Izgradnja RESTful aplikacije s AngularJS-om

Šimić, Ivana

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni nastavnčki studij matematike i informatike

Ivana Šimić

Izgradnja RESTful aplikacije s AngularJS-om

Diplomski rad

Osijek, 2016.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni nastavnčki studij matematike i informatike

Ivana Šimić

Izgradnja RESTful aplikacije s AngularJS-om

Diplomski rad

Voditelj: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2016.

Sadržaj

1. Uvod

Prve web stranice su bile poput enciklopedija - sastojale su se od mnogo informacija koje su korisnici mogli čitati, ali pri tome nisu mogli utjecati na sadržaj ili izled stranice. Zbog toga se pojavila potreba za alatom koji će omogućiti uvođenje interakcije s korisnikom te mu na taj način dozvoliti mijenjanje sadržaja stranice. Na primjer, danas to možemo postići dodavanjem svojih komentara na objavljeni članak, blog post i slično. U tu svrhu je kreiran skriptni jezik JavaScript. Ideja je bila napraviti jezik kojeg će početnici lako shvatiti i jednostavno ga ubaciti u HTML dokument svoje stranice, te time osigurati njezinu interaktivnost. Izvođenje njegovog kôda bi se odvijalo u web pregledniku.

Danas JavaScript možemo koristiti za programiranje kompletne aplikacije - od serverskog do klijentskog kôda. To mogu biti vrlo kompleksne aplikacije, te je zbog toga potrebno dobro strukturirati kôd kao bismo ga mogli lakše održavati. Pri tome nam može pomoći JavaScript framework koji nam najviše odgovara - možemo ga odabrati uzimajući u obzir koliko nam je jasna njegova struktura, ili određene potrebe aplikacije. Napomenimo kako nije nužno da cijela aplikacija bude rađena u JavaScriptu. Često je klijentski dio aplikacije (dio čiji kôd se izvršava u web pregledniku) izrađen u JavaScriptu, dok je serverski dio rađen u nekom drugom jeziku.

U ovom radu ćemo pažnju usmjeriti na klijentsku stranu aplikacije. Ukratko ćemo objasniti koncept arhitekture aplikacije rađene u AngularJS frameworku, te način na koji ona komunicira sa serverskom aplikacijom. Objasnit ćemo kako od serverske strane tražiti da manipulira podacima iz baze podataka. Pri tome ćemo koristiti primjere kôda aplikacije izrađene u sklopu ovog rada.

2. JavaScript

JavaScript je programski jezik Weba. Dio je trojke koju svaki web developer mora naučiti: HTML[1] za definiranje sadržaja, CSS[2] za definiranje izgleda, te JavaScript za definiranje ponašanja web stranice. Nastao je kao skriptni jezik ugrađen u HTML, ali je prerastao tu svoju prirodu i postao bogat jezik za generalnu upotrebu.

Također, treba napomenuti da JavaScript ne treba miješati s programskim jezikom Java. Osim površne sličnosti u sintaksi, JavaScript je potpuno drugačiji od Jave.

U ovom poglavlju donosimo kratku povijest JavaScripta, objasnit ćemo razliku između JavaScripta na klijentskoj i serverskoj strani. Također donosimo pregled popularnih frameworka[3] ovog jezika te kratak uvod u objektno orjentirano programiranje u JavaScriptu.

2.1. Povijest

JavaScript je kreirao Brendan Eich 1995. godine dok je radio u Netscapeu. Ideja je bila napraviti skriptni jezik koji bi bio ugrađen u HTML za web preglednik Netscape Navigator. Tako bi web dizajnerima bilo olakšano unošenje animacija i interaktivnosti u web stranice kako bi one bile manje statične.

Prvo ime JavaScripta bilo je Mocha. Međutim, Netscape je u to vrijeme kao prefiks imena svojih proizvoda dodavao *Live*, te je, u rujnu 1995., ime jezika promjenjeno u LiveScript. U prosincu iste godine je Netscape dobio licencu od tvrtke Sun (današnji Oracle) te je jezik dobio ime koje koristimo i danas - JavaScript. Mnogi smatraju da je ta promjena imena bila marketinški potez te je doprinijela popularnosti jezika jer je tada Java bila vrlo popularna među programerima.

Microsoft je na Netscapeovu objavu JavaScripta za Netscape Navigator odgovorio svojim jezikom za Internet Explorer. Svoju implementaciju dobili su obrnutim inžinjeringom JavaScripta, te ju nazvali JScript. Postalo je jasno da je potrebna standardizacija jezika kako bi programeri mogli biti sigurniji da će njihov kôd raditi u više preglednika. Prvu definiciju standarda za JavaScript donijela je ECMA (European Computer Manufactures Association) 1997. godine. Međutim, kako niti jedna strana nije htjela dati pravo na svoje ime (JavaScript ni JScript), standard je nazvan ECMAScript. Danas većina web preglednika ima podršku za ECMAScript 6 standard.

2.2. *Server-side* i *client-side* JavaScript

JavaScript se može koristiti za cjelokupnu izradu web aplikacija. Dakle, možemo ga koristiti i na serverskoj i na klijentskoj strani. To je zapravo i bila originalna ideja

[1] HTML - HyperText Markup Language
[2] CSS - Cascading Style Sheets
[3] framework - vidi potpoglavlje 2.3. Framework

Netscapea - osmisliti jezik koji će olakšati izradu i održavanje web stranica. Korištenje samo jednog jezika na obje strane aplikacije slijedi upravo tu ideju. Ovo potpoglavlje je samo uvod u korištenje JavaScripta na serverskoj i klijentskoj strani.

Server-side JavaScript (dalje u tekstu SSJS) je termin koji se odnosi na JavaScript kôd koji se izvršava izvan web preglednika. JavaScript kôd na serverskoj strani ima pristup bazama podataka, *file* sistemima i serverima. Prva implementacija SSJS-a bio je Netscapeov LiveWire 1996. godine. Do danas su mnoge tvrtke dale svoje implementacije SSJS-a od kojih treba spomenuti onu Microsoftovu - danas poznatu kao ASP. Uz VBScript, navedena implementacija podržavala je JavaScript i PerlScript.

Ekstenzija JavaScripta na server omogućena je JavaScript engineima. Neki od njih su SpiderMonkey implementiran u programskom jeziku C, te Rhino implementiran u programskom jeziku Java. Osim navedena dva, bitno je spomenuti i Node.js. Node.js je JavaScript framework koji koristi Googleov JavaScript engine V8. On omogućuje izradu web servera i mrežnih alata koristeći JavaScript i kolekcije modula. Moduli pokrivaju input i output, mrežne protokole, kriptografske funkcije itd.

Client-side JavaScript (dalje u tekstu CSJS) je termin koji se koristi za JavaScript kôd koji se izvršava u web pregledniku. Ovakav kôd možemo pisati u zaseban dokument s nastavkom .js kojeg tada uključimo u dokument u kojoj se nalazi naš HTML kôd. Osim toga, možemo ga pisati i unutar samog HTML dokumenta. U tom slučaju kôd pišemo unutar `script` tagova.

CSJS nam omogućuje kontrolu nad ponašanjem web stranice. Na primjer, možemo odrediti što će se dogoditi kada korisnik klikne na gumb koji se nalazi na stranici. Pri tome možemo, na primjer, korisniku prikazati upozorenje, tražiti od njega da potvrdi da želi nastaviti dalje koristiti aplikaciju, preusmjeriti ga na drugu web stranicu i slično. Osim toga, možemo mijenjati i strukturu HTML dokumenta te stranice.

Osim korištenja čistog JavaScript kôda na klijentskoj strani, kao i na serverskoj, možemo korisiti JavaScript frameworke. U sljedećem potpoglavlju donosimo pregled nekoliko frameworka za SSJS i CSJS.

2.3. Framework

Kako je navedeno u prošlom potpoglavlju, ne moramo pisati čisti JavaScript kôd kako bismo koristili JavaScript. Umjesto toga, možemo koristiti framework koji će nam olakšati korištenje JavaSrcipta. Međutim, što je framework?

Framework je koncept koji nam olakšava strukturiranje kôda aplikacije. Kod frameworka je naglasak više na arhitekturi aplikacije nego na sintaksi jezika. Svaki framework može sadržavati alate, biblioteke i API-e vezane zajedno. JavaScript framework ima metode koje nam olakšavaju korištenje JavaScripta. Tako, korištenjem frameworka, često ne moramo pisati drugačiji kôd za različite web preglednike - framework već sam pazi na to da nam kôd radi u svima (ili barem većini). Metode frameworka možemo

koristiti kao gotove, ne brinući se pri tome o kôdu o kojemu one možda ovise. Kako bismo koristili framework na klijentskoj strani, potrebno ga je uključiti u HTML dokument. Često možemo vidjeti da za neki framework piše da je MVC framework. Prije pregleda JavaScript frameworka, pogledajmo prvo što to znači:

MVC (*Model - View - Controller*) je arhitekturalni koncept prema kojem je aplikacija podjeljena na tri cjeline: na podatke (*Model*), prezentaciju tih podataka korisniku (*View*), te ponašanje aplikacije u ovisnosti o korisnikovim akcijama (*Controller*). Dakle, korisnik nešto učini na stranici. To se proslijedi Controlleru koji kontrolira što će se dogoditi. Često to znači da će od Modela tražiti podatke te ih dati Viewu da ih prezentira korisniku. Promotrimo to na primjeru:

Korisniku je prikazana forma za prijavu na stranicu. To što korisnik vidi je View. Neka je korisnik upisao svoje korisničke podatke i kliknio na gumb za prijavu. Kada je kliknio na taj gumb, Controller je poslao podatke Modelu, te traži od njega da mu kaže postoji li registrirani korisnik s tim podacima. Dalje Controller, na temelju odgovora, odlučuje što View treba prikazati. Ako postoji takav korisnik, može korisnika preusmjeriti na njegov profil, a ako ne postoji, prikazati mu odgovarajuću poruku. Pri tome View odlučuje o prezentaciji poruke koju korisnik vidi. Dakle, Model i View zapravo komuniciraju preko Controllera.

Sljedeća tablica donosi pregled nekoliko SSJS frameworka:

Framework	JavaScript engine	Platforma servera
Apache Sling	Rhino	Java servlet container
Aptana Jaxer	SpiderMonkey	Apache HTTP server
ASP	JScript	IIS
Helma	Rhino	Jetty HTTP server
Node.js	V8	/[4]

Tablica 1: SSJS frameworci

Pogledajmo sada CSJS frameworke:

– *AngularJS* je opensource JavaScript framework kojeg održava Google. Omogućuje lakšu izradu i testiranje web aplikacija, te podržava *Dependency Injection*[5]. Primjeri aplikacija izrađenih pomoću AngularJS-a su Youtube on PS3 i yap.TV Facebook Application.

– *Backbone.js* je MVC framework koji se može kombinirati uz bilo koji *library* (npr. jQuery). Pomoću Knockoutback.js-a mu možemo dodati funkcionalnost Knockout.js frameworka. Primjeri aplikacija izrađenih pomoću Backbone.js-a su Sony Entertainment Network i Tracqlist.

[4]Nije potrebna posebna platforma

[5]Koncept u kojem je moguće *ubaciti* objekte u klase umjesto oslanjanja na samu klasu da kreira objekt

– *Ember.js* je MVC framework koji se može koristiti uz jQuery. Njegov Ember inspectora olakšava debugiranje aplikacija. Primjeri aplikacija izrađenih pomoću Ember.js-a su Apple Watch User Guide i Talentbuddy.

– *Knockout.js* je MVVM (*Model - View - ViewModel*) framework kojeg podržava velik broj web preglednika. Nije ovisan o drugim frameworcima. Primjeri aplikacija izrađenih pomoću Knockout.js-a su JsFiddle i eventim.de.

– *React.js* je JavaScript framework kojeg održava Facebook. Primjeri aplikacija izrađenih korištenjem React.js-a su Instagram i Khan Academy.

2.4. Objekti

Za bolje razumijevanje AngularJS frameworka, potrebno je shvatiti i rad s objekima u JavaScriptu. Naime, osnovni tip podataka u JavaScriptu je `object`. Osim toga, u JavaScriptu sve podatke možemo promatrati kao objekte.

Objekt u JavaScriptu sadrži kolekciju *key:value* parova koje nazivamo *property*. Svaki property ima svoje ime (*key*) i vrijednost (*value*). Vrijednost može biti bilo što, a imena su stringovi. Stoga možemo reći da JavaScript objekt mapira stringove u vrijednosti. Ovakva struktura se često može pronaći pod imenima *hashtable*, *dictionary* i *associative array*.

Bitno je napomenuti da objektima manipuliramo preko reference. Što to znači? Neka varijabla `x` referencira objekt. Tada izvršavanjem kôda `var y = x;` nije stvoren novi objekt. Varijabla `y` referencira na isti objekt kao i varijabla `x`. Tako će svaka promjena na tom objektu ostvarena putem jedne od te dvije varijable biti vidljiva i putem druge.

2.4.1. Kreiranje objekata

Objekte možemo kreirati na tri načina: korištenjem objektnih literala, operatora `new` te funkcije `Object.create()`. Pogledajmo svaki od navedenih načina:

Objektni literali

Objektni literal je lista key:value parova odvojenih zarezom i grupiranih unutar vitičastih zagrada. Svaki put kada je evaluiran izraz, objektni literal kreira novu instancu objekta. To znači da, ako ga koristimo unutar petlje, u svakoj iteraciji petlje u kojoj je izraz evaluiran, kreira se novi objekt. Pri tome kreirani objekti ne moraju nužno imati iste vrijednosti. Pogledajmo nekoliko primjera kreiranja objekta na ovaj način:

Primjer 2.1

```
var obj1 = {};                          // objekt bez propertya
var obj2 = { x:0, y:0 };                // objekt s dva propertya
var obj3 = { x:obj2.x+1, y:obj2.y+1 }; // objekt čije vrijednosti ovise
                                        // o vrijednostima objekta obj2
var obj4 = {
  "ime i prezime": "Pero Peric",
  "datum-rodjenja": {                   // vrijednost propertya može biti
    dan: 1,                             // objekt
    mjesec: "svibanj",
    godina: 1986
  }
}
```

U ECMAScript 5 standardu zarez nakon zadnjeg propertya se ignorira, dok se u nekim implementacijama ECMAScript 3 standarda i u pregledniku Internet Explorer evaluira kao pogreška.

Operator `new`

Operator `new` kreira i inicijalizira novi objekt. Iza ključne riječi `new` mora biti invokacija funkcije. Takvo korištene funkcije nazivamo *konstruktori*. Oni služe za inicijalizaciju novog objekta. U JavaScriptu već postoje konstruktori za nativne tipove podataka:

Primjer 2.2

```
var o = new Object(); // prazan objekt, evivalentno korištenju {}
var a = new Array();  // prazno polje, ekvivalentno korištenju []
var d = new Date();   // Date objekt koji predstavlja trenutno vrijeme
```

Također, u JavaScriptu su i funkcije objekti i imaju svoj prototip `Function.prototype` te postoji kostruktor za njihovo kreiranje:

Primjer 2.3

```
var f = new Function("x", "y","return x+y;");
```

Funkcija `f` kreirana u gornjem primjeru za parametre prima x i y, te kao vrijednost vraća njihov zbroj. Međutim, osim navedenog načina, funkcije u JavaScriptu možemo kreirati na još dva, za njih sprecifična načina:

Primjer 2.4

```
var f1 = function (x, y) {
    return x + y;
};
```

```
function f2(x, y) {
    return x + y;
}
```

U oba gore navedena pristupa kreiramo objekte čiji prototip je `Function.prototype`. Međutim, postoje razlike. Promotrimo sljedeći primjer:

Primjer 2.5

1.
```
var z = f1(1,6);
var f1 = function (x, y) {
    return x + y;
};
```

2.
```
var z = f2(1,6)
function f2(x, y) {
    return x + y;
}
```

Prvi pristup će rezultirati greškom, dok će drugi varijabli `z` dodijeliti vrijednost `7`. To je zato što se funkcija dodjeljena varijabli `f1` zapravo kreira za vrijeme izvršavanja kôda, te u trenutku poziva funkcije nije još definirana. U drugom pristupu se funkcija kreira za vrijeme parsiranja bloka kôda, pa je definirana prije kreiranja varijable `z`, tj. prije prvog poziva funkcije.

Napomenimo još da je, osim korištenja postojećih prototipova, moguće kreiranje novih prototipova.

`Object.create()`

ECMAScript 5 standard uveo je novu metodu za kreiranje objekata, a to je kreiranje korištenjem statičke funkcije `Object.create()`. Prvi argument navedene funkcije je prototip objekta kojeg kreiramo. Drugi element je opcionalan i detalje oko tog elementa možete pronaći u [1].

Kako bismo razumjeli ovaj način kreiranja objekta, potrebno je objasniti prototip. Naime, svaki JavaScript objekt ima drugi objekt (ili `null` što je rijetko) povezan s njim. Taj objekt nazivamo prototip (*prototype*). Od njega objekt nasljeđuje propertye. Svi objekti kreirani objektnim literalima imaju isti prototip - `Object.protptype`. Objekti kreirani pomoću operatora `new` za svoj prototip uzimaju `prototype` svog konstruktora. To znači da objekti kreirani pomoći `new Object()` nasljeđuju propertye od `Object.prototype` objekta kao i objekti kreirani pomoću {}. Slično, objekti kreirani pomoću `new Array()` nasljeđuju propertye od objekta `Array.prototype`, a objekti kreirani pomoću `new Date()` nasljeđuju propertye od objekta `Date.prototype`.

`Object.prototype` je jedan od rijetkih objekata u JavaScriptu koji nema svoj prototip. To znači da on ne nasljeđuje propertye niti od jednog objekta. Drugi prototype objekti imaju svoje prototipe. Na primjer, prototip objekta `Date.prototype` je `Object.prototype`. To znači da objekt kreiran pomoću `new Date()` nasljeđuje propertye od objekta `Date.prototype`, ali i od objekta `Object.prototype`. Ovako povezana serija prototype objekata naziva se lanac prototipa (*prototype chain*).

Vratimo se sada na kreiranje objekta. Kako bismo kreirali objekt pomoću funkcije `Object.create()`, potrebno je kao parametar poslati objekt koji će biti prototip novog objekta.

Primjer 2.6

```
var objP = {};
var obj = Object.create(objP);
```

Ovako kreiran objekt `obj` za prototip ima `obj1`. Osim objekata, moguće je poslati i vrijednost `null`, ali tada novi objekt neće imati svoj prototip. Tako neće naslijediti niti jedan property, pa tako ni osnovne metode poput metode `toString()`.

Ako želimo kreirati obični prazni objekt, kao parametar funkcije `Object.create()` trebamo poslati objekt `Object.prototype`. Tako su sljedeći načini kreiranja objekata ekvivalentni:

Primjer 2.7

```
var obj1 = {};
var obj2 = new Object();
var obj3 = Object.create(Object.prototype);
```

2.4.2. Rad s propertyma

Vrijednost propertya JavaScript objekta možemo dohvatiti pomoću dva operatora: *dot* operator (`.`) i uglate zagrade (`[]`). Pri tome, s lijeve strane jednakosti mora biti varijabla koja će referencirati dohvaćenu vrijednost. Ako koristimo *dot* operator, na desnoj strani jednakosti stoje, redom, ime objekta, točka i ime propertya čiju vrijednost želimo dohvatiti. Ako koristimo `[]`, s desne strane izraza stoje ime objekta, te u zagradama izraz koji se može pretvoriti u string. Taj string treba biti ime propertya čiju vrijednost želimo dohvatiti. Pogledajmo primjer:

Primjer 2.8

```
var mj = datum.mjesec;            // dohvaća vrijednost propertya
                                  // mjesec objekta datum
var ime = osoba["ime i prezime"]; // dohvaća vrijednost propertya
                                  //"ime i prezime" objekta osoba
```

Dakle, sljedeća dva izraza imaju jednaku vrijednost:

```
obj.property
obj["property"]
```

Kada koristimo sintaksu kao kod prvog izraza, vrijednosti propertya objekta pristupamo preko indetifikatora. Identifikatori moraju biti doslovno napisani u JavaScript kodu, te njima nije moguće manipulirati dok se program izvršava. S druge strane, kada koristimo sintaksu kao kod drugog izraza, imena propertya su izražena kao stringovi. Stringovima možmo manipulirati za vrijeme izvršavanja programa. Tako, na primjer, možemo pisati `osoba["adresa"+1]`. Na taj način dohvaćamo vrijednost propertya `adresa1` objekta `osoba`. Međutim, ovdje smo mogli koristiti i *dot* operator. Promotrimo sljedeći primjer:

Recimo da pišemo JavaScript program koji će studentu omogućiti unos naziva i ocjena kolegija. Unesene vrijednosti ćemo spremati u objekt `indeks`, a property će imati sljedeći oblik: `"kolegij":ocjena`. Kako student kolegije upisuje za vrijeme izvršavanja programa, ne možemo znati koje kolegije će unijeti. Stoga ne možemo koristiti *dot* operator. Dakle, propertye ćemo postavljati (i njihove vrijednosti dohvaćati) pomoću uglatih zagrada. Na primjer, dio našeg programa mogla bi biti funkcija za unos novih kolegija:

```
function unosOcjene(indeks, kolegij, ocjena) {
  indeks[kolegij] = ocjena;
}
```

Vidimo da nam je ovdje `indeks` objekt u koji spremamo vrijednosti, `kolegij` je string koji označava ime propertya navedenog objekta, a `ocjena` je vrijednost propertya `kolegij`.

Ovdje možemo nadovezati još jedan primjer u kojem koristimo uglate zagrade. Pomoću ovog operatora možemo dohvatiti ocjene iz svih kolegija studenta kako bismo izračunali njegov prosjek. Pogledajmo funkciju kojom ćemo to izračunati:

```
function prosjek(indeks) {
  var sumaOcjena = 0.0;
  var brojOcjena = 0;

  for(var kolegij in indeks) {
    sumaOcjena += indeks[kolegij];
    brojOcjena++;
  }
  if(brojOcjena>0)
    return sumaOcjena/brojOcjena;
  else
    return 0;
}
```

Vidjeli smo kako dohvatiti i promijeniti vrijednost propertya nekog objekta. Ali, što ako property kojeg želimo dohvatiti ne postoji? Dohvaćanje vrijednosti takvog propertya neće rezultirati greškom. Na primjer, naš `obj4` iz primjera 2.1 ima property `"ime i prezime"`. Ako pokušamo dohvatiti vrijednost propertya `obj4.prezime`, kao rezultat ćemo dobiti vrijednost `undefined`. Međutim, ako pokušamo dohvatiti property objekta koji ne postoji, dobit ćemo `TypeError`. To će se dogoditi ako, na primjer, pokušamo sljedeće: `var len = obj4.prezime.length;` jer `obj4.prezime` ne postoji. Dakle, ako nismo sigurni da postoje objekti, ne bismo trebali pokušavati dohvaćati njihove propertye. Ipak, postoji način da izbjegnemo `TypeError`:

Primjer 2.9

```
var len = undefined;
if(obj4) {
  if(obj4.prezime) {
    len = obj4.prezime.length;
  }
}
```

U ovom slučaju će vrijednost `len` varijable ostati `undefined` ako ne postoji objekt `obj4.prezime`. U suprotnom će poprimiti vrijednosti propertya `obj4.prezime.length`. Također, `TypeError` ćemo dobiti i ako pokušamo dohvatiti property `null` ili `undefined` vrijednosti.

Osim slučajeva u kojima traženi property ne postoji, može se dogoditi da pokušamo promijeniti vrijednost propertya koji je *read-only*. Takve propertye ne možemo mijenjati. Takvi su, na primjer, `prototype` propertyi konstruktora ugrađenih u JavaScript. Ipak, ako pokušamo promijeniti takav property, to neće rezultirati greškom, ali property će ostati nepromijenjen. To znači da izvršavanje kôda `Object.prototype = 0;` neće uzrokovati promjene na navedenom propertyu.

Osim kreiranja novih, moguće je i brisati stare propertye. U tu svrhu koristimo operator `delete`. Bitno je napomenuti da ovaj operator ne briše nasljeđene propertye. To znači da ne možemo obrisati property prototipa objekta bez da ga obrišemo na samom prototipu. Međutim, kada obrišemo property prototipa, niti jedan objekt koji je nasljedio objekte tog prototipa više neće imati taj property. Izraz u kojem brišemo property daje vrijednost `true` ako je property uspješno obrisan ili ako `delete` nije imao učinka. `delete` ne može obrisati propertye kojima je atribut `configurable` postavljen na `false`. Tako na primjer ne možemo obrisati property `Object.prototype`. Također, nije moguće obrisati globalni property. Pogledajmo sljedeći primjer:

Primjer 2.10 [6]

```
o = { x:1 };          // o ima property x i nasljeđuje property toString
delete o.x;           // obrisan property x objekta o
delete o.x;           // nema efekta (property ne postoji)
delete o.toString;    // nema efekta (toString je nasljeđen property)
delete 1;             // vraća vrijednost true

delete Object.prototype; // nije moguće obrisati
var x = 1;               // deklaracija globalne varijable
delete this.x;           // nije moguće obrisati
function f() {}          // deklaracija globalne funkcije
delete this.f;           // nije moguće obrisati
```

2.4.3. Serijalizacija

Serializacija objekta je proces pretvorbe stanja objekta u string iz kojega ga kasnije možemo vratiti. Ovaj postupak često primjenjujemo pri slanju podataka s klijentske strane aplikacije na serversku stranu. Vidjet ćemo kako izgleda JavaScript objekt serializiran u JSON, te kako izgleda objekt dobiven iz JSON-a.

JSON (*JavaScript Object Notation*) je format za razmjenu podataka neovisan o programskom jeziku. Od ECMAScript 5 standarda, u JavaScriptu postoje gotove funkcije za serijalizaciju objekata koje možemo koristiti. To su `JSON.stringify()` za pretvorbu objekta u JSON, te `JSON.parse()` pretvorbu JSON-a u objekt. Pogledajmo primjere korištenja navedenih funkcija:

Primjer 2.11 (Pretvorba objekta u JSON format)

```
var o = {x:1, y:{Z:3, w:5}}; // definiranje objekta
var s = JSON.stringify(o); // s je '{"x":1, "y":{"Z":3, "w":5}}'
```

Primjer 2.12 (Dohvaćanje objekta iz JSON formata)

```
// o je objekt koji ima property x, a x je polje [1, 5, false]
var o = JSON.parse('{"x":[1, 5, "false"]}');

// SyntaxError - JSON ne dozvoljava zarez nakon zadnjeg propertya
JSON.parse('{"x":[1,2,3,4],}');
```

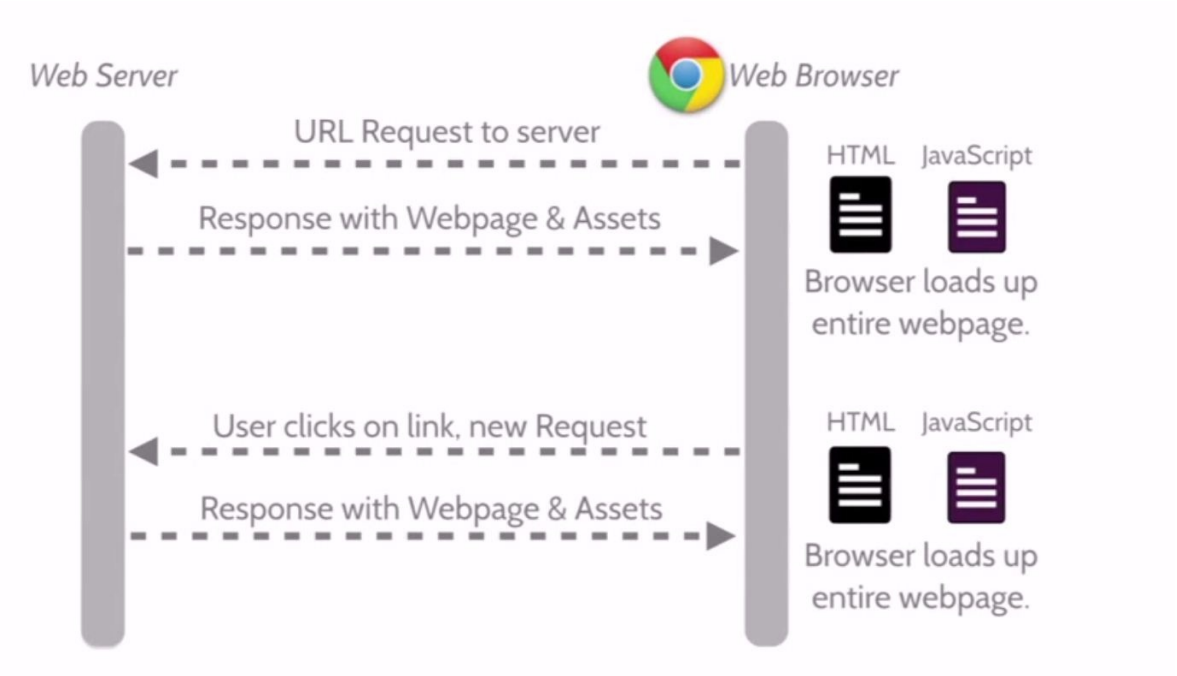
Bitno je napomenuti da JSON ne može prikazati sve JavaScript vrijednosti. Naime, objekti, polja, stringovi, konačni brojevi, `true`, `false` i `null` su podržani, ali `NaN`, `Infinity` i `-Infinity` se serijaliziraju u `null` vrijednost. Nadalje, funkcije, RegExp i Error objekti te `undefined` vrijednost ne mogu biti serijalizirani te su izostavljeni u JSON formatu serijaliziranog objekta.

[6] Primjer preuzet iz [1]

3. AngularJS

AngularJS (dalje u tekstu Angular) je CSJS framework za izradu dinamičkih web stranica. To je projekt otvorenog kôda i podržava ga Google.

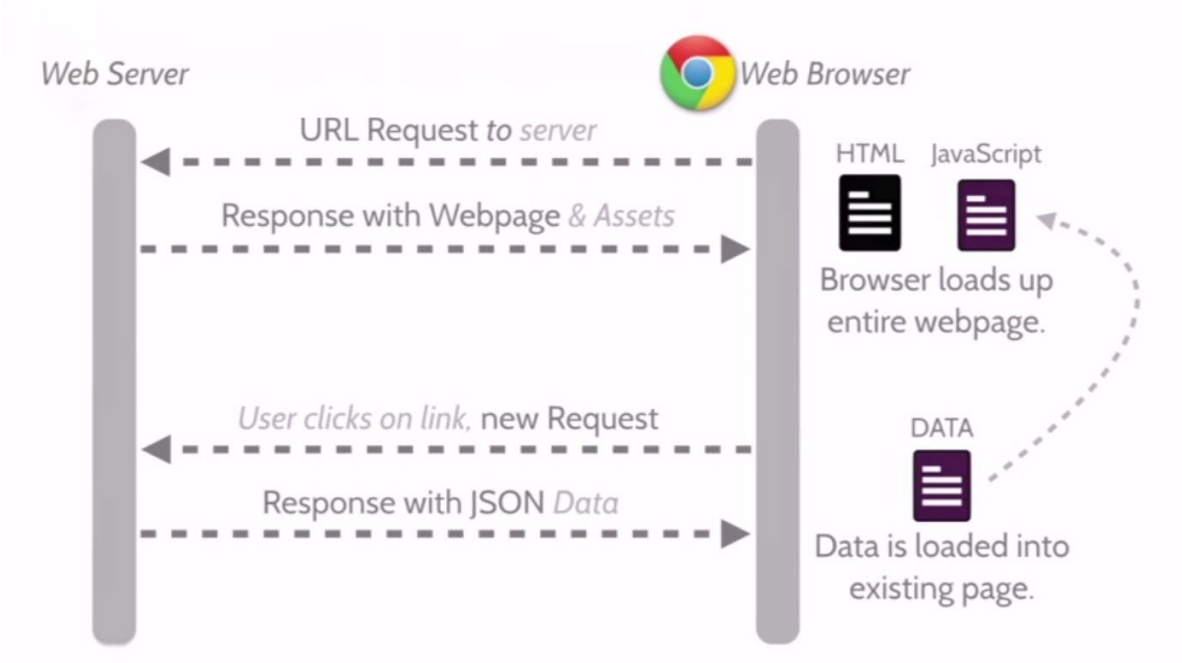
Angular nam pomaže strukturirati JavaScript kôd, lako ga je testirati i pomoću njega možemo efikasne web aplikacije. O strukturi Angular kôda ćemo više vidjeti kasnije, a pogledajmo sada što to pridonosi učitavanju (tj. osvježavanju) web stranice. S jedne strane imamo server, a s druge strane klijenta. Klijent će nam biti web preglednik. Kada korisnik u web preglednik unese URL, na server se šalje zahtjev. Server tada odgovara tako da pošalje web stranicu i njezine dodatke, te se sve učita u web preglednik. Zatim, kada korisnik klikne na npr. vezu, web preglednik na server šalje novi zahtjev. Server tada ponovo šalje cijelu web stranicu sa svim dodacima, te se sve ponovo učita u web preglednik. Donja slika prikazuje takav scenarij:



Slika 1: Klasična komunikacija klijentske aplikacije sa serverom[7]

Kada radimo s Angularom proces je malo drugačiji. Ipak, početak je isti - korisnik unese URL u web preglednik te se šalje zahtjev na server. Server tada šalje cijelu web stranicu sa svim dodacima i to se sve učita u web preglenik. Kada korisnik klikne na vezu, web preglednik šalje novi zahtjev na server. Međutim, server sada ne šalje cijelu stranicu kao prije, nego samo podatke u JSON formatu. JavaScript kôd stranice tada obrađuje dobivene podatke, te na taj način odlučuje što će korisnik vidjeti na ekranu. Na donjoj slici vidimo prikaz ovakve komunikacije:

[7]Slika preuzeta sa Code School tutoriala ([7])



Slika 2: Komunikacija Angular aplikacije sa serverom[8]

3.1. Elementi Angular aplikacije

Prvi korak koji je potrebno napraviti kako bismo omogućili korištenje Angular frameworka na web stranici, potrebno je učitati angular kôd. Zatim, kako bismo povezali Angular aplikaciju s HTML dokumentom, potrebno je u `html` ili `body` tag dokumenta napisati `ng-app="imeAplikacije"`. Na taj način definiramo koja aplikacija će imati kontrolu nad HTML dokumentom. Na primjer, početak HTML dokumenta može izgledati ovako:

Primjer 3.1 (Povezivanje HTML dokumenta i Angular aplikacije)

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" ng-app="onlineExamPrep">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>Online ispiti</title>
    <script src="scripts/jquery-2.1.4.js"></script>
    <script src="scripts/angular.js"></script>
</head>
```

Bitno je napomenuti da je moguće referencirati samo jednu aplikaciju koristeći `ng-app`. Ako u HTML dokumentu više puta spomenemo `ng-app`, Angular će u obzir

[8]Slika preuzeta sa Code School tutoriala ([7])

uzeti samo prvi, te će ta aplikacija imati kontrolu nad dokumentom. Međutim, to ne znači da svoju aplikaciju ne možemo podijeliti na više smislenih cjelina. U tu svrhu koristimo angular module.

3.1.1. Module

Angular modul je cjelina koja se može sastojati od nekoliko vrsta komponenata: controller, directive, service, constant, ... Modul kreiramo na sljedeći način:
`angular.module('imeModula', [ovisnosti])`
Time stvaramo modul naziva 'imeModula', te u uglatim zagradama pišemo nazive modula o kojima on ovisi. To su moduli koje želimo koristiti unutar svog modula. Na primjer, to mogu biti third-party moduli, ili moduli koje smo sami pisali.

Primjer 3.2 (Kreiranje i registracija Angular modula)

```
angular.module('onlineExamPrep', [
    'ngAnimate',
    'ui.bootstrap',
    'ui.router',
    'onlineExamPrep.common'
]);
```

Tako registriran modul dalje možemo dohvaćati te na njega slagati komponente, što ćemo vidjeti prilikom kreiranja controllera, directive i servicea.

3.1.2. Service

Angular servis koristimo za organizaciju kôda koji kasnije možemo koristiti u direktivama i kontrolerima. Na primjer, možemo u servis pisati kôd za dohvaćanje podataka sa servera. Tada je u direktivama potrebno samo injectati servis i pozvati njegovu metodu. Jedna od razlika između direktiva i servisa je ta da servis ne vežemo izravno na HTML dokument niti mu dajemo njegov template, nego njegove metode koristimo unutar direktive ili kontrolera. Također, servis je singleton, što znači da se kreira samo jedna instanca, te se ta instanca koristi gdje god je zareferenciran. S druge strane, direktive možemo kreirati više puta, te se svaki puta kreira nova instanca.

Primjer 3.3 (Kreiranje Angular servisa)

```
angular.module('onlineExamPrep.common')
    .service('ExamService', function (DataService, Paths) {
        var path = Paths.api.endpoint + Paths.api.paths.exam;

        this.getExamCollection = function (pagingParams) {
            return DataService.post(path + '/getPage', pagingParams);
        };
    });
```

U gornjem primjeru je prikazano kreiranje angular servisa koji ovisi o drugim servisima, te ima jednu metodu koji možemo koristiti gdje god ga zareferenciramo. Vidimo da servis kreiramo tako da metodi `.service()` proslijedimo ime servisa, te unutar funkcije koja ga kreira pišemo servise koje koristimo unutar njega. Metode servisa vežemo na sâm objekt, te pišemo `this.imeMetode`. Servis kreiran u primjeru je vezan na modul `onlineExamPrep.common`.

Servisi koji dolaze uz angular imaju prefiks `$`. Tako, na primjer, postoje servisi `$document` za dohvaćanje HTML elemenata, `$window` za dohvaćanje vrijednosti vezanih za prozor u kojemu je aplikacija, te `$http` za slanje HTTP reqestova, o kojem ćemo više reći u poglavlju 4.

3.1.3. Controller

Kontroler u Angularu koristimo za definiranje ponašanja HTML kôda. Kreiramo ga tako da metodi `.controller()` proslijedimo ime kontolera, servise o kojima ovisi, te funkciju za kreiranje kontroler objekta:

Primjer 3.4 (Kreiranje Angular controllera)

```
angular.module('onlineExamPrep')
.controller('examController', ['ExamService', function (ExamService) {
     // kôd za kreiranje controller objekta
    }]);
```

Osim samog kreiranja kontrolera, potrebno ga je i vezati na HTML dokument. U tu svrhu koristimo `ng-controller="ctrl"` čime zapravo kažemo da će tim HTML elementom i svim njegovim child elementima upravljati kontroler `ctrl`.

Primjer 3.5 (Vezanje kontrolera na dokument)

```
<div ng-controller="ctrl">
    <!-- saržaj div elementa, ovdje kontroler ctrl ima utjecaja -->
</div>
```

Bitno je napomenuti i to da kontrolere ne bismo trebali koristiti za manipulaciju DOM elementima, nego u tu svrhu koristiti angular direktive.

3.1.4. Directive

Kada neku komponentu sa cijelim njenim markup-om, te njenom funkcionalnosti želimo koristiti na nekoliko mjesta, dobro je koristiti direktive. Naime, njima možemo dodijeliti HTML template te time izbjeći pisanje istog kôda na više mjesta. Pri tome im template možemo dodijeliti inline - kao string, ili referenciranjem na HTML datoteku. Ipak, radi lakšeg održavanja kôda, dobro je koristiti drugi navedeni način.

Direktive u Angularu možemo opisati kao markere na DOM elementima koji HTML

compileru govore kako se taj element treba ponasati. Dakle, ovom slučaju *kompajliranje* znači povezati direktivu s HTML elementom za omogućavanje interaktivnosti tog elementa.

Primjer 3.6 (Angular direktiva)

```
angular.module('onlineExamPrep.components')
    .directive('oepAlternativeAnswer', function (Paths) {
        'use strict';
        return {
            restrict: 'E',
            templateUrl: Paths + 'question-types/alternative-answer.html',
            scope: {
                vm: '=',
                enableAnswer: '='
            },
            link: function (scope) {
            }
        };
    });
```

Promotrimo gornji kôd: kao i kod kreiranja kontrolera, moramo prvo napisati u kojem modulu se direktiva nalazi. Zatim, unutar metode `.directive` pišemo ime direktive, te funkciju pomoću koje će ona biti kreirana. Toj funkciji kao parametre šaljemo servise koji su nam potrebni unutar nje. Pogledajmo objekt koji ona vraća: `restrict` označava način na koji se direktiva koristi. Slovo `E` znači da ovu direktivu u HTML dokumentu koristimo kao element, dakle, pišemo:

```
<oep-alternative-answer vm="question" enable-answer="vm.examLength">
</oep-alternative-answer>
```

`templateUrl` označava putanju do HTML dokumenta u kojem se nalazi markup direktive, `scope` definira elemente koje direktiva može primiti, te funkcija `link` definira ponašanje direktive ukoliko je to potrebno.

Bitno je primjetiti da se prilikom kreiranja direktive za njezino ime i parametre koristi camelCase, a prilikom njezinog korištenja umjesto velikog slova pišemo povlaku, te malo slovo. To je zato što HTML ne razlikuje velika i mala slova, dok ih JavaScript razlikuje. Također, dobro je svojim direktivama dati prefiks početnih slova aplikacije kako bi se izbjeglo podudaranje tih direktiva s direktivama drugih učitanih modula. Tako, na primjer, sve angular direktive imaju prefiks `ng`: `ngApp`, `ngController`, `ngShow`, ...

4. Komunikacija između AngularJS i RESTful aplikacija

Prije nego krenemo na komunikaciju Angular aplikacije s RESTful API-em, pogledajmo što je to REST:

4.1. REST

REST (*representational state transfer*) je arhitekturalni koncept za aplikacije čije komponente komuniciraju preko mreže. On ignorira detalje implementacije i sintakse protokola kako bi se fokusirao na ulogu komponenata, te pravila njihove interakcije. Interakcija se najčešće odvija preko HTTP protokola korištenjem istih HTTP glagola koje koriste i web preglednici prilikom dohvaćanja web stranica sa servera. Neka od pravila su:

- Odvajanje klijentskog i serverskog dijela aplikacije. Tako, na primjer, na klijentskoj strani nije bitno na koji način se odvija spremanje podataka, dok na serverskoj nije bitno korisničko sučelje. Zbog toga se ti dijelovi mogu implementirati neovisno jedan o drugome.
- Svaki request koji se s klijentske strane šalje serveru mora sadržavati sve podatke potrebne serveru, te se klijentski kontekst nigdje ne čuva između requestova.
- Kako klijentska strana može cachirati response dobiven od servera, svaki response se mora moći cachirati

Web servisi koji slijede pravila REST-a se zovu RESTful API. Ako je pri tome servis baziran na HTTP protokolu, definiran je sljedećim:

- bazni URI, na primjer http://www.online-ispiti.com/ispit
- media tip, najčešće je to JSON, ali može biti i neki drugi, na primjer, XML
- standardne HTTP metode kao što su GET, POST, PUT, DELETE, ...

Pri tome se HTTP metode koriste za sljedeće: GET za dohvaćanje, POST za kreiranje novih, PUT za izmjenu postojećih, te DELETE za brisanje podataka.

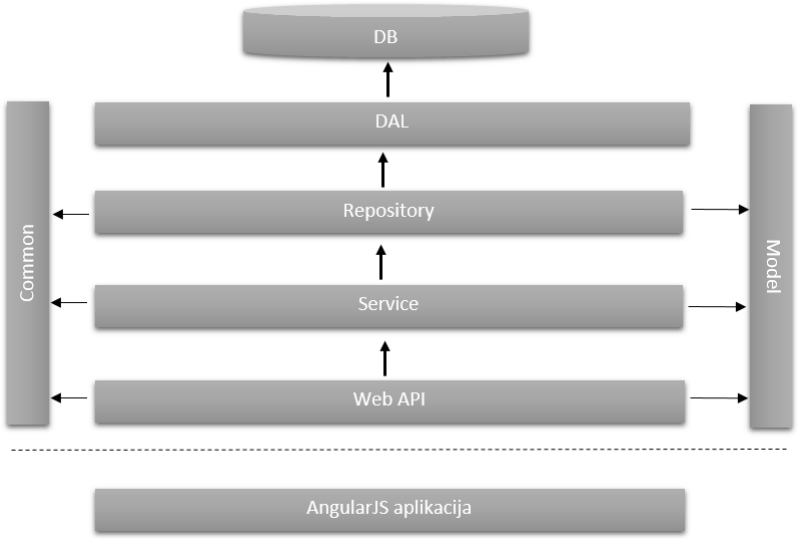
4.2. Projekt: simulator ispita

Projekt izrađen u sklopu ovog rada je zamišljen kao simulator ispita. Korisnik se može registrirati i prijaviti u sustav, te ispitati svoje znanje na ispitima koji se nalaze u sustavu. Kôd projekta se nalazi u javnom GitHub repozitoriju na adresi:
https://github.com/isimic413/OnlineExamPrep/tree/master

Za izradu projekta korištene su sljedeće glavne tehnologije: ASP.NET 4.5.1 i AngularJS v1.4.7. Detaljan popis tehnologija i third party komponenata možete vidjeti u potpoglavlju 4.2.2. Korištene tehnologije.

4.2.1. Arhitektura

Pri izradi arhitekture aplikacije korištena je Layered arhitektura. To znači da se aplikacija sastoji od komponenata (layera) koje međusobno komuniciraju. Način komunikacije ćemo objasniti preko skice:



Slika 3: Arhitektura aplikacije

Klijentska strana aplikacije - AngularJS aplikacija komunicira s Web API komponentom preko HTTP protokola. Web API tada od Service komponente traži podatke. Service, nadalje, od Repository komponente traži podatke koji su joj potrebni za konstruiranje podataka koje je od nje tražila Web API komponenta. Zatim, Repository komponenta, koristeći definicije objekata iz Data Access Layer (DAL) komponente dohvaća tražene podatke, vraća ih Service komponenti koja ih tada obrađuje i predaje Web API komponenti. Tada Web API komponenta preko HTTP responsea klijentskoj aplikaciji daje tražene podatke.

Svaka od komponenata ima svoju ulogu u aplikaciji:

- Data Access Layer - sadrži definicije klasa preko kojih kreiramo objekte koji se mogu mapirati u podatke koje je moguće unositi u bazu podataka (DB). Te klase ne sadrže definicije metoda, nego samo definicije propertya koje objekti mogu imati.

- Repository - sadrži metode za kreiranje, izmjene, dohvaćanje i brisanje podataka iz baze. Pri tome ne sadrži nikakvu business logiku. Na primjer, ako u bazu

želimo spremiti uvijek ili i zadatak i ponuđene odgovore za njega ili ništa od toga, ova komponenta nije toga svjesna. Za taj dio nam služi Service komponenta.

- Service - komponenta u kojoj se odvija business logika aplikacije. Ne može komunicirati s bazom podataka, nego to čini preko Repository komponente. Preko nje dohvaća podatke, te na temelju njih kreira podatke koje vraća na Web API komponentu.

- Web API - na temelju requesta, traži podatke od Service komponente, te ih šalje na klijentsku aplikaciju. Možemo ju shvatiti kao komponentu za prevođenje klijentskog requesta u podatke koje razumije Service komponenta. Ova komponenta je zadužena i za autorizaciju korisnika. Ona odlučuje o tome smije li korisnik vidjeti podatke koje je zatražio.

- AngularJS aplikacija - zadužena za prikaz podataka na ekranu, te interakciju s korisnikom. Ne mora biti svjesna business logike iz Service komponente. Mora moći poslati request te znati pročitati i obraditi serverski response tako da ga može prikazati na ekranu na korisniku razumljiv način.

Angular aplikacija također ima svoju arhitekturu. U glavnom folderu se nalaze index.html dokument, te folderi:

- app - sadržaj aplikacije, tj. kodovi koje smo sami pisali

- content - slike korištene u aplikaciji, te css dokumenti

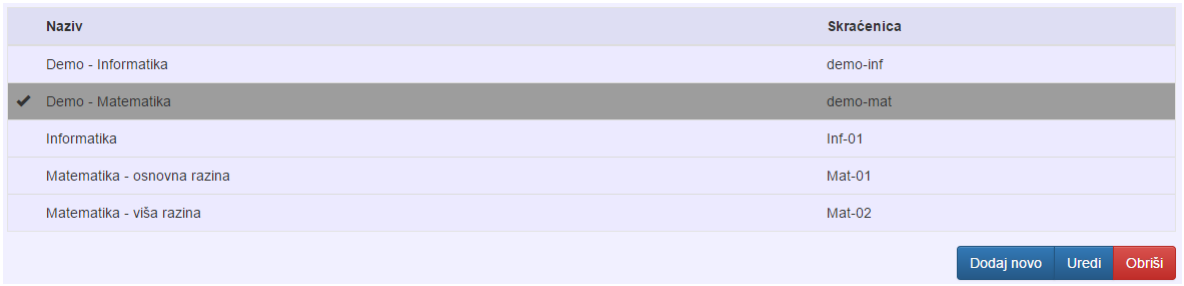
- fonts - korišteni fontovi

- scripts - JavaScript datoteke koje nismo sami pisali - biblioteke, third party komponente, te kôd samog frameworka

Nadalje, app folder je podjeljen na common, components te pages foldere, te sadrži i app.js datoteku. U ovom slučaju, to znači da se aplikacija sastoji od glavnog modula konfiguriranog u app.js datoteci, te od još tri modula, od kojih je svaki smješten u svoj folder. U app.js datoteci možemo odrediti koje interceptore aplikacija koristi, pišemo definicije ruta klijentske aplikacije, što se događa prilikom promjene rute, ...

Common modul sadrži servise koji se koriste u ostalim modulima. Na primjer, u ovom modulu se nalaze servisi koji nam služe da dohvaćanje podataka sa servera.

Components modul sadrži komponente koje koristimo na više mjesta u pages modulu. Tako se direktiva koju koristimo za prikaze stranice podataka nalazi u ovom modulu.



Slika 4: Prikaz stranice podataka u aplikaciji

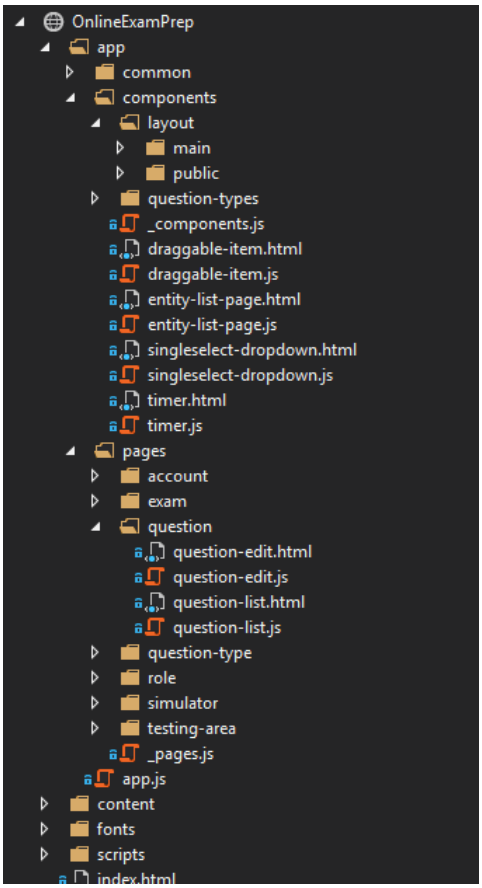
Pages modul sadrži *stranice* aplikacije. Treba napomenuti da ovdje stranica nije zapravo posebna stranica aplikacije, nego jedna direktiva koju koristimo za prikaz podataka na ekran. Iako zbog URL-a imamo dojam da se aplikacija sastoji od više stranica, zapravo se sastoji samo od jedne jer je ovaj projekt izrađen kao SPA (*Single Page Application*) - dakle, postoji samo jedna stranica, te pomoću Angular kôda odlučujemo što će korisnik vidjeti. Nadalje, korišten je HTML5 prikaz URL-ova zbog čega niti jedan URL u ovoj aplikaciji ne sadrži #. Takav prikaz definiramo unutar app.js datoteke na sljedeći način:

Primjer 4.1 (HTML5 prikaz URL-ova)

```
$locationProvider.html5Mode({
        enabled: true,
        requireBase: false
    });
```

Svaka *stranica* aplikacije definirana je svojom rutom, direktivom te html dokumentom. Pri tome se jedna direktiva (s pripadajućim html dokumentom) može koristiti za prikaz više od jedne stranice. Na primjer, stranice za unos novog ispita, te za uređivanje postojećeg koriste istu direktivu - `oepExamEdit`. Unutar direktive tada, prema prosljeđenim parametrima direktiva odlučuje što će prikazati na ekranu.

Stranice su grupirane u foldere tako da svaki folder označava jednu cjelinu koja ne ovisi o cjelini iz nekog drugog foldera. Logika za simulator ispita se nalazi u folderu simulator, logika za unos i uređivanje ispita, te prikaz stranice ispita se nalazi u folderu exam itd. Arhitektura Angular aplikacije je sljedeća:



Slika 5: Arhitektura Angular aplikacije

4.2.2. Korištene tehnologije

Alati:

- Microsoft Visual Studio 2013
- Microsoft SQL Server 2014

Server side:

- ASP.NET 4.5.1
- Transact-SQL
- Entity Framework 6.1.3
- AutoMapper
- Ninject 3.2.2.0
- Ninject Web Common 3.2.1.0

- Microsoft ASP.NET Web API 5.2.3
- Microsoft Owin 3.0.1

Client side:

- Angular 1.4.7
- Angular UI Bootstrap 0.14.3
- Angular UI Router 0.2.15
- Moment.js 2.10.6
- underscore.js 1.8.2
- MathJax.js 2.5

4.3. Komunikacija između AngularJS i RESTful aplikacija

Kao što smo napomenuli, Angular aplikacija s RESTful aplikacijom komunicira preko HTTP protokola. U ovom potpoglavlju ćemo pokazati primjere u kôdu, te primjere requestova. Međutim, pogledajmo prvo kako izgleda kreiranje HTTP requesta u Angularu.

Za slanje HTTP requesta u Angularu se koristi `$http` servis koji koristimo na sljedeći način: `$http(config)`. Pri tome `config` objekt preko svojih propertya sadrži opis requesta.

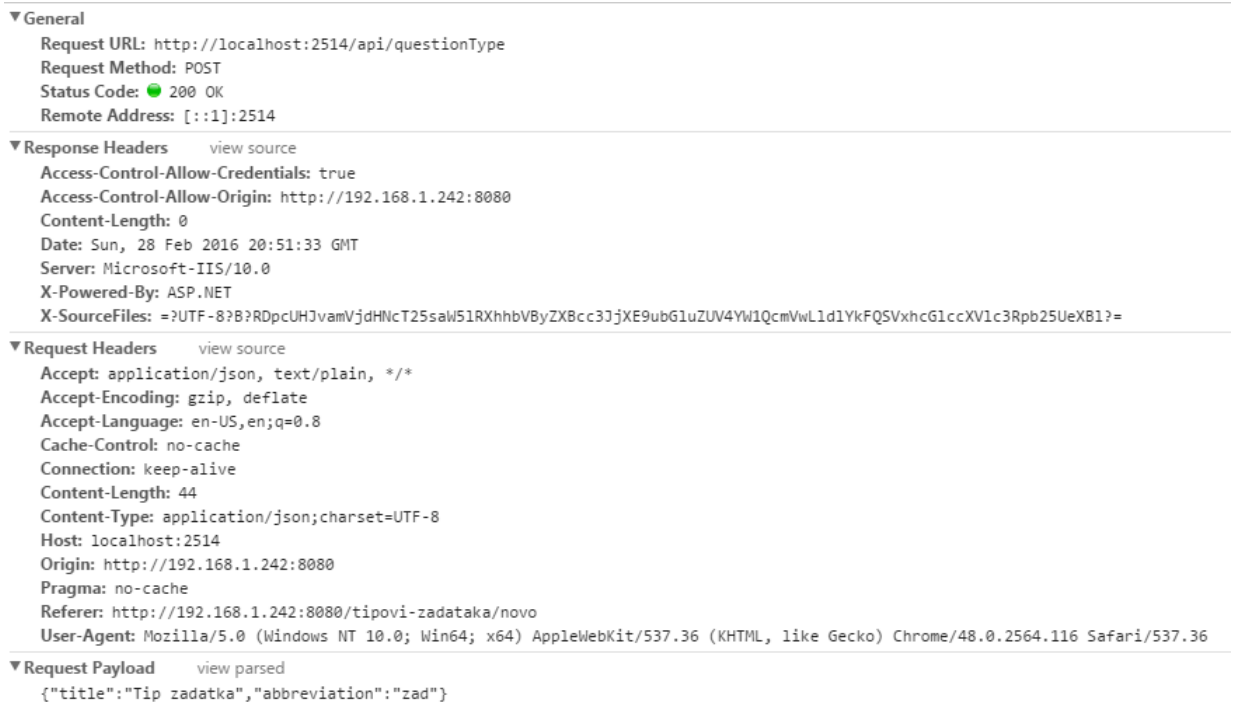
Primjer 4.2 (config objekt za slanje post requesta)

```
var config = {
    method: 'post',
    url: 'http://localhost:2514/api/question-type',
    data: {
        "title": "Tip zadatka",
        "abbreviation":"zad"
    },
    headers: {
        'content-type': 'application/json',
        'Authorization': 'Bearer dsrzn-yxasd54a65ercawereouma...'
    }
};
```

Ovako konfiguriran HTTP request na server šalje objekt koji želimo spremiti u bazu podataka kao novi entitet. Serverov odgovor (response) ima sljedeće propertye:

- data - tijelo responsea, sadrži podatke poslane sa servera

- status - HTTP status kôd. Status kodovi između 200 i 299 se smatraju kodovima uspješnog requesta.
- headers - metoda za dohvaćanje headera
- config - objekt koji je korišten za generiranje requesta
- statusText - HTTP status text responsea



Slika 6: Chrome developer tools - request i response headers post metode

Na gornjoj slici vidimo kako u Chrome developer toolsu možemo pročitati vrijednosti config objekta koji je korišten za kreiranje requesta, te headere i status responsea. Osim navedenog načina slanja requesta iz angular aplikacije, možemo koristiti i metode čiji config objekt ne sadrži url i method propertye. Sve te medode kao parametre primaju odvojeno string koji predstavlja url, te config objekt:

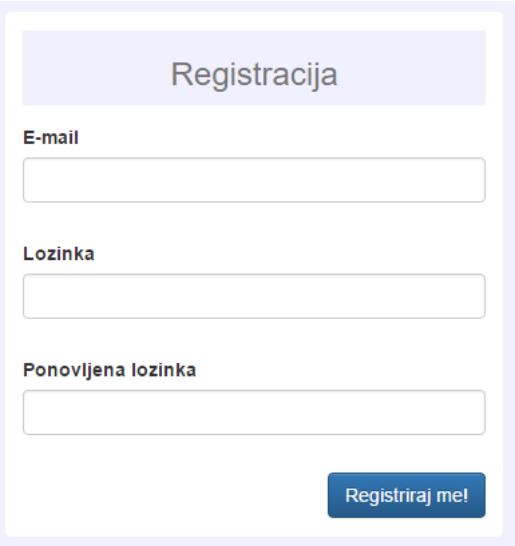
- `$http.get`
- `$http.post`
- `$http.put`
- `$http.delete`

4.3.1. Primjeri iz projekta

U ovom potpoglavlju ćemo proći kroz slanje POST, PUT, GET i DELETE requestova. Proći ćemo kroz aplikaciju, prikazati gdje se koji koristi, te kako su obrađeni na

klijentu.

Kako bi mogao koristiti aplikaciju, korisnik se prvo mora registrirati u sustav.



Slika 7: Registracija korisnika - unos podataka

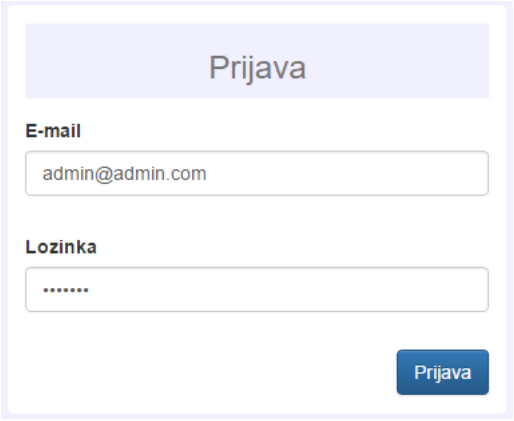
Klikom na gumb "Registriraj me!" se izvršava kôd iz AccountService.js datoteke koja na server šalje request za registraciju korisnika:

Primjer 4.3 (Registracija korisnika)

```
this.registerUser = function (userParams) {
    return $http.post(path + '/register', userParams);
};
```

U ovom primjeru se koristi alternativna metoda čiji parametri su url, te objekt koji sadrži korisničko ime, lozinku, te ponovljenu lozinku. Uspješan request rezultira kreiranjem novog korisnika u bazi podataka koji se sada može prijaviti u sustav.

Za prijavu korisnika koristimo standardnu formu u kojoj je potrebno unijeti korisničko ime i lozinku.



Slika 8: Prijava korisnika - unos podataka

Klikom na prijavu se izvršava sljedeći kôd:

Primjer 4.4 (Prijava korisnika)

```
this.login = function (loginParams) {
    return $http({
        method: 'post',
        url: Paths.token,
        header: {
            'Content-Type': 'application/x-www-form-urlencoded'
        },
        data: 'grant_type=password&username=' + loginParams.userName +
              '&password=' + loginParams.password
    });
};
```

Vidimo da za prijavu korisnika u sustav ne koristimo alternativnu metodu. Sada kreiramo config objekt koji sadrži naziv metode, url, headere, te podatke potrebne serveru kako bi mogao prijaviti korisnika. URL u gornjem kodu je skriven jer se koristi Angular konstanta koju smo nazvali Paths. Paths konstanta se nalazi u common modulu, te sadrži putanje do html dokumenata, te rute za pristup serverskim metodama. Paths.token ima vrijednost: `'http://localhost:2514/token'`
Prilikom uspješne prijave, koja se odvija preko POST metode, server nam šalje i objekt u kojem se nalazi token. U tokenu su spremljeni korisnikov id iz baze podataka, njegova rola, te njegove ovlasti. `access_token` property token objekta nam je dalje potreban za metode na serverskoj strani za koje je nužno da je korisnik prijavljen u sustav. Na primjer, kod traženja podataka potrebnih nakon prijave korisnika. Ti podaci su lookup tablice, tj. podaci koji se ne mijenjaju često, pa ih spremamo radi smanjenja broja slanja requestova na server. Naime, slanje requesta, tj. čekanje responsea može usporiti aplikaciju. Zbog toga je dobro slati što manje requestova, a to postižemo i spremanjem lookup tablica.

Primjer 4.5 (Token objekt)

```
{
    "access_token": "phmTQWprvj1WmEhuKYcj2Z1j_hBYp9j8shxytUgKSDlw...",
    "token_type": "bearer",
    "expires_in": 1209599,
    "userName": "admin@admin.com",
    "id": "04a00e86-93e6-434d-b6da-ff96a55d6d5b",
    ".issued": "Sun, 28 Feb 2016 22:37:50 GMT",
    ".expires": "Sun, 13 Mar 2016 22:37:50 GMT"
}
```

Kako ne bismo morali svaki puta pisati config objekt prilikom slanja requesta na server, kreiran je DataService koji se koristi u ostalim servisima te on zapravo šalje request.

Primjer 4.6 (DataService.js)

```
angular.module('onlineExamPrep.common')
    .service('DataService', function ($rootScope, $http, $q) {

        function sendRequest(config) {
            var deferred = $q.defer();
            $rootScope.loadingContent = true;

            $http(config).then(function (response) {
                $rootScope.error = null;
                deferred.resolve(response.data);
            }, function (data) {
                $rootScope.error = data;
                deferred.reject();
            });

            $rootScope.loadingContent = false;
            return deferred.promise;
        }

        this.get = function (path, params, options) {
            var config = {
                method: 'get',
                url: path,
                data: params
            };
            if (options && options.headers) {
                config.headers = options.headers;
            }
            return sendRequest(config);
        };

        this.post = function (path, params, options) {
            var config = {
                method: 'post',
                url: path,
                data: params
            };
            if (options && options.headers) {
                config.headers = options.headers;
```

```
            }
            return sendRequest(config);
        };

        this.put = function (path, params, options) {
            var config = {
                method: 'put',
                url: path,
                data: params
            };
            if (options && options.headers) {
                config.headers = options.headers;
            }
            return sendRequest(config);
        };

        this.delete = function (path, params, options) {
            var config = {
                method: 'delete',
                url: path,
                data: params
            };
            if (options && options.headers) {
                config.headers = options.headers;
            }
            return sendRequest(config);
        };
    });
```

U ovom servisu kreiramo config objekt za svaki tip metode, te šaljemo request i obrađujemo response koji dobijemo od servera. Na primjer, kako ne bismo u servisima u svakoj metodi morali posebno pisati kada želimo da se na sučelju vidi da se trenutno odvija request, to smo učinili u sendRequest metodi postavljajući vrijednost loadingContent propertya sa root scope-a aplikacije. Nadalje, u sendRequest metodi vidimo jedan od načina na koji možemo dalje procesuirati promise koji nam vrati `$http` metoda. Ako je request uspješan, izvrši se prva, a u suprotnom druga funkcija. Također, dio kôda koji je zadužen za manifestaciju neuspješnog requesta jer odrađen u ovom servisu. To smo učinili zato što je manifestacija greške uvijek ista, te smo izbjegli pisanje tog dijela kôda u svim direktivama koje pozivaju metode servisa.

Sada metoda koja je zadužena za dohvaćanje lookup podataka koristi get metodu data servisa, te ne koristi izravno `$http` servis:

Primjer 4.7 (Dohvaćanje lookup podataka)

```
this.getApplicationData = function () {
return DataService.get(path + '/application-data');
};
```

Primjetimo kako nismo naveli niti Authorization header. Budući da je metoda na serveru zaštićena te je ovim podacima moguće pristupiti samo ako je korisnik prijavljen u sustav, potrebno je poslati i token u authorization headeru. U tu svrhu smo iskoristili jednostavan interceptor. Interceptori služe za presretanje requestova i responsea. U našem slučaju, interceptor je korišten samo za presretanje requesta. U njemu dodajemo authorization header ako token postoji:

Primjer 4.8 (Interceptor)

```
angular.module('onlineExamPrep.common')
    .service('Interceptor', function ($injector) {
        'use strict';

        return {
            request: function (config) {
                var tokenService = $injector.get('TokenService');
                var token = tokenService.getToken();
                if (token) {
                    if (!config.headers) {
                        config.headers = {};
                    }
                    config.headers.Authorization = 'Bearer ' +
                                                token.access_token;
                }
                return config;
            }
        };
    });
```

Dakle, prilikom svakog requesta, prije nego request ode na server, interceptor provjeri postoji li token. Ako postoji, utisne token u authorization header te pušta request dalje.

Primjer 4.9 (Rad s pristiglim podacima)

```
UserService.getApplicationData().then(function (data) {
    var principalData = {
        role: data.role
    };
```

```
    var lookupData = {
        testingArea: data.testingAreas,
        questionType: data.questionTypes
    };
    Principal.setCurrent(principalData);
    Lookups.setLookups(lookupData);

    $state.go('main.home');
});
```

Kao što vidimo, pristigle podatke koristimo kao i svaki drugi JavaScript objekt. Možemo ih slati drugim metodama, kreirati nove objekte koristeći njihove podatke, ...

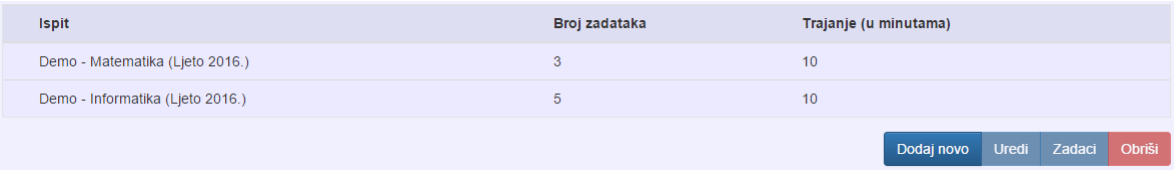
Prisjetimo se: za dohvaćanje podataka koristimo GET metodu. Međutim, GET metoda ne može slati objekte u requestu. Što ako želimo dohvatiti samo jednu stranicu podataka? Recimo da želimo dohvatiti samo prvih 20 podataka - tj. prvu stranicu veličine 20. Možda i prilikom dohvaćanja stranice želimo odrediti i po kojem stupcu podaci trebaju biti sortirani. Imajući na umu da GET metoda ne može slati takve podatke ako ih ne upiše u URL, alternativu možemo pronaći u POST metodi:

Primjer 4.10 (POST metoda za dohvaćanje podataka - dohvaćanje ispita)

```
this.getExamCollection = function (pagingParams) {
    return DataService.post(path + '/page', pagingParams);
};
```

Primjer 4.11 (Dohvaćanje stranice korištenjem POST metode: pristigli podaci)

```
[
  {
    "id": "fb5fc072-172c-4bc3-90df-7a78f2eaba5b",
    "durationInMinutes": 10,
    "numberOfQuestions": 3,
    "title": "Demo - Matematika (Ljeto 2016.)"
  },
  {
    "id": "b21f2b02-be73-490e-a42a-36775b4d30e7",
    "durationInMinutes": 10,
    "numberOfQuestions": 5,
    "title": "Demo - Informatika (Ljeto 2016.)"
  }
]
```



Slika 9: Manifestacija pristiglih podataka na ekranu: ispiti

Korištenje POST metode u ovom kontekstu možemo shvatiti kao zahtjev za kreiranjem nove stranice podataka na temelju poslanih podataka. Ovakvo korištenje POST metode u projektu kojeg koristimo za primjer se pojavljuje na svakom mjestu gdje dohvaćamo stranicu podataka.

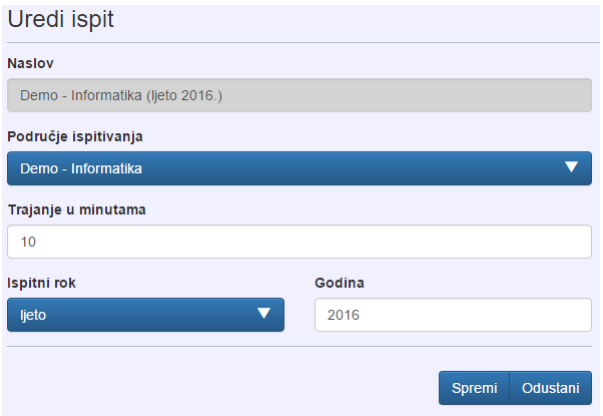
Kada želimo dohvatiti jedan entitet, na primjer jedan ispit, dohvaćamo ga korištenjem GET metode:

Primjer 4.12 (Dohvaćanje jednog ispita)

```
this.getExam = function (examId) {
    return DataService.get(path + '/' + examId, examId);
};
```

Primjer 4.13 (Dohvaćanje jednog ispita: pristigli podaci)

```
{
  "term": 1,
  "year": 2016,
  "durationInMinutes": 10,
  "testingAreaId": "5f96dd15-4caa-4311-881f-bff83227ed0f",
  "id": "fb5fc072-172c-4bc3-90df-7a78f2eaba5b"
}
```



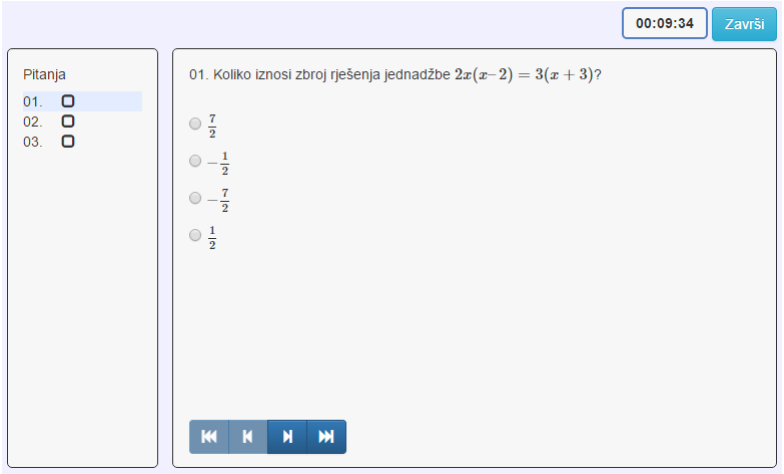
Slika 10: Manifestacija pristiglih podataka na ekranu: ispit

Koju metodu trebamo koristiti ako želimo dohvatiti sve zadatke jednog ispita? Prvo treba odrediti parametre koji su nam potrebni za dohvaćanje takvih podataka. Želimo ih dohvatiti sve, dakle nisu nam potrebni parametri za redni i veličini stranice. Znamo

i da svi zadaci moraju biti s istog ispita, dakle - dovoljan nam je Id tog ispita. To znači da, iako dohvaćamo veći broj podataka, trebamo koristiti GET metodu jer je bitno koliko i kakve parametre šaljemo, a ne koliko podataka očekujemo.

Primjer 4.14 (Dohvaćanje svih zadataka određenog ispita)

```
this.fullExamQuestions = function (examId) {
    return DataService.get(path + '/questions/' + examId, examId);
};
```



Slika 11: Manifestacija pristiglih podataka na ekranu: ispit sa zadacima

Vidimo da se sada na ekranu ne prikazuju svi podaci u isto vrijeme. Kako se ovi podaci koriste za simulator ispita, prikazujemo jedan po jedan zadatak, te sa strane vidimo koliko zadataka ukupno ima u ispitu.

Osim POST i GET metoda, koristimo i PUT i DELETE metode. DELETE metoda, kao i GET metoda, ima samo jedan parametar koji pišemo u URL. Taj parametar mora biti dovoljan da server može prepoznati točno o kojem se entitetu radi. PUT metoda također u URL-u ima Id entiteta, ali ima i objekt s parametrima kao što to ima POST metoda. Ti parametri, tj. vrijednosti propertya objekta će u bazi zamijeniti postojeće:

Primjer 4.15 (PUT I DELETE metode)

```
DELETE: DataService.delete(path + '/' + examId, examId)
PUT: DataService.put(path + '/' + exam.id, exam)
```

Do sada smo vidjeli podatke samo uspješnih requestova. Pogledajmo kako izgleda JSON objekt koji dobijemo od servera ako request nije uspješan. Na primjer, ako pokušamo obrisati entitet koji ne postoji u bazi podataka.

Primjer 4.16 (JSON response neuspješnog requesta - entitet nije pronađen)

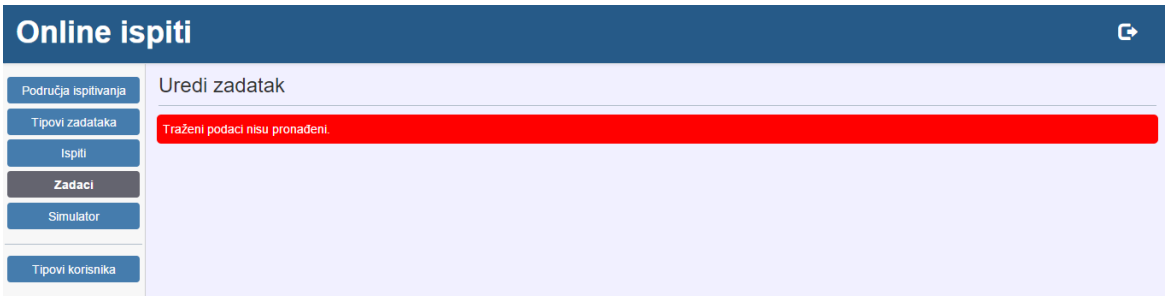
```
{
  "message": "An error has occurred.",
  "exceptionMessage": "Entity with specified id not found.",
```

```
  "exceptionType": "System.Collections.Generic.KeyNotFoundException",
  "stackTrace": "   at OnlineExamPrep.Repository.Repository.
                <DeleteEntityAsync>d__c‘1.MoveNext() in ..."
}
```

Primjer 4.17 (JSON response neuspješnog requesta - metoda nije pronađena)

```
{
  "message": "No HTTP resource was found that matches the request URI
              ’http://localhost:2514/api/question-type/40006ae0-686f’.",
  "messageDetail": "No type was found that matches the controller
                    named ’question-type’."
}
```

Ovakvi podaci osobi koja nije pisala aplikaciju na serveru ne znače puno. Umjesto ispisa ovih podataka na ekran, treba pronaći odgovarajući način da se korisniku prikaže da njegov zahtjev nije prošao. Na primjer, crveno označeno područje koje će ga asocirati na pogrešku, te tekst koji opisuje grešku koja se dogodila. Sljedeći primjer prikazuje jednu od mogućnosti prikaza greške koja se dogodi prilikom dohvaćanja zadatka koji ne postoji u bazi podataka:



Slika 12: Manifestacija greške na ekranu

5. Zaključak

Od prvih web stranica do danas se pristup u njihovom korištenju uvelike promijenio. Korisnici više ne žele samo čitati sadržaj kao tekst ispisan u knjigama, nego žele imati utjecaja na njega. Zbog toga je kreiran JavaScript. Iako mu je prva svrha bila početnicima omogućiti lako dodavanje animacija i interaktivnosti na svoje web stranice, danas se koristi za izradu vrlo kompleksnih aplikacija. Kako su aplikacije pisane u JavaScriptu postajale sve kompleksnije, javila se potreba za dobrim strukturiranjem kôda kojeg bi bilo lakše održavati. U tu svrhu su osmišljeni brojni frameworci. Jedan od njih je i AngularJS.

AngularJS framework nam pomaže dobro strukturirati kôd aplikacije te time olakšava njegovo testiranje i održavanje. Tako nam pisanje svojih Angular servisa koje koristimo kroz aplikaciju smanjuje količinu kôda jer možemo na jednom mjestu osigurati da će se dogoditi određena akcija - npr. prikaz ili skrivanje manifestacije čekanja odgovora sa servera.

Osim toga, bitno je i definirati način komunikacije AngularJS aplikacije sa serverskom aplikacijom. U tome nam može pomoći arhitekturalni koncept REST. Jedna od njegovih prednosti je upravo ta da ne diktira tehnologije koje moramo koristiti pri izradi aplikacije. Imajući to na umu, možemo kreirati više klijentskih aplikacija koje će komunicirati s istom serverskom aplikacijom a koje neće biti pisane istom tehnoologijom.

Literatura

[1] D. Flanagan, *JavaScript - The Definitive Guide*, O'Reilly Media, Inc., Sebastopol, 2011.

[2] G. Mirzaei, M Wadood Majid, *Mastering AngularJS for .NET Developers*, Packt Publishing Ltd., Birningham, 2015

[3] AngularJS, *AngularJS API Docs*,
URL: https://docs.angularjs.org/api

[4] Brendan Eich, *Blog: New JavaScript Engine Module Owner*
URL: https://brendaneich.com/2011/06/new-javascript-engine-module-owner/

[5] Built with Ember
URL: http://builtwithember.io/

[6] GitHub: Backbone, *Projects and Companies using Backbone*
URL: https://github.com/jashkenas/backbone/wiki/Projects-and-Companies-using-Backbone

[7] Gregg Pollack, *Shaping up with Angular.js, Flatlander's Gem Store - Ramp Up*
URL: https://www.codeschool.com/courses/shaping-up-with-angular-js/videos

[8] InfoWorld, *JavaScript creator ponders past, future by Paul Krill*
URL: http://www.infoworld.com/article/2653798/application-development/javascript-creator-ponders-past–future.html

[9] John Papa, *Web Sites Using Knockout.js*
URL: http://www.johnpapa.net/web-sites-using-knockoutjs/

[10] Made with Angular
URL: https://www.madewithangular.com/#/

[11] Mozilla Develper Netvork, *Back to the Server: Server-Side JavaScript On The Rise*
URL: https://developer.mozilla.org/en-US/docs/Archive/Web/Server-Side_JavaScript/Walkthrough

[12] Mozilla Developer Network, *Introduction to Object-Oriented JavaScript*
URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

[13] Readers Park, *Top JavaScript Frameworks (Client & Server Side)*
URL: http://www.readerspark.com/top-javascript-frameworks-client-server-side/

[14] TutorialsPoint, *RESTful Web Services Tutorial*,
URL: "http://www.tutorialspoint.com/restful/"

[15] Wikipedia, *Comparison of JavaScript frameworks*
URL: http://en.wikipedia.org/wiki/Comparison_of_JavaScript_frameworks

6. Sažetak

U ovom radu pokazat ćemo kako povezati klijentsku stranu aplikacije sa serverskom kada one međusobno komuniciraju preko mreže. Pri tome će klijentska strana biti izrađena u JavaScript frameworku AngularJS. Najprije ćemo proći kroz povijest JavaScripta, te ćemo objasniti razlike između klijentskog i serverskog JavaScripta. Za potrebe razumijevanja AngularJS-a ćemo objasniti što je framework, te kako raditi s objektima u JavaScriptu. Također ćemo vidjeti i od kojih elemenata se sastoji AngularJS aplikacija.

U posljednjem poglavlju ćemo objasniti koji preduvjeti su potrebni da bi aplikacija bila RESTful aplikacija. Prikazat ćemo i primjere kôda aplikacije izrađene u sklopu ovog rada kako bismo razumjeli kako AngularJS aplikacija komunicira sa serverskom aplikacijom.

7. Summary

Here we will show how to connect two sides of an application - client side and server side when they communicate through network protocol. Client side application will be built with JavaScript framework AngularJS. First, we will go through JavaScript history and we will explain what are the differences between client side JavaScript and server side JavaScript. For better understanding of AngularJS, we will explain what is framework and how to use objects in JavaScript. We will also explain what elements make an AngularJS application.

In the last chapter, we will explain what it takes for an application to be RESTful. In order to understand the communication between AngularJS application and server side application we will show code examples taken from the application that we built for that purpose.

8. Životopis

Ivana Šimić rođena je 14.09.1990. u Osijeku. Niže razrede osnovne škole pohađala je u Osnovnoj školi Sv. Ane u Osijeku, dok je više razrede pohađala u Osnovnoj školi Tenja u Tenji. Nakon osnovne škole upisala je III. gimnaziju u Osijeku. Tijekom osnovne i srednje škole sudjelovala je na općinskim i županijskim natjecanjima iz matematike, informatike te hrvatskog jezika. 2009. godine upisala je preddiplomski studij matematike na Odjelu za matematiku Sveučilišta J. J. Strossmayera u Osijeku, te se 2012. prebacila na sveučilišni nastavnički studij matematike i informatike. Tijekom studiranja položila je MTA (*Microsoft Technology Associate*) certifikate: *Software development fundamentals* - programski jezik C# (2012.) i *Database Fundamentals* (2015.). Također, sudjelovala je na timskim natjecanjima iz programiranja IEEEXtreme 6.0 i IEEEXtreme 7.0 na kojima se tim plasirao u 25% najboljih timova, te stekla iskustvo rada software developera u tvrtci Mono u Osijeku.