA harcosokkal foglalkozó weboldal, ami megvalósítja a RESTful API-t, képes CRUD műveleteket végrehajtani a harcosokon, mint létrehozás, beolvasás, frissítés és törlés. A harcosok listáját egy kereső mezővel lehet szűrni, így változik az oldalon megjelenített lista a beírt szó hatására. Ha a harcos neve tartalmazza a beírt betűsorozatot, akkor benne marad a listában, ha nem akkor kikerül belőle.

Új harcost az „Add new fighter" feliratú gombbal lehet hozzáadni. Egy harcosnak a következő adatait lehet megadni:

- *név*: A harcos teljes neve, minimum 3 karakter hosszúságúnak kell lennie. Megadása kötelező. Szöveg mező.
- *becenév*: A harcos beceneve, nem kötelező megadni. Szöveg mező.
- *súlycsoport*: Az előre megadott súlycsoportok közül a harcos súlycsoportjának kiválasztása, megadása kötelező, választási lehetőség: 5. Szöveg mező.
- *születési dátum*: A harcos születési dátuma: év, hónap, nap formátumban, megadása kötelező. Dátum mező.
- *szülőváros*: A harcos szülőváros, nem kötelező megadni. Szöveg mező.
- *nemzetiség*: A harcos nemzetisége: minimum 2 karakter, megadása kötelező. Szöveg mező.
- *magasság*: A harcos magassága centiméterben értve, 155 és 205 közé kell, hogy essen, megadása kötelező. Szám mező.
- *súly*: A harcos súlya kilogrammban értve, 50 és 120 közé kell, hogy essen, megadása kötelező. Szám mező.
- *record*: A harcos rekordja Győzelem-Döntetlen-Vereség formában, megadása kötelező.

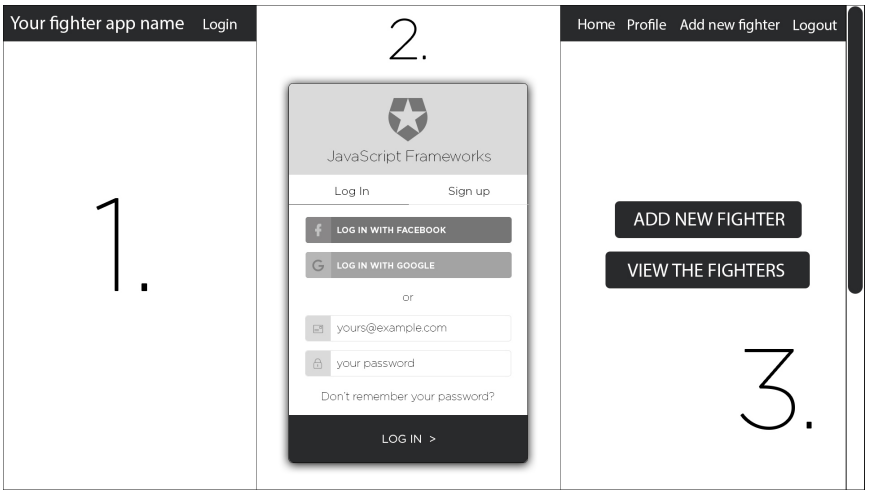
- *harcos avatárjának URL linkje*: Szöveg mező, megadása kötelező.
- *a harcossal foglalkozó oldal URL linkje*: Szöveg mező, nem kötelező megadni.

A harcosok a főoldalon (home) lévő „VIEW THE FIGHTERS” elnevezésű gombra kattintva jelennek meg. Minden harcos sorában egy „View Details” nevezetű kék gomb van, ami átirányít a /fighter/details/:id oldalra, ahol megtekinthetők az adott harcos további adatai. Továbbá ezen az oldalon tudjuk frissíteni az adatokat az Edit gombra kattintva.

Törölni a „Delete” nevű gombbal lehet. A Go Back nevezetű gombbal a harcosokat megjelenítő oldalra tudunk visszanavigálni. A kitöltendő űrlap (form) az „Create” (létrehozás) esetén validációval van ellátva. A fent említett táblázatban szerepelnek a kitöltendő mezők leírásai, illetve megadásuknak a megszorításai. A back-end szolgáltatást a MongoDB adatbázis, a Node.js, mint szerver és a legnépszerűbb Node.js keretrendszer, az Express biztosítja, ami a HTTP kérések irányításáért felel.

Front-end részről, mint személyes preferált keretrendszer az AngularJS és Angular 2. Ezeken kívül a React.js, és a Vue.js került implementálásra.

Az alkalmazás elindítása után a "Login" gombra kattintva felugrik az Auth0 szervíz bejelentkező és regisztrációs képernyője, ahol a felhasználó létrehozhat egy új fiókot, vagy bejelentkezhet a Facebook vagy a Google szolgáltatással, ami után a program átirányítja a főoldalra. (3.1. ábra).

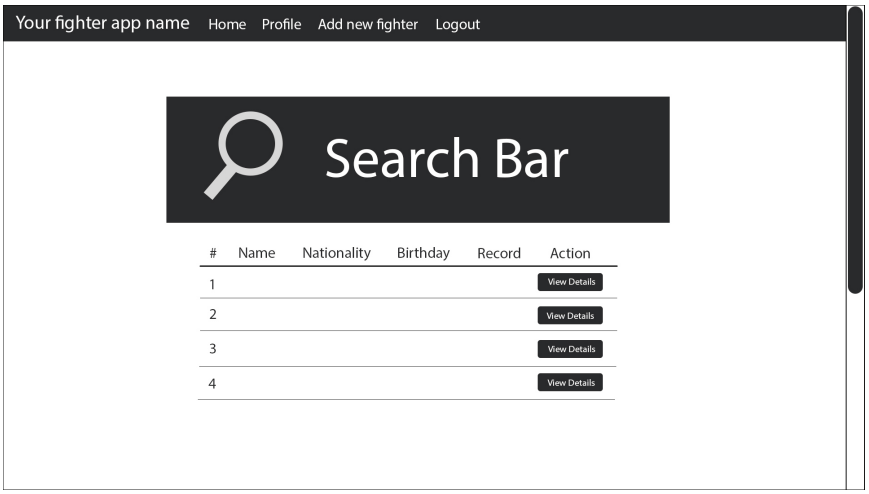


3.1. ábra. Főoldal (Home)

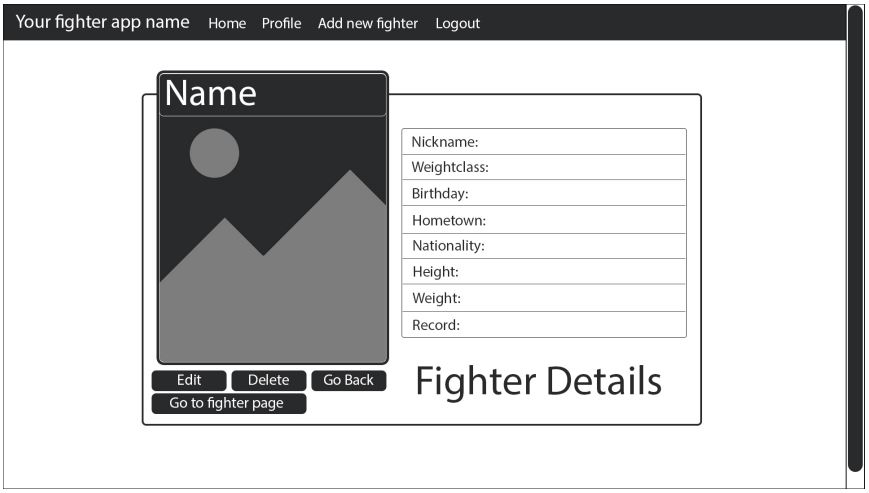
A VIEW THE FIGHTERS feliratú linkre kattintva a /fighters oldal jelenítődik meg a harcosok listájával (3.2. ábra).

A harcos sorában lévő View Details feliratú linkre kattintva az adott harcos adatait tartalmazó oldal jön be /fighters/details/id formában (3.3. ábra).

A details oldalon az "Edit" gombra kattintva a program átirányítja a felhasználót a /fighters/edit/:id oldalra, ahol a harcos adatait tudja frissíteni (3.4. ábra).

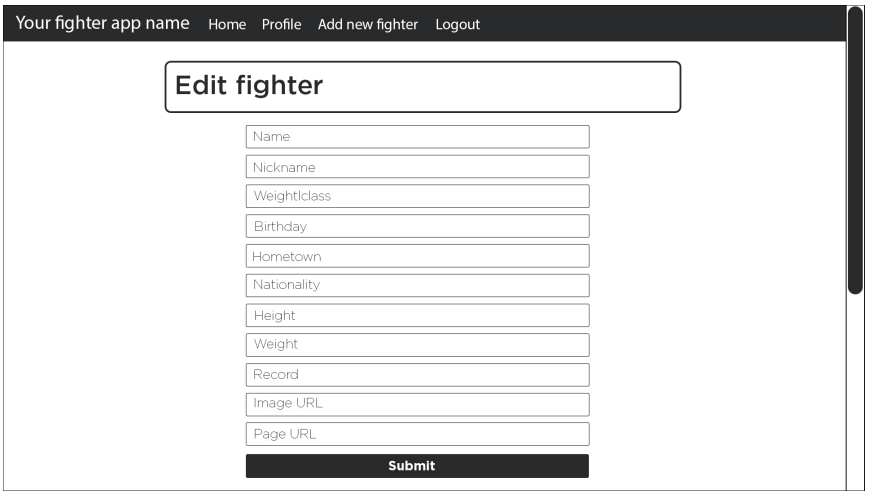


3.2. ábra. Harcos lista oldal (Fighters)

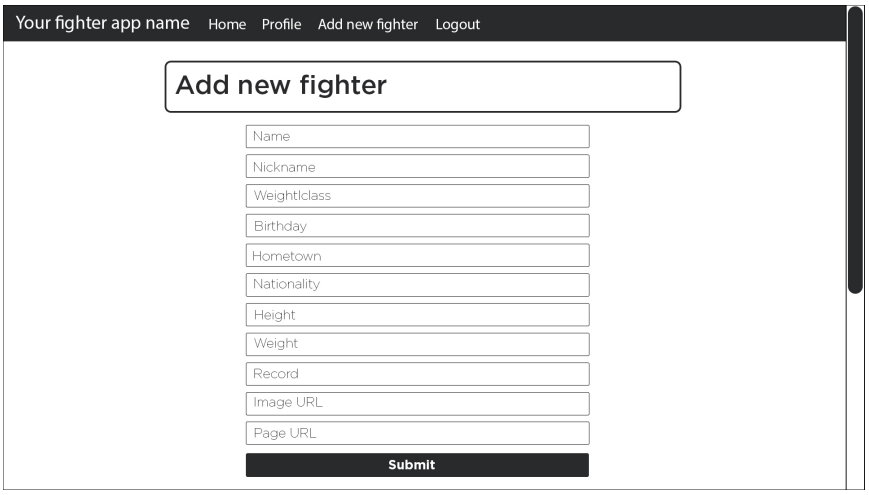


3.3. ábra. Harcos adatait megjelenítő oldal (Fighter Details)

Végül ha a felhasználó bejelentkezés után (3.1. ábra), az ADD NEW FIGHTER feliratú gombra kattint, akkor a /fighters/add oldal ugrik fel (3.5. ábra).



3.4. ábra. Harcos adatait változtató oldal (Edit Fighter Details)



3.5. ábra. Új harcost létrehozó oldal (Add new fighter)

4. fejezet

AngularJS implementáció

CRUD műveletek megvalósítása: (CREATE, READ, UPDATE, DELETE) – Létrehozás, Lekérés, Frissítés, Törlés Az web alkalmazásban ezeket a funkciókat a MMA harcosokon lehet végrehajtani.

A szerver által nyújtott REST API megvalósítása biztosítja a kód működőképességét.

A harcosokon végzett műveleteket a fighters.js nevű fájl tartalmazza.

Ahhoz, hogy minden megfelelően működjön létre kell hozni egy modult az alkalmazásban:

```
var app = angular.module('app');
```

Ezután egy hozzátartozó kontrollert:

```
app.controller('FightersController', ['$scope', '$http', '$location',
'$stateParams',  '$state', function($scope, $http, $location,
$stateParams, $state){
}]);
```

Ebbe a kontrollerbe tehetjük be a különböző funkciókat: A harcosok szerverről való lekérése:

```
$scope.getFighters = function(){
$http({
        method: 'GET',
        url: '/api/fighters'
        }).then(function successCallback(response) {
        $scope.fighters = response.data;
        }, function errorCallback(response) {
});}
```

Ehhez egy GET kérést szükséges elküldeni a szervernek, ami visszaküldi egy response objektumban ez esetben a fighters objektumot, ami a harcosok adatait tartalmazza.

A /api/fighters URL-en lévő adatokat adja vissza, amik JSON formátumban vannak eltárolva.

A következő példakód egy harcos hozzáadását mutatja be, ehhez egy POST HTTP kérés elküldésére van szükségünk:

```
$scope.addFighter = function(){
                console.log($scope.fighter);
                $http({
                        method: 'POST',
                        url: '/api/fighters/',
                        data: $scope.fighter
                }).then(function successCallback(response) {
                        $state.go("fighters");
                }, function errorCallback(response) {
                });
        }
```

Miután a szerver sikeresen teljesítette a kérést, a `$state.go("fighters");` kódsor visszairányítja a felhasználót a 'fighters' nevű route-ra, ami a /fighters oldallal egyezik meg.

Egy harcos adatainak frissítéséhez már szükség van a frissíteni kívánt harcos szerveren tárolt id mezőjére.

```
$scope.updateFighter = function(){
                var id = $stateParams.id;
                $http({
                        method: 'PUT',
                        url: '/api/fighters/' +id,
                        data: $scope.fighter
                }).then(function successCallback(response) {
                        $state.go("fighters");
                }, function errorCallback(response) {
                });
        }
```

Ezt egy változó létrehozásával tehetjük meg, amiben letároljuk az id-t a `$stateParams` segítségével, ami az URL-ből olvassa ki az id-t. Sikeres végrehajtás után a szerver ismét visszairányít a 'fighters' nevű route-ra, tehát a /fighters oldalra, hogy lássuk a végrehajtott változtatásokat.

Egy harcos törléséhez szintén szükségünk van az id-jére, amit a megfelelő HTML oldalon lévő törlés gombnál adunk meg neki, ami a megfelelő harcos id-jével hívja meg a removeFighter nevű függvényt.

```
$scope.removeFighter = function(id){
                $http({
                        method: 'DELETE',
                        url: '/api/fighters/' +id,
                        data: $scope.fighter
                }).then(function successCallback(response) {
                        $state.go("fighters");
                }, function errorCallback(response) {
                });
        }
```

A következő funkció a harcosok táblázatának betűsorozatra való leszűkítése. Ekkor, ha a felhasználó egy kereső mezőbe begépel egy betűt, vagy betűsorozatot, amit az egyik, vagy több harcos neve tartalmaz, akkor a táblázat, ami megjeleníti a harcosokat dinamikusan szűkül le, és mutatja azt, vagy azokat a harcosokat, akinek, vagy akiknek a nevére illik a keresőmező tartalma.

```
<div id="search">
  <input type="text" ng-model="search.name"
  placeholder="Search fighters..."/>
</div>
```

Az kereső mezőn kívül szükség van egy filterre is, amit a `<table> <tbody>` részének első sorába írhatunk be: ez jeleníti meg az egyes harcosok nevét, becenevét, és egy View Details feliratú linket, amire kattintva megtekinthetők az adott harcos adatai.

```
<tbody>
        <tr ng-repeat="fighter in fighters | filter:search">
                <td>{{ fighter.name }}</td>
        <td>{{ fighter.nickname }}</td>
        </tr>
</tbody>
```

4.1. Form validáció

A form validáció többféle módon valósítható meg. Az egyik megvalósítási lehetőség a `$valid` és `$invalid` state-ekkel (állapot):

```
<form name="fighterForm" ng-submit="addFighter(fighterForm.$valid)"
novalidate>
```

A novalidate kulcsszó ahhoz szükséges, hogy kikapcsoljuk a HTML5 alapvető validációs funkcióját.

```
<button type="submit" ng-disabled="fighterForm.$invalid"
                class="btn btn-default">Submit</button>
```

Ha a form még nem érvényes (valid) akkor a Submit (elküldés) gombra való kattintás letiltásra kerül, így a felhasználó nem tudja elküldeni a form-ot.

Szöveg mezők validációját a következőképpen valósítottam meg:

```
<div class="form-group" ng-class="{ 'has-error' :
fighterForm.name.$touched && !fighterForm.name.$dirty ||
fighterForm.name.$invalid && !fighterForm.name.$pristine }">
<label>Name*</label>
<input type="text" class="form-control" name="name"
ng-model="fighter.name" placeholder="Name" ng-minlength="3" required/>
<p ng-show="fighterForm.name.$touched && !fighterForm.name.$dirty &&
!fighterForm.name.$error.minlength || fighterForm.name.$invalid &&
!fighterForm.name.$pristine && !fighterForm.name.$error.minlength"
class="help-block">Name is required.</p>
<p ng-show="fighterForm.name.$error.minlength" class="help-block">
    Name is too short.</p>
</div>
```

Az alkalmazás akkor írja ki a validációs hibaüzenet, ha egyszerre teljesülnek az alábbi feltételek:

- az input mezőbe kattint a felhasználó, de nem ír a mezőbe semmit
- vagy ha a mező értéke nem érvényes, pedig már írt bele

A hibaüzenet a mezőből való kikattintás után jelenik meg. A név mező kötelező hibaüzenet (Name is required.) csak abban az esetben jelenik meg, ha a mező üres, tehát a minimum karakterszám nem teljesülése még nem lehet hiba.

Amint a felhasználó beleír valamit a mezőbe, a „Name is required” hibaüzenet eltűnik és a Név túl rövid (Name is too short.) hibaüzenet jelenik meg egészen addig, amíg a név mező karaktereinek száma el nem éri a 3-at.

A has-error ng-class (AngularJS osztály) hiba esetén a hibaüzenetnek piros színnel és a beviteli mezőnek piros szegéllyel való megjelenítéséért felelős.

A szám mezőknél a min és max direktívákat használtam:

```
<label>Height*</label>
<input type="number" class="form-control" name="height"
ng-model="fighter.height" placeholder="Height" min="155" max="205"
required/>
```

Az előző példához hasonlóan a „Height is required” hibaüzenet nem jelenik meg, csak üres mező esetében. Ezután, ha a beírt számérték nem 155 és 205 közé esik, akkor a „Height value must be between 155 and 205 cm.” hibaüzenet jelenik meg.

A record mezőnél az ng-pattern direktívával szűkítettem le a lehetséges megadható eseteket:

```
<label>Record*</label>
<input type="text" class="form-control" name="record"
ng-model="fighter.record" placeholder="Record"
ng-pattern="/^[0-9]{1,2}-[0-99]{1,2}-[0-99]{1,2}$/" required/>
```

Ezzel elérhető, hogy csak akkor legyen érvényes a record mező értéke, ha az [0-999]-[0-999]-[0-999] formátumú. Ezt a mezőt is kötelező kitölteni így itt is csak akkor jelenik meg a „Record is required” hibaüzenet, ha még nem írt a felhasználó a mezőbe.

Ezután ha nem a fent említett formátumban írt a felhasználó a mezőbe, akkor a „The record must be in Wins-Draws-Losses form.” hibaüzenet jelenik meg, ami azt jelenti, hogy a rekord mezőt győzelem-döntetlen-vereség formában kell kitölteni, például: 30-2-3.

Az image_url-hez hozzáadott

```
ng-pattern="/^https?://.+$/"
```

direktívával pedig a harcos avatarjának URL link validációját készítettem el, ami ha a felhasználó nem http:// vagy https://-el kezdődő URL címet ad meg, akkor „Image URL must be valid.” hibaüzenetet kap.

4.2. Routing

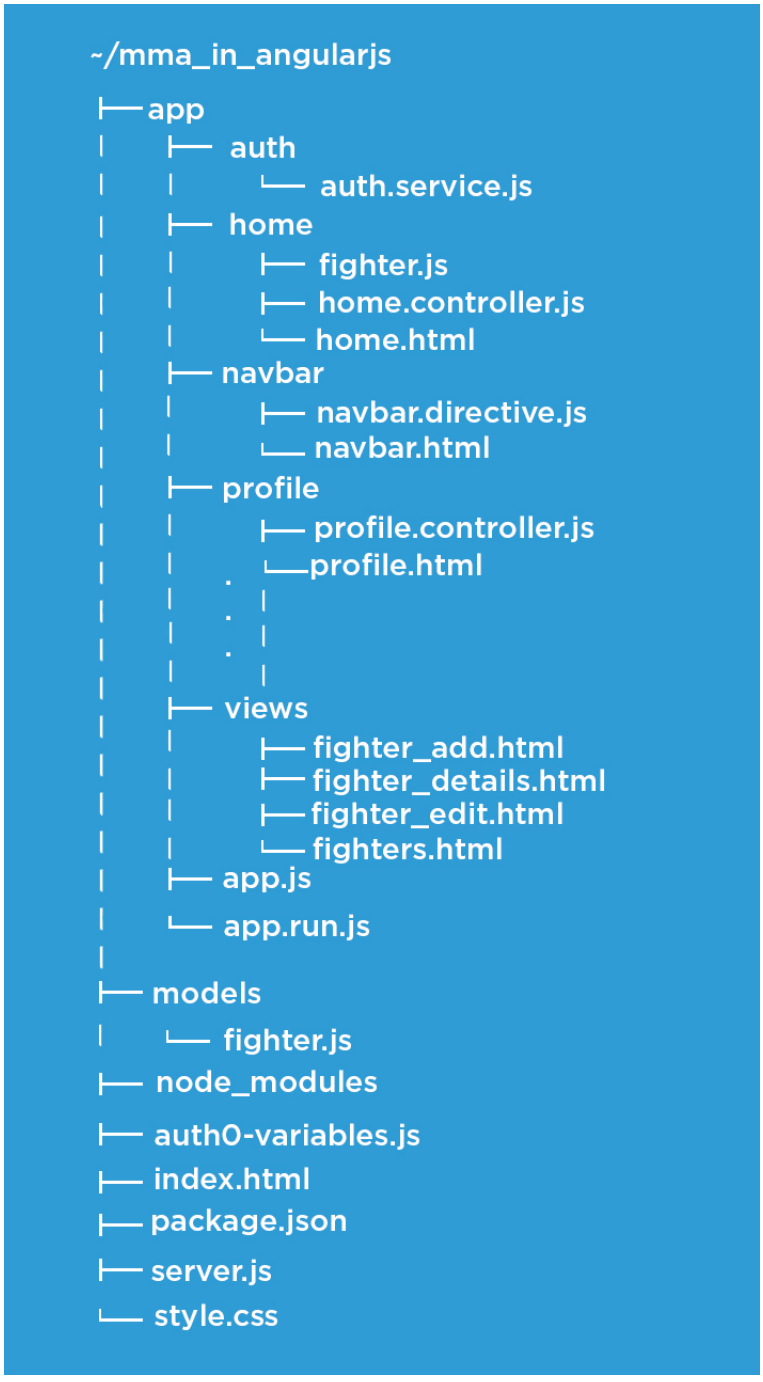
Az oldalak közötti routing az ui.router szervízzel van megoldva. Ehhez az angular module-ba meg kell adnunk az ui.router-t, mint függőséget (dependency).

```
angular
    .module('app', ['ui.router']);
```

Az alábbi kód /fighters oldal megjelenítését biztosítja, a FightersController biztosítja a funkciókat, a /fighters oldalra való navigációkor pedig az alkalmazás a fighters.html template-t tölti be.

```
$stateProvider
.state('fighters', {
        url: '/fighters',
        controller: 'FightersController',
        templateUrl: 'app/views/fighters.html',
        controllerAs: 'vm' });
```

4.3. Projekt struktúra



4.1. ábra. Az AngularJS projekt struktúrája

5. fejezet

Angular2 implementáció

CRUD műveletek: A műveletek megvalósításához szükséges funkciókat a szervíz tartalmazza, ebben az esetben a FighterService.

```
@Injectable()
export class FighterService {
  constructor(private http: Http) { }
  getAllFighters() {
    return new Promise((resolve, reject) => {
      this.http.get('/fighter')
        .map(res => res.json())
        .subscribe(res => {
          resolve(res);
        }, (err) => {
          reject(err);
        });
    });  }  }
```

A getAllFighters() GET kérést küld a szervernek. Ennek a függvénynek a segítségével kérhetők le az eltárolt harcosok. Egy harcos adatainak frissítésére szolgáló függvény az

```
updateFighter(id, data) {
    return new Promise((resolve, reject) => {
        this.http.put('/fighter/'+id, data)
          .map(res => res.json())
          .subscribe(res => {
            resolve(res);
          }, (err) => {
            reject(err);
          });
    });
```

```
  }
```

A HTML oldalakat külön komponensek vezérlik:

Ezeket a komponenseket az „ng g component komponensnév” paranccsal lehet létrehozni, ami automatikus legenerálja a szükséges fájlokat és hozzáadja az új komponenst az app.module.ts fájlhoz.

```
@Component({
  selector: 'app-fighter',
  templateUrl: './fighter.component.html',
  styleUrls: ['./fighter.component.css']
})
export class FighterComponent implements OnInit {
  fighters: any;
  constructor(private fighterService: FighterService) { }
  ngOnInit() {
    this.getFighterList();
  }
  getFighterList() {
    this.fighterService.getAllFighters().then((res) => {
      this.fighters = res;
    }, (err) => {
      console.log(err);
    });}}
```

A „fighters” oldal, ami a harcosok megjelenítésére szolgál egy komponens, a hozzátartozó template-t a templateUrl kifejezéssel tudjuk megadni. A kinézetét a megfelelő nevű .css kiterjesztésű fájl tartalmazza, amit a styleUrls kifejezéssel adhatunk meg.

Ez az oldal a getFighterList() nevű függvény használatával kéri le a szerveren tárolt harcosokat.

A harcosok táblázatának keresőmezővel való szűrése: A harcosok táblázatának szűréséhez egy pipe-ra van szükségünk:

```
export class FilterPipe implements PipeTransform {
  transform(fighters: any, search: any): any {
    if (search === undefined) return fighters;
    return fighters.filter(function(fighter){
        return fighter.name.includes(search);  })}}
```

Ezenkívül szükség van még egy kereső mezőre:

```
<input type="text" [(ngModel)]="search"
placeholder="Search fighters..."/>
```

Használata a | filter:szűrőmező neve paranccsal lehetséges:

```
<tbody>
      <tr *ngFor="let fighter of fighters | filter:search">
        <td>{{ fighter.name }}</td>
        <td>{{ fighter.nickname }}</td>
      </tr>
    </tbody>
```

Továbbá egy import-ra, amit az app.module.ts fájlhoz kell hozzáadnunk:

```
import { FilterPipe } from './filter.pipe';
```

Majd ugyanebben a fájlban az @NgModule declarations részéhez hozzá kell adni a FilterPipe osztályt.

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    FighterComponent,
    FilterPipe
```

5.1. Routing

A routing-hoz szükség van a

```
import { RouterModule } from '@angular/router';
```

sorra, amit az app.module.ts nevű fájlban kell megadnunk. Ezenkívül szükség van még az app.component.html fájlban a <router-outlet></router-outlet> tag-okra.

Route-k (útvonalak) megadása az app.routes.ts fájlban és az

```
import { ROUTES } from './app.routes';
```

sorra az app.module.ts fájlban.

```
export const ROUTES: Routes = [
  { path: ' ', component: HomeComponent },
  { path: 'fighters', component: FighterComponent },
  { path: '**', redirectTo: ' ' }
];
```

Ez a kódrészlet azt mutatja be, hogy ha a felhasználó az URL-ben nem ír be semmit a / jel után, akkor az alkalmazás a HomeComponent nevű komponens kapja meg a vezérlést, ami a home.component.html nevű fájlt tölti be.

```
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
```

A /fighters oldal pedig a FighterComponent-ben meghatározott templateUrl-t és styleUrls-t tölti be.

A lapok közötti navigálás linkekkel történik: A /fighter-create oldalról a /fighters oldalra való visszalépés esetén:

```
<a [routerLink]="['/fighters']">Go back</a>
```

5.2. Form validáció

A form validációnál a disabled-re (letiltott) állítottam a Submit gombot:

```
<button type="submit" class="btn btn-success"
[disabled]="!fighterForm.form.valid">Submit</button>
```

Így ha a fighterForm nevű űrlap (form) nem valid, akkor a Submit gomb ki van kapcsolva.

Mezők validálása az Angular 2 által biztosított direktívákkal történik:

```
<form (ngSubmit)="saveFighter()" #fighterForm="ngForm">
```

A form fejlécében meg kell adni a form nevét, ez esetben „fighterForm”.

Majd a Submit gomb letiltása a [disabled]="!fighterForm.form.valid" sor Submit button-hoz való hozzáadásával érhető el, ami letiltja a gombot, ha a fighterForm nevű form nem érvényes (valid).

```
<button type="submit" class="btn btn-success"
[disabled]="!fighterForm.form.valid">Submit</button>
```

A name mező validálásához meg kell adni a kívánt direktívákat, ebben az esetben, hogy a mező kitöltésekor minimum 3 karakter hosszúságú nevet kell beírnia a felhasználónak. (minlength=”3”)

A „required” jelző pedig a mező kitöltését követeli meg a felhasználótól.

```
<label for="name">Name*</label>
<input type="text" class="form-control" [(ngModel)]="fighter.name"
name="name" id="name" #name="ngModel" required minlength="3">
<div *ngIf="name.errors && (name.dirty || name.touched)"
```

```
        class="alert alert-danger">
        <div [hidden]="!name.errors.required">
        Name is required!
    </div>
    <div [hidden]="!name.errors.minlength">
        Name must be at least 3 characters long.
    </div>
</div>
```

Ha a name mező hibás és már írtak bele, vagy csak belekattintottak, akkor a mező alatt megjelenítődik az épp aktuális hibaüzenet.

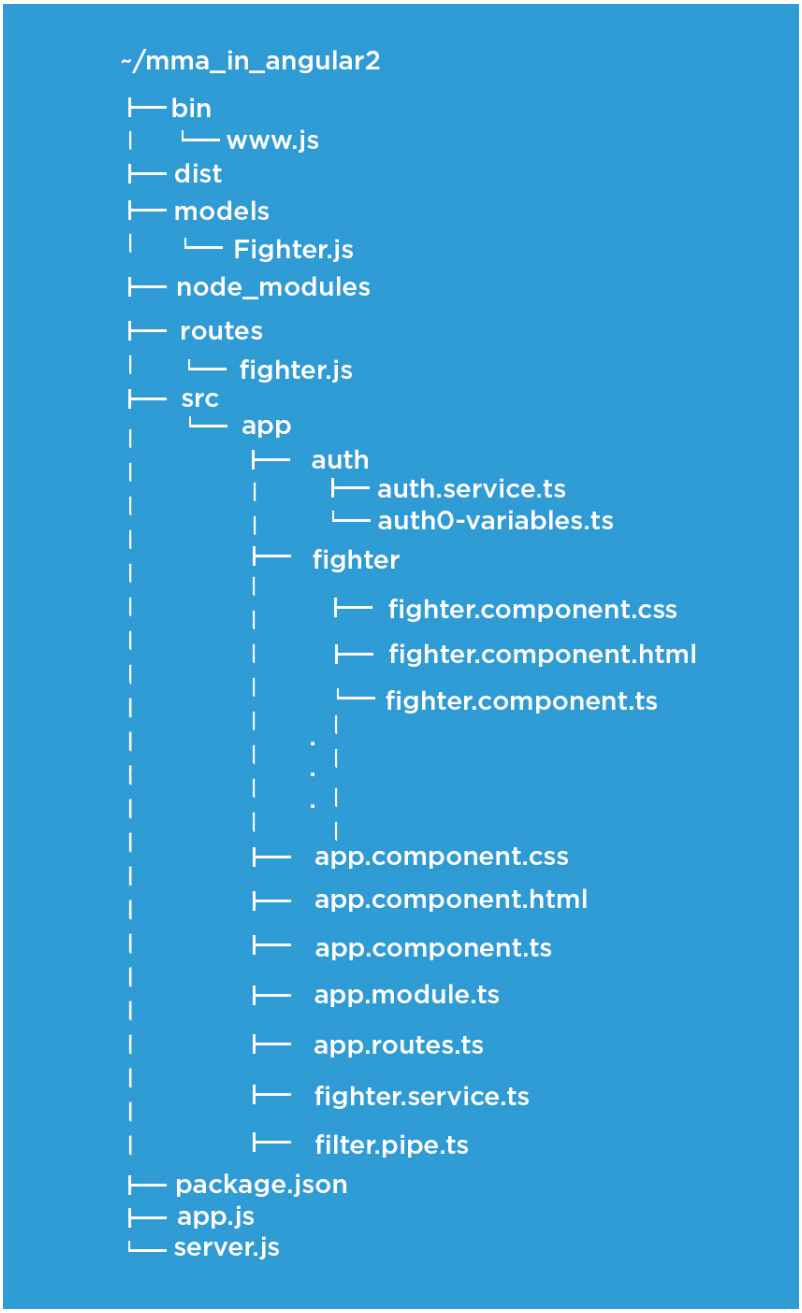
A record mező validálásához pattern-t kell hozzáadni a mezőhöz:

```
<label for="record">Record*</label>
<input type="text" class="form-control" [(ngModel)]="fighter.record"
name="record" id="record" #record="ngModel"
pattern="[0-9]{1,2}-[0-9]{1,2}-[0-9]{1,2}" required>
<div *ngIf="record.errors && (record.dirty || record.touched)"
class="alert alert-danger">
<div [hidden]="!record.errors.required">
        Record is required!</div>
<div [hidden]="!record.errors.pattern">The record must be in
Wins-Draws-Losses form.</div></div>
```

Ha a beírt adatok nem egyeznek meg a pattern-nel, akkor a felhasználó a „The record must be in Wins-Draws-Losses form.” hibaüzenetet kapja.

A harcos avatarjának image_url mezőben megadott URL linkhez tartozó validációnál itt is a pattern="https?://.+" direktívát kell a beviteli mezőhöz hozzáadni, ezután, ha a felhasználó az Image URL linket nem a megfelelő formában adja meg, az „Image URL must be valid!” hibaüzenetet kapja.

5.3. Projekt struktúra



5.1. ábra. Az Angular2 projekt struktúrája

6. fejezet

VueJS implementáció

CRUD műveletek megvalósítása a services/FightersService.js fájlban:

```
export default {
  fetchFighters () {
    return Api().get('fighters')
  },
  addFighter (params) {
    return Api().post('fighters', params)
  },
  updateFighter (params) {
    return Api().put('fighters/' + params.id, params)
  },
  getFighter (params) {
    return Api().get('fighter/' + params.id)
  },
  deleteFighter (id) {
    return Api().delete('fighters/' + id)
    this.fetchFighters();
  }
}
```

A fetchFighters nevezetű függvény visszaadja az Api().get('fighters') kérést, amit az axios modul végez el, ami egy promise alapú http kliens. Ehhez szükséges importálni a FightersService.js fájlban a services/Api.js fájlt, ami biztosítja az axios modult.

```
import Api from '@/services/Api'
```

A getFighter(), és az updateFighter() függvények az URL címből veszik át a harcos id-ját.

A deleteFighter() függvény pedig miután megkapta az id-t, meghívja a fetchFighters() nevű függvényt, ami újra lekéri az aktuálisan eltárolt harcosokat.

Api.js

```
import axios from 'axios'
export default() => {

  return axios.create({
    baseURL: `http://localhost:8081`
  })
}
```

6.1. Routing

Szükséges hozzá importálnunk a vue-router-t és használni:

```
import Router from 'vue-router'
```

Majd a routing-ok (útvonalak) megadásánál a Vue.use(Router) sorral adjuk meg, hogy a program használja Router-ként a beimportált „vue-router" nevezetű modult.

```
export default new Router({
mode: 'history',
  routes: [
{
      path: '/fighters',
      name: 'Fighters',
      component: Fighters
  },
  {
      path: '/fighters/new',
      name: 'NewFighter',
      component: NewFighter  }]})
```

Az alapvető beállítás a vue-router modulban az úgynevezett „hash mode", ami az URL hash-t (#) használja a teljes URL cím reprezentálására, így ha az URL változik, az oldal nem fog újra lefrissülni.

A mode: 'history' használatával a vue-router-t „history mode"-ban használhatjuk, amivel megszabadulhatunk a hash-től az URL címekben. Ez a mód arra használja a history.pushState API-t, hogy elérje, hogy a felhasználó navigálhasson az URL-ek között, anélkül, hogy az oldal újra lefrissülne. Ehhez megfelelő szerver konfiguráció szükséges, különben a felhasználó egyes oldalak elérésekor 404-es hibaüzenetet kaphat.

A path definiálja az oldal URL címét, a component mutatja meg, hogy melyik .vue fájl tartalmát kell megjeleníteni az URL-re való látogatáskor. Ehhez a megadott

komponenseket importálnunk kell.

```
import Fighters from '@/components/Fighters'
import NewFighter from '@/components/NewFighter'
```

A /fighters/new URL megnyitásakor a NewFighter.vue komponens töltődik be, ami tartalmazza a megjelenítendő template-t, amiben egy új harcos létrehozásához szükséges form található.

6.2. Form validáció

A form validálást többféle módon történhet, az egyik ilyen a vee-validate nevezetű modul használatával történő validáció. Ehhez először telepíteni kell a modult npm csomagkezelő, vagy CDN segítségével, majd az alábbiakat kell importálni az alkalmazás main.js fájljában:

```
npm install vee-validate --save
import Vue from 'vue'
import VeeValidate from 'vee-validate'
Vue.use(VeeValidate)
```

A name mező validálása:

```
<div class="form-group" :class="{'has-error': errors.has('name') }">
<label for="Name">Name*</label>
<input type="text" v-model="name" name="name" class="form-control"
        id="Name" placeholder="Name"
        v-validate="'required|alpha_spaces|min:3'">
        <span v-show="errors.has('name')" class="text-danger">
        {{ errors.first('name') }}</span>
</div>
```

A v-validate input mezőhöz való hozzáadása után lehet megadni az úgynevezett szabályokat (rules), ez esetben required, alpha_spaces és min:3, tehát a mező kitöltése kötelező, a felhasználónak csak alfabetikus karaktereket és közöket (space) lehet a mezőbe írnia, egyéb esetben a mező nem érvényes. A min:3 a minimum karakterszámot állítja be 3-ra.

A v-show direktíva használatával piros színű szegéllyel határolt mező jelenik meg, ha a mező értéke még nem valid (érvényes). Ha a felhasználó belekattint a mezőbe, majd kikattint belőle úgy, hogy nem írt a mezőbe semmit, akkor a program által előre definiált hibaüzenet jelenik meg: „The name field is required."

Ezenkívül minden egyes szabályra is az előre meghatározott hibaüzenetek jelennek meg, ha a felhasználó numerikus karaktert ír be, vagy ha kevesebb mint a 3 karakter a

begépelt szöveg hossza.

A szám mezők validációja hasonlóan történik:

```
<div class="form-group" :class="{'has-error': errors.has('height') }">
        <label for="Height">Height*</label>
        <input type="number" class="form-control" name="height"
        placeholder="Height" v-model="height" id="Height"
        v-validate="'required|min_value:155|max_value:205'">
        <span v-show="errors.has('height')" class="text-danger">
        {{ errors.first('height') }}</span>
</div>
```

Itt a height mező értékét adhatjuk meg a min_value és max_value szabályokkal, ha a felhasználó nem 155 és 205 közötti számértéket ír be a mezőbe, akkor kiíródik a hibaüzenet.

A record mezőnél a

```
v-validate="'required|regex:^[0-9]+[-][0-9]+[-][0-9]+$'"
```

megadása után a record mező csak akkor lesz érvényes, ha a felhasználó azt ilyen formában adja meg. Például: 100-10-1

Az image_url mezőnél pedig a

```
v-validate="'required|regex:^https?://.+$'"
```

megadása után a felhasználó csak érvényes URL linket adhat meg a harcos avatarjának.

A form fejlécében lévő @submit.prevent=”validateBeforeSubmit” arra szolgál, hogy mikor a felhasználó rákattint a Submit gombra, a form-ot a program még nem küldi el, csak miután meghívta a „validateBeforeSubmit” nevű függvényt.

```
<form @submit.prevent="validateBeforeSubmit">
```

```
validateBeforeSubmit() {
      this.$validator.validateAll().then((result) => {
        if (result) {
          // eslint-disable-next-line
          this.addFighter()
          return; }});
}
```

Ez a függvény megnézi a validateAll() függvény eredményét, és ha minden mező érvényes, akkor meghívja az addFighter() függvényt, ami a FightersService nevű szervízben van definiálva. Ekkor ha a felhasználó a Submit gombra kattint, az összes hibaüzenet azonnal megjelenik, de az addFighter() függvény még nem hívódik meg.

A NewFighter komponens tartalmaz egy <template></template> tagok közötti részt, amiben a megjelenítendő template-t tudjuk megadni.

A <script></script> tagok között adhatjuk meg a szükséges függvényeket, metódusokat, funkciókat.

Ehhez importálnunk kell a FighterService-t:

```
import FightersService from '@/services/FightersService'

export default {
  name: 'NewFighter',
  data () {
    return {
      name: ' ',
      nickname: ' ' }}
```

A data részben a form-ból átvett mezőknek az értékét állítjuk be egy üres sztring-re. Majd a methods résznél megadjuk, hogy a FighterService-ben lévő addFighter függvény pontosan mit hajtson végre:

```
methods: {
    async addFighter () {
      await FightersService.addFighter({
        name: this.name,
        nickname: this.nickname
      })
      this.$router.push({ name: 'Fighters' })
    }
```

A form mezőiben megadott adatokkal tölti fel a megfelelő mezőket, ezután a függvény megkapja a megfelelő paramétereket és az Api szerviz az axios modulon keresztül egy HTTP POST kéréssel hozzáadja a fighters objektumot az adatbázishoz.

```
addFighter (params) {
    return Api().post('fighters', params)}
```

A megjelenítendő harcosok szűrése Szükséges hozzá egy beviteli mező:

```
<div class="search">
        <input type="text" style="text-align:center;" v-model="search"
        placeholder="Search fighters..."/>
</div>
```

Továbbá a <script></script> tagok között a data () részben definiálnunk kell egy a „search” nevezetű mezőt.

```
export default {
  name: 'fighters'
   data () {
    return {
      fighters: [],
      search:' '  }},
computed: {
        filteredFighters: function(){
                return this.fighters.filter((fighter) => {
return fighter.name.toLowerCase().match(this.search.toLowerCase());
                });}}
}
```

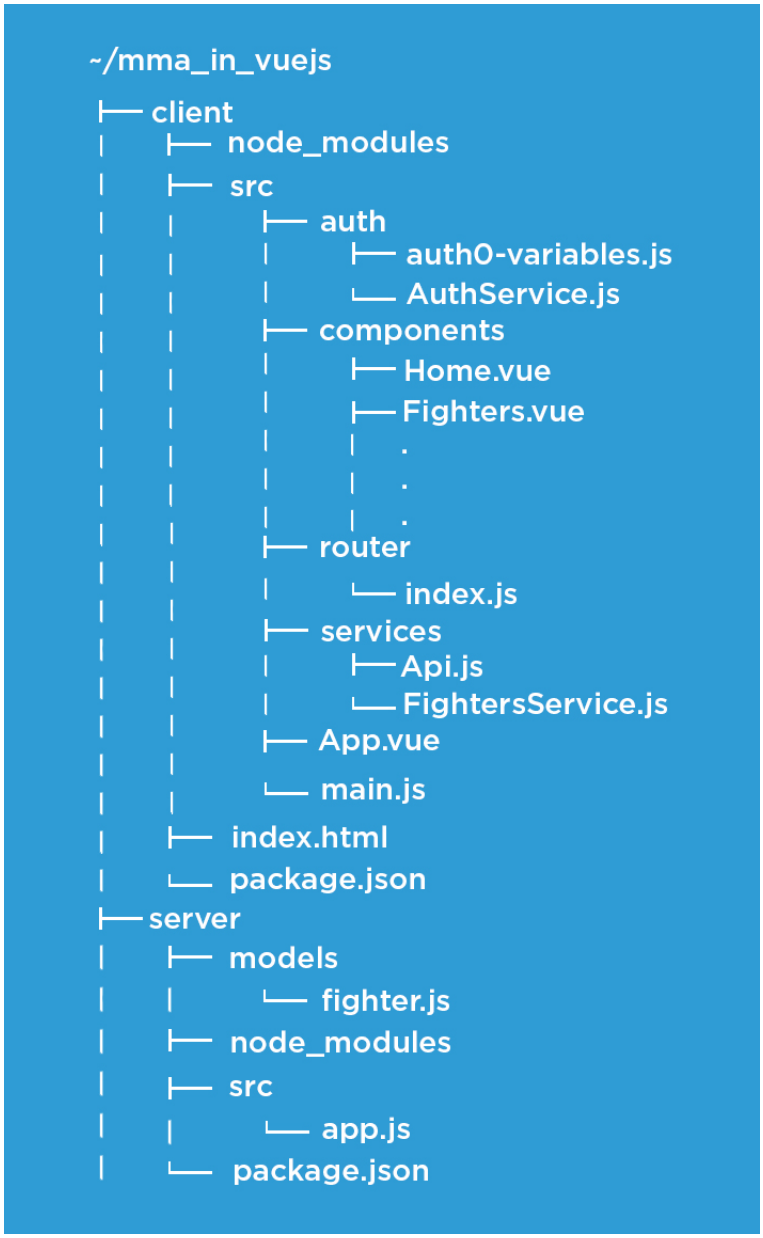
Ezután a filteredFighters() függvény-ben a fighters objektumban lévő fighter-eket szűrjük a .filter kulcsszóval, és ha a harcos „name" mezőjének kisbetűs formára alakított változata megegyezik a „search" nevű beviteli mezőbe beírt érték kisbetűs formára alakított változatával, akkor csak azok a harcosok maradnak a táblázatban, amelyek neve tartalmazza a search mezőbe beírt karaktert, vagy karaktersorozatot.

Ha toLowerCase() függvény alkalmazása mind a name, mind a search mezőre biztosítja, hogy ha a felhasználó kisbetűvel is kezdi a beírt értéket, a program a nagybetűvel kezdődő neveket is benne hagyja a táblázatban.

Ezután a <table> <tbody></tbody> tag-jai között a filteredFighters nevű listán megy végig a ciklus, és jeleníti meg a harcosok adatait, a „search" mező segítségével pedig a beírt érték alapján szűrhetjük a megjelenítendő harcosokat.

```
<tbody>
        <tr v-for="fighter in filteredFighters">
        <td>{{ fighter.name }}</td>
          <td>{{ fighter.nationality }}</td>
        </tr>
</tbody>
```

6.3. Projekt struktúra



6.1. ábra. A Vue.js projekt struktúrája

7. fejezet

ReactJS implementáció

CRUD műveletek megvalósítása

```
constructor(props) {
    super(props);
    this.state = {
      fighters: []
    };
  }

  componentDidMount() {
    axios.get('/api/fighter')
      .then(res => {
        this.setState({ fighters: res.data });
      });
  }
```

A Fighters nevezetű komponensben a /components/Fighters.js fájlban a konstruktorban kell definiálnunk a „fighters” objektumot, a componentDidMount() nvevű függvény pedig az axios modulon keresztül egy http GET kérést küld a szervernek és a /api/fighter oldalról lekéri az aktuális harcosok listáját, majd beletölti a konstruktorban definiált fighters objektumba a response-ban (válasz) kapott adatokat.

Hasonló módon történik csak egy harcos lekérése is, ami a Show nevezetű komponens feladata:

```
componentDidMount() {
    axios.get('/api/fighter/'+this.props.match.params.id)
      .then(res => {
        this.setState({ fighter: res.data });
      });
  }
```

Ebben az esetben szükséges van még a harcos id-jára is, amit a program az URL cím paraméteréből olvas ki. A konstruktorban pedig egy „fighter” objektumot kell megadni.

```
this.state = { fighter: {} };
```

Új harcos létrehozását a Create nevezetű komponens végzi el:

```
constructor() {
    super();
    this.state = {
      name: ' ',
      nickname: ' '
    };}
```

A konstruktorban definiálnunk kell a mezőket üres sztring-ekként. Majd a tényleges létrehozás a harcos adatait kérő form onSubmit=this.onSubmit hatására hívódik meg a POST kérés a szerver felé.

```
onSubmit = (e) => {
    e.preventDefault();
    const { name, nickname} = this.state;
    axios.post('/api/fighter', { name, nickname })
      .then((result) => {
        this.props.history.push("/fighters")
      }); }
```

Az axios modul segítségével egy HTTP POST kérést küldünk el a szerver felé a megfelelő paraméterek megadásával, majd ha sikeres a létrehozás, a program visszairányítja a felhasználót a /fighters oldalra.

Egy harcos adatainak a frissítéséhez szintén a form onSubmit=this.onSubmit eseményére van szükségünk.

```
onSubmit = (e) => {
    e.preventDefault();
    const { name, nickname } = this.state.fighter;
    axios.put('/api/fighter/'+this.props.match.params.id, { name,
    nickname })
      .then((result) => {
        this.props.history.push("/fighters")
      });}
```

Az axios modul segítségével a program egy HTTP PUT kérést küld a szervernek a megfelelő paraméterekkel, majd a frissítés után visszairányítja a felhasználót a /fighters oldalra.

```
delete(id){
    axios.delete('/api/fighter/'+id)
      .then((result) => {
        this.props.history.push("/fighters")
      });}
```

Egy harcos törléséhez pedig a delete() függvény meghívására van szükség, amit egy <button> gomb onClick eseményének meghívásával érhetünk el:

```
<button class="btn btn-danger" onClick={this.delete.bind(this,
this.state.fighter._id)}>Delete</button>
```

A delete() nevű függvénynek átadjuk a harcos id-ját majd az axios modulon keresztül egy HTTP DELETE kérést küldünk a szervernek, ha a program végrehajtotta a törlést, akkor visszairányít a /fighters oldalra.

7.1. Routing

Az oldalak közötti routing az index.js nevű fájlban van megvalósítva:

```
<Router history={history}>
        <div>
                <Route path='/edit/:id' component={Edit} />
        <Route path='/create' component={Create} />
        <Route path='/show/:id' component={Show} />
        </div>
</Router>
```

A path-nál kell megadnunk az URL címet, mellette a component-el adjuk meg, hogy arra a címre navigálva melyik komponensben definiált adatokat kell megjelenítenünk.

7.2. Form validáció

A form validáláshoz a Create komponensben a konstruktorban definiált mezők mellett a következőket kell megadnunk:

```
this.state = {
      name: ' ',
      nickname: ' ',
      formErrors: {name: ' ', weightclasses: ' '},
      nameValid: false,
      wecValid: false,
```

```
      formValid: false
};
```

A formErrors-ban a validálni kívánt mezőket kell definiálni, alapvetően a nameValid és wecValid változó értékét false-ra (hamis) állítjuk, így a formValid változó értéke is hamis lesz.

```
validateField(fieldName, value) {
        let fieldValidationErrors = this.state.formErrors;
    let nameValid = this.state.nameValid;
    let wecValid = this.state.wecValid;
    switch(fieldName) {
    case 'name':
          nameValid = value.length >= 3;
      fieldValidationErrors.name = nameValid ? '': 'value is too short';
      break;
    case 'weightclasses':
      wecValid = value.toString().trim().length;
      fieldValidationErrors.weightclass = wecValid ? '' : ' is required';
      break;
      default:
      break; }
    this.setState({formErrors: fieldValidationErrors,
                    nameValid: nameValid,
                    wecValid: wecValid
                  }, this.validateForm);
}
```

A validateField() nevű függvényben definiálnunk kell pár változót, amik a kontruktorban megadott változók értékeit veszik fel.

Ezután egy switch-case szerkezetben megadhatjuk, hogy melyik esetben milyen hibaüzenetet adjon a program. A „name” mező esetében akkor lesz valid (érvényes) a mező, ha beírt érték hossza nagyobb, vagy egyenlő, mint 3. Tehát 3, vagy több karakteres name mező esetében a nameValid változó értéke true lesz, ellenkező esetben meghatározhatjuk a kiírandó hibaüzenetet.

A „weightclass” mező esetében ha a felhasználó beleír valamit a mezőbe, majd kitörli azt, akkor megjelenítődik a hibaüzenet. A rekord mező esetében:

```
case 'record':
        recValid = value.match(/^([0-9]+)-([0-9]+)-([0-9]+)$/i);
    fieldValidationErrors.record = recValid ? '': ' must be in
    Wins-Draws-Losses form';
```

```
    break;
```

A felhasználó a rekordot csak győzelem-döntetlen-vereség formájában adhatja meg.

A this.setState rész fieldValidationErrors objektumot beletölti a formErrors objektumba, a mezők változóiba pedig a mezők érvényességének aktuális állapotát, majd meghívja a validateForm() függvényt.

```
validateForm() {
 this.setState({formValid: this.state.nameValid && this.state.wecValid});
}
```

A függvény leellenőrzi, hogy érvényesek-e a mezők értékei és ha igen, akkor a form is érvényes lesz, tehát a formValid változó true értéket fog felvenni.

A form „Submit” neveztű gombja pedig mindaddig letiltásra kerül, ameddig a form nem érvényes.

```
<button type="submit" disabled={!this.state.formValid}
class="btn btn-default">Submit</button>
```

Ha a felhasználó beleír valamit a form-ba, a hibaüzenetek még nem fognak megjelenni mert még nem frissítjük az állapotát az input mezőnek, emiatt hozzákell adnunk az alábbi kódot a mezők input-jához.

```
onChange={this.handleUserInput}
```

Ez a mező értékének változásakor meghívja a handleUserInput nevű függvényt, ami eltárolja egy konstansban a mező nevét, és egy másik konstansban a mező értékét, majd beállítja mezőnek az értéket és meghívja a validateField(fieldName, value) függvényt, aminek átadja a mező nevét és értékét, így a függvény az adott case-hez (eset) ugrik és eldönti, hogy a value (érték) alapján valid-e az adott mező, vagy nem.

Ha nem valid, akkor az aktuális mező hibaüzenetét elmenti a „formErrors” nevű objektumba.

```
handleUserInput = (e) => {
    const name = e.target.name;
    const value = e.target.value;
    this.setState({[name]: value},
                  () => { this.validateField(name, value) }); }
```

A hibaüzeneteket a form tetején egy panel-ban jelenítjük meg:

```
<div className="panel panel-default">
        <FormErrors formErrors={this.state.formErrors} />
</div>
```

A nem érvényes mezők piros szegéllyel való megjelenítéséért pedig az alábbi kód felel:

```
<div className=
        {`form-group ${this.errorClass(this.state.formErrors.name)}`}>
<label htmlFor="name">Name*</label>
```

Ez meghívja az errorClass() nevű függvényt a mező aktuális hibaüzenetével, így az érvénytelen mezők a „has-error” class-t kapják meg, tehát piros szegélyük lesz.

```
errorClass(error) {
    return(error.length === 0 ? '' : 'has-error');}
```

Szükség van még a FormErrors nevű osztályra, amit importálnunk kell:

```
import { FormErrors } from './FormErrors';
```

A FormErrors osztály pedig így néz ki:

```
export const FormErrors = ({formErrors}) =>
  <div className='formErrors'>
    {Object.keys(formErrors).map((fieldName, i) => {
      if(formErrors[fieldName].length > 0){
        return (
          <p key={i}>{fieldName} {formErrors[fieldName]}</p>
        )
      } else {
        return ' ';
      } })} </div>
```

Ez a kód végigmegy a formErrors objektumon és megjeleníti az azon belül található hibákat.

7.3. A harcosok táblázatának szűrése

A Fighters.js nevű komponens konstruktorában a CRUD műveleteknél látott „fighters” objektumon kívül egy „filterText” nevű változót kell definiálni egy üres sztring-ként.

```
constructor(props) {
    super(props);
    this.state = {
      filterText: '',
      fighters: []
    };
  }
```

Ezenkívül az updateSearch() nevű függvény frissíti a filterText mező aktuális állapotát.

```
updateSearch(event){
    this.setState({filterText: event.target.value.substr(0,20)}); }
```

A megjelenítendő adatok a render() függvényen belül találhatók, itt meg kell adnunk egy új objektumot, ez esetben a „filteredFighters" nevű objektumot, ami a fighters objektumra hívja meg a filter direktívát, ami megnézi minden fighter-re, hogy a name mezője értékének kisbetűssé alakított változata tartalmazza-e a filterText mező értékének kisbetűssé alakított változatát és ha igen, akkor benne hagyja a filteredFighters objektumban, ha nem, akkor kiveszi belőle. Ha az alábbi kifejezés egyenlő -1-el, akkor amelyik nem tartalmazza, az kikerül az objektumból, így a táblázatból is.

```
render() {
 let filteredFighters = this.state.fighters.filter(
  (fighter) => {
   return fighter.name.toLowerCase()
   .indexOf(this.state.filterText.toLowerCase()) !== -1;
 }
);
```

A táblázaton belül létre kell hozni egy beviteli mezőt, aminek értékét a konstruktorban definiált filterText-re állítjuk be. A filterText mező értékének változásakor meghívja az updateSearch() függvényt a mező aktuális állapotával, így az mindig beállítja neki az újabb állapotot egészen a 20. karakterig.

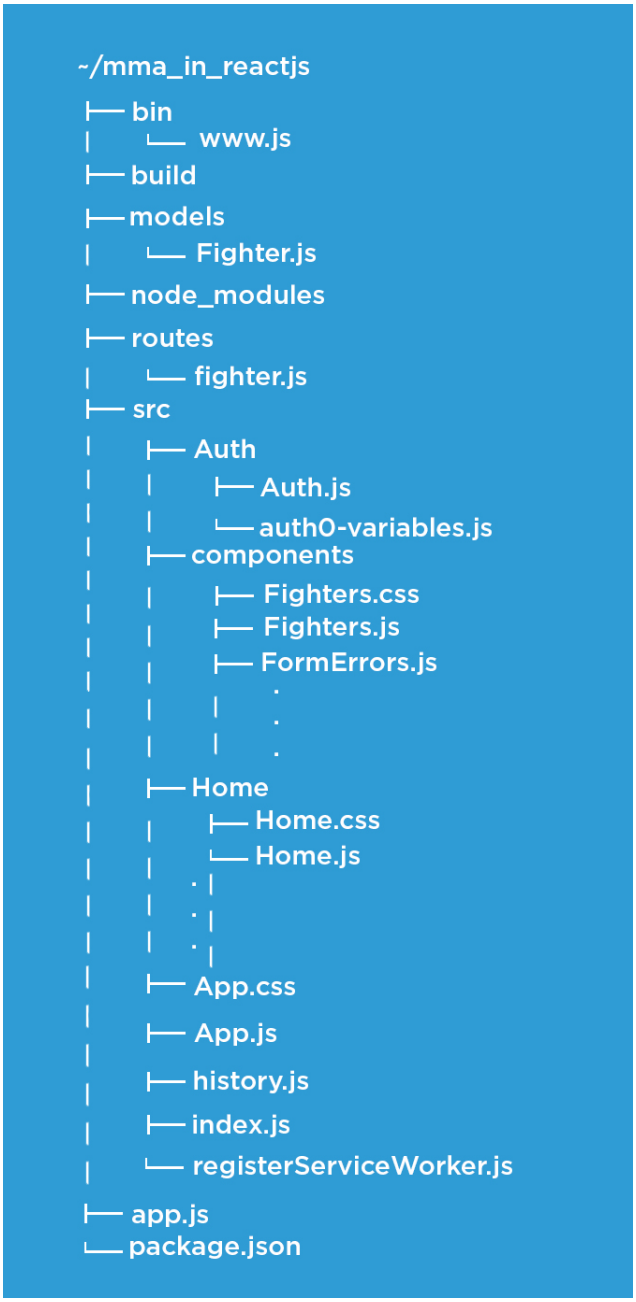
```
return (
  <div className="ftable">
    <div className="search">
      <input type="text" class="searchbar" value={this.state.filterText}
onChange={this.updateSearch.bind(this)} placeholder="Search fighters...">
      </input>
  </div>
```

Az alábbi kód a táblázat <tbody></tbody> tag-jei között itrál végig a filteredFighters objektumon, és megjeleníti a harcosok adatait.

```
<tbody>
{filteredFighters.map(fighter =>
<tr>
        <td>{fighter.name}</td>
        <td>{fighter.nationality}</td>
</tr>
```

```
)}
</tbody>
```

7.4. Projekt struktúra



7.1. ábra. A ReactJS projekt struktúrája

7.5. Auth0 szervíz

```
handleAuthentication() {
    this.auth0.parseHash((err, authResult) => {
      if (authResult && authResult.accessToken && authResult.idToken) {
        this.setSession(authResult);
        history.replace('/home');
      } else if (err) {
        history.replace('/home');
        console.log(err);
        alert(`Error: ${err.error}. Check the console for further details.`);
      }
    });
  }

  setSession(authResult) {
    // Set the time that the access token will expire at
    let expiresAt = JSON.stringify(
      authResult.expiresIn * 1000 + new Date().getTime()
    );
    localStorage.setItem('access_token', authResult.accessToken);
    localStorage.setItem('id_token', authResult.idToken);
    localStorage.setItem('expires_at', expiresAt);
    // navigate to the home route
    history.replace('/home');
  }

  getAccessToken() {
    const accessToken = localStorage.getItem('access_token');
    if (!accessToken) {
      throw new Error('No access token found');
    }
    return accessToken;
  }

  getProfile(cb) {
    let accessToken = this.getAccessToken();
    this.auth0.client.userInfo(accessToken, (err, profile) => {
      if (profile) {
        this.userProfile = profile;
```

```
    }
    cb(err, profile);
  });
}

logout() {
  // Clear access token and ID token from local storage
  localStorage.removeItem('access_token');
  localStorage.removeItem('id_token');
  localStorage.removeItem('expires_at');
  this.userProfile = null;
  // navigate to the home route
  history.replace('/home');
}

isAuthenticated() {
  // Check whether the current time is past the
  // access token's expiry time
  let expiresAt = JSON.parse(localStorage.getItem('expires_at'));
  return new Date().getTime() < expiresAt;
}
```

8. fejezet

Keretrendszerek összehasonlítása

valami	**AngularJS**
Fejlesztő	Google
Github	commit: 8629 contributor: 1603
Google találat	118.000.000 találat
Méret	teljes verzió: 1239 KB (fejlesztéshez - development), minified verzió: 165 K
Elérhető irodalom	https://angularjs.org
Megjelenés, Verzió	2010.október.20, Aktuális verzió: 1.6.6 / 2017.augusztus.18
Típus	JavaScript keretrendszer

Form validáció: AngularJS: ng-class és css class-ok együttes használata, direktívák használata a hibaüzenetek megjelenítéséhez (ng-show)

9. fejezet

Összegzés

A dolgozatomban a jelenleg legnépszerűbb és legelterjedtebb JavaScript keretrendszereket hasonlítom össze. A dolgozat első részében az MVC keretrendszerről, a JavaScript és ECMAScript közötti különbségről, a verziókról és történelmi áttekintésről volt szó. Ezután az a JavaScript keretrendszerek néhány tulajdonságát mutattam le, mint előnyök és hátrányok. A dolgozat fő témája a keretrendszerek (és könyvtárak) több szempontból való összehasonlítása. Ezek a szempontok a CRUD műveletek (Create, Read, Update, Delete) megvalósítása, template-k és a weboldalak közötti routing működése, szűrők és direktívák használata, form validáció, valamint bejelentkezés és regisztráció. A négy webalkalmazás AngularJS, Angular 2, Vue.js, és React.js nyelveken készült el. A back-end részt a MongoDB nevű adatbázis programmal valósítottam meg. A weboldalak az MMA harcosokkal foglalkoznak (MMA Fighters). Véleményem szerint az AngularJS keretrendszer a legkönnyebben megérthető egy laikus számára is, a tesztelése a különböző részeknek egyszerűen megoldható és a kétirányú adatkötés használata megkönnyíti a fejlesztést, ezenfelül az Angular 2-nél és a React.js-nél a komponensek generálásának lehetősége jelentősen lecsökkentette a fejlesztési időt és sokkal komplexebb alkalmazásokat lehet velük készíteni. A Vue.js pedig egy gyors alternatíva, ami ezeknek a keretrendszereknek a legjobb tulajdonságait ötvözi. A JSX nyelvi elemek és a virtuális DOM miatt a React.js több tanulást igényel. A form validálás az Angular 2-nél beépített direktívákkal való használata több, mint egyszerű, a külön mezőkhöz és direktívákhoz tartozó hibaüzeneteket is könnyen meg tudja határozni a felhasználó. A Vue.js esetében pedig a vee-validate nevű modullal és a beépített szabályaival könnyíti meg a validációt. Az AngularJS-nél a kód hosszúsága nem a legmegfelelőbb, React.js-nél pedig a kétirányú adatkötés miatt a megoldásomban frissíteni kellett a mező aktuális értékét annak változtatásakor. Az auth0 szerviz mindegyik keretrendszerhez megtalálható, dokumentációjuk átlátható, követhető.

További tervek, ötletek

A alkalmazások továbbfejlesztése, kibővítése újabb funkciókkal. Tervek között szerepel a képfeltöltés funkciójának megvalósítása, saját Auth0 szervíz bejelentkezési form létrehozása, admin és vendég felület hozzáadása.

Irodalomjegyzék

[1] Bujdosó Gyöngyi, Fazekas Attila: *TEX kezdõlépések*, Tertia Kiadó, Budapest, 1997.

[2] Házy Attila: *Lineáris függvényegyenletek megoldása számítógéppel*, Doktoranduszok fóruma 2005, Miskolc, 2005. november 9., Gépészmérnöki Kar szekciókiadványa, Miskolc, ME ITTC, 2006., 108–113.

[3] Hettl, Mayer, Szabó: *LATEX kézikönyv*, Panem Könyvkiadó, Budapest, 2004.

[4] M. E. Hohmeyer, B. A. Barsky: Rational continuity: parametric, geometric and Frenet frame continuity of rational curves, *ACM Transactions on Graphics*, **8** (1989), 335–359.

[5] TEX Catalogue, `www.ctan.org/tex-archive/help/Catalogue/catalogue.html`