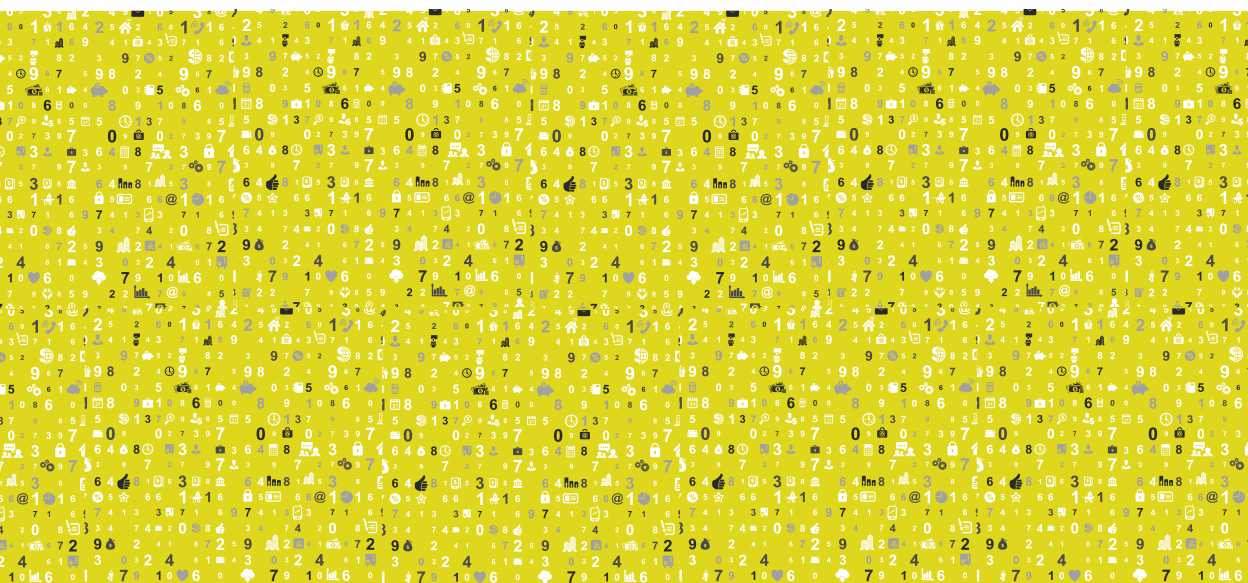




RAZVOJ APLIKACIJA ZASNOVANIH NA VEB SERVISIMA

MARKO MARKOVIĆ



UNIVERZITET SINGIDUNUM
Fakultet zdravstvenih i poslovnih studija

dr Marko Marković

**RAZVOJ APLIKACIJA ZASNOVANIH
NA VEB SERVISIMA**

Prvo izdanje

Valjevo, 2023.

RAZVOJ APLIKACIJA ZASNOVANIH NA VEB SERVISIMA

Autor:

dr Marko Marković

Recenzenti:

dr Boško Nikolić

dr Biljana Tešić

Izdavač:

Univerzitet Singidunum, Fakultet zdravstvenih i poslovnih studija
Železnička 5, Valjevo
fzp.singidunum.ac.rs

Za izdavača:

dr Olivera Nikolić

Tehnička priprema:

dr Marko Marković

Dizajn korica:

dr Katarina Marković

Godina izdanja:

2023.

Tiraž:

500 primeraka

ISBN:

978-86-81206-04-1

Sadržaj

1. RAD SA NIZOVIMA	1
1.1. Regularni nizovi	1
1.2. Asocijativni nizovi	5
1.3. Funkcije za rad sa nizovima	7
1.3.1. Ispis sadržaja niza	7
1.3.2. Određivanje broja elemenata niza	8
1.3.3. Promena redosleda elemenata niza	9
1.3.4. Funkcije za kretanje kroz niz bez korišćenja petlje.....	10
1.3.5. Ostale korisne funkcije za rad sa nizovima	11
1.4. Dvodimenzionalni nizovi.....	12
2. KLASSE I OBJEKTI	15
2.1. Rad sa klasama i objektima	16
2.1.1. Vidljivost polja u klasama.....	18
2.2. Funkcije.....	20
2.2.1. Konstruktori.....	22
2.2.2. Funkcije get i set	24
2.2.3. Magične funkcije get i set	25
2.2.4. Magična funkcija __toString.....	27
2.3. Ključna reč static.....	29
2.3.1. Statičke promenljive	29
2.3.2. Statičke funkcije	30
2.4. Nasleđivanje.....	32
2.4.1. Proces nasleđivanja klasa	32
2.4.2. Vidljivost elemenata prilikom nasleđivanja	34
2.4.3. Ključna reč final.....	36

2.4.4. Redefinisanje funkcija	37
2.5. Polimorfizam.....	38
2.5.1. Interfejsi	39
2.5.2. Apstraktne klase	40
2.5.3. Interfejs ili apstraktna klasa	42
2.5.4. Princip rada polimorfizma.....	42
3. JSON	45
3.1. JSON objekti i njihovo dekodiranje	46
3.2. JSON kolekcija objekata	47
3.3. Pravljenje niza objekata u okviru postojećeg objekta	50
3.4. Kodiranje podataka u JSON format	57
3.5. Prihvatanje i dekodiranje JSON podataka pomoću JS-a.....	59
4. REST API	63
4.1. Pravila imenovanja REST resursa.....	66
4.1.1. Korišćenje imenica za predstavljanje resursa	66
4.1.2. Korišćenje množine za predstavljanje resursa.....	67
4.1.3. Korišćenje kose crte (/) za definisanje hijerarhije	67
4.1.4. Korišćenje srednje crte (-) za poboljšanje čitljivosti url-ova.....	68
4.1.5. Korišćenje malih i velikih slova.....	68
4.1.6. Korišćenje ekstenzija u adresi resursa	68
4.1.7. Korišćenje CRUD funkcija u nazivu resursa	69
4.2. Ograničenja REST arhitekture.....	69
5. RAZVOJ REST APLIKACIJE.....	73
5.1. Struktura aplikacije	73
5.1.1. Važnost strane index.php	74
5.2. Pripremanje baze podataka	75
5.2.1. Pravljenje tabele “radnici”	75

5.2.2. Konekcija ka bazi podataka	76
5.3. Pravljenje veb servisa	78
5.3.1. Definisanje REST pravila	78
5.3.2. Korišćenje .htaccess dokumenta	79
5.3.3. Osnovna struktura veb servisa	81
5.3.4. Učitavanje svih vrednosti	83
5.3.5. Učitavanje određene vrednosti	84
5.3.6. Unos nove vrednosti	84
5.3.7. Izmena postojećih vrednosti	86
5.3.8. Brisanje postojećih vrednosti	87
5.4. Testiranje aplikacije	88
5.5. Frontend aplikacije	92
5.5.1. Pristup elementima html strane	94
5.5.2. Objekat XMLHttpRequest	97
5.5.3. Sadržaj anonimne funkcije za događaj load	99
5.5.4. Definisanje izgleda frontenda	102
6. LARAVEL, PRIPREMANJE OKRUŽENJA.....	105
6.1. Docker Desktop	106
6.2. Windows Subsystem for Linux	106
6.3. Povezivanje Docker-a i WSL-a.....	108
6.4. Microsoft Visual Studio Code	110
6.5. Pokretanje terminala	111
6.6. Laravel Sail, instalacija	114
6.7. Laravel Sail, pokretanje	117
6.8. Konfigurisanje aliasa za Sail	120
6.9. Pristup strukturi direktorijuma	121
7. LARAVEL, OSNOVE KORIŠĆENJA.....	125
7.1. MVC	125

7.2. Struktura Laravel aplikacije	127
7.3. Rutiranje u aplikaciji.....	128
7.4. Blade.....	130
7.5. Blade i šabloni strana	135
7.6. PHP Artisan	140
7.7. Migracije.....	142
7.7.1. Pravljenje nove migracije	148
7.8. Laravel modeli	149
7.9. Laravel kontroleri.....	151
8. LARAVEL, PRAKTIČAN PROJEKAT	153
8.1. Pravljenje početne strane	153
8.2. Pravljenje modela, kontrolera i migracija	155
8.3. Povezivanje početne strane sa kontrolerom	158
8.4. Manuelno dodavanje novog zapisa u bazu podataka	162
8.5. Prikazivanje pojedinačnih zapisa iz baze podataka	167
8.6. Dodavanje novog zapisa u bazu podataka korišćenjem formi	171
8.7. Dodavanje linkova u meni	176
8.8. Uklanjanje radnika iz baze podataka.....	178
8.9. Definisanje validacija i prikaz grešaka	181
8.10. Izmena podataka u bazi podataka	184
Dodatak A	193
Popis slika.....	199
Literatura	2043

Predgovor

Udžbenik je nastao iz potrebe da se objasne tehničke osnove rada i načini korišćenja veb servisa, a namenjen je studentima koji su već položili uvodne predmete koji se bave razvojem softvera, bazama podataka i računarskim sistemima.

Svako poglavlje obrađuje po jednu temu vezanu za razvoj veb servisa. Cilj je da se napredovanjem kroz poglavlja razumeju principi rada veb servisa i tehnike koje omogućavaju njihovo funkcionisanje, kao i da se ta znanja primene na pravljenju praktične *REST* aplikacije, a potom i u okviru platforme *Laravel*. *Laravel* je izabran kao platforma koja integriše sve teme o kojima se govori u ovom udžbeniku, ali i olakšava razvoj tako što sakriva veliki deo realizacije veb servisa – ovaj udžbenik je namenjen onima koji žele da detaljno razumeju kako veb servisi funkcionišu, čak i onda kada se koriste alati (poput *Laravel-a*) koji sakrivaju veliki deo njihove realizacije.

Ovaj udžbenik ne predstavlja iscrpni vodič za učenje *PHP-a* ili platforme *Laravel*, već je njegova namena da predstavi neke od najvažnijih tehnika rada koje se koriste u praktičnom razvoju veb servisa. Zbog toga, autor se nije usmerio na predstavljanje mnoštva funkcija ili objašnjavanja osnovnih koncepata programiranja i baza podataka. Očekuje se da će čitalac već imati neka (makar i početnička) znanja, kao i da će biti spreman da uloži vreme da dodatno samostalno istraži dokumentaciju i ostale izvore za učenje.

Valjevo,
01.06.2023. godine

1. RAD SA NIZOVIMA

Nizovi predstavljaju jednu od najvažnijih i verovatno najpoznatijih struktura podataka u računarskom svetu. Koriste se u raznim praktičnim primenama i izuzetno su korisni zbog jednostavnosti sa kojom se vrši pristupanje i čuvanje podataka. Počivaju na jednostavnoj ideji da je moguće na jednom mestu (pod jednim imenom promenljive) sačuvati više od jedne vrednosti istog tipa podataka.

Nizovi se često koriste kao posrednik za prenos podataka – na primer, često ćemo podatke koje učitavamo iz baze podataka dobiti u formi niza. U nekim situacijama, podaci će stizati preko mreže u formatu kao što je *JSON* – da bismo im pristupili, u većini slučajeva je najjednostavnije da podatke konvertujemo u niz, a potom izvršimo neku operaciju nad njima.

U programskom jeziku *PHP* postoje dve vrste nizova – regularni i asocijativni – o kojima će više reči biti u narednim sekcijama.

1.1. Regularni nizovi

Regularni nizovi su veoma slični nizovima u bilo kom drugom programskom jeziku. Karakteriše ih to što se pozicija svakog elementa u nizu označava brojevima, a prvi element u nizu uvek

dobija poziciju 0. Kako bismo napravili novi niz, koristi se naredba *array()*.

```
$voce = array( "jabuka", "banana", "kruska" );  
  
$voce[] = "breskva";  
  
echo $voce[0] . " - " . $voce[1] . " - " . $voce[2] . "<br>;
```

Elementi koji su prosleđeni naredbi *array()* čine početne vrednosti niza. Čak je i naknadno dodavanje elemenata jednostavna operacija. Nije potrebno voditi računa o poziciji poslednjeg elementa u nizu, već je samo potrebno navesti prazne uglaste zagrade i potom vrednost koju unosimo u niz. Alternativni način dodavanja novog elementa je korišćenje funkcije *array_push(\$naziv_niza, "nova vrednost")*. Ovoj funkciji se prosleđuju dva parametra: naziv niza u koji želimo da dodamo novi element i vrednost koju želimo da dodamo.

Kao što se u primeru iznad vidi, moguće je i jednostavno ispisati elemente pomoću *echo* naredbe tako što ćemo svakom pojedinačnom elementu pristupiti pomoću naziva niza i broja elementa određenog elementa koji se prosleđuje u zagradi. Naravno, u situacijama kada imamo više od samo nekoliko elemenata niza (a takve situacije imamo skoro uvek), podacima ćemo pristupati korišćenjem petlje. Za rad sa nizovima, često je najlakše koristiti petlju *foreach*.

```
foreach( $voce as $vocka ) {  
    echo $vocka . "<br>";  
}
```

Sve što je potrebno da uradimo jeste da u petlji navedemo ime niza i da iza naredbe *as* po slobnom izboru navedemo naziv nove promenljive. Ova promenljiva će u svakoj iteraciji petlje sadržati po jednu vrednost niza, a brojanje će početi od elementa na poziciji 0.

Naravno, može se koristiti i bilo koja druga petlja kao što su *for* ili *while*, ali *foreach* dosta pojednostavljuje posao time što nije potrebno brinuti o brojaču i njegovom uvećavanju prema veličini niza.

Naredni primer predstavlja malo složeniju vežbu korišćenja nizova i petlje za ispis sadržaja u okviru *HTML* tabele.

```
<?php

    $voce = array( "jabuka", "banana", "kruska", "breskva" );

    echo "<table>";
    echo "<tr> <th>Redni broj</th> <th>Voće</th> </tr>";

    foreach( $voce as $kljuc=>$vocka ) {

        echo "<tr>";
            echo "<td>";
                echo $kljuc;
            echo "</td>";

            echo "<td>";
                echo $vocka;
            echo "</td>";
        echo "</tr>";
    }

    echo "</table>";

?>
```

U primeru iznad imamo blago modifikovanu *foreach* petlju – kako bismo dobili informaciju o trenutnoj poziciji elementa u nizu, dodata je promenljiva *\$kljuc*. Ona će automatski biti popunjena rednim brojem elementa, a nešto kasnije, korišćemo je i prilikom rada sa asocijativnim nizovima. Naravno, mogli biste i samostalno da definišete odreženi brojač (na primer, *\$i*), ali u tom slučaju, briga o njegovom uvećavanju u svakoj iteraciji petlja postaje isključivo Vaša.

Iako se radi o jednsotavnom primeru, već možemo da vidimo koliko je komplikovano na istom mestu mešati *PHP* i *HTML* naredbe. Trenutno se u *echo* naredbama vrši ispis *HTML* naredbi. Kako veb strane budu postajale kompleksnije, tako će i *echo* naredbe sadržati još komplikovanije ispise. Zbog toga, trebalo bi da težimo što boljem razdvajanju programskog koda od oznaka koje se tiču prikaza. O tome će više reči biti u drugom delu udžbenika, a posebno kada započnemo korišćenje *Laravel* platforme koja nudi izuzetno dobre mehanizme za obavljanje ovakvog razdvajanja.

Kako bismo poboljšali izgled generisane tabele, možemo dodati i naredni *CSS* odeljak.

```
<style>
    table {
        border: 1px solid black;
        text-align: center;
    }
    th {
        background-color: lightgreen;
    }
    td {
        padding: 5px;
    }
    tr:nth-child(odd) {
        background-color: lightblue;
    }
</style>
```

Pored jednostavnih *CSS* formatiranja tabele i ćelija, dodata je i konstrukcija *tr:nth-child(odd)*. Ova naredba proverava svaki red u tabeli i potom podešava svetlo plavu pozadinsku boju samo neparnim redovima. Ukoliko želimo da pristupimo samo parnim redovima, možemo koristiti *tr:nth-child(even)*.

Nakon pokretanja primera, na ekranu bi trebala da se prikaže naredna tabela (slika 1).

Redni broj	Voće
0	jabuka
1	banana
2	kruska
3	breskva

Slika 1: Rezultat ispisa sadržaja niza u veb pretraživaču.

Samostalni rad

Kao modifikaciju ovog primera, probajte da isključite naredbu *tr:nth-child(odd)* i da bojenje svakog drugog reda uradite tako što ćete u direktno u *foreach* petlji proveriti da li je red neparan i ako jeste, primeniti naredbu za promenu boje. Rezultat treba da izgleda potpuno isto kao u primeru iznad.

Savet: možete koristiti operator % za određivanje da li je broj paran ili neparan.

1.2. Asocijativni nizovi

U određenim situacijama, rad sa regularnim nizovima nije dovoljno intuitivan iz razloga što nam nekada neće biti najjasnije koja se vrednost krije iza određenog broja elementa. U situacijama kao u prethodnom primeru nam nije izrazito važno kojim redosledom i na kojim pozicijama se nalaze određene voćke. Možemo uzeti za primer da je vrednost *breskva* trenutno na poziciji 3. Čak i da smo je smestili na neku drugu poziciju (npr. 2, 5 ili 10) naš primer bi opet radio na gotovo isti način, uz činjenicu da bi se samo određeni elementi (poput *breskve*) nalazili na drugim pozicijama.

U situacijama kada nam je važno da u nizu lako nađemo određeni element, čak i ukoliko dođe do promena njihovih pozicija (a to se može desiti), kao rešenje se nameću *asocijativni* nizovi. Oni

podrazumevaju da elementima u nizu ne pristupamo pomoću određenog indeksa, već korišćenjem određenog imena koje sami možemo da definišemo.

```
$magacin = [  
    "jabuke" => 100,  
    "banane" => 34,  
    "kruske" => 60,  
    "breskve" => 60  
];
```

Promenljiva čuva vezu ka nizu koji se sastoji od parova ključ-vrednost. Ključ se uvek definiše pod jednostrukim ili dvostrukim navodnicima, a kod same vrednosti to zavisi od tipa podataka. U našem slučaju čuvamo broj kilograma svakog voća u magacinu i zbog toga ne koristimo navodnike oko brojeva. Da su vrednosti bile tekstualnog tipa, onda bi navodnici bili obavezni.

Takođe, asocijativni niz je moguće definisati i kao naredbu u jednom redu. Krajnji rezultat je potpuno isti kao u prethodnom primeru.

```
$magacin = [ "jabuke" => 100, "banane" => 34, "kruske" => 60,  
"breskve" => 60 ];
```

Dodavanje novog elementa je takođe jednostavno. Potrebno je samo definisati novi ključ i novu vrednost.

```
$magacin["kajsije"] = 25;
```

Promena određene vrednosti se sprovodi na sličan način.

```
$magacin["jabuke"] = 80;
```

Ukoliko želimo da pristupimo elementima pomoću petlje, možemo uraditi nešto slično kao u narednom primeru.

```
foreach( $magacin as $kljuc=>$vrednost ) {
```

```
} echo "$kljuc -> $vrednost <br>";
```

Sličnu naredbu smo koristili i za obične nizove, jedina razlika je u tome što sada promenljiva \$ključ neće sadržati broj elementa već konkretan naziv iz asocijativnog niza (*jabuke, kruske* itd.).

1.3. Funkcije za rad sa nizovima

Postoji veliki broj dodatnih funkcija za rad sa nizovima u *PHP*-u. Za kompletnu listu svih funkcija sa uputstvima za njihovo korišćenje, najbolje je pogledati zvaničnu dokumentaciju jezika.

U narednim odeljcima predstavimo samo neke funkcije koje mogu biti korisne u svakodnevnom radu.

1.3.1. Ispis sadržaja niza

Ukoliko u toku razvoja programa želimo da detaljnije ispitamo sadržaj neke promenljive (koja ne mora samo biti niz), možemo koristiti funkciju *print_r(\$promenljiva)*. Ova funkcija ispisuje sadržaj promenljive koja joj je prosleđena u grafički uređenom formatu pogodnom za lakše razumevanje, što je posebno pogodno kada radimo sa složenijim nizovima.

```
$magacin = [ "jabuke" => 100, "banane" => 34, "kruske" => 60, "breskve" => 60 ];  
  
echo 'Sadržaj promenljive $magacin <br>';  
  
print_r($magacin);
```

Rezultat izvršenja navedene naredbe prikazan je ispod:

```
Sadržaj promenljive $magacin
```

```
Array ( [jabuke] => 100 [banane] => 34 [kruske] => 60 [breskve]
=> 60 )
```

Funkcija `print_r` bez problema ispisuje sadržaj i višedimenzionalnih nizova.

1.3.2. Određivanje broja elemenata niza

Ukoliko želimo da saznamo koliko elemenata sadrži određeni niz, možemo koristiti funkciju `count($naziv_niza)` ili funkciju `sizeof($naziv_niza)`. Obe funkcije prihvataju naziv niza kao parametar i vraćaju broj elemenata koji taj niz sadrži. Imaju veliku praktičnu upotrebnu vrednost, a možemo ih koristiti i kada želimo da prođemo kroz elemente niza korišćenjem petlje koja nije *foreach*. U narednom primeru, sadržaj niza ćemo ispisati na ekranu korišćenjem petlje *for*. Da bismo prošli kroz sve elemente niza, krenućemo od pozicije 0 i povećavaćemo indeks sve dok ne dođemo do poslednjeg elementa niza – koji je poslednji element niza, saznaćemo upravo pomoću funkcije `count()`. Funkcija `sizeof` je zapravo drugo ime za funkciju `count` – obe funkcije rade potpuno isto i vraćaju istu vrednost.

```
<?php
    $voce = array( "jabuka", "banana", "kruska", "breskva" );

    $duzina_niza = count( $voce );

    for($i = 0; $i < $duzina_niza; $i++) {
        echo $voce[$i];
        echo "<br>";
    }
?>
```

Naravno, funkciju `count` smo mogli da pozovemo i direktno u *for* petlji bez čuvanja njenog rezultata u posebno promenljivoj `$duzina_niza`. Važno je samo obratiti pažnju da u petlji ne bi smeo da stoji znak `<=` umesto `<`. Razlog je jednostavan: Ukoliko niz ima,

na primer, 10 elemenata, to znači da se elementi nalaze na pozicijama od 0 do 9. Ukoliko bismo upotrebili znak `<=` to bi značilo da pokušavamo da pristupimo elementu na nepostojećoj poziciji 10. U tom slučaju, dobili bismo upozoravajuću poruku: “*Notice: Undefined offset*”.

1.3.3. Promena redosleda elemenata niza

Ukoliko želimo da u određenom trenutku promenimo redosled kojim su elementi poređani u regularnom nizu, možemo koristiti funkciju `shuffle($naziv_niza)`. Ova funkcija u okviru niza koji joj je prosleđen menja pozicije elemenata po nasumičnom redosledu. U zavisnosti od veličine niza, elementi će svakim pozivanjem biti pomešani na drugačiji način.

Jedna od primena može biti i situacija kada imate grupu slika i želite da svaki put kada se učita veb strana, budu prikazane samo tri nasumično izabrane. U nizu su sačuvani samo nazivi slika (slike možete da sačuvate u isti folder sa ovim *PHP* skriptom). Potom se poziva funkcija `shuffle`, a zatim prikazujemo tri slike koje se nalaze na pozicijama 0, 1 i 2 u nizu. Iako uvek učitavamo slike sa iste, prve tri pozicije, zahvaljujući tome što je prethodno pozvana funkcija `shuffle`, svaki put možemo očekivati potpuno drugačije slike na prva tri mesta.

```
<?php
    $slike = array( "prva.png", "druga.png", "treca.png",
"cetvrta.png", "peta.png" );
    shuffle($slike);
?>




```

Pažnju zaslužuje i naredba `<?= $slike[0] ?>`. Ona vrši ispis naziva slike koja sa nalazi na prvom mestu u nizu i zapravo predstavlja

sažetiji oblik naredne naredbe (ishod rada obe naredbe je potpuno isti, možete slobodno koristiti onu koja Vam više odgovara):

```
<?php echo $slike[0]; ?>
```

Rezultat pokretanja primera prikazuje slika 2.



Slika 2: Prikaz tri nasumično odabrane slike.

Samostalni rad

U prethodnom primeru smo prikazali tri prve slike iz niza na ekranu tako što smo tri puta ponovili praktično istu naredbu uz promenu indeksa niza. Obzirom da bi trebalo da izbegavamo kopiranje istih naredbi više puta, probajte da poboljšate primer tako što ćete napraviti petlju koja automatski tri puta vrši ispis slike. Na taj način, programski kod će postati pregledniji i sažetiji.

Isto rešenje možete dobiti i na drugi način - umesto korišćenja funkcije *shuffle*, možete koristiti i funkciju *array_rand*. O ovoj funkciji više možete pročitati u sekciji “Ostale korisne funkcije za rad sa nizovima”.

1.3.4. Funkcije za kretanje kroz niz bez korišćenja petlje

U određenim situacijama će se javiti potreba za prolaskom kroz sadržaj niza bez pravljenja petlje. Sve funkcije koje nam u tome pomažu zasnivaju se na postojanju *internog pokazivača* u nizu – ovaj pokazivač pamti trenutnu poziciju elementa (uvek počinje od

elementa na poziciji o) i dalje se pomera u skladu sa pozivima funkcija koje se mogu koristiti, a to su:

- *current()* – vraća vrednost trenutnog elementa;
- *next()* – pomera pokazivač na naredni element i vraća njegovu vrednost;
- *prev()* – pomera pokazivač na prethodni element i vraća njegovu vrednost;
- *end()* – pomera pokazivač na poslednji element u nizu i vraća njegovu vrednost;
- *reset()* – vraća poziciju internog pokazivača na poziciju o i prikazuje taj element;

U primeru ispod možemo videti kako ove funkcije rade u praksi.

```
<?php
    $voce = array( "jabuka", "banana", "kruska", "breskva" );

    echo current($voce) . "<br>";           // jabuka
    echo next($voce) . "<br>";              // banana
    echo end($voce) . "<br>";               // breskva
    echo prev($voce) . "<br>";              // kruska
    echo reset($voce) . "<br>";             // jabuka
    echo current($voce) . "<br>";           // jabuka
?>
```

1.3.5. Ostale korisne funkcije za rad sa nizovima

Pored funkcija koje smo obradili u prethodnim sekcijama, još nekoliko funkcija zaslužuje da skrenemo pažnju na njih i zato ćemo ukratko predstaviti:

- *array_pop(\$naziv_niza)* – već smo koristili funkciju *array_push* koja dodaje element na kraj niza – funkcija *array_pop* je njena suprotnost i ona uklanja poslednji element niza koji je prosledjen kao parametar;

- `array_rand($naziv_niza, 3)` – funkcija vraća nasumično izabrana 3 ključa iz niza koji je prosleđen kao parametar. Ukoliko je broj ključeva koje tražimo veći od 1, funkcija će vratiti niz. Ukoliko tražimo samo jedan nasumični ključ, biće vraćen kao brojna promenljiva (ne kao niz). Drugi parametar funkcije je opcionalan i čak se može i izostaviti.
- `array_reverse($naziv_niza)` – funkcija vraća u obrnutom redosledu sadržaj niza koji je prosleđen kao parametar.
- `sort($naziv_niza)` – sortiranje sadržaja regularnog niza u rastućem poretku. Za sortiranje u opadajućem poretku može se koristiti funkcija `rsort($naziv_niza)`. Postoje i druge varijacije funkcija za sortiranje nizova u zavisnosti od vrste niza (regularni, asocijativni) kao i algoritma koji se koristi za sortiranje. Na primer, `ksort` i `krsort` sortiraju asocijativni niz prema ključevima u rastućem ili opadajućem poretku, `uasort` se koristi za sortiranje elemenata po vrednostima korišćenjem sopstvene funkcije za poređenje itd.

1.4. Dvodimenzionalni nizovi

Do sada prikazani nizovi su isključivo bili jednodimenzionalni. Takve nizove nekada nazivamo vektorima, a možemo ih zamisliti kao niz podataka poređanih u samo jednom redu ili koloni.

U nekim situacijama nam može biti potrebno i više od jedne dimenzije i zbog toga se u praksi česo sreću dvodimenzionalni nizovi (drugi naziv: *matrica*), mada nismo ograničeni na samo dve dimenzije. Matricu je najlakše zamisliti kao tabelu koja ima određeni broj redova i kolona.

Kao primer možemo uzeti sliku 3 koja predstavlja ulazne podatke od kojih želimo da napravimo dvodimenzionalni niz u *PHP*-u.

Naziv	Količina	Cena
Jabuka	100	80

Banana	34	120
Kruška	60	150
Breskva	60	100

Slika 3: Tabela sa ulaznim podacima.

Dvodimenzionalni niz se pravi tako što u okviru osnovnog niza pravimo nove nizove koji predstavljaju svaki pojedinačni red u tabeli.

```
$voce = array(
    array( "Jabuka", 100, 80 ),
    array( "Banana", 34, 120 ),
    array( "Kruska", 60, 150 ),
    array( "Breskva", 60, 100 )
);
```

Nakon što smo napravili niz, možemo pristupiti elementima. Ukoliko želimo da na ekranu ispišemo reč "Banana" ili da promenimo njihovu količinu na 30kg, potrebno je napisati:

```
echo $voce[1][0];

$voce[1][1] = 40;
```

U prvim uglastim zagradama definiše se broj reda, druge zagrade sadrže broj kolone u kojoj se potrebna vrednost nalazi. Ukoliko je potrebno ispisati sadržaj kompletne matrice, potrebno je koristiti petlju – ne samo jednu, već dve, obzirom da je u pitanju niz sa dve dimenzije. Naredni primer prikazuje kako da ispišemo sve podatke o voću iz matrice.

```
echo "<table border=1 >";

for ($red = 0; $red < 4; $red++) {
    echo "<tr>";
    echo "<td>Redni broj: $red</td>";

    for ($kolona = 0; $kolona < 3; $kolona++) {
        echo "<td>" . $voce[$red][$kolona] . "</td>";
```

```
    }  
    echo "</tr>";  
}  
echo "</table>";
```

Spoljašnja *for* petlja zadužena je za prolazak kroz sve redove tabele – u svakoj iteraciji ove petlje, unutrašnja *for* petlja pristupa svim kolonama u određenom redu i pomoću naredbe `$voce[$red][$kolona]` pristupa svakoj ćeliji u matrici i ispisuje je na ekranu. Na ovaj način se praktično ćeliju po ćeliju krećemo kroz tabelu počevši od gornjeg levog ugla do donjeg desnog ugla. Ukoliko u tabeli imamo n kolona i m redova, to znači da ćemo obaviti $n \times m$ operacija na svakoj ćeliji. Za dovoljno velike matrice i složene operacije koje se primenjuju nad sadržajem svake ćelije, može doći do značajnog povećanja u trošenju vremenskih i računarskih resursa.

2. KLASSE I OBJEKTI

Klase i objekti čine najvažnije elemente svih objektno-orijentisanih programskih (OOP) jezika kao što je *PHP*. Suština OOP-a je pravljenje struktura koje su što sličnije onima u realnom svetu – svakodnevno smo okruženi raznim objektima koji imaju određene karakteristike i funkcionalnosti, i koji u okviru sebe mogu da sadrže druge objekte. Upravo takve objekte ćemo pokušati da predstavimo pomoću *PHP*-a. Kako bi uopšte bilo moguće započeti sa pravljenjem objekata, važno je razumeti klase i njihovu vezu sa objektima.

Klasa se može zamisliti kao obrazac koji se koristi za pravljenje određenog objekta. Ona može da sadrži određene funkcije, kao i polja koja čuvaju podatke. Naziv klase je obično apstraktan (*Kuća*, *Automobil*, *Grad*).

Objekat je za razliku od klase nešto konkretno – ako je naziv klase *Grad*, onda se objekat može zvati *beograd* ili *noviSad*. Od jedne klase, može nastati više objekata. Ukoliko je neki objekat nastao od određene klase, onda se kaže da on predstavlja *instancu* te klase. Na primer, *beograd* je instanca klase *Grad*. Potrebno je obratiti pažnju i na neka pravila imenovanja:

- naziv klase i naziv objekta treba da bude napisan bez razmaka;
- ukoliko naziv klase ili objekta ima više reči, svaku novu reč pišemo velikim početnim slovom (*noviSad*);

- naziv klase uvek počinje velikim slovom (*Grad*);
- naziv objekta (kao i nazivi promenljivih) počinje malim slovom.

Iako će se *PHP* kod izvršavati bez problema i ukoliko ova pravila nisu ispoštovana, razvoj će svakako biti lakši uz pridržavanje ovih jednostavnih smernica.

2.1. Rad sa klasama i objektima

Pravljenje nove klase u *PHP*-u je veoma jednostavno. Koristi se ključna reč *class* koju prati naziv klase i jedan komplet vitičastih zagrada koje definišu gde počinje i gde se završava telo klase.

```
class Krug {  
    // ovde se nalazi sadržaj klase  
}
```

Klasa koja je upravo napravljena je zapravo prazna – ona nema nikakve funkcionalnosti ili podatke i stoga trenutno ne radi ništa. Da bi klasa *Krug* bila korisnija, definisaćemo promenljive koje čuvaju koordinatnu poziciju centra kruga, kao i njegov poluprečnik.

```
class Krug {  
    public $x;  
    public $y;  
    public $r;  
}
```

Promenljive definisane u klasi (nazivamo ih i *polja* – engl. *field*) su slične običnim promenljivama u *PHP*-u uz dodatak ključne reči *public* ispred naziva. Ova ključna reč definiše vidljivost određene promenljive, a moguće vrednosti koje se mogu definisati su *public*, *protected* i *private* i o njima će više reči biti u narednim sekcijama. Za sada je dovoljno znati da oznaka *public* znači da je promenljiva

javno dostupna za pristup, što bi u praktičnim primenama trebalo izbegavati ili barem veoma pažljivo koristiti.

Kako bismo napravili objekte klase *Krug*, potrebno je upotrebiti sledeću konstrukciju koja podrazumeva korišćenje ključne reči *new* iza koje sledi naziv klase na osnovu koje pravimo novi objekat.

```
$maliKrug = new Krug();  
$velikiKrug = new Krug();
```

Da bismo podesili poluprečnik objekta (a uzimajući u obzir da su polja označena kao *public*), potrebno je pristupiti pomoću operatora *->*. Obratite pažnju da iako u definiciji klase piše *\$r*, kada se pristupa objektu, navodi se samo *->r*. Razlog je što je znak *\$* već naveden na početku ispred naziva objekta - *\$maliKrug*.

```
$maliKrug->r = 10;  
$velikiKrug->r = 20;
```

Svaki objekat će imati svoju verziju polja *\$x*, *\$y* i *\$r* – promena poluprečnika malog kruga *neće uticati* na poluprečnik velikog kruga. Kompletiran primer koji izračunava površinu malog i velikog kruga uz korišćenje ugrađene *PHP* funkcije *pi()* se nalazi ispod.

```
<?php  
class Krug {  
    public $x;  
    public $y;  
    public $r;  
}  
  
$maliKrug = new Krug();  
$velikiKrug = new Krug();  
  
$maliKrug->r = 10;  
$velikiKrug->r = 20;  
  
echo $maliKrug->r * $maliKrug->r * pi() . "<br>";  
echo $velikiKrug->r * $velikiKrug->r * pi();
```

```
?>
```

Samostalni rad

Koristeći kao osnovu prethodni primer, popunite sva polja objekata *\$maliKrug* i *\$velikiKrug*, izračunajte njihove obime, kao i rastojanje između koordinatnih centara ova dva kruga.

2.1.1. Vidljivost polja u klasama

U prethodnim primerima, svako polje definisano u klasama imalo je modifikator *public* ispred naziva promenljive:

```
<?php
    class Krug {
        public $r;
        // ostatak programskog koda
    }

    $maliKrug = new Krug();
    $velikiKrug = new Krug();

    $maliKrug->r = 10;
    $velikiKrug->r = 20;
?>
```

Ovaj modifikator označava da se određenoj promenljivoj može pristupiti i izvan same klase, a samim tim se može promeniti i njena vrednost. U razvoju softvera, ovo je situacija koju u većini slučajeva želimo da izbegnemo – ukoliko je moguće izmeniti sadržaj promenljive i izvan klase, to znači da ne postoji potpuna kontrola na tom promenljivom.

Upravo da bi se izbegli slični problemi, postoje i dva dodatna modifikatora: *protected* i *private*.

```
<?php
    class Vrednost {
```

```
        public $a;
        protected $b;
        private $c;
    }

    $test = new Vrednost();

    $test->a = 10; // ispravna naredba
    $test->b = 20; // problem – greška pri pristupu
    $test->c = 30; // problem – greška pri pristupu
    ?>
```

Naredna lista sadrži razlike između ovih modifikatora pristupa:

- *public* označava da se promenljivoj može pristupiti i iz klase, i iz svih podklasa, ali i sa bilo kog mesta izvan klase;
- *protected* označava da se promenljivoj može pristupiti iz klase i svih njenih podklasa, ali ne i izvan klase (zbog toga se javlja greška u prethodnom primeru);
- *private* označava da se promenljivoj može pristupiti samo iz klase – pristup iz podklasa ili sa lokacije izvan klase rezultuje greškom prilikom pristupa.

Svaki od navedenih modifikatora ima svoju upotrebnu vrednost u razvoju *PHP* aplikacija. Obzirom da se u ovom udžbeniku neće puno raditi sa opcijama nasleđivanja (pravljenja podklasa), neće se puno koristiti ni modifikator *protected* – najviše će se koristiti *public* i *private*. Na mnogim mestima na kojima je do sada bio korišćen *public* (kao što su promenljive u klasama), od sada će se koristiti *private*.

Naravno, postavlja se pitanje: “Ako je promenljiva klase podešena kao *private* i ne može joj se promeniti vrednost izvan klase, da li to znači da se njena vrednost ne može uopšte menjati?”. Iako je podešavanjem pristupa promenljivoj na *private* zaista sprečen direktan pristup promenljivoj, to ne znači da se ona ne sme menjati

pomoću drugih funkcija u okviru klase. Ove funkcije nazivamo *get* i *set* funkcije i one će biti obrađene u narednim sekcijama ovog poglavlja.

2.2. Funkcije

Prethodni objekti i klase su bili veoma jednostavni – sadržali su određene podatke, ali nisu imali nikakvu ugrađenu funkcionalnost. Čak je i operacija izračunavanja površine kruga bila sprovedena izvan same klase. Dodatno, za izračunavanje površine dva kruga, bilo je potrebno skoro isti kod napisati dva puta:

```
echo $maliKrug->r * $maliKrug->r * pi() . "<br>";  
echo $velikiKrug->r * $velikiKrug->r * pi();
```

Šta bismo mogli da očekujemo ako bi bilo potrebno izračunati površinu 10 krugova – kopiranje iste naredbe toliko puta uz izmenu objekata na koji se obračun odnosi bi zahtevalo suviše ručnog posla i bilo bi pogodno tlo za pravljenje grešaka. Stoga, kako bi programski kod bio jednostavniji, pregledniji i manje podložan greškama, potrebno je potrebne funkcionalnosti grupisati na jednom mestu, a zatim ih samo po potrebi pozivati. U tu svrhu, koriste se *funkcije*. Opšti oblik funkcije koja se može definisati u okviru klase je:

```
public function nazivFunkcije() {  
    // telo funkcije  
}
```

Kada navedeni obrazac primenimo na izračunavanje površine kruga i prethodni primer prerađimo u skladu sa tim, dobijamo kod kao u sledećem listingu:

```
<?php  
class Krug {  
    public $x;  
    public $y;
```

```

        public $r;

        public function izracunajPovrsinu() {
            $p = $this->r * $this->r * pi();
            echo "Povrsina je $p <br>";
        }
    }

    $maliKrug = new Krug();
    $velikiKrug = new Krug();

    $maliKrug->r = 10;
    $velikiKrug->r = 20;

    $maliKrug->izracunajPovrsinu();
    $velikiKrug->izracunajPovrsinu();
?>

```

Potrebno je obratiti pažnju na nekoliko važnih elemenata. Sam obračun površine se više ne vrši izvan klase, već u samoj klasi u okviru odgovarajuće funkcije - *izracunajPovrsinu()*. Zahvaljujući tome, logika funkcije se nalazi samo na jednom mestu i biće po potrebi pozivana onoliko puta koliko je potrebno bez nepotrebnog kopiranja programskog koda (na primer, *\$maliKrug->izracunajPovrsinu()*).

U ovom primeru upotrebljena je i jedna nova naredba - *\$this*. Ova naredba se koristi kako bi se naglasilo da se iz funkcije pristupa promenljivama klase. Naredni primer još bolje ilustruje upotrebu naredbe *\$this*. U okviru njega, definisana je klasa *Broj* i u okviru nje promenljiva *\$a* sa fiksnom vrednošću 5 i funkcija zbir u okviru koje je definisana promenljiva *\$b* sa vrednošću 10. Obzirom da u funkciji *sabiranje* želimo da pristupimo obema promenljivama, za promenljivu *\$a* ćemo morati da dodamo i naredbu *\$this* kako bismo naglasili da ta promenljiva nije definisana direktno u funkciji, već u nadklasi.

```
<?php
    class Broj {
        public $a = 5;

        public function sabiranje() {
            $b = 10;
            $zbir = $this->a + $b;
            echo "Zbir je $zbir<br>";
        }
    }
?>
```

2.2.1. Konstruktori

U objektno-orijentisanom programiranju, konstruktor je metod kojim se inicijalizuju novi objekti i poziva se automatski prilikom pravljenja objekta. Iako to nije bilo izričito naglašeno, konstruktor je pozivan i u prethodnim primerima:

```
$maliKrug = new Krug();
```

U ovom slučaju, radilo se o pozivanju podrazumevanog konstruktora, ali uvek možemo definisati i sopstvenu programsku logiku koja treba da bude pozvana i izvršena u okviru konstruktora:

```
<?php
    class Broj {
        public $a;

        public function __construct($broj) {
            // Vrednost prosleđenu konstruktoru
            // čuvamo u promenljivoj $a.
            $this->a = $broj;
        }

        public function sabiranje() {
            $b = 10;
            $zbir = $this->a + $b;
            echo "Zbir je $zbir<br>";
        }
    }
}
```

```

    }
}

$noviBroj = Broj(20);
$noviBroj->sabiranje(); // Ispis na ekranu: Zbir je 30
?>

```

Primer je modifikovan na način da vrednost *\$a* nije odmah fiksirana na broj 5, već se željena vrednost prilikom pravljenja objekta prosleđuje u zagradama (*\$noviBroj = Broj(20);*). Vrednost upisana između zagrada automatski se prosleđuje metodu *__construct* kao parametar pod imenom *\$broj*. Potom se prosleđena vrednost prenosi u promenljivu klase *\$a* (*\$this->a = \$broj;*) i postaje dostupna i u drugim funkcijama u klasi.

Funkcija za konstruktor se uvek zove *__construct* (obratite pažnju da se pre reči *construct* nalaze dve donje crte), a za razliku od nekih drugih programskih jezika, u *PHP*-u se ne sme definisati nekoliko konstruktora sa različitim brojem parametara, već samo jedan. Broj parametara u konstruktoru može biti proizvoljan – bez parametara, samo jedan ili više parametara.

Samostalni rad

Napraviti klasu koja predstavlja krug. Krug je definisan koordinatama centra i poluprečnikom izraženim u centimetrima. Klasa *Krug* treba da poseduje odgovarajuća polja (koordinate, poluprečnik), konstruktor, kao i metode koji mogu da promene neke od vrednosti (centar, poluprečnik). Potrebno je napraviti i funkcije koje računaju obim i površinu kruga, kao i funkciju koja ispisuje sve podatke o krugu, uključujući i obim i površinu.

Napomena: funkcija za ispis podataka o krugu ne sme ponovo da izračunava obim i površinu, već za to mora da koristi namenske metode.

Kako bi se testirao rad klase *Krug*, napišite programski kod (izvan klase) koji pravi 10 krugova koji imaju prečnike od 2, 4, 8, 16, 32, 64, ... centimetara i koji izračunava zbirni obim i zbirnu površinu svih 10 krugova.

2.2.2. Funkcije *get* i *set*

Kako bi se omogućio pristup promenljivama klase koje su označene kao *private*, potrebno je napraviti funkcije koje pristupaju vrednosti promenljive (dohvatiti, engl. *get*) ili je menjaju (engl. *set*).

```
<?php
class Automobil {
    private $boja;
    public function getBoja() {
        return $this->boja;
    }

    public function setBoja($boja) {
        // Promenljiva $this->boja označava
        // promenljivu
        // definisanu u klasi. Promenljiva $boja je
        // parametar funkcije setBoja()
        $this->boja = $boja;
    }
}

$prviAutomobil = new Automobil();

$prviAutomobil->setBoja("crvena");
echo $prviAutomobil->getBoja(); // ispisuje tekst
"crvena"
?>
```

Iako deluje da bi jednostavnije bilo da je promenljiva *\$boja* označena kao *public* kako bi joj se direktno pristupalo (i kako bi se izbeglo pravljenje *get* i *set* funkcija), zapravo smo na ovaj način dobili sigurniju aplikaciju u okviru koje možemo podesiti kojim poljima se može pristupati – neka polja mogu imati i *get* i *set*, neka samo *get*, neka samo *set*, a neka polja mogu biti u potpunosti zatvorena za pristup (bez bilo kog metoda za pristup). Kao još jedna prednost u odnosu na direktan pristup se može navesti i to što se u

ovim metodima može ugraditi i logika koja vrši dodatne provere ispravnosti podataka.

U prethodnom primeru postoji samo jedna promenljiva sa svojim *get* i *set* funkcijama. Ukoliko bi postojalo 10 promenljivih i ukoliko bi bilo potrebe da svaka od njih ima i *get* i *set* funkcije, bilo bi potrebno napisati 20 različitih funkcija što je prilično velika količina koda zbog koje cela klasa može izgledati dosta nepreglednije. Kao rešenje nameće se korićenje magičnih *get* i *set* funkcija.

2.2.3. Magične funkcije *get* i *set*

Magične funkcije u jeziku *PHP* su specijalne funkcije čije ime počinje sa dve donje crte i koje će biti automatski pozvane u skladu sa određenim događajima u aplikaciji. U prethodnim segmentima ovog poglavlja je već bilo reči od jednoj od ovih funkcija - *__construct()*. Još dve važne funkcije su i *__get()* i *__set()*. Oni omogućavaju jednostavan pristup poljima klase bez potrebe za pravljenjem pojedinačnih *get* i *set* metoda. Ukoliko pogledamo naredni primer, može se videti da imamo samo dve funkcije koje pristupaju poljima *\$boja* i *\$snaga*.

```
<?php
class Automobil {
    private $boja;
    private $snaga;

    public function __get($nazivPolja) {
        // obratite pažnju na dva znaka $
        return $this->$nazivPolja;
    }
    public function __set($nazivPolja, $vrednost) {
        $this->$nazivPolja = $vrednost;
    }
}

$noviAuto = new Automobil();
$noviAuto->boja = 'crvena';
```

```
$noviAuto->snaga = 100;  
  
echo "Boja: $noviAuto->boja; snaga: $noviAuto->snaga ";  
?>
```

Funkcija `__get` prihvata parametar koji predstavlja naziv polja kom je potrebno pristupiti. Kada se na kraju primera pozove naredba `$noviAuto->boja`, zapravo se poziva metod `__get`, a kao parametar mu se prosleđuje vrednost “boja” – ova vrednost biće sačuvana u promenljivoj `$nazivPolja`. Nakon toga, u naredba `return` vraća vrednost `$this->$nazivPolja` - `$this` označava da je potrebno pristupiti promenljivoj klase, a `$nazivPolja` definiše kojoj tačno promenljivoj klase je potrebno pristupiti.

Funkcija `__set` prihvata dva parametra – naziv polja kom je potrebno pristupiti i vrednost koji je potrebno sačuvati u tom polju. Nakon pozivanja naredbe `$noviAuto->boja = 'crvena'`, funkciji `__set` se prosleđuju vrednosti `boja` (čuvaće se u promenljivoj `$nazivPolja`) i `crvena` (čuvaće se u promenljivoj `$vrednost`). Potom se pomoću naredbe `$this->$nazivPolja = $vrednost` u željenom nazivu polja (promenljiva `$boja`) čuva vrednost `crvena`.

Funkcije `__get` i `__set` se mogu proširiti i dodatnim funkcionalnostima. Na primer, moguće je definisati određene kriterijume za pristup poljima – za potrebe demonstracije, modifikovana je `__get` funkcija iz prethodnog primera:

```
public function __get($nazivPolja) {  
    if($nazivPolja == "snaga") {  
        echo "Nije dozvoljeno da učitate snagu.";  
        return;  
    }  
    return $this->$nazivPolja;  
}
```

U *if* naredbi se proverava da li se pokušava pristupiti polju `snaga`. Ukoliko je odgovor tačan, ispisuje se poruka i poziva se naredba `return` bez parametara čime se prekida dalji rad funkcije. U

suprotnom, ukoliko se pokušava nekom drugom polju, funkcija će vratiti traženu vrednost pomoću naredbe `return $this->$nazivPolja`. Ovakve primene su posebno korisne u situacijama kada je potrebno zabraniti mogućnost čitanja ili menjanja vrednosti nekih polja klase ili je potrebno izvršiti određene provere pre pristupanja promenljivama.

2.2.4. Magična funkcija `__toString`

Funkcija `__toString` spada u veoma korisne magične funkcije, a koristi se za jednostavn ispis podataka o objektu, na primer u okviru `echo` naredbe. U prethodnom primeru, u sekciji o magičnim `get` i `set` funkcijama, podaci o automobilu su na samom kraju ispisivani pomoću naredbe:

```
echo "Boja: $noviAuto->boja; snaga: $noviAuto->snaga";
```

Ukoliko postoji potreba da se ispišu podaci o više automobila, ili se javi potreba da se ispisuju podaci o dodatnim poljima o automobilima, ovakav pristup bi ubrzo postao veoma naporan za rad jer bi podrazumevao puno kopiranja i izmena postojećih naredbi. Zapravo je mnogo bolje napraviti funkciju u okviru klase *Automobil* koja je zadužena za ispis podataka o automobilu, a potom tu funkciju samo pozivati po potrebi. Ukoliko se javi potreba za nekom izmenom u ispisu, ona će biti napravljena samo na jednom mestu – u samoj funkciji. Naredni primer predstavlja modifikaciju klase *Automobil*.

```
<?php
class Automobil {
    private $boja;
    private $snaga;

    // get i set funkcije
    public function __get($nazivPolja) {
        // obratite pažnju na dva znaka $
        return $this->$nazivPolja;
    }
}
```

```
        }  
        public function __set($nazivPolja, $vrednost) {  
            $this->$nazivPolja = $vrednost;  
        }  
  
        // Magična funkcija __toString  
        public function __toString() {  
            return "Boja: $this->boja; snaga: $this->snaga";  
        }  
    }  
  
    $noviAuto = new Automobil();  
    $noviAuto->boja = 'crvena';  
    $noviAuto->snaga = 100;  
  
    echo $noviAuto;  
?>
```

U ovom slučaju, poslednji red je znatno pojednostavljen – koristi se samo naredba `echo $noviAuto`. Ova naredba predstavlja pokušaj ispisivanja objekta `$noviAuto` na ekranu pomoću naredbe `echo`. Da nije definisana funkcija `__toString`, rezultat rada programskog koda bi bila greška:

Recoverable fatal error: Object of class Automobil could not be converted to string.

Obzirom da je funkcija `__toString` definisana u klasi `Automobil`, navedena greška se neće javiti, već se pomenuta magična funkcija izvršiti, a izlaz na ekranu će biti:

Boja: crvena; snaga: 100

Magična funkcija `__toString` može biti i od velike koristi prilikom testiranja programa, jer može pomoći u jednostavnom pregledanju sadržaja promenljivih.

2.3. Ključna reč static

U jeziku *PHP* ključna reč *static* označava da se određeno polje ili funkcija u okviru klase koriste, čak iako nije napravljen objekat te klase. Ukoliko se radi o statičkoj promenljivoj, ona će biti dostupna svim objektima određene klase koji će je deliti.

2.3.1. Statičke promenljive

U narednom primeru napravljena je jednostavna klasa koja sadrži samo jedno statičko polje pod nazivom *\$deljeniBroj*.

```
<?php
class StaticPrimer {
    public static $deljeniBroj = 0;
}

$primer1 = new StaticPrimer();
$primer2 = new StaticPrimer();

$primer1::$deljeniBroj = 10;
$primer2::$deljeniBroj = 15;

echo $primer1::$deljeniBroj;
?>
```

Potom se prave objekti *\$primer1* i *\$primer2*, a nakon toga se za objekat *\$primer1* podešava vrednost za *\$deljeniBroj* na 10. U narednom redu se za *\$primer2* promenljiva *\$deljeniBroj* postavlja na 15. Obzirom da se radi o statičkoj promenljivoj, oba objekta zapravo dele samo jednu promenljivu – kada se u poslednjem redu bude ispisao sadržaj promenljive *\$deljeniBroj* u okviru objekta *\$primer1*, biće ispisana vrednost 15 koja je poslednja definisana.

Važno je obratiti pažnju i da se u radu sa statičkim proimenljivama i funkcijama koristi operator *::* umesto operatora *->*.

2.3.2. Statičke funkcije

Slično kao i sa statičkim poljima, i statičke funkcije se mogu koristiti čak iako nije napravljen nijedan objekat klase. U okviru statičke funkcije se ne može koristiti reč *this* iz razloga što možda neće biti napravljena nijedna instanca klase – to je i razlog zašto se za pristup static promenljivama koristi naredba *self*. Stoga, statički metod ne može pristupati poljima koja nisu statička. U narednom primeru napravljena je klasa koja ima jednu statičku promenljivu (*\$brojacObjekata*), jednu statičku funkciju (*kolikoObjekataPostoji()*) i konstruktor.

```
<?php
class BrojacObjekata {
    private static $brojacObjekata = 0;

    public function __construct() {
        // Statičkoj promenljivoj se u okviru klase
        // pristupa pomoću naredbe self.
        self::$brojacObjekata++;
    }

    // Ako u funkciji koristimo static promenljivu,
    // onda i funkcija mora biti static.
    public static function kolikoObjekataPostoji() {
        return self::$brojacObjekata;
    }
}

$v1 = new BrojacObjekata();
$v2 = new BrojacObjekata();
$v3 = new BrojacObjekata();
$v4 = new BrojacObjekata();

// Ako se javna static funkcija poziva izvan klase,
// uvek se navodi ime_klase::ime_static_funkcije()
echo BrojacObjekata::kolikoObjekataPostoji();
?>
```

Prilikom svakog pozivanja konstruktora (tj. pri svakom pravljenju novog objekta klase *BrojacObjekata*), izvršava se naredba *self::\$brojacObjekata++* koja svaki put uvećava vrednost statičke promenljive *\$brojacObjekata* za jedan. Prilikom pravljenja novih objekata klase (pozivanjem naredbe *new BrojacObjekata()*) doći će do uvećanja pomenutog brojača – u njemu će se nalaziti broj koji će u svakom trenutku predstavljati evidenciju koliko objekata klase *BrojacObjekata* je do sada napravljeno.

Pozivanjem statičke funkcije *BrojacObjekata::kolikoObjekataPostoji()* moći će da se pristupi podatku o tome koliko objekata je do tog trenutka napravljeno. Ova funkcija pristupa vrednosti deljene statičke promenljive pomoću narebe *self::\$brojacObjekata*.

Klasa *BrojacObjekata* je napravljena na taj način da je polje *\$brojacObjekata* označeno kao *private* i jedini način da se njegova vrednost uveća je pravljenje novog objekta (pozivanje konstruktora). Izmene te vrednosti nisu omogućene, a jedino je moguće pročitati trenutnu vrednost pomoću funkcije *kolikoObjekataPostoji()*. Na ovaj način je obezbeđeno da ne može doći do bilo kakvih (slučajnih ili namernih) izmena broja napravljenih objekata, stoga možemo biti sigurni da je navedeni broj uvek ispravan.

Samostalni rad

Klasa *BrojacObjekata* je u stanju da vodi evidenciju o napravljenim objektima, ali problem može nastati kada neki objekat bude uklonjen (na primer, ukoliko se pozove naredba *unset(\$v4)*; na kraju skripta). Trenutna implementacija omogućava da broj objekata samo raste, ali ne i da se smanjuje.

Potrebno je prepraviti prethodni primer i napraviti i destruktora (funkcija *__destruct()*) koji će voditi računa o umanjivanju broja objekata i imaće vrlo sličnu logiku kao i konstruktor.

Napomena: Funkcija *__destruct()* je kao i *__construct()* magična funkcija i automatski se poziva kada neki objekat bude uklonjen.

Samostalni rad

Napraviti klasu *Naselje* koja radi sa objektima tipa *Zgrada*. Zgrada treba da čuva barem podatke o adresi, ceni sprata, broju spratova i imenu investitora.

Naselje treba da poseduje određeni broj zgrada koje se čuvaju u nizu. Niz mora biti privatan, a sve operacije se sprovode pomoću odgovarajućih funkcija. U konstruktoru je potrebno definisati koji je početni planirani broj zgrada u naselju (tj. nizu), kao i koji je maksimalni broj zgrada koje je dozvoljeno napraviti.

Potrebno je napraviti funkcije koji omogućavaju:

- dodavanje nove zgrade u naselje (može se dodati samo ako maksimalni broj zgrada nije prevaziđen),
- brisanje zgrade na određenoj adresi,
- ispis podataka o zgradi koja se nalazi na određenoj adresi,
- ispis vrednosti svih zgrada u celom naselju,
- ispis adrese zgrade koja ima najveću cenu po spratu,
- ispis imena investitora koji je napravio najviše zgrada u naselju.

2.4. Nasleđivanje

Nasleđivanje je veoma važan koncept u objektno orijentisanom programiranju koji za cilj ima stvaranje bolje strukture aplikacija i smanjivanje potrebe za kopiranjem i izmenama postojećeg programskog koda. U procesu nasleđivanja uvek postoji nadređena klasa (roditelj) i izvedena klasa koja nasleđuje osobine nadređene klase (dete). Za nasleđivanje se koristi ključna reč *extends*.

2.4.1. Proces nasleđivanja klasa

U cilju što jednostavnijeg objašnjenja procedure nasleđivanja, biće napravljena jednostavna struktura koja oslikava relaciju između nasleđenih klasa:

```

<?php
class Roditelj {
    public $bojaOciju = "zelena";
}

class Dete extends Roditelj {
    // Za sada, klasa Dete je prazna.
}

$prvoDete = new Dete();

echo $prvoDete->bojaOciju;
?>

```

U klasi *Roditelj* definisano je samo polje za boju očiju (i označeno je kao *public* radi jednostavnosti primera). Klasa *Dete* nasleđuje klasu *Roditelj* i sama nema bilo kakav programski kod, ali zato nasleđuje elemente iz klase *Roditelj*. Stoga, kada pomoću naredbe *\$prvoDete = new Dete()* bude napravljen prvi objekat klase *Dete*, automatski će biti napravljen i objekat klase *Roditelj* čije će karakteristike postati dostupne i u objektu klase *Dete*. Zato će poziv za ispis na ekranu *\$prvoDete->bojaOciju* zapravo prikazati boju očiju nadređene klase (zelena boja).

Ukoliko je potrebno obezbediti da boja očiju deteta može da bude drugačija od roditeljske, primer se može preraditi na sledeći način:

```

<?php
class Roditelj {
    public $bojaOciju = "zelena";
}

class Dete extends Roditelj {
    public $bojaOciju;

    public function __construct($bojaOciju) {
        $this->bojaOciju = $bojaOciju;
    }
}

```

```
$prvoDete = new Dete("braon");  
  
echo $prvoDete->bojaOciju;  
?>
```

U klasi *Dete* definisano je polje za boju očiju (zove se identično kao i polje u nadređenoj klasi, *\$bojaOciju*), kao i konstruktor kome se prosleđuje željena boja očiju. Kada u konstruktoru bude pozvana naredba *\$this->bojaOciju = \$bojaOciju* biće definisana boja očiju u izvedenoj klasi za objekat *\$prvoDete* (braon boja). Boja očiju roditelja će ostati nepromenjena (zelena boja). Kada na kraju primera bude pozvana naredba *echo \$prvoDete->bojaOciju*, na ekranu će biti ispisano da dete ima braon boju očiju.

2.4.2. Vidljivost elemenata prilikom nasleđivanja

Kao što je već prikazano u prethodnim sekcijama, postoje tri modifikatora pristupa koji definišu vidljivost elemenata: *public*, *protected* i *private*. *Public* omogućava javnu dostupnost. *Private* je potpuno suprotan od *public* i označava da je element potpuno sakriven u okviru jedne klase. *Protected* se nalazi između preostala dva modifikatora i sakriva element izvan klase, ali će dati pristup elementu iz podklase.

U narednom primeru definisana je klasa *NadredjenaKlasa* i u okviru nje tri jednostavne funkcije, a svaka od njih ima različit modifikator pristupa. Iz ove klase izvedena je nova klasa nazvana *NasledjenaKlasa* koja ima definisan samo konstruktor u okviru koga se pozivaju sva tri metoda iz nadređene klase. Poslednji korak je pravljenje objekta klase *NasledjenaKlasa* u poslednjoj naredbi – pravljenjem objekta se zapravo inicira konstruktor čime se pozivaju funkcije iz nadklase.

```
<?php  
class NadredjenaKlasa {  
    public function javnaFunkcija() {  
        echo "Javna funkcija <br>";  
    }  
}
```

```
    }  
    protected function zasticenaFunkcija() {  
        echo "Zaštićena funkcija <br>";  
    }  
    private function privatnaFunkcija() {  
        echo "Privatna funkcija <br>";  
    }  
}  
  
class NasledjenaKlasa extends NadredjenaKlasa {  
  
    public function __construct() {  
        $this->javnaFunkcija();  
        $this->zasticenaFunkcija();  
        //$this->privatnaFunkcija(); // Greška  
    }  
}  
  
$test = new NasledjenaKlasa();  
?>
```

Pozivanje javne i zaštićene funkcije će proteći bez problema, ali poziv privatne funkcije će prijaviti grešku:

Fatal error: Uncaught Error: Call to private method NadredjenaKlasa::privatnaFunkcija() from context 'NasledjenaKlasa'

Kroz ovaj primer, prikazano je ponašanje modifikatora direktno u klasama i podklasama (*public* i *protected* dozvoljavaju pristup, *private* ne). Situacija se menja kada je potrebno pristupiti elementima izvan klasa. Ispod prethodnog programskog koda, mogu se dodati i sledeće naredbe:

```
$test2 = new NadredjenaKlasa();  
  
$test2->javnaFunkcija();  
$test2->zasticenaFunkcija(); // Greška  
$test2->privatnaFunkcija(); // Greška
```

Napravljen je novi objekat (*\$test2*), a ovaj put nije korišćena nasleđena klasa, već nadklasa kako bi funkcije mogle da budu pozvane direktno. Nakon pokretanja, samo će biti moguće pristupiti javnoj funkciji – razlog je što pri korišćenju izvan klase, samo *public* dozvoljava pristup, a *protected* i *private* ne.

2.4.3. Ključna reč **final**

Ključna reč *final* se koristi u slučajevima kada je iz sigurnosnih ili drugih razloga potrebno sprečiti redefinisane određene funkcije prilikom nasleđivanja.

```
<?php
    class NadredjenaKlasa {
        final public function funkcijaTest() {
            echo "Funkcija za test<br>";
        }
    }

    class NasledjenaKlasa extends NadredjenaKlasa {
        public function funkcijaTest() {
            echo "Redefinisana funkcija za test<br>";
        }
    }

    $test = new NasledjenaKlasa();
?>
```

U ovom slučaju, funkcija u nadklasi je označena kao *final*. Kada se u podklasi pokuša redefinisane funkcije, pojaviće se sledeća greška:

Fatal error: Cannot override final method
NadredjenaKlasa::funkcijaTest()

Ključna reč *final* se može koristiti i prilikom definisanja klase – klasa koje je označena kao *final* se ne može dalje nasleđivati.

2.4.4. Redefinisanje funkcija

U određenim situacijama, javiće se potreba da se u izvedenoj klasi redefiniše funkcija koja već postoji u nadklasi. Razlog može biti što je, na primer, u funkciji u izvedenoj klasi potrebno dodati određene funkcionalnosti koje ne postoje u nadklasi. Naredni primer predstavlja upravo takvu situaciju.

```
<?php
    class Zivotinja {
        public function zvuk() {
            echo "Podrazumevani zvuk<br>";
        }
    }

    class Pas extends Zivotinja {
        public function zvuk() {
            parent::zvuk();
            echo "Avav<br>";
        }
    }

    class Macka extends Zivotinja {
        public function zvuk() {
            echo "Mjau<br>";
        }
    }

    $zuca = new Pas();
    $zuca->zvuk();

    $mica = new Macka();
    $mica->zvuk();
?>
```

Klasa *Zivotinja* definiše osnovnu funkcionalnost koju bi trebalo da ima svaka životinjska vrsta, a to je mogućnost da proizvede neku vrstu zvuka. Vrednost koju ispisuje funkcija *zvuk()* u okviru ove klase se može smatrati kao podrazumevani zvuk svake životinje. Obzirom da ostale životinje koje se koriste u primeru (klase *Pas*, *Macka*) bi trebale da imaju zvuk prilagođen svojoj vrsti, to znači da bi njihovu funkciju *zvuk()* trebalo redefinisati. To se radi tako što se u izvedenoj klasi napravi funkcija koja ima isti *potpis*¹ kao željena funkcija u nadklasi. Zbog toga, kada se pozove ova funkcija (*\$zuca->zvuk()* i *\$mica->zvuk()*), samo će se pozivati metod zvuk iz podklase.

Ukoliko postoji potreba da uz poziv metoda iz podklase bude pozvan i metod sa istim imenom iz nadklase, to se može uraditi pomoću oznake *parent*. U funkciji *zvuk()* u klasi *Pas* dodata je naredba *parent::zvuk()* koja će prvo pozvati i izvršiti funkciju iz klase *Zivotinja*, a onda prikazati i zvuk kojim se oglašava pas. Rezultat izvršenja prethodnog primera se nalazi ispod.

Podrazumevani zvuk

Avav

Mjau

2.5. Polimorfizam

Polimorfizam je važan koncept objektno orijentisanog programiranja prema kome sve klase treba da imaju specifične funkcionalnosti uz korišćenje iste organizacije. Zbog toga nije važno tačno znati od koje klase je nastao koji objekat iz razloga što imaju istu strukturu. U naredne dve sekcije će biti obrađeni interfejsi i apstraktne klase koji čine osnovu za polimorfizam.

¹ Dve funkcije imaju isti potpis ukoliko imaju isto ime i isti broj parametara.

2.5.1. Interfejsi

Interfejsi omogućavaju pisanje specifikacije funkcija koje klase moraju da implementiraju, ali bez navođenja konkretne implementacije tih funkcija.

Pravljenje interfejsa se obavlja na isti način kao i pravljenje klasa, uz korišćenje ključne reči *interface* umesto *class*. U okviru interfejsa navode se funkcije, ali se ne navodi programski kod tih funkcija. Sve funkcije moraju biti označene kao *public*. Sledi definicija jednostavnog interfejsa.

```
interface AutomobilInterfejs {  
  
    // Obratiti pažnju da se na kraju definicije funkcije  
    // nalazi ; umesto {}  
  
    public function upaliMotor();  
    public function ugasiMotor();  
}
```

Proces uvođenja interfejsa u određenu klasu se naziva implementacija i podrazumeva korišćenje ključne reči *implements* – ona se koristi na sličan način kao i ključna reč *extends*.

```
class PutnickiAutomobil implements AutomobilInterfejs {  
  
    // Obratiti pažnju da se sada na kraju definicije funkcije  
    // nalaze {}  
  
    public function upaliMotor() {  
        // Programski kod  
    }  
    public function ugasiMotor() {  
        // Programski kod  
    }  
}
```

Pri nasleđivanju klasa, podklasa nasleđuje samo od jedne nadklase, ali zato jedna klasa prema potrebi može implementirati i više od jednog interfejsa (iza reči *implements* navode se interfejsi razdvojeni zarezima). Sve funkcije iz interfejsa moraju se definisati u klasi koja implementira taj interfejs.

2.5.2. Apstraktne klase

Apstraktne klase su veoma slične interfejsima – koriste se u situacijama kada je potrebno da nasleđene klase implementiraju određeni metod koji nije moguće realizovati u samoj nadklasi.

Apstraktna klasa je klasa koja ima **barem jedan** apstraktni metod. Apstraktni metodi imaju samo naziv i listu parametara u zagradama bez programskog koda u zagradama – taj programski kod se piše u klasama koje naslede apstraktnu klasu. Važno je naglasiti da apstraktne klase mogu imati i funkcije koje nisu apstraktne. Mogu imati čak i definisana polja, ali ona ne mogu biti apstraktna.

Pravljenje apstraktne klase se obavlja pomoću ključne reči *abstract class*. Apstraktne funkcije u okviru te klase se prave takođe pomoću ključne reči *abstract* ispred modifikatora pristupa. Ukoliko se nekoj ne-apstraktnoj klasi doda apstraktna funkcija, i sama klasa se mora proglasiti kao apstraktna.

U narednom primeru, napravljena je klasa *ApstraktniAutomobil* u okviru koje je definisano polje *\$snaga* (koje ne može biti apstraktno), dve apstraktne funkcije i jedna ne-apstraktna.

```
abstract class ApstraktniAutomobil {  
  
    // Zbog jednostavnosti, definisana je početna vrednost za  
snagu  
    private $snaga = 150;  
  
    // Obratiti pažnju da se na kraju definicije funkcije  
    // nalazi ; umesto {}  
  
    abstract public function upaliMotor();  
}
```

```
abstract public function ugasiMotor();

// U apstraktnoj klasi može postojati i ne-apstraktna
funkcija
public function proveraNaga() {
    echo "Naga motora: $this->naga <br>";
}
}
```

Nije moguće napraviti objekat apstraktne klase, ali je zato moguće napraviti novu klasu koja će naslediti apstraktnu klasu. To se radi korišćenjem ključne reči *extends*, a nasleđena klasa mora implementirati sve apstraktne funkcije iz apstraktne klase. U narednom primeru, klasa *TeretnoVozilo* nasleđuje klasu *ApstraktniAutomobil*.

```
class TeretnoVozilo extends ApstraktniAutomobil {

    // Obratiti pažnju da se sada na kraju definicije funkcije
    // nalaze {}

    public function upaliMotor() {
        echo "Upaljen motor...<br>";
    }
    public function ugasiMotor() {
        echo "Ugašen motor...<br>";
    }
}
```

U klasi *TeretnoVozilo* su implementirane apstraktne funkcije iz nadklase. Funkcija *proveraSnaga* je dostupna u podklasi zbog same prirode nasleđivanja. Primer se može testirati pozivanjem sledećih naredbi:

```
$kamion = new TeretnoVozilo();
$kamion->upaliMotor();
$kamion->ugasiMotor();
$kamion->proveraSnaga();
```

Izvršavanjem navedenih naredbi dobija se sledeći ispis na ekranu, čime se jasno vidi da se sve funkcije pozivaju bez problema.

```
Upaljen motor...
Ugašen motor...
Snaga motora: 150
```

2.5.3. Interfejs ili apstraktna klasa

Obzirom na sličnosti između interfejsa i apstraktnih klasa, postavlja se pitanje kako izabrati šta je potrebno koristiti u razvoju aplikacija. Kada je potrebno definisati listu funkcija koje je potrebno implementirati u klasama, koriste se interfejsi. Kada je potrebno definisati listu funkcija koje je potrebno implementirati, ali je pritom potrebno i napraviti i neke konkretne funkcije koje će se koristiti u podklasama, koriste se apstraktne klase.

U situacijama kada i dalje niste sigurni šta da izaberete, dobar pristup je pravljenje interfejsa, a ukoliko ima potrebe, taj interfejs može implementirati apstraktna klasa u kojoj se onda mogu definisati potrebne konkretne funkcije.

2.5.4. Princip rada polimorfizma

Polimorfizam podrazumeva upotrebu apstraktnih klasa ili interfejsa – u ovom slučaju se koristi interfejs, mada bi princip rada bio potpuno isti i sa korićenjem apstraktne klase.

Interfejs *Oblik* definiše pravilo da svaka klasa koja ga bude implementirala mora da definiše funkciju za obračun površine. Klase *Kvadrat* i *Poligon* rade upravo to, a u cilju pojednostavljivanja, funkcija *povrsina()* umesto konkretnog proračuna samo ispisuje poruku kako bi se znalo o kom obliku se radi kada bude pozvana.

```
interface Oblik {
    public function povrsina();
}
```

```

class Kvadrat implements Oblik {
    public function površina() {
        echo "Izračunavanje površine kvadrata <br>";
    }
}

class Poligon implements Oblik {
    public function površina() {
        echo "Izračunavanje površine poligona <br>";
    }
}

```

Naredni korak je pravljenje funkcije *izracunavanjePovrsine* koja se nalazi izvan napravljenih klasa, a koja će biti pozivana kada bude bilo potrebno izračunati površinu nekog oblika. Ova funkcija prihvata parametar tipa *Oblik*, zahvaljujući čemu se može koristiti sa bilo kojim objektom koji je nastao od klase koja implementira interfejs *Oblik*. Upravo ovde se krije moć polimorfizma – kada bude napravljen novi objekat (u ovom slučaju poligon *\$p*) i kada bude prosleđen funkciji za izračunavanje površine kao parametar, biće sačuvan u promenljivoj *\$oblik*.

```

function izracunavanjePovrsine(Oblik $oblik) {
    $oblik->povrsina();
}

$p = new Poligon();
izracunavanjePovrsine($p);

```

Iako na osnovu tipa promenljive *\$oblik* nije moguće reći o kojoj vrsti objekta se radi (*Kvadrat*, *Poligon*), zahvaljujući polimorfizmu, prilikom poziva naredbe *\$oblik->povrsina()*, uvek će biti pozvana funkcija iz odgovarajuće klase. U ovom slučaju, na ekranu će pisati:

```
Izračunavanje površine poligona
```

Upravo zahvaljujući polimorfizmu, moguće je praviti objekte bez potrebe da se vodi računa kog su tačno podtipa – naravno, ukoliko je struktura pravilno organizovana korišćenjem apstraktnih klasa ili interfejsa kao nadređenih elemenata.

Samostalni rad

Kao osnova zadatka se koristi prethodni primer koji se tiče rada sa oblicima. Potrebno ga je izmeniti tako što će se umesto interfejsa koristiti apstraktna klasa. Ova klasa treba da ima sakrivena polja koja predstavljaju jednu stranu geometrijskog oblika i broj strana koje taj oblik poseduje, funkcije za izračunavanje površine i obima, kao i magične funkcije `__get`, `__set` i `__toString`.

Potrebno je napraviti i dve izvedene klase: *Trougao* i *Kvadrat*. Nakon toga, treba testirati primer pomoću funkcije za izračunavanje površine i obima koja kao parametar prihvata tip apstraktne klase koja je korišćena i polimorfno poziva odgovarajuće funkcije.

3. JSON

JSON (JavaScript Object Notation) predstavlja jednostavan format za razmenu podataka koji se piše notacijom *JavaScript*-a. Naravno, to ga ne sprečava da se koristi sa bilo kojim programskim jezikom – moguće je, na primer, generisati *JSON* pomoću *JavaScripta* u veb pretraživaču na osnovu korisničkog unosa, a zatim ga proslediti *PHP* strani koja će dekodirati te podatke i izvršiti neku obradu.

Za razliku od nekih drugih formata i jezika izuzetno je čitljiv za ljude, a *JSON* je moguće i pisati bez korišćenja pomoćnih alata. Sve što je potrebno je običan program za uređivanje teksta. Takođe, zahvaljujući svojoj jednostavnosti, računarska obrada *JSON* zapisa je izuzetno brza. Vrlo jednostavno se određeni podaci mogu pretvoriti u *JSON*, a isto tako je veoma lako sprovesti konverziju i u drugom smeru. Zapravo, u velikom broju slučajeva, nizovi u *PHP*-u će biti od velike koristi u ovim situacijama. Kada se, na primer, učitaju određene podaci iz baze podataka i potrebno ih je poslati dalje (možda kao izlaz nekog veb servisa koji se pravi), prvo će biti izvršena konverzija u niz a zatim u *JSON*. Iako se dodaje jedan korak u proceduru konverzije podataka, zapravo se sam proces puno olakšava jer bi konverzija “sirovih” podataka iz baze podataka u *JSON* bila znatno komplikovanija. U ovom poglavlju, bavićemo se *JSON* formatom i pravilima pisanja, a potom će na praktičnim primerima biti objašnjeno kako se sa njim radi u praktičnim situacijama.

JSON u osnovi predstavlja običan tekstualni format. Sastoji se od dve osnovne strukture:

- *JSON* objekat koji se sastoji od parova atribut-vrednost. Svi programski jezici poznaju ovakvu vrstu strukture i zbog toga lako mogu razumeti ono što je napisano u ovom formatu.
- Uređena kolekcija određenih *JSON* objekata koja ih povezuje na neki način.

3.1. JSON objekti i njihovo dekodiranje

Svaki *JSON* objekat sastoji se određenih parova atribut-vrednost. Prilikom definisanja ovih parova i pravljenja objekata, moraju se ispoštovati sintaksna pravila pravljenja objekata. Sledeći primer prikazuje jednostavan *JSON* objekat.

```
{  
    "hrana": "Breskva",  
    "tip": "Voce"  
}
```

Atributi i vrednosti se pišu pod znacima navoda razdvojeni sa dve tačke. Nakon svakog para atribut-vrednost se navodi zarez, a izuzetak je jedino poslednji par iza koga se ne navodi zarez. Svaki pojedinačni objekat počinje i završava se vitičastim zagradama.

Kako bi ovom objektu bilo moguće pristupiti iz *PHP*-a, potrebno mu je na neki način omogućiti pristup. U narednim primerima, *JSON* će se čuvati u zasebnim fajlovima, a naredni korak će biti distribuiranje preko mreže bez fizičkog čuvanja u fajlu. Za sada, na samom početku, najlakše je sačuvati *JSON* u zasebnoj *PHP* promenljivoj.

```
$podaci = '{  
    "naziv": "Breskva",  
    "tip": "Voce"  
}';
```

Zahvaljujući ovome, *JSON* je zapravo postao *PHP String*. Ukoliko bismo želeli da pristupimo podacima direktno, morali bismo da napišemo novi algoritam kojim bismo *naziv* i *tip* hrane izvukli iz *String*-a. Kako bismo to izbegli, najlakše je *JSON* konvertovati u niz:

```
$hrana = json_decode( $podaci, true );
```

Funkcija *json_decode* vrši konverziju ulaznih *JSON* podataka u *PHP* asocijativni niz. Sada im se može veoma jednostavno pristupiti:

```
echo "Naziv: " . $hrana['naziv'] . "<br>";  
echo "Tip: " . $hrana['tip'] . "<br>";
```

Važno je obratiti pažnju da je drugi parametar funkcije *json_decode* podešen na *true* – to znači da se konverzija vrši u asocijativni niz. Ukoliko je podešen na *false*, biće napravljen objekat. Modifikovana verzija primera zato dobija sledeći oblik:

```
$hrana = json_decode( $podaci, false );  
  
echo "Naziv: " . $hrana->naziv . "<br>";  
echo "Tip: " . $hrana->tip . "<br>";
```

Razlika između dva pristupa je samo u načinu korišćenja, a sami možete odabrati koji Vam više odgovara – pristup pomoću niza ili pomoću objekta.

3.2. JSON kolekcija objekata

U praktičnim primenama, pravljenje samo jednog *JSON* objekta, kao u prethodnom primeru, neće biti dovoljno. Kada se pojavi potreba za pravljenjem više od jednog objekta (npr. više vrsta

hrane), biće važno grupisati ih kao jednu celinu – kolekciju. Pravljenje niza objekata se nameće kao pravo rešenje.

Da bismo demonstrirali tu mogućnost, *JSON* ćemo izdvojiti u poseban dokument sa ekstenzijom *.json* – u ovom slučaju zvaće se *podaci.json*.

```
[  
    {  
        "hrana": "Breskva",  
        "tip": "Voce"  
    },  
    {  
        "hrana": "Jabuka",  
        "tip": "Voce"  
    },  
    {  
        "hrana": "Krompir",  
        "tip": "Povrce"  
    }  
]
```

U ovom primeru imamo tri objekta (tri vrste hrane). Svaki objekat se razdvaja zarezom, a iza poslednjeg objekta se zarez izostavlja. Potom, sve objekte treba grupisati parom uglastih zagrada kojima se naglašava da oni čine elemente jednog niza.

Naredni korak je da se sadržaj dokumenta *podaci.json* učitava iz *PHP* skripta. Za te potrebe, koristi se funkcija *file_get_contents* kojoj se prosleđuje putanja na kojoj se nalazi dokument koji se učitava. Ukoliko se oba dokumenta nalaze u istom folderu, sasvim je dovoljno u promenljivoj *\$adresa* napisati naziv fajla. Ukoliko se ne nalaze u istom folderu, u adresi je potrebno navesti kompletnu putanju na kojoj se nalazi *.json* dokument.

```
$adresa = "podaci.json";  
  
$podaci = file_get_contents( $adresa );
```

```
$hrana = json_decode( $podaci, true );
```

Obtirom da u *.json* dokumentu postoji više objekata u okviru niza, funkcija *json_decode* takođe neće vratiti samo jedan objekat, već niz od tri objekta – a obzirom na vrednost parametra podešenu na *true*, to znači da će i svaki pojedinačni objekat biti konvertovan u niz. To znači da imamo situaciju u kojoj promenljiva *\$hrana* predstavlja niz koji se sastoji od 3 objekta koji su zapravo nizovi – dakle *nizovi u okviru drugog niza – dvodimenzionalni niz*.

Ako bismo želeli da pristupimo drugom objektu iz niza i prikažemo njegov naziv (u ovom slučaju *Jabuka*), potrebno je da napišemo:

```
echo $hrana[1]['naziv'];
```

Ukoliko bismo hteli da prikažemo sve nazive i vrste hrane, direktan pristup koji je upravo prikazan bi zahtevao suviše manuelnog rada. Stoga, biće potrebno napraviti petlju koja će pristupiti svim podacima i ispisati ih na ekranu u okviru tabele. U cilju preglednosti, ispod će biti prikazan objedinjen programski kod ovog primera.

```
<?php
    $adresa = "podaci.json";

    $podaci = file_get_contents( $adresa );

    $hrana = json_decode( $podaci, true );

    echo "<table border=1 >";
    echo "<tr> <th>NAZIV</th> <th>TIP</th> </tr>";

    foreach( $hrana as $objekat ) {
        echo "<tr>";
            echo "<td>" . $objekat['naziv'] . "</td>";
            echo "<td>" . $objekat['tip'] . "</td>";
        echo "</tr>";
    }
}
```

```
echo "</table>";
?>
```

Zahvaljujući tome što je upotrebljena *foreach* petlja, izbegnut je direktan pristup svakom elementu niza, a programski kod ne zavisi od broja objekata koji se mogu pojaviti u *.json* dokumentu. Slika 4 prikazuje kako izgleda izvršenje ovog primera u veb pretraživaču.

NAZIV	TIP
Breskva	Voce
Jabuka	Voce
Krompir	Povrce

Slika 4: Prikaz sadržaja *.json* dokumenta.

Samostalni rad

Dodajte nekoliko novih objekata u *.json* dokument. Nakon što to uradite, namerno napravite neku grešku u *JSON* zapisu (obrišite zarez ili zagradu, stavite zarez tamo gde mu nije mesto). Na ovaj način, videće šta se dešava kada napravite ovakvu vrstu greške i brže ćete moći da nađete problem kada se u budućnosti stvarno javi ta vrsta problema.

Na kraju, napravite modifikaciju ovog primera tako što ćete drugi parametar funkcije *json_decode* promeniti na *false*, potom prilagodite sadržaj *foreach* petlje tako da odražava promenu tog parametra.

3.3. Pravljenje niza objekata u okviru postojećeg objekta

U određenim situacijama, biće potrebno napraviti još složeniju *JSON* strukturu u odnosu na prethodne primere. Razmotrimo

situaciju u kojoj je potrebno napraviti objekat za osobu i omogućiti da svaka osoba može da ima jedno ili više radnih mesta. Naredni dokument se zove *podaci_osobe.json*.

```
[
  {
    "ime": "Ana",
    "posao": [
      {
        "pozicija": "Bankar",
        "plata": "1000"
      },
      {
        "pozicija": "Konsultant",
        "plata": "500"
      }
    ]
  },
  {
    "ime": "Petar",
    "posao": [
      {
        "pozicija": "Glumac",
        "plata": "1500"
      }
    ]
  }
]
```

Prikazani *JSON* kod sadrži niz i dva objekta koji predstavljaju osobe i za koje je u atributu *ime* definisano da se zovu *Ana* i *Petar*. Takođe, svaka osoba ima po još jedan atribut: *posao*. Ovom atributu nije dodeljena samo jedsnotavna tekstualna vrednost kao u slučaju imena, već je napravljen potpuno novi niz, što se vidi po uglastim zagradama. Zahvaljujući tome, za svaka osoba može imati nijedan, jedan ili više od jedan poslova na kojima rade. Svaki od poslova je predstavljen novim objektom koji ima dva atributa: *pozicija* i *plata*.

Ukoliko želimo da na ekranu ispišemo kako se zove radna pozicija osobe koja se zove *Petar*, potrebno je napisati sledeće naredbe:

```
$url = "podaci_osobe.json";  
$data = file_get_contents( $url );  
$osobe = json_decode( $data, true );  
  
echo $osobe[1]['posao'][0]['pozicija'];
```

Prva tri reda rade isto kao i u prethodnom primeru, a ključni je zapravo poslednji red. On pristupa nizu kom se nalaze sve osobe (*\$osobe*), zatim se u prvim uglastim zagradama bira konkretna osoba – obzirom da je *Petar* drugi u nizu, to znači da je potrebno upisati broj 1 (*\$osobe[1]*). Zatim je za izabranu osobu potrebno pristupiti nekom atributu, a to je u ovom slučaju posao (*\$osobe[1]['posao']*). Obzirom da svaka osoba može imati i više poslova i da se oni nalaze smešteni u niz, potrebno je uvesti i treće uglaste zagrade u kojima se definiše koji od poslova za određenu osobu želimo da izaberemo. Obzirom da *Petar* ima samo jedan posao, piše se o (*\$osobe[1]['posao'][0]*). Nakon što je izabran i konkretan posao za određenu osobu, potrebno je pristupiti i određenom atributu za taj posao – u ovom slučaju to je *pozicija* (*\$osobe[1]['posao'][0]['pozicija']*).

Možda pogodna pomoć da bi se izvršila provera da li je prilikom pristupanja određenoj vrednosti otvoren dobar broj parova uglastih zagrada jeste da se izbroji koliko puta su u *JSON* dokumentu otvorene zagrade (i uglaste, i vitičaste!) neposredno pre atributa kom je potrebno primeniti. U navedenom primeru, atributu *pozicija* prethode ukupno četiri otvorene (a da pritom nisu zatvorene) zagrade: 2 uglaste i 2 vitičaste. To je možda i najlakše primetiti na početku dokumenta za *poziciju* koja ima vrednost *Bankar*.

Ukoliko bismo želeli da učitamo sve navedene podatke korišćenjem određene petlje, ali i da prikažemo koliko svaka osoba ukupno zarađuje, to bi bilo moguće uraditi na sledeći način.

```
<?php
```

```

$url = "podaci_osobe.json";
$data = file_get_contents( $url );
$osobe = json_decode( $data, true );

foreach( $osobe as $osoba ) {
    echo "Ime: " . $osoba['ime'] . "<br>";

    $ukupna_zarada = 0;
    foreach( $osoba['posao'] as $posao ) {
        echo "Zaposlen kao " . $posao['pozicija'];
        echo "i ima platu " . $posao['plata'] . "<br>";

        $ukupna_zarada += $posao['plata'];
    }
    echo "UKUPNA ZARADA: $ukupna_zarada <br>";
    echo "-----<br>";
}
?>

```

Obzirom da u dokumentu *podaci_osobe.json* postoji glavni niz (u kom se nalaze sve osobe) i unutrašnji niz za poslove (koji se nalazi u svakoj osobi), potrebno je napraviti dve *foreach* petlje. Spoljašnja petlja pristupa svakoj osobi pojedinačno (*foreach(\$osobe as \$osoba)*), a unutrašnja petlja izlistava sve poslove za po jednu osobu (*\$osoba['posao'] as \$posao*). Svaki posao određene osobe biće dostupan u promenljivoj *\$posao* – sve što je tada potrebno uraditi je pristupiti *poziciji* (*\$posao['pozicija']*) ili *plati* (*\$posao['plata']*).

Nakon što se primer pokrene u veb pretraživaču, dobija se sledeći izlaz:

```

Ime: Ana
Zaposlen kao Bankar i ima platu 1000
Zaposlen kao Konsultant i ima platu 500
UKUPNA ZARADA: 1500
-----
Ime: Petar
Zaposlen kao Glumac i ima platu 1500

```

UKUPNA ZARADA: 1500

Samostalni rad

U *JSON* fajlu pod imenom *podaci_pitanja.json* koji se nalazi ispod sačuvani su tekstovi pitanja, ponuđeni odgovori, kao i označen tačan odgovor na svako pitanje. Potrebno je napisati *PHP* kod koji *JSON* podatke prikazuje u tabeli koja je prikazana na slici 5. Pored ispisa tačnog odgovora u zasebnoj koloni, potrebno je i u koloni *Ponuđeni odgovori* crvenom bojom označiti tačan odgovor. Ispod *.json* fajla sa ulaznim podacima, nalazi se i rešenje zadatka.

Pitanje	Ponuđeni odgovori	Tačan odgovor
10 - 4 = ?	16 6 4 14	6
7 + 5 = ?	12 14 16 2	12
15 + 3 = ?	18 22 9 4	18
15 - 3 = ?	2 12 13 64	12

Slika 5: Izgled tabele sa pitanjima.

podaci_pitanja.json

```
[
  {
    "pitanje": "10 - 4 = ?",
    "ponudjeni_odgovori": [
      "16",
      "6",
      "4",
      "14"
    ]
  }
]
```

```
        "4",
        "14"
    ],
    "tacan_odgovor": "6"
  },
  {
    "pitanje": "7 + 5 = ?",
    "ponudjeni_odgovori": [
      "12",
      "14",
      "16",
      "2"
    ],
    "tacan_odgovor": "12"
  },
  {
    "pitanje": "15 + 3 = ?",
    "ponudjeni_odgovori": [
      "18",
      "22",
      "9",
      "4"
    ],
    "tacan_odgovor": "18"
  },
  {
    "pitanje": "15 - 3 = ?",
    "ponudjeni_odgovori": [
      "2",
      "12",
      "13",
      "64"
    ],
    "tacan_odgovor": "12"
  }
]
```

Rešenje zadatka za samostalan rad

```
<style>
.red {
```

```
        color: red;
        font-weight: 700;
    }
    td {
        text-align: center;
    }
    tr:nth-child(odd) {
        background-color: lightblue;
    }
    th {
        background-color: yellow;
    }
</style>

<?php
    $url = "podaci_pitanja.json";
    $data = file_get_contents($url);
    $pitanja = json_decode($data, true);

    echo "<table border=1 >";
    echo "<tr><th>Pitanje</th><th>Ponudjeni
odgovori</th><th>Tačan odgovor</th></tr>";

    foreach( $pitanja as $pitanje ) {
        echo "<tr>";
        echo "<td>" . $pitanje['pitanje'] . "</td>";
        echo "<td>";

        foreach( $pitanje['ponudjeni_odgovori'] as
$ponudjeno ) {
            if( $ponudjeno ==
$pitanje['tacan_odgovor'] ) {
                echo "<span
class='red'>$ponudjeno</span><br>";
            } else {
                echo $ponudjeno . "<br>";
            }
        }

        echo "</td>";
    }
}
```

```

                                echo "<td>" . $pitanje['tacan_odgovor'] .
"</td>";
                                echo "</tr>";
                                }
                                echo "</table>";
?>

```

3.4. Kodiranje podataka u JSON format

Dosadašnji primeri su koristili gotov *JSON* kod koji je potom dekodiran pomoću funkcije *json_decode*. U stvarnim situacijama, *JSON* sigurno neće biti pisan ručno, već će biti generisan na osnovu podataka koji su, na primer, sačuvani u bazi podaka ili predstavljaju direktan korisnički unos. Tim podacima ćemo u *PHP*-u obično baratati pomoću nekog niza, a upravo je niz struktura podataka koja se lako kodira u *JSON* format. Za primer možemo uzeti naredni asocijativni niz:

```

$podaci = [
    "ime" => "Petar",
    "pozicija" => "Glumac",
    "plata" => 1500
];

```

Njegovo kodiranje u *JSON* obavljammo pomoću funkcije *json_encode* na sledeći način:

```
echo json_encode( $podaci );
```

Nakon pokretanja u pretraživaču, ispisuje se sledeći tekst u formatu *JSON*:

```
{"ime":"Petar","pozicija":"Glumac","plata":1500}
```

Kao što se vidi, vrlo jednostavno je generisan potpuno ispravan *JSON* zapis koji se može dalje prosleđivati i dekodirati, korišćenjem *PHP*-a ili nekog drugog programskog jezika. Štaviše, ne postoji

ograničenje da se samo iz niza može vršiti kodiranje u *JSON* – kao polazna osnova mogu se koristiti čak i *PHP* objekti. Naredni primer (*server.php*) demonstrira kako se vrši ta vrsta kodiranja.

server.php

```
<?php
    // klasa
    class Osoba {
        public $ime;
        public $posao;
        public $plata;
    }

    // pravljenje objekta klase Osoba
    $petar = new Osoba();
    $petar->ime = "Petar";
    $petar->posao = "Glumac";
    $petar->plata = 1500;

    // kodiranje objekta u JSON
    $json_izlaz = json_encode( $petar );
    echo $json_izlaz;
?>
```

U ovom primeru definisana je klasa *Osoba*, a potom je napravljen i jedan objekat klase *Osoba* pod nazivom *\$petar* kome su onda dodeljene i konkretne vrednosti za *ime*, *posao* i *platu*. Krajnji korak je korišćenje funkcije *json_encode* kojoj je prosleđen napravljeni objekat kako bi bio generisan *JSON* zapis koji se potom i ispisuje na ekranu:

```
{"ime":"Petar","posao":"Glumac","plata":1500}
```

JSON ispis koji je rezultat ovo primera ne mora isključivo da bude ispis na ekranu u ovom obliku. Moguće je *JSON* prihvatiti pomoću *JavaScript*-a, izvršiti dekodiranje, a zatim ga grafički urediti i prikazati na ekranu. Krajnji korisnik čak ne mora ni da zna da je u pozadini pozvana *PHP* strana koja je obavila kodiranje objekta i vratila *JSON* zapis. Upravo ovo je uvod u narednu sekciju

u kojoj će biti napravljena strana koja će pomoću *JavaScript*-a pristupiti podacima.

3.5. Prihvatanje i dekodiranje JSON podataka pomoću JavaScript-a

U prethodnom primeru napravljena je *PHP* strana koja vrši kodiranje *JSON*-a i ispisuje ga na ekranu (nalik nekoj vrsti odašiljača). U ovoj sekciji biće napravljena strana (*prijemnik.php*) koja korišćenjem *JavaScript*a prihvata pomenuti *JSON*, dekodira ga i prikazuje podatke na strani.

prijemnik.php

```
<body>
  <h3>Dekodiranje JSON podataka</h3>
  <p id="osobainfo"></p>
</body>

<script>
  var xmlhttp = new XMLHttpRequest();

  // osluškivač događaja
  xmlhttp.onreadystatechange = function() {

    if( this.readyState==4 && this.status == 200 ) {

      osoba = JSON.parse( this.responseText );

      document.getElementById('osobainfo').innerHTML=osoba.ime;
      document.getElementById('osobainfo').innerHTML+="<br>";

      document.getElementById('osobainfo').innerHTML+=osoba.pozadina;
      document.getElementById('osobainfo').innerHTML+="<br>";

      document.getElementById('osobainfo').innerHTML+=osoba.plata;
    }
  }
}
```

```
};  
xmlhttp.open( "GET", "server.php", true );  
xmlhttp.send();  
</script>
```

U okviru *body* elementa se nalazi pasus kome je dodeljen *id* kako bi mu bilo moguće pristupiti kroz *JavaScript* i ubrizgati podatke koji su dobijeni sa strane *server.php*. U okviru sekcije *script* se prvo pravi objekat tipa *XMLHttpRequest*. Ovaj objekat nudi mogućnost slanja zahteva i prihvatanja odgovora sa servera. Ono što je najvažnije, ta komunikacija (slanje i prihvatanje zahteva i podataka) se odvija *bez* ponovnog osvežavanja strane. To znači i da nakon što je učitani sadržaj strane, korišćenjem *XMLHttpRequest*-a se na stranu mogu ubrizgati novi podaci. U slučaju primera iznad, pasus čiji *id* ima vrednost *osobainfo* je u trenutku učitavanja strane prazan, a naknadno će mu, nakon izvršenja *JavaScript*-a i učitavanja *JSON* podataka, biti prosleđen sadržaj koji će se prikazati na ekranu, a kao deo tog pasusa. Ovakav zahtev drugačije nazivamo i *AJAX* (*Asynchronous JavaScript and XML*) zahtev.

Nakon pravljenja objekta *xmlhttp*, dodeljujemo mu osluškivač događaja *onreadystatechange* – ovaj osluškivač se aktivira svaki put kada dođe do promene stanja atributa *readyState* objekta *xmlhttp*. Anonimna funkcija koja je dodeljena ovom osluškivaču događaja u *if* naredbi prati vrednosti *readyState* atributa, ali i atributa *status*. Ovi atributi imaju unapred definisane vrednosti.

Vrednosti koje može da ima promenljiva *readyState*:

- 0 – zahtev (*request*) još nije inicijalizovan;
- 1 – uspostavljena je konekcija ka serveru;
- 2 – zahtev je primljen od strane servera;
- 3 – obrada zahteva je u toku;
- 4 – obrada zahteva je završena, odgovor (*response*) je spreman.

Neke od vrednosti koje može da ima promenljiva *status* (više o statusnim porukama pročitajte u poglavlju 5) su:

- 200 – *OK*;
- 404 – *Strana nije pronađena*.

Kao što se vidi u naredbi *if*, potrebno je da *readyState* ima vrednost 4, kao i da *status* bude 200. To znači da se *JSON* podacima pristupa samo ako je “obrada zahteva završena, a odgovor (*response*) je spreman” i ako je kao status vraćena vrednost “OK”. Ukoliko su ispunjena ova dva uslova, znači da nije došlo do bilo kakvog problema i da se *JSON* podaci mogu preuzeti i učitati. To se radi pomoću funkcije *JSON.parse* koja dekodirane podatke čuva u promenljivoj *osoba* – ovoj funkciji se prosleđuje polje *responseText* – ovo polje je automatski popunjeno odgovorom koji je vratio server – u slučaju ovog primera, to će biti *JSON* tekst koji je pomoću naredbe *echo* ispisana na strani *server.php*.

Naredni korak je pristupanje pasusu *osobainfo* pomoću *JavaScript* funkcije *document.getElementById* koja omogućava da pristupimo nekom elementu *html* strane i da modifikujemo njegov sadržaj bez ponovnog učitavanja cele strane. Ovoj funkciji prosleđujemo podatke o osobi: *osoba.ime*, *osoba.posao* i *osoba.plata*.

Poslednji korak nakon definisanja osluškivača događaja jeste otvaranje konekcije ka serveru korišćenjem funkcije *open* i slanje zahteva pomoću funkcije *send*. Funkciji *open* su prosleđena tri parametra: prvi predstavlja *http* metod koji se koristi (*GET*, *POST*, *PUT*, itd.), drugi adresu na kojoj se nalazi strana koja se poziva, a treći označava da li slanje zahteva treba da se obavlja asinhrono – ukoliko bi bio podešen na *false*, zahtev bio sinhronizovan što je u koliziji sa onim što je potrebno postići, a to je pravljenje asinhronog zahteva (kao što je navedeno, koristimo *AJAX* u čijem nazivu čak postoji reč *asinhrono*). Dakle, treći parametar bi uvek trebao da bude postavljen na *true*.

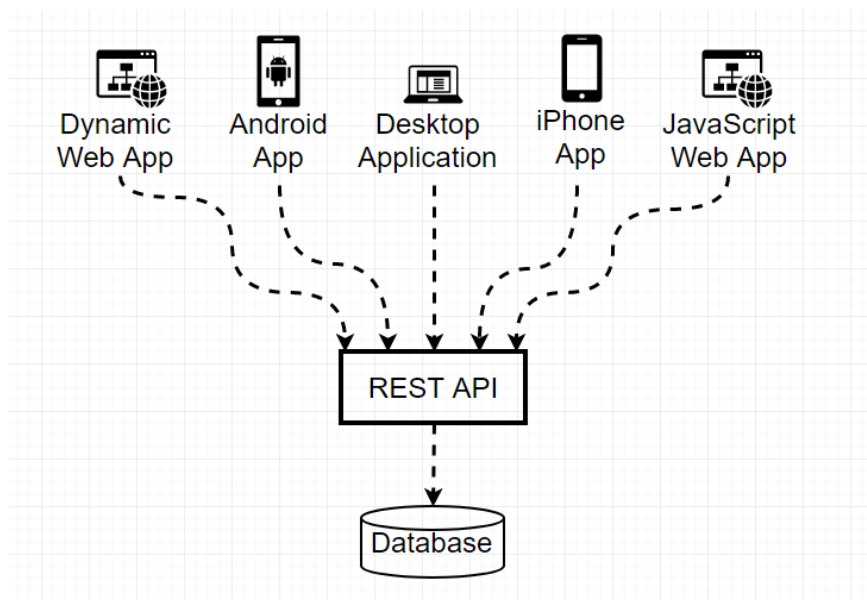
Samostalni rad

Strana *prijemnik.php* iz prethodnog primera odmah nakon učitavanja pokreće *JavaScript* naredbe i prikazuje podatke dobijene sa strane *server.php* u okviru pasusa *osobainfo*. Modifikujte ovaj primer tako što ćete na strani *prijemnik.php* napraviti dugme koje će izvršiti pomenuti *JavaScript* kod – dakle, *JavaScript* ne treba da se poziva automatski prilikom učitavanja, već isključivo na korisnikov zahtev. Tek nakon što se klikne dugme, podaci bi trebalo da se učitaju u pasus. Za ispunjenje ovog zahteva, biće potrebno da napravite *JS* funkciju.

4. REST API

REST (*REpresentational State Transfer*) je stil softverske arhitekture, a često ga pogrešno označavaju kao protokol - zapravo, REST je princip razvoja softvera koji se zasniva na korišćenju protokola HTTP. Ceo koncept je zamišljen na korišćenju jednostavnih formata za opisivanje podataka, a kao način za identifikovanje resursa koriste se URL-ovi (*Uniform Resource Locator*).

Slika 6 prikazuje način na koji *REST API (Application Programming Interface)* sakriva detalje realizacije sistema. REST API je interfejs koji dva računarska sistema koriste da bezbedno razmenjuju informacije preko mreže. U slučaju koji prikazuje slika 6 različite vrste uređaja (veb aplikacije, aplikacije za mobilne uređaje itd.) pristupaju interfejsu (*REST API*) i komuniciraju sa njim, ali je ostatak sistema (u ovom slučaju baza podataka) sakrivena od krajnjih korisnika.



Slika 6: REST API.

Izvor: happycoding.io

Da bi se određeni interfejs zvao *REST API* (ili *RESTful API*, kako se često označava), mora ispuniti određena ograničenja koja propisuje *REST* arhitektura (više o ograničenjima u narednim sekcijama).

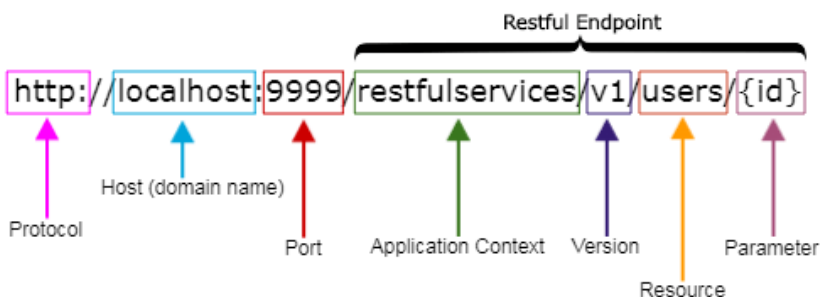
REST se u često povezuje i sa mikroservisima i predstavlja važan arhitekturni stil za njihovo funkcionisanje, zahvaljujući jednostavnosti interfejsa, fleksibilnosti i skalabilnosti zbog kojih se ponaša kao odlična veza između mikroservisa. Jedna od najvećih prednosti korišćenja *REST*-a za mikroservise je što omogućava komunikaciju između mikroservisa bez potrebe da servisi imaju znanje o internim karakteristikama drugih servisa. Ukoliko pogledamo zahteve koje bi tehnologije koje se koriste za komunikaciju između mikroservisa trebalo da ispune, videćemo

odličan dokaz zašto je REST tako dobar izbor – zahtevi za izbor tehnologije za komunikaciju su:²

- *kompatibilnost unazad* prilikom uvođenja novih funkcionalnosti ili delova sistema;
- *eksplicitan interfejs* pomoću koga konzumenti pristupaju mikroservisu – REST podrazumeva da prilikom poziva određenog resursa znamo kako mu se tačno pristupa;
- *korišćenje raznovrskih tehnologija* u izgradnji mikroservisa podrazumeva da različiti servisi mogu biti napravljeni korišćenjem različitih tehnologija. REST će predstavljati jezik koji razumeju sve komponente sistema, nezvezano od tehnologije.
- *jednostavnost servisa za konzumente* podrazumeva lako korišćenje bez obzira na korišćenu tehnologiju tehnologiju – *REST* se koristi da pojednostavi komunikaciju.
- *sakrivanje detalja implementacije* – interna implementacija mikroservisa ne sme da utiče na druge mikroservise.

Da bi se *REST* koristio za komunikaciju između servisa, potrebno je definisati konvencije imenovanja resursa u okviru *URL*-a kojim se vrši pozivanje. Slika 7 prikazuje kako može da izgleda jedna adresa kojom se pristupa određenom resursu – u ovom slučaju prikazuje se korisnik čiji je *id broj* unet umesto *{id}*.

² Newman S., Izgradnja mikroservisa – dizajn sitno granuliranih sistema, O'Reilly, prevod CET, Beograd, 2022.



Slika 7: Struktura REST URL-a.

Izvor: medium.com

Prvi deo adrese zaključno sa portom je intuitivan i zavisiće od konkretne primene. *Application Context* je zapravo naziv aplikacije u okviru koje želimo da pristupimo određenim resursima. Polje *Version* nije obavezno i može se preskočiti. Polje *Resource* bi trebalo da jasno označi o kakvoj vrsti resursa se radi. Parametar koji se unosi nakon imena resursa (često je to *id*), predstavlja jedinstvenu identifikaciju resursa kom je potrebno pristupiti.

4.1. Pravila imenovanja REST resursa

Postoje određena pravila imenovanja REST resursa, a neka od najvažnijih su predstavljena u daljem tekstu.

4.1.1. Korišćenje imenica za predstavljanje resursa

Za predstavljanje resursa treba koristiti imenice, a ne glagole. Dobri primeri su:

```
http://aplikacija.com/radnici
http://aplikacija.com/knjige
http://aplikacija.com/knjige/6
```

Loši primeri su:

```
http://aplikacija.com/prikaziRadnike  
http://aplikacija.com/brisanjeKnjige/6
```

4.1.2. Korišćenje množine za predstavljanje resursa

Za predstavljanje resursa, pogodnije je koristiti množinu imenica, osim u specifičnim slučajevima kada se radi o resursu za koji ne postoji više od jednog primerku (singleton). Dobri primeri su:

```
http://aplikacija.com/knjige/6  
http://aplikacija.com/radnici/123/plata
```

Obratite pažnju da su *knjige* i *radnici* u množini jer se radi o entitetima kojih sigurno ima više od jednog, za razliku od vrednosti *plata* – pojedinačni radnik ima samo jedno polje za unos plate i zato se ono piše u jednini.

Loši primeri su:

```
http://aplikacija.com/radnik  
http://aplikacija.com/radnik/123/ime  
http://aplikacija.com/knjiga/6/naslovi
```

4.1.3. Korišćenje kose crte (/) za definisanje hijerarhije

Kosa crta (/) se koristi za ukazivanje na vezu između resursa. Potrebno je ići od opštije kategorija ka pojedinačnim objektima. Dobar primer takvog url-a je:

```
http://aplikacija.com/fabrika2/sektor1/tim4/radnici/5
```

Adresa kao u prethodnom primeru bi mogla da se koristi za prikazivanje podataka o radniku koji radi u *fabrici 2*, *sektoru 1*, *timu broj 4*, a čiji *id* u sistemu ima vrednost *5*.

4.1.4. Korišćenje srednje crte (-) za poboljšanje čitljivosti url-ova

Ukoliko je potrebno definisati duža imena resursa, može se koristiti srednja crta (-) za poboljšavanje čitljivosti:

`http://aplikacija.com/fabrika-valjevo/sektor-odrzavanja/tim-za-servis`

Napomena: iako nije zabranjeno korišćenje donje crte (_), iz razloga čitljivosti ga je bolje izbegavati i umesto toga koristiti srednju crtu (-).

4.1.5. Korišćenje malih i velikih slova

I mala i velika slova se mogu slobodno koristiti u *REST* adresama, ali bi zbog lakše čitljivosti i sprečavanja grešaka najbolje bilo koristiti samo mala slova. Ako pogledamo sledeći primer:

`http://aplikacija.com/knjige/6`
`http://aplikacija.com/Knjige/6`

iako je raspored slova isti, zbog upotrebe malog i velikog slova *K*, dva navedena linka su zapravo adrese do potpuno različitih resursa.

4.1.6. Korišćenje ekstenzija u adresi resursa

Za definisanje *REST* adrese resursa *ne treba* koristiti ekstenzije datoteka, jer negativno utiču na izgled linka, a kao što će se videti u poglavlju 5, nisu ni potrebne za pristup resursima. Primer loše definisane adrese se nalazi ispod:

`http://aplikacija.com/radnici.php`

4.1.7. Korišćenje CRUD funkcija u nazivu resursa

Prilikom definisanja *REST URL*-a ne treba koristiti imena *CRUD* funkcija – *Create, Read, Update, Delete*. Loši primeri su:

<code>http://aplikacija.com/radnici-dodavanje</code> <code>http://aplikacija.com/radnici-brisanje/5</code>

Sasvim je dovoljno samo definisati resurs kao u primeru ispod

<code>http://aplikacija.com/radnici</code>
--

a potom korišćenjem *HTTP* metoda GET, POST, PUT, DELETE itd. definisati koju operaciju je potrebno obaviti.

U poglavlju 5 koje se bavi praktičnom implementacijom *REST*-a, detaljno je na praktičnom primeru opisano kako se koriste *REST URL*-ovi uz različite *HTTP* metode.

4.2. Ograničenja REST arhitekture

REST arhitektura opisuje šest ograničenja³:

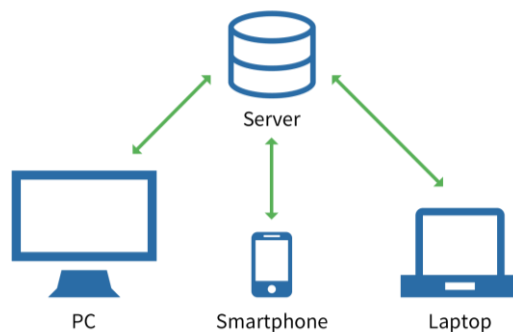
1. **Klijent-server** se zasniva na razdvajanju zavisnosti korisničkog interfejsa od zavisnosti čuvanja podataka (slika 8). Ovakvim razdvajanjem poboljšava se skalabilnost cele aplikacije, što je pogotovo važno za veb okruženje – na ovaj način, komponente mogu da se unapređuju zasebno uz minimizovanje uticaja na druge delove sistema. Omogućava

³ Više detalja možete pronaći u doktorskoj disertaciji u kojoj su postavljene osnove REST arhitekture: Fielding, R. T, Architectural Styles and the Design of Network-based Software Architectures, Dissertation, University of California, Irvine, 2000.

Dostupno na

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

se da klijentska komponenta može da šalje zahteve (*request*), dok serverska komponenta te zahteve prima. Nakon obrade zahteva, server klijentu može poslati i odgovor (*response*).



Slika 8: Model klijent-server.

Izvor: techterms.com

2. **Bez čuvanja stanja (stateless)** – komunikacija zasnovana na REST servisima po prirodi ne može čuvati stanje – svaki zahtev poslat od klijenta ka serveru mora sadržati kompletan skup informacija potreban da se zahtev razume – nije moguće čuvati podatke o kontekstu aplikacije na serveru. Stanje korisničke sesije se u potpunosti čuva isključivo na klijentskoj strani. Zahvaljujući ovakvom načinu rada, server obrađuje svaki klijentov zahtev nezavisno od prethodnih zahteva – klijenti mogu zahtevati resurse u bilo kom redosledu, a svaki zahtev će, bez pamćenja stanja, biti izolovan od drugih zahteva.

3. **Keširanje** – za podatke u okviru odgovora koji se šalje klijentu mora biti jasno označeno da li se mogu keširati (*cacheable*) ili ne (*non-cacheable*). Ako je označeno da se odgovor može keširati, klijent dobija pravo da podatke iz tog odgovora sačuva (kešira) kako bi ih ponovo upotrebio za

kasnije ponovno slanje zahteva iste vrste. Zahvaljujući ograničenjima zasnovanim na keširanju stvarase potencijal da se određene interakcije potpuno eliminišu čime se poboljšava mrežna efikasnost, skalabilnost i percepcija performansi od strane krajnjeg korisnika. Sa druge strane, negativna strana keširanja se ogleda u smanjivanju pouzdanosti ako su keširani podaci suviše zastareli i previše se razlikuju od podataka koji bi bili dobijeni da je zahtev poslat direktno ka serveru.

4. **Uniformni interfejs** – važno svojstvo *REST* arhitekture je postojanje uniformnog intefejsa između različitih komponenti. Na ovaj način se pojednostavljuje arhitektura sistema, a vidljivost interakcija između komponenti je poboljšana.
5. **Slojeviti sistem** – omogućeno je razvijanje sistema koji se sastoji od slojeva napravljenih na takav način da komponente u okviru jednog sloja ne mogu videti šta se dešava dalje od sloja sa kojim direktno komuniciraju. Na ovaj način se dobijaju mnoge prednosti omogućavanjem nezavisnosti komponenti, podsticanja skalabilnosti sistema i efikasnijeg raspoređivanja opterećenja sistema.
6. **Kod na zahtev (code-on-demand)** – *REST* omogućava proširivanje frontend funkcionalnosti izvršavanjem naredbi u obliku skriptova. Ovakav pristup pojednostavljuje klijentsku stranu smanjivanjem broja funkcionalnosti koje je potrebno unapred definisati. Iako se na taj način sistem naknadno lako proširuje, takav pristup može uticati i na smanjivanje vidljivosti komponenti u okviru sistema. *Kod na zahtev* je jedino opciono ograničenje *REST* arhitekture.

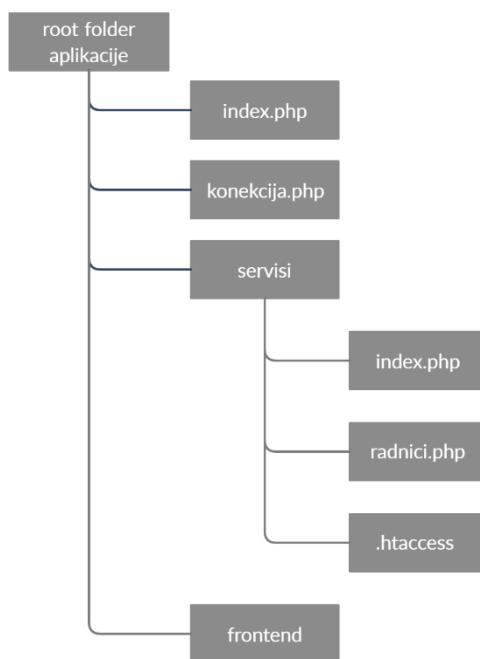
5. RAZVOJ REST APLIKACIJE

Želimo da napravimo test aplikaciju u okviru koje će biti demonstriran rad veb servisa korišćenjem *REST*-a. Iako postoji puno radnih okvira (engl. *framework*) koji sakrivaju kompleksnost realizacije *REST* poziva, razvojem jednog ovakvog primera se može detaljnije uvideti šta se zapravo dešava u pozadini aplikacije. Takođe, akcenat neće biti na frontend-u aplikacije –*HTML*, *CSS* i *JavaScript* se neće puno koristiti, već će se umesto korisničkog dela aplikacije uglavnom koristiti softver *Postman*, o kome će više reči biti kasnije.

U test aplikaciji postoje delovi programskog koda koji mogu da se napišu sažetije ili elegantnije – obzirom da je cilj ovog udžbenika razumevanje rada veb servisa, fokus će biti upravo na njima, uz pisanje što manje koda koji može otežati razumevanje svima koji nemaju dovoljno iskustva u ovoj oblasti.

5.1. Struktura aplikacije

Sama aplikacija je organizovana u dva foldera (*servisi* i *frontend*) i nekoliko pojedinačnih dokumenata (slika 9).



Slika 9: Struktura REST aplikacije.

Ovakvom organizacijom postiže se da je mehanizam za konektovanje na bazu podataka dostupan svim elementima aplikacije (dokument *konekcija.php*), a da je sve vezano za veb servise smešteno u folder *servisi*. Folder *frontend* namenjen je za elemente predviđene za deo aplikacije kome korisnik direktno pristupa (*HTML*, *CSS*, *JS*) i ovaj deo aplikacije nikada ne treba da direktno pristupa podacima (na primer u bazi podataka), već će za to pozivati određeni veb servis iz foldera *servisi*. Za sada, frontend će ostati prazan obzirom da će se za te potrebe koristiti *Postman*.

5.1.1. Važnost strane *index.php*

I u osnovnom folderu i u folderu *servisi* postoji dokument *index.php*. Ovaj dokument zapravo ne mora da ima bilo kakav programski kod (može biti prazan) ili može sadržati naredbu kojom

se korisnik preusmerava na neku drugu stranu (na primer početnu stranu aplikacije, ili stranu za prikaz greške). Preusmeravanje se vrši pomoću *header* funkcije.

Dodavanje strane *index* je važan korak iz razloga što je podrazumevano ponašanje pretraživača da, ukoliko ne postoji *index* strana u folderu, izlista kompletan sadržaj tog foldera krajnjem korisniku (slika 10) – a to je nešto što po svaku cenu treba izbeći. Jednostavnim dodavanjem dokumenta *index.php* u svaki folder sprečavaju se takve situacije i korisnik će ili dobiti beo ekran ili će (što je mnogo bolje) biti preusmeren na neku drugu stranu.

Index of /evidencija/servisi

Name	Last modified	Size	Description
 Parent Directory		-	
 radnici.php	2020-01-12 14:12	3.5K	
Apache/2.4.39 (Win64) PHP/7.2.18 Server at localhost Port 80			

Slika 10: Pozivanje foldera u kom nema index strane.

5.2. Pripremanje baze podataka

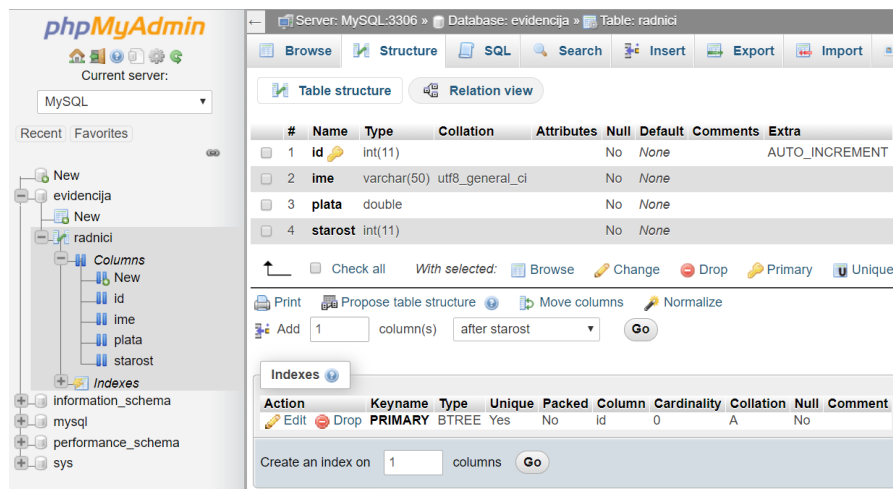
Potrebno je napraviti i bazu podataka koja će se u ovom primeru zvati *evidencija*, a koristiće se MySQL. Moguće je koristiti bilo koji sistem za uređivanje baza podataka, a ukoliko za razvoj koristite okruženja kao što su WAMP, LAMP ili XAMPP, ugrađeni alat *phpmyadmin* će biti od izuzetne koristi.

5.2.1. Pravljenje tabele “radnici”

Nakon pravljenja baze podataka, potrebno je napraviti i tabelu *radnici*. Sve operacije u ovom primeru će koristiti upravo ovu tabelu, a SQL kod za njeno pravljenje se nalazi ispod.

```
CREATE TABLE IF NOT EXISTS `radnici` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ime` varchar(50) NOT NULL,
  `plata` double NOT NULL,
  `starost` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Ukoliko koristite *phpmyadmin*, ove naredbe možete izvršiti na kartici *Query*, a važno je da pre toga, u listi sa leve strane, bude izabrana baza podataka *evidencija* koju ste prethodno napravili. Nakon izvršenja upita, struktura baze podataka će izgledati kao na slici 11.



Slika 11: Struktura baze podataka.

5.2.2. Konekcija ka bazi podataka

U dokumentu *konekcija.php* u osnovnom folderu aplikacije potrebno je napraviti klasu i funkciju koja je zadužena za konekciju ka bazi podataka.

```
<?php
```

```
// dokument konekcija.php
class DBConnection{
    /* Parametri za konekciju */
    var $serverName = "localhost";
    var $username = "root";
    var $password = "";
    var $dbName = "evidencija";
    var $konekcija;
    function poveziSe() {
        $k = mysqli_connect(
            $this->serverName,
            $this->username,
            $this->password,
            $this->dbName) or
            die("Konekcija nije uspela: ".
                mysqli_connect_error());

        if(mysqli_connect_errno()) {
            echo "Problem u povezivanju: ".
                mysqli_connect_error();
            exit();
        } else {
            $this->konekcija = $k;
        }
        return $this->konekcija;
    }
}
?>
```

U ovom dokumentu definisani su parametri za pristup bazi (adresa servera, korisničko ime, lozinka, ime baze podataka). Funkcija *poveziSe* vraća objekat konekcije ka bazi podataka.

5.3. Pravljenje veb servisa

Prvi korak prilikom pravljenja veb servisa jeste definisanje REST pravila.

5.3.1. Definisanje REST pravila

Obzirom da se koristi REST za definisanje pravila pristupanja, potrebno je definisati jedinstvena pravila za pristup resursima, a u ovom slučaju je to tabela sa podacima o radnicima. Sa radnicima je u aplikaciji moguće obavljati pet vrsta operacija i one su prikazane u narednoj tabeli.

Moguće REST adrese i aktivnosti koje se obavljaju njihovim pozivom.

Adresa	Metod	Opis
/radnici	GET	Prikaz liste svih radnika.
/radnici/5	GET	Prikaz podataka o radniku čiji je <i>id</i> 5.
/radnici	POST	Dodavanje novog radnika u bazu podataka. U <i>request</i> -u se šalju i podaci o radniku u <i>JSON</i> formatu. Na primer: { "ime": "Lazar", "plata": "800", "starost": "45" }
/radnici/5	PUT	Izmena podataka o postojećem radniku čiji je <i>id</i> 5 u bazi podataka.

		U <i>request</i> -u se šalju potrebni podaci o radniku u <i>JSON</i> formatu. Na primer: <pre>{ "ime": "Lazar", "plata": "1000", "starost": "45" }</pre>
/radnici/5	DELETE	Brisanje radnika čiji <i>id</i> ima vrednost 5 iz baze podataka.

Kao što ste vidi iz tabele, iz same *REST* adrese je gotovo nemoguće zaključiti koju je adresu potrebno preduzeti. Izabrani metod tu igra izuzetno važnu ulogu jer će upravo on veb servisu dati signal koju akciju zapravo treba preduzeti.

GET metod je podrazumevani metod – ako nije naznačeno koji metod je potrebno koristiti, biće korišćen upravo on. *POST* metod ste verovatno imali prilike da vidite prilikom pravljenja *HTML* formi gde se u *form* tagu često piše *method="POST"*. *PUT* i *DELETE* su metodi sa kojima se možda niste sretali, a o svim ovim metodima biće više reči u narednim sekcijama.

Kako bi uopšte bilo moguće raditi sa *REST* adresama, potrebno je definisati pravila za njihovu konverziju u *.htaccess* dokumentu.

5.3.2. Korišćenje *.htaccess* dokumenta

Obzirom da se za pokretanje test aplikacije koristi veb server *Apache*, potrebno je podesiti i konfiguracioni fajl *.htaccess*. U okviru njega se definišu pravila za korišćenje *REST* veb servisa. Kako bi sve funkcionaisalo ispravno, *.htaccess* fajl je potrebno napraviti u folderu *servisi* (moguće ga je praviti i u drugim folderima, u skladu sa potrebama).

```
RewriteEngine On
RewriteRule ^radnici/?$ radnici.php [NC,L]
RewriteRule ^radnici/([0-9]+)/?$ radnici.php?id=$1 [NC,L]
Options -MultiViews
```

Fajl `.htaccess` se u ovom slučaju koristi za definisanje pravila preimenovanja *REST url*-ova u konkretne *url*-ove u aplikaciji. Definisanje pravila je slično pravilima korišćenja regularnih izraza u nekim jezicima (poput *JavaScript*-a). Obzirom da izučavanje tih pravila izlazi iz opsega ove knjige, dobro mesto za njihovo proučavanje je zvanična dokumentacija na *Apache* veb sajtu.

Stoga, kada korisnik u pretraživaču napiše naredni *REST url*:

```
http://localhost/evidencija/servisi/radnici
```

potrebno je preusmeriti ga na adresu koju razume veb server:

```
http://localhost/evidencija/servisi/radnici.php
```

Po sličnom principu i sledeći *REST url*:

```
http://localhost/evidencija/servisi/radnici/55
```

potrebno je konvertovati u:

```
http://localhost/evidencija/servisi/radnici.php?id=55
```

Navedene *REST url*-ove krajnji korisnik vidi u svom pretraživaču kao adrese kojima pristupa. Konvertovane adrese nisu javno vidljive i koriste se samo da se u okviru aplikacije dođe do određenih resursa.

Korišćenje *REST* adresa predstavlja odličan način da se sakrije realizacija aplikacije jer one ne moraju da odgovaraju strukturi foldera u aplikaciji. U test aplikaciji te izmene nisu velike i svode se

na dodavanje ekstenzije *.php*, ali u realnim situacijama ćete verovatno želeći da *REST* putanje malo više izmenite.

Naredni korak u izradi aplikacije je pravljenje fajla *radnici.php* u kome će se nalaziti kompletna programska logika veb servisa. Na tom mestu će se prihvatati *JSON* ulazni podaci, obavljaće se komunikacija sa bazom podataka i podaci u *JSON* formatu će se slati na prikazivanje u *frontend* aplikacije.

5.3.3. Osnovna struktura veb servisa

Na početku fajla *radnici.php* se prvo pravi konekcija ka bazi podataka, a potom se određuje koji metod je korišćen za pozivanje strane. Ta informacija se čuva u promenljivoj `$_SERVER['REQUEST_METHOD']`, a njena vrednost će biti sačuvana za dalje korišćenje u promenljivoj `$request_method`.

```
<?php
    // Konekcija ka bazi podataka
    include("../konekcija.php"); // include_once();

    $db = new DBConnection();
    $connection = $db->poveziSe();

    // GET, POST, PUT, DELETE
    $request_method = $_SERVER['REQUEST_METHOD'];
```

Naredni korak je pravljenje konstrukcije *switch* koja će na osnovu izabranog metoda sprovesti odgovarajuću akciju (prikaz, dodavanje, izmena itd.).

```
switch($request_method) {
    case 'GET':
        // GET se koristi za preuzimanje podataka o
        radnicima

        if( empty($_GET['id']) ) {
            // Ako nije unet id u REST adresi, to znači
```

```
        // da se učitava lista svih radnika.
        get_radnici();
    } else {
        // Ako je unet id u REST adresi, to znači
        // da se učitava tačno određeni radnik.
        $id = intval($_GET['id']);
        get_radnik($id);
    }
    break;

case 'POST':
    // POST - unos novog korisnika u bazu podataka.
    insert_radnik();
    break;

case 'PUT':
    // PUT - ažuriranje podataka o korisniku.
    $id = intval($_GET['id']);
    update_radnik($id);
    break;

case 'DELETE':
    // DELETE - brisanje radnika iz baze podataka.
    $id = intval($_GET['id']);
    delete_radnik($id);
    break;

default:
    echo "Nepostojeći zahtev.";
    break;
}
```

U samom switch-u se ne nalazi konkretna realizacija, već samo pozivi odgovarajućih funkcija. Takođe, ukoliko postoji, pristupa se i *id*-u iz REST url-a (*\$_GET['id']*), a zatim se vrši njegova konverzija u ceo broj pomoću funkcije *intval*.

U narednim sekcijama biće definisano svih pet potrebnih funkcija koje obavljaju specifične aktivnosti.

5.3.4. Učitavanje svih vrednosti

U slučaju da je potrebno izlistati sve radnike iz baze podataka, poziva se funkcija *get_radnici*. U okviru ove funkcije se definiše i izvršava jednostavan *SELECT* upit čiji se rezultat u while petlji smešta u niz pod nazivom *\$response*. Kao što je već pokazano u poglavlju 3, niz je veoma pogodna struktura podataka za jednostavnu konverziju u *JSON* format.

Nakon što su podaci pripremljeni u while petlji, poziva se funkcija header kojom se definiše tip sadržaja u zaglavlju odgovora (engl. *response*). Na ovaj način se frontendu aplikacije naglašava da će biti isporučen *JSON*. Poslednji korak jeste pozivanje funkcije *json_encode* u cilju konverzije niza *\$response* u *JSON* i njegov ispis na ekranu.

```
function get_radnici() {
    global $connection;
    $query = "SELECT * FROM radnici";
    $response = array();
    $result = mysqli_query($connection, $query);

    while($user=mysqli_fetch_assoc($result)) {
        $response[] = $user;
    }
    header("Content-Type: application/json");
    echo json_encode($response);
}
```

Napomena: U cilju što jednostavnijeg pristupa bazi podataka, objekat *\$connection* je označen kao *global*. To je urađeno kako bi se količina koda za pristup bazi podataka što više smanjila i kako bi suštinska logika za rad veb servisa došla do izražaja.

U praktičnim aplikacijama ili izbegnite korišćenje ključne reči *global*, ili je koristite veoma pažljivo, obzirom da njenom upotrebom nekada mogu nastati problemi koji se teško otkrivaju.

5.3.5. Učitavanje određene vrednosti

Učitavanje podataka o određenom radniku iz baze podataka je veoma slično učitavanju svih radnika uz razliku da se broj rezultata limitira samo na radnika sa određenim *id*-om.

```
function get_radnik($id) {
    global $connection;
    $query = "SELECT * FROM radnici
              WHERE id=".$id." LIMIT 1";

    $response = array();
    $rezultat = mysqli_query($connection, $query);

    while($radnik=mysqli_fetch_assoc($rezultat)) {
        $response[] = $radnik;
    }
    header("Content-Type: application/json");
    echo json_encode($response);
}
```

Jedina razlika u odnosu na funkciju *get_radnici* je što se kao parametar funkciji *get_radnik* prosleđuje *id* željenog radnika i što je upit korigovan u skladu sa tim.

Čak je i *while* petlja ostala ista kao u funkciji *get_radnici*, mada će se ona izvršiti samo jednom. Obzirom na sličnost ove dve funkcije, moguće bi bilo napisati i samo jednu, objedinjenu funkciju.

Napomena: U test aplikaciji, upiti se prave u okviru *string*-a u kom se ručno vrši konkatencija vrednosti poput *\$id*. U realnim aplikacijama, trebalo bi koristiti pripremljene naredbe (engl. *prepared statement*).

5.3.6. Unos nove vrednosti

Unos nove vrednosti je nešto drugačiji u odnosu na proceduru učitavanja vrednosti iz razloga što podaci potrebni za unos novog radnika dolaze u *JSON* formatu.

```

function insert_radnik() {
    global $connection;

    $data = json_decode( file_get_contents('php://input'),
true);
    $ime = $data['ime'];
    $plata = $data['plata'];
    $starost = $data['starost'];

    $query = "INSERT INTO radnici SET ime='".$ime."',
                plata='".$plata."',
                starost='".$starost.'";";

    if( mysqli_query($connection, $query) ) {
        $response = array(
            'status' => 1,
            'poruka' => 'Radnik je uspešno dodat.'
        );
    } else {
        $response = array(
            'status' => 0,
            'poruka' => 'Dodavanje radnika nije uspelo.'
        );
    }
    header("Content-Type: application/json");
    echo json_encode($response);
}

```

JSON podacima se na samom početku funkcije *insert_radnik* pristupa pomoću naredbe:

```

json_decode( file_get_contents('php://input'), true);

```

Ova naredba pristupa ulaznim podacima (*JSON*) pomoću funkcije *file_get_contents*, a zatim vrši njihovu konverziju u asocijativni niz. Dalje se podaci učitavaju iz niza u zasebne promenljive (*\$ime*, *\$plata*, *\$starost*), a potom se izvršava *INSERT* upit. Potrebno je obratiti pažnju da se u upitu ne definiše *id* novog korisnika – polje

id je prilikom pravljenja tabele označeno kao *auto increment* i biće automatski popunjeno.

Nakon izvršavanja upita, u *if* naredbi se proverava da li je dodavanje u bazu podataka uspelo. Odgovor (bio on pozitivan ili ne) se čuva u asocijativnom nizu *\$response* u vidu statusnog koda i statusne poruke. Potom se poziva funkcija *header* i generiše *JSON* na osnovu niza *\$response*.

Dakle, izlazne poruke funkcije su takođe u *JSON* formatu kao i ulazni podaci, obzirom da je u duhu veb servisa da se *JSON* koristi za kompletnu razmenu podataka sa *frontend*-om aplikacije.

5.3.7. Izmena postojećih vrednosti

Funkcija *update_radnik* za izmenu podataka o postojećem radniku prihvata *id* kao i funkcije zasnovane na *GET* metodu, ali i preuzima *JSON* ulazne podatke kao i funkcija za dodavanje novog radnika.

Programski kod ove funkcije je sličan prethodnim funkcijama uz drugačiji upit.

```
function update_radnik($id) {
    global $connection;

    $data = json_decode( file_get_contents('php://input'),
true);
    $ime = $data['ime'];
    $plata = $data['plata'];
    $starost = $data['starost'];

    $query = "UPDATE radnici SET ime=" . $ime .
        " , plata=" . $plata .
        " , starost=" . $starost .
        " WHERE id=" . $id . ";" ;

    if( mysqli_query($connection, $query) ) {
        $response = array(
            'status' => 1,
```

```

        'poruka' => 'Podaci o radniku uspešno
ažurirani.'
    );
} else {
    $response = array(
        'status' => 0,
        'poruka' => 'Ažuriranje nije uspešno.'
    );
}
header("Content-Type: application/json");
echo json_encode($response);
}

```

5.3.8. Brisanje postojećih vrednosti

Brisanje radnika iz tabele *radnici* vrši se pomoću *DELETE* upita, a na osnovu *id*-a koji je prosleđen. Nakon izvršenja upita, proverava se da li je operacija uspešna nakon čega se u *frontend* aplikacije šalje odgovarajuća poruka.

```

function delete_radnik($id) {
    global $connection;

    $query = "DELETE from radnici WHERE id=".$id;

    if( mysqli_query($connection, $query) ) {
        $response = array(
            'status' => 1,
            'poruka' => 'Radnik uspešno obrisano.'
        );
    } else {
        $response = array(
            'status' => 0,
            'poruka' => 'Brisanje radnika nije uspešno.'
        );
    }
    header("Content-Type: application/json");
    echo json_encode($response);
}

```

5.4. Testiranje aplikacije

Poslednji korak koji je preostao je testiranje aplikacije. Obzirom da u bazi podataka trenutno nema podataka, za početak bi dobro bilo dodati jednog ili više radnika pomoću aplikacije *phpmyadmin*. Obzirom da trenutno ne postoji frontend, veb pretraživač bi se mogao koristiti samo za testiranje REST adresa koje ne zahtevaju slanje *JSON* podataka, kao što su GET i DELETE.

Adresa koju bu trebalo kucati u pretraživaču bi u većini slučajeva trebalo da bude u obliku:

localhost/evidencija/servisi/radnici

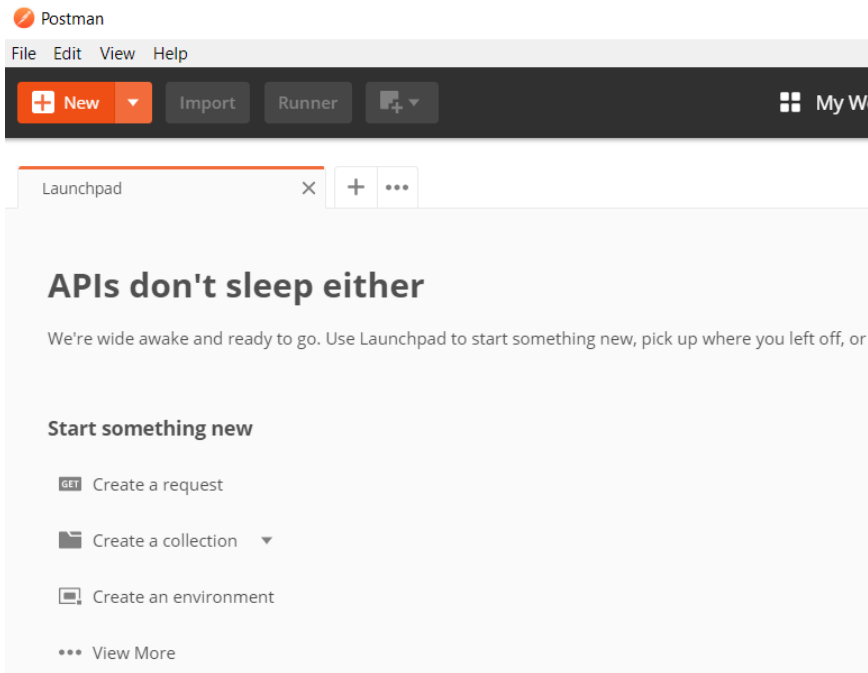
Naziv *evidencija* predstavlja naziv osnovnog direktorijuma aplikacije – obratite pažnju da li ste ga tako nazvali. Kada se ovaj link pokrene u pretraživaču, dobiće se izgled sličan onom na slici 12.

```
[{"id": "1", "ime": "Petar", "plata": "1000", "starost": "35"},  
{"id": "2", "ime": "Lazar", "plata": "1000", "starost": "45"}]
```

Slika 12: Pokretanje REST zahteva za prikaz svih korisnika.

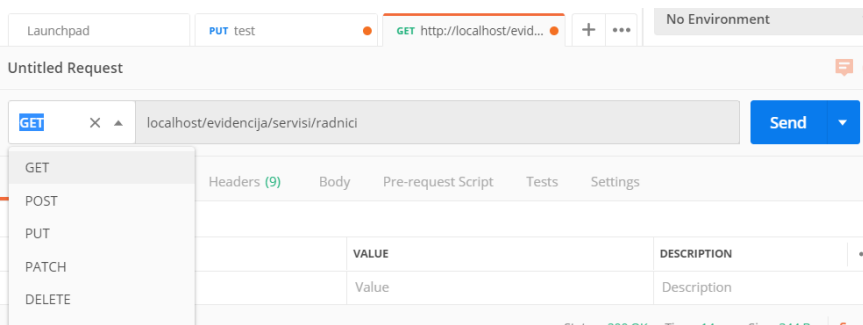
Kao što se vidi, izlaz su zaista podaci u *JSON* formatu, mada njihov izgled u pretraživaču nije baš najpregledniji (mada se mogu instalirati dodaci za formatiranje *JSON*-a), pogotovo kada se uzme u obzir da ih može biti puno. To je još jedan razlog za korišćenje *Postman*-a koji će uz sve prednosti koje nudi, rezultate lepo i formatirati.

Postman je besplatno dostupan na vebu, a nakon instalacije i pokretanja, otvara se početni ekran (slika 13).



Slika 13: Početni ekran u Postman-u.

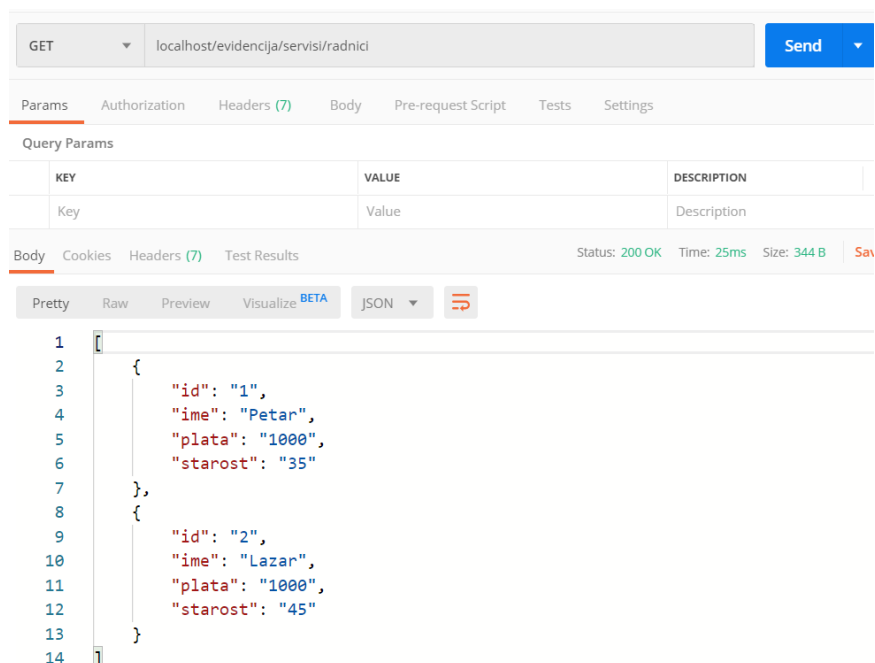
Naredni korak je pravljenje novog zahteva. Moguće je kliknuti na dugme *New* u gornjem levom ili direktno izabrati opciju *Create a request*. Nakon toga, otvara se novi ekran (slika 14).



Slika 14: Izbor metoda i unos REST URL-a u Postman-u.

U padajućoj listi sa leve strane se bira metod koji se koristi (zapravo ih ima znatno više od broja koji se koristi u test aplikaciji), a pored padajuće liste se nalazi polje za unos *REST* adrese. Sa desne strane se nalazi dugme *Send* kojim se šalje zahtev.

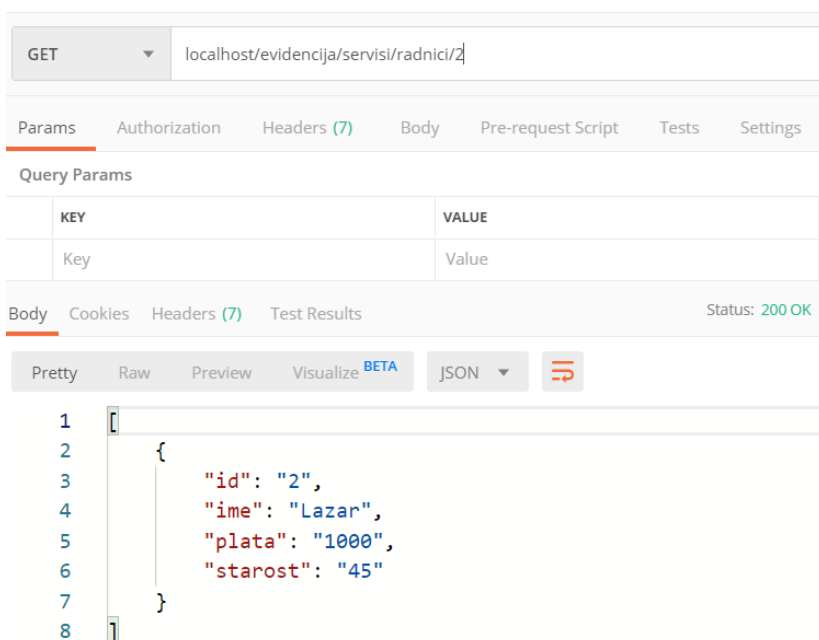
Kada se za metod izabere *GET*, a kao *REST* adresa se unese ista ona koja je malo ranije testirana u pretraživaču, dobija se prikaz sa slike 15.



Slika 15: Prikaz liste svih radnika.

Dakle, radi se o potpuno istim podacima koji su već prikazani u pretraživaču, samo je *JSON* prikaz mnogo jasnije formatiran.

Naredni test je provera prikazivanja samo jednog radnika. Na prethodnom ekranu se vidi da, na primer, jedan od radnika ima id sa vrednošću 2. Kada u *Postman*-u izmenimo *REST* url tako da dodamo broj 2 na kraju i potom ponovo pošaljemo zahtev, dobija se prikaz sa samo jednim rezultatom (slika 16).



Slika 16: Prikaz podataka o radniku čiji id ima vrednost 2.

Nakon sprovedenog testiranja učitavanja podataka iz baze podataka, potrebno je proveriti i kako se dodaju podaci. U Postman-u je potrebno izabrati POST metod i uneti identičnu adresu kao i za prikaz svih korisnika:

localhost/evidencija/servisi/radnici

Samostalni rad

U napravljenoj aplikaciji se koriste *REST* adrese za obavljanje određenih funkcija. Za neke pozive kao što su izmena ili brisanje, u samoj adresi se prosleđuje *id* radnika – na primer, poziva se metod *delete*, a link ima oblik: */radnici/5*.

Potencijalni problem koji se javlja jeste da krajnjeg korisnika ništa ne sprečava da broj 5 promeni u bilo koji drugi broj i samim tim obriše i radnike koji ne bi smeli da se brišu. Rešenje ovog problema jeste da se i id radnika šalje uz *JSON* podatke u *request*-u.

Potrebno je prepraviti aplikaciju na način da se *PUT* i *DELETE* zahtevi izvršavaju bez direktnog unosa *id*-a u *REST url*-u.

5.5. Frontend aplikacije

Postman je odličan alat za testiranje rada servisa koji ste napravili, ali ga u praksi krajnji korisnici neće koristiti. Za njih će biti napravljen interfejs (*frontend* aplikacije) koji će obavljati potrebne pozive i prikazivati rezultat na ekranu. Trenutno postoji izuzetno veliki broj mogućnosti za izradu frontend dela aplikacije, a u cilju prikazivanja što raznovrsnijih primera u ovom udžbeniku, u narednom primeru ćemo veb servis pozvati sa jednostavne *html* strane korišćenjem *JavaScript*-a i *GET* metoda, a na način da prilikom učitavanja podataka sa servera strana neće biti ponovo učitavana. Naravno, izbor tehnologija koja ćete koristiti u realnim aplikacijama ostaje na Vama.

Prvi korak je pravljenje jednostavne *html* strane:

```
<div id="root"></div>

<script src="scripts.js"></script>

<input type="button" onclick="prikaz()" value="Prikaži radnike"
/>
```

Klikom na dugme *Prikaži radnike* biće pozvana funkcija *prikaz()* iz dokumenta *scripts.js*. Funkcija *prikaz()* je zapravo mesto na kom će se odigrati poziv REST servisa korišćenjem *GET* metoda – nakon što ovaj servis vrati rezultate, oni će biti obrađeni i prikazani na *html* strani (bez ponovnog učitavanja strane) u okviru *div*

komponente sa *id* oznakom *root*. Sadržaj funkcije *prikaz()* je naveden u nastavku.

```
function prikaz() {

    // Brišemo prethodni sadržaj (ukoliko postoji)
    // div elementa sa id-om root
    document.getElementById("root").innerHTML = "";

    const app = document.getElementById("root");

    const container = document.createElement('div');
    container.setAttribute('class', 'containerClass');

    app.appendChild(container);

    var request = new XMLHttpRequest();
    request.open('GET',
        'http://localhost/evidencija/servisi/radnici',
        true);

    request.onload = function () {

        // Pristup JSON podacima
        var data = JSON.parse(this.response);
        if (request.status >= 200 && request.status < 400) {
            data.forEach(radnik => {
                const karticaRadnika = document.createElement('div');
                card.setAttribute('class', 'karticaRadnika');

                const h1 = document.createElement('h1');
                h1.textContent = radnik.ime;

                const p = document.createElement('p');
                p.textContent = `Plata: ${radnik.plata}`;

                const p2 = document.createElement('p');
                p2.textContent = `Starost: ${radnik.starost}`;

                container.appendChild(karticaRadnika);
                karticaRadnika.appendChild(h1);
```

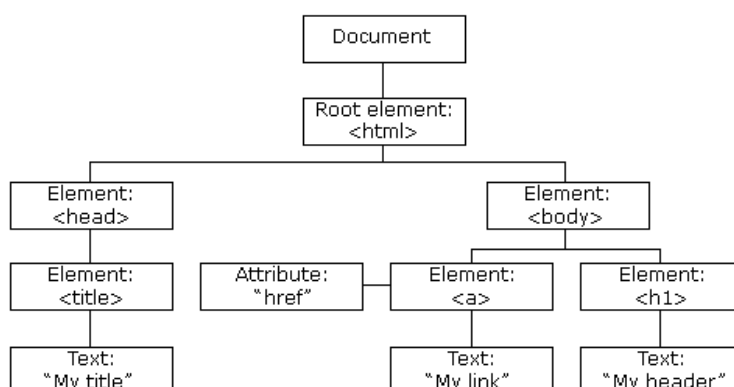
```

        karticaRadnika.appendChild(p);
        karticaRadnika.appendChild(p2);
    });
    } else {
        // Ispis poruke ukoliko dođe do problema
        const errorMessage =
document.createElement('marquee');
        errorMessage.textContent = 'Problem!';
        app.appendChild(errorMessage);
    }
}
request.send();
}
    
```

Kako bi bio jasniji način na koji radi funkcija *prikaz()*, u narednim sekcijama će biti objašnjeni njeni najvažniji delovi.

5.5.1. Pristup elementima html strane

Svi elementi u strukturi *html* strane (za strukturu *html* dokumenta se koristi termin *html DOM*, *Document Object Model*) su ugnežđeni jedan u drugi: *html* tag sadrži *head* i *body*; *body* može sadržati a ili *h1*, svaki od ovih tagova može sadržati druge tagove itd. (slika 17)



Slika 17: Struktura *html* Document Object Model-a.

Izvor: www.w3schools.com

Postoji više *JavaScript* funkcija pomoću kojih se menja *DOM*, a u nastavku će biti objašnjene one koje su korišćene u funkciji *prikaz*.

Funkcija *getElementById* se koristi za pristupanje željenom elementu na strani – u zagradi se unosi id *html* elementa kom želimo da pristupimo. Kada pristupimo određenom elementu, kao što je

```
const app = document.getElementById("root");
```

taj element će biti sačuvan u okviru promenljive *app* (u ovom konkretnom slučaju). Obratite pažnju da je promenljiva označena kao *const* – to znači da je nakon što sačuvamo neku vrednost, ona konstantna i neće biti podložna promenama. U pitanju je dobra praksa kada se koristi funkcija *getElementById* za učitavanje određenog objekta i njegovo čuvanje u promenljivoj.

Obratite pažnju na raspored malih i velikih slova u pozivu funkcije *getElementById*. Iako bi Vam možda prirodnije izgledalo da piše, na primer, *getElementById*, funkcija će raditi ispravno samo ukoliko piše *getElementById*.

Naredba *createElement* služi za pravljenje novih *html* elemenata - u ovom slučaju se pravi element tipa *div* i čuva u promenljivoj *container*:

```
const container = document.createElement('div');
```

Korišćenjem naredbe *setAttribute* možemo promeniti svojstva *html* elementa sa kojim radimo. Na primer:

```
container.setAttribute('class', 'containerClass');
```

za objekat tipa *div* koji smo sačuvali u promenljivoj *container*, možemo promeniti određene atribute – u ovom slučaju, promenjen je atribut *class* i dodeljena mu je vrednost *containerClass*. Rezultat

izvršavanja prethodna dva primera je isti kao i da ste direktno na *html* strani uneli naredbu:

```
<div class="containerClass"></div>
```

Razlika je u tome što je korišćenje *JavaScript* naredbi omogućilo da dinamički pravimo i menjamo delove *html* strane. Naredba *setAttribute* se može koristiti i za promenu svih drugih atributa određenog elementa.

Kako bi se omogućilo dinamičko menjanje strukture strane (odnosno, *html DOM-a*), može se koristiti funkcija *appendChild*. Ona omogućava da u okviru postojećeg elementa dodamo drugi element. To je urađeno i kada je u promenljivoj *app*, dodat element koji je obeležen kao *container*.

```
app.appendChild(container);
```

Ukoliko još jednom pogledamo naredbe:

```
document.getElementById("root").innerHTML = "";  
  
const app = document.getElementById("root");  
  
const container = document.createElement('div');  
container.setAttribute('class', 'containerClass');  
  
app.appendChild(container);
```

rezultat njihovog izvršavanja je potpuno isti kao i kada bi direktno na *html* strani bilo napisano:

```
<div id="root">  
    <div class="containerClass"></div>  
</div>
```

Razlika je u tome što direktnim unosom na html strani dobijamo statički sadržaj, a korišćenjem *JavaScript*-a možemo prilagođavati sadržaj strane dinamički, prema trenutnim potrebama.

5.5.2. Objekat *XMLHttpRequest*

Objekat *XMLHttpRequest* se koristi za slanje zahteva serveru za dostavljanje određenih podataka, a podržavaju ga svi savremeni pretraživači. Korišćenje ovog objekta omogućava ažuriranje veb strane bez potrebe za njenim ponovnim učitavanjem, a nakon što je sadržaj strane već prikazan korisniku. Na ovaj način, stvara se dobar utisak za korisnike aplikacije jer se novi sadržaj „ubrizgava“ u okviru postojeće strane bez potrebe da se ceo prikaz strane ponovo generiše.

Nakon pravljenja novog objekta tipa *XMLHttpRequest* i čuvanja njegove vrednosti u promenljivoj *request*, poziva se funkcija *open()*:

```
var request = new XMLHttpRequest();
request.open('GET',
            'http://localhost/evidencija/servisi/radnici',
            true);
```

Funkcija *open()* prihvata tri parametra:

- *http* metod koji se koristi, kao što je *GET*, *POST*, *PUT*, *DELETE* itd;
- adresa na koju se upućuje zahtev – u našem primeru je to adresa koja je već korišćena za testiranje veb servisa;
- opcioni parametar tipa *Boolean* koji označava da li će se poziv obavljati asinhrono. Ukoliko je vrednost *true* (što je i podrazumevana vrednost, ako parametar nije definisan), obaveštenje o kompletiranoj transakciji (slanje zahteva i vraćanje odgovora) će biti vraćeno kada ona bude završena. U našem primeru, to znači da se klikom na dugme *Prikaži radnike* šalje zahtev ka serveru, a veb strana na kojoj se

nalazi dugme može dalje da se koristi i za druge aktivnosti dok se u pozadini čeka odgovor servera. Ukoliko je parametar podešen na *false*, to znači da zahtev neće biti asinhron (dakle, zahtev će postati sinhronizovan) – u praksi to znači da će klikom na dugme *Prikaži radnike* biti poslat zahtev serveru, ali će veb strana biti “zamrznuta” za sve dalje aktivnosti – korisnik neće moći da bilo šta uradi dok se sinhronizacija sa serverom ne završi i dok se odgovor na zahtev ne vrati na stranu.

U praksi, iz ugla kvaliteta korišćenja aplikacije, trebalo bi koristiti asinhronu zahteve (opcija *true*), osim ukoliko ne postoji veoma opravdan razlog za korišćenje sinhronizovanih zahteva.

Finalni korak prilikom pravljenja objekta *XMLHttpRequest* je definisanje funkcije koja će biti pozvana kada se vrati odgovor servera (događaj *load*) – ona će obraditi odgovor koji je server vratio i pripremiti ga za prikazivanje na strani. Obzirom da se funkcija poziva samo u ovom specifičnom slučaju i neće biti korišćena na drugim mestima u aplikaciji, definisaćemo je kao anonimnu funkciju koja će biti pozvana kada se desi događaj *load*. Da bi se pratilo kada se desio događaj *load*, koristi se rukovalac događaja *onload*.

```
request.onload = function () {  
    // Pristup JSON podacima  
    .....  
}
```

Sadržaj anonimne funkcije za događaj *onload* biće obrađen u narednoj sekciji.

Poslednja, i izuzetno važna, naredba u funkciji *prikaz()* je naredba *send()*. Ona zapravo omogućava slanje samog zahteva (odnosno objekta *XMLHttpRequest*), a bez njenog pozivanja ne bi došlo do

prikazivanja podataka o radnicima na strani. Obratite pažnju na mesto sa kog se ova funkcija poziva – ona je poslednja naredba u funkciji *prikaz()*, ali se nalazi izvan anonimne funkcije za događaj *onload*.

```
request.send();
```

Ukoliko grupišemo na jedno mesto naredbe vezane za pravljenje objekta *XMLHttpRequest* koje su korišćene u ovoj sekciji, dobićemo strukturu prikazanu u nastavku:

```
var request = new XMLHttpRequest();
request.open('GET',
            'http://localhost/evidencija/servisi/radnici',
            true);

request.onload = function () {
    // Pristup JSON podacima
    .....
}

request.send();
```

Dakle, nakon pravljenja objekta tipa *XMLHttpRequest*, poziva se funkcija *open()*, definiše se anonimna funkcija koja rukuje događajem *onload* i poziva se funkcija *send()*.

5.5.3. Sadržaj anonimne funkcije za događaj load

Anonimna funkcija koja se poziva kada se dogodi događaj *load*, obrađuje odgovor koji je server vratio i priprema ga za prikazivanje na strani.

Obzirom da će odgovor koji server vrati biti u *JSON* formatu, prvi korak je parsiranje sadržaja pomoću promenljive *JSON.parse()* – način korišćenja ove funkcije je već prikaz u prethodnim poglavljima:

```
var data = JSON.parse(this.response);
```

Kao parametar funkcije *parse()* prosleđuje se odgovor servera kome pristupamo pomoću *this.response*. Rezultat operacije parsiranja će biti sačuvan u promenljivoj *data*.

Naredni korak je provera statusa izvršenja zahteva od strane servera:

```
if (request.status >= 200 && request.status < 400) {
```

Http statusni kodovi su trocifreni brojevi i pokazuju da li je zahtev upućen serveru bio uspešan i da li je vraćen neki odgovor (ukoliko je uopšte vraćen). ⁴

Statusni kodovi su definisani u opsegu od 0 do 599, a prva cifra definiše grupu kojoj status pripada: Na osnovu prve cifre, postoji pet grupa statusnih odgovora: ⁵

- *1xx* – informativni status: server je primio zahtev, nastavlja se sa obradom;
- *2xx* – uspešan zahtev: server je uspešno primio, razumeo i prihvatio zahtev;
- *3xx* – preusmerenje: dalje akcije se moraju sprovesti kako bi se zahtev kompletirao;
- *4xx* – klijentska greška: zahtev sadrži neispravnu sintaksu i ne može biti ispunjen;
- *5xx* – greška na serveru: server nije uspeo da ispuni zahtev, iako je zahtev ispravan.

U našem slučaju, važno je da statusni kod nema oznaku veću manju od 200 ili veću 400 jer bi to značilo da zahtev nije ispunjen. Sve statusne poruke u opsegu od 200 do 399 pokazuju da se može

⁴ Kompletan lista statusnih poruka dostupna je na adresi: developer.mozilla.org/en-US/docs/Web/HTTP/Status

⁵ Za više detalja o statusnim porukama, pogledajte tekst standarda *RFC 9110* (dostupno onlajn). Ovaj standard definiše semantiku *http-a*.

nastaviti sa obradom odgovora koji je server vratio. U tom slučaju, prelazi se na obradu promenljive *data*, koja sadrži parsirani odgovor servera.

```
data.forEach(radnik => {
```

Korišćenjem petlje *forEach*, prolazi se kroz sve rezultate u promenljivoj *data* – obrađuju se podaci jednog po jednog radnika, tako što se u svakom ciklusu petlje pravi *div* tag koji će sadržati podatke o jednom radniku:

```
const karticaRadnika = document.createElement('div');  
card.setAttribute('class', 'karticaRadnika');
```

Promenljiva *karticaRadnika* će sadržati *div* tag kome je dodeljena klasa '*karticaRadnika*' (nešto kasnije će ova klasa biti definisana u *css*-u). Potom se prave naslov (*h1* tag) i dva pasusa (*p* tag).

```
const h1 = document.createElement('h1');  
h1.textContent = radnik.ime;  
  
const p = document.createElement('p');  
p.textContent = `Plata: ${radnik.plata}`;  
  
const p2 = document.createElement('p');  
p2.textContent = `Starost: ${radnik.starost}`;
```

Promenom vrednosti atributa *textContent* menja se sadržaj napravljenih *html* elemenata. Kako bismo za svakog radnika došli do podataka koji su dobijeni kao odgovor servera, koristimo konstrukciju *radnik.vrednost*. Ukoliko se vratite na početak ovog poglavlja, videćete da su kao vrednosti u *JSON* fajlu definisani *id*, *ime*, *plata* i *starost*. Zbog toga, da bi se iznos plate upisao u *p* element, navodimo:

```
p.textContent = `Plata: ${radnik.plata}`;
```

String koji se dodeljuje atributu *textContent* se može napraviti i konkatenacijom stringova – time bi se izbeglo korišćenje oznake *`\${}*`, ali bi izraz postao manje čitljiv.

Oznaka *`\${}* saopštava *JavaScript*-u da se između zagrada nalazi promenljiva čiju vrednost je potrebno prikazati. Važno je napomenuti da, ukoliko želite koristiti izraze koji uključuju *`\${}*`, morate koristiti iskošene apostrofe da obuhvatite izraz `()` umesto klasičnih apostrofa `()` za obične *String*-ove.

Nakon što su naslov i pasusi napravljeni i popunjeni podacima o radniku, potrebno je dodati ih u *DOM html* strane:

```
container.appendChild(karticaRadnika);  
karticaRadnika.appendChild(h1);  
karticaRadnika.appendChild(p);  
karticaRadnika.appendChild(p2);
```

Prvo se u `div container` dodaje *karticaRadnika*, a potom se u nju ubacuju naslov i pasusi. Navedene procedure pravljenja *karticeRadnika* i popunjavanja podacima se ponavljaju za svakog radnika, onoliko puta koliko radnika ima u *JSON* odgovoru koji je server vratio.

Poslednji blok naredbi je vezan za slučaj da je statusni kod manji od 200 ili veći od 400, što znači da je došlo do nekog problema.

```
} else {  
    // Ispis poruke ukoliko dođe do problema  
    const errorMessage = document.createElement('h3');  
    errorMessage.textContent = 'Problem!';  
    app.appendChild(errorMessage);  
}
```

5.5.4. Definisanje izgleda frontenda

Kako bi se prilagodio izgled strane, može se koristiti *css*. Ispod je dat izgled jednostavne strukture koja se može koristiti kao osnova. Obratite pažnju da su korišćene klase *containerClass* i

karticaRadnika koje su dinamički dodeljene *div* elementima u *JavaScript*-u u funkciji *prikaz()*.

```
* {
  box-sizing: border-box;
}

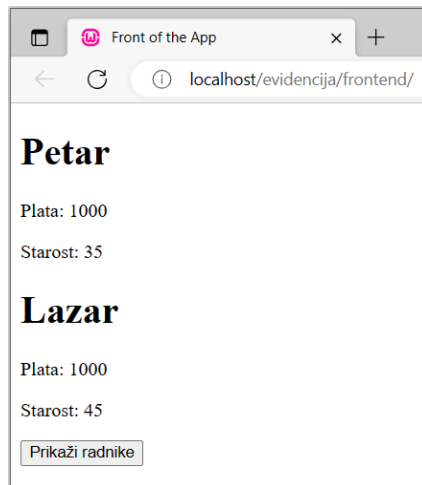
#root {
  max-width: 800px;
}

.containerClass {
  display: flex;
  flex-wrap: wrap;
}

.karticaRadnika {
  background: white;
  box-shadow: 3px 5px 20px rgba(0, 0, 0, .15);
  border-radius: 10px;
  overflow: hidden;
}

.karticaRadnika:hover {
  box-shadow: 3px 5px 20px rgba(0, 0, 0, .2);
}
```

Nakon što se *index.html* strana pokrene u pretraživaču, prikazaće se rezultat sličan onom na slici 18. U zavisnosi od podataka koji se nalaze u tabeli *radnici*, prikaz može biti i drugačiji.



Slika 18: Prikaz izgleda pokrenute aplikacije.

6. LARAVEL, PRIPREMANJE OKRUŽENJA

Postoji više mogućnosti instaliranja *Laravel*-a koje zavise od operativnog sistema, kao i drugih opcija koje budete odabrali. U nastavku je prikazana jedna od mogućnosti instalacije koja koristi:⁶

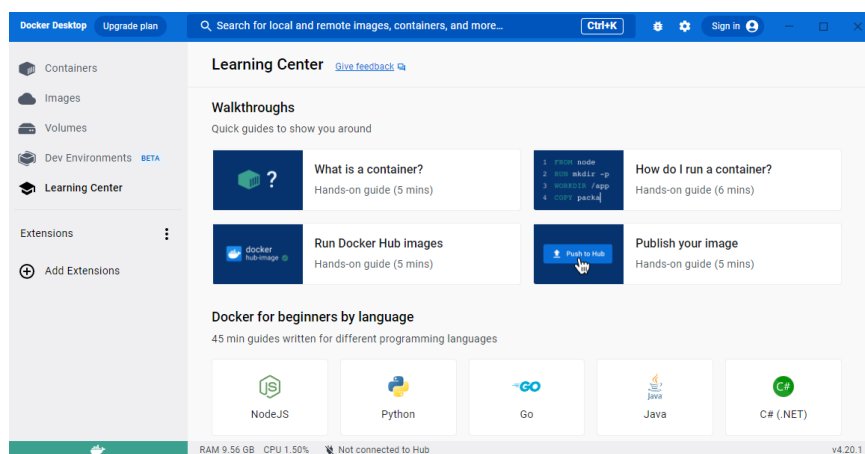
- *Windows* operativni sistem,
- *Docker Desktop* kao platformu za pokretanje aplikacija,
- *WSL (Windows Subsystem for Linux)* koji omogućava instaliranje i korišćenje *Linux* distribucija direktno u okviru *Windows* sistema,
- *Microsoft Visual Studio Code* kao editor za razvoj softvera i
- *Laravel Sail* kao jednostavan interfejs koji se koristi iz komandne linije, a omogućava brz početak razvoja *Laravel* aplikacije bez prethodnog iskustva sa *Docker*-om.

U narednim sekcijama biće prikazan način instalacije i povezivanja navedenih alata.

⁶ U trenutku pisanja ovog udžbenika, poslednja stabilna verzija *Laravel*-a je 10. Za više informacija o instaliranju na drugim operativnim sistemima ili sa drugim kombinacijama alata posetite adresu laravel.com/docs/10.x/installation

6.1. Docker Desktop

Docker je okruženje koje omogućava pravljenje i deljenje kontejnerskih aplikacija i mikroservisa. *Docker* kontejneri rade u okviru virtuelne mašine (u ovom udžbeniku, radi se sa *Linux* kontejnerima), a za njihovo pokretanje koristi se aplikacija *Docker Desktop*.⁷ Nakon što se aplikacija instalira, dobiće se grafičko okruženje slično onom na slici 19:



Slika 19: Izgled aplikacije Docker Desktop.

6.2. Windows Subsystem for Linux

Windows Subsystem for Linux (WSL) je tehnologija koja omogućava pokretanje Linux distribucija i aplikacija na *Windows* operativnom sistemu. *WSL* koristi virtuelizaciju i prevođenje sistemskih poziva da bi obezbedio kompatibilnost i performanse. *WSL* je koristan za razvojne i testne svrhe, kao i za korišćenje Linux alata i komandi na *Windows* sistemima.

Jedna od glavnih prednosti *WSL*-a je integracija dva operativna sistema na jednom računaru, što olakšava razvoj i testiranje

⁷ Aplikacija se može preuzeti sa adrese: docker.com/products/docker-desktop/

aplikacija. Korisnici mogu koristiti svoje omiljene *Linux* alate direktno u *Windows* okruženju, bez potrebe za pokretanjem virtuelne mašine ili dualnog podizanja sistema.

Dodatne *WSL* distribucije ⁸ se jednostavno instaliraju pomoću komande:

```
wsl --install
```

u *PowerShell*-u ili *Command Prompt*-u koji moraju biti pokrenuti u administratorskom režimu (prilikom pokretanja, kliknite ikonicu desnim tasterom miša i izaberite opciju *Run as administrator*). Podrazumevana *Linux* distribucija je *Ubuntu* (koja će se koristiti i u ovom udžbeniku), ali se može promeniti.⁹ Za instalaciju uz navođenje distribucije, može se koristiti naredba (reč *Ubuntu* se može zameniti imenom druge *Linux* distribucije):

```
wsl --install -d Ubuntu
```

Tokom instalacije, biće zatraženo da se unese korisničko ime i lozinka – ove vrednosti će biti potrebne nešto kasnije. Nakon instalacije, može se pokrenuti *Linux* terminal i koristiti *Bash Shell*. Takođe je moguće instalirati i koristiti grafičke aplikacije iz *Linux* okruženja, ukoliko je potrebno.

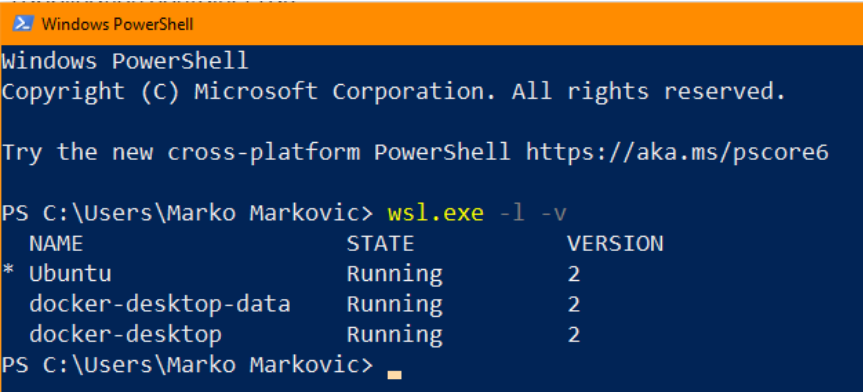
Naredni korak je promena *WSL* verzije na 2 (podrazumevana verzija je 1). Provera verzije vrši se pomoću naredbe:

```
wsl -l -v
```

⁸ *Docker Desktop* ne zahteva instalaciju dodatnih *Linux* distribucija i radiće i bez njih. Ipak, preporuka je da se instalira barem jedna distribucija, u ovom slučaju *Linux Ubuntu*.

⁹ Više informacija o instaliranju i prilagođavanju *WSL*-a dostupno je na linku: learn.microsoft.com/en-us/windows/wsl/install

Slika 20 prikazuje izvršavanje ove komande.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\Marko Markovic> wsl.exe -l -v
  NAME                                STATE             VERSION
*  Ubuntu                             Running           2
   docker-desktop-data                Running           2
   docker-desktop                     Running           2
PS C:\Users\Marko Markovic>
```

Slika 20: Provera verzije WSL-a.

Ukoliko za distribuciju Ubuntu koja je instalirana pomoću naredbe `--install` piše da je u verziji 1, potrebno je primeniti sledeću naredbu:

```
wsl.exe --set-version Ubuntu 2
```

Naziv *Ubuntu* može se zameniti imenom druge *Linux* distribucije koja je potrebna. Može se podesiti i podrazumevana verzija za sve naredne instalacije distribucija:

```
wsl.exe --set-default-version 2
```

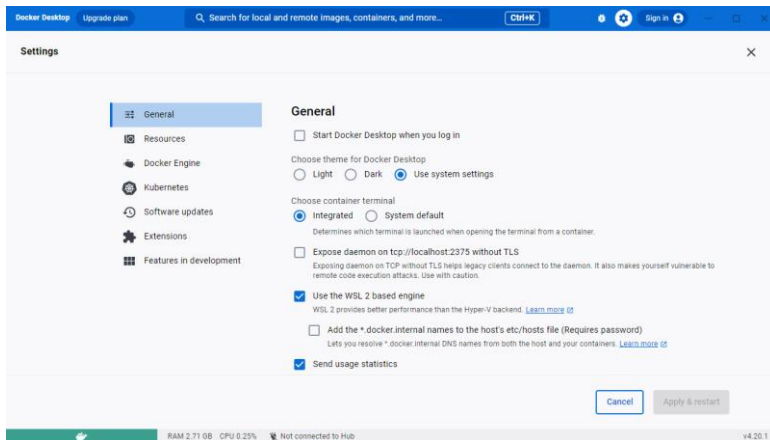
Svaki sledeći put kada se korišćenjem naredbe `--install` doda nova *Linux* distribucija, njena verzija će biti 2.

6.3. Povezivanje Docker-a i WSL-a

Naredni korak je da se u *Docker Desktop* aplikaciji omogući korišćenje WSL-a.

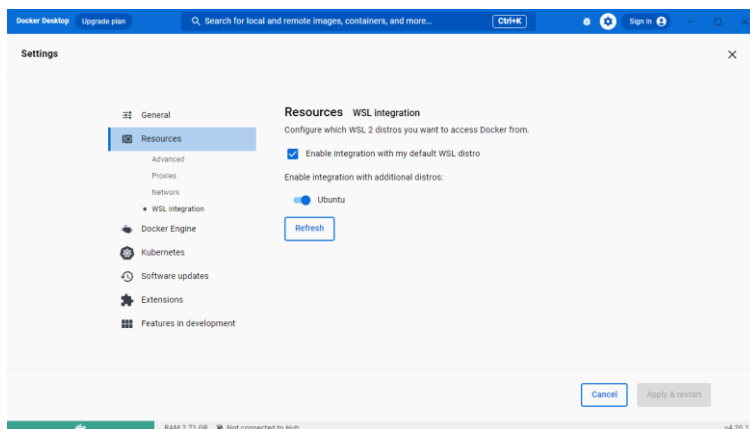
Potrebno je pokrenuti aplikaciju *Docker Desktop*, izabrati opciju *Settings* i u okviru nje otvoriti karticu *General*, a potom aktivirati

opciju *Use WSL 2 based engine* (slika 21). Potom kliknite dugme *Apply & restart*.



Slika 21: Uključivanje WSL-a u aplikaciji Docker Desktop.

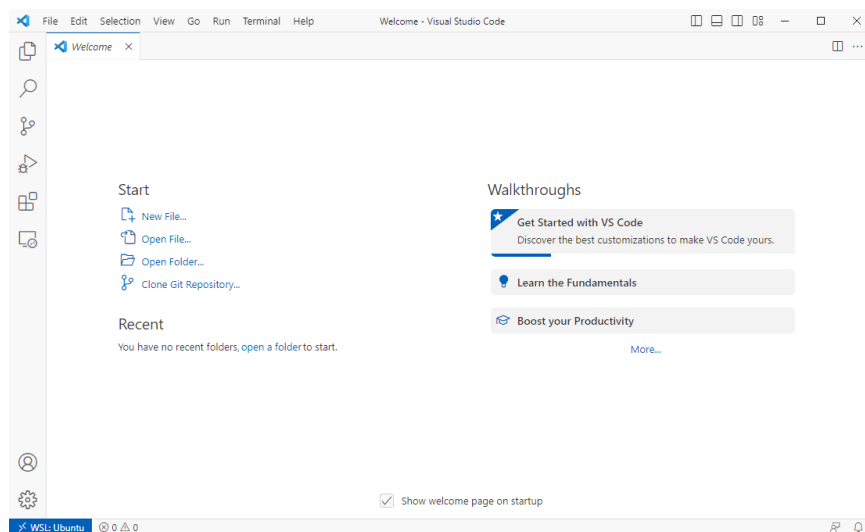
Naredni korak je da se u aplikaciji *Docker Desktop* ponovo otvori *Settings*, zatim kartica *Resources* i u okviru nje *WSL Integration*. Opcija *Enable integration with additional distros* bi trebala da bude aktivirana, a trebalo bi i aktivirati i distribucije koje je potrebno koristiti - u ovom slučaju, treba aktivirati *Ubuntu* (slika 22). Potom kliknite dugme *Apply & restart*.



Slika 22: Uključivanje Linux distribucija u aplikaciji Docker Desktop.

6.4. Microsoft Visual Studio Code

Microsoft *Visual Studio Code* je alat za razvoj softvera za Windows, Linux i Mac operativne sisteme koji omogućava programerima da pišu, testiraju i otklanjaju greške u svojim aplikacijama uz pomoć raznih funkcionalnosti, kao što je isticanje sintakse i inteligentno dovršavanje koda, prilagodljivo korisničko okruženje, ugrađena GitHub integracija, podrška za mnoge programske jezike, mogućnost instaliranja raznih dodataka itd. *Visual Studio Code* je besplatan alat otvorenog koda.¹⁰ Slika 23 prikazuje izgled početnog ekrana aplikacije.

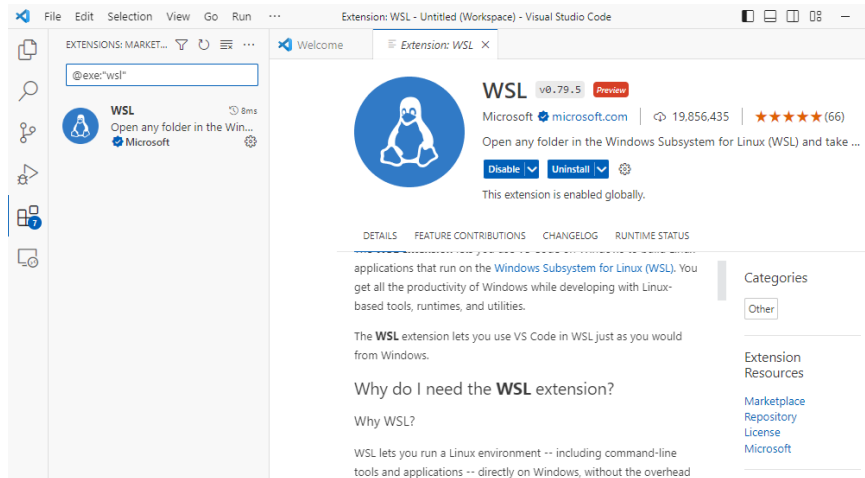


Slika 23: Početni ekran aplikacije Visual Studio Code.

Nakon pokretanja aplikacije, iz menija sa leve strane potrebno je izabrati kategoriju Extensions i u pretrazi pronaći dodatak *WSL*, a potom ga i instalirati. Ovaj dodatak olakšava razvoj aplikacija koje koriste *WSL* direktno iz okruženja *Visual Studio Code* tako što

¹⁰ *Visual Studio Code* se može preuzeti sa adrese code.visualstudio.com/download

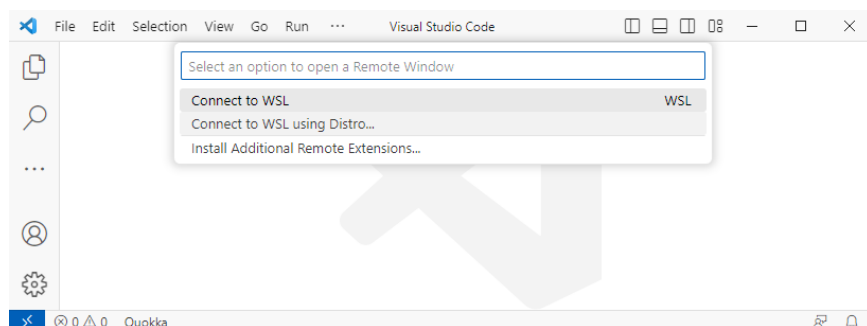
omogućava pristup *Windows* fajl sistemu iz *Linux* kontejnera. Slika 24 prikazuje izgled ekrana nakon instalacije dodatka.



Slika 24: Instalacija dodatka WSL za Visual Studio Code.

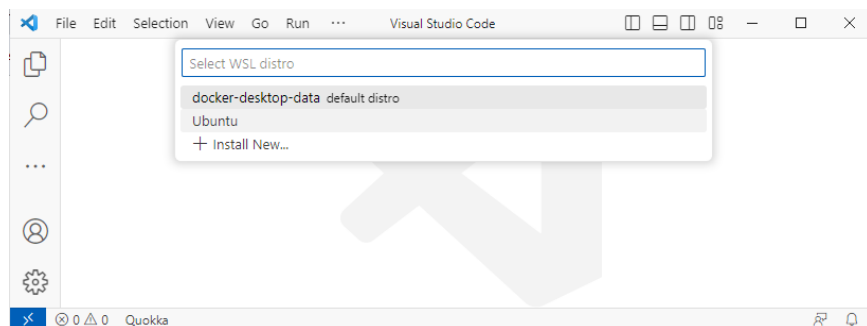
6.5. Pokretanje terminala

Da bi se instalirao *Laravel Sail* i pokrenuo *Laravel* projekat, potrebno je pristupiti terminalu. Moguće je koristiti i sistemski *PowerShell* ili *Command Prompt* – obzirom da je *Visual Studio Code* odlično objedinjeno rešenje za razvoj, terminal se može pokrenuti i u okviru njega – na ovaj način se praktično kompletan razvoj može raditi iz jednog softvera. Slika 25 prikazuje početni ekran – potrebno je u donjem levom uglu kliknuti na plavu ikonicu, a zatim, kada se na vrhu ekrana otvori novi dijalog, izabrati opciju *Connect to WSL using Distro...*



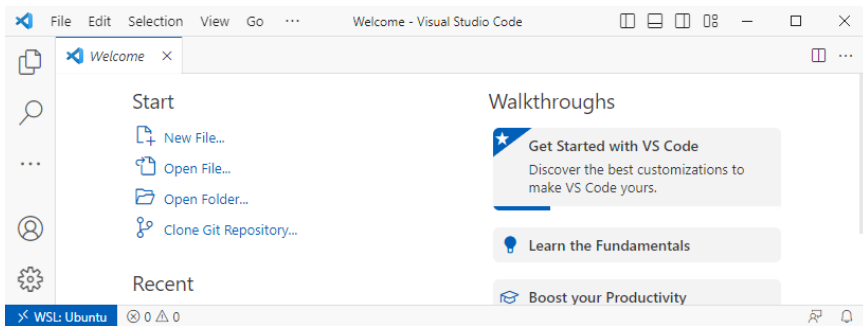
Slika 25: Dodavanje Linux distribucije u Visual Studio Code.

Na ovaj način je započet proces dodavanja nove distribucije u *Visual Studio Code* okruženje, nakon čega je potrebno izabrati koju distribuciju želimo. U ovom slučaju, potrebno je izabrati *Ubuntu* (slika 26).



Slika 26: Izbor Linux distribucije.

Nakon što sistem uspešno doda željenu distribuciju, u donjem desnom uglu u plavom pravougaoniku, pojaviće se ispisana reč *WSL* i naziv verzije *Linux-a* koja se koristi, u ovom slučaju pisaće *WSL:Ubuntu* (slika 27).



Slika 27: Uspešno dodata Linux distribucija.

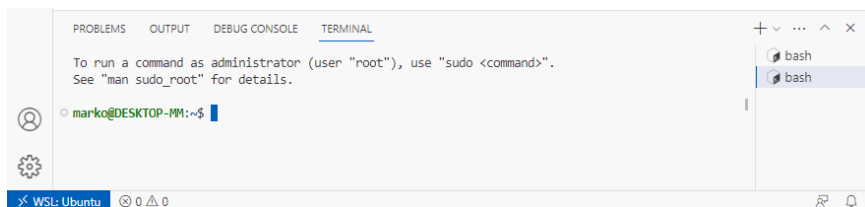
Neposredno pored plavog polja u kom piše ime distribucije se nalazi prikaz broja upozorenja i poruka (trenutno su vrednosti nula), a ukoliko se klikne na njih, otvoriće se panel sa nekoliko kartica – potrebno je otvoriti karticu *Terminal* (slika 28).



Slika 28: Izgled terminala u aplikaciji Visual Studio Code.

Prva reč (ispred znaka @) u terminalu biće korisničko ime koje je definisano prilikom instalacije distribucije *Ubuntu*. Iza znaka @ će se nalaziti naziv računara.

U gornjem desnom uglu terminala nalazi se ikonica sa znakom +. Klikom na njega, pravi se još jedan terminal, a sa desne strane se pojavljuje još jedan panel na kom će biti izlistani svi napravljeni terminali i pomoću koga se lako bira onaj u kom će se trenutno raditi (slika 29).



Slika 29: Rad sa više terminala.

Rad sa više terminala je važan jer će u narednim sekcijama biti potrebe da se koriste dva:

- jedan za instaliranje *Laravel Sail*-a i njegovo pokretanje;
- drugi za izvršavanje *Laravel* naredbi.

6.6. Laravel Sail, instalacija

Laravel Sail je komandni interfejs za rad sa *Laravel*-ovim podrazumevanim razvojnim okruženjem zasnovanim na *Docker*-u. Ukoliko na računaru nemate lokalno instaliran *PHP*, bazu podataka itd., *Laravel Sail* je odlično rešenje jer omogućava jednostavno i automatizovano dodavanje svih potrebnih elemenata za razvoj aplikacija.

Da bi se *Sail* instalirao, potrebno je u terminalu otvoriti folder u kom će se nalaziti novi *Laravel* projekat (za potrebe ovog udžbenika, potrebno je napraviti folder pod imenom *laravel* na *c* particiji – putanja do njega će biti *c:/laravel*). Pristupićemo mu kucanjem sledeće naredbe u terminalu:

```
cd /mnt/c/laravel/
```

Naredba *cd* služi za promenu trenutnog direktorijuma (*cd*, *Change Directory*), a prati je putanja do direktorijuma kom želimo da pristupimo, tj. *c/laravel/*. Primetićete da se na početku putanje pojavila reč *mnt* (*MouNT*). To je zato što se koristi *Linux* kontejner

u okviru *Windows* sistema – da bi putanja mogla da se pronađe, potrebno je navesti *mnt* kako bi *Linux* kontejner znao da treba da učita (*mount*-uje) eksternu particiju koja pripada *Windows* sistemu. Nakon unosa naredbe i njenog izvršavanja, iza ispisa *korisnik@računar* se pojavljuje i putanja do trenutno aktivnog direktorijuma (slika 30).

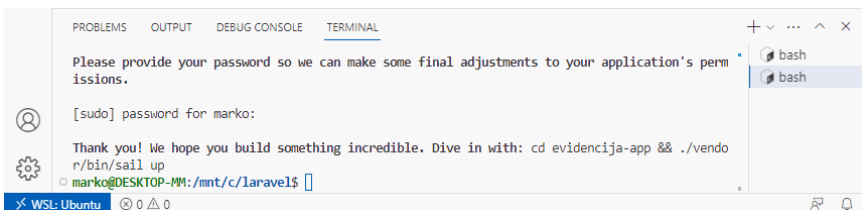


Slika 30: Promena trenutno aktivnog direktorijuma.

Trenutno aktivni direktorijum je mesto u kom će biti napravljen novi *Laravel* projekat i instaliran *Laravel Sail*. Kada se dođe do ove faze, sama instalacija je jednostavna. Najjednostavniji način je korišćenje servisa *laravel.build* koji će preuzeti sve potrebne pakete i napraviti novu *Laravel* aplikaciju. Narednu naredbu je potrebno uneti u terminal:

```
curl -s https://laravel.build/evidencija-app | bash
```

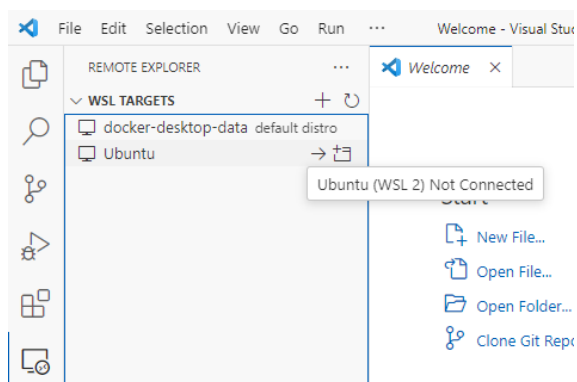
Na ovaj način, počinje pravljenje aplikacije koja će se zvati *evidencija-app*. Naravno, moguće je upisati drugo ime aplikacije. Proces instaliranja će trajati određeno vreme u zavisnosti od brzine računara i internet konekcije, a biće zatražen i unos lozinke koja je definisana prilikom instaliranja distribucije *Ubuntu*. Slika 31 prikazuje izgled terminala kada instalacija bude gotova.



Slika 31: Završena instalacija novog Laravel projekta.

Napomena: Ukoliko, nakon što se unese naredba za početak instalacije, bude prikazana poruka *Docker is not running*, možete pokušati sledeće:

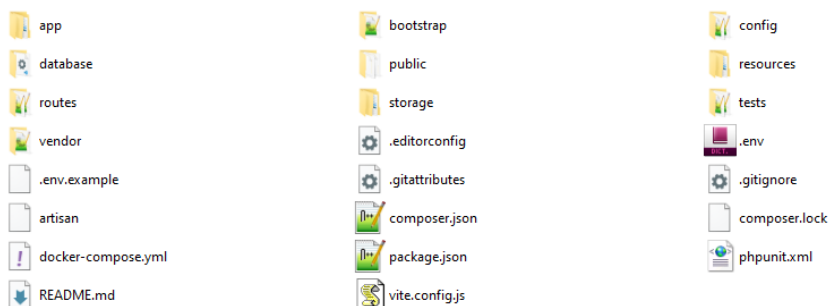
- proverite da li je aplikacija *Docker Desktop* pokrenuta;
- prekinite konekciju klikom na plavi pravougaonik u donjem levom uglu i izborom opcije *Close remote connection*. Potom u meniju sa leve strane izaberite ikonicu *Remote explorer*. U listi koja se otvori izaberite distribuciju (*Ubuntu* u ovom slučaju), zatim pored imena distribucije izaberite opciju *Connect in Current Window* ili *Connect in New Window* (slika 32). Nakon što se uspostavio konekcija, otvorite terminal, unesite putanju do direktorijuma *c:/laravel* i ponovo pokrenite opciju za instaliranje novog *Laravel* projekta.



Slika 32: Pokretanje nove WSL konekcije.

Nakon što je instalacija završena, u direktorijumu *c:/laravel* se pojavio novi direktorijum (*evidencija-app*) u kome se nalaze svi

direktorijumi aplikacije, kao i svi servisi koje je *Laravel Sail* automatski instalirao (slika 33).



Slika 33: Sadržaj direktorijuma aplikacije evidencija-app.

Ukoliko nije drugačije definisano prilikom instalacije, automatski su instalirani servisi kao što su *MySQL*, *Redis* i *Selenium*. Za samostalni izbor servisa prilikom pravljenja novog projekta, može se koristiti ključna reč *with*:

```
curl -s "https://laravel.build/evidencija-app?with=mysql,redis"  
| bash
```

Na ovaj način, eksplicitno je navedeno da će se instalirati *MySQL* i *Redis*.

6.7. Laravel Sail, pokretanje

Nakon što su svi potrebni elementi aplikacije instalirani, potrebno je samo pokrenuti *Laravel Sail*. Prvo je u terminalu potrebno ući u direktorijum aplikacije. U prethodnoj sekciji, kao aktivni direktorijum je već podešena putanja `c:/laravel`, sada je samo potrebno još otvoriti direktorijum aplikacije koji je napravljen u njemu (slika 34):

```
cd evidencija-app
```



Slika 34: Otvaranje direktorijuma aplikacije.

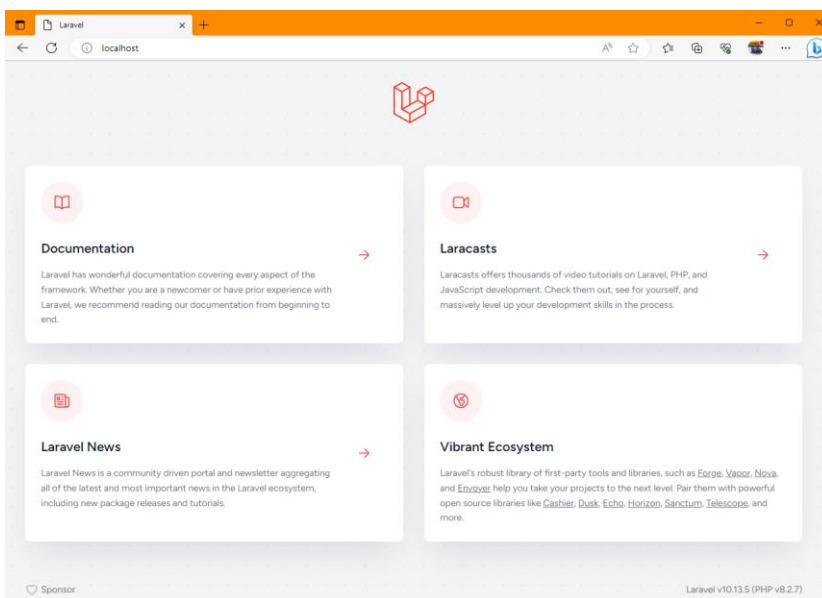
Naredni korak je pokretanje *Sail*-a pomoću naredbe:

```
./vendor/bin/sail up
```

Naredba *sail up* označava da želimo da pokrenemo aplikaciju (*da zaplovimo*). Nakon što se ona izvrši i aplikacija pokrene, u pretraživaču se može pristupiti adresi:

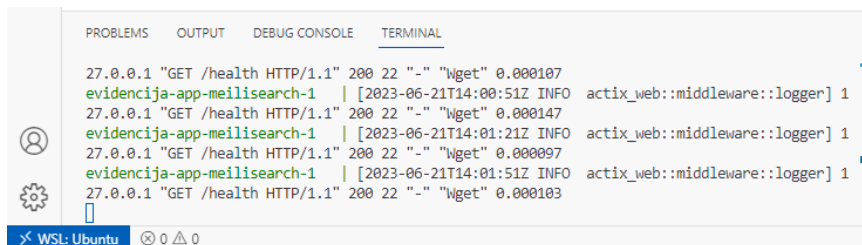
```
http://localhost
```

Ukoliko se u pretraživaču otvori strana kao na slici 35, instalacija novog projekta i njegovo pokretanje su uspeli.



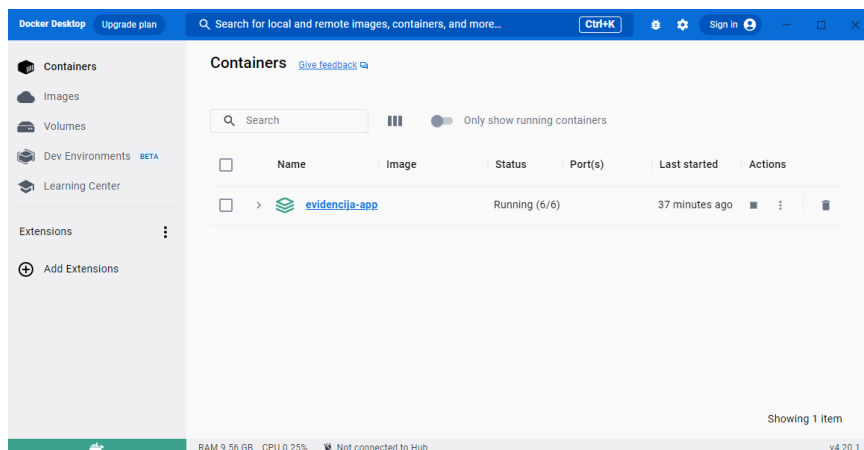
Slika 35: Podrazumevana početna strana Laravel aplikacije.

Slika 36 prikazuje izgled terminala dok je aplikacija pokrenuta – u realnom vremenu će biti prikazivane statusne poruke vezane za rad aplikacije. Ovo je ujedno i razlog zašto će se koristiti dva terminala – prvi (sa slike 36) će biti zauzet praćenjem informacija o pokrenutoj aplikaciji; drugi će se koristiti za izvršavanje naredbi.



Slika 36: Izgled terminala dok je aplikacija pokrenuta.

Slika 37 prikazuje izgled liste kontejnera u aplikaciji *Docker Desktop* – kao što se i vidi, obzirom da je *WSL Ubuntu* instaliran i povezan sa *Docker*-om, nakon pokretanja komande *sail up* se i pojavio u listi pokrenutih kontejnera.



Slika 37: Izgled pokrenutog Docker kontejnera aplikacije *evidencija-app*.

6.8. Konfigurisanje aliasa za Sail

U prethodnoj sekciji je pozivana naredba:

```
./vendor/bin/sail up
```

Putanja koja prethodi (*./vendor/bin/*) opisuje gde se tačno nalazi naredba *sail up* – ukoliko pogledate strukturu aplikacije *evidencija-app*, primetićete da u njoj postoji direktorijum *vendor*, a u njemu direktorijum *bin* koji sadrži dokument *sail*.

Navedena putanja se mora navesti prilikom svakog pokretanja komande *sail*, što može biti naporno. Kako bi se to izbeglo moguće je napraviti *alias* kako bi se pojednostavilo njeno pozivanje. U novom terminalu se može izvršiti naredba:

```
alias sail='[ -f sail ] && sh sail || sh vendor/bin/sail'
```

Nakon toga, za pozivanje naredbe *sail* dovoljno je napisati:

```
sail up
```

Ukoliko radite u novom terminalu, obavezno pre pozivanja naredbe *sail*, podesite putanju na direktorijum aplikacije sa kojom trenutno radite, na primer:

```
cd /mnt/c/laravel/evidencija-app
```

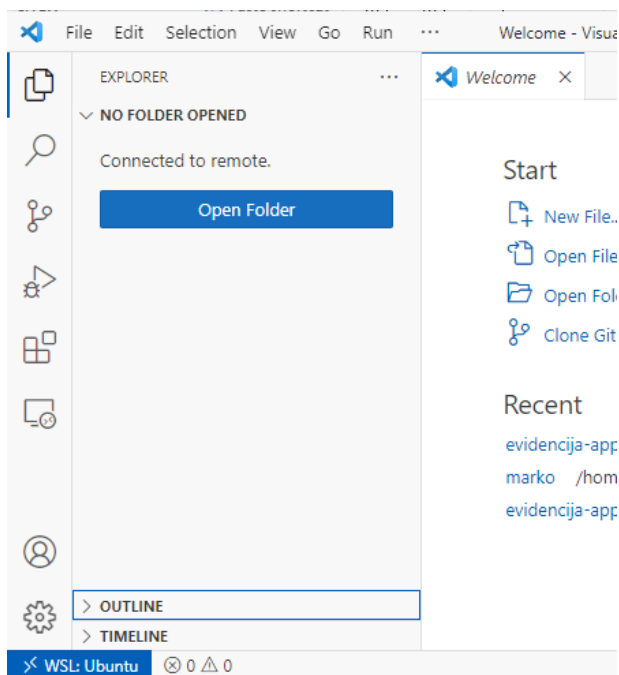
Napomena: Alias se svaki put ponovo pravi za svaki prozor terminala. Ukoliko želite da napravite permanentni alias na nivou aplikacije kako ne biste za svako novo otvaranje terminala morali da ga pravite ponovo, možete ga dodati u konfiguracioni fajl `~/.bashrc` u glavnom direktorijumu aplikacije. Obzirom da takve vrste konfigurisanja izlaze iz okvira ovog udžbenika, nećemo se detaljnije baviti njima.

Pretpostavka je da je napravljen *alias* za korišćenje *sail* komande u narednim primerima.

6.9. Pristup strukturi direktorijuma

U cilju lakšeg pristupa strukturi direktorijuma aplikacije koja se razvija, potrebno je izvršiti dodavanje u alatu *Visual Studio Code* (ranije u ovom poglavlju instaliran je *WSL* dodatak koji to omogućava u okviru okruženja).

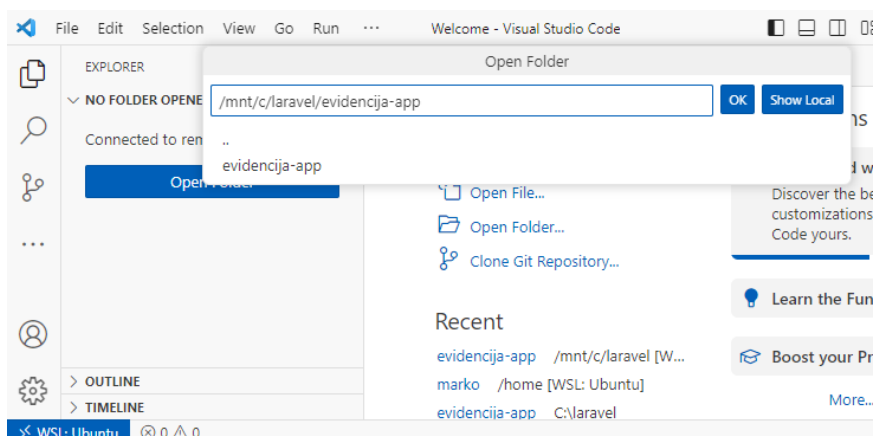
U aplikaciji *Visual Studio Code* (proverite da li je u donjem levom uglu aktivna konekcija *WSL:Ubuntu*) u panelu sa leve strane potrebno je otvoriti prvu opciju – *Explorer* (slika 38).



Slika 38: Visual Studio Code Explorer.

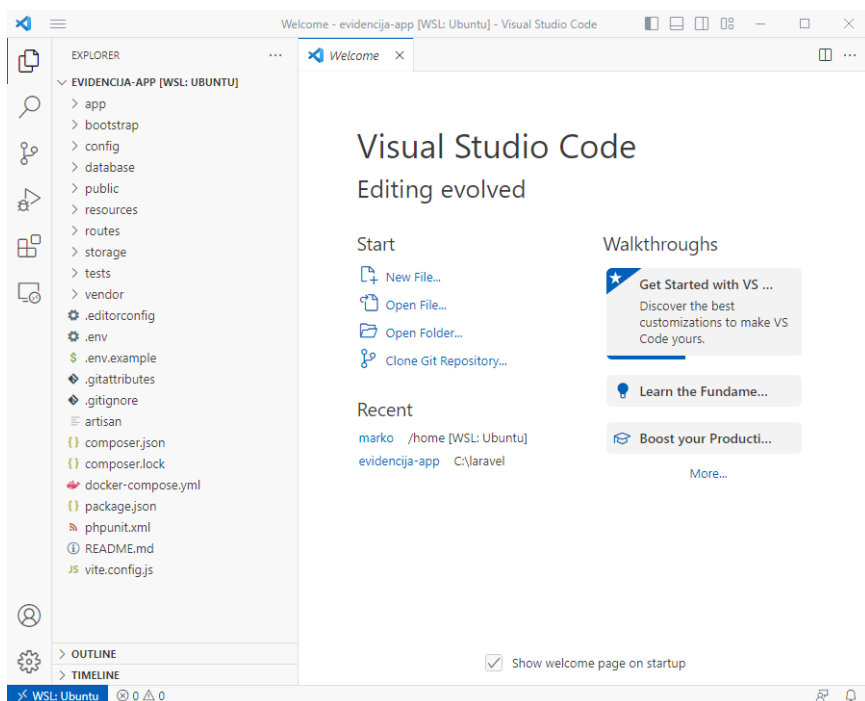
Klikom na dugme Open folder otvara se dijalog u kom je potrebno definisati adresu pomoću koje se pristupa glavnom folderu aplikacije (slika 39) – unosi se adresa koja je već korišćena u terminalu u ranijim sekcijama:

```
/mnt/c/laravel/evidencija-app
```



Slika 39: Unos adrese u Explorer-u.

Nakon što se klikne na dugme *OK*, u panelu *Explorer* se pojavljuje struktura aplikacije *evidencija-app* (slika 40).



Slika 40: Struktura aplikacije *evidencija-app*.

Napomena: u cilju maksimalno efikasnog korišćenje resursa, umesto čuvanja strukture direktorijuma u *Windows* sistemu (i korišćenja *mnt* za pristup), bolje rešenje bi bilo čuvanje u *Linux* sistemu putanja – to su teme koje izlaze iz okvira ovog udžbenika i mogu biti predmet samostalnog rada.

Teme obrađene u ovom poglavlju predstavljaju osnovu daljeg rada sa *Laravel* platformom. Da bi se uspešno isprobale tehnike koje će biti obrađene u narednom poglavlju, potrebno je ispuniti sve korake iz ovog poglavlja, imati pokrenut *WSL:Ubuntu* i napravljen projekat.

7. LARAVEL, OSNOVE KORIŠĆENJA

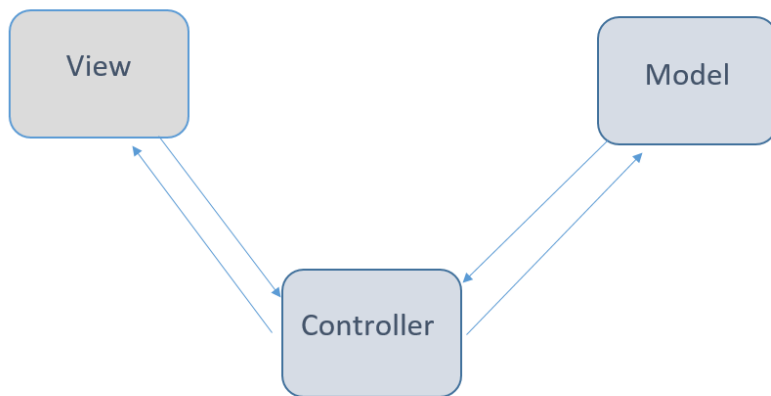
Najvažniji koncepti *Laravel* platforme biće predstavljeni u ovom poglavlju - preduslov da bi sve ispravno radilo jeste da se okruženje konfiguriše na način predstavljen u prethodnom poglavlju. *Laravel* kao platforma ima izuzetno veliki broj mogućnosti, a ovo poglavlje predstavlja najvažnije teme koje je potrebno razumeti kako bi se otpočelo sa razvojem.

7.1. MVC

Rad *Laravel* platforme zasnovan je na *MVC* arhitekturi. *MVC* (*Model-View-Controller*) arhitektura je obrazac softverske arhitekture koji deli aplikacije na tri komponente (slika 41):

- Model (*model*) predstavlja strukturu podataka koja se koristi u aplikaciji, kao i same podatke. Koristi se za dodavanje, izmenu ili uklanjanje podataka iz baze podataka.
- Pogled (*view*) se koristi za korisnički interfejs aplikacije – pogled prikazuje podatke koje je model obezbedio. Pogled ne pristupa direktno modelu, već se za njihovu komunikaciju koristi kontroler.
- Kontroler (*controller*) se ponaša kao posrednik između modela i pogleda. Zadužen je za poslovnu logiku, obradu zahteva i upravljanje podacima korišćenjem modela, a u

saradnji sa pogledom koji će generisati prikaz za krajnjeg korisnika.



Slika 41: MVC arhitektura.

Izvor: towardsdatascience.com

Upotreba MVC arhitekture ima mnogo prednosti:

- omogućen je *lakši razvoj* velikih softverskih sistema;
- postignuto je *logičko razdvajanje komponenti* u aplikaciji – na primer, razdvajanje prikaza podataka od samih podataka;
- *proces razvoja aplikacija* je brži jednostavniji jer se jedna grupa programera može fokusirati na jedan deo sistema (na primer, poslovna logika u kontroleru), dok se druga grupa bavi korisničkim interfejsom (pogledom);
- *održavanje* je lakše jer je struktura aplikacije dobro organizovana.

U narednoj sekciji će biti obrađena struktura direktorijuma u *Laravel* aplikaciji.

7.2. Struktura Laravel aplikacije

Postoji veći broj direktorijuma i datoteka, a u nastavku će biti obrađeni samo oni najvažniji za početak učenja: ¹¹

- *app* – sadrži klase koje su od suštinskog značaja za rad aplikacije. Ovaj direktorijum je i mesto gde se čuvaju klase koje predstavljaju modele (*model*, prvo slovo skraćenice *MVC*).
Direktorijum *app/Http/Controllers* je mesto gde se podrazumevno čuvaju kontroleri aplikacije (*controller*, treće slovo skraćenice *MVC*).
- *bootstrap* – sadrži strukturu koja pokreće aplikaciju. U ovom direktorijumu se izmene prave retko.
- *config* – sadrži konfiguracione datoteke aplikacije na jednom mestu.
- *database* – koristi se za rad sa bazama podataka. U njemu se čuvaju i *migracije* (*migrations*) koje će se koristiti nešto kasnije.
- *public* – sadrži početnu stranu aplikacije, *index.php*. Takođe se koristi i za čuvanje resursa kao što su slike, *CSS* i *JavaScript* fajlovi.
- *resources* – sadrži resurse aplikacije uključujući i poglede (*views*).
- *routes* – sadrži putanje koje se koriste u aplikaciji, a već nekoliko datoteka je napravljeno uz svaki novi *Laravel* projekat.

¹¹ Više podataka o strukturi aplikacije dostupno je na linku <https://laravel.com/docs/10.x/structure>

U narednim sekcijama će biti više reči o korišćenju navedenih direktorijuma.

7.3. Rutiranje u aplikaciji

U direktorijumu *routes*, nalazi se nekoliko *.php* datoteka u kojima se definišu pravila rutiranja u aplikaciji. U nastavku se nalazi početni sadržaj datoteke *web.php* u kojoj se definišu putanje pomoću kojih će se pozivati određene web strane aplikacije (tj. pogledi, *view* – drugo slovo skraćenice MVC):

```
<?php

use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});
```

Ključni deo definisanja nove putanje se nalazi u nekoliko poslednjih redova:

- *Route::get* – poziva se funkcija *get()* u okviru klase *Route* čime se praktično pravi nova putanja.
- *'/'* – označava putanju koja je uneta u pretraživač nakon adrese aplikacije. Kada se koristi znak */* pristupa se početnoj strani aplikacije, u ovom slučaju *localhost/*. Naravno, naziv putanje može biti i drugačiji: ako bi se koristio *'/radnik'* u pozivu funkcije *Route::get*, od korisnika bi se očekivalo da u pretraživaču unese *localhost/radnik*.
- Naredba *return* će biti izvršena samo ako je u pretraživaču pozvana definisana putanja (u ovom slučaju *localhost/*): tada će pomoću funkcije *view()* korisniku biti prikazan pogled pod imenom *welcome*.

Pogled *welcome* je podrazumevano napravljen prilikom pravljenja aplikacije i nalazi se u direktorijumu *resources/views* pod imenom *welcome.blade.php*. Iako je u fajlu *web.php* pogled pozvan samo sa *welcome*, *Laravel* će znati da automatski doda *.blade.php* i pronađe datoteku u navedenom direktorijumu.

Samostalni rad

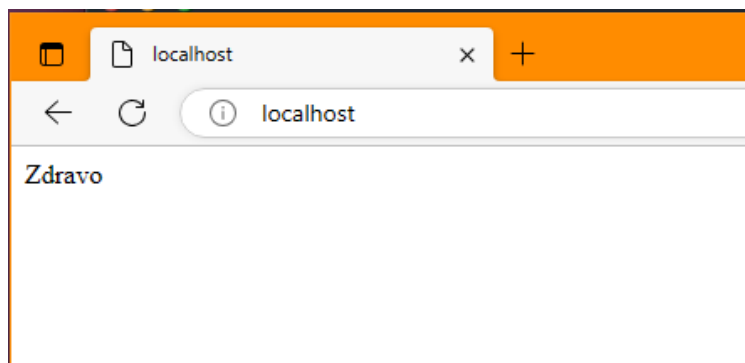
Probajte da napravite neku izmenu u datoteci *welcome.blade.php*. Potom u pretraživaču otkucajte adresu *http://localhost/*. Trebalo bi da, na strani koja se bude otvorila, bude vidljiva izmena napravljena u pogledu *welcome*.

Naredba *sail up* mora biti izvršena u terminalu pre testiranja u pretraživaču.

Naredba *return* je zanimljiva jer može vratiti ne samo naziv pogleda, već i običan *String*.

```
Route::get('/', function () {  
    return 'Zdravo';  
});
```

Slika 42 prikazuje izvršavanje navedene putanje u pretraživaču.

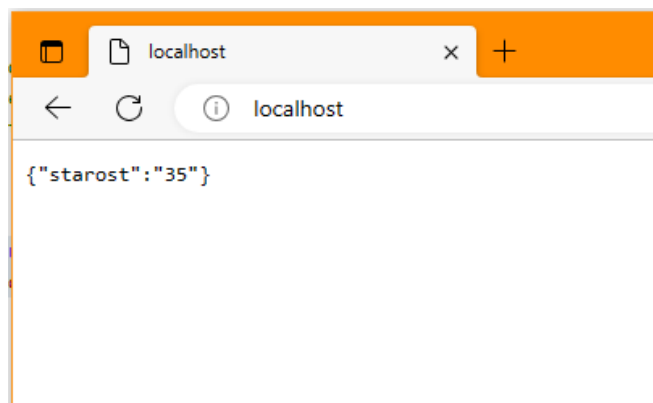


Slika 42: Vraćanje *String* vrednosti prilikom poziva putanje.

Pored String vrednosti, kao rezultat pozivanja putanje, može se vratiti i *JSON* vrednost, što ima izuzetno veliku upotrebnost. Može se napraviti putanja:

```
Route::get('/', function () {  
    return ['starost' => '35'];  
});
```

Nakon pozivanja u pretraživaču, dobija se rezultat kao na slici 43. Kao što se i vidi na osnovu formatiranja prikaza, *Laravel* je razumeo da naredba *return* vraća *JSON* vrednost. U daljem radu ovo može biti veoma korisno – *Laravel* aplikacija može da šalje podatke ka prikazu (*frontend* aplikacije) u *JSON* formatu; prikaz preuzima *JSON* i formatira ga za prikaz pomoću *HTML*-a, *CSS*-a i drugih tehnologija.



Slika 43: Vraćanje *JSON* vrednosti prilikom poziva putanje.

7.4. Blade

U prethodnoj sekciji bilo je reči o pogledima uz prikaz jednog pod imenom *welcome.blade.php* koji se nalazi u direktorijumu:

```
resources/views
```

zajedno sa svim ostalim pogledima koji se koriste u aplikaciji.

Blade je izuzetno koristan mehanizam za pravljenje šablona strana i u okviru *Laravel* platforme, a sve *Blade* naredbe će biti kompajlirane u čist *PHP* programski kod.

Blade dokumenti imaju ekstenziju *.blade.php*.

Kako bi se prikazale mogućnosti *Blade*-a, u datoteci *web.php* posebno je korigovati postojeću putanju tako da izgleda kao u sledećem primeru:

```
Route::get('/', function () {  
    return view('welcome', ['ime' => 'Petar']);  
});
```

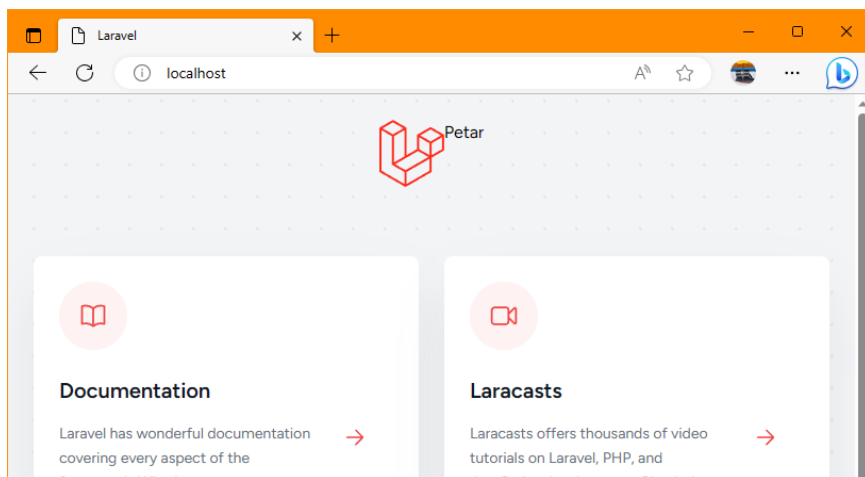
Naredba *return* vraća prikaz po imenu *welcome.blade.php* i tom prikazu prosleđuje novu promenljivu po imenu *ime* i sa dodeljenom vrednošću *Petar*.

Potom je potrebno otvoriti stranu *welcome.blade.php* i u okviru nje dodati narebu koja će prikazati promenljivu *ime*. U ovom slučaju, dodavanje je izvršeno odmah nakon slike u zaglavlju strane:

```
<div class="max-w-7xl mx-auto p-6 lg:p-8">  
  <div class="flex justify-center">  
    <svg viewBox="0 0 62 65" fill="none"  
      <!-- ostatak prikaza svg taga je preskočen -->  
    </svg>  
  
    {{ $ime }}  
  
  </div>
```

Kako bi se podaci navedeni u putanji definisanoj u datoteci *web.php* prikazali, u *welcome.blade.php* fajlu dodata je naredba `{{ $ime }}`. Vrednost koja se upisuje iza znaka *\$* treba da bude ista kao ime

promenljive definisano prilikom pravljenja pogleda. Slika 44 prikazuje rezultat izvršavanja navedenog primera.



Slika 44: Korišćenje Blade naredbe za prikaz vrednosti promenljive.

Prilikom pozivanja naredbe `{{ $imePromenljive }}`, automatski će biti izvršena i *PHP* funkcija `htmlspecialchars()` koja štiti od ubrizgavanja zlonamernog koda kroz sadržaj promenljive.

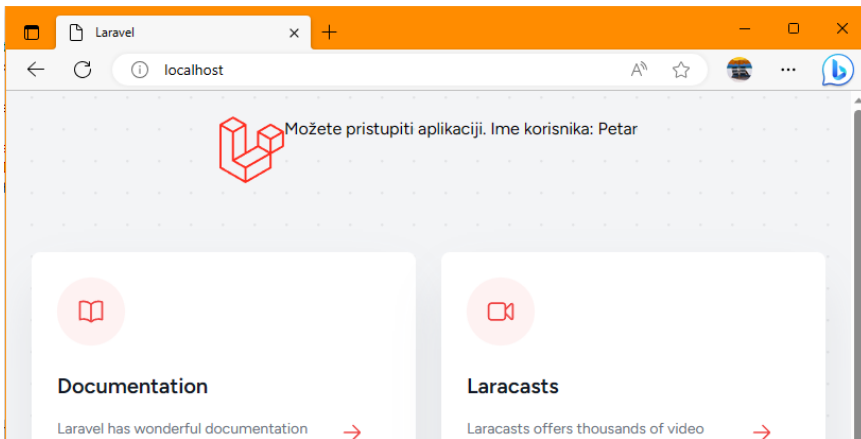
Iako je u prethodnom primeru pogledu poslata samo jedna promenljiva *String* tipa, moguće je slati i veće skupove podataka (nizove), kao i objekte izvedene od klasa koje ste samostalno pravili.

Blade omogućava i pravljenje kompleksnijih struktura. Moguće je praviti i *if-then-else* konstrukcije. Ukoliko se sledeća naredba napiše umesto `{{ $ime }}` u `welcome.blade.php`

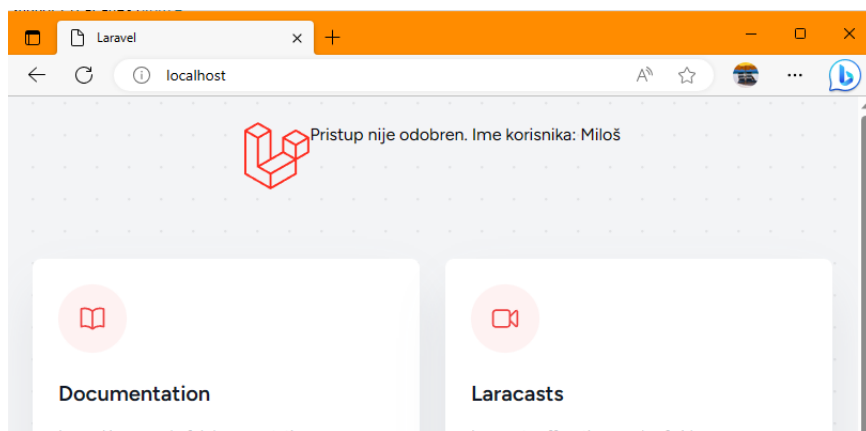
```
<div class="max-w-7xl mx-auto p-6 lg:p-8">
  <div class="flex justify-center">
    <svg viewBox="0 0 62 65" fill="none"
      <!-- ostatak prikaza svg taga je preskočen -->
    </svg>
```

```
@if ($ime == 'Petar')
    Možete pristupiti aplikaciji. Ime korisnika: {{ $ime }}
@else
    Pristup nije odobren. Ime korisnika: {{ $ime }}
@endif
</div>
```

Potrebno je obratiti pažnju da se ime promenljive stavlja između dvostrukih vitičastih zagrada samo onda kada je vrednost potrebno ispisati na ekranu. Ukoliko se ime promenljive koristi, kao u prethodnom primeru, za proveru ispunjenosti uslova, tada se piše bez vitičastih zagrada. Slika 45 prikazuje kako izgleda početna strana aplikacije kada je ime korisnika *Petar*, dok slika 46 prikazuje šta će se dogoditi kada se u pozivu prikaza unese druga vrednost za promenljivu *ime*.



Slika 45: Prikaz strane kada je izvršen then uslov Blade naredbe.



Slika 46: Prikaz strane kada je izvršen *else* uslov *Blade* naredbe.

Moguće je praviti i petlje korišćenjem *Blade* naredbi. U narednom primeru, predstavljene su:

- *for* petlja sa brojačem i koja prikazuje brojeve od 0 do 9;
- *foreach* petlja koja prolazi kroz niz *radnici* i prikazuje ime svakog pojedinačnog radnika;
- beskonačna petlja zasnovana na *while* petlji – izmenom uslova se beskonačna petlja može pretvoriti u *while* petlju koja će se izvršavati tačno određen broj puta.

```
@for ($i=0; $i<10; $i++)  
    Brojač: {{ $i }}  
@endfor
```

```
@foreach ($radnici as $radnik)  
    <span>Ime radnika: {{ $radnik->ime }}</span>  
@endforeach
```

```
@while (true)  
    <span>Beskonačna petlja.</span>  
@endwhile
```

Postoji još dosta *Blade* naredbi (kompletna lista je dostupna u *Laravel* dokumentaciji), još neke će biti prikazane u narednim sekcijama i poglavljima, a za sada će biti obrađene dve – *@isset* i *@empty*, a koriste se na potpuno isti način kao i *PHP* funkcije *isset()* i *empty()*.

```
@isset($ime)
    // promenljiva $ime je definisana i različita je od null
@endisset

@empty($ime)
    // promenljiva $ime je definisana i nije prazna
@endempty
```

Funkcija *@isset* proverava da li je vrednost promenljive *null*. Funkcija *@empty* proverava da li promenljiva “prazna” – pod praznom promenljivom se smatra ukoliko je promenljiva definisana, ali su u nju unete vrednosti kao što je *0*, *0.0*, prazan *String*, prazan niz, *false* itd.

U narednoj sekciji, biće objašnjeno korišćenje *Blade*-a za pravljenje šablona strana.

7.5. Blade i šabloni strana

Pored nekih mogućnosti *Blade*-a prikazanih u prethodnoj sekciji i mnogobrojnih naredbi koje nisu obrađivane, jedna od najvažnijih primena su i šabloni strana – za te potrebe, koriste se naredbe kao što su:

- *@yield('naziv')* označava mesto na kom će biti izvršeno ubacivanje *String* vrednosti u šablonu prilikom pozivanja po vrednosti *naziv*. Ukoliko je potrebno u šablon dodati više od *String*-a (na primer, blok *html* naredbi), u tom slučaju se koristi naredba *@section*. Naredba *@yield* može imati

podrazumevanu *String* vrednost, ali ne i podrazumevanu *html* vrednost.

- `@section('nazivSekcije')` se, kao i `@yield('naziv')`, koristi za ubacivanje sadržaja u šablon na tačno određenom mestu. Ova naredba može sadržati podrazumevane *html* naredbe koje će se prikazati na strani prilikom njenog pozivanja.

Naredba `@section` može da uradi sve što i naredba `@yield`, ali naredba `@yield` ima manje mogućnosti od `@section`. U praksi, koristite `@yield` – ukoliko mogućnost ove naredbe ne budu dovoljne (bude potrebno da se prosledi nešto više od *String* vrednosti), upotrebite `@section`.

Naredba `@section` se uglavnom koristi u kombinaciji sa naredbom `@endsection` ili `@show`.

- `@endsection` označava kraj bloka koji sadrži određenu sekciju bez njegovog prikazivanja na ekranu – sadržaj bloka će biti prikazan tek kada bude pozvan pomoću, na primer, naredbe `@parent`.
- `@stop` ima potpuno istu funkciju kao `@endsection`.
- `@show` je naredba ekvivalentna uzastopnom pozivanju naredbi `@stop` i `@yield('nazivSekcije')`. Ova naredba će završiti započetu sekciju i odmah je prikazati na strani.
- `@extends` služi da određenu stranu “proširimo” mogućnostima koje su definisane u određenom šablonu. Naredba `@extends` označava da “uvozimo” određenu stranu sa šablonom uz mogućnost proširivanja mogućnosti tog šablona.
- `@parent` služi da se pristupi sadržaju komponente “roditelja” u šablonu – ukoliko je u šablonu definisana sekcija, a na određenoj strani pristupimo toj sekciji po njenom imenu bez korišćenja naredbe `@parent`, podrazumevani sadržaj komponente iz šablona će biti uklonjen. Ukoliko se pozove `@parent`, prikazuje se i

podrazumevani sadržaj iz šablona i sadržaj iz strane na kojoj je šablon pozvan.

Kako bi se prikazale mogućnosti navedenih naredbi, potrebno je napraviti novi dokument sa imenom *struktura.blade.php* (u direktorijumu *resources/views*). Ovaj dokument će sadržati šablon koji će biti korišćen za ostale strane u aplikaciji, a osmišljen je tako da prikaže više različitih mogućnosti koje se mogu koristiti. Njegov sadržaj, prikazan je ispod:

```
<html>

<head>
  <title>@yield('naslov')</title>
</head>

<body>

  @section('meni')
    <div style='background-color: lightblue;'>
      Link 1 | Link 2 | Link 3
    </div>
  @show

  <div style='background-color: lightgray;'>
    <p>Podrazumevani sadržaj strane</p>

    @yield('glavni')

  </div>

</body>

</html>
```

Navedeni šablon sadrži nekoliko mesta na kojima je moguće izvršiti promenu prilikom pozivanja šablona i dodati novi sadržaj:

- `@yield('naslov')` je mesto na kom će prilikom poziva šablona biti moguće definisati tačan naziv trenutne strane.
- `@section('meni')` je sekcija koja sadrži podrazumevani meni koji se može prikazati na svakoj strani koja bude pozvala šablon. Obratite pažnju da sekcija sadrži *html* naredbe i da se završava naredbom `@show`.
- `@yield('glavni')` je mesto na kom će prilikom poziva šablona moći da bude dodat sadržaj određene strane.

Kada se ovakav šablon pozove sa određene strane, biće moguće korigovati svaku od definisanih sekcija. U narednoj sekciji naveden je prikaz strane *administracija.blade.php* koja treba da bude smeštena u folderu *resources/views*.

```
@extends('struktura')

@section('naslov', 'Administrativna strana')

@section('meni')
    @parent
    <div>Dodatna linija menija</div>
@endsection

@section('glavni')
    <p>Ovo je sadržaj strane i biće dodat podrazumevanom
    sadržaju koji već postoji u šablonu.</p>
@endsection
```

Naredbe na strani *administracija.blade.php* pozivaju šablon *struktura.blade.php* na sledeći način:

- `@extends('struktura')` definiše koja se strana koristi kao šablon. Ekstenzija *blade.php* se automatski dodaje. Ukoliko

se šablon nalazi izdvojen u poddirektorijumu, u pozivu se putanja može navesti sa `@extends('direktorijum.struktura')`. Obratite pažnju da se koristi `.` umesto `/` za navođenje putanje.

- `@section('naslov', 'Administrativna strana')` omogućava da se tekst `'Administrativna strana'` ubaci na mesto gde se nalazi naredba `@yield('naslov')` u šablonu. Kada strana bude prikazana u pretraživaču, biće prikazan sledeći html:

```
<title>Administrativna strana</title>
```

- `@section('meni')` pristupa sekciji `meni` definisanoj u šablonu, a prati je naredba `@parent` koja će prikazati sadržaj menija definisanog u šablonu uz dodatak sadržaja menija definisanog na trenutnoj strani:

```
<div>Dodatna linija menija</div>
```

U slučaju da naredba `@parent` nije bila pozvana, sadržaj sekcije `meni` u šablonu će biti ignorisan.

- `@section('glavni')` definiše sadržaj koji će biti ubačen u šablon na mestu naredbe `@yield('glavni')`.

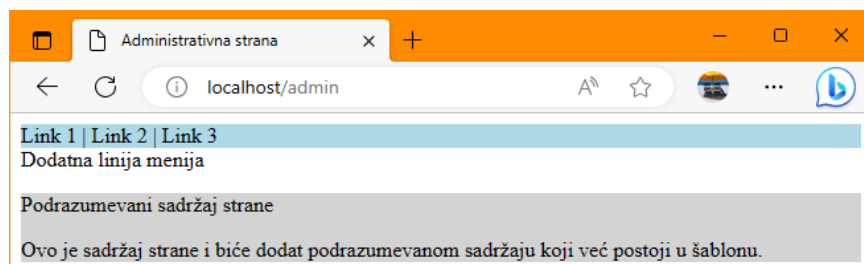
Da bi primer mogao da se testira u pretraživaču, potrebno je u `web.php` fajlu definisati novu putanju:

```
Route::get('/admin', function () {  
    return view('administracija');  
});
```

U putanji se poziva samo krajnja strana, `administracija.blade.php`, a ona u pozadini poziva šablon koji koristi. Strana će se nalaziti na adresi `localhost/admin`.

Napomena: Za krajnjeg korisnika je vidljiv samo link `localhost/admin`, ali on neće znati za postojanje strana `administracija.blade.php` i `struktura.blade.php`. Ovo je još jedna od prednosti korišćenja platforme *Laravel* – detalji realizacije ostaju sakriveni od krajnjih korisnika.

Slika 47 prikazuje izgled strane `administracija.blade.php` nakon pokretanja u pretraživaču.



Slika 47: Izgled strane napravljene prema Blade šablonu.

7.6. PHP Artisan

PHP artisan je koristan alat koji olakšava i automatizuje mnoge aktivnosti u *Laravel*-u. Obzirom da se u ovom udžbeniku koristi *Laravel Sail*, to će morati da se naglasi prilikom pozivanja komande `php artisan` dodavanje prefiksa `./vendor/bin/sail`:

```
./vendor/bin/sail php artisan
```

Obzirom na nečitljivost ovakvog pristupa, savetuje se pravljenje aliasa pomoću komande:

```
alias sail='[ -f sail ] && sh sail || sh vendor/bin/sail'
```

Nakon toga, može se u terminalu koristiti i skraćena verzija:

```
sail php artisan
```

Napomena: U sekciji *Konfigurisanje aliasa za Sail* u prethodnom poglavlju je bilo više reči o korišćenju aliasa za *Laravel Sail*.

Slika 48 prikazuje pravljenje aliasa i pozivanje naredbe *php artisan*:

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ alias sail='[ -f sail ] && sh sail || sh vendor/bin/sail'
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan
Laravel Framework 10.13.5

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display help for the given command. When no command is given display help for the li
st command
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
                     --ansi|--no-ansi Force (or disable --no-ansi) ANSI output
  -n, --no-interaction Do not ask any interactive question
                     --env[=ENV] The environment the command should run under
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output a
Ln 14, Col 62  Spaces: 4  UTF-8  L

```

Slika 48: Pravljenje aliasa i pozivanje naredbe `php:artisan`.

Pri pozivanju naredbe *php artisan*, biće prikazana lista svih komandi koje se mogu koristiti u okviru nje – slika 49 prikazuje prikaz dela liste sa komandama.

```

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
schedule:test      Run a scheduled command
schedule:work      Start the schedule worker
schema
schema:dump        Dump the given database schema
session
session:table      Create a migration for the session database table
storage
storage:link       Create the symbolic links configured for the application
stub
stub:publish       Publish all stubs that are available for customization
vendor
vendor:publish     Publish any publishable assets from vendor packages
view
view:cache         Compile all of the application's Blade templates
view:clear        Clear all compiled view files
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ [

```

Slika 49: Prikaz liste svih komandi `php artisan`.

Ukoliko je potrebno pozvati neku komandu iz ove liste, njeno ime se unosi nakon komande *php artisan*. Na primer, ako je potrebno pozvati komandu *help* kako bi se dobilo više informacija o nekoj drugoj naredbi (na primer, *migrate*), taj poziv bi mogao da izgleda ovako:

sail php artisan help migrate

Slika 50 prikazuje izvršavanje navedene naredbe.

```
marko@DESKTOP-MM:/mnt/c/laravel/evencija-app$ sail php artisan help migrate
Description:
  Run the database migrations

Usage:
  migrate [options]

Options:
  --database[=DATABASE]  The database connection to use
  --force                Force the operation to run when in production
  --path[=PATH]          The path(s) to the migrations files to be executed (multiple
                        values allowed)
  --realpath              Indicate any provided migration file paths are pre-resolved
                        absolute paths
  --schema-path[=SCHEMA-PATH] The path to a schema dump file
  --pretend              Dump the SQL queries that would be run
  --seed                 Indicates if the seed task should be re-run
  --seeder[=SEEDER]      The class name of the root seeder
  --step                 Force the migrations to be run so they can be rolled back in
                        dividually
  --isolated[=ISOLATED] Do not run the command if another instance of the command is
                        already running [default: false]
  -h, --help             Display help for the given command. When no command is given
                        display help for the list command
  -q, --quiet            Do not output any message
  -V, --version          Display this application version
  --ansi|--no-ansi       Force (or disable --no-ansi) ANSI output
  -n, --no-interaction   Do not ask any interactive question
  --env[=ENV]            The environment the command should run under
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 f
                        or more verbose output and 3 for debug
marko@DESKTOP-MM:/mnt/c/laravel/evencija-app$
```

Slika 50: Izvršavanje naredbe *sail php artisan help migrate*.

7.7. Migracije

Migracije predstavljaju jednostavan način za kontrolisanje baza podataka u projektima pravljenjem strukture tabela. Zahvaljujući migracijama, korišćenjem *Laravel* platofme nij lako samo napraviti

novu tabelu, već su izmene strukture postojećih tabela veoma pojednostavljene.

Nakon instalacije novog *Laravel* projekta, već je postavljena osnova za neke migracije koje se mogu koristiti. Sve migracije čuvaju se na putanji:

database/migrations

U svrhu demonstracije, pogledaćemo sadržaj postojećeg dokumenta sa migracijama koji u svoj imenu ima reči *create_users_table.php*. Ovaj dokument sadrži dve funkcije:

- funkciju *up()* koja se koristi da izvrši migraciju;
- funkciju *down()* koja se koristi da poništi efekte pozivanja migracije (odnosno pozivanja funkcije *up()*).

U nastavku je navedeno kako izgleda funkcija *up()* u dokumentu *create_users_table.php*:

```
public function up(): void
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

U okviru funkcije *up()*, poziva se funkcija *create()* klase *Schema*, a kao parametar joj se prosleđuju:

- *naziv tabele* koja se pravi, u ovom slučaju *users*;
- *anonimna funkcija* koja sadrži strukturu tabele koju je potrebno napraviti, a za šta se koristi objekat *\$table* tipa *Blueprint*.

Korišćenjem pomenute anonimne funkcije, pravi se tabela *users*, a u nastavku su objašnjeni najvažniji delovi:

- Funkcija *id()* pravi kolonu *id* sa jedinstvenim identifikatorom za svaki red podataka.
- Funkcija *string()* pravi kolonu koja sadrži podatke tekstualnog tipa: *name*, *email*, *password*.
- Funkcija *unique()* obezbeđuje da svaka vrednost u određenoj koloni mora biti jedinstvena – u ovom slučaju, ne smeju se pojaviti dve iste *email* adrese kod dva različita korisnika.
- Funkcija *timestamp()* pravi kolonu tipa *timestamp* po imenu *email_verified_at*.
- Funkcija *nullable()* označava da navedeno polje može imati i vrednost *null*, tj. nije obavezno popuniti njegovu vrednost. Ova funkcija pozvana za polje *email_verified_at*, obzirom da je logično da je dozvoljeno vreme verifikacije *email* naloga označiti kao *null* sve dok ne bude uneta tačna vrednost (tj, verifikovan mejl).

Funkcija *rememberToken()* pravi kolonu tekstualnog tipa koja će se zvati *remember_token*. U pitanju je sigurnosni mehanizam platforme *Laravel*, a ovu funkciju bi trebalo pozvati za sve tabele koje sadrže liste korisnika.

- Funkcija *timestamps()* služi za automatsko beleženje najvažnijih događaja u određenoj tabeli. Nakon njenog izvršavanja, u tabeli će se pojaviti dve nove kolone: *created_at* i *updated_at*.

Funkcija *down()* je veoma jednostavna i trenutno se svodi samo na pozivanje funkcije *dropIfExists()* koja uklanja tabelu čije je ime prosleđeno kao parametar:

```
public function down(): void
{
    Schema::dropIfExists('users');
}
```

Da bi se testirao rad migracija, mogu se izvršiti odgovarajuće *php artisan* komande koje će automatski pozivati funkcije *up()* i *down()*, a prema potrebi. Najjednostavniji način je pozvati komandu *migrate:refresh* (slika 51) koja će za sve migracije prvo pozvati funkciju *down()*, a potom izvršiti funkciju *up()* za svaku od njih.

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan migrate:refresh

[INFO] Rolling back migrations.

2019_12_14_000001_create_personal_access_tokens_table ..... 76ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 33ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 34ms DONE
2014_10_12_000000_create_users_table ..... 26ms DONE

[INFO] Running migrations.

2014_10_12_000000_create_users_table ..... 83ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 100ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 98ms DONE
2019_12_14_000001_create_personal_access_tokens_table ..... 106ms DONE

marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 51: Pozivanje komande *php artisan migrate:refresh*.

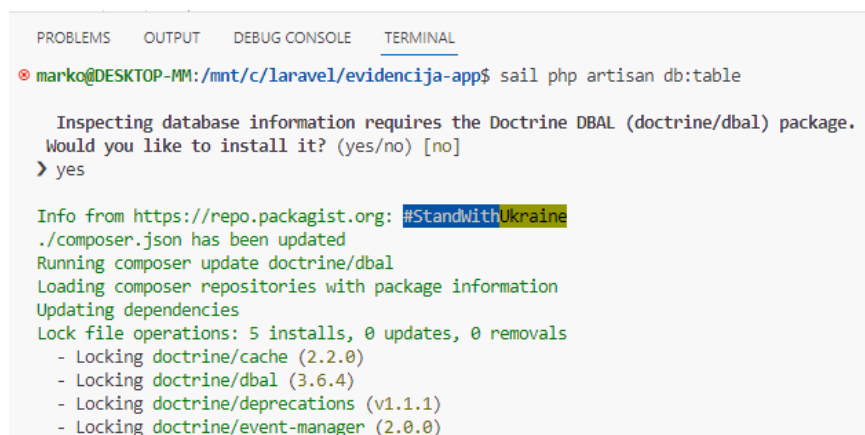
Nakon toga, može se pogledati i struktura tabela. Kao što se i vidi na prethodnoj slici, migracija *create_users_table* je izvršena. Ime tabele koja je napravljena će biti samo *users*.

Da bi se pregledala struktura određene tabele, mogu se koristiti razni *MySQL* alati poput *phpmyadmin*, a može se koristiti i *Visual*

Studio Code terminal, kao u sledećem primeru korišćenjem komande:

```
sail php artisan db:table
```

Prilikom prvog pozivanja ove komande, *Visual Studio Code* će možda zahtevati instalaciju paketa potrebnih za rad sa bazama podataka. Ukoliko se to dogodi, potrebno je uneti opciju *yes* i dozvoliti instalaciju, kao na slici 52.



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan db:table

Inspecting database information requires the Doctrine DBAL (doctrine/dbal) package.
Would you like to install it? (yes/no) [no]
> yes

Info from https://repo.packagist.org: #StandWithUkraine
./composer.json has been updated
Running composer update doctrine/dbal
Loading composer repositories with package information
Updating dependencies
Lock file operations: 5 installs, 0 updates, 0 removals
- Locking doctrine/cache (2.2.0)
- Locking doctrine/dbal (3.6.4)
- Locking doctrine/deprecations (v1.1.1)
- Locking doctrine/event-manager (2.0.0)
```

Slika 52: Instalacija potrebnih paketa za rad sa bazama podataka.

Nakon instalacije, moguće je koristiti *db:table* naredbu – ona kao parametar prihvata naziv tabele, u ovom slučaju je to *users*. Slika 53 prikazuje rezultat izvršavanja. Nazivi kolona i njihovi parametri će odgovarati onome što je definisano u funkciji *up()* za migraciju *create_users_table*.

```

● marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan db:table users

users .....
Columns ..... 8
Size ..... 0.02MiB

Column ..... Type
id autoincrement, bigint, unsigned ..... bigint
name string ..... string
email string ..... string
email_verified_at datetime, nullable ..... datetime
password string ..... string
remember_token string, nullable ..... string
created_at datetime, nullable ..... datetime
updated_at datetime, nullable ..... datetime

Index .....
PRIMARY id ..... unique, primary
users_email_unique email ..... unique

○ marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$

```

Slika 53: Rezultat izvršavanja naredbe `db:table`.

Ukoliko bude potrebe za naknadnom izmenom migracija u vidu izmene naziva kolone, dodavanja kolona ili izmene parametara, potrebno je sačuvati migraciju i ponovo izvršiti komandu *migrate:refresh*. Na primer, u funkciji *up()* za migraciju *users*, možemo promeniti kolonu:

```
$table->string('name');
```

u

```
$table->string('ime');
```

Potom u terminalu treba izvršiti naredbu *migrate:refresh* i nakon toga pregledati strukturu tabele pomoću *db:table*. Slika 54 predstavlja novu strukturu tabele – obratite pažnju na naziv druge kolone koja se sada zove *ime*.

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan db:table users

users .....
Columns ..... 8
Size ..... 0.02MiB

Column ..... Type
id autoincrement, bigint, unsigned ..... bigint
ime string ..... string
email string ..... string
email_verified_at datetime, nullable ..... datetime
password string ..... string
remember_token string, nullable ..... string
created_at datetime, nullable ..... datetime
updated_at datetime, nullable ..... datetime

Index .....
PRIMARY id ..... unique, primary
users_email_unique email ..... unique

marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 54: Nova struktura tabele users nakon izmene migracije.

Naredba *migrate:refresh* nije jedina mogućnost izvršavanja migracija. Pozivanje komande *php artisan* će izlistati sve komande za migracije (slika 55).

```
migrate
migrate:fresh      Drop all tables and re-run all migrations
migrate:install    Create the migration repository
migrate:refresh    Reset and re-run all migrations
migrate:reset      Rollback all database migrations
migrate:rollback   Rollback the last database migration
migrate:status     Show the status of each migration
```

Slika 55: Prikaz liste svih migrate naredbi.

Na primer, moguće je umesto naredbe *migrate:refresh* pozvati naredbu *migrate:rollback*, a potom izvršiti *migrate:install*.

7.7.1. Pravljenje nove migracije

U prethodnim primerima, kao osnova su korišćene postojeće migracije koje su unapred napravljene u svakom novom *Laravel*

projektu. Ali i pravljenje nove migracije je jednostavan postupak za koji se koristi naredba *make:migration*:

```
sail php artisan make:migration create_test_table
```

Navedena naredba će napraviti novu migraciju koja će u imenu sadržati naziv *create_test_table*. Nakon toga, potrebno je izvršiti:

```
sail php artisan migrate:refresh
```

Slika 56 prikazuje kako izgleda terminal nakon izvršenja naredbe *migrate:refresh*. Obratite pažnju da je operacija *rollback* urađena za četiri tabele, a da su potom izvršene migracije za pet, od kojih je jedna nova – *create_test_table*. Tabela koja će biti napravljena u bazi podataka na osnovu ove migracije zvaće se *test*.

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan migrate:refresh

[INFO] Rolling back migrations.

2019_12_14_000001_create_personal_access_tokens_table ..... 183ms
2019_08_19_000000_create_failed_jobs_table ..... 26ms
2014_10_12_100000_create_password_reset_tokens_table ..... 25ms
2014_10_12_000000_create_users_table ..... 28ms

[INFO] Running migrations.

2014_10_12_000000_create_users_table ..... 131ms
2014_10_12_100000_create_password_reset_tokens_table ..... 289ms
2019_08_19_000000_create_failed_jobs_table ..... 701ms
2019_12_14_000001_create_personal_access_tokens_table ..... 314ms
2023_07_04_074455_create_test_table ..... 48ms

marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 56: Izvršavanje naredbe *migrate:refresh* nakon pravljenja nove migracije.

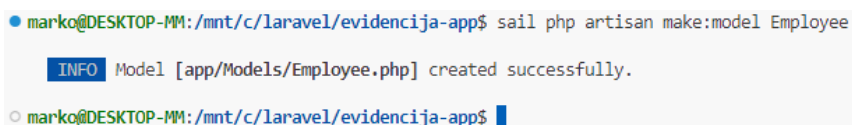
7.8. Laravel modeli

Modeli su prva komponenta u imenu MVC arhitekture. U *Laravel*-u, modele predstavljaju *PHP* klase koje predstavljaju interakciju između kontrolera i baze podataka.

Da bi se model napravio, može se koristiti naredba *make:model* kao u sledećem primeru:

```
sail php artisan make:model Employee
```

Navedena naredba će napraviti novi model pod imenom *Employee*. Slika 57 prikazuje kako izgleda izvršavanje ove naredbe u terminalu.



```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan make:model Employee
INFO Model [app/Models/Employee.php] created successfully.
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 57: Izvršavanje naredbe *make:model*.

Novi modeli se smeštaju u direktorijumu *app/Models*. U nastavku je prikazan podrazumevani sadržaj novog modela *Employee*:

```
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Employee extends Model
{
    use HasFactory;
}
```

Model se koristi za rad sa bazama podataka, a *Laravel* će sakriti veliki deo komunikacije koji se događa u pozadini. Model je u direktnoj vezi sa migracijama: ako se model zove *Employee*, onda bi tabela koju će migracija napraviti trebala da se zove *employees*. Naravno, u modelu je moguće definisati i da će se tabela zvati potpuno drugačije tako što se definiše nova promenljiva u klasi koja predstavlja model:

```
protected $table = 'zaposleni';
```

Na ovaj način, iako se model zove *Employee*, pristupaće tabeli koja se zove *zaposleni* (a ne *employees*). Potrebno je obazrivo koristiti mogućnost za promenu imena tabele, jer ona povlači i neke druge promene (na primer, u migracijama).

7.9. Laravel kontroleri

Kontroleri su treća komponenta u imenu MVC arhitekture koja služi za organizovanje funkcionalnosti aplikacije. Oni u okviru jedne klase grupišu poslovnu logiku koja se tiče određenog entiteta. Na primer, kompletne funkcionalnosti koje se tiču rada sa korisnicima bi trebale da budu grupisane u okviru kontrolera *UserController* – to mogu biti CRUD operacije (Create, Read, Update, Delete) nad korisnicima.

Da bi se kontroler napravio, može se koristiti naredba *make:controller* kao u sledećem primeru:

```
sail php artisan make:controller UserController
```

Svi kontroleri se podrazumevano čuvaju u direktorijumu *app/Http/Controllers*. Slika 58 prikazuje kako izgleda izvršavanje naredbe *make:controller* u terminalu.

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan make:controller UserController
INFO Controller [app/Http/Controllers/UserController.php] created successfully.
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 58: Izvršavanje naredbe *make:controller*.

U nastavku je prikazan podrazumevani sadržaj novog kontrolera *UserController*.

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UserController extends Controller
{
    //
}
```

U narednom poglavlju će rad sa kontrolerima biti detaljnije objašnjen na praktičnom primeru.

8. LARAVEL, PRAKTIČAN PROJEKAT

U ovom poglavlju biće napravljena jednostavna aplikacija, čiji je zadatak da demonstrira načine na koje *Laravel* sakriva delove tehničke realizacije i pojednostavljuje razvoj softvera, ali i načine na koje je primenjena *REST* arhitektura. Ovo poglavlje predstavlja praktičnu primenu znanja stečenih u ovom udžbeniku, a dobro razumevanje rada veb servisa i *REST* arhitekture obrađivanih u prethodnim poglavljima predstavlja odličnu osnovu za shvatanje načina na koji *Laravel* funkcioniše.

Izrada ove *Laravel* aplikacije može započeti i kao potpuno novi projekat, a može se nastaviti i kao nadogradnja projekta iz prethodnog poglavlja. Kako bi sve bilo objedinjeno, u ovom udžbeniku će se novi projekat nadovezati na ranije započet projekat.

Osnovni entitet sa kojim će aplikacija raditi jeste radnik, a radiće se sa istim vrstama podataka koji su već korišćeni prilikom izrade *REST* aplikacije u poglavlju 5.

8.1. Pravljenje početne strane

Izrada projekta započinje od novog pogleda *index.blade.php* koji treba napraviti u direktorijumu *resources/views*. Ovaj pogled biće nova početna strana aplikacije - kada joj korisnik pristupi,

prikazaće mu se lista svih radnika koji postoje u sistemu, kao i opcije dodavanja, brisanja i modifikovanja podataka o o radnicima.

Početna strana koristi šablon *struktura.blade.php* koji je definisan u prethodnom poglavlju.

```
@extends('struktura')

@section('naslov', 'Početna strana')

@section('meni')
    @parent
@endsection

@section('glavni')

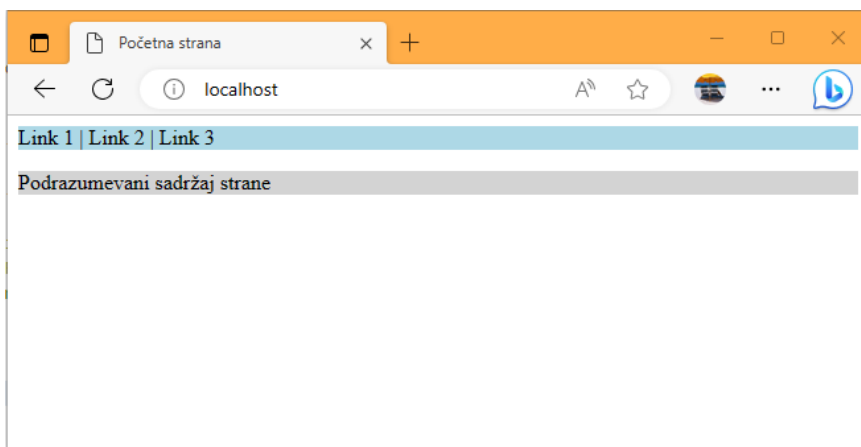
    <!-- Mesto gde će se prikazati lista radnika -->

@endsection
```

U konfiguracionoj datoteci *web.php* se dodaje nova putanja za pristup početnoj strani:

```
Route::get('/', function () {
    return view('index');
});
```

Potom je potrebno početnu stranu (*localhost/*) pozvati u pretraživaču – izgled prve verzije početne strane prikazan je na slici 59.



Slika 59: Prva verzija početne strane aplikacije.

8.2. Pravljenje modela, kontrolera i migracija

Potrebno je napraviti model, kontroler i migraciju, a to se može uraditi ili pojedinačnim pozivanjem komandi:

```
sail php artisan make:model Worker  
sail php artisan make:controller WorkerController  
sail php artisan make:migration create_worker_table
```

ili pozivanjem objedinjene naredbe:

```
sail php artisan make:model Worker -mc
```

Ime modela je zapravo ime *PHP* klase. Trebalo bi da bude u jednini i piše se početnim velikim slovom. Komanda *-mc* označava pravljenje migracije i kontrolera (*migration, controller*). Pogodnije je nazvati model pomoću reči na engleskom jeziku, zato što će *Laravel* na određenim mestima napraviti množinu te reči – na primer, tabelu u kojoj se čuvaju radnici će nazvati *workers*. Zbog

toga je, u ovom slučaju, model nazvan *Worker* umesto *Radnik*. Slika 60 prikazuje izvršavanje naredbe *make:model*.

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan make:model Worker -mc

[INFO] Model [app/Models/Worker.php] created successfully.

[INFO] Migration [database/migrations/2023_07_04_123638_create_workers_table.php] created successfully.

[INFO] Controller [app/Http/Controllers/WorkerController.php] created successfully.

marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 60: Izvršavanje naredbe *make:model*.

Automatski je napravljen model *Worker*, kontroler *WorkerController*, kao i migracija po imenu *create_workers_table.php*. Ukoliko za nazive modela i kontrolera koristite engleske nazive, i množina će biti napravljena pravilno (da je model nazvan *Radnik*, podrazumevani naziv tabele bi postao *radniks*). Naravno, u modelu je uvek moguće i eksplicitno definisati naziv tabele i ona tada može biti nazvana i potpuno drugačije.

Za podatke o radnicima, koristiće se isti podaci kao i u poglavlju 5 u kom je pravljen *REST* aplikacija:

- ime,
- plata i
- starost.

U migraciji *create_workers_table.php* su podrazumevano definisane dve početne vrednosti u funkciji *up()*:

```
$table->id();
$table->timestamps();
```

Vrednost *id* bi trebala da postoji u svakoj tabeli u bazi podataka, zato će poziv funkcije *id()* ostati na prvom mestu.

Vrednost *timestamps* takođe treba ostaviti jer može biti korisna u budućnosti, mada se ove vrednosti neće koristiti u ovom poglavlju.

Pored *id* i *timestamps*, dodaju se nova polja za *ime*, *platu* i *starost*. Novi sadržaj funkcije *up()* sledi:

```
public function up(): void
{
    Schema::create('radnici', function (Blueprint $table) {
        $table->id();
        $table->string('ime');
        $table->integer('plata');
        $table->integer('starost');
        $table->timestamps();
    });
}
```

Ime je po tipu *string*, *plata* i *starost* su *integer*. U realnoj aplikaciji, *starost* ne bi bila sačuvana kao zasebno polje u tabeli, već bi se čuvao datum rođenja radnika, a *starost* bi se prema potrebi izračunavala.

Nakon izmene funkcije *up()*, potrebno je pozvati naredbu *migrate:refresh* da bi se nova migracija izvršila:

```
sail php artisan migrate:refresh
```

Druga opcija je da se umesto *migrate:refresh* pozove samo:

```
sail php artisan migrate
```

Slika 61 prikazuje izvršenje naredbe *migrate*.

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan migrate

INFO Running migrations.

2014_10_12_000000_create_users_table ..... 183ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 172ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 88ms DONE
2019_12_14_000001_create_personal_access_tokens_table ..... 178ms DONE
2023_07_04_074455_create_test_table ..... 29ms DONE
2023_07_04_123638_create_workers_table ..... 68ms DONE

marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 61: Izvršenje naredbe migrate.

Nakon toga, u bazi podataka bi trebala da se pojavi nova tabela, *workers*. Korišćenjem naredbe *db:table* možemo pregledati strukturu nove tabele (slika 62).

```
marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$ sail php artisan db:table workers

workers .....
Columns ..... 6
Size ..... 0.02MiB

Column ..... Type
id autoincrement, bigint, unsigned ..... bigint
ime string ..... string
plata integer ..... integer
starost integer ..... integer
created_at datetime, nullable ..... datetime
updated_at datetime, nullable ..... datetime

Index .....
PRIMARY id ..... unique, primary

marko@DESKTOP-MM:/mnt/c/laravel/evidencija-app$
```

Slika 62: Pregled strukture tabele workers.

8.3. Povezivanje početne strane sa kontrolerom

Na početku ovog poglavlja napravljena je putanja u *web.php*:

```
Route::get('/', function () {
    return view('index');
});
```

unosom putanje *localhost/* u pretraživaču, prikazivala se direktno početna strana (*index.blade.php*), a sada će biti dodat još jedan korak između. Navedenu putanju je potrebno promeniti u:

```
Route::get('/', 'App\Http\Controllers\WorkerController@index');
```

unosom putanje *localhost/* u pretraživaču, od sada će se prvo pozivati kontroler zadužen za radnike (odnosno, *WorkerController*), a u okviru njega će biti pozvana funkcija *index()*. Obratite pažnju da se navodi cela putanja ka nazivu kontrolera (ova putanja odgovara strukturi direktorijuma). Naredni korak je da u ovom kontroleru napravimo funkciju *index()*.

Na putanji *app/Http/Controllers*, u u prethodnoj sekciji automatski je (prilikom pozivanja naredbe *make:model*) napravljen kontroler po imenu *WorkerController* čija je struktura navedena u nastavku:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class WorkerController extends Controller
{
    //
}
```

Kontroler *WorkerController* će sadržati sve funkcije vezane za entitet *Radnik*, a sada je u okviru njega potrebno dodati novu funkciju po imenu *index()* koja izgleda kao u nastavku:

```
class WorkerController extends Controller
{
    public function index()
    {
        $workers = \App\Models\Worker::all();
    }
}
```

```
        return view('index', compact('workers'));
    }
}
```

u promenljivoj *\$workers* čuva se rezultat izvršavanja *Laravel* funkcije *all()* za klasu *Worker* – rezultat njenog rada će biti automatsko preuzimanje podataka o svim radnicima iz tabele radnici bez potrebe da pišemo i izvršavamo upite direktno nad tabelom u bazi podataka.

- naredba *return view()* vraća novi pogled (što je pre rađeno direktno u *web.php* dokumentu) po imenu *index.blade.php* i prosleđuje mu promenljivu koja sadrži sve radnike preuzete iz baze podataka.

Naredni korak je da se na pogledu *index* (tj. početnoj strani aplikacije) dopuni sekcija *glavni*.

```
@section('glavni')

<!-- Mesto gde će se prikazati lista radnika -->

<h1>Prikaz liste radnika</h1>
<ul>
    @foreach ($workers as $worker)
        <li>

            <strong>
                {{ $worker->ime }}
            </strong>

            <br>

            <p>Plata: {{ $worker->plata }} EUR</p>

            <p>Starost: {{ $worker->starost }} godina</p>

        </li>
    @endforeach
</ul>
</section>
```

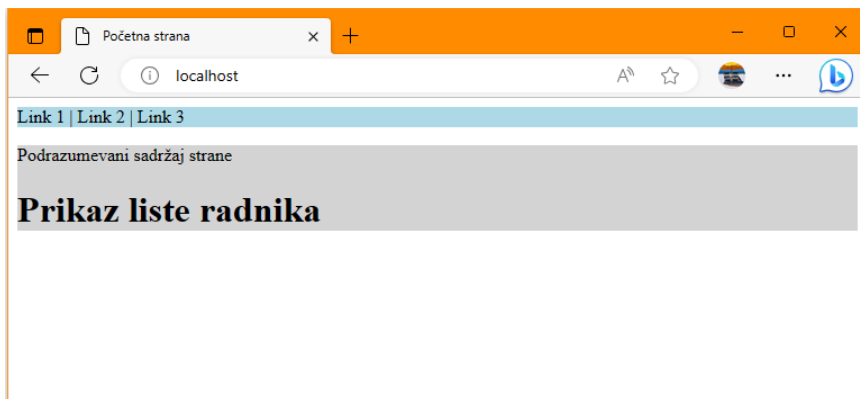
```
</li>
@endforeach
</ul>

@endsection
```

Struktura strane je prilično jednostavna, a najvažnije strukture su:

- petlja *@foreach* koja uzima niz *\$workers* i učitava jednog po jednog radnika u promeljivu *\$worker*;
- naredba *{{ \$worker->ime }}* koja u svakoj iteraciji petlje ispisuje *ime* radnika – slična konstrukcija se koristi i za ispis vrednosti *plata* i *starost*.

Slika 63 pokazuje kako izgleda početna strana kada se pokrene u pretraživaču. Obzirom da trenutno u tabeli *workers* nije dodat nijedan radnik, lista će biti prazna.



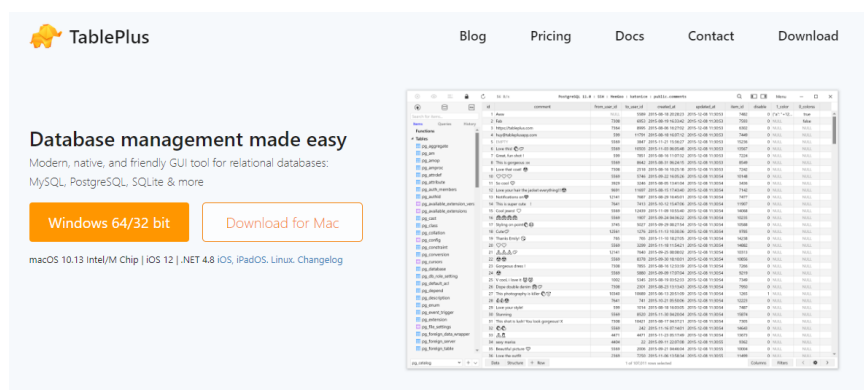
Slika 63: Izgled početne strane uz prikaz prazne liste radnika.

U narednoj sekciji biće predstavljeno manuelno dodavanje novog radnika u tabelu *workers*, kako bi bilo moguće testirati prikaz na početnoj strani.

8.4. Manuelno dodavanje novog zapisa u bazu podataka

Obzirom da aplikacija još uvek nema formu pomoću koje je kroz korisnički interfejs moguće dodati novog radnika u sistem, u svrhe testiranja će to morati da bude urađeno manuelno u bazi podataka.

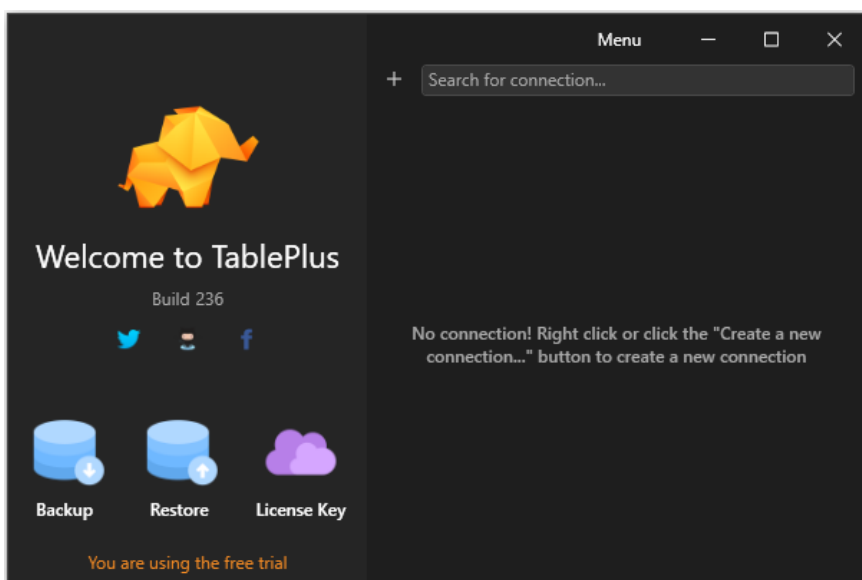
Postoji više alata pomoću kojih se može pristupiti bazi podataka, a kao veoma koristan program, u ovom udžbeniku biće korišćen *TablePlus*. Slika 64 prikazuje početnu veb stranu sa koje se alat može besplatno preuzeti i koristiti.¹²



Slika 64: Početna strana tableplus.com.

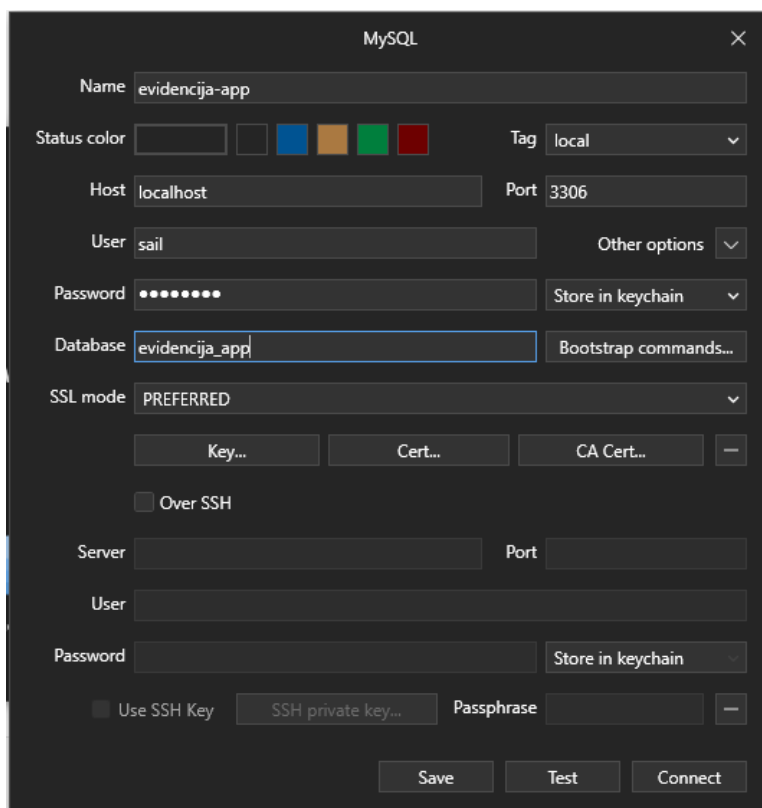
Nakon što se softver preuzme i instalira, početni ekran će izgledati kao na slici 65.

¹² Dostupno za preuzimanje na adresi tableplus.com



Slika 65: Početni ekran alata TablePlus.

Sledeći korak je dodavanje nove konekcije ka *MySQL* bazi podataka. Na početnom ekranu, potrebno je kliknuti znak + i izabrati *MySQL*. Na ekranu koji se otvori (slika 66), potrebno je uneti parametre konekcije.

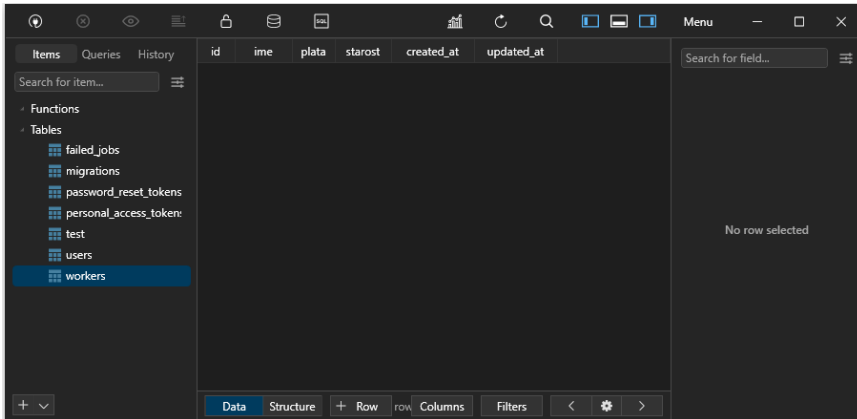


Slika 66: Podešavanje parametara konekcije ka bazi podataka.

Ukoliko niste sigurni koje parametre treba uneti, možete ih pronaći u glavnom direktorijumu *Laravel* aplikacije u dokumentu koji se zove *.env* – ovaj dokument sadrži kompletnu konfiguraciju projekta. U njemu postoji deo koji je automatski generisan prilikom pravljenja projekta:

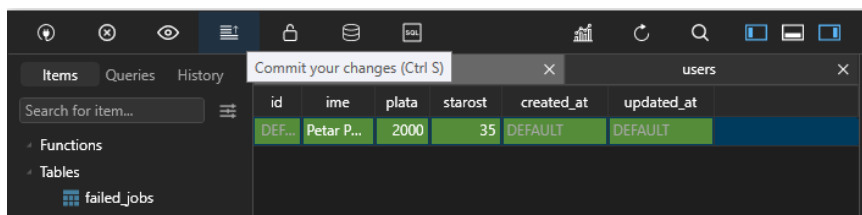
```
DB_CONNECTION=mysql
DB_HOST=mysql
DB_PORT=3306
DB_DATABASE=evidencija_app
DB_USERNAME=sail
DB_PASSWORD=password
```

Nakon što se ovi parametri unesu na ekran sa prethodne slike 66, moguće je proveriti da li je sve u redu klikom na dugme *Test*, a potom se i povezati sa bazom podataka klikom na dugme *Connect*. Slika 67 prikazuje ekran koji će se otvoriti nakon povezivanja na bazu podataka.



Slika 67: Prikaz baze podataka nakon uspostavljanja konekcije.

Nakon što je napravljena veza ka bazi podataka, može se pristupiti dodavanju novog radnika. Sa leve strane, u listi svih tabela treba izabrati tabelu *workers*. Potom na dnu ekrana treba kliknuti na dugme *+ Row*. Nakon toga će biti moguće dodavanje novih vrednosti. Nije potrebno unositi *id* (jer će biti automatski generisan), za sada je dovoljno samo popuniti polja *ime*, *plata* i *starost*. Potom u gornjem levom delu ekrana treba kliknuti na dugme *Commit your changes* (slika 68). Ukoliko se ne izabere ova opcija nakon obavljenog unosa, podaci neće biti sačuvani u bazi podataka.



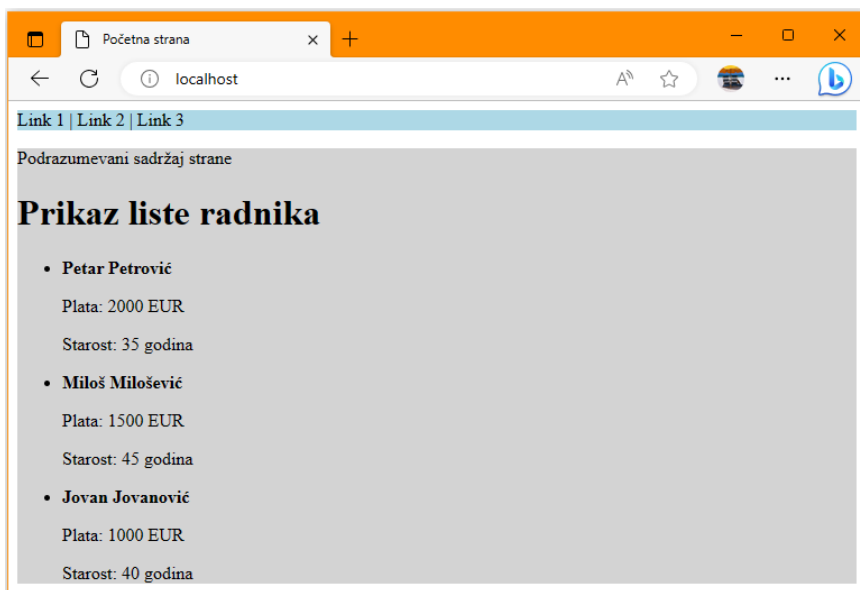
Slika 68: Priprihvatanje unetih izmena u tabeli workers.

Slika 69 prikazuje kako izgleda sadržaj tabele *workers* nakon završenog unosa za tri radnika. Kao što se i može videti, polje *id* je automatski dobilo vrednost. Polja *created_at* i *updated_at* su za sada prazna (*null*).

id	ime	plata	starost	created_at	updated_at
1	Petar P...	2000	35	NULL	NULL
2	Miloš...	1500	45	NULL	NULL
3	Jovan J...	1000	40	NULL	NULL

Slika 69: Sadržaj tabele *workers* nakon završenog unosa za tri radnika.

Nakon završenog dodavanja radnika, potrebno je ponovo pokrenuti aplikaciju u pretraživaču. Slika 70 prikazuje izgled početne strane na kojoj se sada prikazuju radnici koji su dodati u bazu podataka.



Slika 70: Prikaz liste svih radnika na početnoj strani aplikacije.

8.5. Prikazivanje pojedinačnih zapisa iz baze podataka

Obzirom da se broj funkcionalnosti u aplikaciji povećava, potrebno je napraviti malu reorganizaciju da bi svi pogledi koji se bave istim entitetom (u ovom slučaju sve vezano za radnike) bilo grupisano na istom mestu.

U direktorijumu *resources/views* potrebno je napraviti novi poddirektorijum *workers*. Potom treba prebaciti pogled *index.blade.php* u ovaj direktorijum.

```
Route::get('/workers/{id}',  
'App\Http\Controllers\WorkerController@show');
```

U funkciji *index()* u kontroleru *WorkerController* potrebno je dopuniti putanju do pogleda *index* jer se sada nalazi u poddirektorijumu *workers*. Ime pogleda postaje *workers.index*:

```
public function index()
{
    $workers = \App\Models\Worker::all();

    return view('workers.index', compact('workers'));
}
```

Naredni korak je pravljenje funkcije *show()* u kontroleru *WorkerController* čiji je posao prikazivanje podataka o samo jednom radniku na osnovu definisanog *id*-a.

```
public function show($id)
{
    $worker = \App\Models\Worker::find($id);

    return view('workers.show', compact('worker'));
}
```

Funkcija *find(\$id)* na osnovu *id*-a u bazi podataka nalazi traženog radnika i njegove podatke čuva u promenljivoj *\$worker*.

Nakon toga se poziva novi pogled *workers.show* (pogled se zove *show*, a nalazi se u direktorijumu *workers*) i prosleđuje mu se sadržaj promenljive *\$worker*.

Da bi pogled *workers.show* bio uspešno pozvan, potrebno ga je napraviti pod imenom *show.blade.php* na putanji *resources/views/workers*. Njegov sadržaj je skoro isti kao i strana *index*, samo što nema potrebe za *@foreach* petljom jer pogled *show* ispisuje samo jedan rezultat iz baze podataka.

Sadržaj strane *show.blade.php* je prikazan u nastavku:

```
@extends('struktura')

@section('naslov', 'Prikaz podataka o radniku')

@section('meni')
    @parent
@endsection

@section('glavni')

    <!-- Prikaz podataka o pojedinačnom radniku -->
    <h1>Podaci o izabranom radniku</h1>

    <strong>
        {{ $worker->ime }}
    </strong>

    <br>

    <p>Plata: {{ $worker->plata }} EUR</p>

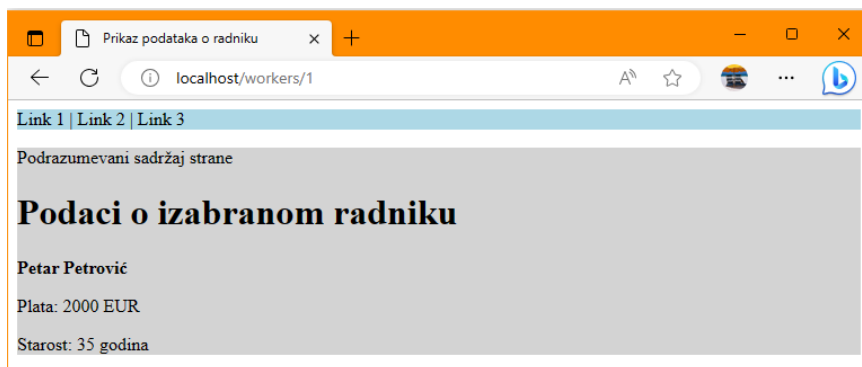
    <p>Starost: {{ $worker->starost }} godina</p>

@endsection
```

Da bi se testiralo da li sve funkcioniše, potrebno je u pretraživaču pozvati putanju:

```
localhost/workers/{id}
```

gde se *{id}* menja *id*-em radnika čije podatke je potrebno prikazati. Slika 71 prikazuje prikaz podataka o radniku čiji je *id* broj 1.



Slika 71: Prikaz podataka o pojedinačnom radniku na osnovu vrednosti id.

Napomena: Obratite pažnju da se za pristup podacima zapravo koriste REST adrese definisane prema pravilima koja su obrađivana u prethodnim poglavljima.

U praktičnim primenama, krajnji korisnik neće unositi direktno adresu u pretraživač, niti ima potrebe da zna *id* radnika čije podatke želi da pregleda. Zato je na početnoj strani potrebno linkovati *ime* radnika i povezati ga direktno sa pogledom *show*.

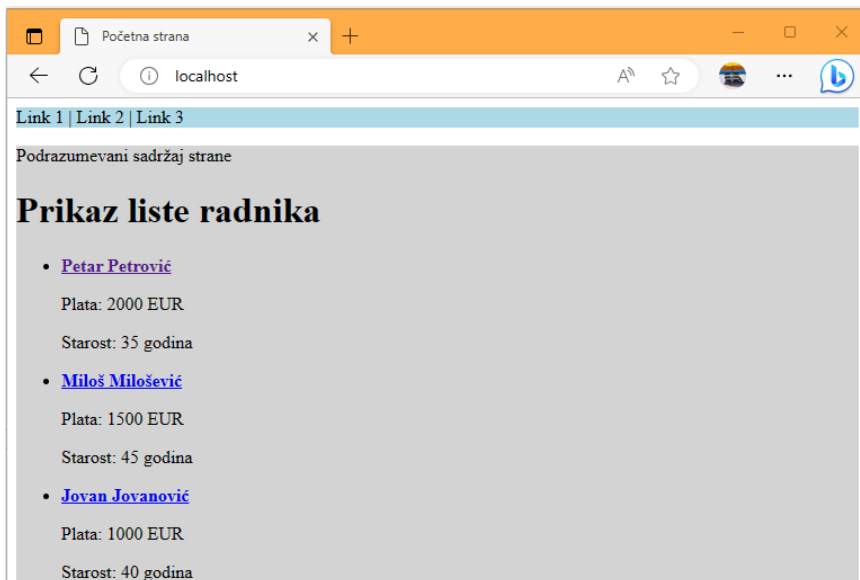
U pogledu *index.blade.php* menja se deo koji ispisuje *ime* radnika:

```
<strong>
    {{ $worker->ime }}
</strong>
```

tako što se umesto jednostavnog ispisa imena generiše link kojim se pristupa prikazu podataka o izabranom radniku na osnovu *id*-a:

```
<strong>
    <a href="/workers/{{ $worker->id }}">
        {{ $worker->ime }}
    </a>
</strong>
```

Nakon pokretanja početne strane u pretraživaču, dobija se prikaz kao na slici 72.



Slika 72: Prikaz liste svih radnika sa linkovanim imenima.

Nakon što se klikne na *ime* određenog radnika, na novom ekranu će biti prikazani njegovi podaci.

8.6. Dodavanje novog zapisa u bazu podataka korišćenjem formi

Kako bi aplikacija bila potpuno funkcionalna za krajnjeg korisnika, potrebno je omogućiti unos podataka o radniku u okviru *html* forme i čuvanje unetih podataka u bazi podataka, a bez korišćenje specijalizovanih alata kao što je *TablePlus*.

Da bi to bilo moguće počće se, kao i do sada, pravljenjem nove putanje u dokumentu *web.php*:

```
Route::get('/worker/create',  
'App\Http\Controllers\WorkerController@create');
```

Napomena: Obratite pažnju na korišćenje jednine u nazivu putanje – *worker/create* – jer se pravi samo jedan novi radnik jednim pozivanjem navedene putanje.

Nakon što korisnik u pretraživaču unese adresu *localhost/worker/create*, biće pozvan kontroler *WorkerController* i njegova funkcija *create()* – ova funkcija je veoma jednostavna i ima jedini zadatak da prikaže pogled koji će sadržati formular koji se popunjava podacima o radniku:

```
public function create()  
{  
    return view('workers.create');  
}
```

Sada je u direktorijumu *resources/views/workers* potrebno napraviti novi pogled *create.blade.php* sa sledećim sadržajem:

```
@extends('struktura')  
  
@section('naslov', 'Unos novog radnika')  
  
@section('meni')  
    @parent  
@endsection  
  
@section('glavni')  
  
    <!-- Unos podataka o radniku -->  
    <h1>Dodavanje novog radnika</h1>  
  
    <form method="POST" action="/worker">  
  
        {{ csrf_field() }}  
  
    </form>  
  
@endsection
```

```
<label for="ime">Ime radnika: </label>
<br>
<input type="text" id="ime" name="ime">

<br><br>

<label for="plata">Plata: </label>
<br>
<input type="text" id="plata" name="plata">

<br><br>

<label for="Starost">Starost: </label>
<br>
<input type="text" id="starost" name="starost">

<br><br>

<button type="submit">Dodaj radnika</button>

</form>
```

@endsection

Do sada je za sve putanje korišćen *get* metod (*Route::get*) jer su do sada podaci samo učitavani i prikazivani, ali ne i unošeni u aplikaciji.

Prvi red forme u pogledu *create.blade.php* izgleda kao u nastavku:

```
<form method="POST" action="/worker">
```

To znači da će, nakon što korisnik klikne na *submit* dugme, biti pozvana putanja */worker* korišćenjem *post* metoda, čime se aplikaciji saopštava da se šalju podaci koje treba sačuvati. U *web.php* treba dodati sledeću putanju:

```
Route::post('/worker',
'App\Http\Controllers\WorkerController@store');
```

Nakon što se iz forme pozove putanja */worker* korišćenjem *post* metoda, biće pozvana funkcija *store()* u *WorkerControlleru* – sadržaj ove funkcije naveden je u nastavku:

```
public function store()
{
    $worker = new \App\Models\Worker;
    $worker->ime = request('ime');
    $worker->plata = request('plata');
    $worker->starost = request('starost');
    $worker->save();

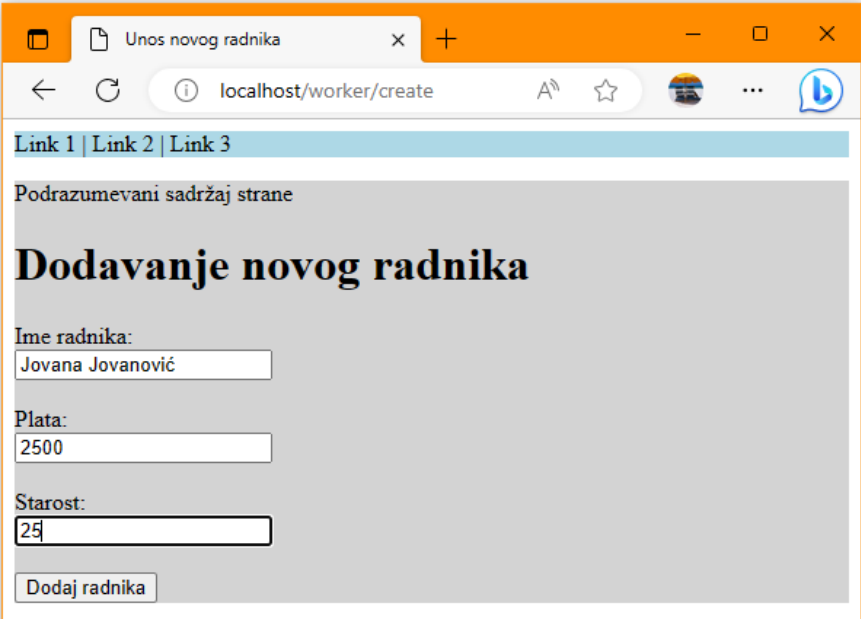
    return redirect('/');
}
```

Funkcija *store()* je zanimljiva jer pravi novi objekat i čuva ga u bazi podataka, bez potrebe za manuelnim pravljenjem *SQL* upita i njihovog izvršavanja:

- u promenljivoj *\$worker* se čuva objekat tipa *Worker*;
- u polja objekta *\$worker* (kao što su *\$worker->ime*, *\$worker->plata*, *\$worker->starost*) se upisuju vrednosti koje je korisnik uneo u formu;
- vrednostima koje je korisnik uneo u formu se pristupa pomoću funkcije *request()* kojoj se kao parametar prosleđuje *id* definisan u *input* poljima u pogledu *create.blade.php*;
- da bi unete vrednosti bile sačuvane u bazi podataka, potrebno je pozvati funkciju *save()* – ona će u pozadini obaviti *SQL* upit;
- naredbom *return redirect('/')* se korisnik automatski preusmerava nazad na početnu stranu.

Napomena: U formi koja se nalazi u pogledu *create.blade.php* se nalazi oznaka `{{ csrf_field() }}`. Ova oznaka predstavlja sigurnosni mehanizam platforme *Laravel* za specifične vrste napada poznate pod imenom *csrf* (*cross-site request forgery*). Dodajte je u svakoj formi koja koristi *post* metod.¹³

Slika 73 prikazuje kako izgleda pogled *create.blade.php* kada se u pretraživaču unese adresa *localhost/worker/create*.



Unos novog radnika

localhost/worker/create

Link 1 | Link 2 | Link 3

Podrazumevani sadržaj strane

Dodavanje novog radnika

Ime radnika:

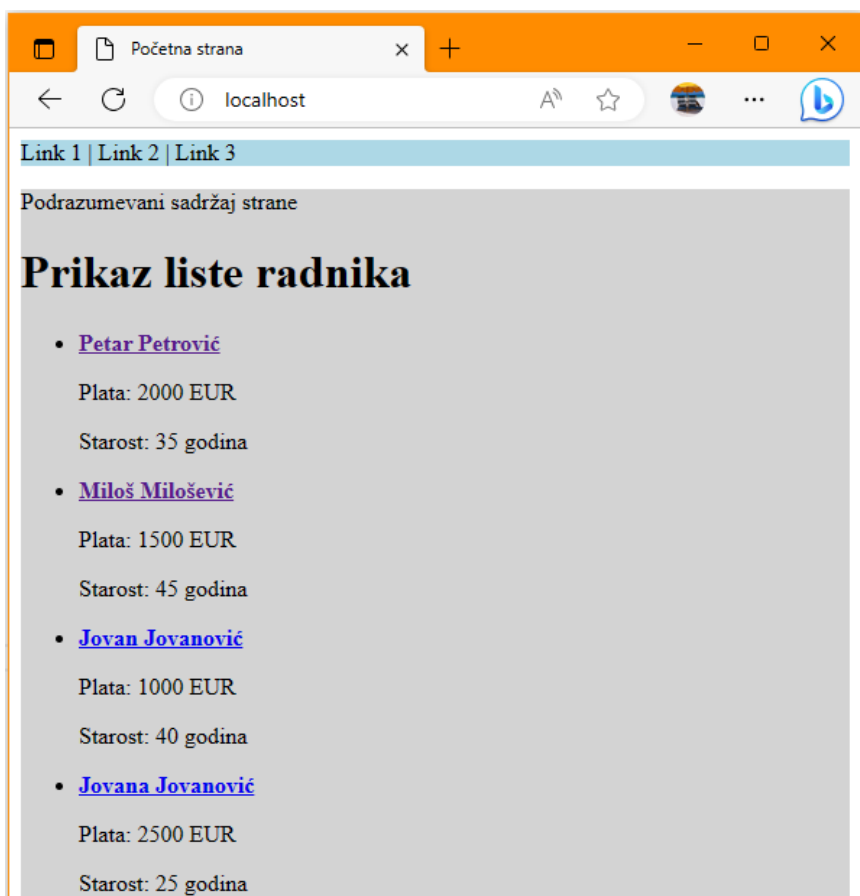
Plata:

Starost:

Slika 73: Forma za dodavanje novog radnika.

Nakon popunjavanja polja i klika na dugme *Dodaj radnika*, biće izvršena redirekcija na početnu stranu na kojoj će ponovo biti učitana lista svih radnika, uključujući i onog koji je upravo dodat (slika 74).

¹³ Više informacija o *csrf* zaštiti dostupno je na adresi laravel.com/docs/10.x/csrf



Slika 74: Prikaz početne strane nakon dodavanja novog radnika.

8.7. Dodavanje linkova u meni

Da bi se dodatno olakšalo korišćenje aplikacije, potrebno je unaprediti meni koji se nalazi na vrhu svake strane. U šablonu *struktura.blade.php* definisana je sekcija *meni*, a njen trenutni sadržaj je prikazan u nastavku:

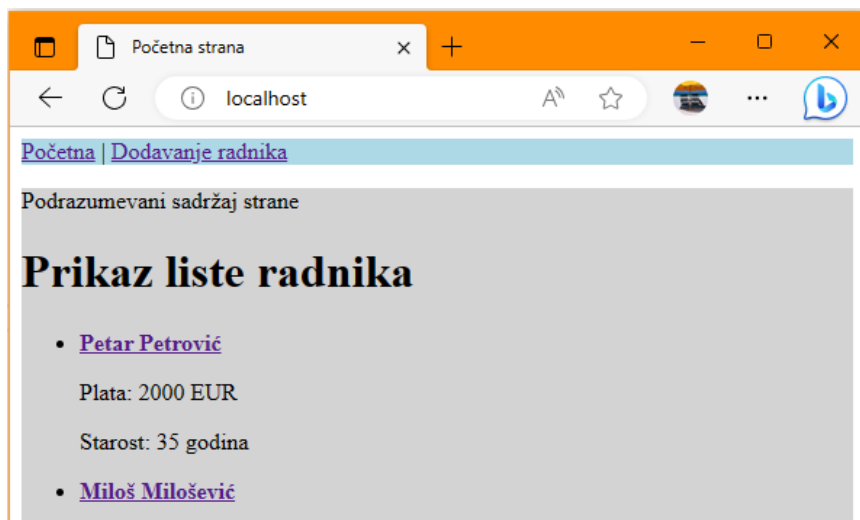
```
@section('meni')
    <div style='background-color: lightblue;'>
        Link 1 | Link 2 | Link 3
```

```
</div>  
@show
```

Obzirom da ovakvi linkovi nisu korisni, potrebno ih je zameniti linkovima koji imaju funkciju:

```
@section('meni')  
  <div style='background-color: lightblue;'>  
    <a href="/">Početna</a>  
    |  
    <a href="/worker/create">Dodavanje radnika</a>  
  </div>  
@show
```

Nakon ove izmene, početna strana izgleda kao na slici 75.



Slika 75: Izgled početne strane nakon uređivanja menija.

8.8. Uklanjanje radnika iz baze podataka

Do sada su napravljene funkcionalnosti dodavanja radnika, prikaza svih radnika i prikaza podataka o pojedinačnom radniku. U ovoj sekciji, biće objašnjeno uklanjanje izabranog radnika iz sistema.

Najpre je potrebno napraviti novu putanju u *web.php* dokumentu:

```
Route::get('/worker/remove/{id}',  
'App\Http\Controllers\WorkerController@remove');
```

Putanja je napravljena prema *REST* pravilima imenovanja, tako da je odmah jasno koja je njena funkcija – za radnika se poziva funkcija *remove* koja će ukloniti samo radnika čiji je *id* definisan. Posao uklanjanja obavlja kontroler zadužen za radnike (dakle, *WorkerController*) i njegova funkcija *remove()* čiji je sadržaj prikazan u nastavku:

```
public function remove($id)  
{  
    \App\Models\Worker::find($id)->delete();  
  
    return redirect('/');  
}
```

Funkcija *remove()* dobija parametar *\$id* (to je vrednost koja se prosleđuje u *REST* adresi), a potom koristi funkciju *find()* da pronađe radnika sa navedenim *id*-em i funkciju *delete()* da ga obriše iz baze podataka. Ponovo, *Laravel* sakriva kompletnu realizaciju komunikacije sa bazom podataka i omogućava da se sve obavi pozivanjem jednostavnih ugrađenih funkcija.

Poslednja naredba je *return redirect('/')* koja korisnika vraća na početnu stranu nakon što obavi operaciju brisanja.

Finalni korak koji je potrebno uraditi da bi sve proradilo je dodavanje dugmeta za brisanje pored svakog korisnika. Na strani

index.blade.php dodajemo sledeći link nakon prikaza imena korisnika:

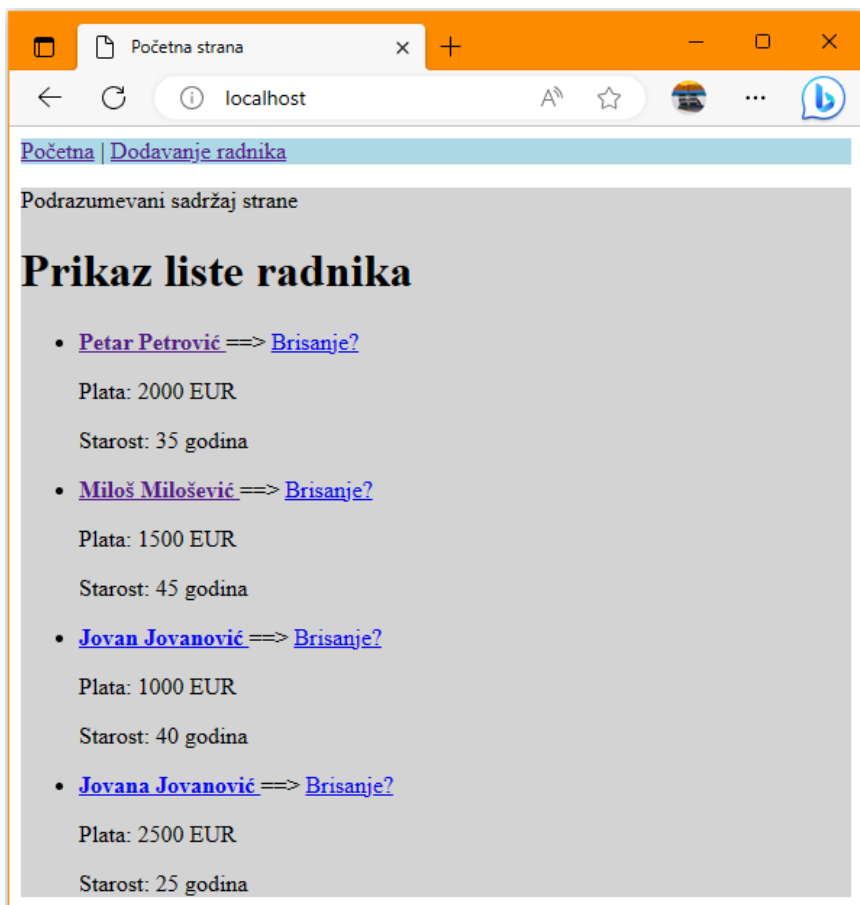
```
<strong>
    <a href="/workers/{{ $worker->id }}">
        {{ $worker->ime }}
    </a>
</strong>

<!-- Link za brisanje korisnika -->

==>

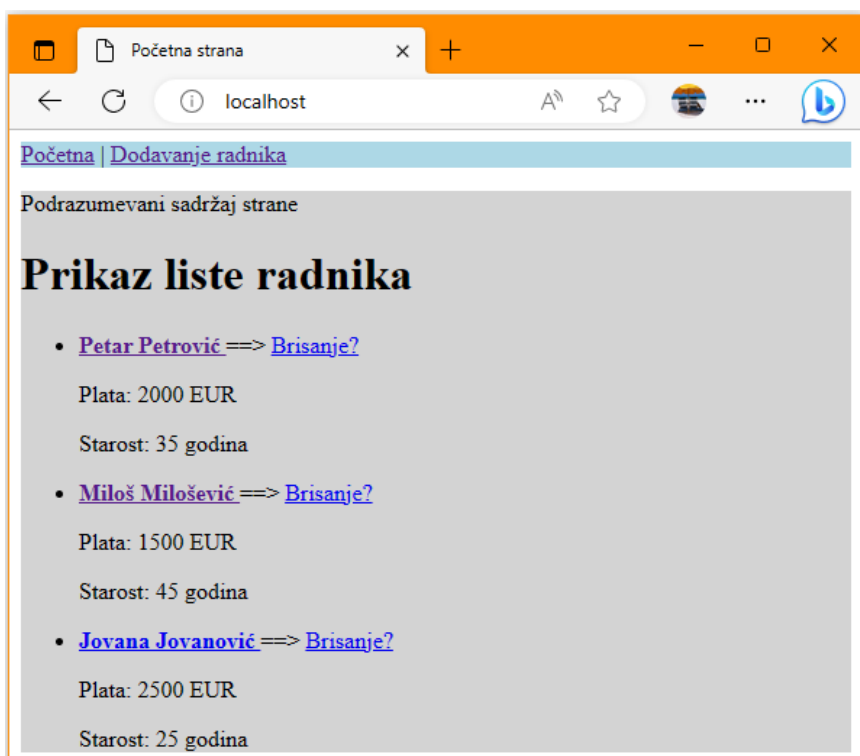
<a href="/worker/remove/{{ $worker->id }}">
    Brisanje?
</a>
```

Nakon dodavanja linka za brisanje, strana će izgledati kao na slici 76.



Slika 76: Početna strana sa dodatom opcijom za brisanje korisnika.

Ukoliko se klikne na link *Brisanje?* pored imena korisnika, iz baze podataka će biti uklonjen samo taj korisnik. Slika 77 prikazuje izgled početne strane nakon što je obrisani korisnik *Jovan Jovanović*.



Slika 77: Izgled početne strane nakon uklanjanja jednog od korisnika.

8.9. Definisane validacije i prikaz grešaka

U aplikacijama koje dozvoljavaju korisnički unos kroz forme, trebalo bi ugraditi mehanizme za validaciju korisničkog unosa i definisati način pomoću kog se na strani mogu prikazati greške koje se dogode u toku rada (kao što su one situacije kada korisnik nije popunio sva polja).

Laravel i u ovom slučaju nudi rešenja problema na veoma elegantan način. Da bi se prikazalo kako validacije funkcionišu, biće dodata jednostavna provera korisničkog unosa prilikom dodavanja

novog radnika. Za dodavanje radnika, zadužena je funkcija *store()* u kontroleru *WorkerController*. Na početku ove funkcije, potrebno je dodati sledeću proveru:

```
public function store()
{
    $this->validate(request(), [
        'ime' => 'required|min:2',
        'plata' => 'required',
        'starost' => 'required'
    ]);

    $worker = new \App\Models\Worker;
    ...
}
```

Prilikom poziva funkcije *validate()*, prosleđuju se dva parametra:

- *request()* predstavlja zahtev za čuvanjem novog radnika u kom se nalaze podaci koje treba sačuvati u bazi podataka (*ime*, *plata*, *starost*);
- niz koji sadrži kriterijume za proveru podataka koje treba upisati – sva tri polja se moraju obavezno popuniti, a ime mora biti najmanje dužine od dva znaka.

Tek ukoliko su navedene provere uspešno izvršene, moći će da se pristupi dodavanju radnika u bazu podataka. Ukoliko podaci nisu prošli provere, korisnik se vraća na stranu na kojoj je unosio podatke u formu, a sve greške će automatski biti sačuvane u objektu koji se zove *\$errors*.

Poslednji korak je da se na strani sa formom za unos proveru da li je nešto sačuvano u objektu *\$errors* i ako jeste, da se ispiše na ekranu. Kako se ne bi na svakoj strani proveravalo da li su se dogodile neke greške, najbolje je napraviti šablonsku stranu koja će se baviti tim poslom. Tu stranu je potrebno sačuvati u *resources/views* pod imenom *greske.blade.php*, a njen sadržaj je prikazan u nastavku:

```
@if (count($errors))
```

```
<div style="color:red;">
  <ul>
    @foreach ($errors->all() as $error)
      <li>{{ $error }}</li>
    @endforeach
  </ul>
</div>
```

```
@endif
```

Naredba *@if* proverava da li u promenljivoj *@errors* postoje upisane greške – ukoliko postoje, petlja *@foreach* će crvenom bojom ispisati svaku od njih u okviru liste.

Na kraju, samo je još potrebno u glavnom šablonu, *struktura.blade.php*, dodati poziv dokumenta *greske.blade.php* – to se može uraditi u sredini dokumenta, između sekcije *meni* i sekcije u kojoj je glavni sadržaj strane:

```
...
```

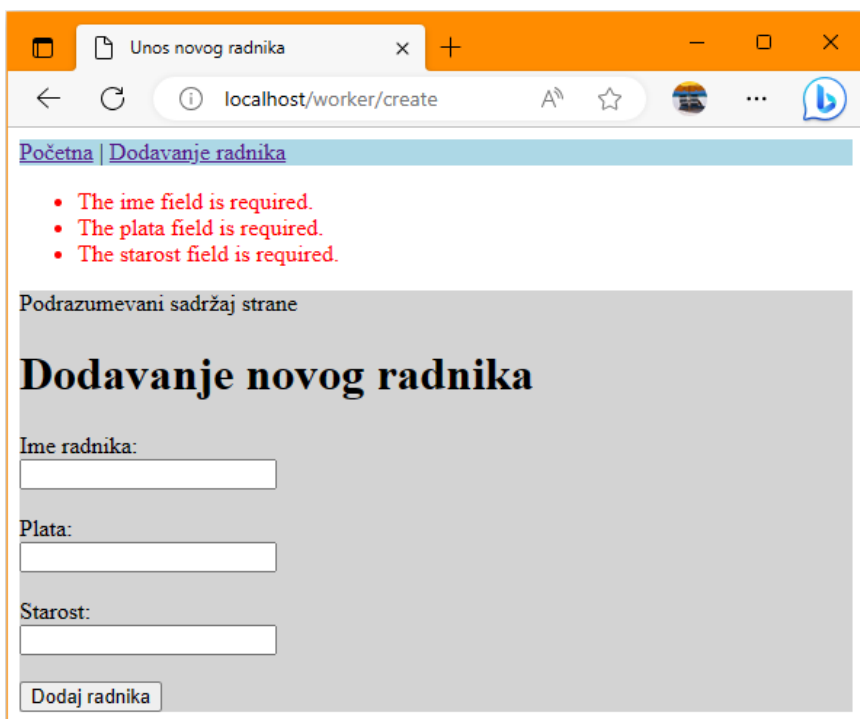
```
@show
```

```
    @include('greske')
```

```
<div style='background-color: lightgray;'>
```

```
...
```

Nakon toga, ukoliko neko polje ne bude ispravno popunjeno prilikom unosa novog radnika, na ekranu će biti prikazana greška (slika 78).



Slika 78: Prikaz grešaka nakon validacije korisničkog unosa.

8.10. Izmena podataka u bazi podataka

U prethodnim sekcijama je prikazano kako se ispisuju podaci o svim radnicima, kako se prikazuju podaci o pojedinačnom radniku, kao i kako se vrše operacije dodavanja i brisanja radnika. Poslednja CRUD (Create, Read, Update, Delete) operacija koju je potrebno obraditi je ažuriranje podataka.

Funkcionalnost ažuriranja podataka o radniku se sastoji iz dva dela:

- prikaz trenutnih podataka o izabranom radniku i
- izmena i čuvanje navedenih podataka u bazi podataka.

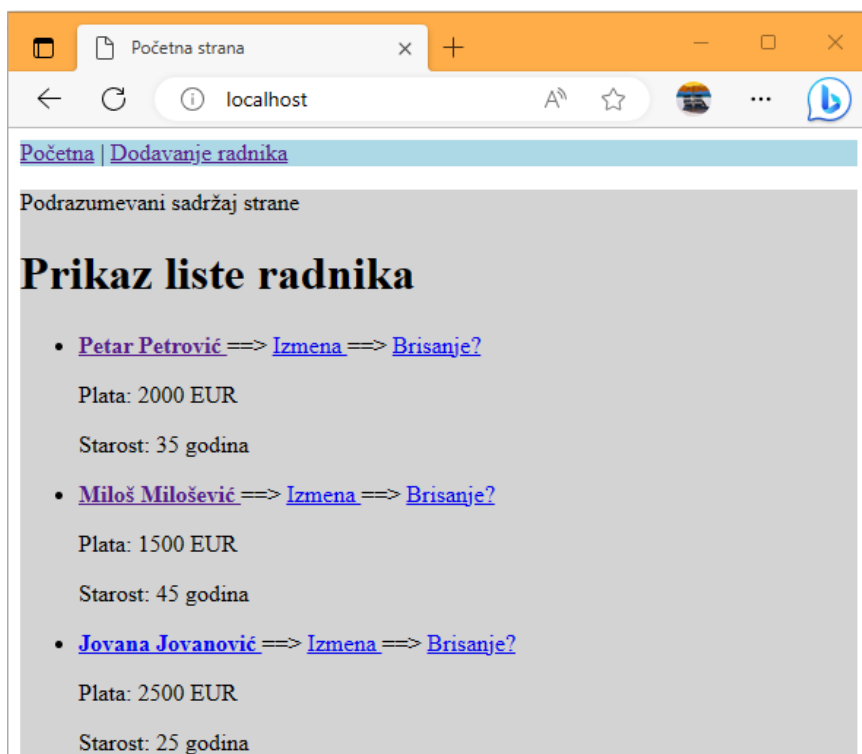
Prvi korak je pravljenje putanje u dokumentu *web.php* – ova putanja će otvoriti stranu na kojoj će se prikazati trenutni podaci izabranog korisnika:

```
Route::get('/worker/updateform/{id}',  
'App\Http\Controllers\WorkerController@updateform');
```

Zatim se na početnoj strani *index.blade.php* pored imena svakog korisnika dodaje link koji će pozivati novu putanju – u nastavku je prikazan deo strane u okviru koga je dodat novi link:

```
<strong>  
    <a href="/workers/{{ $worker->id }}">  
        {{ $worker->ime }}  
    </a>  
</strong>  
  
==>  
  
<-- Novi link za izmenu podataka o korisniku -->  
  
<a href="/worker/updateform/{{ $worker->id }}">  
    Izmena  
</a>  
  
==>  
  
<a href="/worker/remove/{{ $worker->id }}">  
    Brisanje?  
</a>
```

Slika 79 prikazuje novi izgled početne strane.



Slika 79: Početna strana sa dodatom opcijom za izmene.

Sledeći korak je pravljenje funkcije `updateform()` u kontroleru `WorkerController` – ova funkcija je zadužena da otvori novi pogled nakon što se klikne na dugme *Izmjena*:

```
public function updateform($id)
{
    $worker = \App\Models\Worker::find($id);

    return view('workers.updateform', compact('worker'));
}
```

Funkcija `updateform()` prihvata *id* korisnika, a zatim po tom *id*-u iz baze podataka učitava podatke o radniku koristeći ugrađenu

Laravel funkciju *find()* i čuva ih u promenljivoj *\$worker*. Nakon toga usmerava korisnika na novi pogled koji se zove *updateform* i nalazi se u folderu *workers*, a potom mu prosleđuje podatke sačuvane u promenljivoj *\$worker* – novi pogled mora učitati ove podatke i omogućiti korisniku da ih menja.

Naredni korak je pravljenje pogleda na putanji *resources/views/workers* pod imenom *updateform.blade.php*, a koji treba da sadrži sledeće naredbe:

```
@extends('struktura')

@section('naslov', 'Izmena podataka o radniku')

@section('meni')
    @parent
@endsection

@section('glavni')

    <!-- Izmena podataka pojedinačnog radnika -->

    <h1>Izmena podataka - {{ $worker->ime }}</h1>

    <form method="POST" action="/worker/update/{{ $worker->id }}">

        {{ csrf_field() }}

        <label for="ime">Ime: </label>
        <br>
        <input type="text" id="ime" name="ime"
            value="{{ $worker->ime }}">

        <br><br>

        <label for="plata">Plata: </label>
        <br>
        <input type="text" id="plata" name="plata"
            value="{{ $worker->plata }}">
```

```
<br><br>

<label for="starost">Starost: </label>
<br>
<input type="text" id="starost" name="starost"
      value="{{ $worker->starost }}">

<br><br>

<button type="submit">Izmena podataka o
radniku</button>

</form>

@endsection
```

Najvažniji delovi funkcionalnosti su:

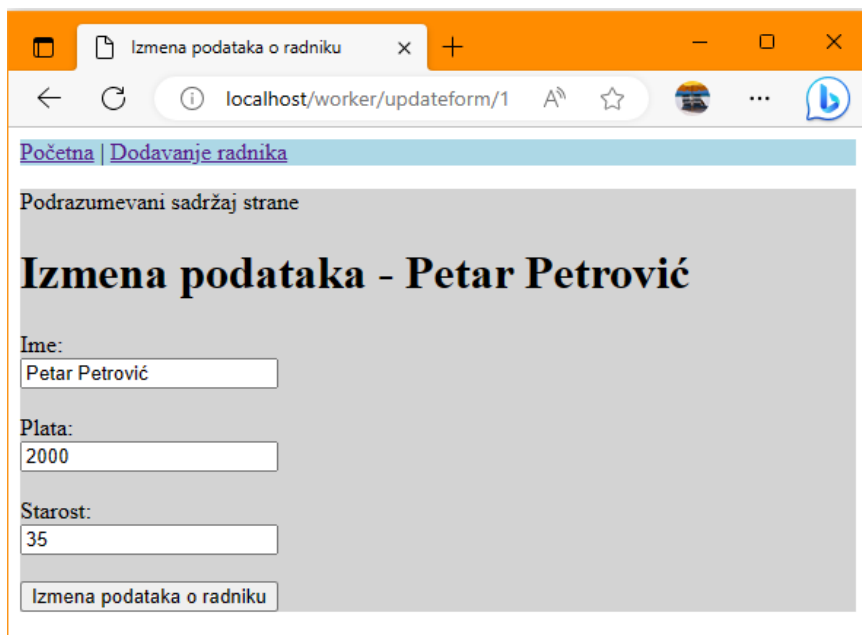
- pristupanje pojedinačnim podacima o radniku se vrši pomoću naredbi `{{ $worker->ime }}`, a reč *ime* može biti zamenjena i drugim vrednostima;
- za slanje podataka nakon izmene se koristi *post* metod i nova putanja koju će biti potrebno definisati:

```
<form method="POST"
      action="/worker/update/{{ $worker->id }}">
```

- vrednosti se upisuju u tekstualna polja korišćenjem svojstva *value*:

```
value="{{ $worker->plata }}"
```

Nakon što se na početnoj strani klikne na link *Izmeni* pored imena nekog radnika, otvoriće se strana prikazana na slici 80.



Slika 80: Prikaz strane za izmenu podataka o radniku.

Potrebno je napraviti i novu putanju u dokumentu *web.php* – ona će koristiti metod *post*, kao što je i definisano u *form* elementu:

```
Route::post('/worker/update/{id}',
'App\Http\Controllers\WorkerController@update');
```

Ova putanja će, nakon što se klikne na dugme *Izmena podataka o radniku* pozvati funkciju *update()* u kontroleru *WorkerController*. Ova funkcija će prihvatiti podatke koje je korisnik modifikovao u formi, izvršiti njihovu validaciju i, ukoliko je sve u redu, sačuvati ih bazi podataka. Sadržaj funkcije *update()* sledi:

```
public function update($id) {

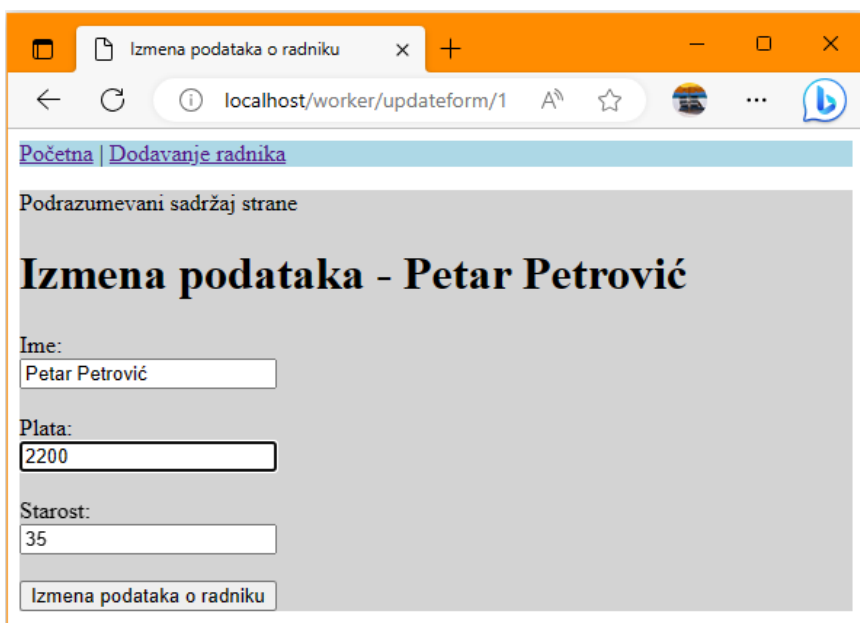
    $this->validate(request(), [

        'ime' => 'required|min:2',
```

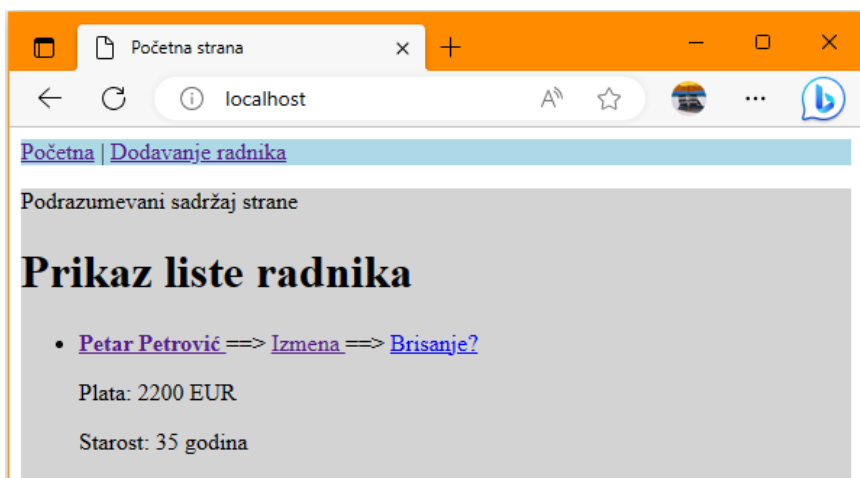
```
'plata' => 'required',  
'starost' => 'required'  
  
]);  
  
$worker = \App\Models\Worker::find($id);  
$worker->ime = request('ime');  
$worker->plata = request('plata');  
$worker->starost = request('starost');  
$worker->save();  
  
return redirect('/');  
}
```

Sadržaj funkcije *update()* je vrlo sličan funkciji *store()* koja je napravljena ranije – glavna razlika je u tome što funkcija *store()* čuva podatke potpuno novog korisnika, a funkcije *update()* nalazi postojećeg korisnika i menja njegove podatke.

Sada je moguće testirati funkcionalnost izmena podataka o radniku – možemo korisniku *Petar Petrović* dati povišicu (slika 81) – nakon klika na dugme *Izmena podataka o radniku*, izvršiće se redirekcija na početnu stranu gde bi trebalo da se prikaže nova vrednost za platu (slika 82).



Slika 81: Izmena podataka o radniku.



Slika 82: Prikaz izmenjenih podataka o radniku na početnoj strani.

Dodatak A

U nastavku je, u cilju bolje preglednosti, u celini dat prikaz nekoliko najvažnijih segmenata *Laravel* aplikacije koja je pravijena u posljednjem poglavlju.

Dokument web.php

Sve putanje koje se odnose na funkcionalnosti vezane za radnike su navedene u nastavku:

```
Route::get('/',  
'App\Http\Controllers\WorkerController@index');  
  
Route::get('/workers/{id}',  
'App\Http\Controllers\WorkerController@show');  
  
Route::get('/worker/create',  
'App\Http\Controllers\WorkerController@create');  
  
Route::post('/worker',  
'App\Http\Controllers\WorkerController@store');  
  
Route::get('/worker/remove/{id}',  
'App\Http\Controllers\WorkerController@remove');  
  
Route::get('/worker/updateform/{id}',  
'App\Http\Controllers\WorkerController@updateform');  
  
Route::post('/worker/update/{id}',  
'App\Http\Controllers\WorkerController@update');
```

Kontroler WorkerController

Klasa *WorkerController* sadrži sve funkcionalnosti koje se tiču entiteta *Worker*.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class WorkerController extends Controller
{
    public function index()
    {
        $workers = \App\Models\Worker::all();

        return view('workers.index', compact('workers'));
    }

    public function show($id)
    {
        $worker = \App\Models\Worker::find($id);
        return view('workers.show', compact('worker'));
    }

    public function create()
    {
        return view('workers.create');
    }

    public function store()
    {
        $this->validate(request(), [

            'ime' => 'required|min:2',
            'plata' => 'required',
            'starost' => 'required'
```

```
]);

$worker = new \App\Models\Worker;
$worker->ime = request('ime');
$worker->plata = request('plata');
$worker->starost = request('starost');
$worker->save();

return redirect('/');
}

public function remove($id)
{
    \App\Models\Worker::find($id)->delete();

    return redirect('/');
}

public function updateform($id)
{
    $worker = \App\Models\Worker::find($id);
    return view('workers.updateform', compact('worker'));
}

public function update($id) {

    $this->validate(request(), [

        'ime' => 'required|min:2',
        'plata' => 'required',
        'starost' => 'required'

    ]);

    $worker = \App\Models\Worker::find($id);
    $worker->ime = request('ime');
    $worker->plata = request('plata');
    $worker->starost = request('starost');
    $worker->save();

    return redirect('/');
```

```
}  
  
}
```

Pogled struktura.blade.php

Dokument *struktura.blade.php* sadrži glavni šablon koji se koristi za sve poglede u aplikaciji.

```
<html>  
  
  <head>  
    <title>@yield('naslov')</title>  
  </head>  
  
  <body>  
    @section('meni')  
      <div style='background-color: lightblue;'>  
        <a href="/">Početna</a>  
        |  
        <a href="/worker/create">Dodavanje radnika</a>  
      </div>  
    @show  
  
    @include('greske')  
  
    <div style='background-color: lightgray;'>  
      <p>Podrazumevani sadržaj strane</p>  
      @yield('glavni')  
    </div>  
  </body>  
</html>
```

Pogled index.blade.php

Strana *index.blade.php* je osnovni pogled pomoću kog se pristupa ostalim delovima aplikacije.

```
@extends('struktura')

@section('naslov', 'Početna strana')

@section('meni')
    @parent
@endsection

@section('glavni')

    <!-- Mesto gde će se prikazati lista radnika -->
    <h1>Prikaz liste radnika</h1>
    <ul>
        @foreach ($workers as $worker)
            <li>

                <strong>

                    <a href="/workers/{{ $worker->id }}">
                        {{ $worker->ime }}
                    </a>

                </strong>

                ==>

                <a href="/worker/updateform/{{ $worker->id }}">
                    Izmena
                </a>

                ==>

                <a href="/worker/remove/{{ $worker->id }}">
                    Brisanje?
                </a>

                <br>

                <p>Plata: {{ $worker->plata }} EUR</p>

                <p>Starost: {{ $worker->starost }} godina</p>

            </li>
        @endforeach
    </ul>

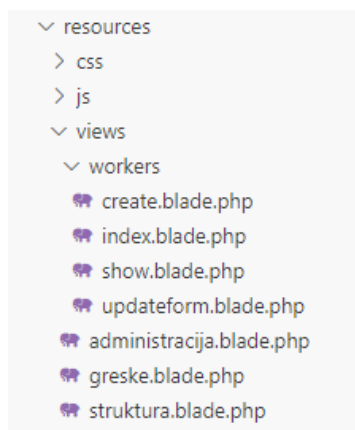
endsection
```

```
</li>  
@endforeach  
</ul>
```

```
@endsection
```

Struktura direktorijuma `resources/views`

Direktorijum *resources/views* je glavno mesto za čuvanje svih pogleda u aplikaciji. Njegova struktura prikazana je slici 83.



Slika 83: Struktura direktorijuma *resources/views*.

Popis slika

Slika 1: Rezultat ispisa sadržaja niza u veb pretraživaču.

Slika 2: Prikaz tri nasumično odabrane slike.

Slika 3: Tabela sa ulaznim podacima.

Slika 4: Prikaz sadržaja .json dokumenta.

Slika 5: Izgled tabele sa pitanjima.

Slika 6: REST API.

Slika 7: Struktura REST URL-a.

Slika 8: Model klijent-server.

Slika 9: Struktura REST aplikacije.

Slika 10: Pozivanje foldera u kom nema index strane.

Slika 11: Struktura baze podataka.

Slika 12: Pokretanje REST zahteva za prikaz svih korisnika.

Slika 13: Početni ekran u Postman-u.

Slika 14: Izbor metoda i unos REST URL-a u Postman-u.

Slika 15: Prikaz liste svih radnika.

Slika 16: Prikaz podataka o radniku čiji id ima vrednost 2.

Slika 17: Struktura html Document Object Model-a.

Slika 18: Prikaz izgleda pokrenute aplikacije.

Slika 19: Izgled aplikacije Docker Desktop.

Slika 20: Provera verzije WSL-a.

Slika 21: Uključivanje WSL-a u aplikaciji Docker Desktop.

Slika 22: Uključivanje Linux distribucija u aplikaciji Docker Desktop.

Slika 23: Početni ekran aplikacije Visual Studio Code.

Slika 24: Instalacija dodatka WSL za Visual Studio Code.

Slika 25: Dodavanje Linux distribucije u Visual Studio Code.

Slika 26: Izbor Linux distribucije.

Slika 27: Uspešno dodata Linux distribucija.

Slika 28: Izgled terminala u aplikaciji Visual Studio Code.

Slika 29: Rad sa više terminala.

Slika 30: Promena trenutno aktivnog direktorijuma.

Slika 31: Završena instalacija novog Laravel projekta.

Slika 32: Pokretanje nove WSL konekcije.

Slika 33: Sadržaj direktorijuma aplikacije evidencija-app.

Slika 34: Otvaranje direktorijuma aplikacije.

Slika 35: Podrazumevana početna strana Laravel aplikacije.

Slika 36: Izgled terminala dok je aplikacija pokrenuta.

Slika 37: Izgled pokrenutog Docker kontejnera aplikacije evidencija-app.

Slika 38: Visual Studio Code Explorer.

Slika 39: Unos adrese u Explorer-u.

Slika 40: Struktura aplikacije evidencija-app.

Slika 41: MVC arhitektura.

Slika 42: Vraćanje String vrednosti prilikom poziva putanje.

Slika 43: Vraćanje JSON vrednosti prilikom poziva putanje.

Slika 44: Korišćenje Blade naredbe za prikaz vrednosti promenljive.

Slika 45: Prikaz strane kada je izvršen then uslov Blade naredbe.

Slika 46: Prikaz strane kada je izvršen else uslov Blade naredbe.

Slika 47: Izgled strane napravljene prema Blade šablonu.

Slika 48: Pravljenje aliasa i pozivanje naredbe php:artisan.

Slika 49: Prikaz liste svih komandi php artisan.

Slika 50: Izvršavanje naredbe sail php artisan help migrate.

Slika 51: Pozivanje komande php artisan migrate:refresh.

Slika 52: Instalacija potrebnih paketa za rad sa bazama podataka.

Slika 53: Rezultat izvršavanja naredbe db:table.

Slika 54: Nova struktura tabele users nakon izmene migracije.

Slika 55: Prikaz liste svih migrate naredbi.

Slika 56: Izvršavanje naredbe migrate:refresh nakon pravljenja nove migracije.

Slika 57: Izvršavanje naredbe make:model.

Slika 58: Izvršavanje naredbe make:controller.

Slika 59: Prva verzija početne strane aplikacije.

Slika 60: Izvršavanje naredbe make:model.

Slika 61: Izvršenje naredbe migrate.

Slika 62: Pregled strukture tabele workers.

Slika 63: Izgled početne strane uz prikaz prazne liste radnika.

Slika 64: Početna strana tableplus.com.

Slika 65: Početni ekran alata TablePlus.

Slika 66: Podešavanje parametara konekcije ka bazi podataka.

Slika 67: Prikaz baze podataka nakon uspostavljanja konekcije.

Slika 68: Prihvatanje unetih izmena u tabeli workers.

Slika 69: Sadržaj tabele workers nakon završenog unosa za tri radnika.

Slika 70: Prikaz liste svih radnika na početnoj strani aplikacije.

Slika 71: Prikaz podataka o pojedinačnom radniku na osnovu vrednosti id.

Slika 72: Prikaz liste svih radnika sa linkovanim imenima.

Slika 73: Forma za dodavanje novog radnika.

Slika 74: Prikaz početne strane nakon dodavanja novog radnika.

Slika 75: Izgled početne strane nakon uređivanja menija.

Slika 76: Početna strana sa dodatom opcijom za brisanje korisnika.

Slika 77: Izgled početne strane nakon uklanjanja jednog od korisnika.

Slika 78: Prikaz grešaka nakon validacije korisničkog unosa.

Slika 79: Početna strana sa dodatom opcijom za izmene.

Slika 80: Prikaz strane za izmenu podataka o radniku.

Slika 81: Izmena podataka o radniku.

Slika 82: Prikaz izmenjenih podataka o radniku na početnoj strani.

Slika 83: Struktura direktorijuma resources/views.

Literatura

Fielding, R. T, *Architectural Styles and the Design of Network-based Software Architectures*, Dissertation, University of California, Irvine, 2000.

Flanagan D., *JavaScript sveobuhvatan vodič*, prevod sedmog izdanja, Mikro knjiga, Beograd, 2021.

Haverbeke M., *JavaScript elokventno*, Mikro knjiga, Beograd, 2019.

Klepmann M., *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, 1st edition, O'Reilly, USA, 2017.

Newman S., *Izgradnja mikroservisa – dizajn sitno granuliranih sistema*, O'Reilly, prevod CET, Beograd, 2022.

Nixon R., *Naučite PHP, MySQL i JavaScript: razvoj dinamičkih veb strana korak po korak*, prevod šestog izdanja, Mikro knjiga, Beograd, 2021.

Pettit T., *MySQL radionica: Praktičan vodič za rad sa podacima i bazama podataka*, Kompjuter biblioteka, Beograd, 2022.

Richards M., Ford N., *Fundamentals of Software Architecture*, O'Reilly, USA, 2020.

Standard RFC 9110, *HTTP Semantics*, 2022.

Staufer M., *Laravel - Radni okvir za izradu modernih PHP aplikacija*, Kompjuter biblioteka, Beograd, 2019.

Svekis L. L., *JavaScript od početnika do profesionalca*, Kompjuter biblioteka, Beograd, 2022.

VandenBrink R., *Linux za mreže*, Kompjuter biblioteka, Beograd, 2022.

Welling L., Thomson L., *PHP i MySQL: razvoj aplikacija za veb*,
prevod petog izdanja, Mikro knjiga, Beograd, 2017.

Veb linkovi

laravel.com

www.w3schools.com

developer.mozilla.org

learn.microsoft.com

docker.com

code.visualstudio.com

tableplus.com

CIP - Каталогизација у публикацији Народна библиотека Србије,
Београд

004.42:004.738.5(075.8)

МАРКОВИЋ, Марко, 1988-

Razvoj aplikacija zasnovanih na veb servisima / Marko Marković. - 1.
izd. -

Valjevo : Univerzitet Singidunum, Fakultet zdravstvenih i poslovnih
studija, 2023 (Valjevo : Univerzitet Singidunum, Fakultet zdravstvenih i
poslovnih studija). - VII, 204 str. : ilustr. ; 24 cm

Tiraž 500. - Bibliografija: str. 203-204.

ISBN 978-86-81206-04-1

a) Веб апликације -- Програмирање

COBISS.SR-ID 120654089