Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Informatikai Intézeti Tanszék

Szakdolgozat

Miskolci Egyetem

JavaScript alapú frontend technológiák összehasonlítása

Készítette:

Zajáros Tamás

Mérnökinformatikus, BSc

Témavezető:

Piller Imre, egyetemi tanársegéd

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 Zajáros Tamás; Neptun-kód: I0VATO, a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzõs mérnök informatikus szakos hallgatója ezennel büntetõjogi és fegyelmi felelõsségem tudatában nyilatkozom és aláírásommal igazolom, hogy "JavaScript alapú frontend technológiák összehasonlítása" 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, 2017. év 11. hó 24. nap

.....

Hallgató

Tartalomjegyzék

1. fejezet

Bevezetés

Napjainkban nagyon sokféle megoldás létezik egy weboldal elkészítésére.

2. fejezet

JavaScript technológiák és keretrendszerek

2.1. MVC keretrendszer

Az MVC keretrendszer (Model-View-Controller) egy olyan felépítési minta, amelynek segítségével az alkalmazásokban lévő adatokat több részre bontjuk, egészen pontosan a modellre, a nézetre és a vezérlőre. A fő cél az alkalmazás rétegeinek egymástól való elkülönítése. Lényege, hogy az egyes részekben lévő adatok változtatását egyszerűbbé teszi a különválasztás által, mert így nem zavarják egymást a komponensek.
A modellben tároljuk azokat az adatokat, amin a vezérlő kódja műveleteket végez és amelyet a nézet által megjelenít. Ilyenek lehetnek például a form-ok (űrlap) gombok, linkek, navigációs sávok, táblázatok, listák. A felhasználó a "Delete" gombra kattintás esetében azt látja, hogy az oldalon megjelenített adatokból eltűnt az, amit éppen kitörölt. Ezt a vezérlőben lehet meghatározni, lekezelni az egyes eseteket, azt hogy a felhasználó interakciója a felülettel mikor milyen eseménnyel, változtatással járjon. A modell-t általában fájlokban, vagy adatbázisban tárolják. Ez a rész tartalmazza az adatok logikai felépítését, de a felhasználó felületről semmilyen fajta információt nem tárol. A vezérlő áll kapcsolatban mind a modell-el, mind a nézettel.

2.2. 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 a Netscape nevű cég fejlesztése, az első böngésző, ami futtatni tudta a JavaScript-et a Netscape 2 volt 1996-ban. A Netscape után a Mozilla cége folytatta a fejlesztését a Firefox nevű böngészőjének. A JavaScript verziószámok 1.0-tól vannak számozva, az utolsó verzió száma: 1.8.5

Az ECMAScript egy nyelvi standard, az ECMA International cég fejlesztése, a JavaScript implementációból fejlődött ki. Az első kiadása 1997-ben jelent meg, a verziószámai 1-től 7-ig vannak sorszámozva. A JScript-et a Microsoft nevű cég fejlesztette ki 1996-ban, mint egy kompatibilis JavaScript nyelvet az Internet Explorer böngészőjükhöz. A JScript verziószámai 1.0-tól 9.0-ig terjednek.

2.3. JavaScript technológiák áttekintése

2.3.1. AngularJS

Előnyei:

- Template-ek használata,
- a kétirányú adatkötés segítségével szinkronizálja a DOM-ot (Document Object Model) és a modell-t
- belső HTML kódok hozzáadása helyett, azonnal a DOM-ot módosítja az oldalon, így gyorsabb,
- MVC / MVVN (Model-View-ViewModel) használata, de közelebb áll az MVVN-hez,

- DI (Dependency Injection) - alkalmazásbeli függőségek kezelése,
- Google cég által fejlesztett és támogatott, hatalmas fejlesztői táborral rendelkezik,
- saját direktívák létrehozásának lehetősége,
- a tesztelésre fordított figyelem, a tesztelés egyszerűsége.

Hátrányai:

- Az Angular 2-t teljesen újraírták így visszafelé nem kompatibilis, azok a fejlesztők, akik Angular 2-t akarnak használni, az alapoktól kell újraírniuk az alkalmazásukat,
- nem támogatja az oldal betöltésének felgyorsítását (szerver-oldali renderelés),
- a JavaScript támogatás kikapcsolása esetén nem lehet elérni a weboldalt,
- a scope-k, és a direktívák használata a kezdők számára bonyolult lehet.

2.3.2. Angular 2

Előnyei:

- Angular 2 CLI-vel (Command Line Interface) az alapból működő alkalmazások létrehozása egyszerűvé válik,
- komponens alapú használat,
- a TypeScript a JavaScript továbbfejlesztése,
- több platformon elérhető, Ionic keretrendszer használatával hibrid mobilalkalmazások létrehozását teszi lehetővé,
- elterjedt kombináció (MEA2N)- MongoDB-Express.js-Angular2-Node.js
- jobb teljesítmény és gyorsaság az AngularJS-hez képest,
- animimációs csomag használatának lehetősége, így a fejlesztőnek nem kell animációs könyvtárakat megtanulnia.

Hátrányai:

- A nyelvnek a megtanulása (TypeScript) több időt igényel azoknak, akik előtte még nem ismerték,
- a DOM közvetlenül történő változtatása miatt lassabb,
- kevés TypeScript és Angular 2 fejlesztő van, aki igazán jó fejlesztő lenne.

2.3.3. Vue.js

Előnyei:

- A keretrendszer használatának megértése és a benne történő fejlesztés egyszerű,
- mérete nagyon kicsi, 30 KB összesen a .gzip úgynevezett "minified", vagyis tömörített változat
- opcionális JSX támogatás,
- kétirányú adatkötés "v-model" használatával,
- gyors fejlesztés,
- Vue CLI (Command Line Interface) használata.

Hátrányai:

- A keretrendszer használatának megértése és a benne történő fejlesztés egyszerű,
- mérete nagyon kicsi, 30 KB összesen a .gzip úgynevezett "minified", vagyis tömörített változat
- opcionális JSX támogatás,
- kétirányú adatkötés "v-model" használatával,
- gyors fejlesztés,
- Vue CLI (Command Line Interface) használata.

2.3.4. React Js

Előnyei:

- Komplexebb web-es alkalmazások készítésére is alkalmas,
- Create-React-App CLI az alkalmazások gyors fejlesztéséhez,
- elterjedt (MERN) - (MongoDB-Express.js-React-Node.js),
- csak akkor frissíti a virtuális DOM-ot, ha szükséges, emiatt nagyon gyors,
- szerver és kliens oldali renderelés,
- komponens alapú, a komponenseket beágyazhatjuk, újra felhasználhatjuk,
- React Native technológiát használ, amellyel teljes értékű mobilalkalmazásokat lehet készíteni.

Hátrányai:

- Hiányzik belőle a kétirányú adatkötés, így az adatváltoztatások esetében saját kód írásával kell orvosolni a problémát,
- csak a "Nézet" (M-View-C) részt képviseli, így többféle könyvtár hozzáadását igényli a különböző feladatok megoldásához,
- tanulása a JSX és ECMAScript 6 miatt több energiát vesz igénybe,
- teljesen más gondolkodást igényel az MVC mintáktól,
- rendesen kidolgozott hatékony kombinációkat találni a React-al történő fejlesztéshez nem egyszerű.

2.3.5. TypeScript

- Egy ingyenes és nyílt forráskódú programozási nyelv,
- a JavaScript továbbfejlesztése,
- arra lett kitalálva, hogy az objektum-orientált programozási nyelvekben jártas fejlesztőknek ne kelljen a JavaScript-hez szükséges gondolkodásmódra átállniuk,
- JavaScript kódra fordít, amely bármely JavaScript motorban és böngészőben képes futni, amelyik támogatja az ES3-at (ECMAScript 3).

2.3.6. CoffeeScript

- Egy nyelv, ami JavaScript kódra fordít,
- javítani próbál a JavaScript-en, kevesebb kóddal ugyanazt az eredményt elérni,
- könnyen olvasható, érhető és gyors.

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

A mintaalkalmazás 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. Amennyiben 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*: egy szöveg mező, a harcos teljes neve, melynek minimum három karakter hosszúságúnak kell lennie, megadása kötelező,
- *becenév*: szintén szöveg mező, a harcos beceneve, nem kötelező megadni,
- *súlycsoport*: az előre megadott súlycsoportok közül a harcos súlycsoportjának kiválasztása, megadása kötelező, a választási lehetőségek száma: 3,
- *születési dátum*: dátum mező, a harcos születési ideje: év, hónap, nap formátumban, megadása kötelező,
- *szülőváros*: szöveg mező, a harcos szülőváros, megadása opcionális,
- *nemzetiség*: szöveg mező, a harcos nemzetisége, minimum két karakter, megadása kötelező,
- *magasság*: szám mező, a harcos magassága centiméterben megadva, 155 és 205 közé kell esnie, megadása kötelező,
- *súly*: szám mező, a harcos súlya kilogrammban értve, 50 és 120 közé kell, hogy essen, megadása kötelező,
- *record*: a harcos rekordja győzelem-döntetlen-vereség formában, megadása kötelező,

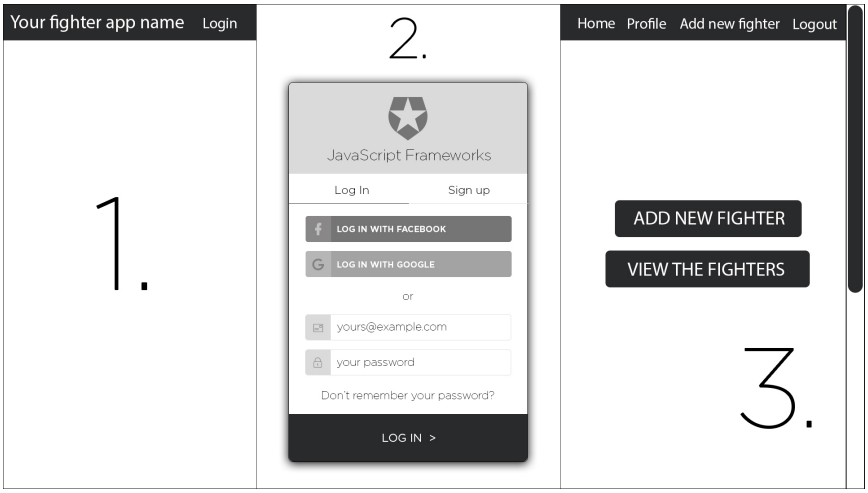
- *harcos avatárjának URL linkje*: szöveg mező, megadása kötelező,
- *a harcossal foglalkozó oldal URL linkje*: szöveg mező, megadása opcionális.

A harcosok főbb adatai, mint név, nemzetiség, születési dátum és rekord, a főoldalon (home) lévő "VIEW THE FIGHTERS" gombra kattintva jelennek meg. Minden harcos sorában egy "View Details" nevű gomb van, ami átirányít a "/fighter/details/:id" oldalra. Itt megtekinthetők az adott harcos további adatai, melyek fentebb kerültek felsorolásra. Továbbá ezen az oldalon tudjuk frissíteni az adatokat az "Edit" gombra kattintva.

Törölni a "Delete" nevű gombbal lehet. A "Go Back" 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 a megadásukra vonatkozó megszorítások. A backend szolgáltatást a MongoDB adatbázis, a Node.js, mint szerver és a legnépszerűbb Node.js keretrendszer, az Express biztosítja, ami a HTTP kérések irányításáért felel.

Frontend 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, ezt követően a program átirányítja a főoldalra. (3.1. ábra).

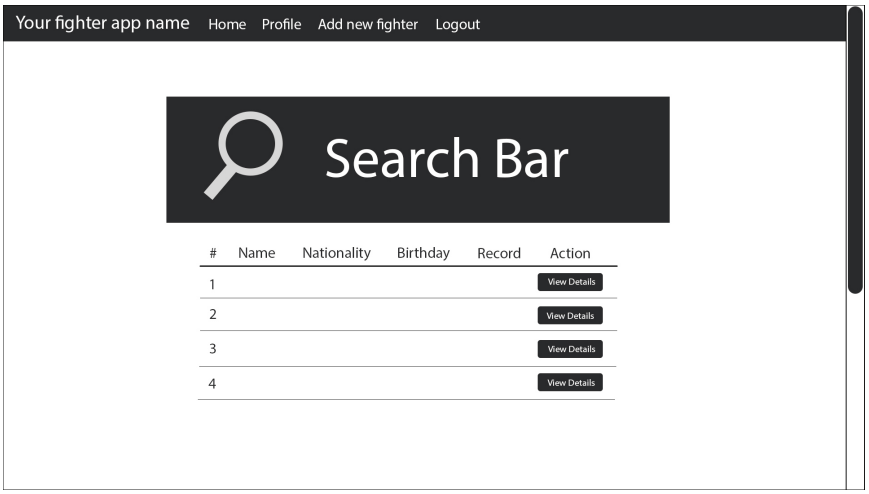


3.1. ábra. Főoldal (Home)

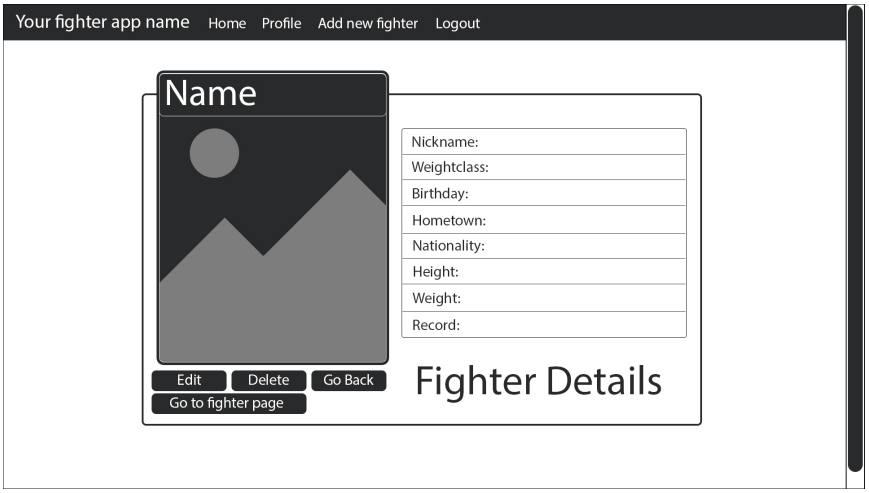
A "VIEW THE FIGHTERS" feliratú linkre kattintva a "/fighters" oldal jelenítődik meg a harcosok listájával (3.2. ábra).

A harcos sorában lévő "View Details" linkre kattintva az adott harcos adatait tartalmazó oldal jelenik meg az "/fighters/details/id" oldalon (3.3. ábra).

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

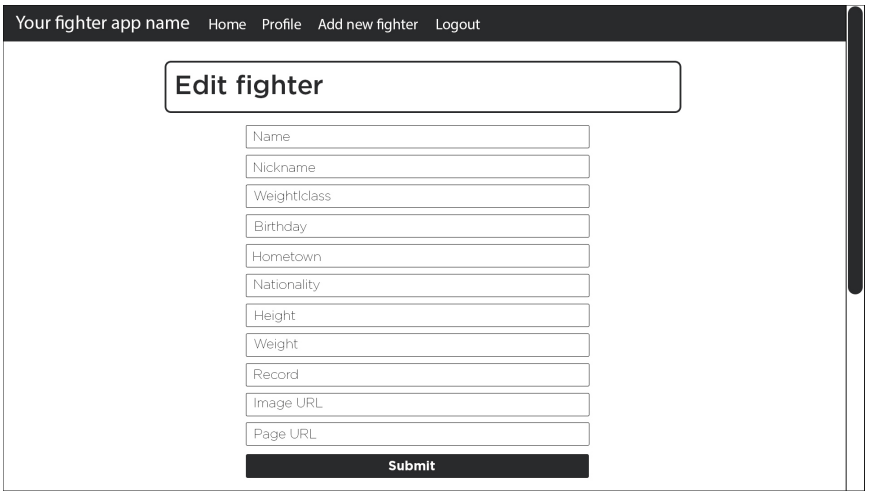


3.2. ábra. Harcos lista oldal (Fighters)

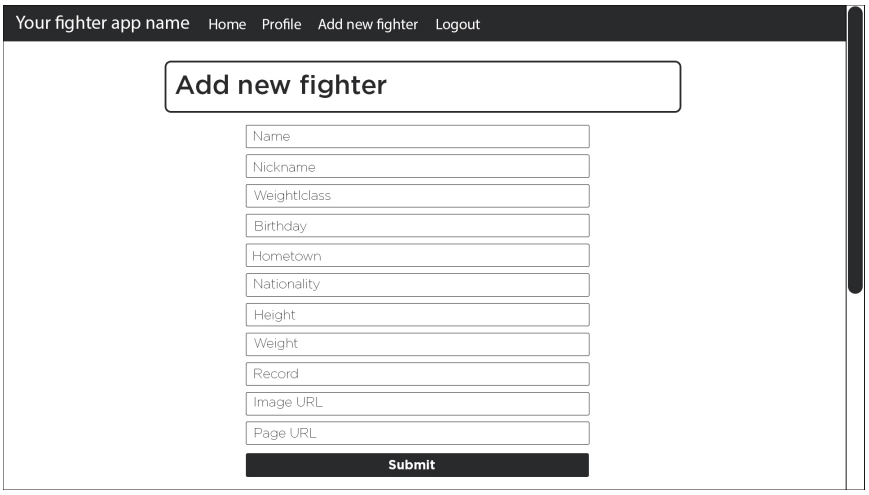


3.3. ábra. Harcos adatait megjelenítő oldal (Fighter Details)

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. A harcos adatait frissítő 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. A webalkalmazásban ezeket a funkciókat az MMA harcosok adatain 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 hozzá tartozó kontrollert kell elkészíteni:

```
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 lekérdezése:

```
$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, amely visszaküldi egy response objektumban a "fighters" objektumot, amely a harcosok adatait tartalmazza.

Az "/api/fighters" URL-en lévő adatokat adja vissza, amik JSON formátumban kerültek eltárolásra.

A következő példakód egy harcos hozzáadását mutatja be, amelyhez egy HTTP POST 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 érintett 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 található "Delete" gombnál adunk meg, ami a megfelelő harcos "id" mezőjének értékével 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 betűsorozat beírása utáni dinamikus leszűkítése. Amennyiben 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>
```

A 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, amelyre 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,
- ha a mező értéke nem érvényes, annak ellenére, hogy már írt bele.

A hibaüzenet a mezőből való kikattintás után jelenik meg. A "Név megadása 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 hármat.

A "has-error" ng-class (AngularJS osztály) hiba esetén a hibaüzenet piros színnel és a beviteli mező piros szegéllyel történő 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 csak üres mező esetében jelenik meg. Ezt követően, amennyiben 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 a harcos avatarjának URL link validációját készítettem el, így ha a felhasználó nem "http://" vagy "https://" kezdetű URL címet ad meg, akkor "Image URL must be valid" hibaüzenetet kap.

4.2. Routing

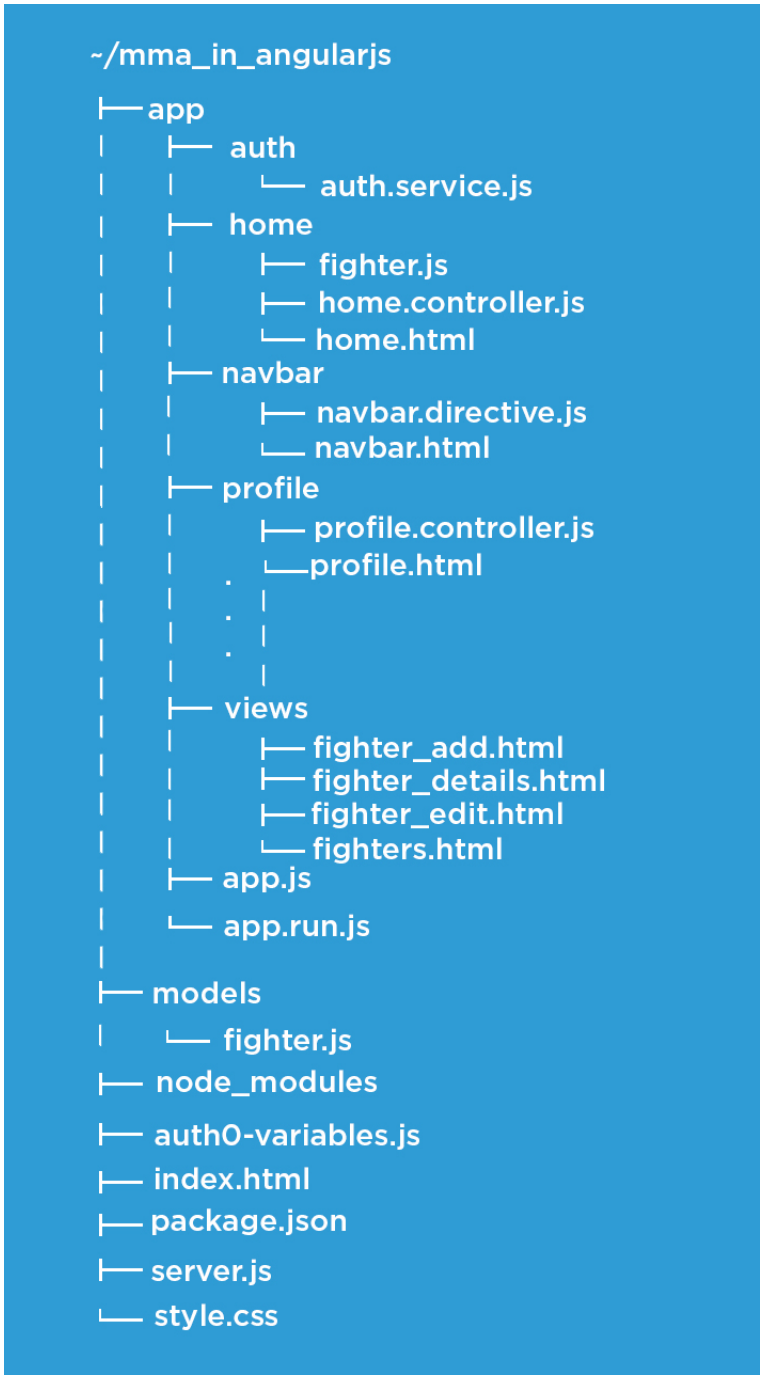
Az oldalak közötti routing az "ui.router" szervízzel van megoldva. Ehhez az "angular module"-ban meg kell adnunk az ui.router-t, mint függőséget (dependency).

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

Az alábbi kód "/fighters" oldal megjelenítését, a FightersController a funkciókat biztosítja, a "/fighters" oldalra való navigációkor pedig az alkalmazás a "fighters.html" nevű 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 szerviz 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 és adataik. Egy harcos adatainak frissítésére szolgál az "updateFighter()" függvény:

```
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 automatikusan 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, amelyet a "styleUrls" kifejezéssel adhatunk meg.

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

A harcosok táblázatának keresőmezővel történő szűréséhez egy úgynevezett "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 szükség van egy "import"-ra, amelyet 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. Továbbá szükséges még az "app.component.html" fájlban a <router-outlet></router-outlet> tag-ok megadása.

Route-k (útvonalak) létrehozása az "app.routes.ts" fájlban történik.

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

Ezenkívül szükséges a következő "import" megadása az "app.module.ts" fájlban:

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

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" gombhoz való hozzáadásával érhető el.

```
<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, a mező kitöltésekor minimum három 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.

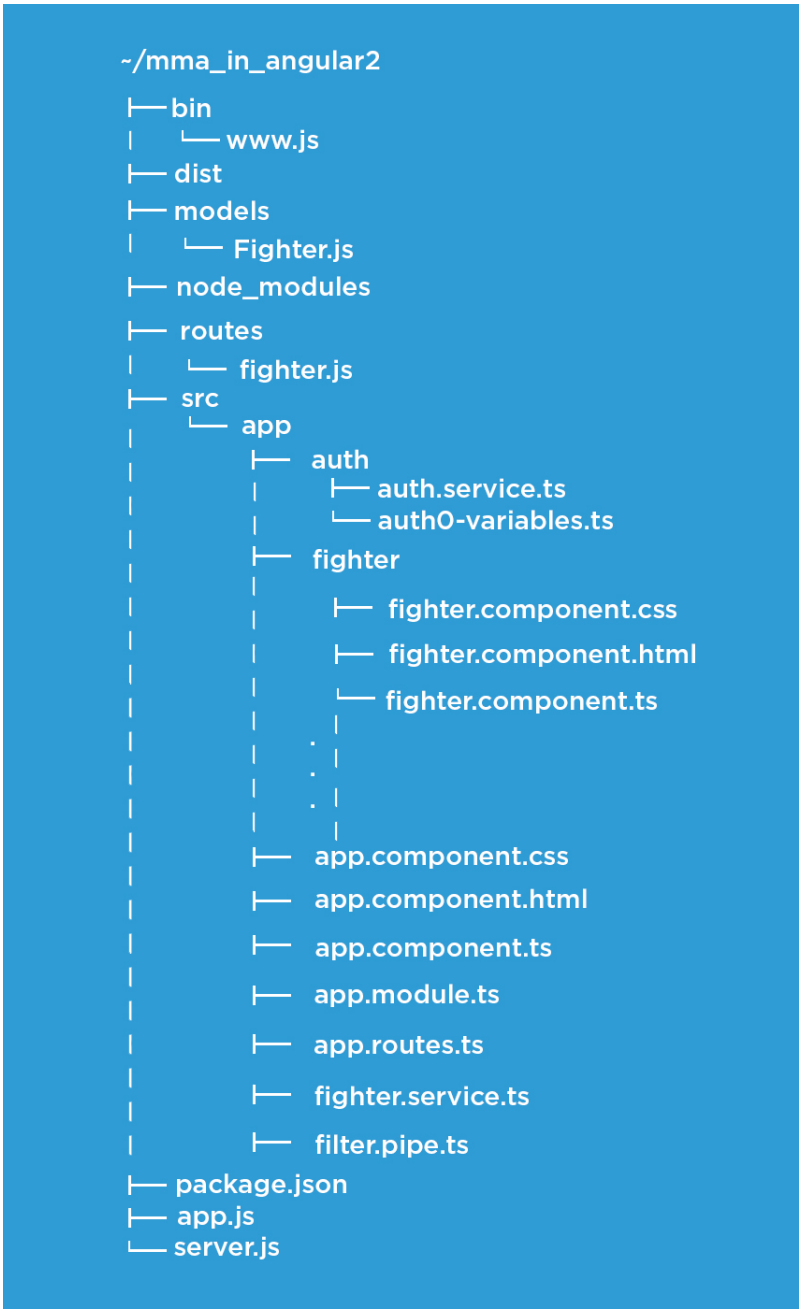
A "record" mező validálásához "pattern"-t kell hozzáadni a mezőhöz:

```
<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 validációjánál itt is a pattern="https?://.+" direktívát kell a beviteli mezőhöz hozzáadni. Ezt követően, ha a felhasználó az "Image URL" linket nem a megfelelő formában adja meg, akkor az "Image URL must be valid!" hibaüzenetet kapja.

5.3. Projekt struktúra



5.1. ábra. Az Angular 2 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" nevű 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-jét.

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

Api.js fájl tartalma:

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

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

6.1. Routing

A routing-hoz szükséges importálni és használni a "vue-router"-t:

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

A routing-ok (útvonalak) létrehozásánál a Vue.use(Router) sorral adjuk meg, hogy a program használja Router-ként a beimportált "vue-router" nevű 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 az history.pushState API használatával megteremti annak a lehetőségét, 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 validálás többféle módon történhet, az egyik ilyen a "vee-validate" modul használatával történő validáció, amelyhez először telepíteni kell a modult az npm csomagkezelő vagy CDN segítségével, majd az alábbiakat kell importálni az alkalmazáshoz tartozó "main.js" fájlban:

```
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 háromra.

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 három 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 megjelenik a hibaüzenet.

A "record" mező a

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

megadása után 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" szer-

vízben 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> tag-ok közötti kódrészletet, amiben a megjelenítendő template-t tudjuk megadni.

A <script></script> tag-ok 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 sztringre. 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 fenti függvény a form mezőiben megadott adatokkal tölti fel a megfelelő mezőket, ezután a függvény megkapja a szükséges 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 szűréséhez 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> tag-ok között a data () részben definiálnunk kell egy a "search" nevű 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());
                });}}
}
```

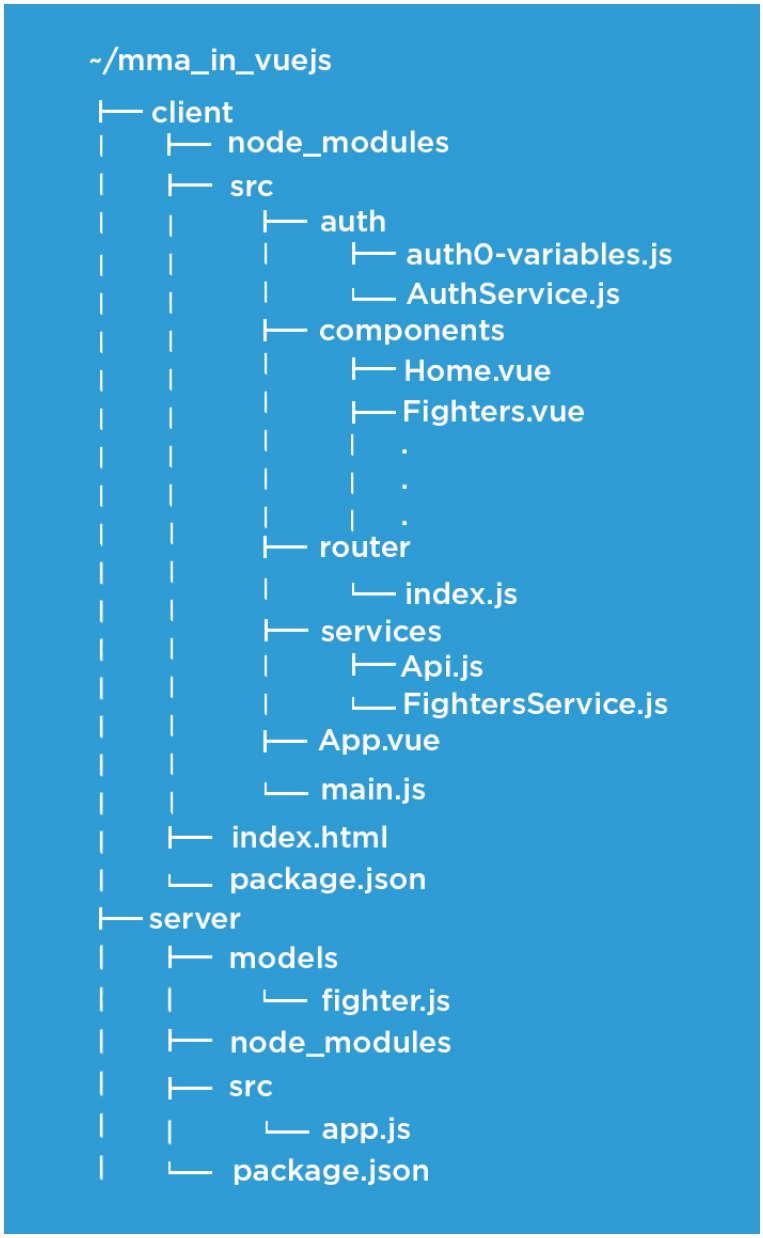
Ezután a "filteredFighters()" függvényben a "fighters" objektumban lévő fightereket szűrjük a .filter kulcsszóval, és ha a harcos "name" mezőjének kisbetűs formára alakított változata megegyezik a "search" beviteli mezőbe beírt érték kisbetűs formára alakított változatával, akkor csak azok a harcosok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

A "toLowerCase()" függvény alkalmazása mind a "name", mind a "search" mezőre biztosítja, hogy a program a nagybetűvel kezdődő neveket is benne hagyja a táblázatban, annak ellenére, hogy a felhasználó kisbetűvel kezdi a beírt karaktersorozatot.

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" nevű komponensben a /components/Fighters.js fájlban, a konstruktorban kell definiálnunk a "fighters" objektumot, a "componentDidMount()" nevű 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 egy harcos lekérése is, ami a "Show" nevű komponens feladata:

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

Ebben az esetben szükség van még a harcos id-jére is, amit a program az URL cím paraméteréből olvas ki. A konstruktorban 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 történik meg.

```
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 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> "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-jét majd az axios modulon keresztül egy HTTP DELETE kérést küldünk a szervernek. Miután a program végrehajtotta a törlést, 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" résznél 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, amelyek 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 a beírt érték hossza nagyobb vagy egyenlő, mint három. Tehát három 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 beír valamit a mezőbe, majd kitörli azt, akkor megjelenítődik a hibaüzenet. A rekord mező validálása:

```
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 a "fieldValidationErrors" objektumot a "formErrors" objektumba, a mezők érvényességének aktuális állapotát pedig a mezők változóiba tölti bele, majd meghívja a "validateForm()" függvényt.

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

A függvény ellenőrzi, 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" nevű gombja pedig mindaddig letiltásra kerül, amíg 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 nem fognak megjelenni, mert még nem frissítjük az állapotát az input mezőnek, emiatt hozzá kell adnunk az alábbi kódot a mezők input-jához.

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

Ez az esemény a mező értékének változtatásakor meghívja a "handleUserInput" nevű függvényt, ami eltárolja egy konstansban a mező nevét és egy másik konstansban a mező értékét. Ezt követően beállítja a 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" 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 az alábbi kód felel:

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

A fenti kódrészlet 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 az alábbiakat tartalmazza:

```
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 fájl konstruktorában a CRUD műveleteknél látott "fighters" objektumon kívül egy "filterText" nevű változót kell definiálni 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. Amely a "fighters" objektumra hívja meg a "filter" direktívát. A filter megnézi minden fighterre, 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 valamelyik harcos esetében az alábbi kifejezés egyenlő -1-el, akkor 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, amelynek értékét a konstruktorban definiált "filterText"-re állítjuk be. A filterText mező értékének változtatásakor meghívja az "updateSearch()" függvényt a mező aktuális értékével, így az mindig beállítja az újabb állapotot egészen a 20. karakterig.

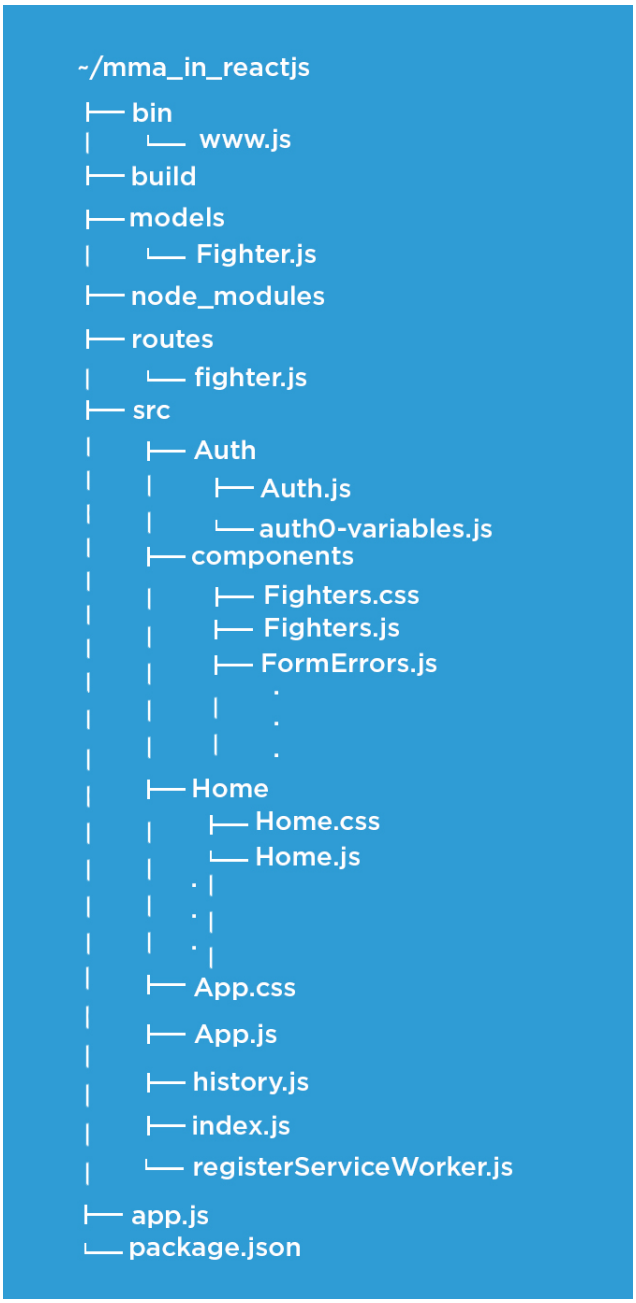
```
return (
  <div className="ftable">
    <div className="search">
      <input type="text" class="searchbar" value={this.state.filterText}
onChange={this.updateSearch.bind(this)} placeholder="Search fighters...">
      </input>
  </div>
```

Az alábbi kód a táblázat <tbody></tbody> tag-jei között iterál 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

8. fejezet

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;
  }
```

9. fejezet

Keretrendszerek összehasonlítása

	AngularJS	**Angular 2**	**React.js**	**Vue.js**
Fejlesztő	Google	Google	Facebook, Instagram	JavaScript könyvtár
Github	commit: 8629, contributor: 1603	commit: 9009, contributor: 540	commit: 9425, contributor: 1139	commit: 2378, contributor: 159
Google találat	118 millió	103 millió	180 millió	23 millió 400 ezer
Méret	1239 KB, min: 165 KB	1044 KB, min: 566 KB	45 KB, min: 6 KB	272 KB, min: 84 KB
Elérhető irodalom	angularjs.org	angular.io	reactjs.org	vuejs.org
Megjelenés Verzió	2010.október 1.6.6	2014.szeptember 2.0.0	2013.március 16.2.0	2014.február 2.5.3
Típus	JavaScript keretrendszer	JavaScript keretrendszer	JavaScript könyvtár	JavaScript keretrendszer

9.0.1. Form validáció

AngularJS:

"ng-class" és css class-ok együttes használata, direktívák használata a hibaüzenetek megjelenítéséhez (ng-show), egyszerűen megvalósítható. A "number" típusú mezők validálása a "min" és "max" direktívákkal minden további gond nélkül megoldható. Továbbá lehetőség van saját érvényesség ellenörző direktívák létrehozására is.

Angular 2:

Az "*ngIf" és "ngModel" direktívák használatával egyszerűen megvalósítható. Lehetőség van továbbá úgynevezett saját érvényesség ellenőrző direktívák létrehozására, ahol meg lehet határozni, hogy mit ne fogadjon el, esetleg mit fogadjon el a program lehetséges input-ként. A "number" típusú mezők validálása csak külön modullal vagy saját érvényesség ellenőrző direktíva létrehozásával érhető el, amely néha kényelmetlenségeket okozhat. Viszont a különböző mezőknél történő saját hibaüzenet megadásának lehetősége, és az azok közötti automatikus váltás kárpótol emiatt.

Vue.js:

A "vee-validate" modul telepítése szükséges hozzá, elérhető 20 validációs szabály, mint például "alpha_spaces", amely azt hivatott ellenőrizni, hogy az adott input mező tartalmaz-e szóközt, ha igen, akkor érvényes a mező. Jól használható név mezők esetében, ahol az a fontos, hogy ne lehessen a mezőben numerikus karakter, de lehessen benne szóköz. A validációs üzenetek beépítettek, de lehetőség van a változtatásukra. Személy szerint ebben a keretrendszerben használható fentebb említett modul miatt ezt a megoldást találtam a legkényelmesebbnek a rendelkezésre álló szabályok sokszerűsége miatt.

React.js:

A form validáció implementálása ebben a keretrendszerben volt a legnehezebb és a legtöbb időt ez vette igénybe, lévén, hogy nincs kétirányú adatkötés, így különböző validációs funkciókat kellett létrehozni a kívánt eredmény eléréséhez, továbbá ennél a megoldásnál a validációs hibaüzenetek csak a kitöltendő form felett lévő panel-ban jelennek meg.

9.0.2. Routing

10. fejezet

Összegzés

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 előnyeit, hátrányait mutattam be. Ezt követően a mintaalkalmazások specifikációja található meg. A dolgozat fő témája a keretrendszerek több szempontból való összehasonlítása. Ezek a szempontok a CRUD műveletek 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 keretrendszerekben készült el. A backend részhez a MongoDB nevű programot használtam az adatok tárolására, a Node.js-t a szerver megvalósításához. A weboldalak az MMA harcosokkal foglalkoznak (MMA Fighters). Véleményem szerint az AngularJS keretrendszer a legkönnyebben megérthető egy webes oldalak világában és technológiáiban nem teljesen jártas ember 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. A React.js használatának megtanulása több időt és erőfeszítést igényel, mindezek mellett jobb megoldás lehet egy nagyobb, komplexebb alkalmazás elkészítéséhez. A Vue.js egy gyors alternatíva, amely ezeknek a keretrendszereknek a legjobb tulajdonságait ötvözi, olyanokat, mint az AngularJS-ben megismert kétirányú adatkötés, JSX támogatás, vagy szerver-oldali renderelés. 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 szerviz bejelentkezési form létrehozása, admin és vendég felület hozzáadása.

Mindegyik keretrendszernek és technológiának vannak előnyei és hátrányai, amikkel a fejlesztők, és a cégek tisztában vannak, ennek ellenére egy jó lehetőség belelátni ezeknek a technológiáknak a működésébe, használatuknak rejtélyeibe és mindenképpen előny többféle keretrendszer elemeinek és működésének megismerése.

11. fejezet

Summary

In my thesis i compared the most popular JavaScript frameworks. In the first part i demonstrated that what is the MVC framework, what is the difference between JavaScript and ECMAScript, there was a part of the versions and lastly a historical overview. Thereafter an overview of JavaScript technologies.
The main subject was to compare the frameworks and libraries in a few aspects. Those are the CRUD operations, templates, routing, filters, directives, form validation, and also login with an option to sign up. The four web applications what i implemented are AngularJS, Angular 2, Vue.js, and React.js. The MongoDB database and the Node.js server provides the backend functionalities. These web applications are deal with MMA Fighters. There is an option to log in or sign up, to add a new fighter, to see the fighters who are stored in the database, to filter the table of fighters, to edit the fighter's details and to delete the fighters.
The AngularJS framework is a very good option to develop web applications, and it is not really hard to learn for those who are not proficient in the web pages world. Maintaining and testing the separated parts is really easy to handle. The two-way data binding is simplify the development phase. In Angular 2, React.js, and Vue.js the option to generate the components and services with their CLI-s is very handy. These frameworks are well-known and very popular in firms. The Vue.js is a framework which is combine the advantages of the other frameworks. It is a fast alternative, and growing quickly. Because of the JSX and ES6 elements the React.js is need more time and efforts to learn how to use the library.
The Auth0 platform is a powerful and popular solution to integrate into any web application. It has a well-founded documentation, and support the most popular frameworks and API-s.
My further plans and ideas to work on in these applications are to add new functions, like image uploading for the fighter's image field, create a custom login and signup form with Auth0, and also to create an administrator and guest interface.
Each one of these frameworks has advantages over the others, and also has disadvantages as well, which the developers or firms could accept. In my opinion it is a good

opportunity to choose from this big lineup of technologies and take time to meet and learn them.

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`

Adathordozó használati útmutató

A szakdolgozatomban elkészítettem négy webalkalmazás programot, amelyek a "JavaScript keretrendszerek" mappában találhatók. Az elkészített alkalmazások az alábbi keretrendszerekben kerültek implementálásra: AngularJS, Angular 2, React.js, Vue.js, így az almappák nevei ezekkel egyeznek meg. Az alkalmazások az MMA harcosokkal foglalkoznak, ezért minden keretrendszer mappájában az "mma_in_keretrendszer neve"
konvenciót használtam.

A keretrendszereket külön-külön részletezem.

Az AngularJS esetében az "mma_in_angularjs" nevű mappában találhatók a fájlok. A szerver indításához szükséges a Node.js program letöltése, telepítése. A harcosok adatbázisban való eltárolásához pedig a MongoDB nevű program letöltésére és telepítésére van szükség annak érdekében, hogy a létrehozott harcosok az oldal lefrissítése után is megmaradjanak.
(Ha a MongoDB telepítésekor nincs megadva, hogy a szerver automatikusan elinduljon, akkor a programok elindítása előtt szükséges a "mongod" nevű parancs kiadása, amelyet a MongoDB telepítési helyén lévő "bin" mappába való navigálás után lehet megtenni.) A "Node.js command prompt" nevű parancssoros alkalmazás megnyitása után be kell lépni a "JavaScript keretrendszerek" nevű mappába, majd az "mma_in_angularjs" nevű mappába. Ezt a "cd" paranccsal tehetjük meg, például "cd JavaScript keretrendszerek". A meghajtóváltás parancsa a "D:", vagy "C:", ez függ a meghajtó betűjelétől. Minden további keretrendszernél is ugyanezeket a műveleteket kell elvégezni.
Ha az "mma_in_angularjs" nevű mappába sikerült belépni, ki kell adni az "npm install" parancsot, amely feltelepíti a szükséges modulokat, majd az "npm start" parancsot, ami elindítja a szervert. A "http://localhost:3000" URL című oldalon megjelenik az "MMA Fighters" weboldal, ez esetben az AngularJS-ben írt verzió.
A főoldalon, a navigációs sávon lévő "Login" gombbal lehet bejelentkezni, itt a Facebook vagy Google szolgáltatások közül lehet választani, vagy a "Sign Up" gombra kattintva, az email cím és jelszó beírásával új felhasználót lehet létrehozni. A "Log In" gombra kattintva a program betölti a főoldalt, ahol két gomb található: az "ADD NEW FIGHTER" és a "VIEW THE FIGHTERS". Az "ADD NEW FIGHTER" nevű

gombra kattintva megjelenik az új harcost létrehozó oldal, ahol a mezők kitöltése után a "Submit" gombra kattintva létre lehet hozni egy új harcost, az oldal ezt követően a harcosokat megjelenítő táblázathoz irányít át. A "Go Back" nevű gombra kattintás után szintén a harcosok listája jelenik meg. Az új harcos létrehozására szolgáló oldal a navigációs sávon lévő "Add new fighter" nevű gombbal is behozható.
A "VIEW THE FIGHTERS" nevű gombra kattintva pedig szintén a harcosokat megjelenítő táblázat jelenik meg. Itt a táblázatban minden harcos sorának végén az "Action", vagyis "Esemény" felirat alatt lévő "View Details" linkre történő kattintás után az adott harcosnak az adatait megjelenítő oldal jön be.
Itt az "Edit" gombra kattintva a harcos adatait módosító oldal jelenik meg, ahol a mezők értékeinek változtatása után a "Submit" gomb lenyomására a harcosokat megjelenítő lista oldala jelenik meg, ahogy a "Delete" gomb és a "Go Back" gomb megnyomása után is. A "Go to fighter page" gombra való kattintás a harcos "page_url" mezőjében megadott weboldalra navigál.
Végül a navigációs sávon lévő "Logout" gombra kattintva a kijelentkezés utáni kezdőoldal jelenik meg, ahol a "Login" gomb található. Ez a felépítés mind a négy keretrendszer esetében az itt leírtaknak felel meg, viszont a projekt elindítása változik a mappába való navigálást követően.

Az Angular 2 esetében az "Angular 2" mappába és az "mma_in_angular2" mappába való belépés után ki kell adni az "npm install" parancsot, amely feltelepíti a szükséges modulokat, majd az "npm start" parancsot, ami elindítja a szervert. Ezután egy újabb "Node.js command prompt" nevű parancssoros alkalmazást kell nyitni. Az "npm install" parancs utáni esetleges hibák elkerülése végett célszerű adminisztrátorként indítani a programot. A program ezen ablakában a "cd src" parancs, majd az "ng build" parancs kiadása szükséges. Ezután a "http://localhost:3000" URL című oldalra lépve jelenik meg az Angular 2 keretrendszerbeli alkalmazás.

A Vue.js esetében az "mma_in_vuejs" nevű mappába való navigálás után a "server" nevű mappába történő belépés szükséges, ahol ki kell adni az "npm install", majd az "npm start" parancsokat, mint az előző keretrendszerek esetében. Ugyanezeket a parancsokat szintén ki kell adni az "mma_in_vuejs" nevű mappában lévő "client" almappába való belépés után is. Ezután a "http://localhost:8080" URL című oldalra navigálva jelenik meg a Vue.js-ben írt alkalmazás.

A React.js esetében az "mma_in_reactjs" mappába történő belépés után az "npm install" parancsot kell beírni, majd az "npm run build", majd az "npm start" parancsok kiadását követően a "http://localhost:3000" URL című oldalra navigálva megjelenik a React.js-ben írt alkalmazás.