

OSNOVE PHP

Tomislav Keščec - predavač

SADRŽAJ

Ovdje je obuhvaćen dio o osnovama PHP-a, uvod u baze podataka, SQL jezik, napredni MySQL, napredno PHP programiranje, spajanje PHP-a s SQL bazama i testiranje. To je ono što se naziva Vanilla PHP.

[Course title]

Dodavanje ekstenzija Visual Studio Code koje su vezane za PHP	13
PHP Intelephase	13
PHP Debug.....	13
Što je PHP?	16
PHP sintaksa	16
PHP izrazi.....	16
PHP iskazi.....	17
PHP direktive include i require	17
PHP interpreter - princip rada	19
PHP varijable i tipovi podataka	22
<code>echo</code> i <code>print</code> za ispis	22
Prebacivanje u novi red sa <code>/n</code> ili <code>PHP_EOL</code>	22
Jednostruki i dvostruki navodnici.....	23
Jednostruki navodnici ("").	23
Dvostruki navodnici ("")	24
PHP varijable	25
PHP varijable - primjer	25
PHP konstante	25
PHP konstante - primjer	26
PHP tipovi podataka	26
Cijeli brojevi (integer).....	26
Realni brojevi (float).....	27
Tekstualni podaci (string).....	27
Logičke vrijednosti (boolean)	28
PHP reference.....	28
Vježba - PHP varijable i konstante.....	29
PHP operatori.....	30
Operatori dodjele	31
Aritmetički operatori.....	31
Operator nad stringom (znakovnim nizom)	32
Kombinirani operator dodjele.....	32
Operatori usporedbe.....	32
Logički operatori.....	33
Operatori inkrementa i dekrementa	33
Pretvaranje tipova (engl. type juggling) u PHP-u	34
Vježba operatori.....	35

PHP matrice	36
Jednodimenzionate matrice	37
Inicijalizacija matrica	37
Dodavanje (populacija) elemenata	38
Višedimenzionalne matrice	38
Inicijalizacija matrica	38
Dodavanje (populacija) elemenata	39
Uklanjanje elemenata	40
Pristup elementima	40
empty i isset funkcije	41
Traženje elemenata	41
Sortiranje	42
Ostale funkcije	43
Vježba 02	44
Vježba 03	46
PHP kontrolne strukture	54
Uvjetovane kontrolne strukture	54
if kontrolna struktura	54
if-else kontrolna struktura	55
if-elseif kontrolna struktura	55
switch-case kontrolna struktura	57
Vježba 1	58
Vježba 2	59
Kontrolne strukture - petlje	60
while petlje	60
do-while petlje	60
for petlje	62
foreach petlje	63
Vježba 1	65
Vježba 2	65
Algoritmi za sortiranje	65
PHP funkcije	71
Deklariranje i pozivanje	71
Vježba 1	72
PHP funkcije – argumenti	72

Vježba 2	73
Strogo tipiziranje	74
Zadatak.....	75
Doseg (engl. scope) varijabli.....	75
Superglobalna varijabla \$GLOBALS.....	77
Varijabla static.....	79
PHP funkcije – kao varijable	79
Anonimne funkcije (Closures)	80
Vježba 3	80
Callback funkcije.....	80
Superglobalne varijable.....	82
\$GLOBALS superglobalna.....	83
\$_SERVER superglobalna.....	84
\$_GET superglobalna	86
\$_POST superglobalna	86
\$_FILES superglobalna	86
\$_REQUEST superglobalna	87
Zadatak.....	87
Rekurzija.....	89
Statičke varijable	90
Generatori	91
Generatori s ključevima.....	92
Pisanje i čitanje tekst datoteke	93
Sanitizacija.....	94
Sanitizacija za HTML kontekst	94
Sa htmlspecialchars()	94
Sa htmlentities()	94
Sanitizacija za URL-ove	95
Sanitizacija za e-mail adrese.....	95
Sanitizacija za integer vrijednosti.....	95
Sanitizacija za stringove	95
Sanitizacija za SQL kontekst	95
Funkcije za pristup datotekama - CRUD.....	96
Zahtjev za Izradu Aplikacije: Upravitelj Knjiga.....	96
Proširena aplikacija: Upravitelj Knjiga.....	107

1. dio - pristupanje aplikaciji i pregled JSON datoteke u kojoj su podaci pohranjeni	107
2. dio - slanje forme POST metodom	117
3. dio - cookie i session.....	127
Zadatak.....	132
Parcijalni ispit	135
Uvod u baze podataka.....	140
Vrste baza podataka.....	141
Distribuirane baze podataka	141
Objektno orijentirane baze podataka	141
Centralizirana baza podataka.....	141
Baze podataka otvorenog koda.....	141
Baze podataka u oblaku	141
NoSQL baze podataka	141
Relacijski model baze podataka	141
Uvod u MySQL.....	142
Osnovi pojmovi o projektiranju baza podataka	142
ER dijagram	143
Upoznavanje sa ER modelima	143
Zašto koristiti ER dijagrame u DBMS-u?.....	143
Simboli korišteni u ER modelu	144
Komponente ER dijagrama.....	145
Tip odnosa i skup odnosa.....	148
Stupanj skupa odnosa	149
Ograničenje sudjelovanja.....	153
Kako nacrtati ER dijagram?	154
ER dijagram (Vježba 1)	155
MariaDB I MySQL: usporedba	158
MariaDB.....	158
MySQL	158
Ključne razlike.....	158
Rukovanje JSON podacima.....	158
Funkcionalnost	159
Autentifikacija korisnika.....	159
Enkripcija	159
Udruživanje niti	159

Ostale razlike	159
Programi za konekciju s bazama podataka	160
phpMyAdmin.....	160
MySQL WorkBench.....	161
DBeaver Community	162
Pokretanje XAMPP s MySQL-om	163
Normalizacija	163
Normalizacija - 1NF	164
Normalizacija - 2NF	164
Normalizacija - 3NF	165
Denormalizacija	165
Vježba	166
SQL jezik	166
SQL sintaknsni dijagram	167
SQL Uobičajeni upiti	169
SHOW DATABASES	169
USE baza_podataka_ime	169
DESCRIBE tablica_ime.....	169
SHOW TABLE.....	169
SHOW CREATE TABLE tablica_ime.....	169
SELECT NOW()	169
SELECT 2 + 4	169
Komentari.....	169
CREATE DATABASE baza_podataka_ime	170
DROP DATABASE baza_podataka_ime	170
CREATE TABLE tablica_ime(stupac1, stupac2, stupac3..)	170
DROP TABLE tablica_ime.....	170
RENAME TABLE stara_tablica_ime T0 nova_tablica_ime.....	170
ALTER TABLE tablica_ime ADD(stupac1, stupac2, stupac 3 ..)....	170
ALTER TABLE tablica_ime DROP(stupac1).....	170
INSERT INTO tablica_ime (stupac1, stupac2, stupac3 . . .) VALUES(vrijednost1, vrijednost2, vrijednost3. . .)	170
UPDATE tablica_ime SET stupac1 = vrijednost1, stupac2 = vrijednost2, stupac3 = vrijednost3.. WHERE uslov	170
DELETE FROM tablica_ime WHERE uslov.....	171

SELECT stupac1, stupac2, stupac3.. FROM tablica_ime WHERE uslov	171
SELECT * FROM tablica_ime.....	171
MAX.....	171
MIN.....	171
LIMIT.....	171
BETWEEN.....	172
DISTINCT.....	172
IN klauzula.....	172
AND.....	172
OR	172
IS NULL.....	173
FOREIGN KEY	173
LIKE	173
ADD i DROP stupci.....	174
SQL tipovi podataka	174
SQL tipovi podataka - numerički	175
SQL tipovi podataka – datum i vrijeme	176
SQL – DDL ključne riječi	176
CREATE	177
DROP	177
TRUNCATE.....	178
SQL – DML ključne riječi	178
INSERT	178
UPDATE	178
DELETE	178
SQL – DCL ključne riječi	179
GRANT.....	179
REVOKE	179
SQL - TCL ključne riječi.....	179
ROLLBACK.....	179
SAVEPOINT	180
SQL - DQL ključne riječi.....	180
SELECT	180
My SQL ograničenja podataka(engl. Data Constraints)	180

Ograničenje NOT NULL.....	180
UNIQUE ograničenje	182
DEFAULT ograničenje	185
Što je ključ (engl. key) u SQL-u	186
SQL spajanja - Cross Join, Full Outer Join, Inner Join, Left Join i Right Join	193
Što je spajanje (engl. join)	193
CROSS JOIN - križno spajanje	195
Kreiranje tablica režisera i filmova za demonstraciju spajanja	199
INNER JOIN - unutrašnje spajanje.....	199
LEFT JOIN - lijevo spajanje	204
RIGHT JOIN - desno spajanje	207
LEFT JOIN / RIGHT JOIN u producijskim aplikacijama	209
FULL OUTER JOIN - potpuno vanjsko spajanje	210
SELF JOIN - samospajanje	211
Višestruki spojevi.....	212
Spajanje s dodatnim uslovima.....	214
MySQL Podupiti.....	214
Podupit s WHERE klauzulom	215
Podupit s operatorima usporedbe	216
Podupit s IN i NOT IN operatorima.....	216
Podupit s klauzulom FROM.....	217
Podupit u korelaciiji.....	217
Podupit s EXISTS i NOT EXISTS.....	217
MySQL WorkBench načini snimanja sheme.....	218
Vježba SQL – DDL ključne riječi	218
Vježba SQL - rad sa SELECT.....	228
Vježba- SELECT sa podupitom	239
Zadaća	246
Uvod u MySQL Procedure	252
DETERMINISTIC ili NONDETERMINISTIC.....	253
Parametri u procedurama.....	254
Varijable u procedurama.....	255
MySQL funkcije.....	256
Razlike između procedura i funkcija.....	258
Napredni MySQL	260

Uvod u MySQL Poglede	260
Kreiranje pogleda	260
Korištenje pogleda	260
Ažuriranje i brisanje pogleda.....	260
Ažuriranje pogleda	260
Brisanje pogleda.....	261
Ograničenja i najbolje prakse	261
Primjer pogleda	261
Uvod u MySQL Transakcije.....	262
Početak transakcije	263
Kontrola transakcija	263
Nivo izolacije	263
Uvod u MySQL Okidače	265
Što su Okidači?	265
Zašto koristiti Okidače?	265
Vrste Okidača	265
Prema vremenu izvršenja:.....	265
Prema vrsti događaja:	266
Kako kreirati okidač.....	266
Okidač prije ažuriranja (BEFORE UPDATE trigger),.....	268
Okidač nakon ažuriranja (AFTER UPDATE trigger)	269
Okidač prije umetanja (BEFORE INSERT trigger)	270
Okidač nakon umetanja (BEFORE INSERT trigger)	271
Okidač prije brisanja (BEFORE DELETE trigger)	273
Okidač nakon brisanja (BEFORE DELETE trigger).....	274
Primjeri okidača predavača	275
Pregled i brisanje okidača	276
Najbolje prakse i napomene	276
Kako ukloniti okidač	276
Kada koristiti okidače	277
Primjer transakcije, okidača i procedure.....	277
Zašto napredno PHP programiranje?.....	290
Pregled Modula	290
Resursi za Učenje.....	290
Uvod u Objektno-Orijentirano Programiranje	290

Osnovne karakteristike.....	291
Klase i objekti	291
Shallow copy i deep copy	292
Kontrola pristupa svojstava i metoda (access modifier).....	297
Ključna riječ \$this	297
Konstruktori i destruktori.....	299
Setteri i Getteri.....	300
Nasljedivanje	302
Polimorfizam	304
Polimorfizam kroz nasljedivanje	304
Interface-i.....	305
Apstraktne klase.....	308
Razlika između korištenja apstraktnih klasa i interface-a.....	313
Statičke Metode	313
self i parent ključne riječi za pristup statickim članovima unutar klase	315
Operator za rješavanje opsega (engl. Scope Resolution Operator)	317
Kako pozvati nadjačanu statičku metodu iz osnovne klase ako je tu metodu nadjačala izvedena klasa	317
self i \$parent ključne riječi za pristup članovima klase	319
Enumeratori	321
Imenski prostori (engl. Namespaces).....	322
Što je namespace?	322
Kako definirati namespace?	323
Kako koristiti namespace?.....	323
Korištenje use izjave.....	323
Logika i vidljivost namespace-a.....	323
Prednosti korištenja namespace-a.....	324
Zadatak: upravljanje događajima u PHP	324
Uvod u iznimke.....	351
Sintaksa iznimki	351
Vrste iznimki.....	352
Exception Driven Development (EDD)	353
Primjer iznimki.....	355
Dizajn obrasci	359
Singleton obrazac	359
Svojstva	362

Struktura.....	362
Primjenjivost	363
Kako implementirati.....	363
Za i protiv.....	364
Idejni (konceptualan) primjer.....	364
Realan primjer	364
Facade obrazac.....	368
Problem i rješenje	368
Analogija u stvarnom svijetu	369
Struktura.....	369
Pseudokod.....	370
Primjenjivost	370
Kako implementirati.....	370
Za i protiv.....	370
SOLID Principi	370
Single Responsibility Principle.....	371
Open/Closed Principle.....	371
Liskov Substitution Principle	372
Interface Segregation Principle	373
Dependency Inversion Principle.....	374
Spajanje PHP s SQL bazama	375
Composer	394
Provjera imamo li instaliran Composer	394
Instalacija Composera	394
Ažuriranje (update) Composera	394
Podešavanje Composera na projektu (datoteka <code>composer.json</code>)	395
Composer naredbe.....	399
Provjera novih verzija ovisnosti.....	400
Inicijalizacija novog GIT repozitorija	400
PSR-4	401
MVC arhitektura.....	402
Suradnja među MVC komponentama.....	403
Organizacija MVC arhitekture	405
<code>index.php</code> ulazna točka	406
Završavamo <code>index.php</code>	409

Definirajmo Router.....	410
Završimo Router	411
Dispatcher	420
Kontroler	425
Fasade	426
Pogledi.....	432
Modeli	435
Kreiranje sistema autentifikacije.....	450
Kreiranje sistema autorizacije	472
Dependency Injection (DI) u PHP-u.....	479
Testiranje.....	482
Vrste testova	482
Jedinični testovi (Unit tests).....	482
Integracijski testovi (Integration tests)	482
Funkcionalni testovi (Functional tests)	482
End-to-end testovi (E2E tests).....	483
Performansijski testovi (Performance tests).....	483
Sigurnosni testovi (Security tests).....	483
Regresijski testovi (Regression tests)	483
Vrste integracijskih testova.....	484
Sistemske integracijske testovi:	484
API integracijski testovi:	484
Baza podataka integracijski testovi.....	484
Test-Driven Development (TDD) i njegova primjena u PHP-u pomoću PHPUnit-a.....	484
Što je TDD?	484
PHPUnit	485
Glavna svojstva PHPUnit-a	485
Instalacija PHPUnit-a	485
Testiranje.....	488
Pokretanje testova	488
Priprema vašeg projekta	488
Struktura testova.....	489
Pisanje testova	489
Razumijevanje rezultata.....	489
Koncept PHPUnit-a.....	490

Dodatne vještine i alati.....	490
Testiranje naše aplikacije unit testovima	491
Što su mock-ovi?	499
Data Providers.....	499
Asertivni izrazi u PHPUnit-u.....	501
Stub u PHP-u.....	503
Primjer Unit testova	506
Kontejneri.....	515
Kubernetes	515
OpenShift.....	516
Razlike između Kubernetes-a i OpenShift-a	516

Dodavanje ekstenzija Visual Studio Code koje su vezane za PHP

PHP Intelephense



Visual Studio Code ima odličnu podršku za automatsko dovršavanje koda. Glavna svojstva su:

- **Automatsko dovršavanje koda** - VS Code koristi IntelliSense za predlaganje relevantnih funkcija, klase, konstanti i drugih simbola na temelju konteksta. Ovo uvelike povećava produktivnost programiranja. Prilikom pisanja PHP koda, Intelephense će automatski predlagati relevantne funkcije, klase, konstante i druge simbole koristeći IntelliSense. Možete prihvati prijedlog pritiskom na Tab ili Enter ili pritisnuti Ctrl+Space da biste ručno pokrenuli IntelliSense i pregledali dostupne prijedloge dovršavanja koda.
- **Dovršavanje predložaka (engl. template) za HTML i Twig** - Kod uključivanja predložaka, VS Code će predložiti postojeće web predloške koje možete odabratи.

A screenshot of the Visual Studio Code interface showing a completion tooltip for the function 'var_dump'. The tooltip displays the function signature, parameters, and a detailed description: 'Dumps information about a variable'. It also shows the PHP documentation block, return type 'void', and a link to the official PHP manual at <https://php.net/manual/en/function.var-dump.php>. The tooltip is overlaid on a dark background of the code editor.

PHP Debug

PHP Debug je snažna ekstenzija za Visual Studio Code koja omogućuje napredne mogućnosti debugiranja PHP koda koristeći Xdebug. Ključna svojstva uključuju:

- **Postavljanje breakpointa** - Možete postaviti breakpointe na željene linije koda da biste prekinuli izvršavanje i pregledali varijable i stanje programa u tom trenutku.

Korak po korak - Možete koračati kroz kod liniju po liniju, ući u funkcije i izaći iz njih da biste pratili tijek izvršavanja.

- **Inspekcija varijabli** - Dok je program zaustavljen na breakpointu, možete pregledati vrijednosti lokalnih varijabli i superglobalnih varijabli poput `$_GET`, `$_POST` itd.
- **Evaluacija izraza** - Možete evaluirati PHP izraze u debug konzoli da biste vidjeli njihove vrijednosti.
- **Podrška za Xdebug** - PHP Debug ekstenzija služi kao adapter između VS Code i Xdebug debugger ekstenzije za PHP.

Da biste koristili PHP Debug, trebate imati instaliran Xdebug na PHP okruženju. Zatim instalirajte PHP Debug ekstenziju iz VS Code tržišta proširenja.

Možete konfigurirati debugger u `launch.json` datoteci, npr. specificirati skriptu za pokretanje, postaviti putanje za mapiranje lokalnih i udaljenih datoteka kod remote debugiranja itd.

Uz ove alate, možete efikasno debugirati PHP aplikacije u Visual Studio Code, pratiti izvršavanje, pregledavati varijable i rješavati probleme u kodu.

```
<?php
define('TEST_CONSTANT', 123);
echo "Hello World\n";
$variableThatIsNotSet;
$anArray = array(1, 'test' => 2);
$largeArray = array_fill(0, 100, 'test');
$arrayWithSpaceKey = array('space key' => 1);
$object = new DateTime;
$aBoolean = True;
$aFloat = 1.23;
$aString = '123';
$aInt = 123;
$aNullValue = null;
$aEmptyString = '';
function a_function($parameter)
{
    echo "function breakpoint";
}
a_function(123);
// Notice (undefined index)
echo $anArray['undefined_index'];
// Exception
throw new Exception('this is an exception');

```

Kako instalirati:

- skinuti verziju za PHP 8.2 (64-Bit): https://xdebug.org/files/php_xdebug-3.2.0-8.2-vs16-x86_64.dll
- `dll` premjestiti u `C:\xampp\php\ext`
- promjeniti mu ime u `php_xdebug.dll`
- otvoriti datoteku `C:\xampp\php\php.ini` s Notepad++
- isključiti `output_buffering = Off`

- Naći dio koji počinje s [XDebug] ili ga kreirati i s copy/paste prebaciti ove redove:

```
zend_extension=xdebug
xdebug.mode=debug
xdebug.start_with_request=trigger
xdebug.modeidekey = VS CODE
xdebug.client_host = "127.0.0.1"
xdebug.discover_client_host = 1
xdebug.log="/tmp/xdebug.log"
xdebug.cli_color = 1xdebug.log_level=0
```

- restartati Apache i treba raditi
- u direktoriju u kojem radimo formira se datoteka `.vscode` u kojoj se nalazi konfiguracijska datoteka `launch.json` koju je po potrebi moguće dodatno iskonfigurirati
- kada je sve OK, instalirati ekstenziju PHP debug, u sredini ekrana na vrhu pojavit će se mali meni kojim se upravlja debugger-om



Što je PHP?

- rekurzivni akronim za „PHP: Hypertext Preprocessor“, prije „Personal Home Page Tools“
- interpretirani server side programski jezik koji se orientira po sintaksi jezika C
- PHP je prvi put razvio Rasmus Lerdorf 1994. godine
- namijenjen prvenstveno programiranju dinamičnih mrežnih stranica
- ističe se širokom podrškom raznih baza podataka i internetskih protokola kao i brojnim programerskim zbirkama
- trenutna verzija 8.3

PHP sintaksa

- PHP interpreter izvršava samo PHP kod unutar svojih markera
- PHP markeri su `<?php` za otvaranje i `?>` za zatvaranje PHP odjeljaka
- svrha svih tih markera je odvojiti PHP kod od ne-PHP koda, kao što je JavaScript kod ili HTML tag
- varijable imaju predznak simbola dolara `(\$)`, a tip ne mora biti unaprijed određen
- imena varijabli su osjetljiva na mala i velika slova (engl. case sensitive)

PHP izrazi

- izrazi (engl. expressions) su jedan od osnovnih pojmoveva svakog višeg programskog jezika, pa i PHP-a
- izraz računa i daje rezultat ili uzrokuje neku aktivnost (koja se ponekad naziva bočni efekt)
- izrazi su kombinacije operatara i operanda; izraz je sve ono što ima neku vrijednost
- vrijednost nekog izraza određuje se na osnovu sintakse izraza, odnosno pravila prvenstva svakog od operatara u izrazu

```
1  <?php
2
3  // Primjer izraza
4  $text = 'Hello World';
5  echo $text;
6
7  $x = 5;
8  $y = $x + 10;
9  echo $y;
10
11 ?>
```

PHP iskazi

- naredbe (engl. statements) ili iskazi određuju neku akciju
- svaka PHP skripta je sastavljena od niza naredbi
- po pravilu, naredbe se izvršavaju onim redom kojim su navedene u programu, ali ovaj redoslijed može se promijeniti nekom od naredbi za prijenos toka programa.

```
1  <?php
2
3  // Primjer izraza
4  $text = 'Hello World';
5  echo $text;
6
7  $x = 5;
8  $y = $x + 10;
9  echo $y;
10
11 ?>
```

PHP direktive include i require

- **INCLUDE**: će pokušati pronaći i uključiti navedenu datoteku svaki put kada je pozvana. Ako datoteka nije pronađena, PHP će izbaciti upozorenje, ali će nastaviti s izvršavanjem.
- **REQUIRE**: će učiniti isto što i INCLUDE, ali će PHP izbaciti grešku umjesto upozorenja ako datoteka nije pronađena.
- **INCLUDE_ONCE**: će učiniti isto što i INCLUDE, ali će uključiti datoteku samo prvi put kada se pozove. Naknadni pozivi bit će zanemarenici.
- **REQUIRE_ONCE**: će učiniti isto što i REQUIRE, ali će uključiti datoteku samo prvi put kada se pozove. Naknadni pozivi bit će zanemarenici.

```
1  <?php
2
3  include_once 'include_require.php';
4  require_once 'include_require.php';
5  include 'include_require.php';
6  require 'include_require.php';
7
8 ?>
```

U praksi je puno bolje koristiti `include_once`. Još bolje je koristiti autoloadere jer sami učitavaju. O tome će biti kasnije riječi. Postoje i situacije kada je bolje koristiti `include` ali su vrlo rijetke. Moguće je naravno koristiti i dvostrukе i jednostrukе navodnike. Jednostruksi navodnici nisu dostupni s našom tastaturom ali moguće je dobiti ih s kombinacijom Alt+39 na numeričkoj.

a.php:

```
<?php  
    echo 'Pozdrav iz a datoteke ';  
  
    include 'b.php';  
?>
```

b.php:

```
<?php  
    echo 'Pozdrav iz b datoteke ';  
  
?>
```

index.php:

```
<h1>Uvod u PHP</h1>  
<?php  
    echo "<p>Backend Developer</p>";  
//error_reporting()  
  
    require 'a.php';  
    include 'b.php';  
?>
```

Izlaz je:

Uvod u PHP

Backend Developer

Pozdrav iz a datoteke Pozdrav iz b datoteke Pozdrav iz b datoteke

Ako želimo da se pojavi samo jednom umjesto `include`, koristit ćemo `include_once`.

Uvod u PHP

Backend Developer

Pozdrav iz a datoteke Pozdrav iz b datoteke

Ako je u `include` ili `include_once` datoteka koju ne može pronaći, javit će grešku.

```
<h1>Uvod u PHP</h1>
<?php
    echo "<p>Backend Developer<p>";

    require 'ab.php'; // ne postoji ta datoteka
    include_once 'b.php';
?>
```

Uvod u PHP

Backend Developer

(!) Warning: require(ab.php): Failed to open stream: No such file or directory in C:\xampp\htdocs\Algebra\PHP Osnove\Predavanje01\index.php on line 6				
Call Stack				
#	Time	Memory	Function	Location
1	0.0006		404768{main}()	...\index.php:0

(!) Fatal error: Uncaught Error: Failed opening required 'ab.php' (include_path='C:\xampp\php\PEAR') in C:\xampp\htdocs\Algebra\PHP Osnove\Predavanje01\index.php on line 6				
(!) Error: Failed opening required 'ab.php' (include_path='C:\xampp\php\PEAR') in C:\xampp\htdocs\Algebra\PHP Osnove\Predavanje01\index.php on line 6				
Call Stack				
#	Time	Memory	Function	Location
1	0.0006		404768{main}()	...\index.php:0

Zbog ovoga u produkciji želimo ugastiti greške kako hakeri ne bi došli do informacija. Zato ugasimo dojavljivanje grešaka s `error_reporting(0)`;

```
<h1>Uvod u PHP</h1>
<?php
    echo "<p>Backend Developer<p>";
    error_reporting(0);

    require 'ab.php'; // ovo izaziva fatal error i zaustavi daljnje izvođenje koda
    include 'ab.php'; // ovdje se dogodio warning
    include_once 'b.php';
?>
```

Ako je uključen `error_reporting()`, nećemo vidjeti niti jednu grešku, ni fatal error ni warning. **Dakle ovo treba biti isključeno isključivo u produkciji!** `error_reporting()` spriječava i zapisivanje u apache log datoteku.

PHP interpreter - princip rada

PHP interpreter je odgovoran za izvršavanje PHP koda. Uzima vaš PHP početni kod kao ulaz, obrađuje ga i proizvodi izlaz, koji je obično HTML ili drugi sadržaj koji se može poslati web browseru ili klijentu. Proces rada PHP interpretera može se podijeliti u nekoliko koraka:

1. Leksička analiza (tokenizacija):

- PHP interpreter čita početni kod i rastavlja ga na manje jedinice koje se nazivaju tokeni. Tokeni su osnovni građevni blokovi PHP jezika, kao što su ključne riječi, varijable, operatori i literalni.

2. Analiza (engl. parsing) sintakse:

- Tokenizirani kod se zatim analizira (engl. parsing) kako bi se izgradilo stablo sintakse, također poznato kao stablo apstraktne sintakse (engl. abstract syntax tree - AST). Stablo sintakse predstavlja hijerarhijsku strukturu koda i provodi pravila gramatike PHP jezika.

3. Semantička analiza:

- Interpreter provodi semantičku analizu na stablu sintakse kako bi osigurao da se kod pridržava pravila PHP jezika i da su varijable i funkcije definirane i pravilno korištene.

4. Kompilacija:

- Interpreter generira međukod (također poznat kao bajt kod) iz stabla sintakse. Ovaj posredni kod predstavlja niži nivo PHP koda i učinkovitiji je za izvođenje.

5. Izvođenje:

- Bajt kod se izvršava s PHP mehanizmom. Stroj tumači bajt kod i izvodi potrebne operacije, kao što su dodjele varijabli, pozivi funkcija i naredbe toka kontrole.

6. Dinamička dodjela tipa:

- PHP je dinamički tipiziran jezik, što znači da se tipovi varijabli određuju tokom izvođenja. Interpreter rukuje pretvaranjima tipova i operacijama u skladu s tim.

7. Funkcija i rješavanje (rezolucija) klase:

- Tokom izvođenja, interpreter rješava pozive funkcija i metoda, kao i definicije klase i nasljeđivanje. Učitava klase i izvršava metode kako se pozivaju.

8. Upravljanje memorijom:

- PHP interpreter upravlja alokacijom (dodjelom) i dealokacijom (oslobađanjem) memorije za varijable i objekte. Također upravlja skupljanjem otpadaka (engl. garbage collection) kako bi povratio memoriju od objekata koji se više ne koriste.

9. Generiranje izlaza:

- Dok se kôd izvršava, interpreter može generirati izlaz, kao što je HTML ili drugi sadržaj, na temelju PHP koda i svih podataka dohvaćenih iz baza podataka ili drugih izvora.

10. Odgovor klijentu:

- Nakon što je izvršavanje dovršeno i bilo koji izlaz je generiran, interpreter šalje rezultat kao sadržaj nazad web serveru, koji ga zatim šalje klijentu (browseru).

Važno je napomenuti da se PHP može izvršavati u različitim okolinama, koji su u sklopu web servera (npr. Apache s PHP modulom ili PHP-FPM) ili kroz komandni interface red (engl. command line interface - CLI). Uz to, PHP podržava različite ekstenzije i biblioteke koje pružaju dodatne funkcije, kao što je povezivanje s bazom podataka, operacija s datotekama i više.

Prođimo kroz to kako PHP interpreter radi kada izvršite jednostavan "Hello, World!" PHP skripta u terminalu pomoću **command line interface-a, tj. CLI**.

1. Komandni red:

- Otvorite terminal ili komandni (naredbeni) red.

2. Pokretanje skripte:

- Dođite do direktorija u kojem se nalazi vaša PHP skripta (nazovimo je `hello.php`).

3. PHP interpreter:

- Pokrenite sljedeću naredbu u terminalu:

```
php zdravo.php
```

4. Obrada (procesiranje) PHP interpretera:

- PHP interpreter čita sadržaj `hello.php` red po red.
- Prepoznaje početni `<?php` tag kao početak bloka PHP koda.

5. Izvođenje:

- Interpretator izvršava `echo` naredbu unutar bloka PHP koda.
- Naredba echo daje string `Hello, World!`.

6. Generiranje izlaza:

- PHP interpreter generira izlaz "Hello, World!".

7. Izlaz terminala:

- Izlaz koji generira PHP interpreter prikazuje se direktno u terminalu.

Evo kako koraci izgledaju na terminalu:

```
$ php hello.php  
Zdravo, svijete!
```

U ovom scenariju, PHP interpreter izvršava PHP skriptu direktno u okruženju terminala. Obrađuje kod, generira izlaz i odmah ga prikazuje u prozoru terminala. Ovaj pristup je koristan za pokretanje PHP skripti bez potrebe za interakcijom web servera ili browsera.

PHP varijable i tipovi podataka

`echo` i `print` za ispis

Za prikaz varijabli najčešće koristimo izjave `echo` i `print`. To nisu funkcije.

`echo` je jezička konstrukcija koja se koristi za ispisivanje jedne ili više vrijednosti na standardni izlaz. Budući da je `echo` izjava, a ne funkcija, ne zahtjeva zagrade, iako ih možete koristiti ako želite.

```
echo "Hello, world!"; // Bez zagrada  
echo("Hello, world!"); // Sa zgradama
```

`echo` može ispisivati više argumenata odvojenih zarezima:

```
echo "Hello, ", "world!", "\n";  
echo "<p>Backend Developer</p> "
```

`print` je također jezička konstrukcija, ali za razliku od `echo`, ponaša se kao funkcija jer vraća vrijednost (uvijek vraća 1). Također, `print` može ispisivati samo jedan argument.

```
print "Hello, world!"; // Bez zagrada  
print("Hello, world!"); // Sa zgradama
```

`echo` može ispisivati više argumenata odvojenih zarezima za razliku od `print`. `echo` ne vraća vrijednost, `print` vraća vrijednost 1.

Prebacivanje u novi red sa `\n` ili `PHP_EOL`

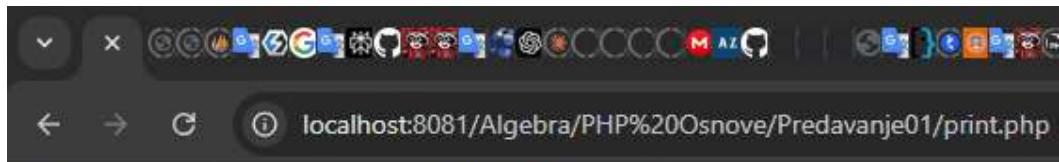
Escape sekvenca

```
<?php  
echo "Ovo je prvi red.\nOvo je drugi red.";  
?>
```

`PHP_EOL` je konstanta u PHP-u koja predstavlja krajnji znak novog reda specifičan za operativni sistem na kojem se skripta izvršava. Na Unix/Linux sistemima to je `\n`, dok je na Windows sistemima `\r\n`. Korištenjem `PHP_EOL`, vaš kod postaje prenosiviji između različitih operativnih sistema:

```
<?php  
echo "Ovo je prvi red.\nOvo je drugi red.\n";  
  
echo "Ovo je treći red.<br> Ovo je četvrti red.<br>";  
  
echo "Ovo je peti red." . PHP_EOL . "Ovo je šesti red." . PHP_EOL;  
  
?>
```

```
Ovo je prvi red.  
Ovo je drugi red.  
Ovo je treći red.<br> Ovo je četvrti red.<br>Ovo je peti red.  
Ovo je šesti red.
```



Ovo je prvi red. Ovo je drugi red. Ovo je treći red.
Ovo je četvrti red.
Ovo je peti red.". PHP_EOL . "Ovo je šesti red.

Jednostruki i dvostruki navodnici

Jednostruki navodnici ("")

- Sve unutar jednostrukih navodnika se tretira kao doslovni tekst.
- Escape sekvene kao što su `\n` se ne tumače (izuzetak je za `\`` i `\```).
- Jednostruke navodnike možemo dobiti sa Alt+39

```
<?php  
echo 'Ovo je novi red\n'; // Ispisat će doslovno: Ovo je novi red\n
```

Kod povezivanja stringova koristeći jednostrukе navodnike:

```
<?php  
$ime = 'Branko';  
echo 'Zdravo, ' . $ime . '!'; // Povezivanje stringova sa točkom  
?>
```

Dvostruki navodnici ("")

- Interpretiraju escape sekvence kao što su `\n`, `\t` itd.
- Interpretiraju varijable unutar stringa.

```
<?php  
$newline = "\n";  
echo "Ovo je novi red$newline"; // Ispisat će: Ovo je novi red (i novi red)
```

Kod dvostrukih navodnika, varijable se mogu direktno interpolirati u string:

```
<?php  
$ime = "Marko";  
echo "Zdravo, $ime!"; // Varijabla $ime će biti interpolirana  
?>
```

PHP varijable

- varijable su veličine koje tokom izvršavanja PHP programa mogu mijenjati svoju vrijednost u PHP-u oznake za varijable moraju početi znakom dolara \$
- iza znaka dolara naziv mora početi slovom ili znakom „_”
- ime varijable ne smije početi brojem, ali ga može sadržavati na bilo kojoj drugoj poziciji u imenu
- PHP razlikuje velika i mala slova u nazivu varijable
- ime varijable mora biti definirano u skladu sa značenjem vrijednosti koja se pamti u okviru varijable
- inicijalna vrijednost varijable se dodjeljuje pomoću znaka (=) jednako

PHP varijable - primjer

```
1  <?php
2      // imena varijabli su osjetljiva na velika i mala slova
3      $ime;
4      $Ime;
5
6      // primjer neispravne definicije varijable
7      $12ime;
8      $-ime;
9      $ ime;
10
11     // primjer ispravne definicije varijable
12     $ime12;
13     $_ime;
14     $ime_korisnika;
15
16     // dodjeljivanje vrijednosti varijabli
17     $ime = 'Marko';
18 ?>
```

PHP konstante

- konstanta sadrži vrijednost isto kao i varijabla, ali za razliku od varijable, kojoj se vrijednost može mijenjati tokom izvođenja programa, vrijednost konstante se ne mijenja
- označke za konstante su bez znaka dolara ispred naziva
- imena konstanti poželjno je pisati velikim slovima kako bi bile prepoznatljive, ali mogu se pisati i malim slovima
- konstantu u programu možemo pisati kao vrijednost bez ikakve označke, (na primjer 3.14 za vrijednost PI)

PHP konstante - primjer

```

1 <?php
2 // Imena konstanti trebalo bi pisati velikim slovima
3 // kako bi ih kasnije u kodu razlikovali
4
5 define('PI', 3.14);
6
7 // Konstante možemo definirati i malim slovima,
8 // ali to nije poželjno
9
10 define('pi', 3.14);
11 ?>

```

Postoje i unaprijed definirane konstante u PHP-u, kao što su [LINE](#)(broj reda), [FILE](#)(putanja i datoteka), [DIR](#) (putanja) Ove konstante vraćaju informacije o trenutnom kontekstu izvršavanja.

Konstante se pozivaju bez \$, za razliku od varijabli.

PHP tipovi podataka

- u PHP-u ne postoje fiksni tipovi podataka
- ne morate definirati tip varijable prije njenog korištenja
- varijablu možete deklarirati bilo kada unutar skripte i pridruživati joj različite tipove podataka tokom izvođenja skripte (iako ovo nije uobičajena niti previše praktična praksa)
- isto tako, možete mijenjati tip podataka neke varijable jednog te istog sadržaja

Tipovi podataka koje podržava PHP su:

- cijeli brojevi (integer)
- realni brojevi (floating-point numbers)
- tekstualni podaci (string)
- logičke varijable
- matrice (nizovi)
- objekti
- **NULL** - varijable kojima nije dodijeljena vrijednost imaju vrijednost tipa **NULL**. **NULL** je nepostojeća vrijednost, prazno polje ili polje bez vrijednosti.

Napomena: i nula (0) je vrijednost i zato ne treba miješati nulu i **NULL**.

Cijeli brojevi (integer)

U ovaj tip varijable možemo pohraniti pozitivne i negativne brojeve u rasponu od -2.147.483.648 do 2.147.483.647 tj. 32 bita podataka.

Možemo ih zapisati u dekadskom, oktalnom ili heksadecimalnom zapisu.

```

1 <?php
2     // pozitivan dekadski broj
3     $int_var = 123;
4     // negativni dekadski broj
5     $int_var = -123;
6     // oktalni broj
7     $int_var = 0123;
8     // heksadecimalni broj
9     $int_var = 0x123;
10 ?>

```

Realni brojevi (float)

Veličina brojeva s pomičnom točkom zavisi o platformi, a najčešća maksimalna vrijednost je $\sim 1.8e308$ s preciznošću na 14 decimala (IEEE 64-bitni format).

Budite pažljivi kada koristite realne brojeve. Naime, njihova točnost nije garantirana (ima veze sa pretvaranjem ovog broja u njegov binarni ekvivalent).

Recimo, 0.33333 nikada neće biti točno prebačen u binarni ekvivalent).

Zato, nemojte ih uspoređivati za jednakost i vjerovati im do posljednje decimale.

```

1 <?php
2     // Realne brojeve možemo pisati na dva načina
3     $float_var = 0.123;
4     // ili
5     $float_var = 1.123e8;
6
7     // Ne ispravan primjer relanog broja
8     $float_var = 0,123;
9 ?>

```

Tekstualni podaci (string)

- stringovi se mogu specificirati korištenjem jednog od dva seta delimitera
- sadržaj varijable tipa string se nalazi između navodnika. Možete koristiti dvostrukе i jednostrukе navodnike
- postoje razlike u ispisu sadržaja, ovisno o tipu navodnika koje koristite

PHP tipovi podataka – tekstualni podaci (string)

Korištenjem dvostrukih navodnika ("") možete koristiti 'special characters'.

To su posebni znakovi koji govore PHP-u da izvrši određene radnje pri ispisu sadržaja varijable.

To su znakovi koji slijede iza znaka backslash (\).

On se ujedno koristi za preskakanje određenog znaka unutar stringa.

```

1 <?php
2 // Realne brojeve mojemo pisati na dva načina
3 $float_var = 0.123;
4 // ili
5 $float_var = 1.123e8;
6
7 // Ne ispravan primjer relanog broja
8 $float_var = 0,123;
9 ?>
```

Drugi način odvajanja stringova su jednostruki navodnici (').

Kad je string unutar jednostrukih navodnika, jedini escape znakovi koje će PHP razumjeti su "\N" i "\\".

Varijable navedene unutar jednostrukih navodnika se neće koristiti kao varijable, već kao najobičniji niz znakova.

Logičke vrijednosti (boolean)

Logički tip podataka ima dvije moguće vrijednosti : true i false.

Koristi se u okviru izraza, varijabli, konstanti, kao i bilo koji drugi primitivni tip.

Svaki izraz se može prikazati kao logički (boolean):

0, 0.0, '0', " su vrijednosti za false

sve ostale vrijednosti se prepoznaju kao true

```

1 <?php
2 // Logička vrijednost true
3 $true = true;
4 // Logička vrijednost false
5 $false = false;
6
7 // Ovaj tip podataka je također
8 // rezultat logičkih izraza
9 // (npr. iz if uvjeta) te nekih PHP funkcija:
10 $age = 20;
11 $is_of_age = ($age >= 18);
12 // Kao rezultat vrijednost varijable is_of_age
13 // će biti logička vrijednost true
14 echo $is_of_age;
15 ?>
```

PHP reference

Oznaka za referencu je znak „&”.

Reference najbolje možemo objasniti na primjeru s referencom (vidi sliku).

Drugim iskazom kreiramo kopiju varijable `$a` koju imenujemo kao `$b`. Nakon toga će u daljem toku programa, ako promijenimo vrijednost varijable `$a`, vrijednost varijable `$b` i dalje ostati nepromjenjena.

Korištenjem reference efekt će biti drukčiji. Ako u daljem toku programa promijenimo vrijednost varijable `$a`, vrijednost varijable `$b` bit će automatski promjenjena u novu vrijednost koju ima varijabla `$a`.

```
<?php  
    // Primjer bez reference  
    $a = 5;  
    $b = $a;  
    $a = 6;  
    print $b; // 5  
  
    // Vrijednost varijable b ostaje nepromjenjena  
  
    // Primjer s referencom  
    $a = 5;  
    $b = &$a;  
    $a = 6;  
    print $b; // 6  
    // Vrijednost varijable b će poprimiti novu vrijednost varijable a  
?>
```

Operator `$b` stvara referencu na varijablu `$a`, što znači da `$b` postaje alternativno ime za istu varijablu kao `$a`.

Vježba - PHP varijable i konstante

- Definirajte nekoliko različitih varijabli i dodijelite im sljedeće tipove podataka:
 - o cijeli broj (integer)
 - o realni broj (floating-point number)
 - o tekstualni podatak (string)
 - o logička vrijednost (boolean)

- Ispišite definirane varijable.
- Definirajte nekoliko konstanti koje poznajete (npr. pi), te ih zatim ispišite.
- Definirajte varijablu `a` i dodijelite joj neku vrijednost. Zatim definirajte varijablu `b` i referencirajte je na varijablu `a`. Ispišite varijablu `b`. Zatim promijenite vrijednost u varijabli `a` i ponovo ispišite varijablu `b`.

```
$a = 10;  
$b = &$a;  
  
echo "Prije promjene:\n";
```

```
echo "\$a = $a\n";
echo "\$b = $b\n";

$b = 15; // Mijenjamo vrijednost preko reference $b

echo "Nakon promjene preko $b:\n";
echo "\$a = $a\n";
echo "\$b = $b\n";

$a = 20; // Mijenjamo vrijednost preko $a

echo "Nakon promjene preko $a:\n";
echo "\$a = $a\n";
echo "\$b = $b\n";
```

Rezultat je:

```
Prije promjene:
$a = 10
$b = 10
Nakon promjene preko 15:
$a = 15
$b = 15
Nakon promjene preko 20:
$a = 20
$b = 20
```

```
$ime = "Saša";
echo strtoupper($ime); // SAŠA
echo mb_strtoupper($ime, "utf-8"); //SAŠA
```

```
echo htmlspecialchars("<script>alert(\" DGSDGSDGSDG \")</SCRIPT>");
```

Rezultat je:

```
&lt;script&gt;alert(" DGSDGSDGSDG ");&lt;/SCRIPT&gt;
```

PHP operatori

- u svakom programskom jeziku postoje razni operatori kojima se vrši manipulacija, matematičke operacije ili usporedba vrijednosti
- operatori omogućavaju izvršavanje operacija nad varijablama i konstantama, na primjer zbrajanje, množenje itd.
- operatori se mogu pisati s jednim, dva ili tri znaka (unarni, binarni i ternarni operatori)

U PHP-u postoje sljedeći tipovi operatora:

- operator dodjele
- aritmetički operatori
- operator nad stringom (znakovnim nizom)
- kombinirani operator dodjele
- operatori usporedbe
- logički operatori
- operatori inkrementa i dekrementa
- ostali operatori

Operatori dodjele

Dodjeljuje vrijednost varijabli.

Nema značenje „identično je”.

Pravo značenje je „dodjeljuje se vrijednost izraza na desnoj strani varijabli na lijevoj strani.”

Varijable se mogu pojavljivati i na desnoj strani jednakosti.

```
1 <?php
2     $a = 5;
3     // varijabli a dodjeljujemo vrijednost 5
4
5     $b = 10;
6     $c = $b;
7     // varijabli c dodjeljujemo vrijednost
8     // varijable b
9 ?>
```

Aritmetički operatori

operator zbrajanja (+)

operator oduzimanja (-)

operator množenja (*)

operator dijeljenja (/)

operator modul (%)

```
1 <?php
2     $a = 15;
3     $b = -5;
4
5     echo $a + $b;
6     echo $a - $b;
7     echo $a * $b;
8     echo $a / $b;
9     echo $a % $b;
10    ?>
```

Ne treba posebno objašnjavati ove operatore, osim operatorka modul.

Modul vraća ostatak cjelobrojnog dijeljenja a ako je djeljenik manji vraća njega.

Rezultat operatorka % ima isti predznak kao djeljenik.

```
<?php
    $a = 10;
    $b = 20;
    print $a % $b; // 10
?>
```

Operator nad stringom (znakovnim nizom)

Jedini operator nad stringovima je točka „.”

Ovaj operator se koristi za pridruživanje (konkatenaciju) stringova: spajanje dva i više znakovna izraza u jedan.

```
1 <?php
2     $a = 'Algebra - ';
3     $b = 'PHP ';
4     $c = 'Osnove';
5
6     // Konkatenacija više stringova
7     echo $a.$b.$c;
8 ?>
```

Kombinirani operator dodjele

Operator	Upotreba	Ekvivalentan izrazu
<code>+=</code>	<code>\$a += \$b</code>	<code>\$a = \$a + \$b</code>
<code>-=</code>	<code>\$a -= \$b</code>	<code>\$a = \$a - \$b</code>
<code>*=</code>	<code>\$a *= \$b</code>	<code>\$a = \$a * \$b</code>
<code>/=</code>	<code>\$a /= \$b</code>	<code>\$a = \$a / \$b</code>
<code>%=</code>	<code>\$a %= \$b</code>	<code>\$a = \$a % \$b</code>
<code>.=</code>	<code>\$a .= \$b</code>	<code>\$a = \$a.\$b</code>

```
1 <?php
2     $a = 15;
3     $b = 10;
4
5     echo $a += $b; // Ekvivalent $a = $a + $b
6     echo $a -= $b; // Ekvivalent $a = $a - $b
7     echo $a *= $b; // Ekvivalent $a = $a * $b
8     echo $a /= $b; // Ekvivalent $a = $a / $b
9     echo $a %= $b; // Ekvivalent $a = $a % $b
10    echo $a .= $b; // Ekvivalent $a = $a . $b
11 ?>
```

Operatori usporedbe
operator jednako (`==`)

operator identično (`==`)

operator različito (`!=` ili `<>`)

operator manje od (`<`)

operator veće od (`>`)

operator manje ili jednako od (`<=`)

operator veće ili jednako od (`>=`)

Kao rezultat operadora usporedbe uvijek dobivamo logičku vrijednost `true` (istina) ili `false`

(laž) u ovisnosti o rezultatu usporedbe.

```

1 <?php
2   $a = 10;
3   $b = '10';
4
5   // Varijabla a je jednaka varijabli b (true)
6   var_dump ($a == $b);
7   // Varijabla a nije identična varijabli b (false)
8   var_dump ($a === $b);
9   // Varijabla a nije identična varijabli b (false)
10  var_dump ($a != $b);
11  // Varijabla a nije različita od varijable b (false)
12  var_dump ($a < $b);
13  // Varijabla a nije identična varijabli b (false)
14  var_dump ($a > $b);
15  // Varijabla a je jednaka varijabli b (true)
16  var_dump ($a <= $b);
17  // Varijabla a je jednaka varijabli b (true)
18  var_dump ($a >= $b);
19 ?>

```

Logički operatori

operator negacije (`!`) npr. `!$b`

- vraća true ako je `$b false` i obrnuto

operator konjukcije (`&&`) npr. `$a && $b`

- vraća `true` samo ako su i `$a` i `$b true`

operator konjukcije (`and`) npr. `$a and $b`

- isto kao `&&` samo niži prioritet

operator disjunkcije (`||`) npr. `$a || $b`

- vraća `true` ako je ili `$a` ili `$b true`

operator disjunkcije (`or`) npr. `$a or $b`

- isto kao `||` samo niži prioritet

```

1 <?php
2   $a = 10;
3   $b = 0;
4
5   // Vraća false pošto je varijabla a true
6   var_dump (!$a);
7   // Vraća false pošto je varijabla b false
8   var_dump ($a && $b);
9   // Vraća false pošto je varijabla b false
10  var_dump ($a and $b);
11  // Vraća true pošto je varijabla a true
12  var_dump ($a || $b);
13  // Vraća true pošto je varijabla a true
14  var_dump ($a or $b);
15 ?>

```

Operatori inkrementa i dekrementa

Operatori inkrementa (`$a++`) i dekrementa (`$a--`) su unarni operatori koji vrijednost varijable pored koje su pozicionirani uvećavaju ili umanjuju za jedan.

```

1 <?php
2     $a = 5;
3     $a++;
4     echo $a;
5     // vrijednost varijable a
6     // uvećala se za 1
7     $b = 10;
8     $b--;
9     // vrijednost varijable b
10    // umanjila se za 1
11    $c = $b--;
12    // Vrijednost varijable c je poprimila
13    // vrijednost varijable b, a nakon toga
14    // se je vrijednost varijable b umanjila za 1
15    $c = --$b;
16    // Vrijednost varijable c je poprimila
17    // vrijednost varijable b umanjene za 1
18 ?>

```

Pretvaranje tipova (engl. type juggling) u PHP-u

Type juggling je automatsko pretvaranje podatkovnih tipova u PHP-u kada se operand neodgovarajućeg tipa koristi u matematičkoj ili logičkoj operaciji. PHP pokušava automatski pretvoriti operative u odgovarajući tip podataka za tu operaciju.

Primjeri type juggling-a:

1. Pretvaranje stringa u broj:

- `"42" + 2 = 44` (string `"42"` se pretvara u broj 42)
- `"foo" + 2 = 2` (string `"foo"` se pretvara u broj 0)

2. Pretvaranje boolean u broj:

- `true + 2 = 1` (true se pretvara u broj 1)
- `false + 2 = 0` (false se pretvara u broj 0)

3. Pretvaranje null u broj:

- `null + 2 = 0` (null se pretvara u broj 0)

4. Pretvaranje array u boolean:

- `[1, 2] ? true : false` (matrica koja nije prazna se pretvara u true)
- `[] ? true : false` (prazna matrica se pretvara u false)

Type juggling može dovesti do neočekivanih rezultata ako se ne koristi oprezno. Preporučuje se eksplicitno pretvaranje tipova kada je to potrebno, kako bi se izbjegle neočekivane situacije.

Na primjer, umjesto "42" + 2 bolje je koristiti intval("42") + 2 za eksplisitno pretvaranje stringa u integer.

Vježba operatori

```
<?php

$a = 10;
$b = 20;

echo $a % $b; // 10
echo $a += $b; // 30
echo $a.$b;    //3020 type juggling rezultata u string

$b++;          // $b = $b + 1
var_dump($b); //21

$c = $b++;     // $b = 22, $c = 21
var_dump($c);
var_dump($b);
$c = ++$b;     // $b = 23, $c = 23
var_dump($c);
var_dump($b);

// Operatori usporedbe
$x = 10;
$y = "10";    // vrijednost ista ali nije tip podatka

var_dump($x == $y);    // true
var_dump($x === $y);   // false jer tip nije isti
var_dump($x != $y);    // false
var_dump($x <> $y);   // false
var_dump($x !== $y);   // true jer je suprotno od ==

$y = "A";
var_dump($x > $y);    // false (ne znamo zašto)
var_dump((int)"A");    // int(0)(ne znamo zašto)

// Operatori logički
$a = 0;
$b = 20;
var_dump(!$x);        // false
var_dump($x && $y);  // true
```

```
var_dump($x || $y); // true
```

```
<?php

$a = 5;
$b = 2;
$c = "Prvi string.";
$d = "Drugi string.";

echo -$a . "<br>"; // negacija
echo -$b . "<br>"; // negacija
echo $a + $b . "<br>"; // suma
echo $a - $b . "<br>"; // razlika
echo $a * $b . "<br>"; // množenje
echo $a / $b . "<br>"; // djeljenje
echo $a % $b . "<br>"; // modul
echo $a ** $b . "<br><br>"; // potenciranje

$f = $c . $d;
echo $f . "<br>";

$a += $b;
echo $a . "<br>";

var_dump($a > $b); // true jer je 7 veće od 2
echo "<br>";

var_dump(++$a); // pre-inkrement, $a je uvećan pa ispisani 7+1=8
echo "<br>";
var_dump(--$b); // pre-dekrement, $b je umanjen pa ispisani 2-1=1

?>
```

PHP matrice

Ako imate iskustva s drugim programskim jezicima ili strukturama podataka općenito, možda ste upoznati s dvije strukture podataka koje su vrlo česte i korisne:

- **liste**
 - o poredani skup elemenata
- **mape**
 - o skup elemenata identificiranih ključevima

U PHP-u ne postoje liste i mape, postoje matrice (nizovi). Matrica je struktura podataka koja implementira obje, listu i mapu. Postoji razlika.

Liste (indeksirane matrice) su uređene kolekcije vrijednosti. Svaka vrijednost u listi ima svoj indeks, počevši od 0.

Mape (asocijativne matrice) su posebna vrsta lista u PHP-u gdje svaki element ima svoj ključ (key) i vrijednost (value).

Populacija matrice u PHP-u uključuje kreiranje i punjenje matrice podacima. **Matrice u PHP-u mogu biti jednodimenzionalne (linearne) ili višedimenzionalne (npr. dvodimenzionalne matrice, poznate kao tablice).**

Jednodimenzione matrice

Inicijalizacija matrica

Postoje različite opcije za inicijalizaciju matrice. Možete inicijalizirati praznu matricu ili možete inicijalizirati matricu s podacima. Postoje i različiti načini pisanja istih podataka u matrice.

U ovom primjeru radimo deklaraciju i inicijalizaciju:

```
// Prazna matrica - deklaracija
$jednodimenzionalnaMatrica = array();

// Matrica s inicijalnim vrijednostima
$jednodimenzionalnaMatrica = array(1, 2, 3, 4, 5);

// Alternativni način korištenjem kratke sintakse
$jednodimenzionalnaMatrica = [1, 2, 3, 4, 5];
```

Pogledajmo idući primjer. Ima u sebi deklaraciju i inicijalizaciju. `$empty1` i `$empty2` dva su načina kreiranja prazne matice.

Kasnije ćete vidjeti da se listama rukuje kao [mapama](#). Interno su matrice `$names1` i `$names2` mapa, a ključevi su poredani brojevi.

Ključevi matrice mogu biti bilo koje alfanumeričke vrijednosti, kao što su slova, znakova ili brojeva.

Vrijednosti matrice mogu biti bilo što: slova, znakovi, brojevi, logičke vrijednosti, drugi nizovi i slično.

```
<?php
$empty1 = [];
$empty2 = array();
// Inicijalizacija matrice (liste) s podacima
$names1 = ['Harry', 'Ron', 'Hermione'];
$names2 = array('Harry', 'Ron', 'Hermione');
// Inicijalizacija matrice (mape) s podacima
$status1 = [
    'ime' => 'James Potter',
```

```
'status' => 'dead'  
];  
$status2 = array(  
    'ime' => 'James Potter',  
    'status' => 'dead'  
)  
?  
?
```

Dodavanje (populacija) elemenata

```
$jednodimenzionalnaMatrica[] = 6; // Dodaje 6 na kraj matrice  
$jednodimenzionalnaMatrica[0] = 10; // Mjenja prvi element matrice u 10
```

Višedimenzionalne matrice

Ova matrica je mapa koja sadrži dvije matrice (mape).

Svaka mapa sadrži različite vrijednosti kao što su tekstualni podaci, decimalne vrijednosti i logičke vrijednosti.

Inicijalizacija matrica

```
// Prazna dvodimenzionalna matrica  
$dvodimenzionalnaMatrica = array();  
  
// Matrica s inicijalnim vrijednostima  
$dvodimenzionalnaMatrica = array(  
    array(1, 2, 3),  
    array(4, 5, 6),  
    array(7, 8, 9)  
);  
  
// Alternativni način korištenjem kratke sintakse  
$dvodimenzionalnaMatrica = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
];
```

```
<?php  
// Višedimenziona matrica  
$books = [  
    '1984' => [  
        'autor' => 'George Orwell',  
        'finished' => true,  
        'rate' => 9.5
```

```
],  
    'Romeo i Julia' => [  
        'autor' => 'William Shakespeare',  
        'finished' => false  
    ],  
];  
?>
```

Dodavanje (populacija) elemenata

Matrice nisu nepromjenjive, odnosno mogu se mijenjati nakon inicijalizacije.

Sadržaj matrice možete promijeniti neovisno o tome tretirate li ga kao mapu ili kao listu.

Ako matricu tretirate kao listu, dodajete novi element na kraj matrice.

Ako matricu tretirate kao mapu, potrebno je odrediti ključ koji želite nadjačati.

```
<?php  
    // Matrica kao lista  
    $names = ['Harry', 'Ron', 'Hermione'];  
    // Matrica kao mapa  
    $status = [  
        'name' => 'James Potter',  
        'status' => 'dead',  
        'age' => 24  
    ];  
    // Dodavanje novog elementa na kraj matrice  
    $names[] = 'Neville';  
    // Nadjačavanje ključa u matrici  
    $status['age'] = 32;  
    print_r($names); // ispis matrice  
    echo "<br>"; // dodatni red  
    print_r($status); // ispis matrice  
?>
```

Rezultat je:

```
Array ( [0] => Harry [1] => Ron [2] => Hermione [3] => Neville )  
Array ( [name] => James Potter [status] => dead [age] => 32 )
```

Uklanjanje elemenata

Ako trebate ukloniti element iz matrice, umjesto dodavanja ili ažuriranja, možete upotrijebiti funkciju `unset`.

Nova matrica `$status` sadrži samo ključ `name`.

```
<?php
// Inicijalizacija matrice
$status = [
    'ime' => 'James Potter',
    'status' => 'dead'
];

// Uklanjanje elementa iz matrice
unset($status['status']);
print_r($status);
?>
```

Pristup elementima

Pristup elementima matrice je jednostavan i vrši se preko ključeva.

Kod matrica mapa to je jednostavnije zato što su ključevi unaprijed definirani.

Kod nizova lista prvo morate razumjeti kako liste funkcioniraju.

Već znate da se liste interno tretiraju kao mape s numeričkim ključevima. Prvi ključ je uvijek **0**;

Dakle, matrica s n elemenata imat će ključeve od **0** do **$n-1$** .

Pokušaj pristupa ključu koji ne postoji u matrici vratit će vrijednost `null` i ispisati obavijest da u kodu radite nešto pogrešno.

```
<?php
// Inicijalizacija matrice kao liste
$names = ['Harry', 'Ron', 'Hermione'];
// Pristup elementu
print_r($names[1]); // Ron
// Pristup elementu koji ne postoji
print_r($names[4]);

// Inicijalizacija matrice kao mape
$status = [
    'name' => 'James Potter',
    'status' => 'dead',
    'age' => 24
];
```

```
// Pristup elementu
print_r($status['name']); // James Potter
//Pristup elementu koji ne postoji
print_r($status['birthday']);
?>
```

empty i isset funkcije

`Empty` i `isset` su korisne funkcije za ispitivanje sadržaja matrice.

Ako želite saznati sadrži li matrica elemente ili je prazan, to možete učiniti pomoću funkcije `empty`.

Funkcija `isset` vraća `true` ili `false` ovisno o tome postoji li traženi element u matrici ili ne.

```
<?php
$array = [];
$names = ['Harry', 'Ron', 'Hermione'];

var_dump(empty($array)); // true
var_dump(empty($names)); // false
var_dump(isset($names[2])); // true
var_dump(isset($names[3])); // false
?>
```

Traženje elemenata

Jedna od najčešće korištenih funkcija u traženju elemenata u matrici je `in_array`.

Ova funkcija uzima dvije vrijednosti, vrijednost koju želite tražiti i matricu u kojoj tražite.

Funkcija vraća `true` ako je vrijednost u matrici, a u suprotnom je `false`.

Ponekad je korisnija funkcija `array_search`.

Ova funkcija djeluje na isti način, osim što, umjesto da vrati logičku vrijednost, vraća ključ tamo gdje se nalazi vrijednost koju tražimo, ili `false` ako vrijednost nije pronađena.

```
<?php

$operacijski_sistemi = array("Windows", "Linux", "macOS", "Android", "iOS");

$operativni_sistem = "Linux";
if (in_array($operativni_sistem, $operacijski_sistemi)) {
    echo "Operativni sistem '$operativni_sistem' je pronađen u matrici. <br>";
} else {
    echo "Operativni sistem '$operativni_sistem' nije pronađen u matrici. <br>";
}

$mobilni_os = "Android";
if (in_array($mobilni_os, $operacijski_sistemi)) {
```

```
echo "Mobilni operativni sistem '$mobilni_os' je pronađen u matrici. <br>";
} else {
    echo "Mobilni operativni sistem '$mobilni_os' nije pronađen u matrici. <br>";
}

$osoba = array("ime" => "Marko", " prezime" => "Marković", "starost" => 35);

$prezime = "Marković";
$key = array_search($prezime, $osoba);
if ($key !== false) {
    echo "Prezime '$prezime' je pronađeno u vrijednostima prezime. Ključ pretrage je
" . $key . ".<br>";
} else {
    echo "Prezime '$prezime' nije pronađeno u vrijednostima prezime.<br>";
}

$starost = 35;
$key = array_search($starost, $osoba);
if ($key !== false) {
    echo "Starost od '$starost' je pronađena u vrijednostima starost. Ključ pretrage
je " . $key . ".<br>";
} else {
    echo "Starost od '$starost' nije pronađena u vrijednostima starost.<br>";
}

?>
```

Rezultat je:

```
Operativni sistem 'Linux' je pronađen u matrici.
Mobilni operativni sistem 'Android' je pronađen u matrici.
Prezime 'Marković' je pronađeno u vrijednostima prezime.
Ključ pretrage je prezime.
Starost od '35' je pronađena u vrijednostima starost.
Ključ pretrage je starost.
```

Sortiranje

Matrica se može sortirati na različite načine.

Postoji mogućnost da je redoslijed koji vam je potreban različit od trenutnog.

Matrica je prema zadanim postavkama sortiran prema redoslijedu u kojem su dodani elementi, ali matricu možete sortirati prema njegovom ključu ili prema vrijednosti, uzlazno i silazno.

Nadalje, kad sortirate matricu prema vrijednostima, možete odabrati da sačuvate njihove ključeve ili generirate nove.

Potpuni popis funkcija nalazi se na službenim internetskim stranicama u dokumentaciji.

<http://php.net/manual/en/array.sorting.php>

Ove funkcije uvijek uzimaju jedan argument, matricu i ne vraćaju ništa. Umjesto toga, one direktno sortiraju matricu koju im proslijedujemo.

Prvo inicijaliziramo matricu i dodijelimo je `$properties`.

Tada kreiramo tri varijable koje su kopije originalne matrice.

Zašto to radimo?

Ako sortiramo originalnu matricu, više nećemo imati originalni sadržaj. To nije ono što želimo u ovom konkretnom primjeru, jer želimo vidjeti kako različite funkcije sortiranja utječu na istu matricu.

Na kraju izvodimo tri različite vrste sortiranja i ispisujemo rezultate.

```
1 <?php
2 // Inicijalizacija niza
3 $properties = [
4     'firstname' => 'Tom',
5     'surname' => 'Riddle',
6     'house' => 'Slytherin'
7 ];
8 // duplicitanje niza u nove varijable
9 $properties1 = $properties2 = $properties3 = $properties;
10 // sortiranje po vrijednosti uzlazno
11 // ključevi se resetiraju
12 sort($properties1);
13 var_dump($properties1);
14 // sortiranje po vrijednosti uzlazno
15 // zadržava prvotne ključeve
16 asort($properties3);
17 var_dump($properties3);
18 // sortiranje po ključu uzlazno
19 ksort($properties2);
20 var_dump($properties2);
21 ?>
```

Ostale funkcije

Postoji oko 80 različitih funkcija povezanih s matricama. Kao što možete pretpostaviti, za neke od njih nikada nećete ni čuti jer one imaju vrlo specifične svrhe.

Kompletan popis možete naći na:

<http://php.net/manual/en/book.array.php>

Možemo dobiti popis ključeva matrice funkcijom `array_keys` i popis njegovih vrijednosti funkcijom `array_values`.

Možemo dobiti broj elemenata u nizu funkcijom `count`.

I možemo spojiti dva ili više matrica u jedan funkcijom `array_merge`.

```
1 <?php
2 $properties = [
3     'firstname' => 'Tom',
4     'surname' => 'Riddle',
5     'house' => 'Slytherin'
6 ];
7 // Dohvaćanje ključeva elemenata niza
8 $keys = array_keys($properties);
9 var_dump($keys);
10 // Dohvaćanje vrijednosti elemenata niza
11 $values = array_values($properties);
12 var_dump($values);
13 // Brojanje elemenata niza
14 $size = count($properties);
15 var_dump($size); // 3
16 // Spajanje dva niza u jedan
17 $good = ['Harry', 'Ron', 'Hermione'];
18 $bad = ['Dudley', 'Vernon', 'Petunia'];
19 $all = array_merge($good, $bad);
20 var_dump($all);
21 ?>
```

Vježba 02

Definirajte nekoliko različitih varijabli i dodijelite im sljedeće tipove podataka:

- cijeli broj (`integer`)
- realni broj (`floating-point number`)
- tekstualni podatak (`string`)
- logička vrijednost (`boolean`)
- Ispišite definirane varijable.

Definirajte nekoliko konstanti koje poznajete (npr. `pi`), te ih zatim ispišite.

Definirajte varijablu `a` i dodijelite joj neku vrijednost. Zatim definirajte varijablu `b` i referencirajte je na varijablu `a`. Ispišite varijablu `b`. Zatim promjenite vrijednost u varijabli `a` i ponovo ispišite varijablu `b`.

```
<?php

// cijeli broj (integer)
$cijeli = 3;

// realni broj (floating-point number)
```

```
$realni = 7.65;

// tekstualni podatak (string)
$str = "Ovo je string.";

// logička vrijednost (boolean)
$logicki = true;

// ispis konkatenacijom
echo '$cijeli = ' . $cijeli . '<br>' . '$realni = ' . $realni . '<br>' . '$str = '
. $str . '<br>' . '$logicki = ' . $logicki . '<br><br>';
// ispis interpolacijom
// echo "$cijeli = $cijeli\n$realni = $realni\n$str = $str\n$logicki =
$logicki\n\n";

// konstante
define('PI', 3.14159265); // Pi
define('OULER', 2.7182818284); // Oulerov broj
define('BRZINA_ZVUKA', 340); // brzina zvuka u m/s
define('OMB', "O moj bože");

// ispis konstanti
echo PI . "<br>";
echo OULER . "<br>";
echo BRZINA_ZVUKA . "<br>";
echo OMB . "<br><br>";

$a = 10;
$b =& $a; // b je referenciran na varijablu a

$a = 500; // vrijednost $a se primjenila
echo "\$a:" . $a . " " . "\$b:" . $b. "<br>; // promjenom $a mijenja se i $b

$b = 1000; // vrijednost $b se primjenila
echo "\$a:" . $a . " " . "\$b:" . $b. "<br>; // promjenom $b mijenja se i $a

?>
```

Vježba 03

Definirajte varijable `a`, `b`, `c` i `d`. Varijablama `a` i `b` dodijelite vrijednost tipa `integer`, a varijablama `c` i `d` vrijednost tipa string.

Nad varijablama `a` i `b` primijenite sve aritmetičke operatore i ispišite rezultate.

Nad varijablama `c` i `d` primijenite operator konkatenacije i dobivenu vrijednost dodijelite varijabli `f`, te ispišite vrijednost varijable `f`.

Nad varijablama `a` i `b` primijenite jedan od kombiniranih operatora dodjele i ispišite rezultat.

Nad varijablom `a` i `b` primijenite operator usporedbe (veće od) i pomoću `var_dump()` funkcije ispišite rezultat.

Nad varijablom `a` primijenite operator inkrementa i istovremeno ispišite rezultat.

Nad varijablom `b` primijenite operator dekrementa i istovremeno ispišite rezultat.

```
<?php

$a = 5;
$b = 2;
$c = "Prvi string.";
$d = "Drugi string.";

echo -$b . "<br>";      // negacija
echo $a + $b . "<br>";   // suma
echo $a - $b . "<br>";   // razlika
echo $a * $b . "<br>";   // množenje
echo $a / $b . "<br>";   // djeljenje
echo $a % $b . "<br>";   // modul
echo $a ** $b . "<br><br>"; // potenciranje

$f = $c . $d;
echo $f . "<br>";

$a += $b;
echo $a . "<br>";

var_dump($a > $b);      // true jer je 7 veće od 2
echo "<br>";

var_dump(++$a);          // pre-inkrement, $a je uvećan pa ispisano 7+1=8
echo "<br>";
var_dump(--$b);          // pre-dekrement, $b je uvećan pa ispisano 2-1=1
```

```
?>
```

Evo malo primjera:

```
<?php

// Kreiraj matricu (niz) koji sadrži imena 5 voćaka
// Dodaj na kraj matrice dvije nove voćke korištenjem uglatih zagrada i ugrađene php
array funkcije
// Izbriši prvi element u matrici

// indeks kreće od 0
$fruits = ["jabuke", "banane", "narandže", "kivi", "mango"];

// ako napišemo broj van opsega dobit ćemo upozorenje
// Undefined array key i php se nastavi izvoditi
print_r($fruits[4]);

// dodavanje elemenata matrice
$fruits[] = "ananas";
$fruits[8] = "breskve"; // preskočimo 6 i 7
$fruits[] = "grožđe"; // ovo je 9, nastavlja

array_push($fruits, "kajsije"); // Pusha element na kraj matrice

// var_dump($fruits); // ispisuje ružno
print_r($fruits); // ispisuje ljepše

var_dump($fruits == false); // prazan niz je false

// brisanje
unset($fruits[0]); // ukloni jabuke iz matrice ali ostavi rupu
// izbaci banane i reindeksira sve elemente, dakle kreće od 0
array_shift($fruits);

print_r($fruits);

// kreirati 2 matrice koji sadrže po 3 broja
// Spoji ove 2 matrice u jednu

$prvi = [2,3,7];
$drugi = [2,5,9];
$spojeniNiz = array_merge($prvi, $drugi);

print_r($prvi);
```

```
print_r($drugi);
print_r($spojeniNiz);

$spojeniNiz = [$prvi,$drugi]; // ovo pravi dvodimenzionalnu matricu
[2,3,7][2,5,9]
print_r($spojeniNiz);

$spojeniNiz = [...$prvi,...$drugi]; // radi dekompoziciju na članove
[2,3,7,2,5,9]
print_r($spojeniNiz);

// Kreiraj niz koji sadrži 5 ocjena. Izračunaj prosječnu ocijenu
$ocjene = [2,4,5,3,4];
$rezultat = array_sum($ocjene) / count($ocjene);
print_r($rezultat);
echo $rezultat;

// Kreiraj matricu s 10 brojeva i izdvoji sve brojeve veće od 5 u novu matricu

$brojevi = [1,8,6,4,8,3,7,1,0,5];
$brojeviVeciOdPet = array_filter($brojevi, function($broj) { // callback
funkcija u {}
    return $broj > 5; // anonimna funkcija, ako umjesto $broj > 5 stavimo
true, dobijemo sve brojeve
});

print_r($brojeviVeciOdPet); // rezultat je [1]=>8, [2]=>6, [4]=>8, [6]=>7
$reindeksiraniBrojevi = array_values($brojeviVeciOdPet);
print_r($reindeksiraniBrojevi);?>
```

Rezultat je:

```
mangoArray
(
    [0] => jabuke
    [1] => banane
    [2] => narandže
    [3] => kivi
    [4] => mango
    [5] => ananas
    [8] => breskve
    [9] => grožđe
    [10] => kajsije
)
```

```
C:\xampp\htdocs\Algebra\PHP Osnove\Predavanje04\arrays.php:24:  
bool(false)  
Array  
(  
    [0] => narandže  
    [1] => kivi  
    [2] => mango  
    [3] => ananas  
    [4] => breskve  
    [5] => grožđe  
    [6] => kajsije  
)  
Array  
(  
    [0] => 2  
    [1] => 3  
    [2] => 7  
)  
Array  
(  
    [0] => 2  
    [1] => 5  
    [2] => 9  
)  
Array  
(  
    [0] => 2  
    [1] => 3  
    [2] => 7  
    [3] => 2  
    [4] => 5  
    [5] => 9  
)  
3.63.6Array  
(  
    [1] => 8  
    [2] => 6  
    [4] => 8  
    [6] => 7  
)  
Array  
(  
    [0] => 8  
    [1] => 6
```

```
[2] => 8  
[3] => 7  
)
```

Još jedan primjer:

Kreiraj matricu (niz) koji sadrži podatke o studentima

1. Filtriraj studente koji imaju prosječnu ocjenu iznad 3.5
2. Izračunaj prosječnu ocjenu svih studenata koji su prošli filtraciju
3. Odredi broj studenata u svakoj godini studija

```
$studenti = [  
    ["ime" => "Ana", " prezime" => "Anić", "godina" => 1, "prosjek" => 4.2],  
    ["ime" => "Ivan", " prezime" => "Ivić", "godina" => 2, "prosjek" => 3.1],  
    ["ime" => "Marko", " prezime" => "Mrkovski", "godina" => 3, "prosjek" => 3.7],  
    ["ime" => "Lucija", " prezime" => "Lucić", "godina" => 1, "prosjek" => 4.8],  
    ["ime" => "Hrvoje", " prezime" => "Hrvatko", "godina" => 2, "prosjek" => 4.0],  
];  
  
// Filtriraj studente koji imaju prosječnu ocjenu iznad 3.5  
$vrloDobriStudenti = array_filter($studenti, function($student) {  
    return $student["prosjek"] > 3.5; // uglatom zagradom dolazimo do elementa  
dvodimenzionalne matrice  
});  
print_r($vrloDobriStudenti); // nemamo Ivana Ivanića  
  
// Izračunaj prosječnu ocijenu svih studenata koji su prošli filtraciju  
// array_column vraća vrijednost iz jednog stupca matrice,  
// identificiranog sa ključem, treba nam prosjek  
$arrayProsjeka = array_column($vrloDobriStudenti, "prosjek");  
print_r($arrayProsjeka);  
$prosjekVrloDobrihStudenata = array_sum($arrayProsjeka) /  
count($vrloDobriStudenti);  
print_r($prosjekVrloDobrihStudenata);  
  
// array_column ovdje vraća vrijednost iz stupaca godina ali  
// iz matrice $studenti a ne iz $vrloDobriStudenti  
$arrayGodina = array_column($studenti, "godina");  
// array_count_values broji pojavljivanje svake različite vrijednosti u matrici  
$brojStudenataUSvakojGodini = array_count_values($arrayGodina);  
print_r($brojStudenataUSvakojGodini);
```

```
?>
```

Rezultat je:

```
Array
(
    [0] => Array
        (
            [ime] => Ana
            [prezime] => Anić
            [godina] => 1
            [prosjek] => 4.2
        )

    [2] => Array
        (
            [ime] => Marko
            [prezime] => Mrkovski
            [godina] => 3
            [prosjek] => 3.7
        )

    [3] => Array
        (
            [ime] => Lucija
            [prezime] => Lucić
            [godina] => 1
            [prosjek] => 4.8
        )

    [4] => Array
        (
            [ime] => Hrvoje
            [prezime] => Hrvatko
            [godina] => 2
            [prosjek] => 4
        )
)
```

Zadatak 4:

Imate niz s brojevima. Vaš zadatak je filtrirati niz tako da ukloni sve brojeve manje od 10, zatim udvostručiti preostale brojeve, i na kraju vratiti novi niz s tim transformiranim elementima. Novonastali niz treba sortirati od najmanjeg prema najvećem broju.

```
$ulazniNiz = [2, 5, 10, 15, 20, 25, 30, 3, 7, 8, 12, 17];
```

Novi niz koji sadrži samo brojeve veće ili jednake 10 iz ulaznog niza, svaki udvostručen.

Zabranjeno korištenje bilo kakvih petlji (`for`, `foreach`, `while`, `do-while`).

Možete koristiti funkcije visokog reda za rad s nizovima kao što su `array_filter`, `array_map`, i slično.

Ove funkcije, ali i druge koje vam mogu pomoći u rješavanju, možete pogledati na sljedećem linku.

https://www.w3schools.com/php/php_ref_array.asp

Rezultat koji trebate dobiti:

```
$izlazniNiz = [20, 24, 30, 34, 40, 50, 60];
```

```
<?php  
$ulazniNiz = [2, 5, 10, 15, 20, 25, 30, 3, 7, 8, 12, 17];  
  
// Koristimo array_filter za filtriranje brojeva većih ili jednakih 10  
// callback funkcija u {} - anonimna funkcija vraća veće ili jednake 10  
  
$filtriraniNiz = array_filter($ulazniNiz, function($vrijednost) {  
    return $vrijednost >= 10;  
});  
  
// array_map primjenjuje callback na elemente niza $filtriraniNiz,  
// između {} callback funkcija  
$izlazniNiz = array_map(function($vrijednost) {  
    return $vrijednost * 2;  
}, $filtriraniNiz);  
  
print_r($izlazniNiz);  
  
// Sortiranje rezultata jer nisu elementi prebačeni redom a to se traži u zadatku  
sort($izlazniNiz);  
  
// Ispis rezultata  
print_r($izlazniNiz);
```

```
?>
```

Rezultat je:

```
Array
(
    [2] => 20
    [3] => 30
    [4] => 40
    [5] => 50
    [6] => 60
    [10] => 24
    [11] => 34
)
Array
(
    [0] => 20
    [1] => 24
    [2] => 30
    [3] => 34
    [4] => 40
    [5] => 50
    [6] => 60
)
```

PHP kontrolne strukture

Kontrolne strukture su kao znakovi za preusmjeravanje prometa.

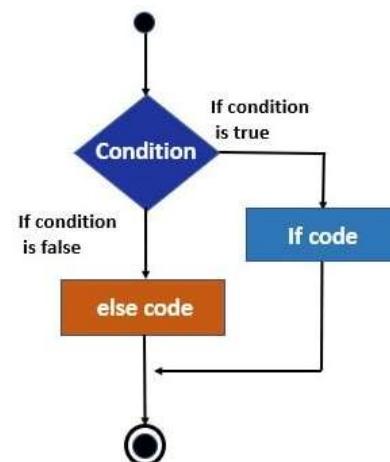
Usmjeravaju tok izvršenja koda ovisno o nekim unaprijed definiranim uslovima.

Postoje različite kontrolne strukture, ali možemo ih kategorizirati u uslovne i petlje.

Uvjetne nam omogućavaju da odaberemo hoćemo li izvršiti izjavu ili ne.

Petlja izvršava izjavu onoliko puta koliko nam je potrebno.

Pogledajmo svaku od njih!



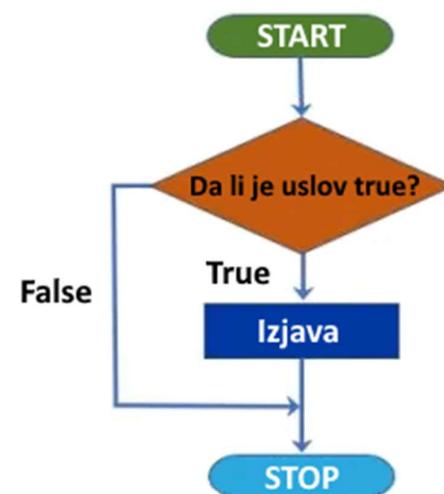
Uvjetovane kontrolne strukture

Provjeravaju uslov koji je uvijek logički izraz.

Ako je izraz istinit, izvršit će sve što se nalazi unutar njegovog bloka koda.

Blok koda je grupa izjava unutar {} zagrada.

Neke od uvjetovanih kontrolnih struktura su `if`, `if-else`, `if-elseif` i `switch-case`.



`if` kontrolna struktura

U ovom primjeru koristimo dvije uvjetovane kontrolne strukture `if`.

Definira se ključnom riječi `if` iza koje slijedi logički izraz u zagradama te blok koda unutar {} zagrada.

Ako je izraz istinit, izvršit će blok, inače će ga preskočiti.

```

1 <?php
2 echo "Before the conditional.";
3 if (4 > 3)
4 {
5     echo "Inside the conditional.";
6 }
7 if (3 > 4)
8 {
9     echo "This will not be printed.";
10}
11 echo "After the conditional.";
12 ?>
  
```

Uslov `3 > 4` se nikada neće izvršiti, što je jasno.

if-else kontrolna struktura

Uslov možete proširiti dodavanjem ključne riječi `else`. Ona govori PHP-u da izvrši neki blok koda ako prethodni uslovi nisu bili zadovoljeni.

Unutar bloka `else`, kod će se izvršiti samo ako uslov u bloku `if` nije zadovoljen.

```
1  <?php
2  if (2 > 3)
3  {
4      echo "Inside the conditional.";
5  }
6  else
7  {
8      echo "Inside the else.";
9  }
10 ?>
```

`else` se ovdje nikada neće izvršiti.

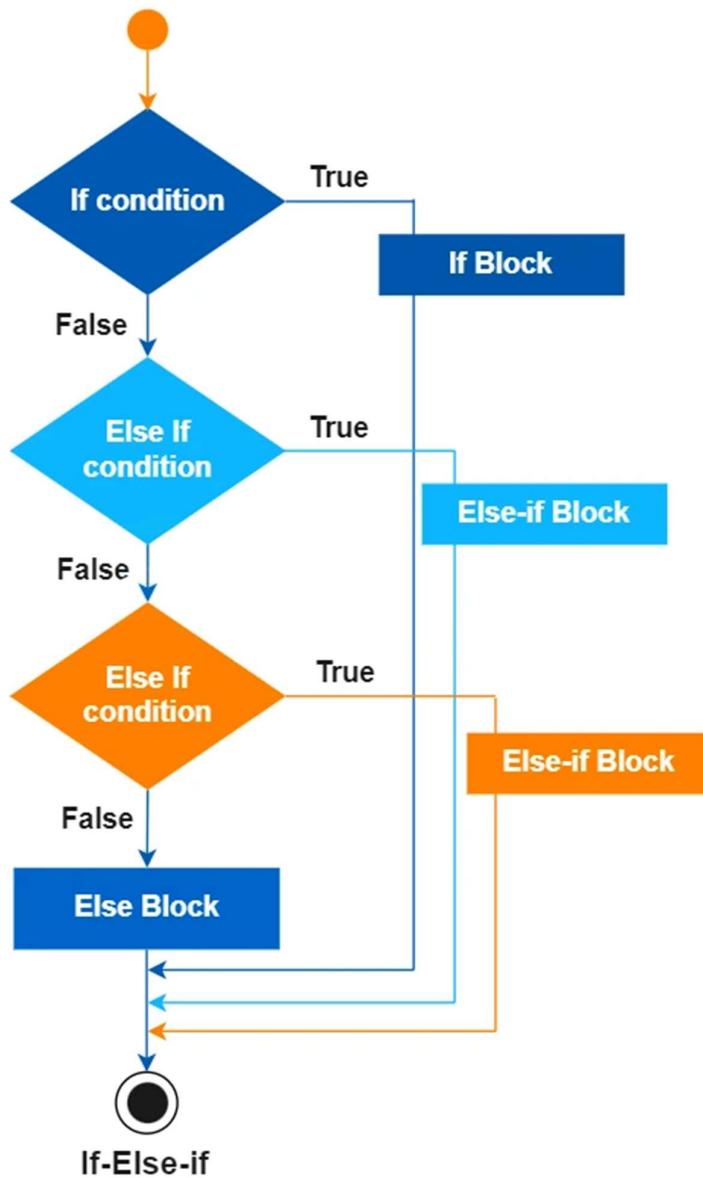
if-elseif kontrolna struktura

Na kraju, možete dodati i ključnu riječ `elseif` nakon čega slijedi drugi uslov i blok koda.

Nakon `if`-a možete dodati `elseif` uslove koliko god puta želite. Ako dodate `else`, on mora biti posljednji u lancu uslova.

Također imajte na umu da će, čim PHP nađe uslov koji je istinit (true), prestati provjeravati ostale uslove.

```
1  <?php
2  if (4 > 5) {
3      echo "Not printed";
4  } elseif (4 > 4) {
5      echo "Not printed";
6  } elseif (4 == 4) {
7      echo "Printed.";
8  } elseif (4 > 2) {
9      echo "Not evaluated.";
10 } else {
11     echo "Not evaluated.";
12 }
13 if (4 == 4) {
14     echo "Printed";
15 }
```



```
$studenti =[  
    "Ana" => 95,  
    "Ivan" => 85,  
    "Petar" => 75,  
    "Maja" => 65,  
    "Jasna" => 55,  
    "Marko" => 45,  
    "Iva" => 35,  
    "Luka" => 25,  
    "Klara" => 15,  
    "Filip" => 5  
];
```

```
foreach ($studenti as $ime => $bodovi) {  
    echo "Student/ica $ime je dobio ocjenu ";  
    if ($bodovi > 92) {  
        echo "odličan";  
    } elseif($bodovi > 75) {  
        echo "vrlo dobar";  
    } elseif($bodovi > 62) {  
        echo "dobar";  
    } elseif($bodovi > 50) {  
        echo "dovoljan";  
    } else {  
        echo "nedovoljan";  
    }  
    echo "<br>";  
}
```

switch-case kontrolna struktura

Još jedna kontrolna struktura slična `if-else` je `switch-case`.

Ova struktura provjerava samo jedan izraz i izvršava blok ovisno o njegovoj vrijednosti.

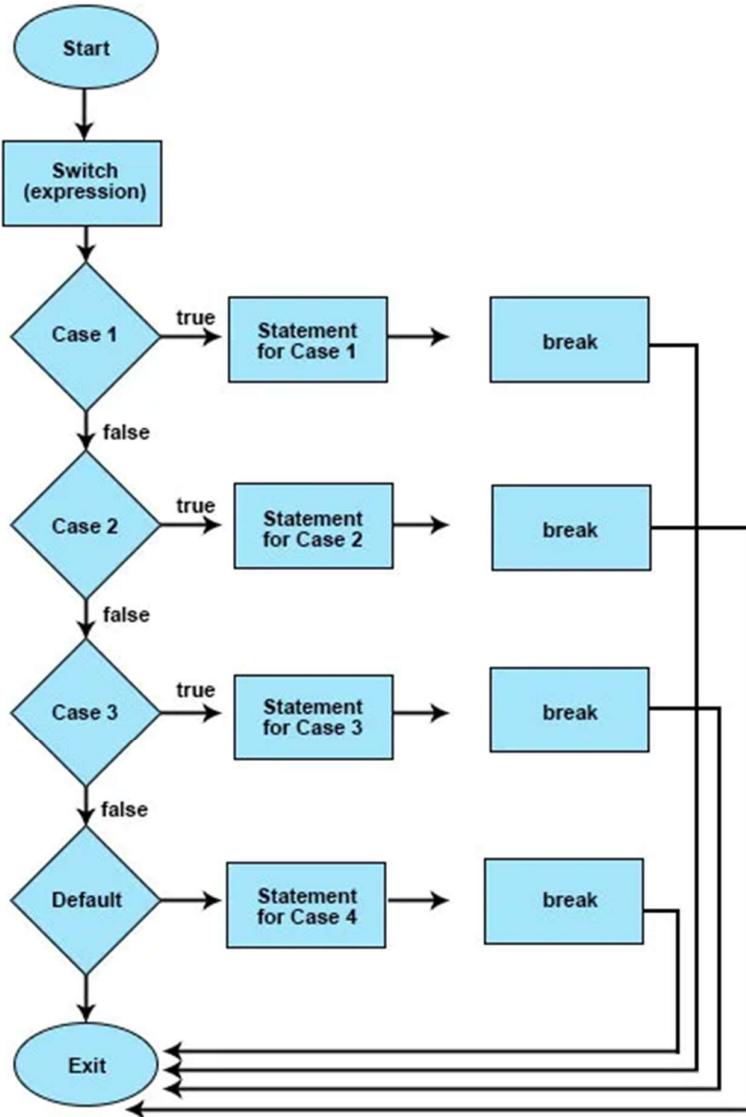
`switch` uzima izraz, u većini slučaja varijablu, a zatim definira niz slučajeva.

Kada slučaj odgovara trenutnoj vrijednosti izraza, izvršava se kod unutar njega.

Čim PHP pronađe izjavu o prekidu (`break`), izlazi iz kontrolne strukture.

U slučaju da nijedan slučaj nije jednak izazu, PHP izvršava zadalu postavku (`default`), ako ona postoji, ali to nije obavezno definirati.

```
1  <?php  
2  $title = 'Twilight';  
3  switch ($title) {  
4      case 'Harry Potter':  
5          echo "Nice story, a bit too long.";  
6          break;  
7      case 'Lord of the Rings':  
8          echo "A classic!";  
9          break;  
10     default:  
11         echo "Dunno that one.";  
12         break;  
13     }  
14  ?>
```



Vježba 1

- Definirajte varijable `a`, `b` i `c`, te im istim redoslijedom dodijelite vrijednosti 5,10 i 15.
- Koristeći uvjetovani tip kontrolne strukture provjerite je li vrijednost `b` između `a` i `c`.
- Ako je uslov istinit, ispišite da je `b` između `a` i `c`, a ako je uslov lažan ispišite da nije.
- Kod mora raditi i ako zamjenimo vrijednosti u varijablama `a` i `c`.

<?php

// Zadatak 1

```
$a = 20;
$b = 20;
```

```
$c = 10;  
// rezultat true/false  
//(ako je b između a i c) = true  
//(ako b nije između a i c) = false  
  
// (($b > $a i $b < $c) ili ($b < $a && $b > $c));  
var_dump(($b > $a && $b < $c) || ($b < $a && $b > $c));
```

Vježba 2

Koristeći uvjetovani tip kontrolne strukture switch ispišite koji je trenutno dan u tjednu.

Za ispravno izvršenu vježbu koristite PHP funkciju [date\(\)](#). Nazivi dana moraju biti na hrvatskom jeziku.

```
$danUTjednu = date("N");  
  
switch($danUTjednu) {  
    case 1:  
        echo "Danas je ponedjeljak";  
        break;  
    case 2:  
        echo "Danas je utorak";  
        break;  
    case 3:  
        echo "Danas je srijeda";  
        break;  
    case 4:  
        echo "Danas je četvrtak";  
        break;  
    case 5:  
        echo "Danas je petak";  
        break;  
    case 6:  
        echo "Danas je subota";  
        break;  
    case 7:  
        echo "Danas je nedjelja";  
        break;  
    default:  
        echo "Nepoznat dan";  
        break;
```

Kontrolne strukture - petlje

Petlje su kontrolne strukture koje omogućavaju izvršavanje određene izjave nekoliko puta, tj. onoliko puta koliko vam je potrebno.

Možete ih koristiti u nekoliko različitih scenarija, ali najčešći je u interakciji s matricama.

Na primjer, zamislite da imate matricu s elementima, ali ne znate što je u njemu. Želite ispisati sve njegove elemente pa pomoću petlje iterirate kroz matricu.

Postoje četiri vrste petlje. Svaka od njih ima svoje načine upotrebe, ali općenito, jednu vrstu petlje možete transformirati u drugu.

while petlje

`while` je najjednostavnija petlja. Izvodi blok koda sve dok izraz u uslovu ne poprimi vrijednost `false`.

U ovom primjeru prvo definiramo varijablu `$i` te joj dodijelimo vrijednost `1`.

Zatim imamo uslov u petlji `$i < 4`. Petlja će izvršavati blok koda dok je uvjet istinit (`true`).

Kao što vidite, unutar petlje svaki put povećavamo vrijednost `$i` za `1`, tako da petlja završava nakon `4` ponavljanja. Kada vrijednost `$i` dosegne `4`, uvjet postaje lažan (`false`), te se time petlja završava.

```
<?php  
$i = 1;  
  
while ($i < 4) {  
    echo "Ovo je iteracija broj: $i\n";  
    $i++;  
}  
?>
```

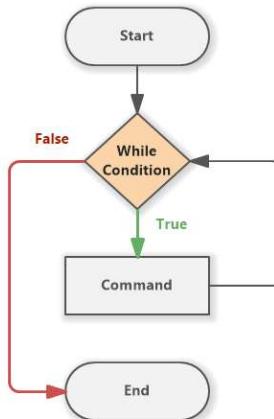
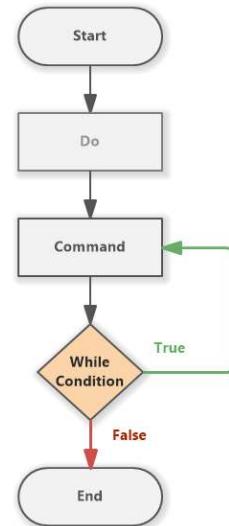
Rezultat je:

```
Ovo je iteracija broj: 1  
Ovo je iteracija broj: 2  
Ovo je iteracija broj: 3
```

do-while petlje

Petlja `do-while` je vrlo slična petlji `while`.

Jedina je razlika što će se blok koda barem jednom izvršiti kada je uvjet lažan (`false`).

WHILE**DO-WHILE**

U primjeru su definirane dvije petlje s istim izrazom i blokom koda, ali ako ih izvršite, vidjet ćete da je samo kod unutar `do-while` izvršen.

U oba slučaja izraz je od početka lažan (`false`), pa `while` ni ne ulazi u petlju, dok `do-while` ulazi u petlju jednom.

```

<?php
echo "sa while: ";
$i = 1;
while ($i < 0) {
    echo $i . " ";
    $i++;
}

echo "sa do-while: ";
$i = 1;
do {
    echo $i . " ";
    $i++;
} while ($i < 0)
?>
  
```

Rezultat je:

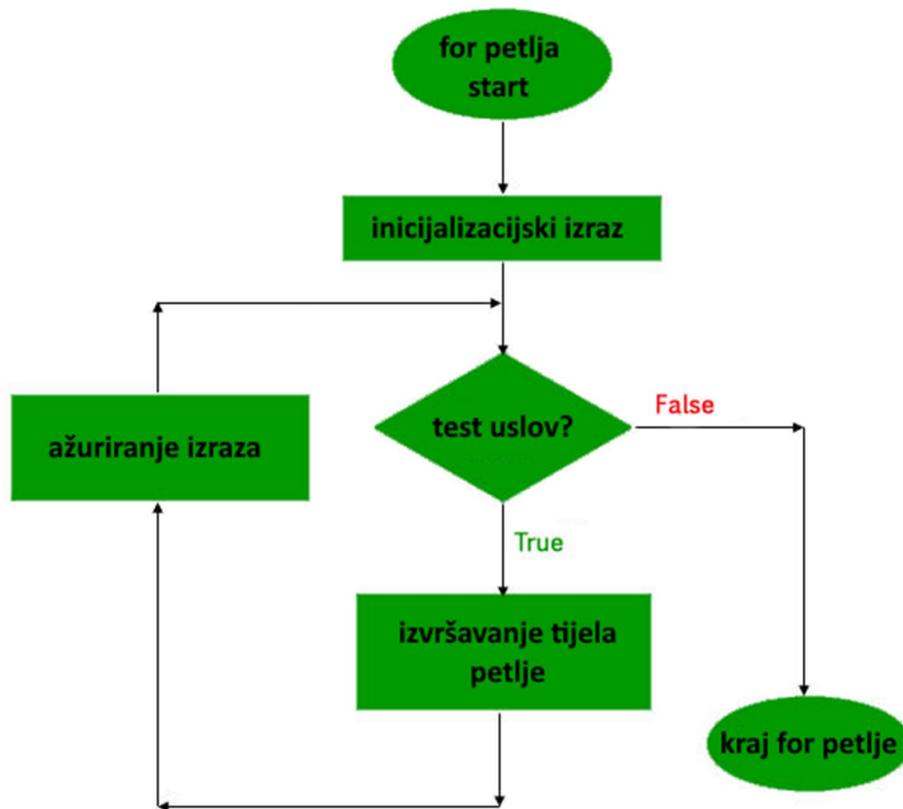
`sa while: sa do-while: 1`

for petlje

Petlja for je najkompleksnija od sve četiri petlje.

Definira inicijalni izraz, stanje izlaza i kraj iteracijskog izraza.

Kad PHP prvi put dolazi do petlje, izvršava ono što je definirano kao inicijalni izraz. Zatim ispituje uvjet i, ako je rezultat istinit (true), ulazi u petlju.



```
<?php
for ($i = 1; $i < 10; $i++) {
    echo $i . " ";
}
?>
```

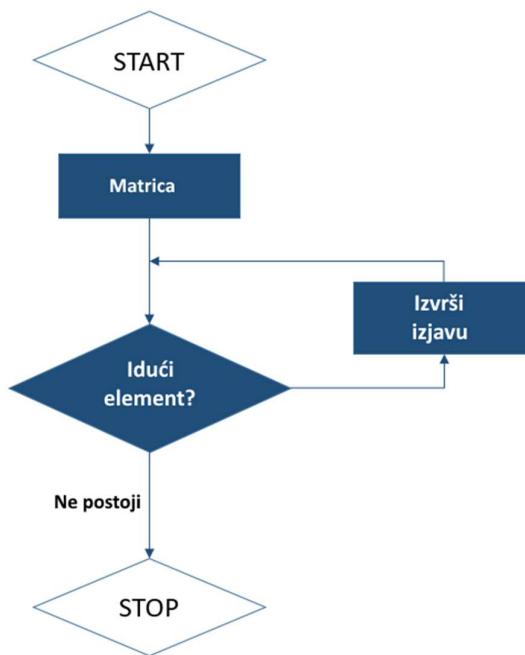
Ovo ispiše brojeve od 1 do 9.

```
for($i = 1; $i <= 10; $i++) { // treba paziti da ne otvorimo beskonačnu petlju
    if($i ===3){
        continue; // ako je $i = 3 preskočit ćemo petlju
    }
    echo $i . "<br>";
    // break; // ako ovo stavimo nakon prvog puta zaustavit će se petlja
}
```

foreach petlje

Posljednja, ali ne najmanje bitna, vrsta petlje je **`foreach`**. Ova petlja se koristi za matrice i omogućuje iteriranje matrice u cijelosti, čak i ako ne znate njegove ključeve. Postoje dvije mogućnosti za sintaksu, kao što možete vidjeti u sljedećim primjerima.

Petlja **`foreach`** prihvata matrice - u ovom slučaju **`$names`** - i određuje varijablu koja će sadržavati vrijednost unosa matrice. Možete vidjeti da ne trebamo specificirati nijedan krajnji uvjet, jer će PHP po broju elemenata u matrici znati koliko puta treba iterirati. Po želji možete odrediti varijablu koja sadrži ključ svake iteracije, kao u drugoj petlji.



```

<?php
$names = ['Harry', 'Ron', 'Hermione'];
foreach ($names as $name) {
    echo $name . " ";
}
foreach ($names as $key => $name) {
    echo $key . " ->" . $name . " ";
}
?>
  
```

Rezultat je:

Harry Ron Hermione 0 ->Harry 1 ->Ron 2 ->Hermione

```
<?php
// asocijativna matrica s imenovanim ključevima
$studenti =[

    "Ana" => 95,
    "Ivan" => 85,
    "Petar" => 75,
    "Maja" => 65,
    "Jasna" => 55,
    "Marko" => 45,
    "Iva" => 35,
    "Luka" => 25,
    "Klara" => 15,
    "Filip" => 5

];

// foreach petlja $ime je ključ a $bodovi je vrijednost
foreach ($studenti as $ime => $bodovi) {
    echo "Student/ica $ime je dobio ocjenu ";
    if ($bodovi > 92) {
        echo "odličan";
    } elseif($bodovi > 75) {
        echo "vrlo dobar";
    } elseif($bodovi > 62) {
        echo "dobar";
    } elseif($bodovi > 50) {
        echo "dovoljan";
    } else {
        echo "nedovoljan";
    }
    echo "<br>";
}

// Napišite PHP skriptu koja provjerava koji je dan u tjednu i ispisuje
odgovarajuću poruku.

$danUTjednu = date("N");

switch($danUTjednu) {
    case 1:
        echo "Danas je ponedjeljak";
        break;
    case 2:
        echo "Danas je utorak";
        break;
```

```
case 3:  
    echo "Danas je srijeda";  
    break;  
case 4:  
    echo "Danas je četvrtak";  
    break;  
case 5:  
    echo "Danas je petak";  
    break;  
case 6:  
    echo "Danas je subota";  
    break;  
case 7:  
    echo "Danas je nedjelja";  
    break;  
default:  
    echo "Nepoznat dan";  
    break;  
}  
  
for($i = 1; $i <= 10; $i++) { // treba paziti da ne otvorimo beskonačnu petlju  
if($i ===3){  
    continue; // ako je $i = 3 preskočit ćemo petlju  
}
```

Vježba 1

- Koristeći petlju **while**, ispišite prvih deset brojeva.
- Koristeći petlju **for**, ispišite sve parne brojeve do 100.

Vježba 2

- Definirajte varijablu **names** i dodijelite joj niz koji sadrži pet imena.
- Koristeći petlju **foreach**, iz niza ispišite ključeve i pripadajuće im vrijednosti.

Algoritmi za sortiranje

PHP koristi Quicksort. Moguće je vidjeti kompleksnost na

<https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/>

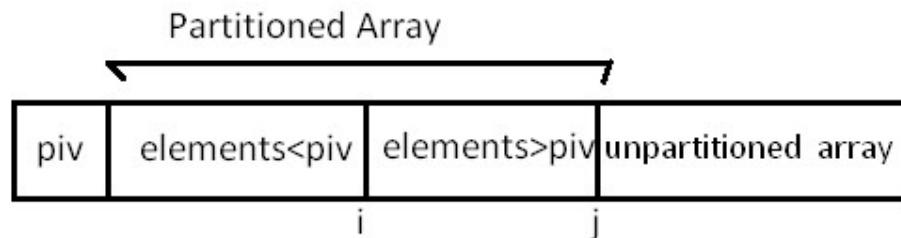
Quick sort temelji se na pristupu podijeli i svladaj koja se temelji na ideji odabira jednog elementa kao središnji (engl. pivot) elementa i dijeljenja matrice oko njega tako da: Ljeva strana središnjeg (engl.

pivot) elementa sadrži sve elemente koji su manji od pivot elementa a Desna strana sadrži sve elemente veće od pivota.

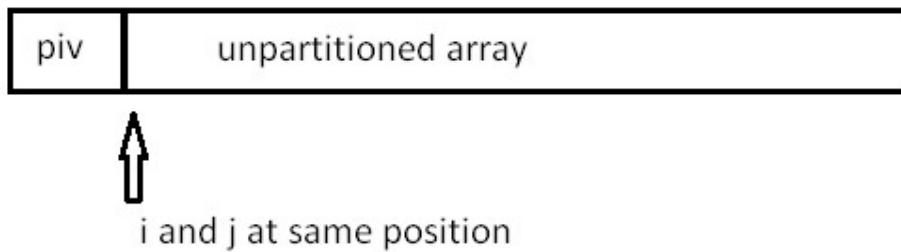
Smanjuje zahtjeve za prostorom i uklanja upotrebu pomoćne matrice koji se koristi za merge sort. Odabir slučajnog pivota u matrici u većini slučajeva rezultira poboljšanom potrošnjom vremena.

Implementacija :

Odaberite prvi element matrice kao pivot element. Prvo ćemo vidjeti kako se particija (podijeljeni dio) matrice odvija oko pivota.



Initially :



U donjoj implementaciji korištene su sljedeće komponente: Ovdje,

`A[]` = matrica čiji elementi se sortiraju

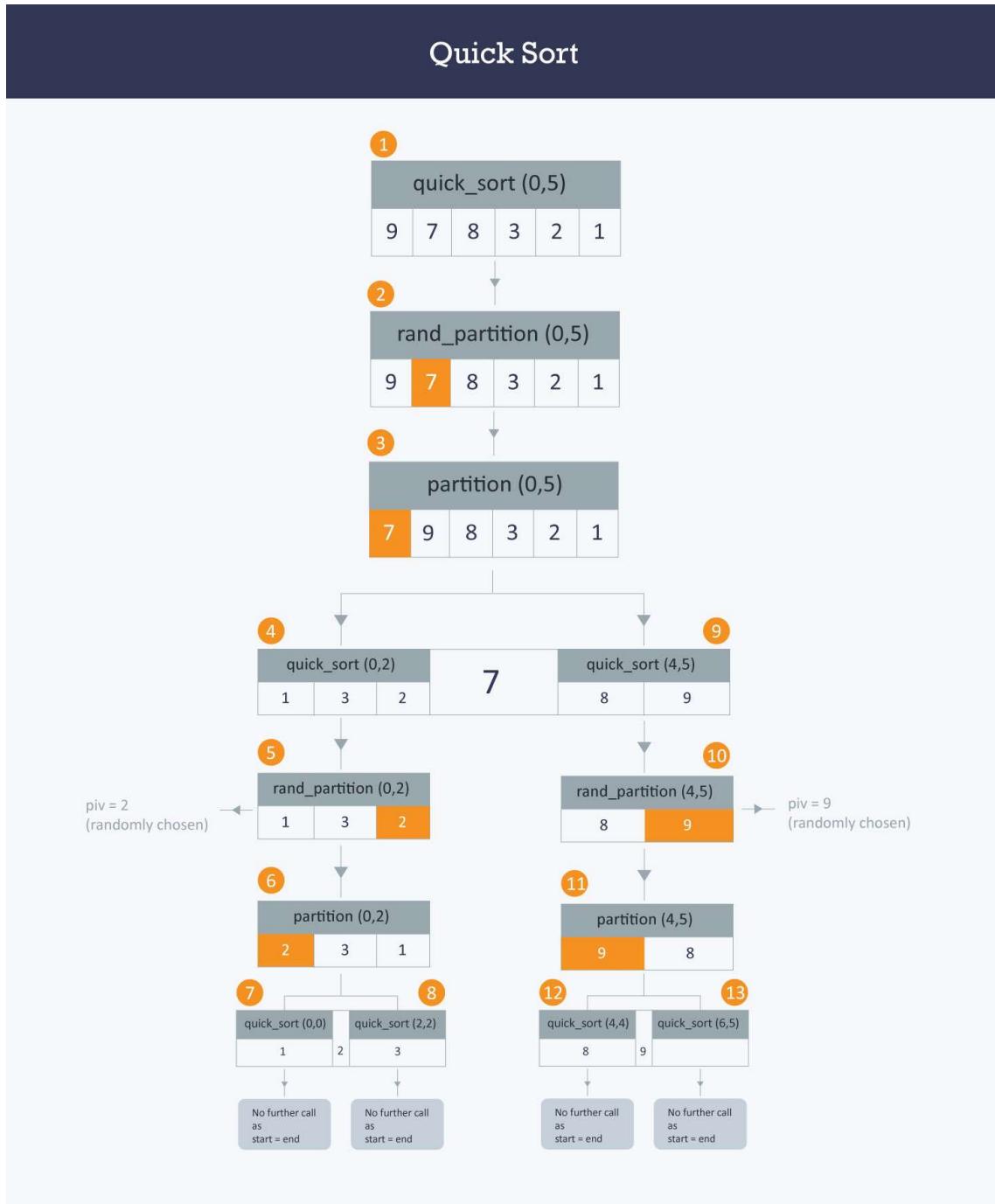
`start`: Krajnja lijeva pozicija matrice

`end`: Krajnji desni položaj matrice

`i`: Granica između elemenata koji su manji od pivota i onih koji su veći od pivota

`j`: Granica između particioniranog i neparticioniranog dijela niza

`piv`: Pivot element



Ovdje nalazimo pravilan položaj pivot elementa preuređivanjem matrice pomoću partijske funkcije. Zatim dijelimo niz na dvije polovice lijevu stranu stožera (elementi manji od pivot elementa) i desnu stranu pivota (elementi veći od pivot elementa) i primjenjujemo isti korak rekursivno.

Primjer: imate matricu `A = [9, 7, 8, 3, 2, 1]`. Uočite na donjem dijagramu da funkcija `randpartition()` odabire pivot slučajno kao `7` i zatim ga mijenja s prvim elementom matrice, a zatim se poziva `partition()` funkcija, koja dijeli matricu na dvije polovice. Prva polovica ima elemente manje od `7`, a druga polovica ima elemente veće od `7`.

Za elemente manje od 7, 5 u pozivu, funkcija `randpartition()` slučajno odabire pivot element, a zatim ga mijenja s prvim elementom i poziva se funkcija `partition()`. Nakon 7. i 8. poziva, ne može se više pozivati jer je u oba poziva ostao samo jedan element. Slično, možete promatrati redoslijed poziva za elemente veće od 7.

```
<?php

function partition(array &$A, int $start, int $end): int {
    $i = $start + 1;
    $piv = $A[$start]; // Prvi element se uzima kao pivot

    for ($j = $start + 1; $j <= $end; $j++) {
        // Reorganiziraj niz tako da elementi manji od pivota budu na jednoj strani
        // a oni veći na drugoj.
        if ($A[$j] < $piv) {
            // Zamjeni elemente na poziciji $i i $j
            $temp = $A[$i];
            $A[$i] = $A[$j];
            $A[$j] = $temp;

            $i += 1;
        }
    }
    // Stavi pivot na njegovo odgovarajuće mjesto
    $temp = $A[$start];
    $A[$start] = $A[$i - 1];
    $A[$i - 1] = $temp;

    return $i - 1; // Vrati poziciju pivota
}

function quick_sort(array &$A, int $start, int $end): void {
    if ($start < $end) {
        // Pozicija pivot elementa
        $piv_pos = partition($A, $start, $end);

        // Sortiraj lijevu stranu pivota
        quick_sort($A, $start, $piv_pos - 1);

        // Sortiraj desnu stranu pivota
        quick_sort($A, $piv_pos + 1, $end);
    }
}
```

```
// Primjer upotrebe:  
$A = [9, 7, 8, 3, 2, 1];  
quick_sort($A, 0, count($A) - 1);  
  
print_r($A);  
?>
```

Quick Sort koristi rekurziju kako bi kontinuirano dijelio niz na manje dijelove i sortirao ih. Ova podjela se nastavlja sve dok svaki podniz ne bude trivijalan za sortiranje (tj. dužine 0 ili 1).

Parametri funkcije su: `array &$A` referenca na niz koji se sortira, `int $start` indeks prvog elementa trenutnog podmatrice koja se sortira, `int $end` indeks posljednjeg elementa trenutnog podmatrice koji se sortira. `array &$A` je referenca na matricu koja se sortira. Referenca `&` omogućava funkciji da se direktno mijenja originalna matrica a ne njegova kopija.

`$i = $start + 1;` ovdje indeks `$i` pokazuje na prvo mjesto nakon početnog indeksa `$start`.

`$piv = $A[$start];` znači da se prvi element podmatrice uzima se kao pivot. Pivot je ključni element oko kojeg će se niz reorganizirati.

`for ($j = $start + 1; $j <= $end; $j++)` je petlja prolazi kroz sve elemente podmatrice, počevši od elementa odmah nakon pivota. `if ($A[$j] < $piv)` znači da ako je trenutni element manji od pivota, zamjenjuje se s elementom na poziciji `$i`. Nakon provjere, element na poziciji `$i` zamjenjuje se s elementom na poziciji `$j`, a `$i` se povećava za `1`. Ovaj korak premješta sve manje elemente na lijevu stranu matrice

Nakon završetka petlje, pivot se postavlja na poziciju `$i - 1`. Na taj način, pivot je na ispravnoj poziciji, s manjim elementima lijevo od njega i većim desno. S `return $i - 1;` funkcija vraća indeks na kojem se sada nalazi pivot.

Funkcija `quick_sort` ima parametre funkcije `array &$A` je referenca na niz koji se sortira, `int $start` je indeks početka trenutnog podmatrice koji se sortira, `int $end` je indeks kraja trenutne podmatrice koji se sortira.

`if ($start < $end)` provjera osigurava da se rekursija zaustavi kada podniz postane toliko mali (ili prazan) da nije potrebno daljnje sortiranje. Ako je `$start` jednak ili veći od `$end`, funkcija se vraća bez daljnog djelovanja.

`partition($A, $start, $end)`: Poziva se funkcija `partition` koja vraća indeks pivota nakon što je matrica reorganizirana. `quick_sort($A, $start, $piv_pos - 1);` rekursivno sortira lijevu stranu pivota. `quick_sort($A, $piv_pos + 1, $end);` rekursivno sortira desnu stranu pivota.

Inicijaliziramo matricu `$A`, pozivamo `quick_sort` funkciju i ispisujemo sortiranu matricu.

Evo i druge verzije sa `rand_partition` umjesto `partition`.

```
<?php  
function partition(array &$A, int $start, int $end): int {  
    $i = $start + 1;  
    $piv = $A[$start]; // Prvi element se uzima kao pivot  
  
    for ($j = $start + 1; $j <= $end; $j++) {
```

```

    // Reorganiziraj niz tako da elementi manji od pivota budu na jednoj strani
    // a oni veći na drugoj.
    if ($A[$j] < $piv) {
        // Zamjeni elemente na poziciji $i i $j
        $temp = $A[$i];
        $A[$i] = $A[$j];
        $A[$j] = $temp;

        $i += 1;
    }
}

// Stavi pivot na njegovo odgovarajuće mjesto
$temp = $A[$start];
$A[$start] = $A[$i - 1];
$A[$i - 1] = $temp;

return $i - 1; // Vrati poziciju pivota
}

function rand_partition(array &$A, int $start, int $end): int {
    // Odabiremo slučajnu poziciju pivota između start i end
    $random = $start + rand(0, $end - $start);

    // Zamjenjujemo pivot s prvim elementom
    $temp = $A[$random];
    $A[$random] = $A[$start];
    $A[$start] = $temp;

    // Pozivamo standardnu partition funkciju
    return partition($A, $start, $end);
}

function quick_sort(array &$A, int $start, int $end): void {
    if ($start < $end) {
        // Pozicija pivot elementa
        // Koristimo rand_partition umjesto partition
        $piv_pos = rand_partition($A, $start, $end);

        // Sortiraj lijevu stranu pivota
        quick_sort($A, $start, $piv_pos - 1);

        // Sortiraj desnu stranu pivota
        quick_sort($A, $piv_pos + 1, $end);
    }
}

```

```
}

// Primjer upotrebe:
$A = [9, 7, 8, 3, 2, 1];
quick_sort($A, 0, count($A) - 1);

print_r($A);
?>
```

U funkciji `rand_partition`: koristimo `rand(0, $end - $start)` da odaberemo slučajnu poziciju unutar podmatrice. Zatim taj element zamjenimo s prvim elementom podmatrice (`$start`). Na taj način, slučajno odabrani element postaje pivot. Nakon što smo slučajno odabrali pivot, zamjenimo ga s prvim elementom matrice da bi funkcija `partition` mogla raditi s njim na isti način kao i ranije.

Nakon zamjene, poziva se standardna `partition` funkcija koja reorganizira elemente u odnosu na nasumično odabrani pivot.

U funkciji `quick_sort` smo zamjenili standardni poziv `partition` s `rand_partition`, čime omogućavamo da pivot bude slučajno odabran pri svakom pozivu particije. Ostatak funkcije ostaje isti.

PHP funkcije

Funkcija je blok koda za višekratnu upotrebu koja, uz unos, izvodi neke radnje i po želji vraća neki rezultat.

Već znate nekoliko unaprijed definiranih funkcija kao što su `empty`, `in_array` ili `var_dump`. Te funkcije dolaze s PHP-om, ali možete vrlo lako izraditi vlastite.

Funkcije možete upotrijebiti kada imate dijelove koda u aplikaciji koje je potrebno nekoliko puta izvršiti ili za definiranje neke funkcionalnosti.

PHP ne podržava preopterećene (engl. overload) funkcije. Preopterećenje se odnosi na sposobnost deklariranja dvije ili više funkcija s istim nazivom, ali različitim argumentima. Argumente možete deklarirati čak i ako ne znate koje tipove podataka prihvaćaju, tako da PHP ne bi mogao odlučiti koju će funkciju koristiti.

Deklariranje i pozivanje

Funkcija ima naziv, argumente (opcionalno) i sadrži blok koda.

Po želji, može definirati koju vrijednost treba vratiti.

Naziv funkcije mora slijediti ista pravila kao i naziv varijable, odnosno mora započeti slovom ili crtom za podvlačenje, a može sadržavati sva slova, brojeve ili crte za podvlačenje.

Naziv funkcije ne smije biti rezervirana riječ, kao što su imena unaprijed definiranih funkcija.

Pozivanje funkcije se radi tako da se napiše naziv funkcije i na kraju doda zagrade.

```

1  <?php
2  // deklariranje funkcije
3  function writeHello(){
4      return "Hello World";
5  }
6  // pozivanje funkcije
7  writeHello();
8
9  ?>

```

Ako funkcija ima argumente potrebno ih je navesti unutar zagrada `(tip $ime_parametra1, tip $ime_parametra2, ...)`. Iza dvotočke navodi se tip izlaza iz funkcije. Ako funkcija ništa ne vraća, treba navesti `void`. Iza toga idu vitičaste zgrade `{}` i unutar njih tijelo funkcije. U zadnjem redu funkcije ide naredba `return` i u zagradi `()` ono što vraćamo.

Vježba 1

Proizvoljno deklarirajte funkciju koja vraća neki tekst.

Pozovite funkciju i vraćenu vrijednost dodijelite varijabli.

```
Ispišite vrijednost varijable.
function vratiTekst(): string {      // string znači da očekuje vraćanje stringa
    return 'Ovo je neki tekst';
}

$tekst = vratiTekst();
echo $tekst;
```

PHP funkcije – argumenti

Funkcija dobiva podatke izvana putem argumenata. Možete definirati bilo koji broj argumenata, uključujući 0 (nijedan).

Argumenti trebaju imati naziv kako bi se mogli koristiti unutar funkcije; ne mogu postojati dva argumenta s istim nazivom.

Prilikom pozivanja funkcije, podatke argumentima morate poslati istim redoslijedom kao što je deklarirano.

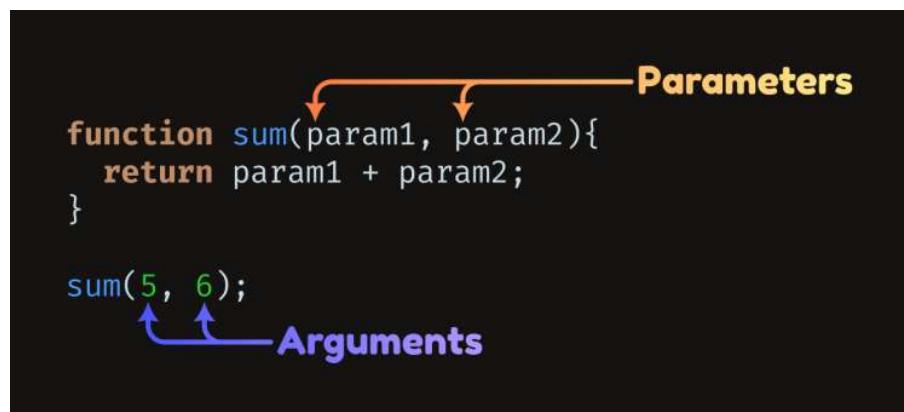
Funkcija može sadržavati neobavezne argumente, odnosno niste obavezni poslati vrijednost za te argumente. Kad deklarirate funkciju, trebate navesti zadane vrijednosti za te argumente. Dakle, u slučaju da korisnik ne daje vrijednost, funkcija će koristiti zadanu.

```

1  <?php
2  function addNumbers($a, $b, $printResult = false) {
3      $sum = $a + $b;
4      if ($printResult) {
5          echo 'The result is ' . $sum;
6      }
7      return $sum;
8  }
9
10 $sum1 = addNumbers(1, 2);
11 $sum1 = addNumbers(3, 4, false);
12 $sum1 = addNumbers(5, 6, true); // it will print the result
13
14 ?>

```

Do 8.0 verzije PHP-a, što se tiče parametra, oni su mogli biti samo pozicijski. Ako su bili definirani parametri param1 i param2 u funkciji, slanje argumenata moralo je ići po tim pozicijama.



Od verzije 8.0 moguće je koristiti imenovane parametre. Na temelju imenovana parametra, neovisno o redoslijedu moguće je poslati parametre. Ako koristite imenovane parametre, ne možete koristiti pozicijske.

```

sum(param1, param2);
sum(param2:3, param1:5);

```

Vježba 2

Proizvoljno deklarirajte funkciju koja ima dva argumenta (name i surname). Funkcija treba konkatenerati podatke iz argumenata tako da između postoji razmak i dodijeliti ih lokalnoj varijabli, zatim treba vrijednost u varijabli pretvoriti u velika slova i to vratiti kao rezultat.

- Pozovite funkciju i vraćenu vrijednost dodijelite varijabli.
- Ispišite vrijednost varijable.

```

function fullName(string $name, string $surname): string {
    $result = $name . ' ' . $surname;
    return mb_strtoupper($result);
}

$fullName = fullName(surname:'John', name:'Doe');

```

```
echo $fullName;
```

Parametre možemo pozvati pozicijski

```
$fullName = fullName('John', 'Doe');
```

ili imenovani kao iznad:

```
$fullName = fullName(surname: 'John', name: 'Doe');
```

Prednost imenovanih je da ne moramo paziti na redoslijed a i čitkiji su.

Strogo tipiziranje

Direktiva `declare(strict_types=1);` u PHP-u omogućava strogo tipiziranje, što znači da PHP striktno provjerava tipove podataka koji se koriste u funkcijama. Kada je ova direktiva uključena, PHP zahtijeva da argumenti funkcija i povratne vrijednosti budu točno onog tipa koji je naveden u deklaraciji funkcije. Ako nije, PHP će generirati fatalnu grešku.

```
declare(strict_types=1);
```

Bez `strict_types=1`:

Ako funkcija očekuje određeni tip podataka, a vi proslijedite podatak koji se može automatski pretvoriti (npr. proslijedite `int` gdje je očekivan `float`), PHP će napraviti implicitnu konverziju.

Sa `strict_types=1`:

Kada je `strict_types` uključen, PHP neće automatski konvertirati tipove podataka. Ako se proslijedi podatak pogrešnog tipa, PHP će generirati grešku.

Primjer bez `strict_types=1`:

```
function add(int $a, int $b): int {
    return $a + $b;
}

echo add(2, '3'); // Ispisat će 5, jer će PHP pretvoriti string '3' u integer 3
```

Primjer sa `strict_types=1`:

```
declare(strict_types=1);

function add(int $a, int $b): int {
    return $a + $b;
```

```
}
```

```
echo add(2, '3'); // Fatalna greška: Argument 2 mora biti tipa int, string  
prosljeden
```

Ova direktiva važi samo za datoteku u kojoj je deklarirana. Ako je uključite u jednoj datoteci, to neće automatski utjecati na druge datoteke.

Primjenjuje se samo na argumente funkcija i povratne vrijednosti. Ne utječe na ostale tipove podataka u programu.

Korištenje strogog tipiziranja tj. `strict_types=1`:

- **Povećava sigurnost koda:** Programer je siguran da funkcije primaju i vraćaju točno one tipove podataka koji su deklarirani.
- **Smanjuje mogućnost grešaka:** Programeri su prisiljeni biti pažljiviji s tipovima podataka, što može spriječiti suptilne i teške za otkriti bugove.
- **Pojačava eksplicitnost koda:** Kod postaje jasniji, jer se tipovi podataka moraju striktno pridržavati onoga što je deklarirano.

Zadatak

Napiši funkciju koja vraća neki tekst. Pozovite funkciju i vraćenu vrijednost spremite u varijablu. Ispišite vrijednost varijable.

```
function vratiTekst(): string {    // string znači da očekuje vraćanje stringa  
    return 'Ovo je neki tekst';  
  
}  
  
$tekst = vratiTekst();  
echo $tekst;
```

Doseg (engl. scope) varijabli

Po dosegu varijable dijelimo na:

- **lokalne varijable** - vidljive samo unutar funkcije
- **globalne varijable** - varijable definirane izvan funkcija, vidljive su unutar cijele datoteke i svih uključenih datoteka.

Varijable definirane unutar funkcija ili klase imaju lokalni doseg. To znači da im možemo pristupiti samo unutar njihove definicije, osim ako nisu vraćene kao povratna vrijednost funkcije ili ako su definirane kao statičke varijable.

Varijable definirane izvan funkcija ili klase imaju globalni doseg. To znači da im možemo pristupiti iz bilo kojeg dijela programa. Međutim, direktno pristupanje globalnim varijablama unutar funkcija nije moguće. Umjesto toga, moramo koristiti ključnu riječ `global` ili superglobalnu varijablu `$GLOBALS`.

Ovakva naredba unutar funkcije uzrokuje potencijalne probleme:

```
$global_var = "Ovo je globalna varijabla";

function printGlobalVar() {
    global $global_var;
    echo $global_var;
}

printGlobalVar(); // Ispisuje "Ovo je globalna varijabla"
```

Jer varijabla koja je imala drugu vrijednost van funkcije a u funkciji smo je promijeni, mijenja vrijednost u cijelom kodu.

Pogledajmo npr. `array_filter` funkciju:

```
$even = array_filter([1, 2, 3, 4, 5], function($value) {
    return $value % 2 === 0;
});
```

Napisat ćemo vlastitu funkciju:

```
// Custom array_filter funkcija bez korištenja callback funkcije
function customArrayFilter(array $array): array {
    $result = [];
    foreach ($array as $value) {
        if (($value % 2 === 0) {
            $result[] = $value;
        }
    }
    return $result;
}
```

```
// Custom array_filter funkcija sa korištenjem callback funkcije
function customArrayFilter(array $array, callable $callback): array {
    $result = [];
    foreach ($array as $value) {
        if ($callback($value)) {
            $result[] = $value;
        }
    }
    return $result;
}
```

`array $array` je ulazna matrica koju želimo filtrirati. Prvi `array` specificira tip argumenta koji funkcija očekuje a `$array` je naziv varijable unutar funkcije koja će primiti ulaznu matricu, odlučili smo je tako nazvati, mogli smo bilo kako. `callable $callback` je funkcija (callback) koja će se pozivati za svaki element matrice. Callback funkcija vraća `true` ili `false`, ovisno o tome da li element treba biti uključen u rezultat. Funkcija `customArrayFilter` vraća novu matricu koja sadrži samo one elemente iz originalne matrice koji su prošli filter (tj. za koje je callback funkcija vratila `true`). Upravo to smo i htjeli. `$result = [];` kreira (inicijalizira) praznu matricu koja će sadržavati filtrirane elemente. Petlja `foreach ($array as $value)` prolazi kroz svaki element matrice `$array`. `if ($callback($value))` poziva callback funkciju s trenutnim elementom `$value` kao argumentom. Ako callback funkcija vrati `true`, to znači da element zadovoljava uvjete i treba biti uključen u rezultirajuću matricu. `$result[] = $value;` dodaje trenutni element `$value` u matricu `$result` ako je callback funkcija vratila `true`. Nakon što su svi elementi prošli kroz filter, funkcija vraća rezultirajuću matricu `$result`, koja sadrži sve elemente koji zadovoljavaju uslov definiran u callback funkciji.

`customArrayFilter` koristimo na sljedeći način:

```
// Definicija matrice
$numbers = [1, 2, 3, 4, 5];

// Callback funkcija koja filtrira parne brojeve
$evenNumbers = customArrayFilter($numbers, function($number) {
    return $number % 2 === 0;
});

// Ispis rezultata
print_r($evenNumbers); // Očekivani ishod: [2, 4]
```

Superglobalna varijabla \$GLOBALS

Bolje je koristiti ako već moramo **superglobalne varijable**:

```
$tekst = 'Ovo je neki tekst';
echo $GLOBALS['tekst']; // ovime $tekst postaje globalna varijabla od svuda dostupna
```

Ne možemo pregaziti tu `$GLOBALS` varijablu ali možemo promijeniti sadržaj:

```
$globals['age'] = "null"
```

Ali možemo ukloniti sa `unset` element `$GLOBALS` matrice.

```
unset($GLOBALS['tekst']);
```

Također možemo ispisati sve `$GLOBALS` varijable koje su u matrici:

```
print_r($GLOBALS);
```

```
1  <?php
2  // Globalna varijabla
3  $a = 'Algebra';
4
5  function variableScope() {
6      // Lokalna varijabla
7      $a = 'Backend developer';
8      // Ispis lokalne varijable
9      echo $a;
10 }
11 // Ispis Globalne varijable
12 echo $a;
13
14 ?>
```

Garbage collectori brinu o tome da se lokalne varijable obrišu. Garbage Collector je mehanizam koji automatski pronađi i uklanja objekte u memoriji koji više nisu potrebni programu. To pomaže u sprječavanju "memory leakova" - situacija u kojima program neprestano zauzima sve više memorije, što može dovesti do usporavanja ili čak rušenja aplikacije.

`$GLOBALS` referencira sve varijable dostupne u globalnom opsegu.

Asocijativna matrica koji sadrži reference na sve varijable koje su trenutno definirane u globalnom opsegu skripte. Imena varijabli su ključevi matrica.

```
<?php
$a = 1;
$GLOBALS = $GLOBALS;
$GLOBALS['a'] = 2;

echo $a; // 1
echo $GLOBALS['a']; // 2
?>
```

```
<?php
function test() {
    // foo je file ili object a koristi se kao ime varijable
    $foo = "lokalna varijabla";

    echo '$foo u globalnom dosegu: ' . $GLOBALS["foo"] . "\n";
    echo '$foo u trenutnom dosegu: ' . $foo . "\n";
}
```

```
$foo = "Primjer sadržaja";
test();
?>
```

Varijabla static

Druga važno svojstvo doseg varijabli je staticka varijabla. Statička varijabla postoji samo u lokalnom dosegu funkcija, ali ne gubi vrijednost kada izvršenje programa napusti funkciju.

Varijabla `$a` se inicijalizira samo u prvom pozivu funkcije i svaki put kada se pozove funkcija, ispisat će vrijednost `$a` i povećavati je.

```
<?php
function incrementCounter() {
    static $counter = 0;
    $counter++;
    echo $counter;
}

incrementCounter(); // Ispisuje "1"
incrementCounter(); // Ispisuje "2"
incrementCounter(); // Ispisuje "3"
```

PHP funkcije – kao varijable

PHP podržava koncept funkcija kao varijable. To znači da će, ako je vrijednost varijable jednaka nazivu funkcije, PHP tražiti funkciju s istim nazivom i pokušati je izvršiti.

Između ostalog, ovo se može koristiti za implementaciju povratnih poziva.

Ovaj način pozivanja funkcija neće raditi s ugrađenim PHP funkcijama kao što su: `echo`, `print`, `unset`, `isset`, `empty`, `include`, `require` i slične.

```
<?php
// Ovo je omotač (engl. wrapper) funkcija oko echo naredbe
function pozdrav($ime) {
    echo "Pozdrav, $ime!";
}

$func = 'pozdrav';
$func('Marko'); // // Ovo poziva pozdrav()
?>
```

Kada pozovemo varijablu `$func` s argumentima, PHP će pokušati pronaći i izvršiti funkciju s istim nazivom kao vrijednost varijable, u ovom slučaju `pozdrav()`.

Evo još primjera:

```
<?php
$print_function = 'print_r';
$print_function('Hello, world!'); // Ispisuje "Hello, world!"

$print_function = 'printf';
$print_function('Goodbye, world!'); // Ispisuje "Goodbye, world!"
?>
```

Anonimne funkcije (Closures)

Funkcije možete dodijeliti varijablama i kao anonimne funkcije (closures):

```
<?php
$square = function($x) {
    return $x * $x;
};

$result = $square(5); // $result će biti 25
?>
```

U ovom primjeru, anonimna funkcija je dodijeljena varijabli `$square`. Zatim pozivamo varijablu s argumentom `5` i dobivamo rezultat `25`.

Vježba 3

Proizvoljno deklarirajte funkciju koja ima jedan argument (number). Funkcija treba imati lokalnu varijablu u koju će pribrojiti vrijednost argumenta number te istu vratiti kao rezultat. Vrijednost treba biti zadržana kod svakog poziva funkcije.

Deklarirajte funkciju kao varijablu.

Pozovite funkciju pomoću varijable, a kao vrijednost argumenta pošaljite slučajan broj u rasponu od 1 do 10 te ispišite rezultat.

Ponovite postupak pet puta.

Callback funkcije

Ova funkcija, '`math`', uzima dva integera i poziva callback funkciju koja nešto radi.

callable ovdje znači „može se moći pozvati”, ako se pošalje nešto što nije `callable` funkcija će pući.

```
function math(int $a, int $b, callable $callback): int {  
    return $callback($a, $b);
```

`callable $callback` parametar očekuje funkciju koja će biti proslijeđena kao argument. Ta funkcija će se pozvati unutar `math` funkcije

Ako pozovemo tu funkciju `math` sa argumentima 5 i 10, to nije dovoljno jer joj moramo poslati funkciju koja nešto napravi. Ovdje je definirana **anonimna funkcija**. Npr. ovako:

```
math(5, 10, function($a, $b) {  
    return $a + $b;  
});
```

Dakle kada pozovemo funkciju `math` pošaljemo argumente 5 i 10 a treći argument dođe iz anonimne funkcije. Ova funkcija je definirana unutar samog poziva `math`. Kada se funkcija `math` pozove, ona proslijeđuje vrijednosti 5 i 10 anonimnoj funkciji koja vraća 15. Ovaj rezultat se pohranjuje u varijablu `$var` i zatim ispisuje pomoću `echo`.

Umjesto anonimne funkcije mogli smo definirati funkciju `add` koju možemo koristiti kada god želimo zbrajanje:

```
function add($a, $b) {  
    // Zbroji dva integera i vrati rezultat  
    return $a + $b;
```

Možemo je koristiti samostalno:

```
add(5, 10);
```

ali je možemo koristiti i kao callback funkciju koju šaljemo `math` funkciji:

```
echo math(5, 10, 'add');
```

Mora biti u navodnicima jer PHP može dodjeljivati funkcije varijabli na način da ih definira kao string.

```
$zbroji = 'add';
```

U varijablu `$zbroji` je pohranjen naziv funkcije `add`. Varijabla sada sadrži string `add`.

```
echo $zbroji;
```

`echo $zbroji;` ispisuje `add`.

```
echo $zbroji(5, 10);
```

Ovo poziva funkciju s tim imenom jer ona postoji, to je referenca na funkciju `add` i PHP će je pozvati kao da smo napravili `add(5, 10)`. Rezultat je 15!

Evo cijelog koda:

```
function math(int $a, int $b, callable $callback): int {  
    // Pozovi callback funkciju i vrati njen rezultat
```

```
    return $callback($a, $b);
}

// Pozovi 'math', proslijedi 5 i 10 kao argumente
// Proslijedi funkciji 'add' kao callback argument

// pozivanje math s anonimnom funkcijom kao callback-om
$var = math(5, 10, function($a, $b) {
    return $a + $b;
});
echo $var;

// Ovo je callback funkcija koju 'math' poziva
function add($a, $b) {
    // Zbroji dva integera i vrati rezultat
    return $a + $b;
}

echo add(5,10);
echo math(5,10,'add');

// Kreiraj varijablu koja sadrži 'add' ime funkcije
$zbroji = 'add';
echo $zbroji;

// Pozovi 'add' funkciju pomoću varijable koja sadrži njen naziv
// Ovo je isto kao pozivanje 'add(5, 10)'
echo $zbroji(5, 10);
```

Superglobalne varijable

Superglobals - ugrađene varijable koje su uvijek dostupne u svim dometima (djelokruzima, dosezima).

Superglobalne varijable u PHP-u su predefinisane varijable koje su dostupne u svim područjima vašeg koda. Mogu se koristiti bez potrebe za deklaracijom global ključne riječi unutar funkcija ili metoda. Ove varijable pružaju pristup informacijama o okruženju, HTTP zahtjevima, serveru i još mnogo toga. Evo svih superglobalnih varijabli u PHP-u:

Osim **\$GLOBALS** i **\$_SERVER** koje smo spomenuli postoje i druge superglobalne varijable:

- **\$GLOBALS**
- **\$_SERVER**

- `$_GET`
- `$_POST`
- `$_FILES`
- `$_COOKIE`
- `$_SESSION`
- `$_REQUEST`
- `$_ENV`

Superglobali se ne mogu koristiti kao varijable varijable unutar funkcija ili metoda klase.

`$GLOBALS` superglobalna

O superglobalnoj matrici \$GLOBALS rekli smo dosta toga kada smo razgovarali o dosegu. Evo jednog primjera kako možemo mijenjati stanje varijable koja je van funkcije unutar neke funkcije i referenciranjem varijable.

`params_reference.php`

```
<?php
declare(strict_types=1);

$age = 30;

function promjeniGodine(int $godine): void {
    $GLOBALS['age'] = $godine;
}

promjeniGodine(40);
echo $age; //40

// 
function promjeniGodineReference(int &$godine, int $value): void {
    $godine = $value;
}

// pozivanjem funkcije promjeniGodineReference parametar se referencira
// i mijenja $age
promjeniGodineReference($age, 50);
echo $age; //50

$noveGodine = 10;
// $promjeniGodineReference(50, 50); // ovo izaziva fatalnu grešku jer se ne
referencira na ništa a mora na varijablu
promjeniGodineReference($noveGodine, 30);
echo $noveGodine; // 30 jer je vezan referencom iz funkcije
echo $age; // $age je i dalje 50, nije se promjenio

?>
```

`declare(strict_types=1);` uključuje strogu tipizaciju tako da PHP striktno provjerava tipove podataka koji se koriste u funkcijama. Ako se funkciji proslijedi podatak pogrešnog tipa, to će rezultirati fatalnom greškom. Varijabla `$age` inicijalno je postavljena na vrijednost `30`. Globalna varijabla `GLOBALS['age']` se koristi za promjenu vrijednosti globalne varijable `$age` unutar funkcije. Funkcija prima parametar `$godine` i postavlja globalnu varijablu `$age` na tu vrijednost. Nakon poziva funkcije, vrijednost varijable `$age` se mijenja na `40`, jer funkcija direktno mijenja globalnu varijablu. `echo $age;` ispisuje `40`.

U funkciji `promjeniGodineReference` prvi parametar `$godine` proslijeđen po referenci. To znači da će sve promjene ove varijable unutar funkcije biti reflektirane i izvan funkcije. Funkcija postavlja vrijednost referencirane varijable na `$value`. Varijabla `$age` se mijenja na `50` jer je proslijeđena po referenci i vrijednost je promijenjena unutar funkcije. `echo $age;` ispisuje `50`.

Idući dio poziva funkciju `promjeniGodineReference` s `$noveGodine`. Varijabla `$noveGodine` se mijenja iz `10` u `30`, jer je proslijeđena po referenci. `echo $noveGodine;` ispisuje `30`. Na kraju ispišemo `$age`; i vidimo da je još uvijek `50`, `echo $age;` jer je posljednji put promijenjena u pozivu funkcije `promjeniGodineReference` i nije bila uključena u kasnije pozive funkcije.

`$_SERVER` superglobalna

`$_SERVER` je matrica koje sadrži informacije kao što su zaglavlja, putanje i lokacije skripti. Unose u ovoj matrici kreira web server, zato nema garancije da će svaki web server pružiti bilo što od toga; serveri mogu izostaviti neke ili dati druge koji nisu ovdje navedeni. Međutim, većina ovih varijabli uključena je u » CGI/1.1 specifikaciju i vjerojatno će biti definirana.

Evo neki indeksa

`'PHP_SELF'`

Naziv datoteke skripte koja se trenutno izvodi, u odnosu na root dokumenta. Na primjer, `$_SERVER['PHP_SELF']` u skripti na adresi <http://primjer.com/foo/bar.php> bilo bi `/foo/bar.php`. Konstanta `__FILE__` sadrži punu putanju i naziv tekuće datoteke (tj. uključene) datoteke. Ako PHP radi kao procesor komandnog reda, ova varijabla sadrži naziv skripte.

```
<?php
$count = 0;
// naziv datoteke u odnosu na root
echo ++$count . " " . $_SERVER['PHP_SELF'] . "<br>";
// puni naziv putanje i datoteke
echo ++$count . " " . __FILE__ . "<br>";
// Naziv hosta
echo ++$count . " " . $_SERVER['SERVER_NAME'] . "<br>";
// Identifikacijski string servera
echo ++$count . " " . $_SERVER['SERVER_SOFTWARE'] . "<br>";
// Naziv i revizija pomoću kojeg je str. zatražena
echo ++$count . " " . $_SERVER['SERVER_PROTOCOL'] . "<br>";
// način zahtjeva za pritup GET, HEAD, POST, PUT
```

```
echo ++$count . ". " . $_SERVER['REQUEST_METHOD'] . "<br>";  
// Vremenska oznaka početka zahtjeva  
echo ++$count . ". " . $_SERVER['REQUEST_TIME'] . "<br>";  
// Vremenska oznaka početka zahtjeva sa milisec.  
echo ++$count . ". " . $_SERVER['REQUEST_TIME_FLOAT'] . "<br>";  
// String upta pomoću kojeg je pristupljeno str.  
echo ++$count . ". " . $_SERVER['QUERY_STRING'] . "<br>";  
// Root dir. dokumenta pod kojim se tekuća skripta izvršava  
echo ++$count . ". " . $_SERVER['DOCUMENT_ROOT'] . "<br>";  
// IP adresa s koje korisnik gleda str.  
echo ++$count . ". " . $_SERVER['REMOTE_ADDR'] . "<br>";  
// Port na korisnikovom kompu za komunikaciju s web serv.  
echo ++$count . ". " . $_SERVER['REMOTE_PORT'] . "<br>";  
// Apsolutni naziv putanje skripte koja se izvodi.  
echo ++$count . ". " . $_SERVER['SCRIPT_FILENAME'] . "<br>";  
// Vrijednost dana direktivi u konf. datoteci web servera  
echo ++$count . ". " . $_SERVER['SERVER_ADMIN'] . "<br>";  
// Port koji web server koristi za komunikaciju  
echo ++$count . ". " . $_SERVER['SERVER_PORT'] . "<br>";  
// String koji sadrži verziju servera i naziv virt. komp.  
echo ++$count . ". " . $_SERVER['SERVER_SIGNATURE'] . "<br>";  
// Sadrži putanju trenutne skripte  
echo ++$count . ". " . $_SERVER['SCRIPT_NAME'] . "<br>";  
// URI koji je dan za pristup ovoj stranici  
echo ++$count . ". " . $_SERVER['REQUEST_URI'] . "<br>";  
?>
```

Rezultat je:

1. /Algebra/PHP Osnove/Predavanje 09/obrisi_me.php
2. C:\xampp\htdocs\Algebra\PHP Osnove\Predavanje 09\obrisi_me.php
3. localhost
4. Apache/2.4.58 (Win64) OpenSSL/3.1.3 PHP/8.2.12
5. HTTP/1.1
6. GET
7. 1714576553
8. 1714576553.0054
- 9.
10. C:/xampp/htdocs
11. 127.0.0.1
12. 59833
13. C:/xampp/htdocs/Algebra/PHP Osnove/Predavanje 09/obrisi_me.php
14. postmaster@localhost

```
15. 8081
16.
Apache/2.4.58 (Win64) OpenSSL/3.1.3 PHP/8.2.12 Server at localhost Port 8081

17. /Algebra/PHP/Osnove/Predavanje_09/obrisi_me.php
18. /Algebra/PHP%20osnove/Predavanje%2009/obrisi_me.php
```

[\\$_GET superglobalna](#)

Asocijativna matrica varijabli proslijeđena tekućoj skripti pomoću URL parametara (tzv. string upita tj. engl. query string). Primijetite da se polje ne popunjava samo za GET zahtjeve, već za sve zahtjeve s query stringom.

Zamislimo da u PHP datoteci imamo sljedeće:

```
<?php
echo 'Hello ' . htmlspecialchars($_GET["name"]) . '!';
?>
```

Ako je kod u istoj datoteci ili sam korisnik unio u statusnu liniju browsera ovo <http://primjer.com/proba.php?name=Sanja>

U browseru ćemo dobiti [Hello Sanja!](#)

htmlspecialchars se koristi za pretvaranje posebnih znakova u HTML entitete. Ovo je ključno za sprječavanje XSS napada (Cross-Site Scripting) i osiguravanje ispravnog prikaza korisničkog unosa u HTML dokumentima.

Ova superglobalna varijabla je dostupna u svim dosezima kroz skriptu. Nema potrebe praviti globalnu varijablu da bi ste joj pristupili unutar funkcija ili metoda. [\\$_GET](#) varijable se proslijeđuju kroz [htmlspecialchars\(\)](#) koji dekodira bilo koje kodiranje u stringu.

[\\$_POST superglobalna](#)

Asocijativna matrica varijabli proslijeđena tekućoj skripti pomoću HTTP POST metode u zahtjevu.

```
<?php
echo 'Hello ' . htmlspecialchars($_POST["name"]) . '!';
?>
```

Prepostavimo da je kod u istoj datoteci ili sam korisnik unio u statusnu liniju browsera ovo <http://primjer.com/proba.php?name=Branko>

U browseru ćemo dobiti [Hello Branko!](#)

[\\$_FILES superglobalna](#)

Asocijativna matrica koja sadrži stavke prenesene pomoću HTTP POST metode. Učitavanje datoteke zahtjeva HTTP POST metodu forme s enctype atributom postavljenim na **multipart/form-data**.

Napravimo datoteku [test.html](#) i u njoj napravimo formu:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
<form action="testscript.php" method="POST" enctype="multipart/form-data">
    <input type="file" name="datoteka">
    <input type="submit" value="Pošalji">
</form>
</body>
</html>
```

No file selected.

Forma poziva `testscript.php` metodom POST i postavljenim multipart/form-dana atributom. Upravo kako treba. Odaberemo bilo koju datoteku i nakon toga dugme Pošalji.

```
<?php
echo "Datoteka:" . $_FILES['datoteka']['name'] . "<br>";
echo "Tip : " . $_FILES['datoteka']['type'] . "<br>";
echo "Veličina : " . $_FILES['datoteka']['size'] . "<br>";
echo "Temp ime: " . $_FILES['datoteka']['tmp_name'] . "<br>";
echo "Greška : " . $_FILES['datoteka']['error'] . "<br>";
?>
```

Forma će pozvati skriptu i ispisati podatke o odabranoj datoteci: naziv, tip (npr. image/jpg, text/plain), veličinu privremeno ime datoteke (temp ime) i moguće greške.

`$_REQUEST` superglobalna

Kombinira podatke iz `$_GET`, `$_POST` i `$_COOKIE`. Ne preporučuje se korištenje zbog sigurnosnih razloga, jer može izazvati konfuziju.

Zadatak

Napiši funkciju koja ima dva parametra (name, surname). Funkcija treba konkatenirati name i suranome i zapisati u lokalnu varijablu. Zatim vrijednost u lokalnoj varijabli treba pretvoriti u velika slova. Funkcija treba vratiti vrijednost lokalne varijable. Pozovite funkciju i spremite vraćenu vrijednost u varijablu. Ispišite vrijednost varijable.

```
echo $GLOBALS['tekst']; // globalna varijabla postaje tekst varijabla
unset($GLOBALS['tekst']);
//print_r($GLOBALS);
```

```
function fullName(string $name, string $surname): string {
    $result = $name . ' ' . $surname;
    return mb_strtoupper($result);
}

$fullName = fullName('John', 'Doe');
echo $fullName . "<br>";

$fullName = fullName(surname: 'Jane', name: 'Doe');
echo $fullName . "<br>";
```

Rezultat je:

```
JOHN DOE30
DOE JANE30
```

Rekurzija

Rekurzija je u matematici i računarstvu metoda definiranja funkcija u kojima se definirajuća funkcija primjenjuje unutar definicije. Naziv se općenitije rabi za opis procesa ponavljanja objekata na sličan način.

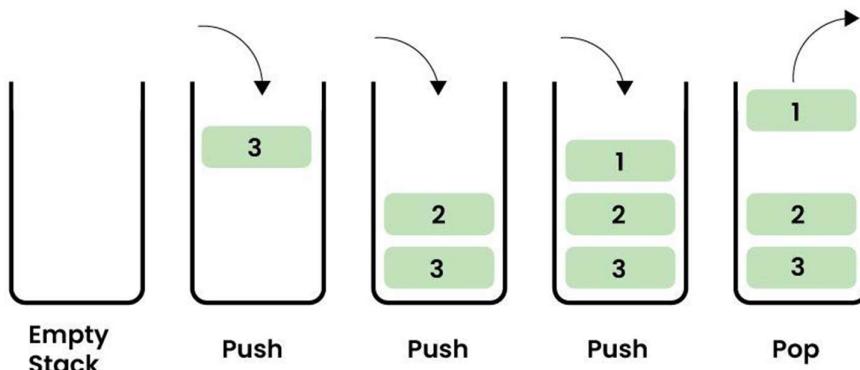
Faktorijel i Fibonačijevi brojevi su dobri primjeri rekurzije.

Idea rješavanja problema pomoću rekurzije svodi se na to da se problem podijeli na manje podprobleme istog tipa, zatim se svaki podproblem podijeli na podpodprobleme istog tipa i tako dalje, sve dok se pod... podproblem ne može riješiti na jednostavan način. Taj način se zove **bazni slučaj**.

Rekurzivne funkcije su funkcije koje pozivaju same sebe. Pritom pri svakom REKURZIVNE FUNKCIJE pozivu funkcija dobiva novi dio stack memorije, tj. stack frame. Kako memorija nije beskonačne veličine, postupak u nekom trenutku treba zaustaviti tj. potrebno je odrediti bazni slučaj: slučaj kada prestaje pozivanje funkcije. Inače dolazi do prekoračenja steka (engl. stack overflow).

Tri su pravila kojih se treba držati prilikom pisanja rekurzivnih funkcija:

1. rekurzivni algoritam mora imati bazni slučaj,
2. rekurzivni algoritam mijenja stanje parametara prema baznom slučaju,
3. rekurzivni algoritam poziva sam sebe.



```
<?php
function countdown(int $number): int {
    if ($number === 0) {
        return 0;
    }
    echo $number . '<br>';
    return countdown($number - 1);
}

$a = countdown(3);
?>
```

Pogledajmo kako to izgleda na računanju faktorijela:

```
<?php
function faktorijal(int $number): int {
    if ($number === 1) {
        return 1;
    }

    return $number * faktorijal($number - 1);
}

echo faktorijal(3); // 6=3*2*1
?>
```

Isto kao i gore kada dođemo do 1, faktorijel od 1 je 1. Baza rekurzije je definiran uslov za zaustavljanje rekurzije. Ovo sprječava beskonačnu rekurziju. Ako broj nije 1, funkcija množi trenutni broj s faktorijalom broja manjeg za jedan. Funkcija se tako poziva sama na sebe, ali s argumentom \$number - 1.

Ovo je rekurzivni slučaj koji se ponavlja dok ne dođe do baze rekurzije. Svaki rekurzivni poziv vraća rezultat koji se koristi u izrazu u prethodnom pozivu funkcije, sve dok se cijeli niz rekurzivnih poziva ne evaluira do konačnog rezultata.

Statičke varijable

```
<?php
function randomAdd(int $number): int {

    static $sum = 0; //inicijalizira se samo prvi puta, mora biti definiran u
    //funkciji!
    $sum += $number;
    return $sum;
}

$sum = randomAdd(5);
echo $sum; // 5
$sum = randomAdd(5);
echo $sum; // 10

$test ='randomAdd'; // definiran string
var_dump($test(5)); // čim su ispred stringa zagrade, PHP traži funkciju, rezultat
// je 15

?>
```

Funkcija prima podatak kroz parametar i pribroji varijabli `sum` bez `static` dobijamo varijablu koja je i ušla `static` definira varijablu koja ostaje zapamćena između poziva funkcija prvi put se napravi inicijalizacija varijable i ostaje zapamćena u funkciji.

Ključna riječ `static` definira varijablu koja ostaje zapamćena i izvan funkcije (između poziva funkcija).

Generatori

Generatori u PHP-u su posebna vrsta funkcija koja omogućava efikasnije upravljanje memorijom i lakšu iteraciju kroz velike skupove podataka. Generatori se uvode pomoću ključne riječi `yield`, koja omogućava vraćanje vrijednosti iz funkcije bez gubitka njenog stanja. Ovo omogućava kreiranje "lijenih" sekvenci podataka, gdje se elementi generiraju po potrebi, umjesto da se svi učitavaju u memoriju odjednom.

Generatori u PHP-u koriste `yield` za proizvodnju vrijednosti u svakoj iteraciji. Funkcija s `yield` ponaša se kao iterator, vraćajući vrijednost pri svakom pozivu, dok pamti gdje je stala. Kada se generator ponovno pozove, nastavlja od mjesta gdje je zadnji `yield` završio, sve dok se ne iscrpi ili dok se ne pozove `return`.

Omogućavaju da funkciju pauziramo do iduće iteracije. Zamislimo da imamo datoteku koju trebamo pročitati (npr. log datoteku) koja je velika (npr. gigabajt). Trebamo je učitati po liniju. To je loše rješenje jer trebamo učitavati red po red - to je benefit za memorijske resurse. Vidjet ćemo kako upogoniti generator za čitanje liniju po liniju. `function generator()` koristi `yield` a ne `return`:

```
<?php
function generator(): Generator {
    yield 'Ovo je prvi tekst';      // yield je umjesto return
    yield 'Ovo je drugi tekst';
    yield 'Ovo je treći tekst';

}

$gen = generator();
foreach($gen as $value) {
    echo $value . '<br>';
}
?>
```

Petlja pauzira i vrati vrijednost. Tako ćemo kasnije napraviti i s datotekom. To je benefit generatora.

```
Ovo je prvi tekst
Ovo je drugi tekst
Ovo je treći tekst
```

Evo još jednog generatora koji pravi matricu brojeva:

```
function generirajBrojeve($limit) {
    for ($i = 1; $i <= $limit; $i++) {
```

```
        yield $i;
    }
}

foreach (generirajBrojeve(5) as $broj) {
    echo $broj . PHP_EOL;
}
```

U ovom slučaju, generator `generirajBrojeve` vraća svaki broj od 1 do `limit` bez potrebe za kreiranjem cijelog niza u memoriji.

Ako kreiramo funkciju koja vraća cijelu matricu, ona može trošiti puno memorije. Primjer:

```
function sviBrojevi($limit) {
    $brojevi = [];
    for ($i = 1; $i <= $limit; $i++) {
        $brojevi[] = $i;
    }
    return $brojevi;
}
```

Za razliku od generatora, ova funkcija zauzima memoriju za cijeli niz brojevi, čak i ako se koristi samo dio podataka. Generator rješava ovaj problem stvaranjem vrijednosti samo kada su potrebne.

Generatori s ključevima

Generatori također mogu vraćati ključ-vrijednost parove, što može biti korisno kad je potrebno raditi s asocijativnim matricama:

```
function generirajKljuceve() {
    yield "prvi" => "Jedan";
    yield "drugi" => "Dva";
    yield "treći" => "Tri";
}

foreach (generirajKljuceve() as $kljuc => $vrijednost) {
    echo "$kljuc: $vrijednost" . PHP_EOL;
}
```

Ovaj generator vraća ključ-vrijednost parove bez potrebe za cijelim asocijativnim nizom u memoriji.

Pisanje i čitanje teksta datoteke

```
<?php

$filename = 'names.txt';

// Otvori datoteku za čitanje - ako otvaranje ne uspije skripta će se zaustaviti
$resource = fopen($filename, 'r') or die('Ne mogu otvoriti datoteku $filename');
// pročitaj sve iz datoteke
echo fread($resource, filesize($filename));
// obavezno zatvori datoteku
fclose($resource);

// Otvori datoteku za pisanje
$resource = fopen($filename, 'a');
// nadodaj novu stavku u datoteku
//fwrite($resource, "\nJohn Doe"); // može i ovako
fwrite($resource, PHP_EOL . "John Doe");
// obavezno zatvori datoteku
fclose($resource);

// Otvori datoteku za čitanje
$resource = fopen($filename, 'r') or die('Ne mogu otvoriti datoteku $filename za
čitanje');
$names = []; // prazna matrica
while (!feof($resource)) {
    $name = fgets($resource);
    if (trim($name)) {
        $names[] = $name;
    }
}
fclose($resource);
var_dump($names);

?>
```

`fopen($filename, 'r')` otvara datoteku `names.txt` u režimu za čitanje (`r`). `fread($resource, filesize($filename))` čita cijeli sadržaj datoteke u string koristeći veličinu datoteke (`filesize($filename)`). `fclose($resource)` zatvara datoteku nakon što je čitanje završeno.

`fopen($filename, 'a')` otvara datoteku u režimu za dodavanje (`a`), što znači da će se novi sadržaj dodati na kraj datoteke bez brisanja postojećeg sadržaja. `fwrite($resource, PHP_EOL . "John Doe")` dodaje novi red u datoteku s tekstom "John Doe". `PHP_EOL` dodaje novi red u skladu s operativnim sistemom. `fclose($resource)` zatvara datoteku nakon što je pisanje završeno.

`fopen($filename, 'r')` ponovno otvara datoteku za čitanje. `$names = []`; inicijalizira praznu listu (matricu) u koju će se spremati imena. `while (!feof($resource))` petlja koja ide kroz datoteku dok ne dođe do kraja datoteke (feof - end of file). `$name = fgets($resource)` čita jedan red iz datoteke. `trim($name)` uklanja eventualne praznine (whitespace) oko imena. `$names[] = $name;` ako red nije prazan, dodaje ga u matricu `$names`. `fclose($resource)` zatvara datoteku. `var_dump($names)` ispisuje strukturu i sadržaj matrice `$names`, koji sada sadrži sve redove iz datoteke.

Sanitizacija

Pronalazimo po ID s funkcijom `getBookByID`. Prethodno smo napravili **sanitizaciju** (engl. sanitize, dezinficiranje). To je proces čišćenja ili dezinficiranja ulaznih podataka kako bi se uklonili ili neutralizirali potencijalno štetni sadržaji, kao što je zlonamjerni kod, SQL injekcija, XSS (Cross-Site Scripting) napada (sa JavaScript kodom) ili druge sigurnosne prijetnje. Cilj sanitizacije je osigurati da se ulazni podaci mogu sigurno koristiti u aplikaciji.

Dakle ne možemo se pouzdati da nešto kliknuti. Korisniku nitko ne brani da gore u link red nešto ne upiše.

Ulagni podaci, kao što su oni iz HTML formi, URL parametara ili kolačića, često dolaze od korisnika ili vanjskih izvora i mogu biti zlonamjerni. Bez sanitizacije, aplikacija može postati ranjiva na napade koji mogu narušiti sigurnost sustava, poput krađe podataka, kompromitacije računa ili oštećenja baze podataka.

Najlakše je sve problematične znakove (`<`, `>`, `&`, `"`, `'`) koji browseru naglašavaju da će se izvršiti neka skripta u njihove HTML entitete sprječavajući XSS napade. Sanitizaciju možemo raditi za HTML kontekst, za URL-ove, za e-mail adrese, za integer vrijednosti, za stringove i za za SQL kontekst.

Sanitizacija za HTML kontekst

Sa `htmlspecialchars()`

`htmlspecialchars()` konvertira posebne HTML znakove (kao što su `<`, `>`, `&`) u njihove HTML entitete, sprječavajući XSS napade. Evo primjera koji sanitizira HTML kontekst:

```
$user_input = '<script>alert("XSS napad!")</script>';
$safe_input = htmlspecialchars($user_input, ENT_QUOTES, 'UTF-8');
// Rezultat: &lt;script&gt;alert("XSS napad!")&lt;/script&gt;';
```

Sa `htmlentities()`

`htmlentities()` pretvara sve primjenjive znakove u njihove HTML entitete, čime osigurava da se posebni znakovi kao što je `<i>` ne interpretiraju kao HTML kod.

```
$korisnicki_ulaz = '<script>alert("XSS napad!");</script>';
$siguran_ulaz = htmlentities($user_input, ENT_QUOTES, 'UTF-8');
echo $ siguran_ulaz;
// Rezultat: &lt;script&gt;alert("XSS napad!");&lt;/script&gt;';
```

Ako korisnički unos treba biti prikazan na web stranici, `htmlentities()` će osigurati da se taj unos prikaže kao običan tekst, a ne kao HTML kod. Pretvaranjem specijalnih znakova u entitete,

`htmlentities()` pomaže spriječiti XSS napade gdje napadač pokušava izvršiti skriptu putem korisničkog unosa.

Razlika između `htmlspecialchars()` i `htmlentities()` je što `htmlspecialchars()` pretvara samo posebne HTML znakove (<, >, &, ", '), dok `htmlentities()` pretvara sve znakove koji imaju HTML entitet, uključujući znakove s dijakritičkim znakovima, specijalne znakove itd.

Sanitizacija za URL-ove

`filter_var()` s filtrom `FILTER_SANITIZE_URL` sanitizira URL uklanjanjem ne-valjanih znakova:

```
$url = "http://primjer.com/?param=<script>";  
$safe_url = filter_var($url, FILTER_SANITIZE_URL);  
// Rezultat: http://primjer.com/?param=
```

Sanitizacija za e-mail adrese

`filter_var()` s filtrom `FILTER_SANITIZE_EMAIL` uklanja nevažeće znakove iz e-mail adrese:

```
$email = "pero.peric@primjer.com<script>";  
$safe_email = filter_var($email, FILTER_SANITIZE_EMAIL);  
// Rezultat: pero.peric@primjer.com
```

Sanitizacija za integer vrijednosti

`filter_var()` s filtrom `FILTER_SANITIZE_NUMBER_INT` uklanja sve znakove osim +, - i brojeva:

```
$broj = "123abc";  
$siguran_broj = filter_var($broj, FILTER_SANITIZE_NUMBER_INT);  
// Rezultat: 123
```

Sanitizacija za stringove

`filter_var()` s filtrom `FILTER_SANITIZE_STRING` uklanja HTML i PHP tagove iz stringa.

```
$string = "<h1>Zdravo svijete!</h1>";  
$safe_string = filter_var($string, FILTER_SANITIZE_STRING);  
// Rezultat: Zdravo svijete!
```

Sanitizacija za SQL kontekst

Pripremljeni SQL izrazi: Umjesto ručne sanitizacije, bolje je koristiti pripremljene izraze s PDO-om ili MySQLi-jem koji automatski pravilno sanitiziraju ulaz.

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE email = ?");  
$stmt->execute([$email]);
```

O ovome će još biti riječi.

Sanitizacija podataka je ključna mjera za zaštitu aplikacija od različitih sigurnosnih rizika i poteškoća. Kombiniranjem sanitizacije s validacijom podataka, možete značajno smanjiti rizike i poboljšati sigurnost vaše PHP aplikacije.

Funkcije za pristup datotekama - CRUD

Najjednostavniji način pristupa je po JSON standard koji služi za razmjenu podataka. Ljudima je lako čitati i pisati. Strojevi ga lako analiziraju i generiraju. JSON je tekstualni format koji je potpuno neovisan o jeziku, ali koristi konvencije koje su poznate programerima C-ovitelji jezika, uključujući C, C++, C#, Java, JavaScript, Perl, Python i mnogi drugi. Ova svojstva čine JSON idealnim jezikom za razmjenu podataka.

CRUD (engl. **C**reate, **R**ead, **U**pdate, **D**elete) u programiranju označava kreiranje, čitanje, ažuriranje i brisanje. To su četiri osnovne operacije trajnog pohranjivanja. CRUD se također ponekad koristi za opisivanje konvencija korisničkog interfejsa koje olakšavaju pregled, pretraživanje i promjenu informacija pomoću kompjuterskih formi i izvještaja.

Zahtjev za Izradu Aplikacije: Upravitelj Knjiga

Opis Projekta:

Potrebljeno je razviti PHP aplikaciju naziva „**Upravitelj Knjiga**“. Aplikacija će omogućiti korisnicima da interaktivno upravljaju kolekcijom knjiga pohranjenom u JSON formatu. Korisnici će moći dodavati nove knjige, pretraživati postojeće, ažurirati ih ili ih brisati.

Funkcionalni Zahtjevi:

1. Upravljanje Knjigama:
 - a. Dodavanje Knjige: Korisnik može dodati knjigu s detaljima kao što su naslov, autor i godina izdanja.
 - b. Pretraživanje Knjige po ID-u: Korisnik može pretražiti knjigu koristeći jedinstveni identifikator (ID).
 - c. Ažuriranje Knjige: Korisnik može ažurirati informacije o knjizi.
 - d. Brisanje Knjige: Korisnik može obrisati knjigu iz kolekcije.
2. Pohrana Podataka:
 - a. Sve knjige će biti pohranjene u datoteci **books.json**.
 - b. Aplikacija treba učitavati i spremiti knjige iz ove datoteke kada god se izvrši bilo koja operacija.

Vidimo da trebamo 4 osnovne funkcije: dodaj knjigu, pretraži knjigu po ID, ažuriraj knjigu i obriši knjigu. Trebamo još dvije funkcije: učitaj sve knjige i spremi knjige iz memorije u JSON datoteku.

Do sada smo knjige spremali u memoriju. Sada bi ih htjeli spremiti u JSON datoteku, tj. pohraniti u datoteku i naravno dohvatiti iz nje.

Prvo ćemo napisati funkciju za učitavanje knjiga iz JSON datoteke. Gledamo da imamo strukturu sa matricom dakle višedimenzionu matricu. Svaka pojedina knjiga imat će više matrica.

```
// Učitavanje knjiga iz json datoteke
function loadBooks(string $path = BOOK_FILE): array {
    // pokušava dohvatiti json datoteku
```

```
if (!file_exists($path)) {  
    return [];  
}  
$books = file_get_contents($path); // false ako nije pronađena datoteka  
return json_decode($books, true);
```

Sa redom string `$path = BOOK_FILE` definirana je i pridružena početna vrijednost putanje `$path` kao konstanta. `file_get_contents()` vraća sadržaj datoteke kao string, što nam odgovara. Naravno trebamo taj string deserijalizati da ga dobijemo u nama čitljivom obliku. Problem je što `file_get_contents()` osim stringa može vratiti `false` ako ne pročita datoteku, dakle `json_decode()` traži string (dekodira JSON string u asocijativnu matricu) a može dobiti i `false`. Zbog toga je potrebno ubaciti dio koji ako nema datoteke vraća praznu matricu. To je lako testirati ako nemamo datoteku a pozovemo funkciju `loadBooks()`. `return` funkcije vraća broj bajtova koji su zapisani u datoteku ili `false` ako nije uspio zapisati. U ovom slučaju možemo imati krivu putanju ali to ne znači da će doći do pucanja. Dakle ako i navedemo krivu putanju, datoteku ćemo opet kreirati.

LoadBooks()

Funkcija će dojaviti grešku ako nemamo dio sa `if`-om.

Još bolje je ako to napravimo na slijedeći način:

```
$array = loadBooks();
```

Iduća je na redu funkcija koja će spremati knjigu u JSON datoteku. To će biti `saveBooks()`. Prvi parametar šalje matricu sa knjigama a idući parametar šalje putanju. Logično je da na početku definiramo konstantu jer datoteku `knjige.json` koristimo i u prethodnoj funkciji za učitavanje knjiga.

```
const BOOK_FILE = 'knjige.json';
```

To je konstanta `BOOK_FILE`, koja predstavlja putanju do JSON datoteke u kojoj su pohranjene knjige.

Prije je php koristio ovaj oblik:

```
define ('knjige.json', BOOK_FILE)
```

Promjena nastaje od verzije PHP-a 8. Mogu se koristiti jedni i drugi navodnici. Ako mijenjamo stari kod a radimo u starijem PHP, moguće su greške.

U prethodnoj funkciji za učitavanje knjiga iz JSON datoteke koristili smo funkciju `json_decode` a ovdje ćemo koristiti funkciju `json_encode`. `json_decode` radi deserijalizaciju (tj. dekodira JSON string u asocijativnu matricu) a radi `encode` serijalizaciju podataka (tj. kodira asocijativnu matricu u JSON format). Ako želite lijepo pročitati moguće je koristiti drugi argument kao flag tj. `JSON_PRETTY_PRINT`. Kada otvorimo podatke u novim browserima oni su čitljivi i bez toga. Kod spremanja trebamo razmotriti da li nam treba podatak da li je dobro spremljeno. `file_put_contents($path, $jsonBooks)` sprema JSON string u datoteku.

```
// Spremanje knjiga u json datoteku
```

```
// prvi parametar šalje matricu sa knjigama, idući parametar šalje putanju
function saveBooks(array $books, string $path = BOOK_FILE): void {
    // Spremanje i slanje na pohranu
    $jsonBooks = json_encode($books, JSON_PRETTY_PRINT);
    // Ako ne postoji, kreiramo
    file_put_contents($path, $jsonBooks);
```

Da bi smo isprobali ovu funkciju da li radi napraviti ćemo sljedeći primjer:

```
saveBooks([
    [
        'id' => 1,
        'title' => 'Harry Potter',
        'author' => 'J.K. Rowling',
        'year' => 1997
    ],
    [
        'id' => 2,
        'title' => 'The Lord of the Rings',
        'author' => 'J.R.R. Tolkien',
        'year' => 1995
    ],
    [
        'id' => 3,
        'title' => 'The Great Gatsby',
        'author' => 'F. Scott Fitzgerald',
        'year' => 1925
    ]
]);
```

Ovo će stvoriti tri sloga s knjigama.

Iduća funkcija je dodavanje nove knjige [addBook](#).

```
// Dodavanje (pohranjivanje) nove knjige
function addBook(string $title, string $author, int $year): int {
    $books = loadBooks(); // učitaj knjige iz JSONa
    // Odredi maksimalnu vrijednost iz jednog stupca u ulaznoj matrici
    // ako nema zapisa vrati 1
    // array_column pronađe u višedimenzionoj matrici ključ ID i sve
    // vrijednosti koje pronađe generira u novu matricu
    $newId = empty($books) ? 1 : max(array_column($books, "id")) + 1;
    // dodaj podatak o knjizi
    $books[] = [
        'id' => $newID, // generira ID u jedinstvenom obliku
```

```

        'title' => $title,
        'author' => $author,
        'year' => $year
    ];
    saveBooks($books);           // spremi knjige u JSON
    // vrati ID
    return $newId;              // ono što izlazi iz funkcije je novi ID
}

```

`array_column($books, "id")` kreira matricu svih ID-ova knjiga. `max(array_column($books, "id")) + 1` postavlja novi ID kao najveći postojeći ID + 1. Ako nema knjiga, ID je 1.

Nova knjiga se dodaje u matricu, a zatim se cijela matrica pohranjuje u datoteku.

Sada ćemo napraviti ažuriranje postojeće knjige. Trebamo učitati knjigu a zatim je pronaći po ID jer je drugačije ne možemo ažurirati. Dakle prvo trebamo napisati funkciju za dohvaćanje knjige.

```

// Dohvati knjigu po ID-u
// Ako neko pošalje id knjige a nje nema tada treba vratiti null,
// dakle za izlaz je dovoljno ?array
function getBookById(int $id): ?array{
    $books = loadBooks();
    // trebamo dobiti podatke o jednoj knjizi (jedan slog)
    // prodi petljom po svakom elementu matrice, svaka knjiga je nova matrica
    // kada pronađe element dobili smo knjigu i izlazimo sa return
    foreach ($books as $book) {
        if ($book['id'] === $id) {
            return $book;
        }
    }
    array_map(function($book) {
        if ($book['id'] === $id) {
            return $book;
        }
    }, $books);
    return null;
}

```

Ova funkcija prolazi kroz sve knjige i vraća knjigu s traženim ID-om ili `null` ako nije pronađen. Opet imamo `?array` što znači `array | null`. Mogli smo koristiti `for`, `foreach` i `array_map` za rješenje. Pogledajmo razliku. `array_map` može mapirati cijelu `foreach` aSa `array_map` koristili bi callback funkciju. ovdje smo koristili `foreach` petlju. `array_map` može mapirati cijelu matricu i neće se zaustaviti kada pronađe prvi element. To je velika razlika između `foreach` i `array_map`. `foreach` kreće uhvatiti prvi element i kako je taj element matrice jednak `$id`, na return izlazi van iz funkcije. Tako dobivamo samo prvi element koji zadovoljava uslov. U ovom slučaju `array_map` nije praktičan. Kada bi išli s ovime raditi po autoru gdje je moguće više knjiga koristili bi `array_map`. Ipak ukloniti ćemo verziju s `array_map`.

Vratimo se na ažuriranje postojeće knjige tj. na funkciju `updateBook`. Prvo učitamo sve knjige sa `$books = loadBooks()`. Nakon ažuriranja knjige trebamo vratiti knjigu na isto mjesto. Da bi smo to napravili, nemamo njegov indeks da bi smo ga zapisali. Želimo reći `$books[1] = $book` ako pronađemo da je to indeks sa brojem 1 (ili neki drugi). Trebamo proći petljom `foreach` po `$books`:

```
// Ažuriranje postojeće knjige
// ? ispred argumenta znači da vrijednost koju šaljemo ne mora biti string
// možemo pisati string | null
// null govori da ne postoji nikakva vrijednost, vraća bool da bi smo vratili
// logičku vrijednost
function updateBook(int $id, ?string $title = null, ?string $author = null, ?int
$year = null): bool {
    $books = loadBooks(); // učita sve knjige iz JSON-a
    $book = getBookById($id); // dohvati knjigu po ID-u
    foreach($books as &$book){ // petlja prolazi po elementima
        // radimo referenciranje i tako mijenjanjem podatka na book varijabli
        // mijenjamo podatak u matrici
        if($book['id'] === $id){ // da li je id elementa jednak ulaznom elementu
            // ako je null uzmi vrijednost sa desne strane,
            // to je vrijednost koja je bila prije,
            // ako vrijednost prilikom update-a nije poslana
            // (došao je null) vraća vrijednost koja je bila
            // ako u title nije null zapiši tu vrijednost
            $book['title'] = $title ?? $book["title"];
            $book['author'] = $author ?? $book["author"];
            $book['year'] = $year ?? $book["year"];
            saveBooks($books);
            return true; // zavšavamo petlju nakon update-a i vracamo true
        }
    }
    return false;
}
```

Ako je matrica prazna, `foreach` nema što raditi. Koristimo referencu na prvi element i imao pristup cijelom podatku. Provjeravamo uslovom da li je ID koji se nalazi u prvom elementu identičan ID koji je ušao u našu funkciju. Imamo pristup preko ključa. Parametar koji smo poslali ne mora biti string, može biti i `null`. Dakle možemo pisati `string|null` ali smo dalje dužni poslati vrijednost. Ona može biti `null` ali mora biti poslana. `??` operator koji vraća prvu ne-null vrijednost između dvije opcije. U izrazu `book['title'] = $title ?? $book["title"]`; ako je `null` uzima izraz s desne strane. To je izraz koji je i bio unutra. Ako neko prilikom update-a ne pošalje `$title` nego dode `null` po default-u mi želimo vratiti vrijednost koja je bila. Ako u `$title` nije `null` onda uzmi vrijednost i upiši u lijevu stranu. Ako knjiga s danim ID-om postoji, ažurira se i lista se pohranjuje u datoteku. Tako smo napravili i za `$author` i `$year`. Pošto je `&$book` referenca, promjena će biti napravljena direktno na jer smo se referencirali točno na element. Pošto mi mijenjamo vrijednosti ključeva `title`, `author` i `year`, nad referencom, poanta je da ta referenca pokazuje u matricu i mi direktno promjenu radimo na matrici.

Promjenu pohranjujemo sa `saveBooks($books)` i kažemo `return true` da izađemo iz petlje. ako pošaljemo ID kojeg nema kod ide na `return false`. Zato je u zaglavlju funkcije definirano da je izlazni parametar `bool`.

Da bi smo provjerili da li radi trebamo pozvati `updateBook` funkciju sa imenovanim parametrima:

```
updateBook(id: 1, year:1998);
```

Moguće je rješenje sa indeksima umjesto sa referencama. Isto ćemo napraviti `foreach` petlju. Promjenu nećemo raditi na `$book` nego na `$books[$key] [$title]` jer smo u `foreach` prvo izvukli ključ pa vrijednost tog ključa. Možemo reći da se radi o indeksu.

```
foreach($books as $key => $book){    // petlja prolazi po elementima
    // radimo referenciranje i tako mijenjanjem podatka na book varijabli
    // mijenjamo podatak u matrici
    if($book['id'] === $id){      // da li je id elementa jednak ulaznom elementu
        // ako je null uzmi vrijednost sa desne strane,
        // to je vrijednost koja je bila prije,
        // ako vrijednost prilikom update-a nije poslana
        // (došao je null)vraća vrijednost koja je bila
        // ako u title nije null zapiši tu vrijednost
        $book[$key]['title'] = $title ?? $book["title"];
        $book[$key] ['author'] = $author ?? $book["author"];
        $book[$key] ['year'] = $year ?? $book["year"];
        saveBooks($books);
        return true;    // zavšavamo petlju nakon update-a i vracamo true
    }
}
return false;
```

Ovo je neki puta bolje rješenje jer ne možemo slučajno promijeniti referencu i promijeniti podatke. Ovaj puta ostavit ćemo rješenje s referencom.

Ostalo nam je još brisanje knjige po ID-u.

```
// Brisanje knjige po ID-u
function deleteBook(int $id): bool {
    $books = loadBooks();
    // koristi $id koji bi inače bio van scope-a, use će to dozvoliti
    // u callback funkciji time koristimo varijablu roditeljske funkcije
    // use ne funkcioniра kada imamo imenovane funkcije
    // nego samo sa anonimnim funkcijama
    $newBooks = array_filter($books, function($book) use ($id){
        return $book['id'] !== $id;
    });
}
```

```
// ako se broj knjiga ne razlikuje vracamo false
if(count($newBooks) === count($books)){
    return false;
}
// spremi knjige ako se broj knjiga ne razlikuje
saveBooks($newBooks);
return true;
}
```

Imamo parameter \$id Umjesto `foreach` ovdje smo koristili `array_filter`. Ovdje filtrira knjige, zadržavajući samo one koje nemaju dani ID. Ako se broj knjiga promjeni nakon filtriranja, knjiga je uspješno obrisana. `$newBooks` mogli smo napisati sa tzv. lambda funkcijom:

```
$newBooks = array_filter($books, fn($book) => $book['id'] !== $id);
```

`function` je zapisan sa `fn`. Ovdje nema `return` nego vraća ono što je nakon `=>`.

Pojavit će se greška jer imamo problem što ne vidimo \$id a ne vidimo ga jer ga zovemo iz `array_filter` koji poziva drugu funkciju, dakle funkcija unutar funkcije. Dakle iako je `deleteBook` imamo za parametar `$id`, mi ga ne vidimo. Kako bi mogli pristupiti moramo koristiti `use` koji govori da možemo pristupiti nivou iznad unutar lokalnog scope-a naše callback funkcije.

```
$newBooks = array_filter($books, function($book) use ($id){
    return $book['id'] !== $id;
});
```

Tek sada možemo pristupiti `$id` i napraviti usporedbu u `return`. `use` ne funkcioniра kada imamo imenovane funkcije nego samo sa anonimnim funkcijama.

Uspoređujemo broj knjiga u novoj i staroj matrici. Ako se razlikuje, zapisujemo novu matricu. Bez tog uslova ne bi znali da li smo išta obrisali.

Testiranje brisanja sloga kojeg nema:

```
var_dump(deleteBook(4));
```

dobjijemo `false`.

Ako probamo obrisati s npr. 4:

```
deleteBook(3);
```

Vrati nam se `true`.

Evo i cijelog koda:

```
<?php
```

```
const BOOK_FILE = 'knjige.json';

// Učitavanje knjiga iz json datoteke
function loadBooks(string $path = BOOK_FILE): array {
    // pokušava dohvatiti json datoteku
    if (!file_exists($path)) {
        return [];
    }
    $books = file_get_contents($path); // false ako nije pronađena datoteka

    return json_decode($books, true);
}

$array = loadBooks();
$array = "Nova knjiga";
var_dump($array);

// Spremanje knjiga u json datoteku
// prvi parametar šalje matricu sa knjigama, idući parametar šalje putanju
function saveBooks(array $books, string $path = BOOK_FILE): void {
    // Spremanje i slanje na pohranu
    $jsonBooks = json_encode($books, JSON_PRETTY_PRINT);
    // Ako ne postoji, kreiramo
    file_put_contents($path, $jsonBooks);
}

saveBooks([
    [
        'id' => 1,
        'title' => 'Harry Potter',
        'author' => 'J.K. Rowling',
        'year' => 1997
    ],
    [
        'id' => 2,
        'title' => 'The Lord of the Rings',
        'author' => 'J.R.R. Tolkien',
        'year' => 1995
    ],
    [
        'id' => 3,
        'title' => 'The Great Gatsby',
    ]
]);
```

```

        'author' => 'F. Scott Fitzgerald',
        'year' => 1925
    ]
);

// Dodavanje (pohranjivanje) nove knjige
function addBook(string $title, string $author, int $year): int {
    $books = loadBooks(); // učitaj knjige iz JSON-a
    // Odredi maksimalnu vrijednost iz jednog stupca u ulaznoj matrici, ako nema
    // zapisa vrati 1
    // array_column pronađe u višedimenzionoj matrici ključ ID i sve vrijednosti
    // koje pronađe generira u novu matricu
    $newId = empty($books) ? 1 : max(array_column($books, "id")) + 1;
    // dodaj podatak o knjizi
    $books[] = [
        'id' => $newId, // generira ID u jedinstvenom obliku
        'title' => $title,
        'author' => $author,
        'year' => $year
    ];
    saveBooks($books); // spremi knjige u JSON
    // vrati ID
    return $newId; // ono što izlazi iz funkcije je novi ID
}

// Ažuriranje postojeće knjige
// ? ispred argumenta znači da vrijednost koju šaljemo ne mora biti string, možemo
// pistati string | null
// null govori da ne postoji nikakva vrijednost, vraća bool da bi smo vratili
// logičku vrijednost
function updateBook(int $id, ?string $title = null, ?string $author = null, ?int
$title = null): bool {
    $books = loadBooks(); // učita sve knjige iz JSON-a
    $book = getBookById($id); // dohvati knjigu po ID-u
    foreach($books as &$book){ // petlja prolazi po elementima
        // radimo referenciranje i tako mijenjanjem podatka na book varijabli
        // mijenjamo podatak u matrici
        if($book['id'] === $id){ // da li je id elementa jednak ulaznom elementu
            // ako je null uzmi vrijednost sa desne strane, to je vrijednost koja
            // je bila prije,
            // ako vrijednost prilikom update-a nije poslana (došao je null)vraća
            // vrijednost koja je bila
            // ako u title nije null zapiši tu vrijednost
        }
    }
}

```

```
$book['title'] = $title ?? $book["title"];
$book['author'] = $author ?? $book["author"];
$book['year'] = $year ?? $book["year"];
saveBooks($books);
return true; // zavšavamo petlju nakon update-a i vracamo true
}
}
return false;
}

// ovako bi glasila petlja ako ne želimo koristiti referencu
/*foreach($books as $key => $book){ // petlja prolazi po elementima
    // radimo referenciranje i tako mijenjanjem podatka na book varijabli
    // mijenjamo podatak u matrici
    if($book['id'] === $id){ // da li je id elementa jednak ulaznom elementu
        // ako je null uzmi vrijednost sa desne strane,
        // to je vrijednost koja je bila prije,
        // ako vrijednost prilikom update-a nije poslana (došao je null)vraća
        vrijednost koja je bila
        // ako u title nije null zapiši tu vrijednost
        $book[$key]['title'] = $title ?? $book["title"];
        $book[$key] ['author'] = $author ?? $book["author"];
        $book[$key] ['year'] = $year ?? $book["year"];
        saveBooks($books);
        return true; // zavšavamo petlju nakon update-a i vracamo true
    }
}
return false;
}*/



// funkciju za ažuriranje možemo provjeriti na sljedeći način
updateBook(id: 1, year:1998);

// Dohvati knjigu po ID-u
// Ako neko pošalje id knjige a nje nema tada treba vratiti null,
// dakle za izlaz je dovoljno ?array
function getBookById(int $id): ?array{
    $books = loadBooks();
    // trebamo dobiti podatke o jednoj knjizi (jedan slog)
    // prođi petljom po svakom elementu matrice, svaka knjiga je nova matrica
    // kada pronađe element dobili smo knjigu i izlazimo sa return
```

```
foreach ($books as $book) {
    if ($book['id'] === $id) {
        return $book;
    }
}
array_map(function($book) {
    if ($book['id'] === $id) {
        return $book;
    }
}, $books);
return null;
}

// Brisanje knjige po ID-u
function deleteBook(int $id): bool {
    $books = loadBooks();
    // koristi $id koji bi inače bio van scope-a, use će to dozvoliti
    // u callback funkciji time koristimo varijablu roditeljske funkcije
    // use ne funkcioniра kada imamo imenovane funkcije
    // nego samo sa anonimnim funkcijama
    $newBooks = array_filter($books, function($book) use ($id){
        return $book['id'] !== $id;
});
    // ako se broj knjiga ne razlikuje vracamo false
    if(count($newBooks) === count($books)){
        return false;
    }
    // spremi knjige ako se broj knjiga ne razlikuje
    saveBooks($newBooks);
    return true;
}

var_dump(deleteBook(4));

?>
```

Proširena aplikacija: Upravitelj Knjiga

1. dio - pristupanje aplikaciji i pregled JSON datoteke u kojoj su podaci pohranjeni

Ovaj primjer obrađen je u Predavanju 09. Prošli smo većinu superglobalnih datoteka. Predavač je otvorio `index.php` i `functions.php`. Iskoristit ćemo kod iz prošlog zadatka Upravitelj knjiga i staviti ga u `functions.php`. Malo ćemo ga modificirati tako da ostanu samo funkcije. Obrisat ćemo konstantu `BOOK_FILE` i mesta gdje se pojavljuje kao putanja u `loadBooks` i `saveBooks`. `loadBooks` treba dobiti neku putanju. Nemamo mogućnost učitati knjige ako ne znamo gdje se datoteka nalazi. Upravo je izmjena koda često problematična kod programiranja. Sada smo napravili sami sebi problem. Mi sad ovdje vidimo potencijalnu grešku jer se nalazi sve u jednoj datoteci. Kreirat ćemo `constants.php` gdje ćemo premjestiti konstantu:

```
<?php  
const BOOKS_FILE = "books.json";  
?>
```

Tu konstantu koristit ćemo iz `functions.php`, dodajmo ovaj red na početku:

```
include_once 'constants.php';
```

Prošla aplikacija imala je određenu slabost jer su funkcije dozvoljavale da se funkciji pošalje druga putanja. Promijenili smo ih tako da se oslanjaju na konstantu a ne na parametar. U funkcijama `loadBooks` i `saveBooks` koje učitavaju knjige iz json datoteke, odnosno spremaju ih u tu istu datoteku.

Više nije moguće promjeniti putanju i nema mogućnosti zloupotrebe. To je naravno rješenje za ovaj case, nije za sve slučajeve. Sada funkcije izgledaju ovako:

```
// Učitavanje knjiga iz json datoteke.  
function loadBooks(): array {  
    // pokušava dohvatiti json datoteku  
    if (!file_exists(BOOKS_FILE)) {  
        return [];  
    }  
    // čita datoteku u string, ako ne postoji vrati false a taj slučaj je pokriven  
    $books = file_get_contents(BOOKS_FILE);  
  
    return json_decode($books, true);
```

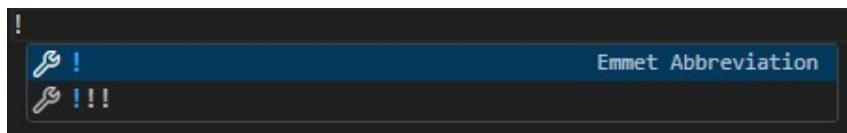
```
// Spremanje knjiga u json datoteku  
// prvi parametar šalje matricu sa knjigama, idući parametar šalje putanju  
function saveBooks(array $books): void {  
    // Spremanje i slanje na pohranu  
    $jsonBooks = json_encode($books, JSON_PRETTY_PRINT);  
    // Ako ne postoji, kreiramo datoteku iz konstante  
    file_put_contents(BOOKS_FILE, $jsonBooks);
```

Problem je što sada u `functions.php` ne javlja grešku. Na nivou Visual Studio Code-a radi s imenskim prostorima (engl. namespace). i `functions.php` i `constants.php` se nalaze u istom direktoriju. Na žalost oni se ne vide. To što trebamo napraviti je uključiti autoloader. Trebamo učitati određenu klasu a to ćemo raditi u naprednom dijelu. Sada ćemo ručno to uraditi. `constants.php` treba u `functions.php` a ne u `index.php`. U `index.php` napišemo `include 'functions.php'`; a u `function.php` napišemo `include 'constants.php'`; Ako u kodu imamo include koji uključuju duplikat uključivanja datoteke a u `constants.php` je konstanta doći će do greške jer se ista konstanta ne može uključiti dva puta. Zato je bolje odraditi sa `include_once` ili `require_once`. Ako promjenimo include u obje datoteke u `include_once`, greška će nestati. Upravo zato služe autoloader-i.

Želimo isključiti mogućnost da funkcije rade s neke druge putanje osim naše i tako treba prepraviti kod. To znači da ćemo u `loadBooks` izbrisati parametar a u provjeri da li postoji datoteka umjesto `$path` koristiti `BOOKS_FILE` kao i kod učitavanja. To moramo napraviti i kod `saveBooks` dakle parametar izbrisemo a kod spremanja knjiga u json datoteku `$path` pretvorimo u `BOOKS_FILE`.

Time smo refaktorirali kod i naravno moguće su potencijalne greške. Sada smo izbjegli da neko može poslati funkcijama neku putanju koju ne želimo.

U `index.php` kreirat ćemo html dokument tako da u Visual Basic Studiu pritisnemo ! u Emmet-u:



Pojavi se osnovna struktura html dokumenta. Naravno trebamo i zatvoriti php tag iza `include_once` reda sa `?>`. Napisat ćemo `<title>Books Manager</title>`. Napravit ćemo tablicu, tako da ćemo kreirati jedan `<tr>` tag i dva `<td>`:

```
<table>
  <tr>
    <td width=50%></td>
    <td width=50%></td>
  </tr>
</table>
```

U prvom stupcu želimo ispisati knjige koje imamo (ako ih imamo). To ćemo napraviti sa `<h1>Books</h1>` i onda ćemo opet ubaciti tablicu sa `<table></table>`. Iza te tablice razradit ćemo slučaj ako nemamo knjiga. Možemo ništa ne ispisati u tablicu ali bolje je da napišemo neku poruku. Uči ćemo u php i napraviti uslov:

```
<?php
$books = loadBooks();
if($books):
?>
.
.
.
<?php else: ?>
<p>No books!</p>
```

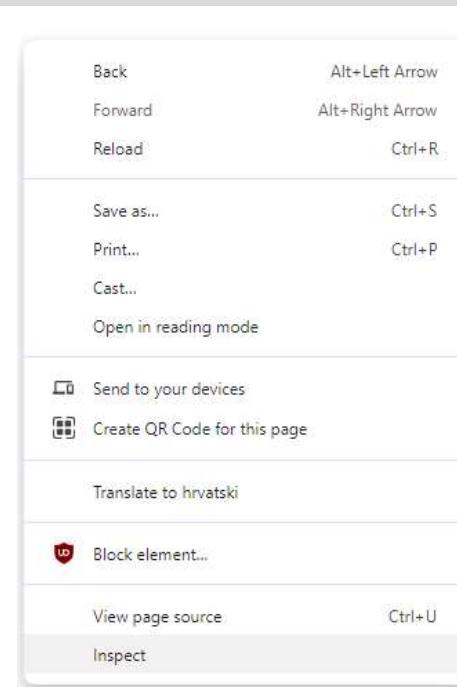
```
<?php endif ?>
```

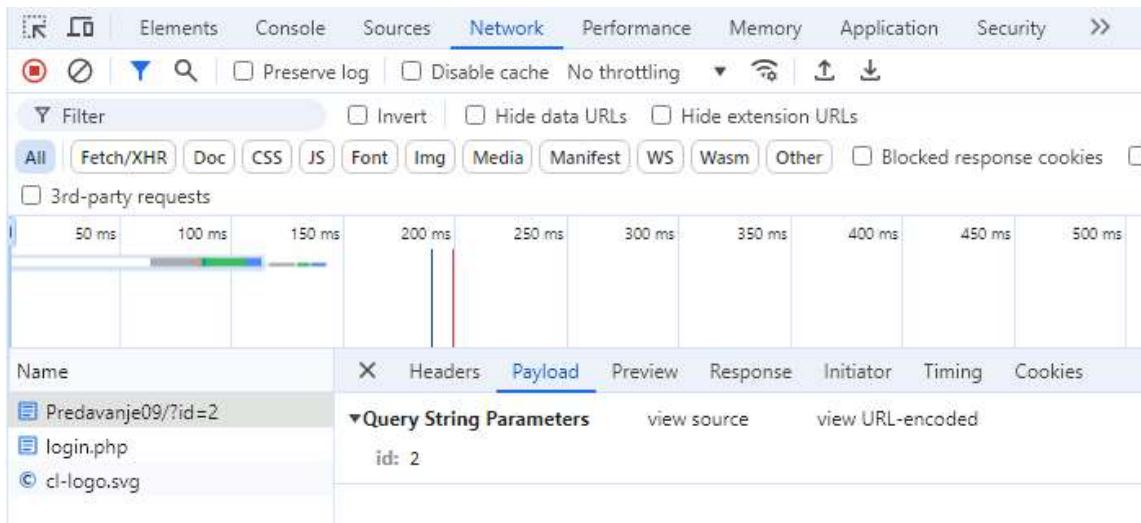
Puno je zgodnije umjesto vitičastih zagrada koristiti dvotočke :. U else smo ubacili paragraf i u njemu ispisali da nema knjiga. U dio koji smo predvidjeli tablicu (gore s tri točke) ćemo napisati:

```
<table width="100%">
<tr>
    <th>Title</th>
    <th>Author</th>
    <th>Year</th>
</tr>
<?php foreach($books as $book): ?>
<tr>
    <td>
        <?php
            echo "<a
href="" . $_SERVER['PHP_SELF'] . "?id=" . $book['id'] . "\">" . $book['title'] . "</a>" ?>
    </td>
    <td><?php echo $book['author'] ?></td>
    <td><?php echo $book['year'] ?></td>
</tr>
<?php endforeach ?>
</table>
```

Za svaku knjigu želimo ispisati podatke. Za to koristimo foreach petlju. Za svaku knjigu iskoristimo jedan <tr>. Kako se radi o asocijativnoj matrici ispisujemo \$book[] a u zgraditi pišemo ključeve title, author i year. Želimo dodati link kojim bi došli do više informacija u knjizi. Želimo neki box sa više informacija. Kasnije ćemo to proširiti s potencijalno nekom slikom koju ćemo uploadovati. U monolitnim aplikacijama će se pojavljivati takvi slučajevi.

S <a href dali smo putanju. Kroz request dali smo id. U zaglavljtu možemo poslati query string parametar tj. da šaljemo podatke kroz URL. Upišemo link (ili neki drugi gdje je primjer) <http://localhost:8081/Algebra/PHP%20snove/Predavanje09/id=2> Ako desnom tipkom na površini Google Chrome browsera odaberemo Inspect, pa zatim na desnom meniju Network. Nakon toga refreshamo stranicu (ili ponovo unesemo link ako je bio neki redirect, koji ćemo kasnije napraviti) i odaberemo Payload.





Unutra se nalaze parametri koji su poslati. Parametre odvajamo s &. Npr. `?id=2&name=test`. Možemo poslati što želimo. Ne možemo poslati matricu ali možemo poslati string. Query string parametri se ne koriste za složene podatke za dohvaćanje pojedinog zapisa. Najčešće se to koristi u browserima gdje je to moguće i promijeniti. Na postojeći URL link doda ono što smo napisali u `href`. U različitim potencijalnim situacijama to može biti problematično. Apsolutnu putanju izvukli smo s `". $_SERVER['PHP_SELF'] ."`. `var_dump($_SERVER['PHP_SELF'])` vraća:

```
string '/Algebra/PHP Osnove/Predavanje09/index.php' (length=42)
```

i to je ono što nam treba. To će nam koristi kada želimo napraviti self request na istu datoteku.

Poanta je kada kliknemo na knjigu vrati nam se `$book['id']` i to možemo vidjeti u Payload.

`$_REQUEST` superglobalna varijabla sadrži informacije koje su poslane u request-u pomoću metode `$_GET`, `$_POST` i `$_COOKIE`. To znači da metodu kojom je poslano možemo vidjeti u `Header` u Google Chrome. `$_GET`, `$_POST` su HTML mettode. To znači da ako pogledamo na Network i Headers vidjet ćemo da je metoda GET.

X	Headers	Preview	Response	Initiator	Timing	Cookies
▼ General						
Request URL: http://localhost:8081/Algebra/PHP%20Osnove/Predavanje09/						
Request Method:	GET					
Status Code:	200 OK					
Remote Address:	[::]:8081					
Referrer Policy:	strict-origin-when-cross-origin					

Ako je request došao metodom `GET` tada će zahtjev živjeti u superglobalnoj varijabli `$_GET` ali `$_REQUEST`.

Ako ima ID, na interakciju korisnika kada klikamo na različite linkove, vidimo broj ID-a. Sada s tom vrijednosti pomoću funkcije, možemo ispisati sve podatke. Superglobalna varijabla `$_GET` dobija asociativnu matricu ako nije prazan. To su informacije o svim query string parametrima.

Iza tablice želimo ispitati da li postoji asocijativna matrica ili možemo pitati postoji li ID kao ključ u tom asocijativnoj matrici.

```
if (isset($_GET["id"])) {
    echo $_GET["id"];
}
```

Ako postoji ispisat ćemo ga. To znači da ako ga ima bit će ispisano kada stisnemo na link od stavke u bazi. Ako nema tog podatka bit će prazno, što je logično. To je poanta. Sada s tom vrijednošću možemo dohvatiti sve podatke o knjizi i ispisati ih. Da probamo s `$_POST` nećemo dobiti ništa jer nije ni poslano s tom metodom nego s `$_GET`.

Sada ćemo ovo modificirati. `if` ćemo staviti s `:` (dvotočka zamjenjuje vitičastu zagradu) a završiti s `<?php endif ?>`. Umjesto dvotočaka `:` koje smo koristili u `if...else...endif` strukturi, možemo koristiti i vitičaste zagrade `{}` (Alt+123, Alt+125). Lakše je čitati sintaksu sa `:`

<https://www.php.net/manual/en/control-structures.foreach.php>

```
if (isset($_GET["id"])):
    // superglobalna je već varijabla ali
    $id = htmlentities($_GET["id"]);
    $book = getBookById($id); // pronađi po ID

    if ($book):
    ?>
        <table width="100%">
            <tr>
                <td width="50%">
                    
                </td>
                <td width="50%">
                    <?php
                        echo "
                            <p>Title: ".$book['title']."' </p>
                            <p>Author: ".$book['author']."' </p>
                            <p>Year: ".$book['year']."' </p>
                        ";
                    ?>
                </td>
            </tr>
        </table>
    <?php endif; endif; ?>
```

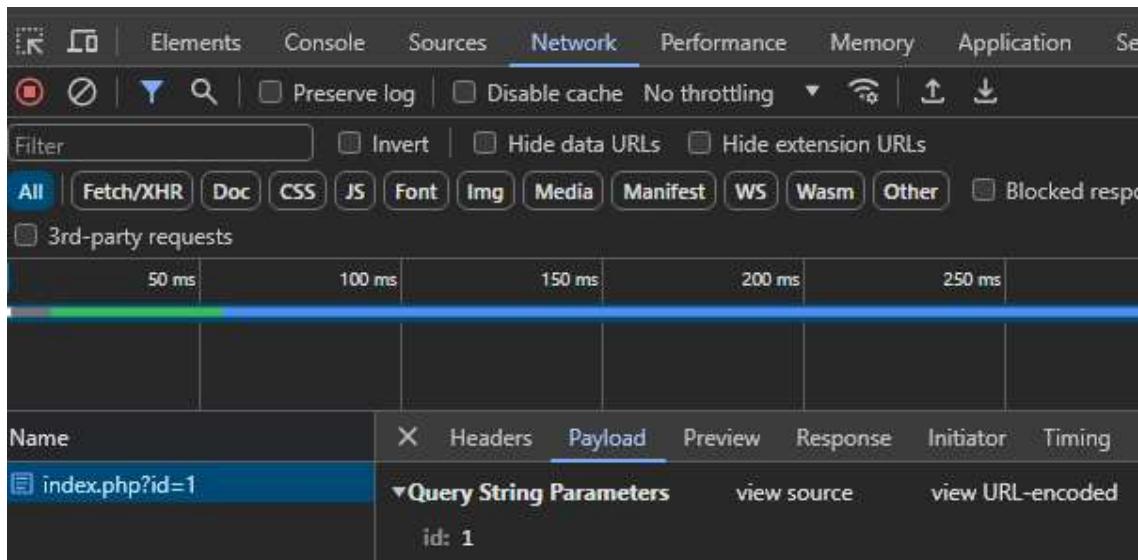
Želimo u prvom stupcu ispisati svoje knjige (ako ih imamo). Napisat ćemo `<h1>` tag u kojem ćemo napraviti još jednu tablicu sa `<table>`. Želimo ubaciti provjeru ako nema knjiga da ispišemo poruku a ako ih ima da ispišemo tablicu.

Monolitne aplikacije su kombinacija su PHP i HTML. Ako je neka vrijednost zapisana u varijablu, dostupna je i u kodu ispod.

Dobra praksa je učitati neku vrijednost zapisati u varijablu (pomoću neke funkcije) i ispitivati tu varijablu sa `if`. Nije dobro u izjavi odmah upisivati vrijednost u varijablu npr. `$books = loadBooks();`: jer je vrlo lako pogriješiti i zamijeniti taj izraz s `$books == loadBooks();`. Nakon toga ispišemo zaglavje Naslov, Autor i Godinu. Petlja koja ispisuje knjige izgledala bi ovako:

```
<?php foreach($books as $book): ?>
<tr>
    <td><?php echo $book['title'] ?></td>
    <td><?php echo $book['author'] ?></td>
    <td><?php echo $book['year'] ?></td>
</tr>
<?php endforeach ?>
```

Možemo prepraviti kod tako da šaljemo parametar tako da na kraj linka `index.php` dodamo `?id=1`. Vidjet ćemo u Inspect sljedeće:



Želimo napraviti da kada kliknemo na knjigu pošaljemo parametar na sličan način.

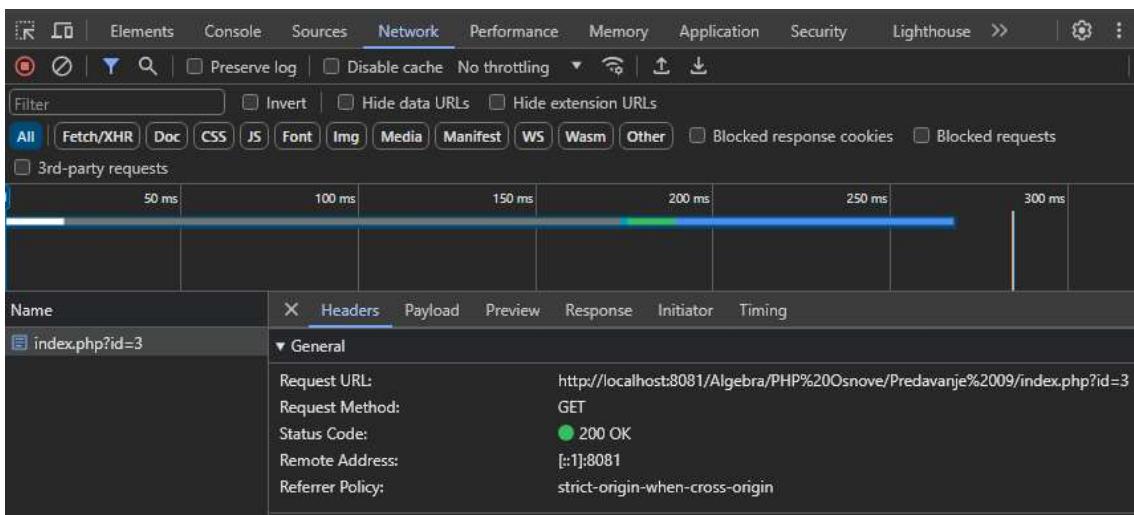
```
<table border="1">
<tr>
    <td width="50%">
        <h1>Knjige</h1>
        <?php
        $books = loadBooks();
        // ne može sa filesize() = 0 jer želimo vidjeti sadržaj matrice
```

```

if($books): // ako je loadBooks vratio matricu u kojoj nešto ima
?>
<table width="100%">
<tr>
<th>Naslov</th>
<th>Autor</th>
<th>Godina</th>
</tr>
<?php foreach($books as $book): ?> <!-- u $book je asoc. mat. -->
<tr>
<!-- superglob. varijabla daje naziv datoteke
u odnosu na root dok., često se koristi za self request -->
<td>
<?php
echo "<a href=\"".$_SERVER['PHP_SELF'] . "?id=" . $book
['id'] . "\">" . $book['title'] . "</a>" 
?>
</td>
<td><?php echo $book['author'] ?></td>
<td><?php echo $book['year'] ?></td>
</tr>
<?php endforeach ?>
</table>
<?php else: ?> <!-- a ako je loadBooks vratio null ili praznu mat. -->
<p>Nema knjiga!</p>
<?php endif ?>
</td>
<td width="50%"></td>
</tr>
</table>

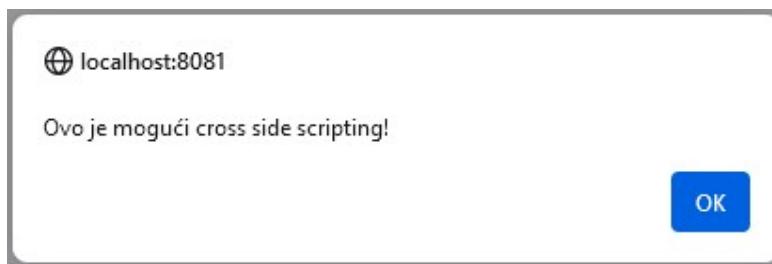
```

Kada je korisnik kliknuo, želimo taj podatak uhvatiti i obraditi. Ispod tablice, prije `</body>` ćemo obraditi sa superglobalnom varijablom `$_REQUEST` koja sadrži podatke koji su poslani putem metode `$_GET`, `$_POST` i `$_COOKIE`. Postoji više metoda koje dolaze pomoću http-a. Kako da utvrđimo kojom metodom dolaze? Otvorimo tu stranicu sa Chromom, odemo na Inspect, pa na Network jahač na vrhu i kada napravimo reload stranice odaberemo Headers. Tamo možemo vidjeti da je Request Method GET. To znači da će podaci živjeti u superglobalnoj varijabli `$_REQUEST` i `$_GET`.



Netko tko zna može umetnuti java script skriptu sa zlonamjernim kodom koji je ubačen kao parametar. Na primjer:

```
http://localhost:8081/Algebra/PHP%20Osnove/Predavanje%2009/index.php?id=<script>alert("Ovo je mogući cross side scripting!")</script>
```



Ne možemo se osloniti da će korisnik samo kliknuti, on u prostor linka može upisati šta god želi. Dakle treba onemogućiti na taj način da sve tagove pretvorimo u specijalne znakove sa `htmlentities`. To je **sanitizacija**. Treba sanitizirati sve podatke koji ulaze iz requesta. Ako je u stringu skripta ona se neće izvršiti. Bitno je da nakon korištenja `htmlentities` rukujemo isključivo s varijablom. Ako u ovom primjeru pokušamo poslati skriptu, PHP će to ispisati kao tekst jer je to pretvorio u tekst. U konačnici dolazi do greške.

Onemogućili smo sanitizacijom XSS i možemo nastaviti dalje. Rekli smo da nam `getBookById($id)` može vratiti `null`, pa shodno tome, mi ćemo ponovo provjeriti da li imamo knjigu s `if ($book)`. Opet ćemo koristiti `:` umjesto bloka vitičastih zagrada. Kako se radi o ugniježđenom `if` u drugom `if`, završit ćemo s `<?php endif; endif; ?>`. Ubacili smo tablicu s `<tr>` tagom i dva `<td>`. U oba ćemo napraviti `<td width="50%">`. U drugom smo ispisali podatke o knjizi.

Sada ćemo dodati novu formu (obrazac) s desne strane, pokraj ispisa popisa knjiga, kod drugog `<td width="50%">`. Za sada je prazan. Dodat ćemo naslov `<h1>Add Book</h1>`. Napraviti ćemo obrazac koji će biti POST. Kada šaljemo slike (datoteke) one moraju biti POST. Kada želimo preseliti s lokalnog kompjutera na server, mora biti POST metoda. `action` u formi će biti će biti na datoteku u kojoj se nalazi. `<form action=< ?> method="post"></form>` ili `<form action="" method="post">`. Dakle to se isto dogodi ako je action prazan. Možemo željeti da je druga

datoteka zadužena na obradu npr. `add-book.php` i tada će druga datoteka raditi obradu. U paragraf `<p>` ćemo staviti `Title:` i `<input type="text">`.

```
<label for="title">Title:</label>
<input type="text" name="title" id="title" required>
```

Bitno je da `name` ne smijemo izostaviti. `id` je veza između `name` i `input`. Dodat ćemo još jedan paragraf `<p>` i tu ćemo staviti autora i `input` tag za tog autora.

```
<label for="author">Author:</label>
<input type="text" name="author" id="author" required>
```

evo i godine:

```
<label for="year">Year:</label>
<input type="number" name="year" id="year" required>
```

Problem ove frontend validacije je da možemo unijeti slova `e`. Npr. `3e`. Zato frontend ne može biti jedini dio gdje se radi validacija.

Omogućit ćemo još i upload fotografije:

```
<label for="cover">Book Cover:</label>
<input type="file" name="cover" id="cover" required>
```

Možemo reći i `multiple` ako želimo više datoteka uploadovati.

Na kraju dodat ćemo dugme:

```
<button type="submit">Add Book</button>
```

Sada imamo cijelu formu. Na sva polja stavit ćemo `required`, tako da moramo popuniti sve. Na cijeloj formi u testiranju možemo ugasiti validaciju tako da dodamo na kraj forme ključnu riječ `novalidate`.

```
<form action="" method="post" novalidate>
```

Evo dosadašnjeg koda `index.php`:

```
<?php include_once 'functions.php'; ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Books Manager</title>
</head>
<body>
    <table border="1">
        <tr>
            <td width="50%">
                <h1>Knjige</h1>
            <?php
```

```

$books = loadBooks();
// ne može sa filesize() = 0 jer želimo vidjeti sadržaj matrice
if($books): // ako je loadBooks vratio matricu u kojoj nešto ima
?>


| Naslov | Autor | Godina |
|--------|-------|--------|
|--------|-------|--------|


<?php else: ?> <!-- loadBooks je vratio null ili praznu mat. -->


Nema knjiga!


<?php endif ?>

 </td>  </table> <?php if(isset($_GET['id'])): // da li je var. postavljena (različita od NILL)? $id = htmlentities($_GET['id']); // sanitiziraj podatke // bitno je da nakon ovoga ne koristimo $_GET['id'] već varijablu jer // nismo zaštićeni od cross scriptinga! echo $id; // NE $_GET['id'] $book = getBookById($id); // Dohvati knjigu po sanitiziranom ID-u if ($book): // imamo li uopće knjigu |
```

```

?>
<table width="100%">
<tr>
<td width="50%">
    SLIKA
</td>
<td width="50%">
<?php
echo "
<p>Naslov: " . $book['title'] . "</p> // konkatenacija
<p>Autor: " . $book['author'] . " </p>
<p>Godina: " . $book['year'] . " </p>
";
?>
</td>
</tr>
<!-- prvi od provjere varijable, drugi od provjere imamo li knjigu --&gt;
&lt;?php endif; endif;?&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>

```

2. dio - slanje forme POST metodom

Server kada vidi `index.php` izvrati aplikaciju prvo na interpretatoru jer browser to ne može napraviti. Prvo pročita sve knjige s lijeve strane funkcijom `loadBooks()` i ispiše sve knjige koje su u JSON datoteci. Ako nema knjiga u JSON datoteci, ispiše da nema knjiga jer je varijabla `$books` prazna. Iza toga, klasični HTML proslijeđuje i ispisuje različite stvari. `REQUEST_METHOD` provjerit će da li je poslano metodom `POST`, što u našem slučaju nije jer je poslano `GET` metodom. Daljnji kod provjerit će da li je u superglobalnoj varijabli `$_GET` postavljen ključ `id`, jer ako je, imamo zahtjev za prikaz pojedine knjige gdje ćemo prikazati tablicu sa slikom, naslovom, autorom i godinom. Sliku ćemo dohvati preko funkcije `getBookById` kojoj ćemo poslati `id`. `id` ćemo dobiti iz superglobalne varijable `$_GET` koju ćemo provući kroz `htmlentities` kako bi smo spriječili XSS napad. Klikom na link smo mogli dobiti podatke o naslovu autoru i godini tako da su čitani iz JSON datoteke. Kada dobijemo knjigu preko `id`, prikazat ćemo tablicu u kojoj su informacije o knjizi: slika, naslov, autor, godina. U ovom dijelu ćemo dodati još neke funkcionalnosti. S desne strane dodat ćemo dio koji će imati dio za dodavanje knjige. Bit će potrebno unijeti naslov, autora tekstualno i godinu numerički. Također napraviti ćemo dugme za dodavanje naslovnice knjige i dugme za zapisivanje (zvat će se Dodaj knjigu).

Korisnik treba unijeti formu za novu knjigu. U trenutku kada se pritisne dugme aktivira se forma koja govori gdje treba napraviti request (na samog sebe). U request-u ćemo koristiti `POST` metodu. Kada se dogodi taj request, cijela datoteka `index.php` će se ponovo učitati. Sve se od početka događa isto osim što će ovoga puta provjera s `REQUEST_METHOD` potvrditi da je poslano metodom `POST` i odradit će se dio koda koji se do sada nije odradio. Tu ćemo povući autora, naslov, godinu i sliku. Upload ćemo tu sliku, dodat ćemo to u JSON datoteku i nakon toga ćemo napraviti redirekciju sa `$_SERVER['PHP_SELF']`, sada se ponovo aktivira request koji sve ponovo prođe (sada prikazuje novu knjigu ako je imao). Ponovo se prikazuje tablica i ako kliknemo na link vidimo detalje.

U prvom dijelu tablice sa tagom `<td width="50%">` stavili smo dio koji ispisuje knjige i stavlja linkove. drugi dio tablice, također s tagom `<td width="50%">` još trebamo dopisati sa redoljedom kao u prethodnom ulomku. Za početak dodajemo naslov (kao i u gornjem dijelu) da bude jasno čemu služi desna strana tablice:

```
<h1>Dodavanje knjige</h1>
```

Nakon toga napraviti ćemo formu s POST metodom jer želimo slati datoteke. u formi je potrebno zadati `action`, `method` i `enctype`. `action` atribut može ostati prazan a možemo mu zadati php datoteku (**ne može biti funkcija**). U ovom slučaju trebalo bi gledati na samog sebe a to postižemo sa superglobalnom varijablom `$_SERVER['PHP_SELF']` koja daje naziv datoteke u odnosu na korijen (engl. root) dokumenta, dakle self request tj. obrada ide u istoj datoteci. U ovom slučaju sintaksa bi bila „`<?php echo $_SERVER['PHP_SELF'] ?>`“. Isto postižemo i ako navodnike ostavimo praznim.

Odvojiti ćemo dio za naslov, autora, godinu i naslovnicu knjige u različite paragrafe (browseri automatski dodaju jedan prazan red prije i nakon `<p>` elementa) radi bolje preglednosti. U paragrafu unijeti ćemo label i input. id atribut iz inputa je veza između labela i input (Ako stisnemo na Naslov: pointer miša se pozicionira u polje input). Kod naslova i autora `type` atribut za `input` je „`text`“, što je i logično. Za godinu `type` atribut za `input` je „`number`“, Kod ovog validatora postoji problem jer je moguće unijeti slovo e, što je matematički ispravno ali ne i za ovu godinu. Dakle frontend ne smije biti jedni dio gdje se validiraju stvari. Za datoteku kod `input` taga `type` atribut mora biti „`file`“.

```
<p>
    <label for="title">Naslov:</label>
    <!-- ne smijemo preskočiti name ni za jednu labelu!
        require je oznaka da je polje obavezno -->
    <input type="text" name="title" id="title">
</p>
<p>
    <label for="author">Autor:</label>
    <input type="text" name="author" id="author" >
</p>
<p>
    <label for="year">Godina:</label>
    <input type="number" name="year" id="year" >
</p>
<p>
    <label for="cover">Naslovica knjige:</label>
    <input type="file" name="cover" id="cover">
</p>
```

Potencijano je moguće dozvoliti učitavanje više datoteka sa atributom `multiple`. Red bi tada glosio:

```
<input type="file" name="cover" id="cover" multiple>
```

Potrebno je da na kraju dodamo dugme koje će aktivirati formu. Njega ćemo isto staviti u paragaf:

```
<p>
    <button type="submit">Dodaj knjigu</button>
```

```
</p>
```

Time imamo cijelu formu (obrazac).

Moguće je na kraju svakog polja staviti `required` čime mora to polje biti popunjeno. Ovo je dobro ali ne za testiranje. Kod testirana moguće je ugasiti SVE validacije ako u redu gdje definiramo formu stavimo `novalidate`.

```
<form action="" method="post" novalidate>
<p>
    <label for="title">Naslov:</label>
    <!-- ne smijemo preskočiti name ni za jednu labelu!
        require je oznaka da je polje obavezno -->
    <input type="text" name="title" id="title" required>
</p>
<p>
    <label for="author">Autor:</label>
    <input type="text" name="author" id="author" required>
</p>
<p>
    <label for="year">Godina:</label>
    <input type="number" name="year" id="year" required>
</p>
<p>
    <label for="cover">Naslovnica knjige:</label>
    <input type="file" name="cover" id="cover" required>
</p>
<p>
    <button type="submit">Dodaj knjigu</button>
</p>
</form>
```

Kako je loša preglednost podataka predavač je izmjenio tag `<table>` odmah iza `<body>` taga u `<table border="1" cellspacing="5" cellpadding="15">`. Cell spacing je prostor između svake ćelije. Cell padding je prostor između rubova ćelije i sadržaja ćelije i zadano je 0 ali smo to sada izmjenili..

With Padding			With Spacing		
hello	hello	hello	hello	hello	hello
hello	hello	hello	hello	hello	hello
hello	hello	hello	hello	hello	hello

`<td width="50%">` ispravio je u `<td width="50%" valign="top">`. valign atribut određuje da je okomito poravnanje sadržaja unutar ćelije tablice. Može još piti top, middle, bottom ili baseline . Ako

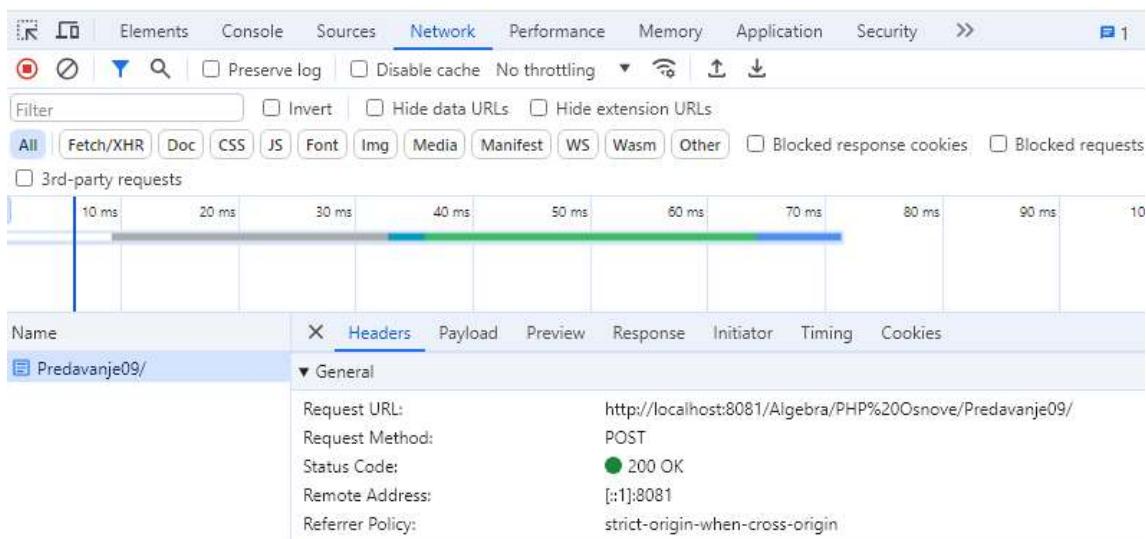
nije zadano, sadržaj ćelije tablice je okomito poravnat prema sredini. Naravno, bolje rješenje bilo bi ovo napraviti s CSS-om.

Napravili smo request na istu datoteku gdje se i nalazimo metodom POST. Ako pogledamo u Payload trebali bi vidjeti da je nešto poslano kroz request. Odaberemo Inspect->Network pa odaberemo Headers i vidimo da je poslano s POST. Ako odemo na Payload i odaberemo `index.php` vidjet ćemo title, author, year i cover nismo ništa unijeli nećemo ništa ni vidjeti u Payload. Ako nešto unesemo i stisnemo Add Book, vidjet ćemo ono što smo unijeli. Ako pogledamo u Headers, Content-Type koji smo poslali je `application/x-www-form-urlencoded`. Ovaj Content-Type ne uključuje binarne podatke. Dakle ovo neće raditi dok ne postavimo u `form: enctype="multipart/form-data"`.

```
<form action="" method="post" novalidate enctype="multipart/form-data">
```

Ako sada pogledamo sada Payload uz nove podatke s Add Book, vidjet ćemo da je cover tipa (binary).

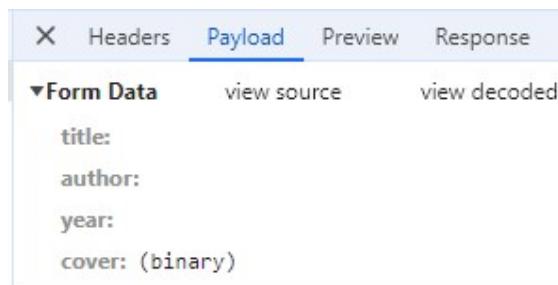
Ako kliknemo na dugme Dodaj knjigu podaci bi trebali biti poslani istoj datoteci (u `action=""`), dakle request istoj datoteci ali ovaj puta metodom POST. Dakle kažemo iz browsera Inspect->Network i stisnemo dugme Dodaj Knjigu.



The screenshot shows the Network tab of the Chrome DevTools developer console. A single request is listed under the 'All' tab. The request details are as follows:

- Name: Predavanje09/
- Request URL: `http://localhost:8081/Algebra/PHP%20Oslove/Predavanje09/`
- Request Method: POST
- Status Code: 200 OK
- Remote Address: `[::1]:8081`
- Referrer Policy: strict-origin-when-cross-origin

Vidjet ćemo da je Request Method upravo POST. Na Payload dijelu vidi se `title`, `author`, `year` i `cover`, samo što su prazni jer nismo ništa unijeli.



The screenshot shows the Payload tab of the developer console. It displays the form data sent in the POST request:

- `title:`
- `author:`
- `year:`
- `cover: (binary)`

Ako probamo unijeti testne podatke njih ćemo i vidjeti:

title: test
author: test
year: 1234
cover: IMG_20150625_102727.jpg

Zahtjev koji smo poslali išao je Request metodom POST. Ako pogledamo Content-Type on je `application/x-www-form-urlencoded`. Problem je sto taj Content-Type ne uključuje binarne podatke nego samo ime datoteke. Dakle problem je u frontend kodu. U formi treba biti uključeno da se šalju binarni podaci i to na način da se tagu `form` doda atribut `enctype="multipart/form-data"`. Red sada glasi ovako:

```
<form action="" method="post" novalidate enctype="multipart/form-data">
```

Kada ponovo učitamo stranicu i probamo testirati vidjet ćemo da je cover binarni, što smo i htjeli.

title: test
author: test
year: 1234
cover: (binary)

Ako napravimo refresh stranice, ponovili smo zadnji request koji je **POST**, dakle ostaju podaci iako ih za sada ne vidimo u poljima. Ako stisnemo Enter na stranici, tada ćemo ponovo učitati stranicu i Request Method je **GET**. Česta situacija u monolitnim aplikacijama je da aplikacija pita želite li napraviti novi request. Zato je jako bitno da kasnije možemo raditi neke provjere u PHP-u.

Iza forme možemo pitati da li je request metodom POST, što znači da se aktivirala forma.

```
if ($_SERVER["REQUEST_METHOD"] === "POST") { // da li se zahtjev aktivirao s $_POST metodom
    // podaci završavaju u superglobalnoj varijabli $_POST a datoteka završava u $_FILES
    // print_r("<pre>");
    // var_dump($_POST);
    // var_dump($_FILES);
    $title = htmlentities($_POST["title"]);
    $author = htmlentities($_POST["author"]);
    $year = htmlentities($_POST["year"]);
    // višedimenzionalna matrica koju dohvatom iz superglobalne $_FILES
    $cover = $_FILES["cover"];
    //var_dump($cover);
    // ovime osiguravamo da mora napraviti upload, UPLOAD_ERR_OK je zamjena za 0
```

```
if ($cover["error"] === UPLOAD_ERR_OK) {
    $targetDir = "uploads/"; // definiramo target direktorij
    // dodan time da ne bi bilo pregaženih datoteka, dobra praksa
    $targetFile = $targetDir . time() . "_" . basename($cover["name"]);

    if(move_uploaded_file($cover["tmp_name"], $targetFile)){
        $id = addBook($title, $author, $year, $targetFile);
        echo "<p>Book added with id: $id</p>";
        // ovo pravi GET request i refresh stranice
        header("Location: ".$_SERVER['PHP_SELF']);
    } else {
        echo "<p>Failed to add new book!</p>";
    }
} else {
    echo "<p>Failed to upload file!</p>";
}
}
```

Ako napravite `var_dump($_POST);` vidjet ćemo `array(3) { ["title"]=> string(0) "" ["author"]=> string(0) "" ["year"]=> string(0) "" }` ali u superglobalnoj varijabli `$_POST` nema binarne datoteke.

```
<?php
if ($_SERVER["REQUEST_METHOD"] === "POST") {
    var_dump($_POST);
}
?>
```

Dodavanje knjige

Naslov:

Autor:

Godina:

Naslovnica knjige:
 Choose File No file chosen

```
array(3) { ["title"]=> string(4) "test" ["author"]=>
string(4) "test" ["year"]=> string(4) "1234" }
```

Vidimo da superglobalna varijabla `$_POST` sadrži title, author i year što je u redu ali ne sadrži file zato što je to binarni zapis i `$_POST` nema s tim veze. To odlazi u superglobalnu varijablu `$_FILES`.

Dodat ćemo red: `var_dump($_FILES);`

Vidimo da je pod `cover`, matrica sa 6 elemenata. Tu su informacije o datoteci i putanja privremene datoteke gdje je pohranjena. Na nama je da tu datoteku preselimo. Vidimo element `error` ako je došlo do greške ili npr. `size` gdje vidimo ako je veličina slučajno 0.

Vidimo putanju privremene datoteke gdje je pohranio server. Mi je trebamo nekuda prebaciti. Vidimo također da li je došlo do greške, vidimo veličinu datoteke.

Možemo dodati i ovaj red ispred da vidimo lјepše ispisano:

```
print_r("<pre>");
```

```
array(3) {
    ["title"]=>
    string(4) "test"
    ["author"]=>
    string(4) "test"
    ["year"]=>
    string(4) "1234"
}
array(1) {
    ["cover"]=>
    array(6) {
        ["name"]=>
        string(23) "IMG_20150625_102727.jpg"
        ["full_path"]=>
        string(23) "IMG_20150625_102727.jpg"
        ["type"]=>
        string(10) "image/jpeg"
        ["tmp_name"]=>
        string(24) "C:\xampp\tmp\phpB566.tmp"
        ["error"]=>
        int(0)
        ["size"]=>
        int(166737)
    }
}
```

Dodavanje knjige

Naslov:

Autor:

Godina:

Naslovnica knjige:

No file chosen

```
array(3) { ["title"]=> string(4) "test" ["author"]=>
string(4) "test" ["year"]=> string(4) "1234" } array(1)
{ ["cover"]=> array(6) { ["name"]=> string(23)
"IMG_20150625_102727.jpg" ["full_path"]=>
string(23) "IMG_20150625_102727.jpg" ["type"]=>
string(10) "image/jpeg" ["tmp_name"]=> string(24)
"C:\xampp\tmp\phpB566.tmp" ["error"]=> int(0)
["size"]=> int(166737) }
```

Datoteku kada negdje želimo prikazati moramo imati njenu putanju i nju moramo pohraniti u funkciju `addBook` koja služi za dodavanje (pohranjivanje) nove knjige. Ona prima `$title`, `$author` i `$year`. Kada pohranimo datoteku, moramo imati putanju do nje kako bi je mogli prikazati. Drugi način je da vaš binarni zapis pretvorite u BASE64 zapis. To je ništa drugo nego tekstualna reprezentacija binarnog zapisa, koja se može kodirati i dekodirati. Možemo uzeti nešto binarno, pretvoriti ga (kodirati ga) u BASE64 i imati tekstualni zapis. Kasnije ga možemo dekodirati u binarni zapis. Tako se fotografija umjesto da se pohranjuje na serveru pohranjuje u bazu. Postoje i BLOB-ovi (binarni objekti) ali i treća rješenja ([AWS S3](#), [Azure Blob storage](#)). Oni rade tako da binarni zapis iz forme pohranimo u Blob storage i dobijemo nazad link. Sa malim aplikacijama dovoljno je pohraniti datoteku na svoj server. Dakle vraćamo se na funkciju `addBook`. Naravno to nisu besplatna rješenja. Sa malim bazama to se ne isplati. Zato ćemo proširiti `addBook` funkciju s četvrtim parametrom `$imagePath`. Dodat ćemo i `image` u `$books[]` matricu. Odredit ćemo neki direktorij gdje ćemo spremati sve naše datoteke.

U našem slučaju, ručno ćemo kreirati i poddirektorij `uploads`.

Sada možemo dodati `$cover` koji ga čita iz superglobalne varijable `$_FILES`.s `$_FILES["cover"]`. Tako dobivamo cijelu matricu. Tamo može biti i više datoteka, dakle sve koje su poslane metodom POST.

Ako nema zadan atribut `name` u `input` tagu forme, PHP neće moći napuniti superglobalne varijable i kod neće raditi.

```
$title = htmlentities($_POST['title']); // htmlentities pretvara HTML znakove u  
HTML entitete  
$author = htmlentities($_POST['author']);  
$year = htmlentities($_POST['year']);  
// višedimenzionalna matrica koju dohvatomo iz superglobalne $_FILES  
$cover = $_FILES['cover'];
```

Dodajemo uslov `if ($cover["error"] === UPLOAD_ERR_OK)` gdje `UPLOAD_ERR_OK` je konstanta sa sadržajem 0. i nakon toga definiramo `targetDir` i `targetFile`:

```
$targetDir = "uploads/"; // definiramo target direktorij  
// dodan time da ne bi bilo pregaženih datoteka, dobra praksa  
$targetFile = $targetDir . time() . "_" . basename($cover["name"]);
```

S `time()` smo spriječili da slučajno prepisemo ime. Potencijalno se može dogoditi isto ime, tako da ovo nije idealno rješenje. Tada se koriste random generatori.

Iza toga premjestit ćemo datoteku:

```
if(move_uploaded_file($cover["tmp_name"], $targetFile)){  
    $id = addBook($title, $author, $year, $targetFile);  
    echo "<p>Dodana knjiga s ID-om: $id</p>";  
    header("Location: . $_SERVER[PHP_SELF]");  
} else {  
    echo "<p>Ne mogu uploadovati datoteku</p>";  
}
```

Ugrađena funkcija `move_uploaded_file` ima parametre privremenog direktorija, u našem slučaju `$cover["tmp_name"]` i direktorija gdje želimo pomaći datoteku. Nakon toga zapisujemo sa `addBook` funkcijom. Dobit ćemo ispis da li je knjiga uspješno upload irana ili ne.

Ovdje ne bi bilo loše napraviti provjeru koja provjerava ekstenziju datoteke da bi prenijeli samo sliku. Da se ne bi desilo da nam neko na server prebaci `dll` ili `exe` datoteku. To nećemo za sada...

Trebamo rješiti da ako nema slike da to i dojavi i to ćemo tako da dodamo else za `if ($cover["error"] === UPLOAD_ERR_OK)`.

Probat ćemo dodati novu knjigu sa svim podacima, tako da unesemo ispravan podatak. To radi (dovavi ispod `Book added with id: ...`) ali ne vidimo tu knjigu dodanu na popisu knjiga. Ako napravimo refresh

stranice, program će još jednom napraviti zapis. I tako za svaki refresh. I to je problem. Ako pogledamo `books.json` datoteku vidjet ćemo isti zapis više puta.

Potencijalno bi mogli reći da sam program napravi `GET` request (tj. redirect) i istovremeno smo osvježili listu.

```
header("Location: ". $_SERVER['PHP_SELF']);
```

Funkcija header u PHP-u se koristi za slanje HTTP zaglavlja (headers) direktno iz PHP skripte na klijent (obično web browser). Ovo je korisno za upravljanje različitim aspektima HTTP odgovora, kao što su preusmjeravanja, određivanje tipa sadržaja, kontrola keširanja, postavljanje kolačića i drugi zadaci vezani uz HTTP protokol. Može se koristiti za preusmjeravanja, postavljanje tipa sadržaja (engl. Content-Type), kontrolu keširanja (engl. Caching), postavljanje kolačića (engl. Cookies), postavljanje HTTP statusnih kodova. U ovom slučaju, naredba iznad radi preusmjeravanje korisnika na istu stranicu koja je trenutno učitana.

S time smo dobili da se lista osvježi, spriječili smo refresh zadnjeg posta tj. ubili smo dvije muhe jednim udarcem.

`$_SERVER['PHP_SELF']` je superglobalna varijabla koja sadrži putanju trenutne skripte. Ovo uključuje ime skripte koja se trenutno izvršava.

Ako ovo pogledamo sa Inspect->Network iz browsera, možemo vidjeti da je header napravio request (redirekciju), tj. da ne vidimo payload koji je poslan.

The screenshot shows the Network tab in the Chrome DevTools developer console. A POST request to `http://localhost:8081/Algebra/PHP%20Oslove/Predavanje09/index.php` is selected. The Headers section shows the following details:

Name	Value
Request URL	<code>http://localhost:8081/Algebra/PHP%20Oslove/Predavanje09/index.php</code>
Request Method	POST
Status Code	302 Found
Remote Address	[::1]:8081
Referrer Policy	strict-origin-when-cross-origin

Response Headers

Name	Value
Connection	Keep-Alive
Content-Length	2841
Content-Type	text/html; charset=UTF-8
Date	Thu, 09 May 2024 07:54:08 GMT
Keep-Alive	timeout=5, max=100
Location	/Algebra/PHP Oslove/Predavanje09/index.php
Server	Apache/2.4.58 (Win64) OpenSSL/3.1.3 PHP/8.2.12
X-Powered-By	PHP/8.2.12

Request Headers

Name	Value
Accept	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding	gzip, deflate, br, zstd
Accept-Language	en-US,en;q=0.9
Cache-Control	max-age=0
Connection	keep-alive
Content-Length	167223
Content-Type	multipart/form-data; boundary=----WebKitFormBoundaryAvf8mgFFGze29dyrllocalhost:8081
Host	localhost:8081
Origin	http://localhost:8081
Referer	http://localhost:8081/Algebra/PHP%20Oslove/Predavanje09/index.php
Sec-Ch-Ua	"Chromium";v="124", "Google Chrome";v="124", "Not-A.Brand";v="99"
Sec-Ch-Ua-Mobile	?0
Sec-Ch-Ua-Platform	"Windows"
Sec-Fetch-Dest	document
Sec-Fetch-Mode	navigate

2 requests | 3.7 kB transferred | .

Ovo podvučeno je taj `POST` zahtjev - vidimo `multipart/form-data`. U dokumentaciji za superglobalnu varijablu `$_POST` piše ako je došao content-type multipart/form-dana u HTTP POST requestu, taj podatak se pohranjuje u superglobalnu varijablu `$_POST`. I to je ono što se pohranjuje u payload. O tome brine PHP. Dakle to punjenje isključivo radi PHP. Sadržaj superglobalne možemo prebaciti u svoju varijablu. To smo upravo napravili sa `htmlentities` (očistili ulaz koji korisnik unosi) i dijelovima `$_POST`.

Po ekstenziji je moguće doći do informacije da li je dobar tip datoteke koju korisnik pokušava uploadovati. Za to nam je dovoljan MIME tj. media tip (engl. Multipurpose Internet Mail Extensions). `dll` datoteka je npr. „`application/x-msdownload`“. Kada stavimo sliku vidimo da je ona „`image/jpeg`“. Na temelju „`type`“ moguće je explodirati i ograničiti samo na jpeg datoteke ili neke druge. Moguće je i da je datoteka koju korisnik pokušava uploadovati prevelika. Moguće je zabraniti

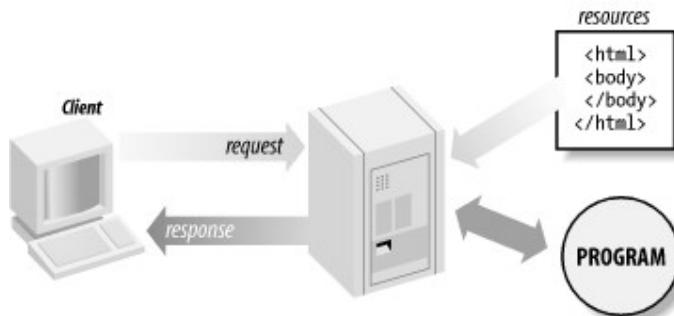
upload datoteke koje imaju preveliki „`size`“ (vidi prethodne primjere). Možemo npr. prihvati datoteku a nakon toga je mi obradimo sa [GD image](#). To je ugrađena grafička biblioteka za dinamičko manipuliranje slikama (npr. cropanje, smanjivanje).

Ponovimo kako radi ova aplikacija. Korisnik kada pokrene aplikaciju i aktivirao se `index.php`. Aktivirao se interpreter je ga ne zna pokrenuti browser. Interpreter uključi funkcije na vrhu. Pokreće se html kod. Zatim opet ide php kod gdje se s `loadBooks` učitavaju sve knjige. Ako ih ima, kreira se tablica i ispisuju se sve knjige. Ako nema niti jedne knjige ispisuje se tekst `No books!` To prepoznajemo po varijabli koja je prazna. Nakon toga s html kodom ide forma koja se prikazuje. Nakon toga opet php kod koji provjerava da li je `$_SERVER["REQUEST_METHOD"] === "POST"` tj. da li se zahtjev aktivirao s `$_POST` metodom. Kako nije ide dalje. Dolazi opet do PHP koda koji pita da li je u superglobalnoj varijabli `$_GET` postavljen ključ ID, tj. `isset($_GET["id"])`. Ako je, imamo zahtjev za prikaz pojedine knjige i prikazuje se tablica s podacima o samoj knjizi. Ako pogledamo kako smo dohvatali sliku, toj je funkcija `getBookById` preko `$id`. Prikazujemo `$title`, `$author` i `$year`.

Ako unesemo knjigu i pritisnemo dugme `Add Book`, aktivira se forma tj action koji govori gdje treba napraviti request. U našem slučaju na samog sebe. Metoda se koristi za taj request je post. Kada se dogodio request ponovo se učitala datoteka `index.php` od početka. Opet php kod koji provjerava da li je `$_SERVER["REQUEST_METHOD"] === "POST"` tj. da li se zahtjev aktivirao s `$_POST` metodom. Ovaj puta je odgovor true. Dakale povlači se `$title`, `$author`, `$year`, `$cover`. Napravi se upload slike, ostatak se doda u json datoteku i napravi redirekcija s `header("Location: ".$_SERVER['PHP_SELF'])`. To smo napravili zato da se aktivira ponovo request i on prođe ponovo onu skalameriju. Učita se ponovo se iz jsona i prikazuju se u tablici. Ako kliknemo na element prikazuje se u donjem dijelu s lijeve strane slika a s desne podaci. Ovime vidimo kako cijeli proces funkcionira.

3. dio - cookie i session

Komunikacija klijenata sa serverom ide tako da klijent pošalje **request** i dobije **response**. Server može imati stotine klijenata.



Najčešće useri imaju username i password za autentifikaciju. Nakon toga korisnik ode na neku stranicu i server mu vrati content (neki sadržaj). Nakon toga user želi na neku drugu stranicu a sistem autentifikacije mora ga nekako prepoznati. Ako ga ne prepozna pitat će ga opet za username i password. Sistem autentifikacije koristi se samo jednom a ne više puta. Radi sigurnosti danas se koristi i 2 faktorska autentifikacija (obično preko mobitela). Znamo user name i password a mobitelom možemo napraviti potvrdu. Vrlo često se koristi i otisak prsta. Nije dobra ideja da aplikacija na svaki request pita za username i password ili traži potvrdu preko mobitela. Zato i postoji sistem autentifikacije. Postoji sistem autorizacije kroz session (danasa popularan [JSON Web token](#)) i kroz tokene. Ako govorimo o autorizaciji kroz session, to znači da će server svakom klijentu odaslati session ID. Sesije se obično koriste za web aplikacije dok se tokeni obično koriste za konekcije sa serverom.

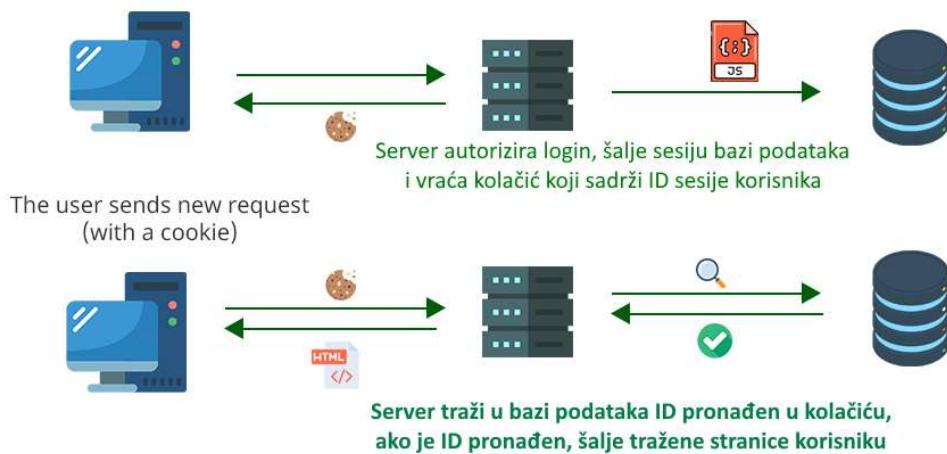
Sesija je mala datoteka, najsličnija JSON formatu, koja pohranjuje informacije o korisniku, kao što je jedinstveni ID, vrijeme login-a i isteka itd. Generira se i pohranjuje na serveru kako bi server mogao pratiti zahtjeve korisnika. Korisnik dobiva neke od tih podataka, obično ID, kao kolačić (engl. cookies) koji će biti poslani sa svakim novim zahtjevom, tako da server može prepoznati ID i autorizirati zahtjeve korisnika.

Postupak autentifikacije teče ovako:

1. Korisnik šalje login zahtjev (engl. request) na server.
2. Server autentificira login zahtjev, šalje sesiju bazi podataka (u našem slučaju sve radi server) i vraća kolačić (engl. cookies) koji sadrži ID sesije korisnika.
3. Korisnik sada šalje novi zahtjev (s kolačićem).
4. Server provjerava u bazi podataka (u našem slučaju kod sebe) ID pronađen u kolačiću, ako pronađe taj ID šalje tražene stranice korisniku.

Potencijalni problem je da neko ukrade session ID. Npr. neko ga pogodi i izgenerira (sistem koji ga generira nije dovoljno kompleksan) ili sniffingom ga uzme npr. [WireShark](#)-om jer nije https. Treći način je Cross-site scripting (XSS). XSS omogućava napadačima da ubace (gurnu) skriptu s HTML-om ili Javascript-om na strani klijenta u forme.

Korisnik šalje login zahtjev (engl. request)



Napisat ćemo `Login.php` kako bi nas server mogao prepoznati. U `index.php` napisat ćemo:

```
session_start();
```

Ovime se na serveru kada dobije zahtjev od klijenta za početnom stranicom, server generira session ID i zalijepi ga u cookie. Taj session cookie će browser zaprimiti i pohraniti. Kasnije se taj sesijski cookie koristi za identifikaciju korisnika. Mi ćemo osigurati mehanizam da u session zapišemo neki podatak i onda provjeravamo da li u tom session-u postoji taj podatak.

Npr. korisnik se prijavio sa username i password, napravili smo `session_start()` i u superglobalnu varijablu `SESSION` zapisali smo neki podatak i redirektali korisnika na `index.php`. Na `index.php` startamo session i pitamo da li u superglobalnoj varijabli `SESSION` postoji taj podatak. Ako postoji, ostavimo ga, ako ne postoji napravimo redirekciju na login.

Pokrenimo aplikaciju sa uključenim `session_start()`. Možemo vidjeti da pod Inspect->Application->Cookies postoji `PHPSESSID` koji kao vrijednost ima određeni broj.

Name	Value	Do...	Path	Exp...	Size	Htt...	Ses...	Sa...	Part...	Prio...
PHPSESSID	1brj15o06n5sbc71fu6t1hpc1f	loc...	/	Ses...	35					Me...

Ako otvorimo browser u anonimnom načinu rada (ili Private Window kod Firefoxa), vidjet ćemo potpuno drugi broj u `PHPSESSID` jer su to u biti dva odvojena klijenta sa različitim session ID. Nakon svakog refresha stranice, broj je uvijek isti. Zlonamjerni korisnik može do ovog broja doći vrlo jednostavno sa javascript skriptom koju bi stavio u npr. `index.php` ispod `<title>` taga:

```
<script>
let cookie = document.cookie;
console.log(cookie);
</script>
```

```
PHPSESSID=1brj15o06n5sbc71fu6t1hpc1f
```

Možemo prepraviti skriptu:

```
<script>
let cookie = document.cookie.split(";").reduce((prev, curr) => {
    const [key, value] = curr.split("=");
    prev[key.trim()] = value;
    return prev;
}, {});
console.log(cookie.PHPSESSID);
</script>
```

```
1brj15o06n5sbc71fu6t1hpc1f
```

Umjesto `console.log` možemo koristiti i `alert`.

Ova skripta dohvati cookie i samo ga ispiše u konzolu. Recimo da ta skripta pošalje taj `PHPSESSID` na neki server. Moguće je napraviti skriptu koja će popunjavati automatski podatke. Čim neko poslje toga dođe na aplikaciju, ima problem. Zamislimo da neka od varijabli nije sanitizirana sa `htmlspecialchars` nego direktno počupana iz requesta i pohranjena u varijablu i da je to pohranjeno u JSON datoteci. Skripta može biti automatizirana tako da pošalje `PHPSESSID` na naš server.

U PHP postoji čitav niz različitih `session funkcija`. Jedna od njih koja će nam trebati je `session_regenerate_id` za generiranje sesijskog ID-a. Korisnik kada prvi puta dođe na `index.php` starta se `session_start()`. Kada se dogodi novi request reći ćemo da regenerira id:

```
session_regenerate_id()
```

Dakle na svaki request dobijemo novu vrijednost ID-a. Kada korisnik sumnjivih namjera dođe do `PHPSESSID`, on više ne vrijedi. Kada se pokuša predstaviti s nevažećim ID, nema tog ID broja.

Nakon startanja `session_start()` imamo pristup superglobalnoj `$_SESSION`.

Vratimo se na `login.php`. Cilj nam je kada neko dođe na login i naiđe na formu za prijavu (sada ćemo je obići). Ako se neko uspješno logirao onda u superglobalnu varijablu `$_SESSION` zapisujemo podatak. Mora biti pokrenut `session_start()`. Postaviti ćemo da username bude "admin". Želimo nešto zapisati nešto u `$_SESSION`. Kada se dogodi request sa `login.php`, na `index.php`, PHP će vidjeti da imamo startan session, ID je isti jer kroz browser dobije session ID (isti je korisnik) i omogućiti nam da pristupimo superglobalnoj varijabli `$_SESSION`.

`login.php`:

```
<?php

session_start();
session_regenerate_id();

if ($_GET["login"] ?? false) {
    $_SESSION["username"] = "admin";
    header("Location: index.php");
    exit;
}

echo "Nisi prijavljen!";
```

Ako nismo pokrenuli session nemamo pristup `$_SESSION["username"]`. Kada se dogodi request sa `login.php` na `index.php` on će vidjeti da imamo startan session, id je isti, jer će kroz browser dobiti taj session id i omogućiti nam da pristupimo superglobalnoj varijabli `$_SESSION["username"]`. To znači da ćemo u `index.php` staviti `if($_SESSION["username"] != "admin")` i da ako jeste ide preusmjeravanje na `login.php`:

U `index.php` na sam početak dodat ćemo dio:

```
<?php
include_once 'functions.php';
```

```
session_start();

session_regenerate_id();

var_dump($_COOKIE["username"]);
// ako je session username različit od admin, napravi redirekciju
if($_SESSION["username"] !== "admin"){
    header("Location: login.php");
    exit;
}

?>
```

Ovdje vidimo dio koji provjerava ima li u superglobalnoj varijabli `$_SESSION` username i da li je različit od admin, desni se redirekcija na `login.php`. Time smo zaštitili index.php. Neko tko pokuša doći mora imati u superglobalnoj varijabli `$_SESSION` ključ `"username"` i vrijednost tog ključa mora biti `"admin"`. Da bi to dobili moramo to napraviti u `login.php`. Ako dođemo na `index.php`, ide preusmjeravanje na `login.php` i tekst `Nisi prijavljeni!`

Međutim ako unesemo `login.php?login=true`, ide preusmjeravanje na `index.php` i otvara se stranica. Ako napravimo refresh na `index.php` vidjet ćemo da i to radi. Radi zato što provjera `if($_SESSION["username"] !== "admin")` nije različita, dakle session ID se podudaraju. Nedostatak je potencijalni mogući hijacking. Rješenje toga je staviti `htmlentities` za sanitiziranje inputa.

Session se uništi kada se zatvori browser. Session se dijeli unutar browsera. Pomoću kolačića moguće je riješiti da korisnik ostane zapamćen. Kolačić se može flag-ati da se može pristupiti samo sa http requestom, dakle nema pristupa java scriptom. U login možemo dodati setcookie naredbu iznad `header()` naredbe:

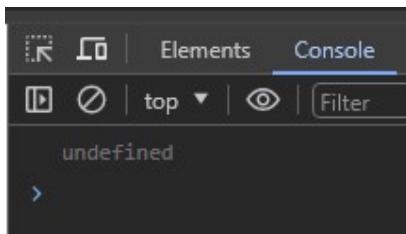
```
setcookie("username", "admin", time() + 3600, "/", "", false, true);
```

Kolačić se ne može obrisati, ali se obriše kada ode u prošlost (umjesto `+` stavimo `-`). Može biti do 64 Kb. Prvi parametar je ime, idući je vrijednost, kada ističe, putanja, domena, secure i http-only. Moguće je ograničiti u kojim direktorijima kolačić djeluje (za cijelu putanju `"/"`). Ako je secure na `true`, kolačići su kriptirani. Zadnji parametar omogućava da vidimo kolačić na Inspect->Application. Ako je na `false` (a treba biti na `true`) riskiramo da neko java scriptom uhvati kolačić.

Napravimo skriptu u java scriptu u `index.php`:

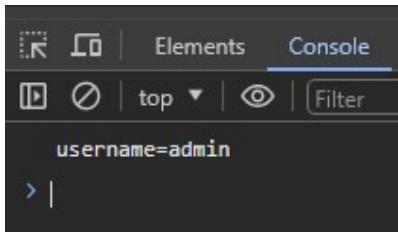
```
<script>
let cookie = document.cookie.split(";").find(cookie =>
cookie.includes("username"));
console.log(cookie);
</script>
```

Sa uključenim `httponly` to izgleda ovako:



Application								
Name	Value	Do...	Path	Exp...	Size	Htt...	Sec...	Set...
PHPSESSID	ca3q78skoqbntnc9got1rm0sc	loc...	/	Ses...	35			
username	admin	loc...	/	202...	13	✓		

Sa isključenim `httpOnly` to izgleda ovako:



Application								
Name	Value	Do...	Path	Exp...	Size	Htt...	Sec...	Set...
PHPSESSID	uv6ichh1a9n9+96sc483t71go	loc...	/	Ses...	35			
username	admin	loc...	/	202...	13			

U PHP je moguće čitati kolačić preko superglobalne varijable `$_COOKIE`.

```
var_dump($_COOKIE["username"]);
```

Odgovor je:

```
string(5) "admin"
```

PHP može pročitati vrijednost iz kolačića ali samo ako nije http-only na true.

Zadatak

- Datoteka: Podaci se nalaze u datoteci `podaci.txt`. Svaki red u datoteci sadrži jedan zapis u formatu `ime;prezime;godine`.
- Čitanje datoteke: Skripta treba otvoriti i pročitati datoteku red po red.
- Parsiranje podataka: Svaki red treba razdvojiti na tri dijela koristeći znak ;. Svaki od tih dijelova treba pohraniti u asocijativnu matricu s ključevima ime, prezime i godine.
- Pohrana u matricu: Sve asocijativne matrice trebaju biti pohranjeni unutar jedne višedimenzionalne matrice.

- Ispis matrice: Nakon što su svi podaci pročitani i pohranjeni, ispisati cijelu višedimenzionalnu matricu.

Primjer sadržaja datoteke **podaci.txt**:

Na kraju izvođenja skripte, očekivani ispis treba izgledati ovako:

Primjer sadržaja datoteke **podaci.txt**:

```
Ivan;Horvat;35
Marta;Kovačić;28
Array
(
    [0] => Array
        (
            [ime] => Ivan
            [prezime] => Horvat
            [godine] => 35
        )

    [1] => Array
        (
            [ime] => Marta
            [prezime] => Kovačić
            [godine] => 28
        )
)
```

Napomena: Skripta treba ispisati grešku ukoliko datoteka ne postoji ili ne može biti pročitana.

```
<?php
const PODACI = "podaci.txt";
```

```
// Učitavanje txt datoteke
function loaddata(): array {
    $mat = []; $ima_li = 0;
    // ima li datoteke?
    if (!file_exists(PODACI)) {
        // ako ne postoji, vrati praznu matricu
        echo "Nema datoteke " . PODACI . "<br>";
        return [];
    }

    // čita datoteku u string, ako ne postoji prekini izvrsenje
    $datoteka = fopen(PODACI, "r");

    while(! feof($datoteka)) { // čita red po red do kraja datoteke
        $red = fgets($datoteka); // učitaj jedan red i stavi ga u $red
        $dijelovi = explode(";", $red); // razdvoji $red na dijelove
        $ima_li = $ima_li + strlen(str_replace(";", "", $red));
        $mat2 = array("ime"=>$dijelovi[0], "prezime"=>$dijelovi[1],
        "godine"=>$dijelovi[2]);
        array_push($mat, $mat2); // dodaj u dvodimenzionu matricu $mat matricu
    }
    fclose($datoteka);
    if ($ima_li < 3){
        echo "Nema podataka u datoteci " . PODACI . "<br>";
        return [];
    }
    else
        return $mat; // vrati dvodimenzionu matricu
}

$a = loaddata();
print_r($a);
```

Parcijalni ispit

Razvoj jednostavne PHP aplikacije za analizu riječi

1. Kreirajte aplikaciju (vidi sliku) koja će iz datoteke `words.json` u desnu tablicu ispisati sve riječi koje su analizirane.
2. S lijeve strane kreirajte obrazac kroz koji će se unositi nova riječ.
3. Unesenu riječ treba obraditi na sljedeći način:
 1. Polje ne smije biti prazno
 2. Izbrojati broj slova u riječi
 3. Izbrojati suglasnike i samoglasnike u riječi (za ovu funkcionalnost kreirajte funkcije)
4. Obradenu riječ treba zapisati u `words.json` datoteku te ju prikazati u tablici.

Upišite željenu riječ!

Upišite riječ:

Riječ	Broj slova	Broj suglasnika	Broj samoglasnika
test	4	3	1



Polje ne smije biti prazno!

Upišite riječ:

Riječ	Broj slova	Broj suglasnika	Broj samoglasnika
test	4	3	1



Upišite riječ:

Riječ	Broj slova	Broj suglasnika	Broj samoglasnika
test	4	3	1
developer	9	5	4

Aplikacija radi na taj način da korisnik unosi riječ u formu i klikne na dugme "[Pošalji](#)". PHP skripta u `functions.php` uzima tu riječ, sanitizira je, broji slova, samoglasnike i suglasnike. Ove informacije se pohranjuju u `words.json` datoteku. Svi pohranjeni podaci iz `words.json` datoteke se prikazuju u tablici na desnoj strani stranice.

`<meta charset="UTF-8">` postavlja enkodiranje na UTF-8 kako bi podržalo dijakritičke znakove. U `body` tagu stranice koristi se `include` direktiva za uključivanje `functions.php` datoteke, koja sadrži logiku obrade podataka. Stranica je podijeljena u dva dijela unutar HTML tablice: Lijevi dio sadrži HTML formu za unos riječi i desni dio prikazuje tablicu s podacima o riječima koje su ranije unesene i obrađene.

Korisnik unosi riječ u tekstualno polje forme (`<input type="text">`) i šalje je na obradu klikom na dugme "Pošalji". Forma koristi POST metodu za slanje podataka i osigurana je pomoću `htmlspecialchars()` funkcije koja štiti od XSS napada.

Nakon toga, program ide na učitavanje i prikaz podataka iz `words.json`. Ako JSON datoteka postoji, učitavaju se podaci iz nje pomoću `file_get_contents()` funkcije. Podaci se dekodiraju iz JSON formata u PHP matricu pomoću `json_decode()` funkcije. Svaka riječ i njeni atributi (broj slova, broj samoglasnika, broj suglasnika) prikazuju se u tablici.

Sada idemo na `functions.php`. Funkcija prima unesenu riječ kao parametar. Unesena riječ pretvara se u mala slova kako bi pretraga bila neosjetljiva na velika/mala slova. Pomoću `str_split()` funkcije riječ se razbija na pojedine znakove. Znakovi se pretražuju unutar matrice `['a', 'e', 'i', 'o', 'u']` koji predstavlja samoglasnike. Broj samoglasnika se povećava svaki put kada znak odgovara nekom od samoglasnika.

Glavni dio koda za obradu unosa provjerava se da li je POST metoda korištena za slanje podataka. Ako je unesena riječ nije prazna, podaci se obrađuju: koriste se `htmlspecialchars()` i `trim()` funkcije za sanitizaciju unosa, računa se broj slova, broj samoglasnika i broj suglasnika. Provjeravamo postoji li unesena riječ već u podacima. Izvrti petlju i usporedi sa unesenom riječi. Ako pronađe tu riječ postavi varijablu `$vec_postoji` na `true` i izađe iz petlje. Ako riječ ne postoji u matrici, podaci se dodaju u JSON datoteku: ako datoteka `words.json` postoji, podaci se učitavaju i dekodiraju u PHP matricu, nova riječ s njenim atributima dodaje se na kraj te matrice, ažurirani podaci ponovno se spremaju u `words.json` koristeći `file_put_contents()`.

Da bi smo spriječili ponovno slanje podataka s refresh stranice dodali smo `header`.

Ako datoteka `words.json` ne postoji ili ne može biti pročitana, skripta neće generirati grešku nego će kreirati novu datoteku ili izbaciti poruku o grešci. Ako korisnik ne unese riječ, prikazuje se poruka "Unesite riječ".

`index.php`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- osigurava da stranica ispravno prikazuje dijakritičke znakove. -->
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Parcijalni ispit</title>
</head>

<body>

<?php include 'functions.php'; ?>

<table border='0' cellpadding='20'>
    <tr>
        <td><h1><center>Upiši željenu riječ!</center></h1>
        <td>
```

```
<!-- HTML forma za unos riječi -->
<!-- Forma koristi POST metodu za slanje podataka na istu stranicu
koristeći zaštitu od XSS -->
<form action=<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>" method="post">
    <label>Upišite riječ:</label><br>
    <!-- Unesi riječ u input polje a s dugmetom ga pošalji na server --
    >
    <input type="text" id="enterWord" name="enterWord"
placeholder="Unesi neku riječ..." required><br><br>
    <button type="submit">Pošalji</button>
</form>
</td>
<td>
<?php
// Učitavanje i prikaz prethodnih riječi iz words.json prilikom svakog
učitavanja stranice
if (file_exists('words.json')) { // Ako datoteka postoji
    $podaci = json_decode(file_get_contents('words.json'), true); // //
dekodiraj u varijablu
    if (!empty($podaci)) {
        echo "<table border='1' cellspacing='2' cellpadding='5'>";
        // Ispiši zaglavlje tablice
        echo "<tr>
            <th>Riječ</th>
            <th>Broj slova</th>
            <th>Broj suglasnika</th>
            <th>Broj samoglasnika</th>
        </tr>";

        foreach ($podaci as $item) {
            // Provjeri da li postoje ključevi 'enterWord',
            'brojSlova', 'brojSuglasnika' i 'brojSamoglasnika' prije ispisa
            if (isset($item['enterWord'], $item['brojSlova'],
            $item['brojSuglasnika'], $item['brojSamoglasnika'])) {
                echo "<tr>";
                echo "<td>" . htmlspecialchars($item['enterWord']) .
            "</td>";
                echo "<td>" . $item['brojSlova'] . "</td>";
                echo "<td>" . $item['brojSuglasnika'] . "</td>";
                echo "<td>" . $item['brojSamoglasnika'] . "</td>";
                echo "</tr>";
            }
        }
    }
}
```

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

</body>
</html>
```

function.php

```
<?php

// Samoglasnici - Prvo se riječ pretvara u mala slova, a zatim se prolazi
// kroz svaki znak u riječi i broji se ako je znak samoglasnik.
function countVowel($enterWord) {
    $samoglasnici = ['a', 'e', 'i', 'o', 'u'];
    $brojSamoglasnika = 0;

    foreach (str_split(strtolower($enterWord)) as $znak) {
        if (in_array($znak, $samoglasnici)) {
            $brojSamoglasnika++;
        }
    }
    return $brojSamoglasnika;
}

if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Provjeri da li je unešena riječ u input polje
    if (!empty($_POST["enterWord"])) {
        $enterWord = htmlspecialchars(trim($_POST["enterWord"])); // Sanitizacija
        unosa

        // Učitavanje postojećih podataka iz JSON datoteke
        $podaci = [];
        if (file_exists('words.json')) { // ako postoji datoteka
            $json_content = file_get_contents('words.json'); // učitaj je u
            varijablu
            if ($json_content !== false) {
```

```
$podaci = json_decode($json_content, true) ?? []; // dekodiraj, ako  
nije dekodiraj  
}  
}  
  
// Provjera postoji li unesena riječ već u podacima  
$vec_postoji = false;  
foreach ($podaci as $item) {  
    if (strcasecmp($item['enterWord'], $enterWord) == 0) { // uporedi  
        neosjetljivo na velika/mala slova  
        $vec_postoji = true;  
        break;  
    }  
}  
  
if (!$vec_postoji) {  
    // Ako riječ ne postoji, dodaj je u podatke  
    $brojSlova = strlen($enterWord);  
    $brojSamoglasnika = countVowel($enterWord);  
    $brojSuglasnika = $brojSlova - $brojSamoglasnika;  
  
    // Dodavanje nove riječi u postojeće podatke na kraj matrice  
    $podaci[] = array(  
        'enterWord' => $enterWord,  
        'brojSlova' => $brojSlova,  
        'brojSuglasnika' => $brojSuglasnika,  
        'brojSamoglasnika' => $brojSamoglasnika  
    );  
  
    // Spremanje ažuriranih podataka u JSON datoteku  
    file_put_contents('words.json', json_encode($podaci,  
JSON_PRETTY_PRINT));  
}  
  
// Spriječavanje ponovnog slanja podataka na refresh  
header("Location: " . $_SERVER["PHP_SELF"]);  
exit();  
} else {  
    echo "<p>Unesite riječ</p>";  
}  
}  
?  
>
```

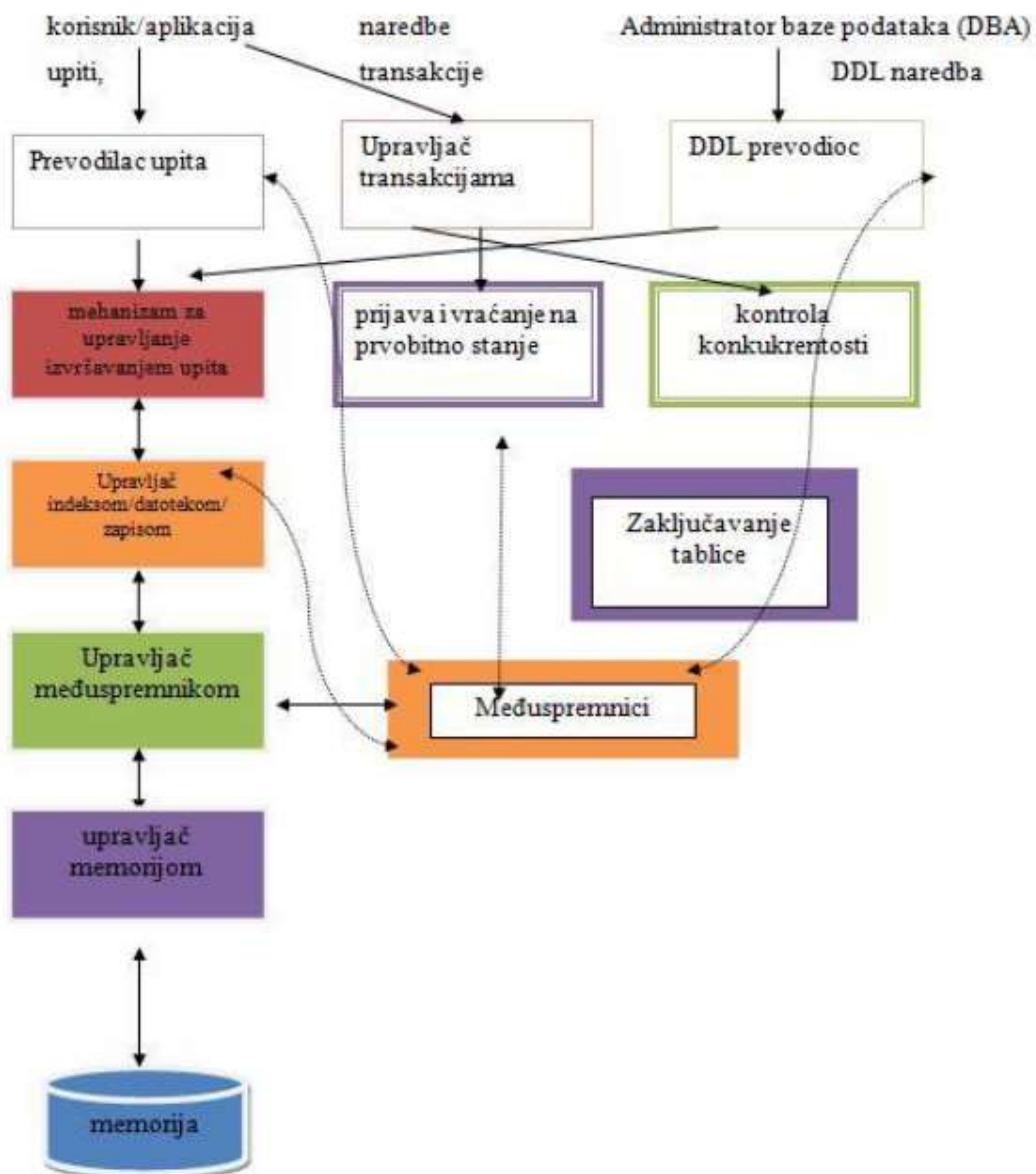
Uvod u baze podataka

Baza podataka je organizirani skup podataka pohranjenih u računalu na sustavan način, tako da joj se računalni program može obratiti prilikom odgovaranja na problem.

Svaki se zapis za bolji povratak i razvrstavanje obično prepoznaće kao skup elemenata (činjenica) podataka.

Predmeti vraćeni u odgovoru na upitnike postaju informacije koje se mogu koristiti za stvaranje odluka koje bi inače mogle biti mnogo teže ili nemoguće za stvaranje.

Računalni program korišten za upravljanje i ispitivanje baze podataka nazvan je sustav upravljanja bazom podataka.



Vrste baza podataka

Distribuirane baze podataka

Podaci nisu na jednom mjestu i raspoređeni su po različitim lokacijama. Informacije se prikupljaju u zajedničku bazu s lokalnih kompjutera. Ovu mogućnost imaju samo velike baze podataka.

Objektno orientirane baze podataka

Ova vrsta računalne baze podataka podržava pohranu svih vrsta podataka. Podaci se pohranjuju u obliku objekata. Objekti koji se čuvaju u bazi podataka imaju atribute i metode koje definiraju što učiniti s podacima. PostgreSQL je primjer objektno orientiranog relacijskog DBMS-a.

Centralizirana baza podataka

To je centralizirana lokacija, a korisnici iz različitih sredina mogu pristupiti tim podacima. Ova vrsta računalnih baza podataka pohranjuje aplikacijske procedure koje korisnicima pomažu pristupiti podacima čak i s udaljene lokacije.

Baze podataka otvorenog koda

Ova vrsta baze podataka pohranjuje informacije koje se odnose na operacije. Uglavnom se koristi u području marketinga, odnosa sa zaposlenicima, korisničke službe, baza podataka. MariaDB je takav primjer.

Baze podataka u oblaku

Baza podataka u oblaku je baza podataka koja je optimizirana ili izgrađena za takvo virtualizirano okruženje. Mnogo je prednosti baze podataka u oblaku, od kojih neke mogu platiti kapacitet pohrane i propusnost. Također nudi skalabilnost na zahtjev, zajedno s visokom dostupnošću.

NoSQL baze podataka

NoSQL baza podataka koristi se za velike skupove distribuiranih podataka koji ne zahtjevaju fiksnu shemu. Postoji nekoliko problema s izvedbom velikih podataka koje učinkovito rješavaju relacijske baze podataka. Ova vrsta računalne baze podataka vrlo je učinkovita u analizi velikih nestrukturiranih podataka. NoSQL baza podataka označava „Ne samo SQL“ ili „Ne SQL“. Iako bi bolji izraz bio "NoREL", NoSQL se udomaćio. NoSQL baze podataka uglavnom se kategoriziraju u četiri tipa: par ključ-vrijednost, orientirani na stupce, temeljeni na grafikonu i orientirani na dokument. Svaka kategorija ima svoje jedinstvene atributte i ograničenja. Primjeri takvih baze je MongoDB, Redis, Firebase (Realtime i Fierestore), Cassandra, Couchbase, DynamoDB, Elasticsearch.

<https://www.guru99.com/hr/nosql-tutorial.html>

Relacijski model baze podataka

Relacijski model baze podataka je poseban tip baze podataka kod kojeg se organizacija podataka zasniva na relacijskom modelu.

Podaci se u ovakvim bazama organiziraju u skup relacija između kojih se definiraju određene veze.

Relacija se definira kao skup **n-torki** sa istim atributima, definiranih nad istim domenima iz kojih mogu da uzimaju vrijednosti. Pojam n-torce se odnosi na redove (zapise, slogove) u relacijskoj tablici koji imaju neku vrijednost za atribute. N-torka s istim atributima su redovi koji imaju jednaku vrijednost atributa. Ovo ne želimo u relacijskom modelu.

U relacijskim bazama podataka, svaka relacija mora da ima definiran **primarni ključ**, koji predstavlja atribut pomoću kojeg se jedinstveno identificira svaka **n-torka**. Ako tablica nema definirani primarni ključ ili jedinstveni ključ, sistem ne može sprječiti unos duplikata. Relacija optionalno može da posjeduje i **strani ključ**, preko kojeg ostvaruje vezu sa drugim relacijama.

Upravljanje ovakvim bazama podataka se realizira preko **sistema za upravljanje relacijskim bazama podataka**.

Među najpopularnijim takvim sustavima danas su: **Microsoft SQL Server**, **Oracle Database**, **MySQL** i drugi. Većina tih sistema koristi upitni jezik **SQL** za manipulaciju podacima.

Uvod u MySQL

MySQL je besplatan, open source sistem za upravljanje bazom podataka. Uz **PostgreSQL**, **MySQL** je čest izbor baze za projekte otvorenog koda, te se distribuira kao sastavni dio serverskih Linux distribucija, no također postoje inačice i za ostale operacijske sustave poput Mac OS-a, Windowsa itd.

MySQL baza je slobodna za većinu uporaba. Ranije u svom razvoju, **MySQL** baza podataka suočila se s raznim protivnicima **MySQL** sustava organiziranja podataka jer su joj nedostajale neke osnovne funkcije definirane SQL standardom. Naime, **MySQL** baza je optimizirana kako bi bila brza nauštrb funkcionalnosti. Nasuprot tome, vrlo je stabilna i ima dobro dokumentirane module i ekstenzije te podršku od brojnih programskih jezika: **PHP**, **Java**, **Perl**, **Python**...

MySQL baze su relacijskog tipa, koji se pokazao kao najbolji način skladištenja i pretraživanja velikih količina podataka i u suštini predstavljaju osnovu svakog informacijskog sustava, tj. temelj svakog poslovog subjekta koji svoje poslovanje bazira na dostupnosti kvalitetnih i brzih informacija.

MySQL i **PHP** su osvojili veliki dio tržišta jer su open source i besplatni za koristiti.

Osnovi pojmovi o projektiranju baza podataka

Prije upuštanja u rad sa bilo kojim **DBMS** sistemom, pa tako i sa **MySQL**-om potrebno je dizajnirati odgovarajući izgled baze podataka, odnosno napraviti shemu baze, koja se u kasnjem postupku prevodi u određen broj tablica koje se koriste za pohranjivanje podataka.

Osnovi element koji se pohranjuje u bazi naziva se **entitet**. **Entitet** može biti bilo što: osoba, neki objekt, događaj, služba u nekoj organizaciji i sl. dakle stvari iz stvarnog života o kojima želimo čuvati informacije.

Drugi važan pojam u teoriji baza podataka jeste **relacija**. Kao što u stvarnom životu postoje određeni međusobni odnosi između dvije ili više osoba, događaja i sl. tako se i u bazama podataka mogu pojaviti određeni odnosi ili relacije između raznih entiteta, koji se na odgovarajući način predstavljaju unutar same baze.

Prema vrsti, relacije se mogu podijeliti na relacije **jedan prema jedan, jedan prema više odnosno više prema jedan te više prema više.**

ER dijagram

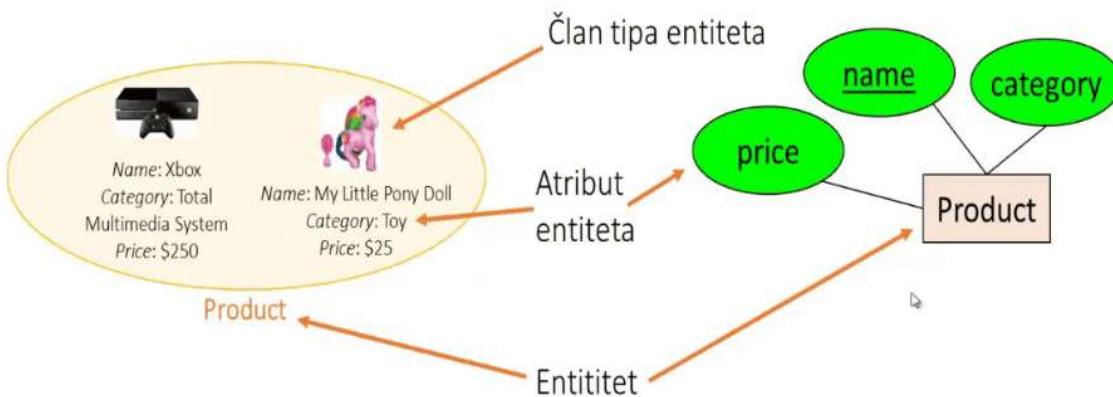
Model entiteta i veza nekog sustava, izražavamo preko entiteta, atributa i veze pomoću dijagrama nazvanog **ER dijagram (Entity Relationship)**.

Velika prednost ER dijagrama jeste u tome što se lako crtaju i razumiju.

Dijagram sadrži tri osnovne konstrukcije:

Entitete, Atribute i Veze.

Postoji više rješenja problema – neka su optimalnija od drugih. Različiti su načini prikaza i ovise od programa u kojem se kreiraju.



Upoznavanje sa ER modelima

Peter Chen razvio je ER dijagram 1976. godine. ER model je stvoren da pruži jednostavan i razumljiv model za predstavljanje strukture i logike baza podataka. Od tada se razvio u varijacije kao što su **Enhanced ER Model** i **Objekt Relationship Model**.

Relacijski model entiteta je model za identificiranje entiteta koji će biti predstavljeni u bazi podataka i prikaz načina na koji su ti entiteti povezani. ER podatkovni model specificira razvojnu šemu koja grafički predstavlja cjelokupnu logičku strukturu baze podataka.

Dijagram odnosa entiteta (engl. Entity Relationship Diagram) objašnjava odnos između entiteta prisutnih u bazi podataka. ER modeli koriste se za modeliranje objekata iz stvarnog svijeta poput osobe, automobila ili tvrtke i odnosa između tih objekata iz stvarnog svijeta. Ukratko, ER dijagram je strukturni format baze podataka.

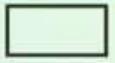
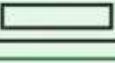
Zašto koristiti ER dijagrame u DBMS-u?

- ER dijagrami se koriste za predstavljanje ER modela u bazi podataka, što ih čini lakim za pretvaranje u relacije (tablice).
- ER dijagrami pružaju svrhu modeliranja objekata u stvarnom svijetu što ih čini izrazito korisnima.

- ER dijagrami ne zahtijevaju nikakvo tehničko znanje niti hardversku podršku.
- Ovi dijagrami su vrlo lako razumljivi i lako ih je izraditi čak i neiskusnom korisniku.
- Daje standardno rješenje za logičnu vizualizaciju podataka.

Simboli korišteni u ER modelu

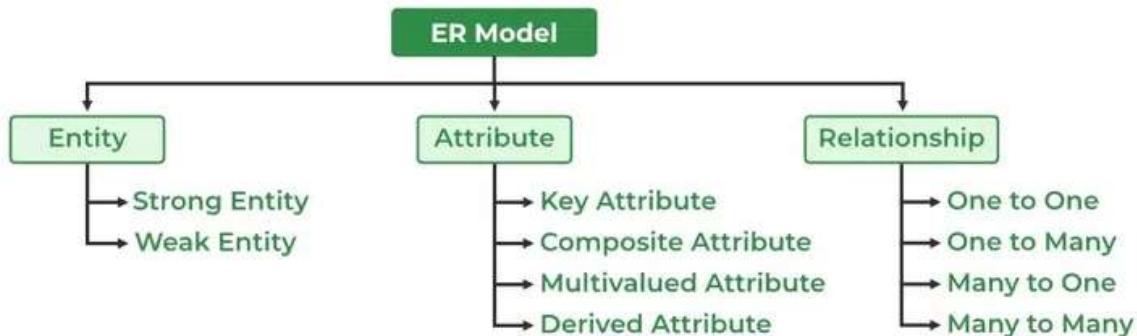
- **Pravokutnici:** pravokutnici predstavljaju entitete u ER modelu.
- **Elipse:** Elipse predstavljaju atribute u ER modelu.
- **Dijamant:** dijamanti predstavljaju odnose među entitetima.
- **Linije:** Linije predstavljaju atribute entitetima i skupovima entiteta s drugim vrstama odnosa.
- **Dvostruka elipsa:** Dvostrukе elipse predstavljaju atribute s više vrijednosti
- **Dvostruki pravokutnik:** Dvostruki pravokutnik predstavlja slab entitet.

Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

Simboli koji se koriste u ER dijagramu

Komponente ER dijagrama

ER model sastoji se od entiteta, atributa i odnosa među entitetima u sustavu baze podataka.



Komponente ER dijagrama

Entitet

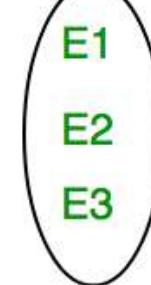
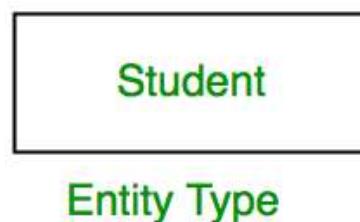
Entitet može biti objekt s fizičkim postojanjem – određena osoba, automobil, kuća ili radnik – ili može biti objekt s konceptualnim postojanjem – firma, posao ili sveučilišni tečaj.

Postoje dvije vrste entiteta

1. Opipljivi entitet – koji se može dodirnuti kao automobil, osoba itd.

2. Neopipljivi entitet – koji se ne može dodirnuti kako zrak, bankovni račun itd.

Skup entiteta: Entitet je objekt vrste entiteta, a skup svih entiteta naziva se skup entiteta. Na primjer, E1 je entitet koji ima tip entiteta Student, a skup svih studenata naziva se skup entiteta. U ER dijagramu, tip entiteta predstavljen je kao:



Skup entiteta

Možemo predstaviti skup entiteta u ER dijagramu, ali ne možemo predstaviti entitet u ER dijagramu jer je entitet red i stupac u relaciji, a ER dijagram je grafički prikaz podataka.

1. Snažan entitet

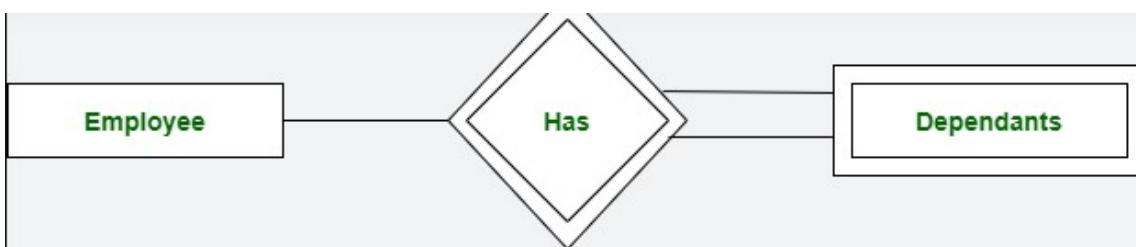
Snažan entitet je vrsta entiteta koji ima ključni atribut. Jaki entitet ne ovisi o drugom entitetu u šemi. Ima primarni ključ koji pomaže u njegovoј jedinstvenoj identifikaciji, a predstavljen je pravokutnikom. Oni se nazivaju tipovi jakog entiteta.

2. Slab entitet

Tip entiteta ima ključni atribut koji jedinstveno identificira svaki entitet u skupu entiteta. Ali postoji neka vrsta entiteta za koju se ključni atributi ne mogu definirati. To se naziva slabim tipovima entiteta.

Na primjer, firma može pohraniti podatke o uzdržavanim osobama (roditeljima, djeci, supružniku) radnika. Ali uzdržavane osobe ne mogu postojati bez zaposlenih. Dakle, Uzdržavani će biti Slab tip entiteta , a Radnik će biti tip identifikacijski entiteta za uzdržavane, što znači da je Snažan tip entiteta.

Slab tip entiteta predstavljen je dvostrukim pravokutnikom. Sudjelovanje slabih tipova entiteta uvijek je ukupno. Odnos između slabog tipa entiteta i njegovog identificirajućeg jakog tipa entiteta naziva se identifikacijskim odnosom i predstavljen je dvostrukim rombom.



Jak entitet i slab entitet

Atributi

Atributi su svojstva koja definiraju tip entiteta. Na primjer, `Roll_No`, `Name`, `DOB`, `Age`, `Address` i `Mobile_No` su atributi koji definiraju tip entiteta `Student`. U ER dijagramu, atribut je predstavljen elipsom.



Atribut

1. Ključni atribut

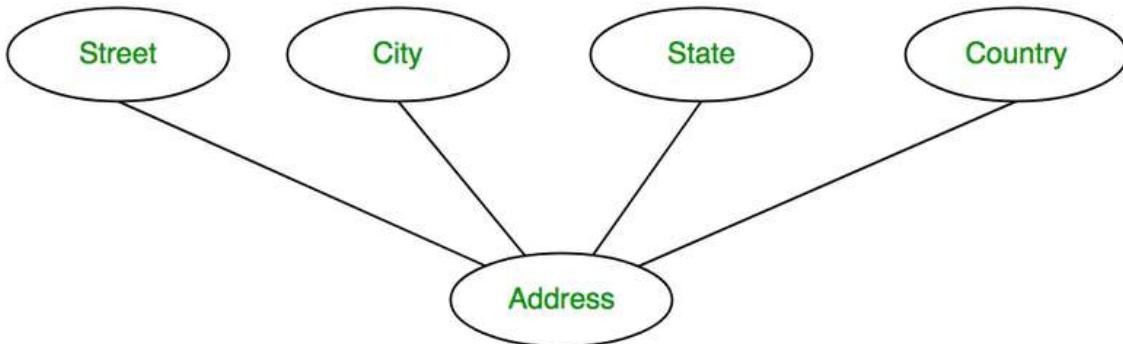
Atribut koji jedinstveno identificira svaki entitet u skupu entiteta naziva se ključni atribut. Na primjer, `Roll_No` bit će jedinstven za svakog učenika. U ER dijagramu, ključni atribut je predstavljen elipsom s linijom ispod nje.



Ključni atribut

2. Kompozitni atribut

Atribut sastavljen od mnogih više atributa naziva se složeni (kompozitni) atribut. Na primjer, **Address** atribut studenta tipa entiteta sastoji se od **Street**, **City**, **State**, **Country**. U ER dijagramu, kompozitni atribut je predstavljen ovalom koji se sastoji od elipsi.



Kompozitni atribut

3. Atribut s više svojstava

Atribut koji se sastoji od više od jedne vrijednosti za dani entitet. Na primjer, **Phone_No** (može biti više od jednog za određenog učenika). U ER dijagramu, atribut s više vrijednosti predstavljen je dvostrukom elipsom.



Atribut s više svojstava

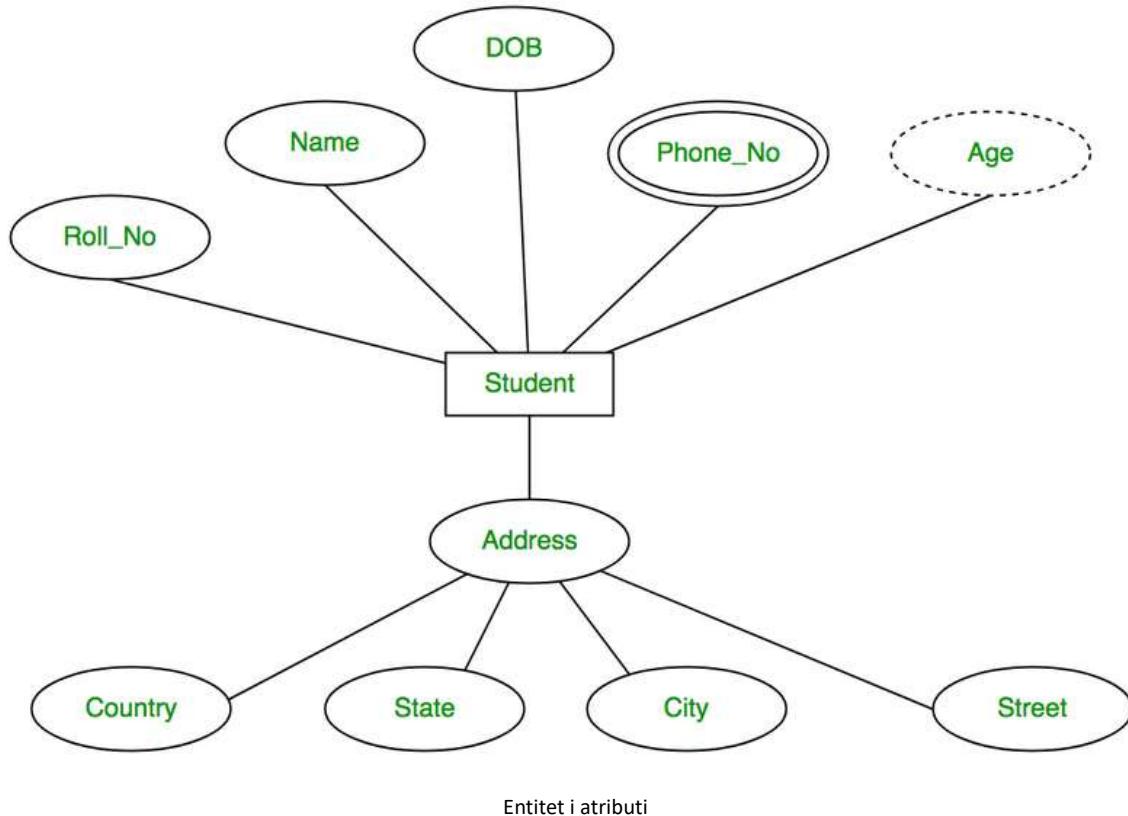
4. Derivirani atribut

Atribut koji se može derivirati (izvesti) iz drugih atributa tipa entiteta poznat je kao derivirani atribut. npr.; **Age** (može se derivirati iz DOB-a). U ER dijagramu, derivirani atribut je predstavljen isprekidanim elipsom.



Derivirani (izvedeni) atribut

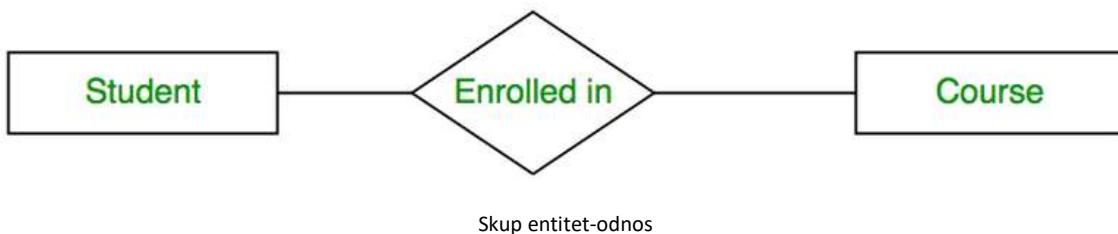
Kompletan entitet tipa Student sa svojim atributima može se predstaviti kao:



Entitet i atributi

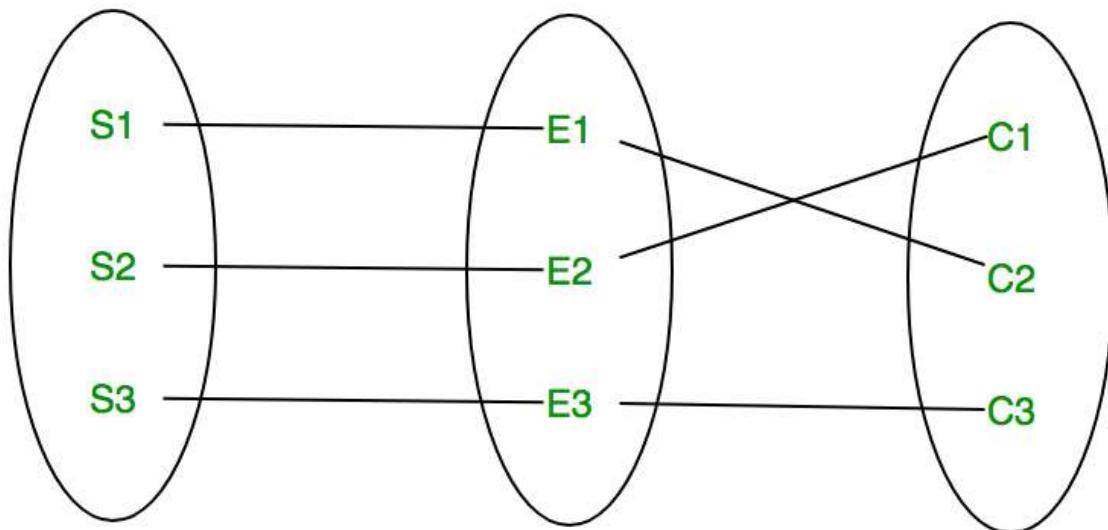
Tip odnosa i skup odnosa

Tip odnosa (engl. Relationship Type) predstavlja povezanost između tipova entiteta. Na primjer, 'Upisan' je tip odnosa koji postoji između tipa entiteta Student i Tečaj (Course). U ER dijagramu, tip odnosa je predstavljen rombom i povezuje entitete linijama.



Skup entitet-odnos

Skup odnosa istog tipa skup odnosa (engl. relationship set). Sljedeći skup odnosa prikazuje S1 kao upisanog u C2, S2 kao upisanog u C1 i S3 kao registriranog u C3.

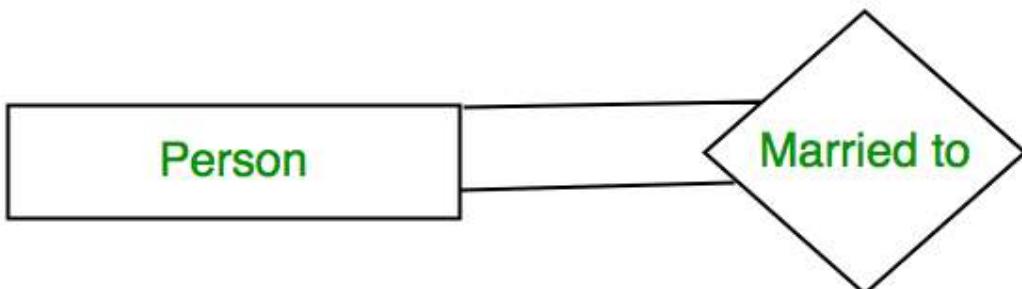


Skup odnosa (engl. Relation Set)

Stupanj skupa odnosa

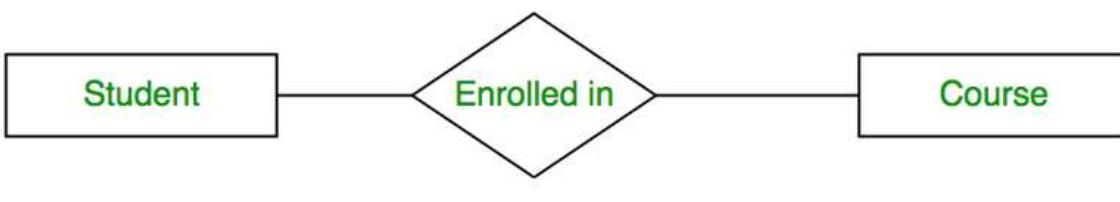
Broj različitih skupova entiteta koji sudjeluju u skupu odnosa (relationship set) naziva se stupanj skupa odnosa (engl. degree of a relationship set).

1. Unarni odnos: Kada postoji samo JEDAN skup entiteta koji sudjeluje u relaciji, odnos se naziva unarni odnos. Na primjer, jedna osoba je u braku sa samo jednom osobom.



Unarni odnos

2. Binarni odnos: Kada postoje DVA skupa entiteta koji sudjeluju u odnosu, odnos se naziva binarni odnos. Na primjer, student je upisan na kolegij (engl. Course).



Binarni odnos

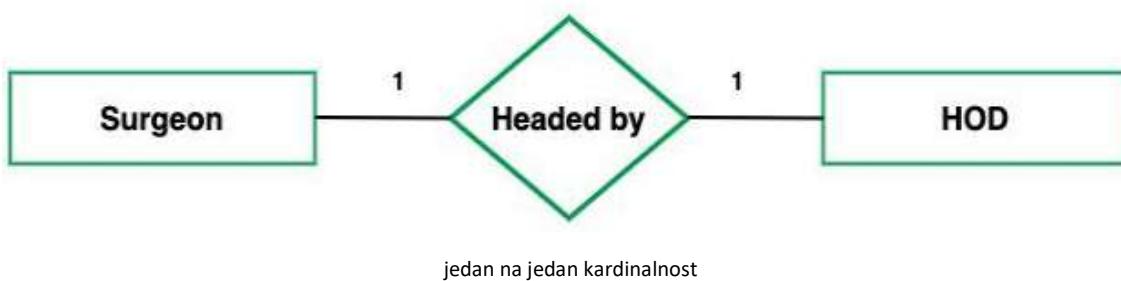
3. Ternarni odnos: Kada postoji n skupova entiteta koji sudjeluju u relaciji, odnos se naziva n-arni odnos.

Kardinalnost

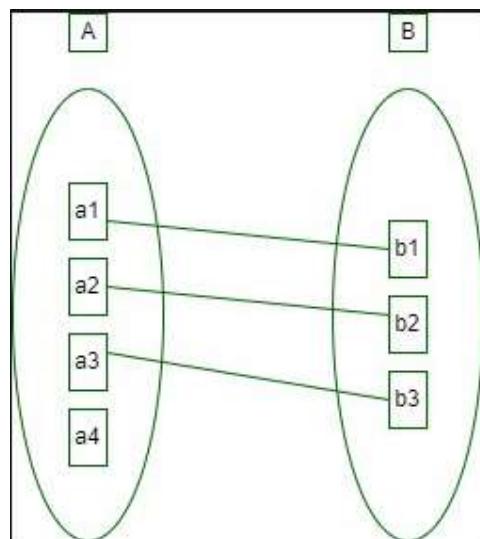
Broj puta kada entitet skupa entiteta sudjeluje u skupu odnosa poznat je kao kardinalnost . Kardinalnost može biti različitih vrsta:

1. Jedan-na-jedan: Kada svaki entitet u svakom skupu entiteta može sudjelovati samo jednom u odnosu, kardinalnost je jedan-na-jedan. Prepostavimo da se muškarac može oženiti jednu ženu, a žena se udati za jednog muškarca. Dakle, odnos će biti jedan na jedan.

ukupan broj tablica koje se mogu koristiti u ovome je 2.



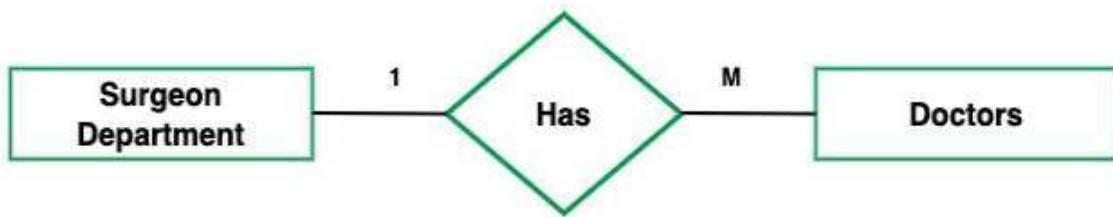
Korištenjem skupova, može se predstaviti kao:



Postavite prikaz jedan-na-jedan

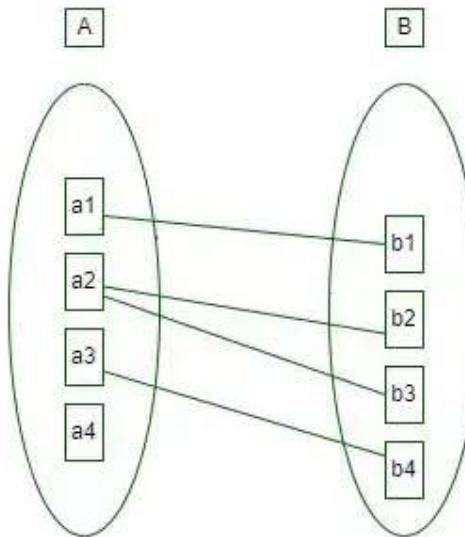
2. Jedan-prema-više: I u mapiranju jedan-prema-više gdje svaki entitet može biti povezan s više od jednog entiteta, a ukupan broj tablica koje se mogu koristiti u ovome je 2. Prepostavimo da jedan odjel kirurgije može primiti mnogo liječnika. Dakle, kardinalnost će biti 1 prema M. To znači da jedan odjel ima mnogo liječnika.

ukupan broj tablica koje se mogu koristiti je 3.



jedan prema više kardinalnosti

Koristeći skupove, kardinalnost jedan prema više može se predstaviti kao:



Postavite reprezentaciju jedan prema više

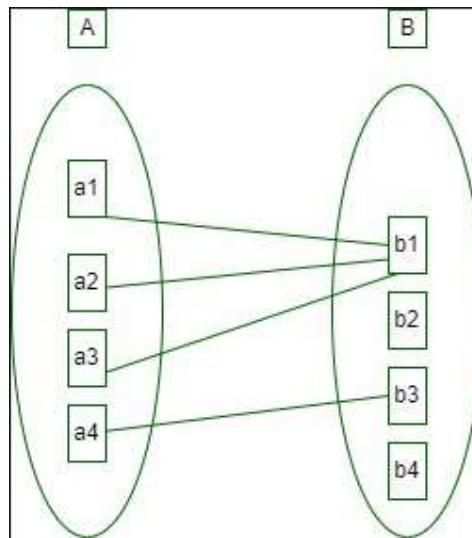
3. Više-na-jedan: Kada entiteti u jednom skupu entiteta mogu sudjelovati samo jednom u skupu odnosa, a entiteti u drugim skupovima entiteta mogu sudjelovati više od jednom u skupu odnosa, kardinalnost je više prema jedan. Pretpostavimo da student može slušati samo jedan kolegij, ali jedan kolegij može slušati mnogo studenata. Dakle, kardinalnost će biti n do 1. To znači da za jedan kolegij može biti n studenata, ali za jednog studenta, postojat će samo jedan kolegij.

Ukupan broj tablica koje se mogu koristiti u ovome je 3.



više na jedan kardinalnost

Korištenjem skupova, može se predstaviti kao:

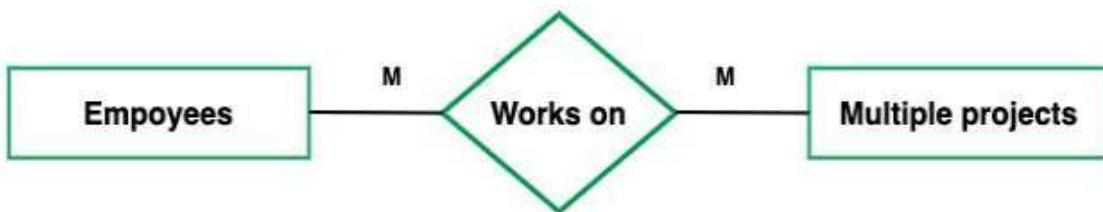


Skup reprezentacije više prema jednom

U ovom slučaju, svaki student pohađa samo 1 kolegij, ali 1 kolegij je slušalo mnogo studenata.

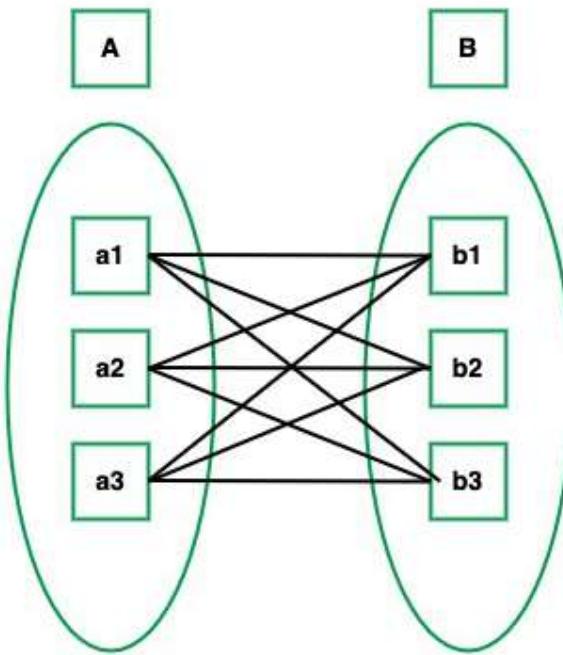
4. Više-prema-više: Kada entiteti u svim skupovima entiteta mogu više puta sudjelovati u odnosu, kardinalnost je više prema više. Pretpostavimo da student može pohađati više od jednog predmeta i da jedan predmet može pohađati mnogo studenata. Dakle, odnos će biti više prema više.

ukupan broj tablica koje se mogu koristiti u ovome je 3.



kadinalnost više prema više

Korištenjem skupova, može se predstaviti kao:



Reprezentacija skupa više-prema-više

U ovom primjeru, student S1 je upisan u C1 i C3, a kolegij C3 su upisali S1, S3 i S4. Dakle, to su odnosi mnogo-prema-više.

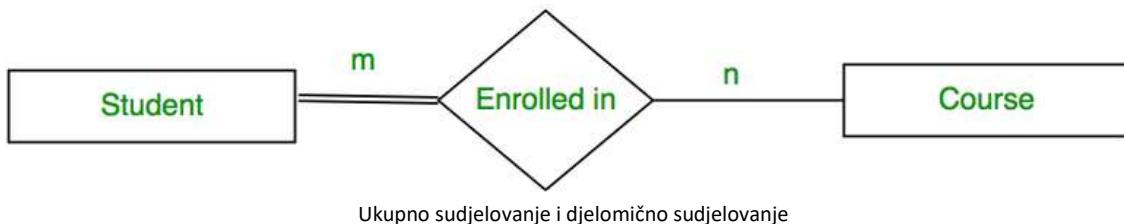
Ograničenje sudjelovanja

Ograničenje sudjelovanja (engl. Participation Constraint) primjenjuje se na entitet koji sudjeluje u skupu odnosa.

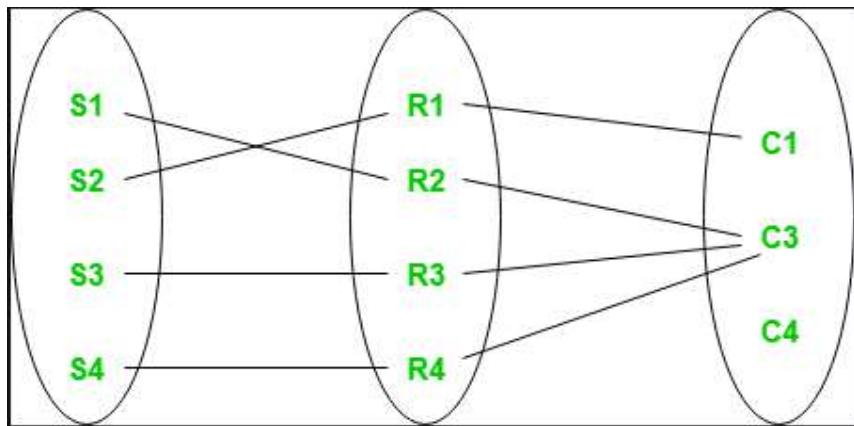
1. Ukupno sudjelovanje – Svaki entitet u skupu entiteta mora sudjelovati u odnosu. Ako svaki student mora upisati predmet, sudjelovanje studenata bit će ukupno. Ukupno sudjelovanje prikazano je dvostrukom linijom u ER dijagramu.

2. Djelomično sudjelovanje – Entitet u skupu entiteta može, ali i NE mora sudjelovati u odnosu. Ukoliko neke kolegije nitko od polaznika ne upiše, sudjelovanje u kolegiju bit će djelomično.

Dijagram prikazuje skup odnosa 'Upisan u' (engl. 'Enrolled in') sa skupom entiteta studenta koji ima ukupno sudjelovanje i skupom entiteta predmeta koji ima djelomično sudjelovanje.



Koristeći Set, može se predstaviti kao,



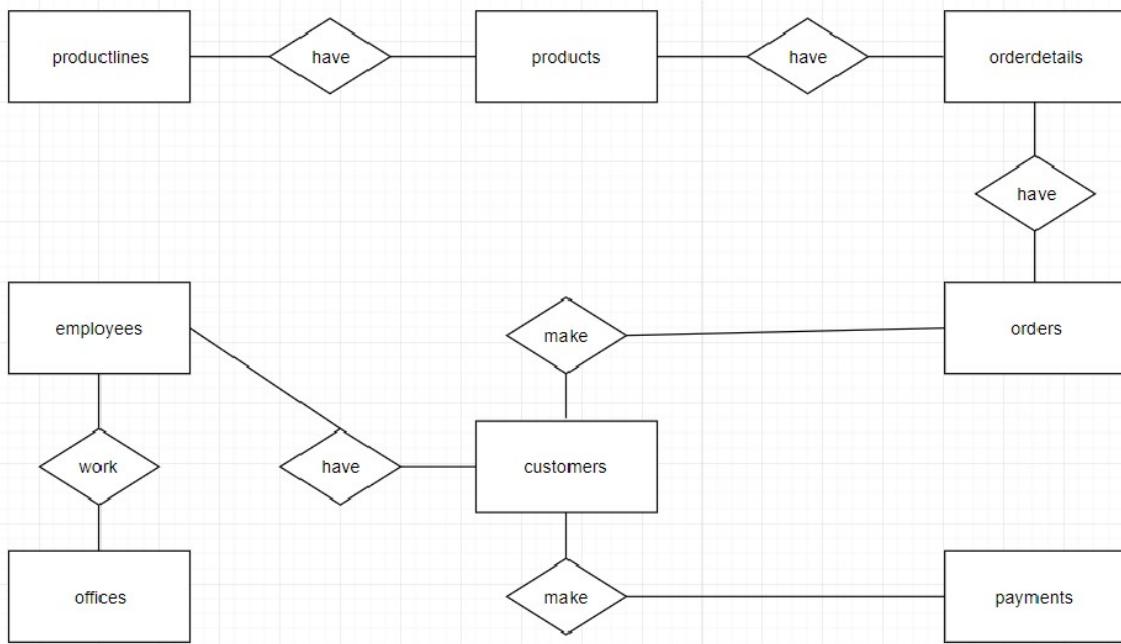
Set prikazuje ukupno sudjelovanje i djelomično sudjelovanje

Svaki student u skupu entiteta Student sudjeluje u odnosu, ali postoji kolegij C4 koji ne sudjeluje u odnosu.

Kako nacrtati ER dijagram?

- Prvi korak je identificiranje svih entiteta, njihovo postavljanje u pravokutnik i označavanje u skladu s tim.
- Sljedeći korak je identificirati odnos između njih i postaviti ih u skladu s tim pomoću dijamanta i osigurati da Odnosi nisu međusobno povezani.
- Ispravno priložite atribute entitetima.
- Uklonite suvišne entitete i odnose.
- Dodajte odgovarajuće boje kako biste istaknuli podatke prisutne u bazi podataka.

ER model se koristi za modeliranje logičkog prikaza sustava iz perspektive podataka koji se sastoji od ovih simbola:



ER dijagram (Vježba 1)

Kreirajte ER dijagram za poslovanje jedne videoteke.

Videoteka članovima izdaje članske iskaznice te se na temelju članskog broja osoba identificira kako bi mogla posuditi filmove.

Filmovi su složeni po žanrovima.

Videoteka ima definiran cjenik za izdavanje hit filma, film koji nije hit te starog filma.

Jedan film može biti na DVD-u ili BlueRay-u.

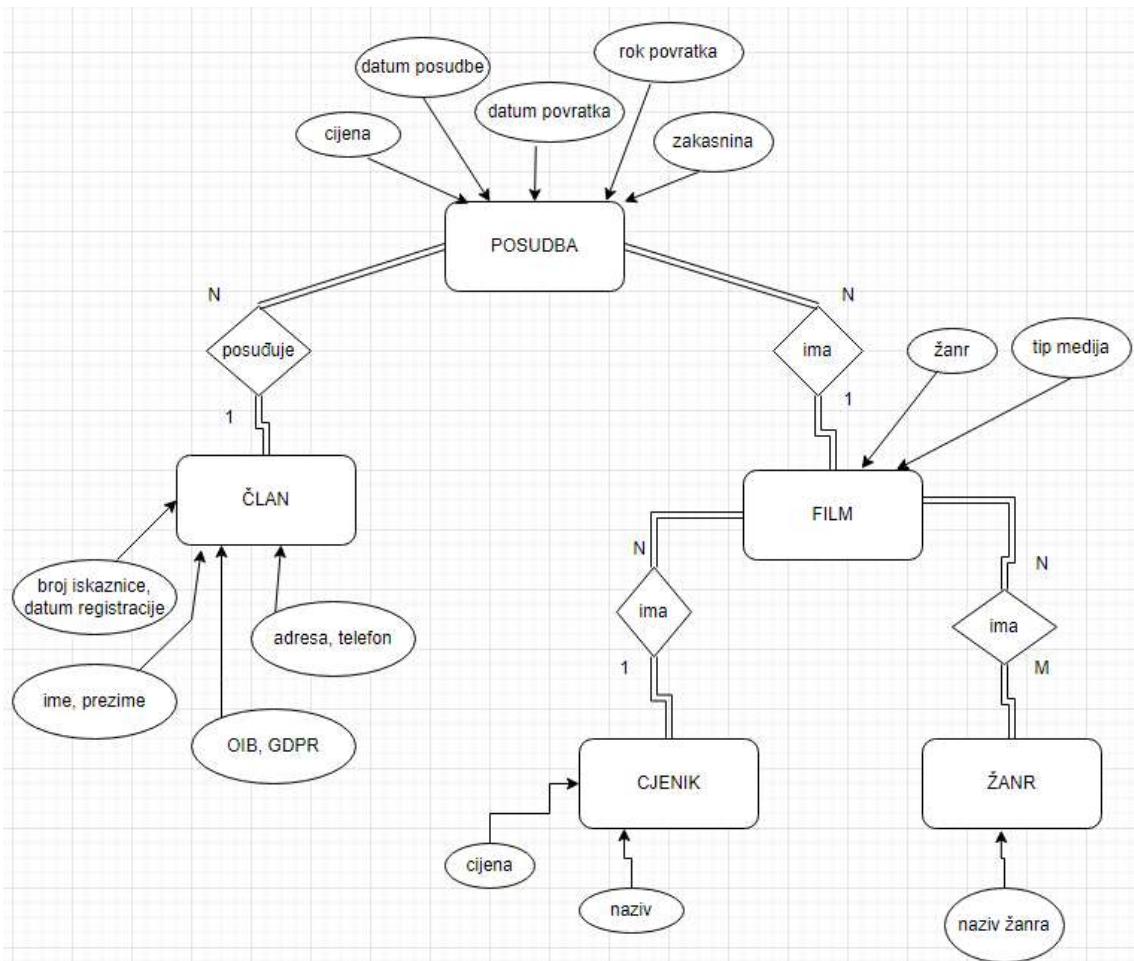
Film se posuđuje na rok od jednog dana i ako ga član ne vratí u navedeno vrijeme, zaračunava mu se zakasnina.

Ovdje imamo entitet **Član**. **Film** je drugi entitet. Žanr možemo pisati uz film kao atribut ili može biti zasebni entitet. Žanru se može promijeniti ime. Npr. horor se zove strava i užas. Trebamo znati ima li film jedan žanr ili više. Dakle bolje je da **Žanr** bude entitet. **Cjenik** za izdavanje hit filma i cjenik koji je za stari film. Pitanje je hoćemo li cjenik vezati uz filmove ili ne. Ako vežemo cijenu za film, a kasnije se cijena promjeni imamo mogući problem. Ako u posudbi imamo zapisanu cijenu onda izbjegnemo problem. Ali tada imamo zapisanu cijenu na 2 mesta. Pogledajmo primjer: Imamo film čija je cijena 2 € i sutra odlučimo dići cijenu na 3 €. Pitanje je što sa onima što su posuđivali filmove prije? Zaključak je da trebamo neki price book. U izvještaju želimo dobiti informaciju koliko je novaca od posudbe određenog filma. Cijenu ne možemo držati samo uz film. Moramo je držati u entitetu **Posudba** - od do i koji film, te cijena. Zakasnину trebamo račun po cijeni filma ili je trebamo mijenjati. O tome ovisi

kako ćemo napraviti. Dane su specifikacije ali nisu razrađene do kraja. Odlučit ćemo se da je zakasnina jednaka cijeni filma. Nećemo imati više istih filmova.

Film može biti DVD ili BlueRay, to će biti atribut filma. Ako je član vratio film nakon više dana obračunava se zakasnina.

Veza između Član i Film: 1 član može posuditi više filmova ali 1 film može posuditi više članova (naravno ne istovremeno tj. ne isti dan). **Član i Film** nisu direktno vezani nego preko **Posudbe**. Svaki entitet treba imati primarni ključ. Pogledajmo kako će se reflektirati veza između **Člana i Filma** unutar **Posudbe**. Morat ćemo imati strane ključeve. U **Posudbi** će veza između **Člana i Filma** funkcionišati između tako da ćemo imati stupac koji je primarni ključ u **Članu** a on postaje strani ključ u **Posudbi** i primarni ključ u **Filmu** a on isto postaje strani ključ u **Posudbi**. Imamo jednu jedinstvenu **Posudbu** koja se sastoji od primarnog ključa, od stranog ključa koji je primarni ključ **Člana** i od stranog ključa koji je primarni ključ **Filma**. Na taj način ćemo stvoriti vezu između **Člana i Filma** i dobiti many-to-many vezu. Mogu postojati tri strana ključa, tada obično jedan strani ključ nije mandatory nego optional.



Nacrtano s <https://app.diagrams.net>

Film ima neku cijenu u **Cjeniku**. Vežemo **Cjenik za Film**. Film ima neki **Žanr** i vežemo ih. Trenutno to je jedan žanr. 1 član može imati više posudbi (1-N). Po posudbi možemo imati više filmova (N-1). Ako

posudimo više filmova, bit će zapis u Posudbi sa više redova jer je moguće vratiti filmove pojedinačno. Bilježimo datum posudbe i datum vraćanja.

Jedan Film može imati jednu cijenu u Cjeniku ali jedna cijena može biti na više filmova (N -1). Veza između filmova i žanra je many-to-many (N-N).

U idućem koraku definirat ćemo attribute za svaki entitet. Crtamo ih u elipsi. Za **Član** su to: broj iskaznice, datum registracije; ime, prezime; OIB, GDPR; adresa, telefon. **Posudba** ima: cijenu; datum posudbe; datum povratka; rok povratka i zakasnina. **Film** ima žanr i tip medija. **Cjenik** cijenu i naziv. **Žanr** ima naziv žanra.

Imamo Film vezan za cjenik. Potencijalan problem imamo sa cijenom jer Film A košta 3 € 2023. godinu. 2024. digli smo cijenu na 4 € (npr. zbog inflacije). U **Cjeniku** napravimo promjenu, u **Posudbi** imamo relaciju prema **Filmu**, **Film** ima relaciju prema **Cjeniku**. Kada izvučemo podatke iz posudbe vidimo krivu cijenu i vidimo da je određeni korisnik platio višu cijenu. To trebamo rješiti. Promjenom Cjenika trebamo zadržati podatke u Posudbi.

MariaDB I MySQL: usporedba



MariaDB

MariaDB je open-source fork MySQL-a stvoren 2009. MariaDB je poboljšana verzija MySQL-a koja je kompatibilna s prethodnim verzijama. Dolazi s raznim ugrađenim svojstvima i mnogim sigurnosnim i izvršnim poboljšanjima koja nedostaju u MySQL-u. MariaDB podržava iste svojstva kao i MySQL, ali nudi i dodatne.

Zamijenivši MySQL, MariaDB je postao besprijekoran proces za većinu aplikacija i CMS-a (engl. (engl. Content management system), posebno za WordPress. Postojeći softver, iz popularnog CMS-a alata do aplikacija kao što je phpMyAdmin radi odlično, a stvarni podaci mogu se izvesti/vesti bez ikakvih promjena.

MySQL

MySQL je relacijska baza podataka (RDBMS) koja se prvi put pojavila 1995. U to je vrijeme Microsoft i Oracleova vlasnička rješenja dominirala tržištem.

MySQL je sustav za upravljanje relacijskom bazom podataka otvorenog koda s korijenima u SQL-u ili jeziku strukturiranih upita MySQL je među najkorištenijim bazama podataka u svijetu, ali nipošto nije jedina. Razvijen u C/C++, MySQL je besplatan i otvorenog koda te je napravio značajan napredak.

Međutim, tokom Oracleove akvizicije [Sun Microsystems](#), neki od viših inženjera koji su radili na razvoju MySQL-a smatrali su da postoji sukob interesa između MySQL-a i Oracleove komercijalne baze podataka – [Oracle Database Server](#).

Ključne razlike

Rukovanje JSON podacima

MariaDB i MySQL podržavaju JSON format i izvršavaju mnoge iste funkcije. Međutim, MySQL pohranjuje JSON izvještaje kao binarne objekte, dok ih MariaDB pohranjuje u obliku stringova tj. kao [LONGTEXT](#). MySQL i MariaDB također ne podržavaju sve JSON funkcije. MariaDB podržava [JSON_QUERY](#)

i `JSON_EXISTS`, dok MySQL ne podržava. Slično, MySQL podržava originalni JSON tip podataka `JSON_TABLE` koji MariaDB ne podržava.

Funkcionalnost

Budući da su njihovi razvojni putovi bili različiti, MariaDB i MySQL sada nude malo drugačiju funkcionalnost.

Na primjer, MySQL nudi dinamičke stupce (polja) koji vam omogućavaju definiranje višestrukih vrijednosti podataka u jednom stupcu i izmjenu stupca pomoću funkcija. Dinamički stupci omogućavaju maskiranje podataka radi zaštite osjetljivih informacija.

Nasuprot tome, MariaDB podržava nevidljive stupce u prikazima baze podataka. Nevidljivi stupci nisu navedeni kada korisnik izvodi naredbu `SELECT` ili traži vrijednost u naredbi `INSERT`.

Autentifikacija korisnika

MySQL ima komponentu `validate_password` koju možete koristiti za povećanje sigurnosti lozinke.

MariaDB to nema prema zadanim postavkama, ali ali nudi tri pluginu za validator. Možete koristiti ove plugine za povećanje zaštite lozinki. U verziji 10.4, MariaDB je uvela [plugin za provjeru autentičnosti ed25519](#) kako bi zamijenila svoju prethodno korištenu provjeru autentičnosti SHA-1. To pomaže u autentifikaciji korisnika i sigurnom pohranjivanju lozinki.

Enkripcija

MySQL i MariaDB šifriraju (enkriptiraju) podatke u mirovanju i u prijenosu.

MySQL omogućava admin korisnicima da konfiguriraju i šifriraju redo i undo logove (zapise dnevnika), dok ne šifrira prostor privremenih tablica ili binarnih logova. Redo logovi osiguravaju trajnost transakcija i sprečavaju zapisivanje prljavih stranica na disk u trenutku kvara. Undo log pohranjuje verziju podataka prije nego što se transakcija dogodi, koja se može koristiti za vraćanje.

S druge strane, MariaDB podržava binarnu enkripciju logova i privremenu enkripciju tablice.

Udruživanje niti

Udruživanje niti (engl. Thread pooling) omogućuje bazi podataka da optimizira svoje resurse uparivanjem novih veza s već postojećim nitima. Veliki kapacitet udruživanja niti ključan je za aplikacije koje žele skalirati i paralelno opsluživati hiljade korisnika. Kada imamo više programskih niti, govorimo o višeprogramskom radu koji se izvršava paralelno.

MariaDB uključuje udruživanje niti u svom pluginu za udruživanje niti (engl. thread pool plugin), koji je dio community izdanja. MariaDB nudi mogućnost upravljanja s preko 200.000 veza odjednom.

MySQL ima plugin za udruživanje niti na svojoj enterprise verziji. Međutim, ne može rukovati sa toliko veza kao MariaDB.

Ostale razlike

MariaDB ima visoku razinu kompatibilnosti i podržava PL/SQL, za razliku od MySQL-a. PL/SQL je kombinacija SQL-a i proceduralnih mogućnosti programskih jezika. Razvila ga je Oracle Corporation početkom 90-ih kako bi poboljšala mogućnosti SQL-a. PL/SQL je jedan od tri ključna programska jezika

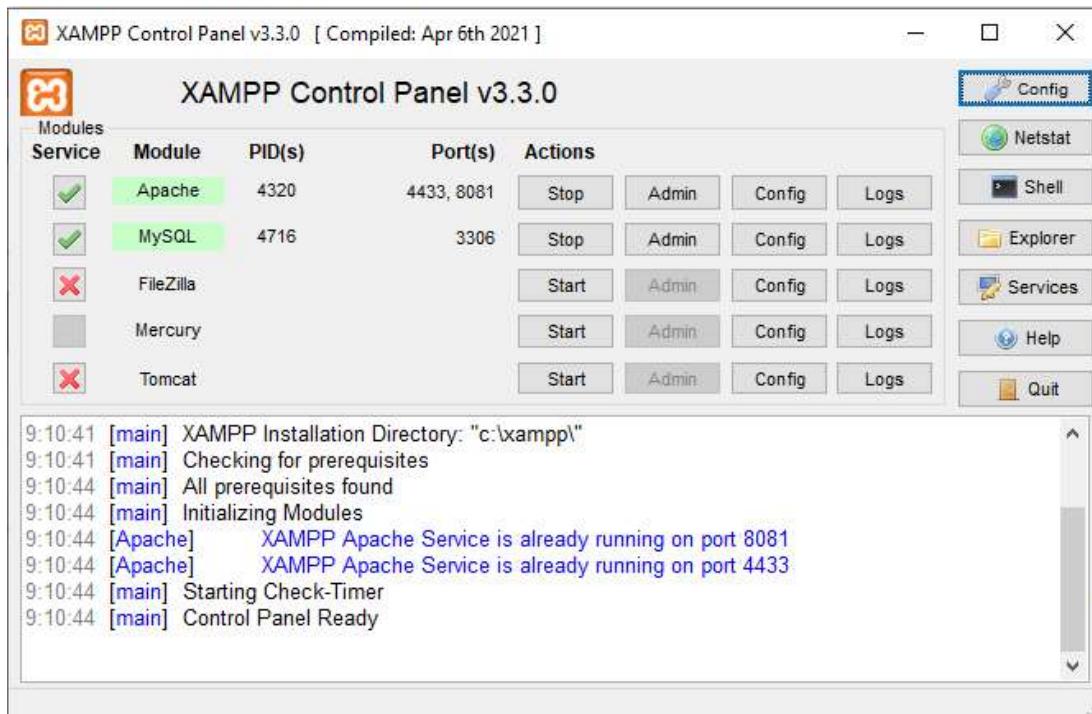
ugrađena u Oracle Database, zajedno sa samim SQL-om i Javom. MariaDB je nešto brži od MySQL-a u replikaciji i postavljanju upita.

MySQL baze podataka koriste InnoDB i AES za šifriranje podataka u mirovanju. MariaDB podržava privremenu enkripciju dnevnika i binarnu enkripciju dnevnika.

Programi za konekciju s bazama podataka

phpMyAdmin

Po default-u XAMPP instalira kao bazu MariuDB a ne MyPHP. MariuDB treba pokrenuti tj. MySQL.



Kada pokrenemo MySQL, potrebno jer pokrenuti phpMyAdmin sa adresom localhost:8081/phpmyadmin/

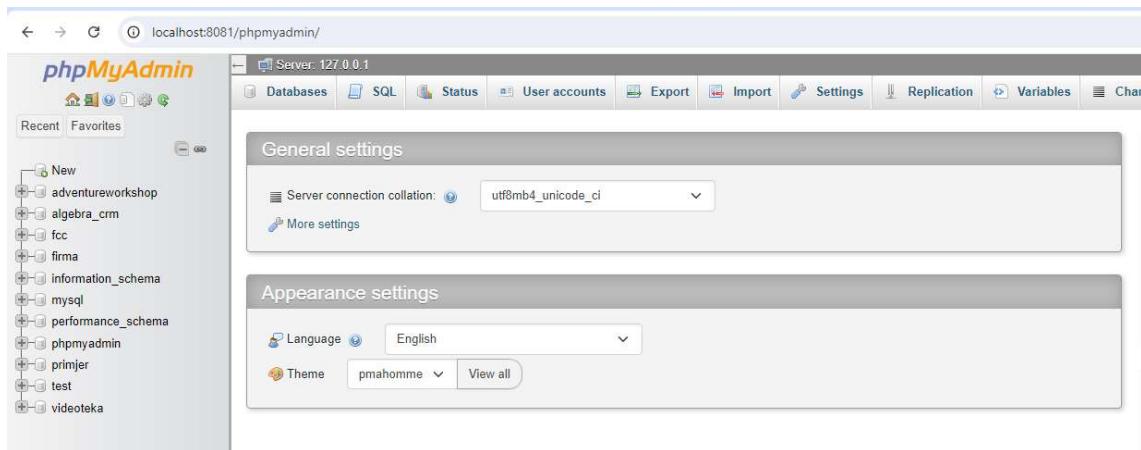
The screenshot shows the 'Database server' configuration page. It lists the following details:

- Server: 127.0.0.1 via TCP/IP
- Server type: MariaDB
- Server connection: SSL is not being used
- Server version: 10.4.32-MariaDB - mariadb.org binary distribution
- Protocol version: 10
- User: root@localhost
- Server charset: UTF-8 Unicode (utf8mb4)

Na početnoj stranici vidimo na koju bazu je spojen. Dakle upravo na MariaDB 10.4.32. Razlike između MariaDB i MySQL su minimalne jer je Oracle preuzeo MySQL i stavio ga pod licencu, ostala je Community Server verzija koja je besplatna ali ne zna se do kada. phpMyAdmin je aplikacija kojom se

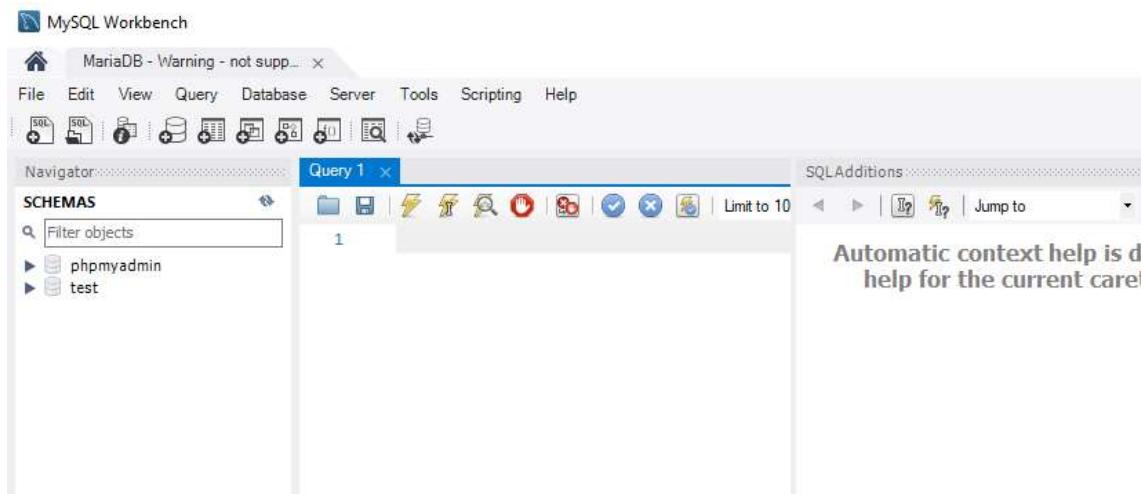
konektiramo na bazu. To je GUI koji omogućava lakše upravljanje bazom. phpMyAdmin je web aplikacija. Moguće se connectati na MySQL i na druge načine, npr. Javom.

Sistem upravljanja relacijskom bazom podataka (Relational database management system -RDBMS) je zbirka programa i mogućnosti koji IT timovima i drugima omogućavaju kreiranje (stvaranje), ažuriranje, administriranje i drugu interakciju s relacijskom bazom podataka. RDBMS pohranjuju podatke u obliku tablica, pri čemu većina komercijalnih sustava za upravljanje relacijskim bazama podataka koristi Structured Query Language (SQL) za pristup bazi podataka. Međutim, budući da je SQL izmišljen nakon početnog razvoja relacijskog modela, nije neophodan za korištenje RDBMS-a.

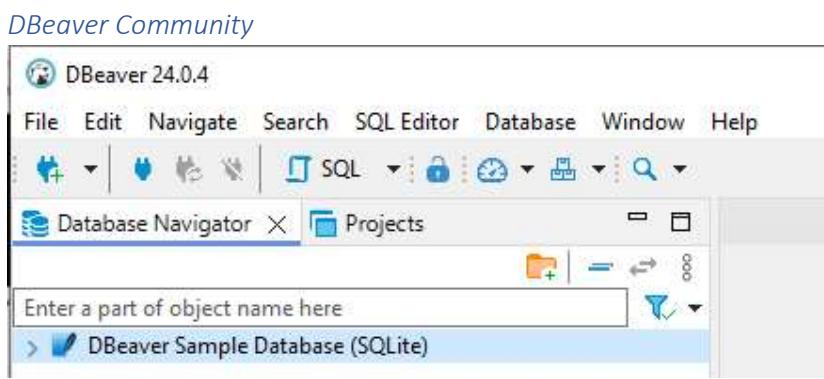
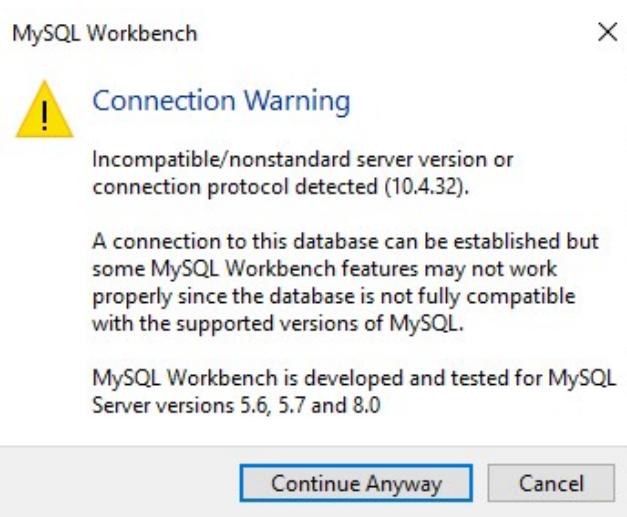
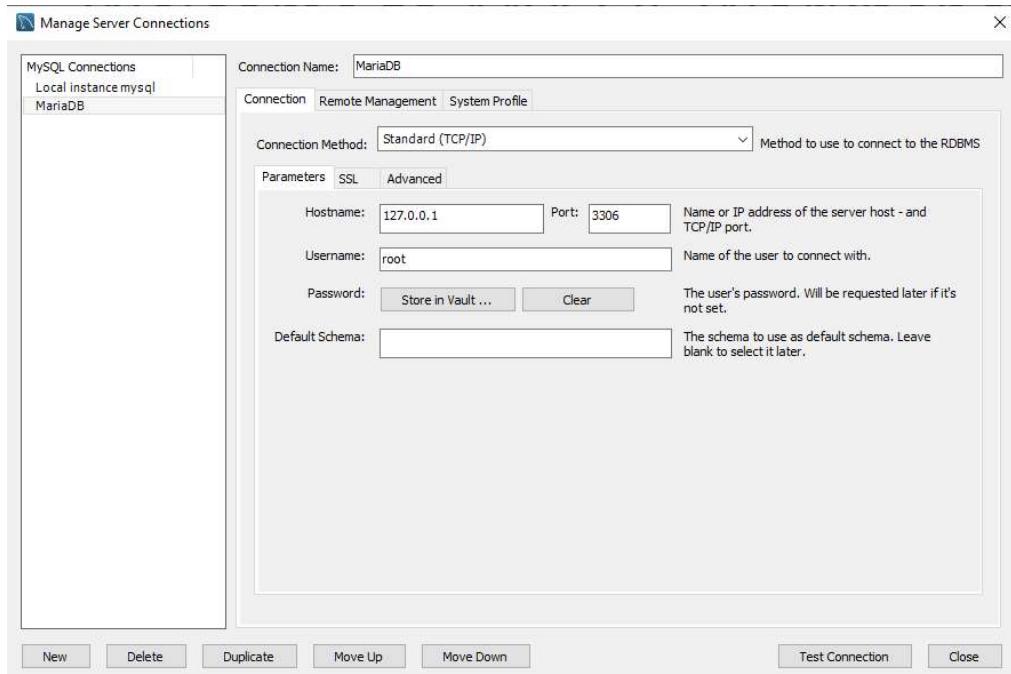


MySQL WorkBench

MySQL WorkBench je potrebno download-ovati i instalirati.



Mogući su problemi sa MariomDB.



Pokretanje XAMPP s MySQL-om

1. Zaustavite MySQL (koji je zapravo MariaDB) na upravljačkoj ploči XAMPP-a.
2. Potrebno je skinuti sa [MySQL community servera](#) zip arhivu (Windows 64 bit verziju). U ovom trenutku zadnja je 8.4.0. LTS.
3. Preimenovati `C:\xampp\mysql` u `C:\xampp\mariadb`
4. Raspakujte preuzetu zip arhivu u `C:\xampp\mysql`. Provjerite da li ste u poddirektorijima prisutni direktoriji `bin`, `include`, `lib` itd.
5. Kopirajte `C:\xampp\mariadb\bin\my.ini` u `C:\xampp\mysql\bin`
6. Otvorite `C:\xampp\mysql\bin\my.ini` u editoru i staviti komentar u red koji počinje s `key_buffer=u [mysqld]` odlomku.
7. Otvorite komandni red (cmd a ne PowerShell) i pokrenite sljedeće naredbe:

```
cd C:\xampp\mysql  
bin\mysqld --initialize-insecure  
start /b bin\mysqld  
bin\mysql -u root  
    CREATE USER pma@localhost;  
    SOURCE C:/xampp/phpMyAdmin/sql/create_tables.sql;  
    GRANT SELECT, INSERT, DELETE, UPDATE, ALTER ON phpmyadmin.* TO pma@localhost;  
    ALTER USER root@localhost IDENTIFIED WITH mysql_native_password BY '';  
    ALTER USER pma@localhost IDENTIFIED WITH mysql_native_password BY '';  
\q  
bin\mysqladmin -u root shutdown
```

8. Pokrenite Apache i MySQL na XAMPP upravljačkoj ploči.
9. Idite na <http://localhost/phpmyadmin> i provjerite je li vaš poslužitelj baze podataka sada prijavljen kao MySQL Community Server.

Alternativa je koristiti [WAMPServer](#) koji instalira i MySQL i MariaDB. Također omogućava različite verzije PHP-a, MySQL-a i MariaDB i njihovu fleksibilnu zamjenu.



Normalizacija

- Kako bismo mogli napraviti bazu podataka koja neće sadržavati kritične pogreške u dizajnu, pomaže nam normalizacija.
- Normalizacija je postupak organiziranja podataka s ciljem kreiranja učinkovite, pouzdane (sačuvan integritet baze podatka) i fleksibilne baze podataka.
- Postupak se temelji na matematički dokazanim tvrdnjama. To znači ako slijedimo normalizaciju, rezultat će biti dobivanje dobre baze podataka.
- Pri dizajniranju baze podataka potrebno je donijeti odluke o tome koje entitete načiniti, koje će atributi ti entiteti sadržavati, te kakve relacije između entiteta treba uspostaviti.
- Normalizacija (engl. Normalization) je postupak primjene niza pravila kojima se osigurava optimalna struktura baze podataka. Normalne forme su niz primjene tih pravila.

- Svakom slijedećom normalnom formom dobiva se bolja struktura podataka nego u prethodnoj normalnoj formi. Za nas su najznačajnije prve tri normalne forme.

Normalizacija - 1NF

- Prva normalna forma zahtjeva **atomičnost (nedjeljivost vrijednosti)** polja tablice i da svi zapisi moraju imati isti broj polja. To znači da se u istom polju ne mogu zapisivati npr. ime i prezime. Ovime se podaci strukturaju u tablice i stupce.
- Razlog postojanja ovog pravila je to što je nemoguće pretraživati ili sortirati podatke prema imenu ili prezimenu ukoliko se obje vrijednosti nalaze u istom polju.
- Slijedeći zahtjev koji tablica mora zadovoljiti da bi bila u 1NF je ne sadržavati vrijednosti koje se ponavljaju.

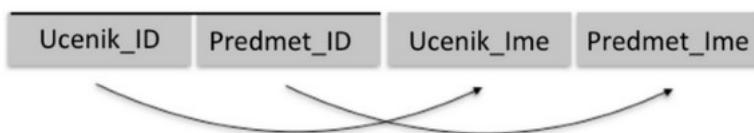
Predmet	Sadržaj
Programiranje	JavaScript, C#, SQL
Web	HTML, CSS, Asp Net

Predmet	Sadržaj
Programiranje	JavaScript
Programiranje	C#
Programiranje	SQL
Web	HTML
Web	CSS
Web	Asp Net

Normalizacija - 2NF

- Druga normalna forma zahtjeva da je zadovoljena 1NF (nadovezuje se na 1NF) i sva polja moraju biti jednoznačna i u potpunosti ovisiti o glavnem ključu (engl. primary key).
- Drugim riječima u svakoj tablici moraju se zapisivati podaci o samo jednom subjektu. Primjer nepoštivanja 2NF je tablica u kojoj se zapisuju podaci o projektu zajedno s podacima o učeniku koji radi na projektu.
- Kako bi tablicu prebacili u 2NF potrebno je takvu tablicu razdvojiti u dvije tablice. Postupak razdvajanja podataka u dvije tablice zove se dekompozicija. Uklanjamo suvišne podatke iz tablice koji se primjenjuju na više redova i stavljamo ih u odvojene tablice.

Projekt_Ucenik



Tablica nije u 2NF

Ucenik

Ucenik_ID

Ucenik_Ime

Predmet_ID

Projekt

Predmet_ID

Predmet_Ime

Tablice su u 2NF*Normalizacija - 3NF*

- Treća normalna forma zahtjeva da je zadovoljena 2NF i sva polja koja nisu dio glavnog ključa ne smiju međusobno ovisiti jedno o drugom - eliminira tranzitivnu ovisnost.
- Treba izbaciti sva izračunavanja u tablici npr. kada uz polja [Cijena] i [Količina] želimo pomnožiti te podatke i pohraniti u polje [Ukupno].
- Problem se javlja kada ažuriramo polje [Cijena] ili polje [Količina] jer se polje [Ukupno] ($[Cijena]*[Količina]$) ne ažurira automatski. Zato ga treba izbaciti po 3NF.

Ucenik_detaljno

Ucenik_ID

Ucenik_ime

Grad

Post_broj

**Tablica nije u 3NF****Ucenik**

Ucenik_ID

Ucenik_Ime

Post_broj

Posta

Post_broj

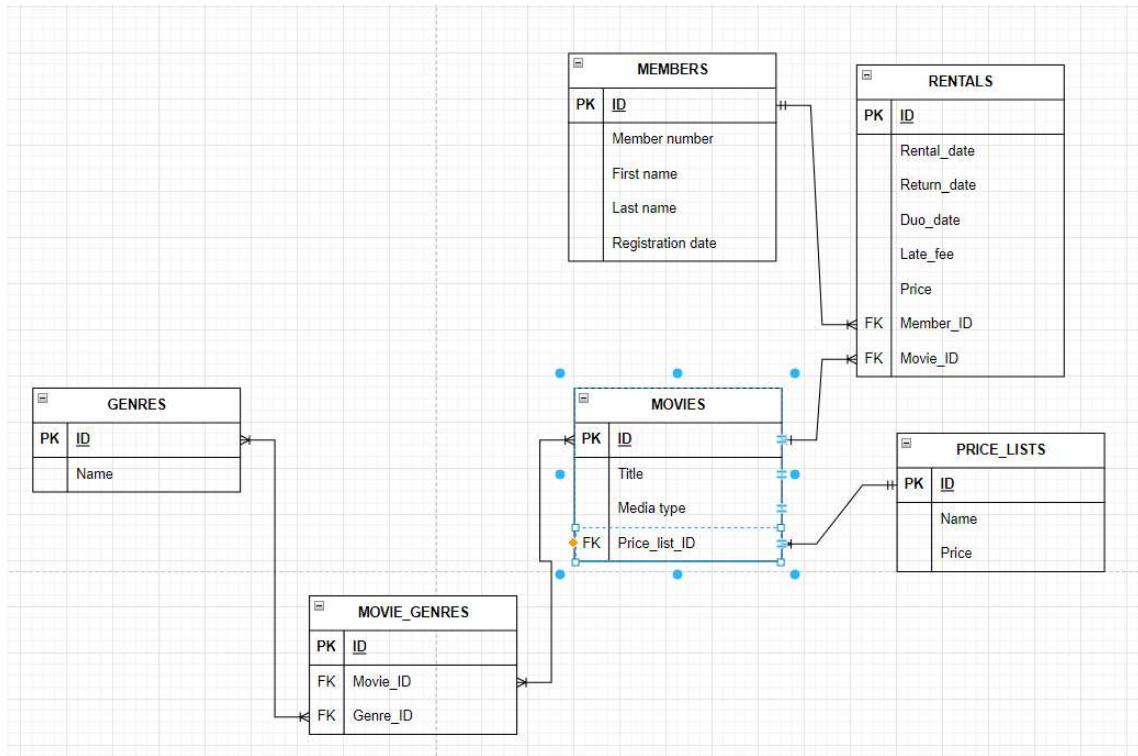
Grad

Tablice su u 3NF*Denormalizacija*

- Iako je cilj korisnika normalizacija u mnogim slučajevima ima smisla denormalizaciju (engl. denormalisation) ili namjerno odstupanje od normaliziranih formi.
- Osnovni razlog za denormalizaciju je poboljšanje svojstva brzine baze podataka.
- Prilikom pitanja odustajanja od normalizacije moramo se pitati je li nam poboljšanje svojstava ili lakoća održavanja baze podataka.

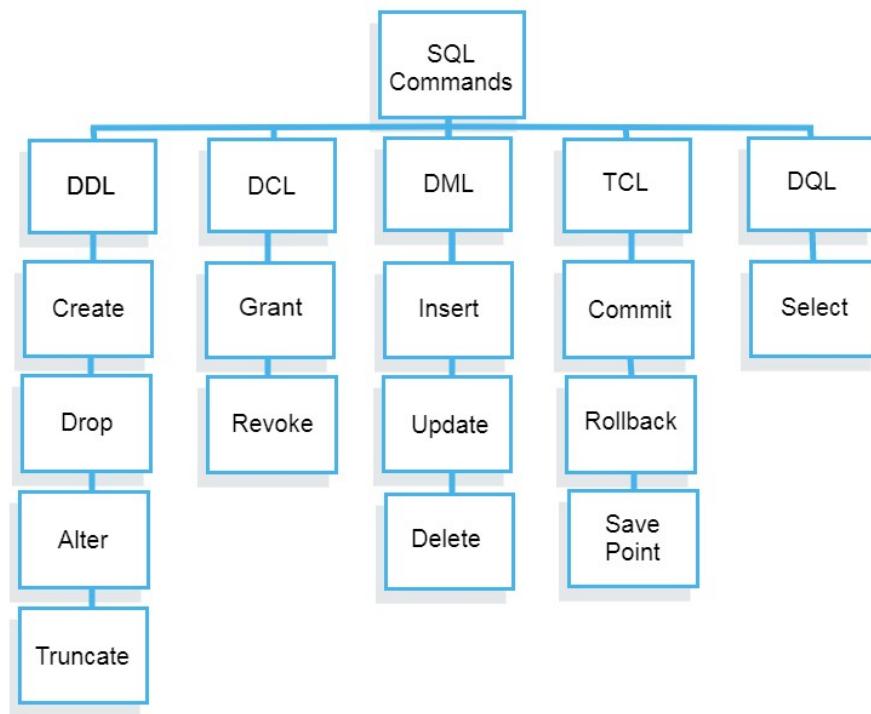
Vježba

Nakon što ste u prošloj vježbi kreirali ER dijagram za bazu podataka videoteke, istu normalizirajte i kreirajte dijagram shemu



SQL jezik

- Structured Query Language (SQL) je strukturni upitni jezik, programski jezik visokog nivoa.
- Najpopularniji je računalni jezik za izradu, traženje, ažuriranje i brisanje podataka iz relacijskih baza podataka.
- SQL je standardiziran preko standarda ANSI i ISO.
- SQL jezik sastoji se od ključnih riječi na engleskom koje se mogu podijeliti u nekoliko grupa:
 - o **DCL** (engl. Data Control Language) – jezik za kontrolu podataka, prava i dopuštenja
 - o **DML** (engl. Data Manipulation Language) – jezik za manipulaciju podacima, omogućuje izmjenu instance baze podataka umetanjem, mijenjanjem i brisanjem podataka
 - o **DDL** (engl. Data Definition Language) – jezik za definiranje strukture ili sheme baze podataka
 - o **DQL** (engl. Data Query Language) – jezik za izvršavanje upita nad podacima iz baze podataka



SQL sintaksni dijagram

- Riječi napisane velikim slovima rezervirane su riječi SQL-a koje treba pisati točno kako je navedeno.
- Riječi napisane nakošeno predstavljaju varijable koje, kada pišete izraz, treba zamijeniti nazivima objekata ili vrijednostima.
- Okomita crta (|) razdvaja ponuđene opcije, od kojih je moguće izabrati samo jednu.
- Uglate zagrade ([]) uokviruju stavke koje nisu obavezne, a vitičaste zagrade ({}) one koje jesu.

```

SELECT
  [ALL | DISTINCT | DISTINCTROW ]
  [HIGH_PRIORITY]
  [STRAIGHT_JOIN]
  [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
  [SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
  select_expr [, select_expr ...]
  [FROM table_references
    [PARTITION partition_list]
  [WHERE where_condition]
  [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_condition]
  [ORDER BY {col_name | expr | position}
    [ASC | DESC], ...]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  [PROCEDURE procedure_name(argument_list)]
  [INTO OUTFILE 'file_name'
    [CHARACTER SET charset_name]
    export_options
  | INTO DUMPFILE 'file_name'
  | INTO var_name [, var_name]]
  [FOR UPDATE | LOCK IN SHARE MODE]]

```

`select_expr` označava koje stupce selektiramo iz baze. `DISTINCT` označava da ne uzimamo potencijalne duplike iz baze (npr. za neka prebrojavanja). Primjer je ako imamo kupovine i neke transakcije a želimo saznati koji je broj kupaca kod nas kupovao. `FROM table_references` se koristi ako spajamo više tablica (npr. `CROSS JOIN`). Možemo selektirati iz jedne tablice ili iz više tablica ako koristimo `CROSS JOIN`, pomoću `WHERE` opcije. Mogu se naravno koristiti i drugi tipovi `JOIN`. `PARTITION partition_list` služi za partpcioniranje baze na više različitih particija kako bi smanjili opterećenje baze i razdvojili opterećenje na različite servere.

`WHERE where_condition` kaže daj mi neki uslov postavi i preko njega filtriramo cijeli `SELECT`. Npr. dohvati korisnike koji su obavili kupovinu nakon nekog datuma. `GROUP BY` je grupiranje po nekom stupcu. Koristi se kada u `SELECT` izrazu želimo odabrati kupce i njihova prezimena i želimo prebrojati koliko kupaca ima. S `DISTINCT` smo rekli da ćemo ukloniti duplike. Kako ne želimo dobiti van par stotina redova već želimo samo da piše broj redova. Za to postoje agregatne funkcije `COUNT`, `AVERAGE` i slične. Kada koristimo takve funkcije onda koristimo obavezno `GROUP BY` po stupcu.

`HAVING where_condition` se koristi nad grupiranim podacima i mora doći iza `GROUP BY`. Inače `SELECT` izraz mora poštovati redoslijed ali izraze koji nisu obavezni (u uglatim zagradama), a ne trebaju nam, možemo slobodno preskočiti.

`LIMIT` služi za limitiranje blokova (engl. chunks) kada imamo ogromne količine podataka, npr. milione zapisa u tablici. Tada radimo paginatore i u chunkove hvatamo paginatore. Koristimo `offset` unutar `LIMIT`.

Možemo potencijalno pozivati procedure s `PROCEDURE procedure_name`. Možemo zapisati upit u datoteku `INTO OUTFILE 'file_name'` a možemo ga i dumpati ga `INTO DUMPFILE 'file_name'`.

Postoji mogućnost kontrole zaključavanja nekih tablica dok se nešto događa.

Evo primjera:

```
SELECT FirstName  
FROM Student  
Where RollNo > 15;
```

SQL Uobičajeni upiti

SHOW DATABASES

Prikazuje informacije o postojećim bazama podataka na serveru:

Baze podataka '**information_schema**' , '**mysql**' i '**performance_schema**' su sistemске baze podataka koje interno koristi MySQL server. Baza podataka 'test' namijenjena je testiranju koje se daje tokom instalacije.

Database
adventureworkshop
fcc
information_schema
mysql
performance_schema
phpmyadmin
test
videoteka

USE baza_podataka_ime

Ovo postavlja bazu podataka kao trenutnu bazu podataka na MySQL serveru.

Za prikaz trenutnog naziva baze podataka koji je postavljena, koristite sintaksu:

```
SELECT DATABASE();
```

DESCRIBE tablica_ime

Vrati opis stupaca **tablica_ime** s imenima Polja, Tipom polja, Null (može li biti ili ne), Ključevima, Podrazumijevanom vrijednošću, Dodatkom (npr. da li je **auto_increment**)

SHOW TABLE

Prikazuje popis definiranih tablica u odabranoj bazi podataka.

SHOW CREATE TABLE tablica_ime

Ovo prikazuje potpunu **CREATE TABLE** naredbu kojom možemo kreirati tablicu.

SELECT NOW()

Prikazuje trenutni datum i vrijeme.

SELECT 2 + 4

Izvrši računsku operaciju i ispiše vrijednost.

Komentari

SQL podržava komentare u jednom redu ili u više redova:

```
/* Ovo je višeredni komentar */  
# Ovo je komentar u jednom redu.  
-- I ovo je komentar u jednom redu
```

`CREATE DATABASE baza_podataka_ime`

Ova naredba stvara novu bazu podataka.

`DROP DATABASE baza_podataka_ime`

Ova naredba briše bazu podataka.

`CREATE TABLE tablica_ime(stupac1, stupac2, stupac3..)`

Ova naredba kreira novu tablicu sa zadanim stupcima.

```
CREATE TABLE radnik(
    'id' INTEGER NOT NULL AUTO_INCREMENT,
    'ime' VARCHAR(30) NOT NULL,
    'profil' VARCHAR(40) DEFAULT 'inženjer',
    PRIMARY KEY ('id')
)ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

imate stupac 'id' kao AUTO_INCREMENT s ograničenjem primarnog ključa koji osigurava da svaki ID ima povećanu vrijednost, izbjegavajući duplicitanje. Odabrani mehanizam za pohranu je 'InnoDB' koji dopušta ograničenje stranog ključa i povezane transakcije. Definirana kodna stanica je UTF8 sa našim slovima.

`DROP TABLE tablica_ime`

Ovo briše tablicu koju navedemo.

`RENAME TABLE stara_tablica_ime TO nova_tablica_ime`

Ova naredba mijenja ime tablice u novo ime.

`ALTER TABLE tablica_ime ADD(stupac1, stupac2, stupac 3..)`

Ova naredba dodaje stupce postojećoj tablici.

`ALTER TABLE tablica_ime DROP(stupac1)`

Ova naredba briše navedene stupce iz postojeće tablice.

```
INSERT INTO tablica_ime (stupac1, stupac2, stupac3 . . ) VALUES(vrijednost1,
vrijednost2, vrijednost3. . )
```

Ova naredba umeće novi zapis u tablicu s navedenim vrijednostima.

```
UPDATE tablica_ime SET stupac1 = vrijednost1, stupac2 = vrijednost2, stupac3 =
vrijednost3.. WHERE uslov
```

Ova naredba ažurira zapise (slogove) u tablici s novim danim vrijednostima za stupce.

`DELETE FROM tablica_ime WHERE uslov`

Ova naredba briše slogove iz tablice

`SELECT stupac1, stupac2, stupac3... FROM tablica_ime WHERE uslov`

Ova naredba se izvršava i daje zapise (slogove) iz određenih stupaca iz tablice koji odgovaraju uslovuu nakon `WHERE` klauzule.

`SELECT * FROM tablica_ime`

Umjesto navođenja jednog stupca ili više stupaca, možete koristiti zvjezdicu (*) koja predstavlja sve stupce tablice. Ovaj upit dohvaća sve zapise iz tablice.

COUNT

Funkcija `COUNT` koristi se za vraćanje ukupnog broja zapisa koji odgovaraju uvjetu iz bilo koje tablice.

To je jedna od poznatih agregatnih funkcija

Primjer :

```
SELECT COUNT(*) from student;
```

MAX

Koristi se za dobivanje maksimalne numeričke vrijednosti određenog stupca tablice.

Primjer :

```
SELECT MAX(bodovi) FROM student_izvjestaj;
```

Funkcije MIN i MAX rade na numeričkim kao i na abecednim vrijednostima.

MIN

Koristi se za dobivanje minimalne numeričke vrijednosti određenog stupca tablice.

Primjer :

```
SELECT MIN(bodovi) FROM student_izvjestaj;
```

LIMIT

Koristi se za postavljanje ograničenja broja zapisa u skupu rezultata.

Primjer :

```
SELECT *
from student limit 4, 10;
```

To daje 10 zapisa počevši od 5. zapisa.

BETWEEN

Koristi se za dobivanje zapisa od navedene donje granice do gornje granice.

Time se provjerava nalazi li se vrijednost unutar zadanog raspona.

Primjer :

```
SELECT * FROM zaposleni  
WHERE dob between 25 TO 45;
```

DISTINCT

Ovo se koristi za dohvaćanje svih različitih zapisa izbjegavajući sve duplike.

Primjer :

```
SELECT DISTINCT profil  
FROM zaposleni;
```

IN klauzula

Ovo provjerava je li red sadržan u skupu zadanih vrijednosti.

Koristi se umjesto upotrebe toliko OR klauzule u upitu.

Primjer :

```
SELECT *  
FROM employee  
WHERE age IN(40, 50, 55);
```

AND

Ovaj uslov u MySQL upitima koristi se za filtriranje podataka rezultata na temelju AND uslova.

Primjer :

```
SELECT NAME, AGE  
FROM student  
WHERE bodovi > 95 AND ocjena = 7;
```

OR

Ovaj uslov u MySQL upitima koristi se za filtriranje podataka rezultata na temelju OR uslova.

Primjer :

```
SELECT *  
FROM student  
WHERE address = 'LJudevita Gaja bb' OR address = 'Zaprešić';
```

IS NULL

Ova se ključna riječ koristi za booleovu usporedbu ili za provjeru je li podatkovna vrijednost stupca null.

Primjer:

```
SELECT *
FROM zaposleni
WHERE broj_ugovora IS NULL;
```

FOREIGN KEY

Koristi se za ukazivanje na PRIMARY KEY druge tablice.

Primjer :

```
CREATE TABLE kUPCI
(
    id INT AUTO_INCREMENT PRIMARY KEY,
    IME VARCHAR(30) NOT NULL,
);

CREATE TABLE Narudžbe
(
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    FOREIGN KEY (id) REFERENCES Kupci (id)
);
```

LIKE

Ovo se koristi za dohvaćanje zapisa koji odgovaraju određenom STRING uzorku.

Primjer :

```
SELECT *
FROM ZAPOSLENI
WHERE ime LIKE 'Sh%';

SELECT *
FROM zaposleni
WHERE ime LIKE '%Sh%';
```

Znak postotka (%) u upitu predstavlja nulu ili više znakova.

JOIN

Spajanja radimo s dvije ili više tablica baze podataka radi dohvaćanja podataka na temelju zajedničkog polja.

Postoje različiti tipovi spajanja s različitim imenima u različitim bazama podataka.

Uobičajeno poznata spajanja su ***SELF JOIN***, ***OUTER JOIN***, ***INNER JOIN***, ***LEFT JOIN*** i ***RIGHT JOIN***.

ADD i DROP stupci

Novi stupac može se dodati u tablicu baze podataka, ako je potrebno kasnije.

Primjer :

```
ALTER TABLE zaposleni ADD COLUMN plaća VARCHAR(25);
```

Slično, bilo koji stupac može se izbrisati iz tablice baze podataka.

Primjer :

```
ALTER TABLE zaposleni DROP COLUMN plaća
```

SQL tipovi podataka

- Tip podataka je skup vrijednosti koji može poprimiti i operacije koje su definirane na tom skupu vrijednosti.
- Postoje tri osnovne vrste podataka:
 - **Tekstualni**
 - **Numerički**
 - **Datum/vrijeme**
- Samim odabirom vrste podataka zapravo definirate koja vrsta informacije će biti pohranjena u to polje tablice, odnosno temeljem onoga što želite pohraniti u bazu.
- Odabirom ispravne vrste podataka znatno možete utjecati na performanse same MySQL baze podataka.

SQL tipovi podataka - tekstualni

- Maksimalnu dužinu znakova kod definicije tipa podatka upisujete u uglate zagrade (), npr. **VARCHAR** (50), što znači da to polje može primiti string maksimalne dužine do 50 znakova.
- **CHAR** tip podataka zbog toga što je fiksne dužine je malo brži kod **VARCHAR**, jer je u cijelom stupcu zapravo ne mijenja svoju dužinu, već ostaje isti. No zbog fiksne dužine troši više memorije nego **VARCHAR**.
- **BLOB** i **TEXT** su po svojoj dužini jednaki i zapravo predstavljaju verziju VARCHAR tipa podataka koji može spremiti više od 255 znakova, točnije 65535.

Tip podatka	Opis
CHAR(veličina)	Sadrži string fiksne dužine (može sadržavati slova, brojke i posebne znakove). Fiksna veličina navedena je u zagradi. Može pohraniti do 255 znakova
VARCHAR(veličina)	Sadrži string promjenjive dužine (može sadržavati slova, brojke i posebne znakove). Maksimalna veličina navedena je u zagradi. Može pohraniti do 255 znakova. Napomena: Ako stavite vrijednost veću od 255, ona će se pretvoriti u tip TEXT
TINYTEXT	Sadrži string s maksimalnom dužinom od 255 znakova
TEXT	Sadrži string maksimalne dužine od 65535 (2^16 znakova)

BLOB	BLOB je Binary Large OBjects. Sadrži do 65.535 bajtova podataka
MEDIUMTEXT	Sadrži string s maksimalnom dužinom od 16.777.215 (2^24 znakova)
MEDIUMBLOB	Za BLOB (Binary Large OBjects). Sadrži do 16.777.215 bajtova (2^24 znakova)
LONGTEXT	Sadrži string s maksimalnom dužinom od 4.294.967.295 (2^32 podataka)
LONGBLOB	Za BLOB (binary Large OBjects). Sadrži do 4.294.967.295 bajtova (2^32 podataka)
ENUM(x,y,z,itd.)	Omogućuje unos popisa mogućih vrijednosti. Možete navesti do 65.535 vrijednosti na popisu ENUM. Ako je umetnuta vrijednost koja nije na popisu, bit će umetnuta prazna vrijednost. Napomena: Vrijednosti su poredane redoslijedom kojim ste ih unijeli. Možete unijeti moguće vrijednosti u ovom formatu: ENUM('X','Y','Z')
SET	Slično ENUM-u osim što SET može sadržavati do 64 stavke popisa i može pohraniti više od jednog izbora

SQL tipovi podataka - numerički

- Cijeli brojevi (integer) imaju dodatnu opciju zvanu **UNSIGNED** koja određeni numerički tip umjesto u raspon od negativnog do pozitivnog prebacuje u pozitivni raspon počevši od nule.

Tip podataka	Opis
TINYINT(veličina)	-128 do 127 uobičajeno. 0 do 255 UNSIGNED*. Najveći broj znamenki može biti naveden u zagradama
SMALLINT (veličina)	-32.768 do 32.767 uobičajeno. 0 do 65535 UNSIGNED*. Maksimalan broj znamenki može biti naveden u zagradi
MEDIUMINT (veličina)	-8388608 do 8388607 uobičajeno. 0 do 16777215 UNSIGNED*. Maksimalan broj znamenki može biti naveden u zagradi
INT(veličina)	-2.147.483.648 do 2.147.483.647 uobičajeno. 0 do 4.294.967.295 UNSIGNED*. Maksimalan broj znamenki može biti naveden u zagradi
BIGINT(veličina)	-9.223.372.036.854.775.808 do 92.2337.203.6854.775.807 uobičajeno. 0 do 18446744073709551615 UNSIGNED*. Maksimalan broj znamenki može biti naveden u zagradi
FLOAT(veličina,d)	Mali broj s pomičnim decimalnim zarezom. Najveći broj znamenki može se navesti u parametru veličine. Maksimalni broj znamenki desno od decimalne točke naveden je u parametru d
DOUBLE(veličina, d)	Veliki broj s pomičnim decimalnim zarezom. Najveći broj znamenki može se navesti u parametru veličine. Maksimalni broj znamenki desno od decimalne točke naveden je u parametru d
DECIMAL(veličina, d)	DOUBLE pohranjen kao string, dopuštajući fiksnu decimalnu točku. Najveći broj znamenki može se navesti u parametru veličine. Maksimalni broj znamenki desno od decimalne točke naveden je u parametru d

SQL tipovi podataka – datum i vrijeme

DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS.
TIMESTAMP	YYYYMMDDHHMMSS.
TIME	HH:MM:SS.

Tip podatka	Opis
DATE()	Datum . Format: GGGG-MM-DD Napomena: Podržani raspon je od '1000-01-01' do '9999-12-31'
DATETIME()	Kombinacija datuma i vremena. Format: GGGG-MM-DD HH:MI:SS Napomena: Podržani raspon je od '1000-01-01 00:00:00' do '9999-12-31 23:59:59'
TIMESTAMP()	Vremenska oznaka. Vrijednosti TIMESTAMP-a pohranjuju se kao broj sekundi od Unix epohe ('1970-01-01 00:00:00' UTC). Format: GGGG-MM-DD HH:MI:SS Napomena: Podržani raspon je od '1970-01-01 00:00:01' UTC do '2038-01-09 03:14:07' UTC
TIME()	Vrijeme. Format: HH:MI:SS Napomena: Podržani raspon je od '-838:59:59' do '838:59:59'
YEAR()	Godina u dvoznamenkastim ili četveroznamenkastim formatima. Napomena: Dopuštene vrijednosti u četveroznamenkastim formatima: 1901 do 2155. Dopuštene vrijednosti u dvoznamenkastim formatima: 70 do 69, predstavljaju godine od 1970. do 2069.

Na kraju, postoji nekoliko drugih vrsta podataka koje ćete koristiti:

Tip podatka	Opis
BOOLEAN	Pohranjuje TRUE ili FALSE vrijednosti
ARRAY	Skupna duljina i uređena kolekcija elemenata
MULTISET	Neuređena zbirka elemenata promjenjive duljine
XML	Pohranjuje XML podatke

SQL – DDL ključne riječi

- DDL se sastoji od SQL naredbi koje se mogu koristiti za definiranje sheme baze podataka.
- Koristi se za stvaranje i izmjenu strukture objekata baze podataka u bazi podataka.
- DDL ključne riječi
 - o **CREATE** - koristi se za izradu baze podataka ili njenih objekata (poput tablice, indeksa, funkcija, prikaza, postupka pohrane i okidača).
 - o **DROP** - koristi se za brisanje objekata iz baze podataka.
 - o **ALTER** - koristi se za promjenu strukture baze podataka.

```
1 CREATE DATABASE [IF NOT EXISTS] database_name  
2 [CHARACTER SET charset_name]  
3 [COLLATE collation_name]
```

```
1 DROP DATABASE [IF EXISTS] database_name;
```

```
1 CREATE TABLE [IF NOT EXISTS] table_name(  
2     column_1_definition,  
3     column_2_definition,  
4     ...  
5     table_constraints  
6 ) ENGINE=storage_engine;
```

```
1 ALTER TABLE table_name  
2 ADD  
3     new_column_name column_definition  
4     [FIRST | AFTER column_name]
```

```
1 DROP [TEMPORARY] TABLE [IF EXISTS] table_name [, table_name] ...  
2 [RESTRICT | CASCADE]
```

CREATE

Naredbe **CREATE** koriste se za definiranje sheme strukture baze podataka:

```
Create database university;  
Create table students;  
Create view for_students;
```

DROP

DROP naredba uklanja tablice i baze podataka iz RDBMS-a.

```
Drop object_type object_name;  
Drop database university;  
Drop table student;
```

ALTER

ALTER naredba omogućava promjenu strukture baze podataka.

Možemo dodati novi stupac u tablicu:

```
ALTER TABLE table_name ADD column_name STUPAC-definicija;
```

Ili možemo izmijeniti postojeći stupac u tablici:

```
ALTER TABLE MODIFY(COLUMN DEFINITION....);
```

Npr.:

```
alter table branko add subject varchar;
```

TRUNCATE

TRUNCATE naredba se koristi za brisanje svih redova iz tablice i oslobođanje prostora koji sadrži tablicu:

```
TRUNCATE TABLE table_name;
```

SQL – DML ključne riječi

- Data Manipulation Language (DML) omogućuje izmjenu instance baze podataka umetanjem, mijenjanjem i brisanjem njezinih podataka. Odgovoran je za izvođenje svih vrsta izmjena podataka u bazi podataka.
- Postoje tri osnovne konstrukcije koje programu baze podataka i korisniku omogućavaju unos podataka i informacija su:
 - INSERT
 - UPDATE
 - DELETE

INSERT

Ova se naredba koristi za umetanje podataka u redak tablice.

```
INSERT INTO TABLE_NAME (col1, col2, col3,... col N)  
VALUES (value1, value2, value3, .... valueN);
```

ili

```
INSERT INTO TABLE_NAME  
VALUES (value1, value2, value3, .... valueN);
```

Na primjer:

```
INSERT INTO students (RollNo, FirstName, LastName) VALUES ('60', 'Tom', 'Erichsen');
```

UPDATE

Ova se naredba koristi za ažuriranje ili izmjenu vrijednosti stupca u tablici.

```
UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE  
CONDITION]
```

Na primjer:

```
UPDATE students  
SET FirstName = 'Jhon', LastName= 'Wick'  
WHERE StudID = 3;
```

DELETE

Ova se naredba koristi za uklanjanje jednog ili više redova iz tablice.

```
DELETE FROM table_name [WHERE condition];
```

Na primjer:

```
DELETE FROM students  
WHERE FirstName = 'Jhon';
```

SQL – DCL ključne riječi

DCL (Data Control Language) uključuje naredbe kao GRANT i REVOKE, koje su korisne za davanje "prava i dopuštenja". Ostale dozvole kontroliraju parametre sustava baze podataka.

GRANT

Ova se naredba koristi za davanje privilegija korisničkog pristupa bazi podataka.

```
GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;
```

Na primjer:

```
GRANT SELECT ON Users TO 'Tom'@'localhost';
```

REVOKE

Korisno je povući dopuštenja od korisnika.

```
REVOKE privilege_name ON object_name FROM {user_name | PUBLIC | role_name}
```

Na primjer:

```
REVOKE SELECT, UPDATE ON student FROM BCA, MCA;
```

SQL - TCL ključne riječi

Jezik kontrole transakcija ili TCL naredbe bave se transakcijama unutar baze podataka.

```
Commit;
```

Na primjer:

```
DELETE FROM Students  
WHERE RollNo =25;  
COMMIT;
```

ROLLBACK

Naredba Rollback omogućuje poništavanje transakcija koje još nisu spremljene u bazu podataka.

```
ROLLBACK;
```

Primjer:

```
DELETE FROM Students  
WHERE RollNo =25;
```

SAVEPOINT

Ova naredba vam pomaže da postavite točku spremanja unutar transakcije.

```
SAVEPOINT SAVEPOINT_NAME;
```

Primjer:

```
SAVEPOINT RollNo;
```

SQL - DQL ključne riječi

Data Query Language (DQL) koristi se za dohvaćanje podataka iz baze podataka. Koristi samo jednu naredbu:

SELECT

Ova naredba vam pomaže odabratiti atribut na temelju uvjeta opisanog **WHERE** klauzulom.

```
SELECT expressions  
FROM TABLES  
WHERE conditions;
```

Na primjer:

```
SELECT FirstName  
FROM Student  
WHERE RollNo > 15;
```

MySQL ograničenja podataka(engl. Data Constraints)

Ograničenje NOT NULL

Ograničenje NOT NULL u MySQL-u služi kao pravilo provjere ispravnosti za stupce. U **NOT NULL** ograničenjima stupac ne može sadržavati **NULL** vrijednosti. Ovo ograničenje garantira da svaki red u tablici mora imati vrijednost za taj određeni stupac. Ograničenje **NOT NULL** sprječava umetanje praznih podataka ili podataka koji nedostaju u stupac te tablice. Zbog ovog ograničenja povećava se točnost i pouzdanost podataka, a baza podataka postaje robusnija

Primjer:

Kreiranje tablice 'users' bez **NOT NULL** početnog ograničenja.

```
CREATE TABLE users (
```

```

        user_id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(50),
        email VARCHAR(100),
        age INT
);

```

Ubacimo neke podatke u tablicu i pogledajmo:

	user_id	username	email	age
1	ram_123	ram123@gmail.com	30	NULL
2	sumitb17	sumit@gmail.com	NULL	
3	rohit264	rohits@yopmail.com	35	NULL
4	mahi	msd@gmail.com	NULL	
5	Bob	bobzy@gmail.com	25	NULL
	NULL	NULL	NULL	NULL

Dodavanje NOT NULL ograničenja u postojeći stupac

Da biste dodali ograničenje NOT NULL postojecem stupcu, potrebna je direktna izmjena strukture tablice. Korištenjem SQL naredbi ovo se ograničenje može dodati kako bi se osigurala **dosljednost (konzistencija) podataka**. Evo primjera dodavanja ograničenja NOT NULL u stupac 'email' u tablici 'users':

```

ALTER TABLE korisnici
MODIFY COLUMN email VARCHAR(100) NOT NULL;

```

Prikaz izmijenjene strukture tablice:

	Field	Type	Null	Key	Default	Extra
▶	user_id	int	NO	PRI	NULL	auto_increment
	username	varchar(50)	YES		NULL	
	email	varchar(100)	NO		NULL	
	age	int	YES		NULL	

Objašnjenje: Ova naredba modificira stupac 'email' u tablici 'korisnici', specificirajući da ne može sadržavati **NULL** vrijednosti. Pokušaj umetanja NULL u 'mail' (dovest će do greške zbog **NOT NULL** ograničenja). Izjava za umetanje neće uspjeti jer krši **NOT NULL** ograničenje dodano u stupac 'email', osiguravajući da se **NULL** vrijednosti ne mogu umetnuti.

Uklanjanje NOT NULL ograničenja

Uklanjanje NOT NULL ograničenja iz stupca može utjecati na postojeće podatke i integritet . Za uklanjanje **NOT NULL** ograničenja iz stupca, može se koristiti sljedeći SQL upit.

```

ALTER TABLE users
MODIFY COLUMN email VARCHAR(100);

```

Prikaz promijenjene strukture tablice nakon uklanjanja ograničenja `NOT NULL`.

	Field	Type	Null	Key	Default	Extra
▶	user_id	int	NO	PRI	NULL	auto_increment
	username	varchar(50)	YES		NULL	
	email	varchar(100)	YES		NULL	
	age	int	YES		NULL	

Objašnjenje: Uklanjanjem '`NOT NULL`' ograničenja iz stupca, dozvoljeno je umetanje `NULL` vrijednosti i to utječe na konzistentnost podataka. Nakon izmjene stupca '`email`' možemo umetnuti `NULL` vrijednosti u stupac '`email`'.

`UNIQUE` ograničenje

`UNIQUE` ograničenje osigurava da vrijednosti u stupcu ili grupi stupaca ostanu jedinstvene, sprječavajući dvostrukе unose u stupac i održavajući cjelovitost tablice.

`UNIQUE` ograničenje u MySQL-u sprječava da dva zapisa imaju identične vrijednosti u stupcu. `UNIQUE` ograničenje može sadržavati nulte vrijednosti sve dok je kombinacija vrijednosti jedinstvena. To ga čini drugačijim od `PRIMARY KEY` jer ograničenje primarnog ključa ne može sadržavati nulte vrijednosti.

U tablici može postojati više `UNIQUE` ograničenja, ali samo jedno `PRIMARY KEY` ograničenje.

`UNIQUE` ograničenje u MySQL-u je kao golman koji osigurava da podaci uneseni u stupac ili grupu stupaca ostanu različiti. Ako se unese dvostruka vrijednost, MySQL će javiti grešku osiguravajući da je svaka vrijednost u određenom stupcu jedinstvena.

Na primjer, može se koristiti u stupcima koji sadrže adrese e-pošte, ID brojeve ili kodove proizvoda čime se osigurava integritet podataka.

Sintaksa

Sintaksa za korištenje `UNIQUE` ograničenja u MySQL-u može varirati ovisno o tome dodajete li ograničenja tokom kreiranja ili tom ažuriranja tablice. Ovisno o situaciji, možete koristiti sljedeće sintakse:

`UNIQUE` ograničenje kod `CREATE TABLE`

```
CREATE TABLE tablica_ime (
    stupac_ime tip_podatka UNIQUE,
    ...
);
```

Za dodavanje `UNIQUE` ograničenja na naredbu `CREATE TABLE` koristite sljedeću sintaksu:

```
CREATE TABLE tablica_ime (
    stupac_ime1 tip_podatka,
```

```

stupac_ime2 tip_podataka,
...
UNIQUE (stupac_ime1, stupac_ime2)
);

```

UNIQUE ograničenje kod ***ALTER TABLE***

Da biste dodali ***UNIQUE*** ograničenje tokom ažuriranja tablice, koristite sljedeću sintaksu:

```

ALTER TABLE tablica_ime
ADD CONSTRAINT ogranicenje_ime UNIQUE (stupac_ime);

```

Primjer MySQL ***UNIQUE*** ograničenja

Pretpostavimo da imamo tablicu pod nazivom ***products*** sa stupcem ***product_code*** u kojem smo inicijalno postavili ***UNIQUE*** ograničenje kako bismo osigurali da je svaki kod proizvoda jedinstven:

```

CREATE TABLE proizvodi (
    product_id INT AUTO_INCREMENT PRIMARY KEY ,
    product_code VARCHAR(20) UNIQUE ,
    product_name VARCHAR (100),
    cijena DECIMAL (10, 2)
);

```

To bi spriječilo da dva koda proizvoda imaju isti kod, održavajući cjelovitost tablice.

Prikaz strukture tablice:

	Field	Type	Null	Key	Default	Extra
▶	product_id	int	NO	PRI	NULL	auto_increment
	product_code	varchar(20)	YES	UNI	NULL	
	product_name	varchar(100)	YES		NULL	
	price	decimal(10,2)	YES		NULL	

Ubacimo neke podatke u tablicu ***product***:

	product_id	product_code	product_name	price
1	P001	Product A	29.99	
2	P002	Product B	39.99	
3	P003	Product C	19.99	
▶ 4	P004	Product D	49.99	
*	NULL	NULL	NULL	NULL

U tablici ***product*** svaki red predstavlja proizvod s jedinstvenom šifrom proizvoda, nazivom proizvoda i cijenom. Stupac ***product_code*** ograničen je da bude jedinstven, tako da će ti unosi biti prihvati

samo ako su šifre proizvoda različite. Ako pokušamo umetnuti duplicitanu vrijednost u stupac `product_code`, javit će se greška (Dvostruki unos za ključ '`products.product_code`').

Odbacivanje UNIQUE ograničenja s `DROP INDEX`

Da bismo uklonili UNIQUE ograničenje, moramo koristiti jednostavnu naredbu:

```
ALTER TABLE product
DROP INDEX product_code;
```

Nakon uklanjanja `UNIQUE` ograničenja tablica će strukture biti:

	Field	Type	Null	Key	Default	Extra
▶	<code>product_id</code>	int	NO	PRI	NULL	auto_increment
	<code>product_code</code>	varchar(20)	YES		NULL	
	<code>product_name</code>	varchar(100)	YES		NULL	
	<code>price</code>	decimal(10,2)	YES		NULL	

Izraz `DROP INDEX` uklanja UNIQUE ograničenje iz stupca `product_code` i dopušta mogućnost postojanja dvostrukih kodova proizvoda u stupcu. To bi moglo utjecati na dosljednost podataka u tablici.

Dodavanje jedinstvenog ključa s `ALTER TABLE`

Koristeći naredbu `ALTER TABLE`, možete dodati `UNIQUE` ograničenje postojećoj tablici.

```
ALTER TABLE product
ADD CONSTRAINT unique_product_code UNIQUE (product_code);
```

Nakon dodavanja `UNIQUE` ograničenja s `ALTER TABLE` struktura tablice će biti:

	Field	Type	Null	Key	Default	Extra
▶	<code>product_id</code>	int	NO	PRI	NULL	auto_increment
	<code>product_code</code>	varchar(20)	YES	UNI	NULL	
	<code>product_name</code>	varchar(100)	YES		NULL	
	<code>price</code>	decimal(10,2)	YES		NULL	

Ovaj izraz `ALTER TABLE` dodaje novo `UNIQUE` ograničenje pod nazivom `unique_product_code` stupcu `product_code` unutar tablice `product`. Ovo ograničenje osigurava da svaki kod proizvoda ostane jedinstven, sprječavajući umetanje duplikata.

Ograničenje `UNIQUE` razlikuje se od ograničenja `PRIMARY KEY` jer primarni ključ ne može imati nulte vrijednosti i može se koristiti samo jednom u tablici.

DEFAULT ograničenje

DEFAULT ograničenje vraća zadatu vrijednost za stupac tablice. DEFAULT vrijednost stupca je vrijednost koja se koristi u slučaju kada ne postoji vrijednost koju je odredio korisnik.

Za korištenje ove funkcije stupcu treba biti dodijeljena DEFAULT vrijednost. Inače će generirati grešku.

Sintaksa

```
DEFAULT (naziv_stupca)
```

Dodavanje **DEFAULT** ograničenja s **CREATE TABLE** izjavom

Sintaksa

```
CREATE TABLE tablica_ime (
    stupac1 tip_podatka DEFAULT default_vrijednost,
    STUPAC2 tip_podatka DEFAULT default_vrijednost, ... );
```

Primjer:

```
CREATE TABLE kupci (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ime VARCHAR(255) NOT NULL,
    email VARCHAR(255) DEFAULT 'example@domain.com',
    grad VARCHAR(50) DEFAULT 'Nepoznat'
);
```

Dodavanje **DEFAULT** ograničenja s **ALTER TABLE** izjavom

Možemo dodati **DEFAULT** ograničenje već postojećoj tablici pomoću naredbe **ALTER TABLE**.

Sintaksa:

```
ALTER TABLE tablica_ime ALTER stupac_ime SET DEFAULT default_vrijednost;
```

Primjer

```
ALTER TABLE kupci ALTER grad SET DEFAULT 'Zagreb';
```

Kreirajmo MySQL tablicu "blog_objave"

```
CREATE TABLE blog_objave (
    objava_id INT AUTO_INCREMENT PRIMARNI KLJUČ,
    naslov VARCHAR(255) NOT NULL,
    sadržaj TEXT NOT NULL,
    datum_objave DATE DEFAULT CURRENT_DATE
```

```
);
```

Što je ključ (engl. key) u SQL-u

Ključ u SQL-u je vrijednost koja se koristi za jedinstvenu identifikaciju zapisa u tablici. SQL ključ je jedan stupac ili kombinacija više stupaca koji se koriste za jedinstvenu identifikaciju redova ili torki u tablici. SQL ključ se koristi za identifikaciju duplicitnih informacija, a također pomaže uspostaviti odnos između više tablica u bazi podataka.

Napomena: Stupci u tablici koji se NE koriste za jedinstvenu identifikaciju zapisa nazivaju se **neklučni stupci**.

Primarni ključ

Primarni ključ je stupac ili skup stupaca koji jedinstveno identificira svaki red u tablici. Primarni ključ slijedi ova pravila:

- Primarni ključ mora sadržavati jedinstvene vrijednosti. Ako se primarni ključ sastoji od više stupaca, kombinacija vrijednosti u tim stupcima mora biti jedinstvena.
- Stupac primarnog ključa ne može imati **NULL** vrijednosti. Svaki pokušaj umetanja ili ažuriranja NULL u stupcima primarnog ključa rezultirat će pogreškom. Imajte na umu da MySQL implicitno dodaje NOT NULL ograničenje stupcima primarnog ključa.
- Tablica može imati samo jedan primarni ključ



Korištenje **PRIMARY KEY** kod **CREATE TABLE**

Možete KREIRATI primarni ključ dok kreirate tablicu pomoću naredbe **CREATE TABLE**.

Sintaksa

```
CREATE TABLE tablica_ime (
    stupac1 tip_podataka,
    stupac2 tip_podataka,
    PRIMARY KEY (stupac1)
);
```

```
CREATE TABLE tablica_ime (
    stupac1 tip_podataka PRIMARY KEY,
    stupac2 tip_podataka,
);
```

```
//Kompozitni primarni ključ
CREATE TABLE tablica_ime (
    stupac1 tip_podataka,
    stupac2 tip_podataka,
    stupac3 tip_podataka,
    PRIMARY KEY (stupac1, stupac2)
);
```

Primjer

Uput za izradu tablice i dodavanje primarnog ključa pomoću izjave `CREATE TABLE`:

```
CREATE TABLE NekaTablica(Id int, Ime varchar(20), PRIMARY KEY(Id));
```

Korištenje `PRIMARY KEY` kod `ALTER TABLE`

Možete dodati primarni ključ u već postojeću tablicu pomoću naredbe `ALTER TABLE`. Ako tablica već ima jedan primarni ključ, tada ne možete dodati drugi pomoću izjave `ALTER TABLE`.

Sintaksa

```
ALTER TABLE tablica_ime ADD PRIMARY KEY (stupac_ime);
```

Primjer

Uput za dodavanje primarnog ključa pomoću izjave `ALTER TABLE`:

```
ALTER TABLE NekaTablica ADD PRIMARY KEY (Ime);
```

Uklanjanje primarnog ključa

Uklanjanje primarnog ključa iz tablice možete koristiti naredbu `ALTER TABLE`.

```
ALTER TABLE tablica_ime DROP PRIMARY KEY;
```

Primarni ključ	Jedinstveni ključ
Svaka vrijednost u stupcu primarnog ključa mora biti jedinstvena.	Svaka vrijednost u stupcu jedinstvenog ključa mora biti jedinstvena, ali tablica može imati više jedinstvenih ključeva.
Stupac primarnog ključa ne može sadržavati NULL vrijednosti.	Jedinstveni stupac ključa dopušta jednu NULL vrijednost.
Tablica može imati samo jedan primarni ključ.	Tablica može imati više od jednog jedinstvenog ključa.
Prema zadanim postavkama, primarni ključ kreira klasterirani indeks (utječe na redoslijed fizičke pohrane podataka).	Zadani indeks za jedinstveni ključ nije klasteriran.

Strani ključ i referentni integritet

```

1 [CONSTRAINT constraint_name]
2 FOREIGN KEY [foreign_key_name] (column_name, ...)
3 REFERENCES parent_table(column_name,...)
4 [ON DELETE reference_option]
5 [ON UPDATE reference_option]

```

Strani ključ je stupac ili grupa stupaca u tablici koji se povezuje na stupac ili grupu stupaca u drugoj tablici. **FOREIGN KEY** je polje u tablici koje se odnosi na jedinstveni ključ u drugoj tablici. Koristi se za povezivanje jedne ili više tablica. Drugi naziv za **FOREIGN KEY** je referentni ključ . Stvara vezu (relaciju) između dvije tablice čime se stvara referentni integritet.

FOREIGN KEY stvara odnos između stupaca u trenutnoj tablici ili recimo tablici A (onoj sa stranim ključem) i referentnoj tablici ili tablici B (onoj s jedinstvenim ključem). Ovo stvara odnos roditelj-dijete gdje se tablica sa STRANIM KLJUČEM u podređenoj tablici odnosi na stupac primarnog ili jedinstvenog ključa u nadređenoj tablici. Strani ključ održava referentni integritet između podređene i nadređene tablice pomoću klauzula **ON DELETE** i **ON UPDATE**.

- Strani ključ može imati različit naziv od primarnog ključa
- Osigurava da redovi u jednoj tablici imaju odgovarajuće redove u drugoj
- Za razliku od primarnog ključa, oni ne moraju biti jedinstveni. Najčešće nisu
- Strani ključevi mogu biti **null** iako primarni ključevi ne mogu

Kako definirati FOREIGN KEY u MySQL-u

Ograničenje **FOREIGN KEY** je ograničenje relacijske baze podataka koje osigurava referentni integritet između dvije tablice. Uspostavlja vezu između stupca ili skupa stupaca u jednoj tablici, koja se naziva podređena tablica, i primarnog ključa ili stupaca jedinstvenog ključa u drugoj tablici, poznatoj kao nadređena tablica.

Metoda **CREATE** može se koristiti za definiranje vašeg **FOREIGN KEY**-a u MySQL-u:

```

CREATE TABLE podređena_tablica (
    podređeni_id INT PRIMARY KEY,
    nadređeni_id INT,
    drugi_stupci DATATYPE,
    FOREIGN KEY (nadređeni_id) REFERENCES nadređena_tablica(nadređeni_id)
);

```

podređena_tablica: naziv tablice u kojoj se definira strani ključ.

podređeni_id: Primarni ključ je stupac od **podređena_tablica**.

nadređeni_id: Stupac stranog ključa u **podređena_tablica** koji se odnosi na primarni ključ u nadređenoj tablici.

Shvatimo ovo na primjeru:

Imamo dvije tablice, Korisnici i Narudžbe . Tablica Korisnici bit će nadređena tablica jer sadrži PRIMARY KEY (`korisnik_id`) i referencirana je stranim ključem (`korisnik_id`) . A tablica Narudžbe bit će podređena tablici.

```
CREATE TABLE korisnici (
    korisnik_id INT PRIMARY KEY,
    korisnik_ime VARCHAR(50),
    email VARCHAR(100) UNIQUE NOT NULL
);

CREATE TABLE narudžbe (
    narudžba_id INT PRIMARY KEY,
    korisnik_id INT,
    narudžba_datum DATE,
    ukupan_iznos DECIMAL(10, 2),
    FOREIGN KEY (user_id) REFERENCES korisnici(user_id)
);
```

 **Strani ključ**



ID ČLANA	MOVIES RENTED
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshall
2	Daddy's Little Girls
3	Clash of the Titans

Strani ključ upućuje na Primarni ključ
Strani ključ može imati samo vrijednosti prisutne u Primarnom ključu
Može imati drugačije ime od Primarnog primarni ključa

 **Primarni ključ**

ID ČLANA	FULL NAMES	PHYSICAL ADDRESS	SALUTATION
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3 rd Street 34	Mr.
3	Robert Phil	5 th Avenue	Mr.

Primjer stranog ključa koji koristi djelovanje `SET NULL`

`SET NULL` djelovanje u ograničenju stranog ključa koristi se za brisanje ili ažuriranje reda u nadređenoj tablici dok se stupac stranog ključa u podređenoj tablici postavlja na `NULL`. `[NULL]` se ovdje odnosi na prazno ili nepoznato]

Pojednostaviti će ovo uz pomoć primjera:

Imamo dvije tablice: `radnici` i `odjeli`. Tablica radnika sadrži informacije o radnicima zajedno s `odjel_id` (odjel kojem pripadaju).

Tablica odjela ima naziv odjela i odgovarajući ID u stupcu `odjel_id`. (npr. 101 za HR, 102 za IT...)

Tablica radnika ima `odjel_id` kao strani ključ koji se odnosi na primarni ključ (`odjel_id`) u tablici odjela. Sada u scenariju gdje je odjel izbrisana, odgovarajući `odjel_id` u tablici radnika automatski će se promijeniti na `NULL`.

Dakle, čak i kada se odjel izbriše, još uvijek imamo evidenciju radnika koji su bili iz tog odjela. Ovo je važno pravilo koje nam pomaže zadržati podatke i izbjegći pogreške u tablici radnika čak i kada se nešto u tablici odjela promjeni ili izbriše.

```
drop database if exists obrisi_me;

create database if not exists obrisi_me
character set utf8mb4 collate utf8mb4_general_ci;

use obrisi_me;

CREATE TABLE odjeli (
    odjel_id INT PRIMARY KEY,
    odjel_ime VARCHAR(50) NOT NULL
);

-- zatim kreiramo tablicu radnika s stranim ključem koristeći SET NULL djelovanje

CREATE TABLE radnici (
    radnik_id INT PRIMARY KEY,
    radnik_ime VARCHAR(50) NOT NULL,
    odjel_id INT,
    FOREIGN KEY (odjel_id) REFERENCES odjeli(odjel_id) ON UPDATE SET NULL ON DELETE
SET NULL
);

-- Umetanje uzorka podataka u tablice

INSERT INTO odjeli (odjel_id, odjel_ime) VALUES
```

```

(1, 'HR'),
(2, 'Inženjering'),
(3, 'Marketing');

INSERT INTO radnici (radnik_id, radnik_ime, odjel_id) VALUES
(101, 'Sanja Ulipi', 1),
(102, 'Branko Ćelap', 2),
(103, 'Smiljka Božičković', 3);

-- Prikaz podataka u tablici radnici i tablici odjeli
SELECT * FROM radnici;
SELECT * FROM odjeli;

```

Sada ažurirajmo `odjel_id` u tablici odjela, promijenimo `odjel_id` 2 u `odjel_id` 4 i postavimo ga na `NULL` u tablici radnika

```

UPDATE odjeli SET odjel_id = 4 WHERE odjel_id =
2;
SELECT * FROM radnici;

```

radnik_id	radnik_ime	odjel_id
101	Sanja Ulipi	1
102	Branko Ćelap	NULL
103	Smiljka Božičković	3

Sada izbrišite odjel i postavimo odgovarajući `odjel_id` na `NULL` u tablici zaposlenika

```

DELETE FROM odjeli WHERE odjel_id = 1;
SELECT * FROM radnici;

```

radnik_id	radnik_ime	odjel_id
101	Sanja Ulipi	NULL
102	Branko Ćelap	NULL
103	Smiljka Božičković	3

Kako ukloniti strani ključ

Strani ključ se može ispustiti korištenjem klauzule `DROP FOREIGN KEY`. Evo osnovne sintakse:

```

ALTER TABLE <naziv_tablice>
DROP FOREIGN KEY ime_stranog_ključa;

```

Definiranje stranog ključa pomoću `ALTER TABLE`

Strani ključ možemo definirati pomoću naredbe `ALTER TABLE` u MySQL-u. Ovo je slučaj kada već imate tablicu i trebate dodati strani ključ u nju.

Ograničenje stranog ključa pomaže u održavanju referentnog integriteta između dvije tablice.

Može se koristiti sljedeća sintaksa:

```

ALTER TABLE podređena_tablica
ADD FOREIGN KEY (podređeni_stupac)
REFERENCES nadređena_tablica (nadređeni_stupac);

```

Evo primjera:

```
CREATE TABLE kupci (
    kupac_id INT PRIMARY KEY,
    kupac_ime VARCHAR(50) NOT NULL
);

-- tablica bez stranog ključa
CREATE TABLE narudžbe (
    narudžba_id INT PRIMARY KEY,
    narudžba_datum DATE,
    kupac_id INT
);
```

Sada ćemo u tablici narudžbe dodati strani ključ s ALTER TABLE

```
ALTER TABLE narudžbe
ADD CONSTRAINT fk_narudžba_kupac_id
FOREGIN KEY (kupac_id) REFERENCES kupci(kupac_id);
```

Ako ne želimo ograničenju dati ime, koristit ćemo ovu sintaksu:

```
ALTER TABLE narudžbe
FOREGIN KEY (kupac_id) REFERENCES kupci(kupac_id);
```

Dakle preskočili smo drugi red.

MySQL ima pet referentnih opcija:

- **CASCADE**: ako je red iz nadređene tablice izbrisana ili ažurirana, vrijednosti odgovarajućih redova u podređenoj tablici automatski se brišu ili ažuriraju.
- **SET NULL**: ako je red iz nadređene tablice izbrisana ili ažurirana, vrijednosti stupca stranog ključa (ili stupaca) u podređenoj tablici postavljene su na NULL.
- **RESTRICT**: ako red iz nadređene tablice ima odgovarajući redak u podređenoj tablici, MySQL odbija brisanje ili ažuriranje redaka u nadređenoj tablici.
- **NO ACTION**: isto je kao i RESTRICT.
- **SET DEFAULT**: prepoznaće MySQL parser. Međutim, ovu radnju odbacuju i InnoDB¹ (default engine u MySQL) i NDB tablice.

Kompozitni ključ

Složeni ključ je primarni ključ sastavljen od više stupaca koji se koriste za jedinstvenu identifikaciju zapisa.

¹ InnoDB je mehanizam za pohranu podataka za sistem upravljanja bazom podataka MySQL i MariaDB . Od izdanja MySQL 5.5.5 2010., zamijenio je MyISAM kao zadatu vrstu tablice MySQL-a. Omogućava standardne transakcijska svojstva uskladjena s ACID-om (atomičnost , dosljednost , izolacija , trajnost - engl. to macity, consistency, isolation, durability), zajedno s podrškom za strani ključ (deklarativni referentni integritet).

Recimo da u nekoj našoj bazi podataka imamo dvije osobe s istim imenom Robert Phil, ali žive na različitim mjestima.

Kompozitni ključ			
Robert Phil	3 rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5 th Avenue	Clash of the Titans	Mr.

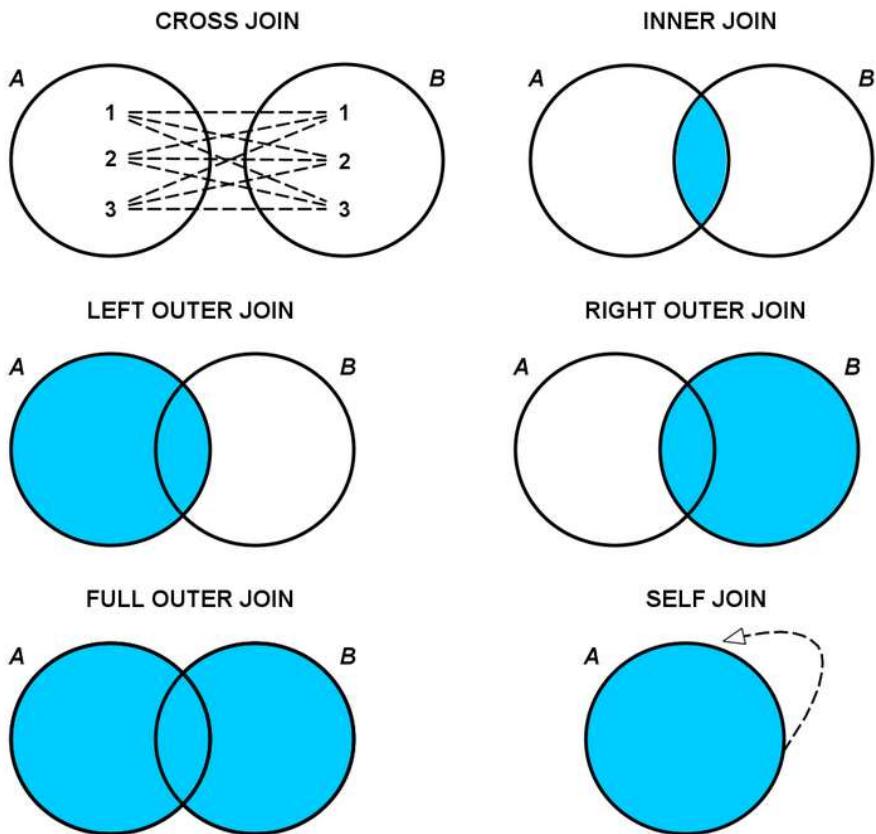
Imena su uobičajena. Zato vam je potrebno osim imena i adresu za jedinstvenu identifikaciju zapisa.

Zato zahtijevamo i puno ime i adresu kako bismo jedinstveno identificirali zapis. To je kompozitni ključ.

SQL spajanja - Cross Join, Full Outer Join, Inner Join, Left Join i Right Join

SQL spajanja (engl. join) omogućavaju da restrukturamo naše razdvojene tablice iz baze podataka nazad u odnose tako da nam pomažu pokrenuti naše aplikacije.

Ovdje ćemo razmotriti različite vrste spajanja u SQL-u i kako ih koristiti.



Što je spajanje (engl. join)

Spajanje je operacija koja kombinira dva reda zajedno u jedan red.

Ti su redovi obično iz dvije različite tablice—ali ne moraju biti.

Prije nego što pogledamo kako napisati samo spajanje, pogledajmo kako bi izgledao rezultat spajanja.

Uzmimo za primjer sistem koji pohranjuje informacije o korisnicima i njihovim adresama.

Redovi iz tablice koja pohranjuje korisničke informacije mogu izgledati ovako:

id	name	email	age
1	John Smith	johnsmith@gmail.com	25
2	Jane Doe	janedoe@Gmail.com	28
3	Xavier Wills	xavier@wills.io	3
...			
(7 rows)			

A redovi iz tablice koja pohranjuje informacije o adresi mogu izgledati ovako:

id	street	city	state	user_id
1	1234 Main Street	Oklahoma City	OK	1
2	4444 Broadway Ave	Oklahoma City	OK	2
3	5678 Party Ln	Tulsa	OK	3
(3 rows)				

Mogli bismo napisati odvojene upite kako bismo našli podatke o korisniku i podatke o adresi – ali idealno bismo mogli napisati jedan upit (engl. query) i dobili sve korisnike i njihove adrese u istom skupu rezultata (engl. result set).

To je upravo ono što nam spajanje (engl. join) omogućava!

Uskoro ćemo pogledati kako napisati ta spajanja, ali ako spojimo naše korisničke podatke s našim podacima o adresi, mogli bismo dobiti rezultat kao što je ovaj:

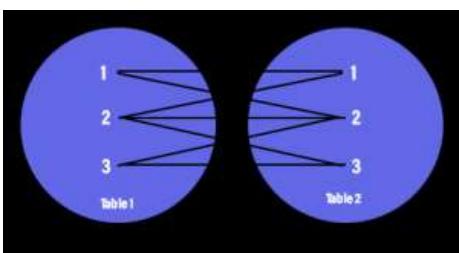
id	name	email	age	id	street	city	state	user_id
1	John Smith	johnsmith@gmail.com	25	1	1234 Main Street	Oklahoma City	OK	1
2	Jane Doe	janedoe@Gmail.com	28	2	4444 Broadway Ave	Oklahoma City	OK	2
3	Xavier Wills	xavier@wills.io	35	3	5678 Party Ln	Tulsa	OK	3
(3 rows)								

Ovdje vidimo sve naše korisnike i njihove adrese u jednom finom skupu rezultata.

Osim stvaranja kombiniranog skupa rezultata, još jedna važna upotreba spojeva (engl. joins) je povlačenje dodatnih informacija u naš upit koje možemo filtrirati.

Na primjer, ako želimo poslati fizičku poštu svim korisnicima koji žive u Oklahoma Cityju, mogli bismo upotrijebiti ovaj skup rezultata i filtrirati na temelju **city** stupca.

CROSS JOIN - križno spajanje



Najjednostavnija vrsta spajanja koju možemo napraviti je CROSS JOIN ili "Kartezijev produkt". Ovo spajanje uzima svaki red iz jedne tablice i spaja ga sa svakim redom druge tablice.

Sintaksa:

```
SELECT * FROM tablica1  
CROSS JOIN tablica2;
```

Prvo kreirajmo dvije vrlo jednostavne tablice i u njih ubacimo neke podatke:

```
drop database if exists fcc;  
  
create database if not exists fcc  
character set utf8mb4 collate utf8mb4_general_ci;  
  
use fcc;  
  
CREATE TABLE slova(  
    slovo TEXT  
);  
  
INSERT INTO slova(slovo) VALUES ('A'), ('B'), ('C');  
  
CREATE TABLE brojevi(  
    broj TEXT  
);  
  
INSERT INTO brojevi(broj) VALUES (1), (2), (3);
```

```

1 • drop database if exists fcc;
2
3 • create database if not exists fcc
4   character set utf8mb4 collate utf8mb4_general_ci;
5
6 • use fcc;
7
8 • CREATE TABLE slova(
9   slovo TEXT
10 );
11
12 • INSERT INTO slova(slovo) VALUES ('A'), ('B'), ('C');
13
14 • CREATE TABLE brojevi(
15   broj TEXT
16 );
17
18 • INSERT INTO brojevi(broj) VALUES (1), (2), (3);
19

```

Naše dvije tablice `slova` i `brojevi`, imaju samo jedan stupac: jednostavno tekstualno polje.

Sada ih spojimo zajedno, s `CROSS JOIN`:

Kad bismo imali dvije liste – jedna sadrži `1, 2, 3` i drugi sadrži `A, B, C` – Kartezijev produkt ili umnožak te dvije liste bio bi ovaj:

Svaka vrijednost s prve liste uparena je sa svakom vrijednošću druge liste.

Napišimo ovaj isti primjer kao SQL upit.

```
SELECT *
FROM slova
CROSS JOIN brojevi;
```

Result Grid			
		slovo	broj
▶	A	1	
	B	1	
	C	1	
	A	2	
	B	2	
	C	2	
	A	3	
	B	3	
	C	3	

Ovo je najjednostavnija vrsta spajanja koju možemo napraviti —ali čak i u ovom jednostavnom primjeru možemo vidjeti spajanje na djelu: dva odvojena reda (jedan iz `slova` a drugi iz `brojevi`) su spojena zajedno u jedan red (slog).

Iako se o ovoj vrsti spajanja često govorи samo kao o pukom akademskom primjeru, ono ima barem jedan dobar slučaj upotrebe: pokrivanje opsega datuma.

Spajanje sa CROSS JOIN i s rasponom datuma

Jedan zanimljiv slučaj je uzeti svaki red iz tablice i primijeniti ga na svaki dan unutar datumskog raspona. Ovo smo napravili s UNION ALL.

Recimo, na primjer, da ste radili aplikaciju koja prati dnevne zadatke - stvari poput pranja zubi, doručkovanja ili tuširanja.

Ako želite generirati zapis za svaki zadatak i za svaki dan u prošlom tjednu, možete upotrijebiti CROSS JOIN nasuprot datumskog raspona.

Da bismo napravili ovaj datumski raspon, možemo upotrijebiti funkciju generirana_serija:

```
WITH recursive generirana_serija AS (
    SELECT (CURRENT_DATE - INTERVAL 5 day) AS Datum
    UNION ALL
    SELECT Datum + interval 1 day
    FROM generirana_serija
    WHERE Datum < CURRENT_DATE
select * FROM generirana_serija;
```

Funkcija generirana_serija koristi tri dijela.

Prvi parametar je početna vrijednost. U ovom primjeru koristimo CURRENT_DATE - INTERVAL 5 day. Ovo vraća tekući datum minus pet dana—ili "prije pet dana."

Drugi dio je UNION ALL koji kombinira rezultate dvije i više SELECT naredbi (omogućava i duplike).

Idući SELECT je "interval koraka"—ili koliko želimo povećati vrijednost svaki puta. Budući da se radi o dnevnim zadacima koristit ćemo interval od jednog dana i odabiremo SELECT Datum + (INTERVAL 1 day), dakle početni datum uvećan za jedan dan. Ovdje imamo praktično petlju iz generirana_serija koja uzima datum dok nije manji od današnjeg datuma.

Kada se sve skupi, ovo generira seriju datuma koji počinju prije pet dana, završava danas i ide dan po dan. Primjetite da stupac ima alias zadan sa AS Datum kako bi izlaz bio malo ljepši.

Rezultat ovog upita je zadnjih pet dana plus današnji dan:

Result Grid	
	Datum
▶	2024-05-16
	2024-05-17
	2024-05-18
	2024-05-19
	2024-05-20
	2024-05-21

Vratimo se našem primjeru zadatka po danima, stvorimo jednostavnu tablicu za zadatke koje želimo dovršiti i ubacimo nekoliko zadataka:

```
CREATE TABLE zadaci(
    ime TEXT
);

INSERT INTO zadaci(ime) VALUES
('Oprati zube'),
('Doručkovati'),
```

```
('Tuširanje'),
('Obući se');
```

Naša `zadaci` tablica ima samo jedan stupac, `ime`, a u ovu smo tablicu umetnuli četiri zadatka.

Sada idemo s `CROSS JOIN` našim zadacima s upitom za generiranje datuma:

```
SELECT
    zadaci.ime,
    datumi.Datum
FROM zadaci
CROSS JOIN
(
    WITH recursive generirana_serija AS (
        SELECT (CURRENT_DATE - INTERVAL 5 day) AS Datum
        UNION ALL
        SELECT Datum + interval 1 day
        FROM generirana_serija
        WHERE Datum < CURRENT_DATE)
    SELECT * FROM generirana_serija
) AS datumi;
```

(Budući da naš upit za generiranje datuma nije stvarna tablica, samo ga pišemo kao podupit.)

Iz ovog upita vraćamo naziv zadatka i dan, a skup rezultata izgleda ovako:

	ime	Datum
▶	Oprati zube	2024-05-16
	Oprati zube	2024-05-17
	Oprati zube	2024-05-18
	Oprati zube	2024-05-19
	Oprati zube	2024-05-20
	Oprati zube	2024-05-21
	Doručkovati	2024-05-16
	Doručkovati	2024-05-17
	Doručkovati	2024-05-18
	Doručkovati	2024-05-19
	Doručkovati	2024-05-20
	Doručkovati	2024-05-21
	Tuširanje	2024-05-16
	Tuširanje	2024-05-17
	Tuširanje	2024-05-18

Result 20 ×

Kao što smo očekivali, dobivamo red za svaki zadatak za svaki dan u našem datumskom rasponu.

CROSS JOIN je najjednostavnije spajanje koje možemo učiniti, ali da bismo pogledali sljedećih nekoliko tipova trebat će nam realističnija postavka tablice.

Kreiranje tablica režisera i filmova za demonstraciju spajanja

Kako bismo ilustrirali neke vrste spajanja, upotrijebit ćemo primjer filmova i filmskih režisera.

U ovoj situaciji, film ima jednog režisera, ali film ne mora imati režisera—zamislite da se najavljuje novi film, ali izbor režisera još nije potvrđen.

Naša **režiseri** tablica će pohraniti ime svakog režisera, a tablica **filmovi** će pohraniti naziv filma kao i referencu na režisera filma (ako ga ima).

Kreirajmo te dvije tablice i u njih ubacimo neke podatke:

```
CREATE TABLE režiseri(
    id SERIAL PRIMARY KEY,
    ime TEXT NOT NULL
);

INSERT INTO režiseri(ime) VALUES
('John Smith'),
('Sanja Ulipi'),
('Xavier Wills'),
('Bev Scott'),
('Bree Jensen');

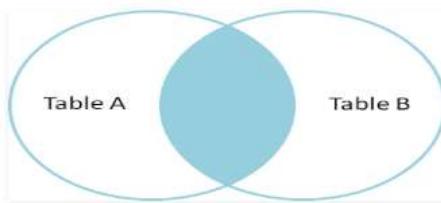
CREATE TABLE filmovi(
    film_id SERIAL PRIMARY KEY,
    ime TEXT NOT NULL,
    id INTEGER REFERENCES režiseri
);

INSERT INTO filmovi(ime, id) VALUES
('Film 1', 1),
('Film 2', 1),
('Film 3', 2),
('Film 4', NULL),
('Film 5', NULL);
```

Imamo pet režisera, pet filmova, a tri od tih filmova imaju režisere. Režiser ID 1 ima dva filma, a Režiser ID 2 ima jedan.

INNER JOIN - unutrašnje spajanje

Sljedeći tip spajanja, **INNER JOIN** jedan je od najčešće korištenih tipova spajanja.



Unutarnje spajanje vraća samo redove za koje je uslov spajanja istinit.

U našem primjeru, unutarnje spajanje između naše `filmovi` i `režiseri` tablice vratio bi samo zapise u kojima je filmu dodijeljen režiser.

`INNER JOIN` korištenjem dvije tablice (`ON` klauzula)

Sintaksa je:

```
SELECT <tablica1.stupac1>,<tablica1.stupac2>,...,<tablica2.stupac1>,...  
FROM tablica1 INNER JOIN tablica2  
ON tablica1.uslov_stupac = tablica2.uslov_stupac  
WHERE [uslov];  
GROUP BY <grupa-po-nazivu-stupca>  
HAVING [uslov];
```

`tablica1` je naša prva tablica, `tablica2` je naša druga tablica, a `uslov_stupac` je stupac koji se koristi za spajanje obje tablice pomoću ključne riječi `INNER JOIN`. Također, klauzule `WHERE`, `GROUP BY`, i `HAVING` su opcione i mogu se koristiti ovisno o SQL upitu.

Dobra praksa je koristiti aliase imena tablica s kraćim imenima kako bi izbjegli ponavljanje dugih naziva kada imate kompleksne `INNER JOIN` upite.

Vratimo se na primjer.

Prikaži sve filmove koji imaju režisera:

```
SELECT *  
FROM filmovi  
INNER JOIN režiseri  
ON režiseri.id = filmovi.id;
```

Naš rezultat pokazuje tri filma koja imaju režisera:

film_id	ime	id	id	ime
1	Film 1	1	1	John Smith
2	Film 2	1	1	John Smith
3	Film 3	2	2	Sanja Ulipi

Prikaži sve režisere koji imaju film:

```
SELECT *  
FROM režiseri  
INNER JOIN filmovi
```

```
ON filmovi.id = režiseri.id;
```

id	ime	film_id	ime	id
1	John Smith	1	Film 1	1
1	John Smith	2	Film 2	1
2	Sanja Ulipi	3	Film 3	2

Budući da smo prvo naveli `režiseri` tablicu u ovom upitu i odabrali smo sve stupce (`SELECT *`) iz obje tablice, prvo vidimo `režiseri` podatke stupca, a zatim stupce iz `filmovi` — ali rezultati su isti.

Ovo je korisno svojstvo unutrašnjeg spajanja, ali ne vrijedi za sve vrste spajanja

Budući da unutarnje spajanje uključuje samo redove koji odgovaraju uslovu spajanja, redoslijed dviju tablica u spajanju nije bitan, osim za redoslijed prikazanih stupaca. Da nismo koristili (`SELECT *`) morali bi nabrojati polja iz obje tablice koja želimo vidjeti. Gotovo nikad ne želimo vidjeti sva polja. Npr. `id` se pojavljuje 2 puta, a to ne želimo:

Evo istog primjera ali sa imenovanim poljima:

```
SELECT filmovi.film_id, filmovi.ime, filmovi.id, režiseri.ime
FROM filmovi
INNER JOIN režiseri
ON režiseri.id = filmovi.id;
```

Ključna riječ **ALIAS**

Ovo smo mogli napisati i kao:

```
SELECT F.film_id, F.ime, R.id, R.ime
FROM filmovi AS F
INNER JOIN režiseri AS R
ON R.id = F.id;
```

film_id	ime	id	ime
1	Film 1	1	John Smith
2	Film 2	1	John Smith
3	Film 3	2	Sanja Ulipi

Primijetite da smo `id` ovaj puta uzeli iz tablice `režiseri`. Mogli smo kao prije uzeti `id` iz `filmovi`. Svejedno je. Izbacili smo polje koje ne želimo vidjeti.

Često želimo aliase napisati skraćeno:

```
SELECT F.film_id, F.ime, R.id, R.ime
FROM filmovi F
INNER JOIN režiseri R
ON R.id = F.id;
```

INNER JOIN korištenjem više tablica (**ON** klauzule)

Moguće je zadati i više **INNER JOIN** naredbi i spojiti više tablica pomoću uslova spajanja stupca:

```
SELECT <list-of-column-names>
FROM table1
INNER JOIN tablica2 ON <uslov-stupac>
INNER JOIN tablica3 ON < uslov-stupac >
.....
INNER JOIN tablicaN ON < uslov-stupac >;
```

INNER JOIN s klauzulom **GROUP BY** i **HAVING**

GROUP BY i **HAVING** se mogu koristiti zajedno s **INNER JOIN** za pisanje složenih upita.

Sintaksa:

```
SELECT <tablica1.stupac1>,<tablica1.stupac2>,...,<tablica2.stupac1>,...,
       FROM tablica1 INNER JOIN tablica2
       ON tablica1.uslov_stupac = tablica2.uslov_stupac
       GROUP BY <grupa-po-stupac-ime >
       HAVING [uslov];
```

Ako imamo upit koji su režiseri radili na filmovima koristit ćemo grupiranje po imenu režisera:

```
SELECT R.ime
FROM filmovi F
INNER JOIN režiseri R
ON R.id = F.id
GROUP BY R.ime;
```

Grupiranje možemo vidjeti i na taj način da vidimo koliko je tko snimio filmova;

```
SELECT *, COUNT(F.id) AS 'Snimi(o/la) filmova'
FROM filmovi F
INNER JOIN režiseri R
ON R.id = F.id
GROUP BY R.ime;
```

ime	Snimi(o/la) filmova
John Smith	2
Sanja Ulipi	1

Sada osim grupiranja po imenu režisera, zanima nas koliko je koji režiser snimio filmova:

```
SELECT R.ime, count(*) AS 'Snimi(o/la) filmova'
```

```
FROM filmovi F
INNER JOIN režiseri R
ON R.id = F.id
GROUP BY R.ime
HAVING count(*) > 0;
```

ime	Snimi(o/la) filmova
John Smith	2
Sanja Ulipi	1

INNER JOIN s WHERE ključnom riječi

Možete koristiti ključnu riječ **WHERE** za dodatno filtriranje torki koje zadovoljavaju određeni uslov naveden u **WHERE** klauzuli.

```
SELECT <lista-imena-stupaca>
FROM tablica1 INNER JOIN tablica2
ON tablica1.uslov_stupac = table2.uslov_stupac
WHERE [uslov];
```

Evo prošlog primjera proširenog za **WHERE** klauzulu. Tražimo ima li filmova u kojima je režiser bio John Smith.

```
SELECT F.id, F.ime, R.id, R.ime
FROM filmovi F
INNER JOIN režiseri R
ON R.id = F.id
WHERE R.ime = 'John Smith';
```

Primijetite da su tablice **filmovi** i **režiseri** kod **FROM** i **INNER JOIN** obrnuti. Radi i jedno i drugo potpuno jednako.

INNER JOIN s USING klauzulom

Ako je naziv stupca u kojem se dvije tablice trebaju spojiti isti u objema tablicama, tada možemo koristiti klauzulu **USING**, koja uzima naziv stupca za spajanje kao ulaz. Ovdje ne moramo specificirati imena tablica ispred stupaca. Također, automatski se izbacuju duplikati povezanih stupaca u rezultatu.

Sintaksa:

```
SELECT <lista-imena-stupaca>
FROM tablica1 INNER JOIN tablica2
USING(<naziv-stupca-spajanja>);
```

USING možemo koristiti kada obje tablice dijele stupac potpuno istog imena s kojim ih spajamo. U našem slučaju korist ćemo:

```
SELECT ... FROM filmovi INNER JOIN režiseri USING (id) WHERE ...
```

Prikažimo tko je režirao Film 3:

```
SELECT id, F.film_id, F.ime, R.id, R.ime
FROM filmovi F
INNER JOIN režiseri R
USING (id)
WHERE F.ime = 'Film 3';
```

id	film_id	ime	id	ime
2	3	Film 3	2	Sanja Ulipi

Za ilustraciju ovo je moguće napraviti i s ON i tada će glasiti:

```
SELECT F.id, F.ime, R.ime
FROM filmovi F
INNER JOIN režiseri R
ON R.id = F.id
WHERE F.ime = 'Film 3';
```

Primijetite da umjesto `id` piše `F.id` u klauzuli `SELECT`. Izraz bi bio dvosmislen da nismo napisali iz koje tablice dolazi.

***INNER JOIN** korištenjem operatora*

Možemo koristiti različite SQL operatore zajedno s MySQL `INNER JOIN`, ti operatori mogu biti aritmetički operatori kao što su `+`, `-`, `*`, `/`, `%` ili mogu biti operatori uspoređivanja kao što je jednako (`=`), nije jednako (`!=`), veći od (`>`), manji od (`<`) ili čak mogu biti logički operatori kao što su `AND`, `OR`, `ANY`, `BETWEEN`, `EXISTS` itd.

```
SELECT <tablica1.stupac1>,<tablica1.stupac2>,...,<tablica2.stupac1>,... .
FROM tablica1 INNER JOIN tablica2
ON tablica1.uslov_stupac = tablica2.uslov_stupac
WHERE <uslov_stupac> <operator> <vrijednost>;
```

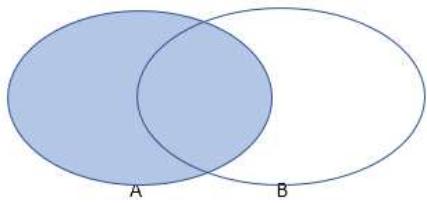
Evo primjera. Prikaži režisere koji su snimili barem 1 film a njihovog filma je veći od 1

```
SELECT F.film_id, F.ime, R.id, R.ime
FROM režiseri R
INNER JOIN filmovi F
ON F.id = R.id
WHERE F.film_ID >1;
```

***LEFT JOIN** - lijevo spajanje*

Ove dvije sljedeće vrste spajanja koriste modifikator (`LEFT` ili `RIGHT`) koji utječe na to koji će podaci tablice biti uključeni u skup rezultata.

Napomena: **LEFT JOIN** i **RIGHT JOIN** se također mogu nazivati **LEFT OUTER JOIN** i **RIGHT OUTER JOIN**.



Ova se spajanja koriste u upitima gdje želimo vratiti sve podatke određene tablice i ako postoje, također i podatke povezane tablice.

Ako pridruženi podaci ne postoje, svejedno dobivamo sve podatke "primarne" tablice.

To je upit za informacijama o određenoj stvari i bonus informacijama ako te bonus informacije postoje.

LEFT JOIN korištenjem dvije tablice s **ON** klauzulom

Sintaksa:

```
SELECT stupac1, stupac2, ...
FROM tablica1
LEFT JOIN tablica2
ON tablica1.stupac_ime = table2.stupac_ime;
```

Ovo će biti jednostavno razumjeti na primjeru. Pronađimo sve filmove i njihove režisere, ali nije nas briga imaju li režisera ili ne. Ako imaju, to je bonus:

```
SELECT *
FROM filmovi
LEFT JOIN režiseri
ON režiseri.id = filmovi.režiser_id;
```

film_id	ime	id	id	ime
1	Film 1	1	1	John Smith
2	Film 2	1	1	John Smith
3	Film 3	2	2	Sanja Ulipi
4	Film 4	NULL	NULL	NULL
5	Film 5	NULL	NULL	NULL

Upit slijedi isti obrazac kao i prije—samo smo specificirali spajanje kao **LEFT JOIN**.

U ovom primjeru, **filmovi** tablica je "lijeva" tablica.

Ako napišemo upit u jednom redu, ovo će biti malo lakše vidjeti:

```
... FROM filmovi LEFT JOIN režiseri ...
```

Lijevo spajanje vraća sve zapise iz "lijeve" tablice.

Lijevo spajanje vraća sve redove iz "desne" tablice koji odgovaraju uslovu spajanja.

Redovi iz "desne" tablice koji ne odgovaraju uslovu spajanja vraćaju se kao **NULL**.

	id	ime	režiser_id	id	ime
▶	1	Film 1	1	1	John Smith
	2	Film 2	1	1	John Smith
	3	Film 3	2	2	Sanja Ulipi
	4	Film 4	NULL	NULL	NULL
	5	Film 5	NULL	NULL	NULL

Gledajući taj skup rezultata, možemo vidjeti zašto je ova vrsta spajanja korisna za upite tipa "sve ovo i, ako postoji, nešto od onoga".

LEFT JOIN s USING klauzulom

```
SELECT *
FROM filmovi
LEFT JOIN režiseri
    USING(id);
```

Idući primjer pronalazi sve filmove i njihove režisere bez obzira da li je filmu dodijeljen režiser ili ne. Rješenje je isto kao i u prošlom primjeru ali s **USING** klauzulom.

```
SELECT *
FROM filmovi
LEFT JOIN režiseri
    USING(id);
```

Upit dobiva sve zapise iz lijeve tablice **filmovi** i odgovarajuće zapise iz desne tablice **režiseri**. Upit koristi klauzulu **USING** koja izvodi lijevo spajanje na temelju stupca koji je zajednički objema tablicama (**id**) koji je zajednički objema tablicama i ima zapise koji se podudaraju.

LEFT JOIN s GROUP BY klauzulom

Možemo koristiti klauzulu GROUP BY s ključnom riječi LEFT JOIN za pregled organiziranih rezultata na temelju određenih stupaca.

```
SELECT R.ime, COUNT(F.id) AS 'Broj filmova'
FROM filmovi F
LEFT JOIN režiseri R
    ON R.id = F.id
GROUP BY R.ime;
```

ime	Broj filmova
NULL	0
John Smith	2
Sanja Ulipi	1

GROUP BY se koristi u MYSQL-u za grupiranje redaka koji imaju slične vrijednosti. Ovaj upit vraća broj filmova koje je napravio svaki režiser.

LEFT JOIN s WHERE klauzulom

Možemo koristiti WHERE klauzulu s LEFT JOIN za pregled zapisa koji ispunjavaju određeni zahtjev. ovo omogućuje filtriranje podataka iz tablica.

```
SELECT F.film_id, F.ime, R.id, R.ime
FROM filmovi F
LEFT JOIN režiseri R
    ON R.id = F.id
WHERE F.ime = 'Film 3';
```

film_id	ime	id	ime
3	Film 3	2	Sanja Ulipi

LEFT JOIN za dobivanje zapisa bez podudaranja

Upit koji vraća neuparene zapise unutar tablica filmovi i režiser dat će nam koji su filmovi bez režisera:

```
SELECT *
FROM filmovi
LEFT JOIN režiseri
    ON režiseri.id = filmovi.id
WHERE filmovi.id IS NULL;
```

film_id	ime	id	id	ime
4	Film 4	NULL	NULL	NULL
5	Film 5	NULL	NULL	NULL

Možemo pogledati i koji su režiseri bez filmova:

```
SELECT *
FROM režiseri
LEFT JOIN filmovi
    ON režiseri.id = filmovi.id
WHERE filmovi.id IS NULL;
```

ime	film_id	ime	id
Xavier Wills	NULL	NULL	NULL
Bev Scott	NULL	NULL	NULL
Bree Jensen	NULL	NULL	NULL

Korištenje LEFT JOIN za pronalaženje podudaranja

U našem prethodnom upitu pronašli smo režisere koji ne pripadaju filmovima.

Koristeći našu istu strukturu, mogli bismo pronaći režisere koji pripadaju filmovima mijenjajući naš WHERE uslov da tražimo redove u kojima podaci o filmu nisu **NULL**:

```
SELECT *
FROM režiseri
LEFT JOIN filmovi
    ON filmovi.režiser_id = režiseri.id
WHERE filmovi.id IS NOT NULL;
```

id	ime	id	ime	režiser_id
1	John Smith	1	Film 1	1
1	John Smith	2	Film 2	1
2	Sanja Ulipi	3	Film 3	2

Ovo se može činiti zgodnim, ali zapravo smo upravo ponovno implementirali **INNER JOIN**!

RIGHT JOIN - desno spajanje

RIGHT JOIN radi točno kao **LEFT JOIN** — osim što su pravila o dvije tablice obrnuta.

Primarna svrha RIGHT JOIN je dohvaćanje svih zapisa iz DESNE (ili prve) tablice, bez obzira na to postoje li podudarni zapisi u desnoj (ili drugoj) tablici. Ovo osigurava da nijedan podatak nije DESNO u skupu rezultata. Na primjer, ako imamo dvije tablice RADNICI i projekti i izvedemo DESNO spajanje između njih, tada možemo saznati koji su to zaposlenici koji do sada nisu dodijeljeni projektu. Sintaksa:

```
Select column1, column2, ...
FROM tablica1
RIGHT JOIN tablica2
```

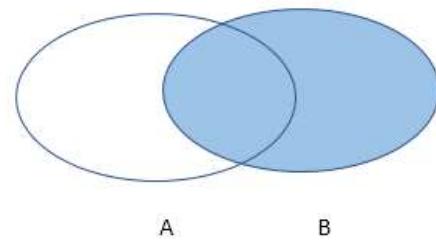
```
ON tablica1.stupac_ime = table2.stupac_ime;
```

Upotrijebimo isti upit kao gore, ali zamjenimo LEFT JOIN s RIGHT JOIN:

Ovo prikazuje sve režisere i filmove koje su snimili. Ako režisere nisu snimili filmove, također će biti na popisu.

```
SELECT *
FROM filmovi
RIGHT JOIN režiseri
    ON režiseri.id = filmovi.režiser_id;
```

id	ime	režiser_id	id	ime
1	Film 1	1	1	John Smith
2	Film 2	1	1	John Smith
3	Film 3	2	2	Sanja Ulipi
NULL	NULL	NULL	3	Xavier Wills
NULL	NULL	NULL	4	Bev Scott
NULL	NULL	NULL	5	Bree Jensen



Naš skup rezultata sada vraća svaki `režiseri` red i ako postoji, `filmovi` podatke.

Sve što smo učinili jest promijenili koju tablicu smatramo "primarnom" — tablicom iz koje želimo vidjeti sve podatke bez obzira na to postoje li povezani podaci.

RIGHT JOIN s klauzulom GROUP BY

```
SELECT R.ime, COUNT(F.id) AS 'Snimi(o/la) filmova'
FROM filmovi F
RIGHT JOIN režiseri R
    ON R.id = F.id
GROUP BY R.ime;
```

ime	Snimi(o/la) filmova
Bev Scott	0
Bree Jensen	0
John Smith	2
Sanja Ulipi	1
Xavier Wills	0

RIGHT JOIN s klauzulom WHERE

U ovom slučaju RIGHT JOIN rdi potpuno isto kao i primjer s LEFT JOIN. Cilj je prikazati tko je režirao 'Film 3' s WHERE klauzulom.

```
SELECT F.film_id, F.ime, R.id, R.ime
FROM filmovi F
RIGHT JOIN režiseri R
    ON R.id = F.id
WHERE F.ime = 'Film 3';
```

RIGHT JOIN s klauzulom dobivanje zapisa bez podudaranja

Ako želimo dobiti filmove bez režisera:

```
# neupareni zapisi unutar tablca - koji filmovi su bez režisera
SELECT *
FROM filmovi
RIGHT JOIN režiseri
    ON režiseri.id = filmovi.id
WHERE filmovi.id IS NULL;
```

Ako upitom želimo dobiti režisere bez filma:

```
SELECT *
FROM režiseri
RIGHT JOIN filmovi
    ON režiseri.id = filmovi.id
WHERE filmovi.id IS NULL;
```

LEFT JOIN / RIGHT JOIN u producijskim aplikacijama

U producijskoj aplikaciji uvijek koristim samo **LEFT JOIN** i nikad ne koristim **RIGHT JOIN**.

Radim to jer, po mom mišljenju, a **LEFT JOIN** čini upit lakšim za čitanje i razumijevanje.

Kada pišem upite, volim početi s "osnovnim" skupom rezultata, recimo svim filmovima, a zatim unijeti (ili oduzeti) grupe stvari iz te baze.

Budući da volim započeti s bazom, **LEFT JOIN** odgovara ovom načinu razmišljanja. Želim sve redove iz moje osnovne tablice ("lijeve" tablice) i uslovno želim redove iz "desne" tablice.

U praksi, mislim da nikada nisam niti video **RIGHT JOIN** u proizvodnoj aplikaciji. Nema ništa loše u **RIGHT JOIN** —samo mislim da otežava razumijevanje upita.

Ponovno pisanje **RIGHT JOIN**

Ako želimo preokrenuti naš gornji scenarij i umjesto toga vratiti sve režisere i uslovno njihove filmove, možemo lako prepisati **RIGHT JOIN** u **LEFT JOIN**.

Sve što trebamo učiniti je promijeniti redoslijed tablica u upitu i promijeniti **RIGHT** u **LEFT**:

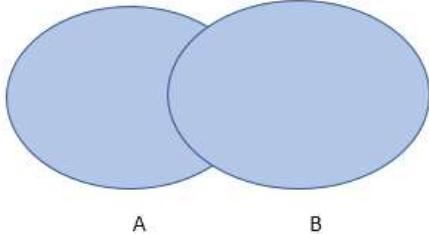
```
SELECT *
FROM režiseri
LEFT JOIN filmovi
```

```
ON filmovi.režiser_id = režiseri.id;
```

FULL OUTER JOIN - potpuno vanjsko spajanje

Sada kada imamo neke podatke s kojima možemo raditi, pogledajmo **FULL OUTER JOIN**.

FULL OUTER JOIN ne postoji u MySQL za razliku od drugih baza podataka. Moramo ga simulirati i napraviti na taj način.



U našem primjeru, naša tablica `filmovi` ima referencu na režisera preko `režiser_id` stupca, a ovaj stupac odgovara `id` stupcu tablice `režiseri`. Ovo su dva stupca koja ćemo koristiti kao uslove za spajanje.

A B

Evo kako ćemo napisati ovaj spoj između naše dvije tablice:

```
SELECT *
FROM filmovi
LEFT JOIN režiseri
  ON režiseri.id = filmovi.režiser_id
UNION
SELECT *
FROM filmovi
RIGHT JOIN režiseri
  ON režiseri.id = filmovi.režiser_id;
```

Obratite pažnju na uslov pridruživanja koji smo naveli koji povezuje film s režiserom: `ON režiseri.id = filmovi.režiser_id` Nalazi se u oba JOIN koji su spojeni sa `UNION`. Ako želimo da se iste stavke pojave više puta koristit ćemo `UNION ALL`.

Naš skup rezultata izgleda kao čudan Kartezijev produkt neke vrste:

	id	ime	režiser_id	id	ime
▶	1	Film 1	1	1	John Smith
	2	Film 2	1	1	John Smith
	3	Film 3	2	2	Sanja Ulipi
	4	Film 4	NULL	NULL	NULL
	5	Film 5	NULL	NULL	NULL
	NULL	NULL	NULL	3	Xavier Wills
	NULL	NULL	NULL	4	Bev Scott
	NULL	NULL	NULL	5	Bree Jensen

Prvi redovi koje vidimo su oni u kojima je film imao režisera, a naš uslov spajanja procijenjen je na istinit.

Međutim, nakon tih redova vidimo svaki od preostalih redova iz svake tablice — ali s `NULL` vrijednostima za koje druga tablica nije imala podudaranje.

Također vidimo još jednu razliku između `CROSS JOIN` i ove sumulacije `FULL OUTER JOIN` ovdje. `FULL OUTER JOIN` vraća jedan zaseban red iz svake tablice — za razliku od one `CROSS JOIN` koja ima više.

SELF JOIN - samospajanje

SELF JOIN je obično spajanje a ovdje se tablica spaja sama sa sobom. Demonstrirat ćemo kako implementirati **SELF JOIN** koristeći klauzule **INNER JOIN** i **LEFT JOIN**, pružajući jasne i koncizne primjere za prikaz različitih scenarija u kojima se **SELF JOIN** može primijeniti.

Samospajanje se vrši pročešljanjem redova iz iste tablice na temelju nekih specifičnih uslova. Za izvođenje SELF JOIN Alias imena se koristi s uslovom spajanja na temelju stupaca koji definiraju odnos. Nema ključne riječi SELF JOIN, već je to običan JOIN gdje se tablica pridružuje sama sebi. Ovo spajanje se obično koristi za stvaranje hijerarhije pojednostavljivanjem dohvaćanja međusobno povezanih podataka unutar iste tablice.

Sintaksa je sljedeća:

```
SELECT *
FROM tablica_ime AS t1
JOIN tablica_ime AS t2 ON t1.stupac_ime = t2.stupac_ime;
```

Za konkretan primjer definirat ćemo novu tablicu zaposleni i u nju ubaciti par vrijednosti:

```
CREATE TABLE zaposleni(
    zaposleni_id INT PRIMARY KEY,
    ime varchar(30),
    manager_id INT,
    FOREIGN KEY (manager_id) references zaposleni(zaposleni_id)
);

insert into zaposleni values
(1, 'Daniel', null);
insert into zaposleni values
(2, 'Dario', 1);
insert into zaposleni values
(3, 'Sanja', 2);
insert into zaposleni values
(4, 'Dinko', 1);
insert into zaposleni values
(5, 'Smiljka', 3);

SELECT * from zaposleni;
```

zaposleni_id	ime	manager_id
1	Daniel	NULL
2	Dario	1
3	Sanja	2
4	Dinko	1
5	Smiljka	3

MySQL SELF JOIN korištenjem INNER JOIN klauzule

U ovom primjeru, e1 i e2 su aliasi iste tablice (`zaposleni`) i spajamo stupce `manager_id` i `zaposlen_id` iste tablice koristeći unutrašnje spajanje (`INNER JOIN`) koje uključuje sve podatke koji su zajednički iz e1 i e2 aliasa. Budući da ovdje koristimo unutrašnje spajanje, svi redovi iz tablice zaposlenih s e1 i e2 bit će prikazani u rezultatu samo ako odgovaraju uslovu `e1.manager_id = e2.zaposlen_id` bit će prikazan u rezultatu.

```
SELECT e1.zaposlen_id AS zaposlen_id,
       e1.zaposlen_ime AS Radnik,
       e2.zaposlen_ime AS 'Njegov Manager'
  FROM zaposleni AS e1
 INNER JOIN zaposleni AS e2
    ON e1.manager_id = e2.zaposlen_id;
```

zaposlen_id	Radnik	Njegov Manager
2	Dario	Daniel
3	Sanja	Dario
4	Dinko	Daniel
5	Smiljka	Sanja

Nije moguće napraviti spajanje bez i e1 i e2.

MySQL SELF JOIN korištenjem LEFT JOIN klauzule

U OVOM primjeru, e1 i e2 su aliasi iste tablice i spajamo stupce `manager_id` i `employee_id` iste tablice pomoću lijevog spajanja (`LEFT JOIN`) koje uključuje sve podatke iz aliasa e1. Budući da ovdje koristimo lijevo spajanje, svi redovi iz tablice zaposlenih s e1 bit će prikazani u rezultatu i samo odgovarajući redovi iz iste tablice s aliasom e2 na temelju uslova `e1.manager_id = e2.zaposlen_id` bit će prikazani u rezultatu.

```
SELECT e1.zaposlen_id AS zaposlen_id,
       e1.zaposlen_ime AS Radnik,
       e2.zaposlen_ime AS 'Njegov Manager'
  FROM zaposleni AS e1
 LEFT JOIN zaposleni AS e2
```

zaposlen_id	Radnik	Njegov Manager
1	Daniel	NULL
2	Dario	Daniel
3	Sanja	Dario
4	Dinko	Daniel
5	Smiljka	Sanja

Moguć je i primjer sa `INNER JOIN` i `LEFT JOIN`. Moguće je i u tablici koristiti osim imena i prezime. Ako su u različitim poljima, pisali bi:

```
CONCAT(e1.zaposlen_ime, ' ', e1.zaposlen_prezime) AS Radnik
CONCAT(e2.zaposlen_ime, ' ', e2.zaposlen_prezime) AS Manager
```

Višestruki spojevi

Vidjeli smo kako spojiti dvije tablice, ali što je s višestrukim spajanjem u redu?

Zapravo je prilično jednostavno, ali da bismo to ilustrirali potrebna nam je treća tablica: `ulaznice`.

Ova će tablica predstavljati prodane ulaznice za film:

```
CREATE TABLE ulaznice
```

```

    id SERIAL PRIMARY KEY,
    film_id INTEGER NOT NULL
    REFERENCES filmovi
);

INSERT INTO ulaznice(film_id) VALUES (1), (1), (3);

```

Tablica `ulaznice` ima samo `id` i referencu na film: `movie_id`.

Također smo ubacili dvije ulaznice prodane za film ID 1 i jednu prodanu ulaznicu za film ID 3.

Sada se spojimo `režisere` s `filmovi`, a zatim `filmovi` i `ulaznice`.

```

SELECT *
FROM režiseri
INNER JOIN filmovi
    ON filmovi.režiser_id = režiseri.id
INNER JOIN ulaznice
    ON ulaznice.film_id = filmovi.id;

```

Budući da su ovo unutrašnji spojevi (engl. inner join), redoslijed kojim pišemo spojeve nije bitan. Mogli smo početi s `ulaznice`, zatim spojiti `filmovi`, a zatim spojiti `režiseri`.

Opet se svodi na ono što pokušavate postaviti upit i što čini upit najrazumljivijim.

U našem skupu rezultata primjetit ćemo da smo dodatno uzeli redove koji se vraćaju:

	id	ime	id	ime	režiser_id	id	film_id
1	John Smith	1	Film 1	1	1	1	1
1	John Smith	1	Film 1	1	2	1	
2	Sanja Ulipi	3	Film 3	2	3	3	3

Ovo ima smisla jer smo dodali još jedan `INNER JOIN`. To u stvari dodaje još jedan uslov "AND" našem upitu.

Naš upit u biti glasi: "*vraća sve režisere koji pripadaju filmovima koji također imaju prodaju ulaznica*".

Ako bismo umjesto toga željeli pronaći režisere koji pripadaju filmovima koji možda još nisu još uvijek prodali ulaznice, mogli bismo zamijeniti naš posljednji `INNER JOIN` s `LEFT JOIN`:

```

SELECT *
FROM režiseri
INNER JOIN filmovi
    ON filmovi.režiser_id = režiseri.id
LEFT JOIN ulaznice
    ON ulaznice.film_id = filmovi.id;

```

Vidimo da se **Film 2** sada vratio u skup rezultata:

	id	ime	id	ime	režiser_id	id	film_id
1	John Smith	1	Film 1	1	1	1	1
1	John Smith	1	Film 1	1	2	1	
2	Sanja Ulipi	3	Film 3	2	3	3	3
1	John Smith	2	Film 2	1	NULL	NULL	

Za ovaj film nije bilo prodanih ulaznica, pa je prethodno isključen iz skupa rezultata zbog **INNER JOIN**.

Riješite za vježbu kako pronaći režisere koji pripadaju filmovima koji nemaju ulaznice u prodaji?

Spajanje s dodatnim uslovima

Slično **WHERE** klauzuli, našim uslovima spajanja možemo dodati onoliko uslova koliko želimo.

Na primjer, ako želimo pronaći filmove s režiserima koji se ne zovu "John Smith", mogli bismo dodati taj dodatni uslov našem spoju s **AND**:

```
SELECT *
FROM filmovi
INNER JOIN režiseri
ON režiseri.id = filmovi.režiser_id
AND režiseri.ime <> 'John Smith';
```

id	ime	režiser_id	id	ime
3	Film 3	2	2	Sanja Ulipi

Možemo koristiti sve operatore koje bismo stavili u **WHERE** klauzulu u ovom uslovu spajanja.

Dobivamo isti rezultat iz ovog upita ako **WHERE** umjesto toga stavimo uslov u klauzulu:

```
SELECT *
FROM filmovi
INNER JOIN režiseri
ON režiseri.id = filmovi.režiser_id
WHERE režiseri.ime <> 'John Smith';
```

<https://www.freecodecamp.org/news/sql-joins-tutorial/#left-join-right-join>

<https://www.geeksforgeeks.org/mysql-inner-join/?ref=lbp>

MySQL Podupiti

Podupit je ugrađen unutar drugog upita i djeluje kao ulaz ili izlaz za taj upit. Podupiti se također nazivaju unutarnji upiti i mogu se koristiti u raznim složenim operacijama u SQL-u.

Podupiti pomažu u izvršavanju upita koji ovise o izlazu drugog upita. Podupiti su u zagradama.

MySQL podupit se može koristiti s vanjskim upitom koji se koristi kao ulaz u vanjski upit. Može se koristiti s klauzulama SELECT, FROM i WHERE. MySQL podupit se prvo izvršava prije izvršenja vanjskog upita.

Postavimo okolinu

Prije pisanja upita napravimo jednostavne tablice za izvođenje operacija. Napravit ćemo dvije tablice **Radnici** i **Odjeli**.

```
CREATE TABLE radnici (
    radnik_id INT PRIMARY KEY,
    ime VARCHAR(50) NOT NULL,
    plaća NUMERIC(10),
    Odjel VARCHAR(20)
);

CREATE TABLE Odjeli (
    odjel_id INT PRIMARY KEY,
    odjel_ime VARCHAR(50) NOT NULL
);
```

Dodajmo neke vrijednosti u ove tablice.

```
INSERT INTO Odjeli (odjel_id, odjel) VALUES
    (1, 'IT'),
    (2, 'Računovodstvo'),
    (3, 'Tehnička podrška');
```

odjel_id	odjel
1	IT
2	Računovodstvo
3	Tehnička podrška

```
INSERT INTO radnici (radnik_id, ime, plaća, Odjel) VALUES
    (100, 'Sanja Ulipi', 2000, "Prodaja"),
    (101, 'Branko Ćelap', 3000, "IT"),
    (102, 'Smiljka Božičković', 2400, "IT");
```

radnik_id	ime	plaća	Odjel
100	Sanja Ulipi	2000	Prodaja
101	Branko Ćelap	2800	IT
102	Smiljka Božičković	2400	IT

Podupit s **WHERE** klauzulom

Odaberimo radnike iz odjela koji imaju **odjel_id** jednak 1.

```
SELECT *
FROM Radnici
WHERE odjel=(SELECT odjel FROM Odjeli WHERE odjel_id=1);
```

Iz podupita dobijemo IT zato što je u odjelu samo **odjel_id** = 1. Cijeli upit vratit će:

radnik_id	ime	plaća	Odjel
101	Branko Ćelap	2800	IT
102	Smiljka Božičković	2400	IT

Podupit s operatorima usporedbe

Operator manje od ($<$)

Odaberimo radnike čija je plaća manja od prosječne plaće svih zaposlenika.

```
SELECT *
FROM Radnici
WHERE plaća < (SELECT avg(plaća) from Radnici)
```

radnik_id	ime	plaća	Odjel
101	Branko Ćelap	2800	IT
102	Smiljka Božičković	2400	IT
NULL	NULL	NULL	NULL

Podupit vraća 2400. Upit vraća radnike koji imaju plaću manju od tog iznosa.

Podupit s operatorom veće od ($>=$)

Odaberemo radnike čija je plaća veća ili jednaka prosječnoj plaći svih radnika

```
SELECT *
FROM Radnici
WHERE plaća >= (SELECT avg(plaća) from Radnici)
```

radnik_id	ime	plaća	Odjel
101	Branko Ćelap	2800	IT
102	Smiljka Božičković	2400	IT

Podupit opet vraća 2400. Kako tražimo radnike čija je plaća veća ili jednaka, takav rezultat i dobijemo.

Podupit s **IN** i **NOT IN** operatorima

IN operator

Odaberimo sve radnike čiji se odjel nalazi u tablici odjela.

```
SELECT *
FROM Radnici
WHERE odjel IN (SELECT odjel FROM Odjeli);
```

radnik_id	ime	plaća	Odjel
101	Branko Ćelap	2800	IT
102	Smiljka Božičković	2400	IT

Podupit vrati sve odjele a upit sve radnike koji su u tim odjelima.

NOT IN operator

Odaberimo sve radnike čiji odjel nije u tablici odjela.

```
SELECT *
FROM Radnici
WHERE odjel NOT IN (SELECT odjel FROM Odjeli);
```

radnik_id	ime	plaća	Odjel
100	Sanja Ulipi	2000	Prodaja

Podupit s klauzulom FROM

Ovaj podupit omogućava odabir svih odjela iz tablice `Radnici` s ugniježdenim upitom.

```
SELECT odjel
FROM (SELECT * from Radnici) AS A;
```

Odjel
Prodaja
IT
IT

Podupit vraća cijelu tablicu `Radnici` sa svim stupcima a vanjski a upit iz nje vadi samo odjel.

Podupit u korelaciji

Podupit u korelaciiji (engl. Correlated subquery) je onaj koji koristi stupce iz **vanjske tablice** za izvršenje.

Odaberimo `radnik_id` i `ime` radnika iz `Radnici` gdje je plaća manja od prosječne plaće, a odjel je isti kao vanjska tablica.

```
SELECT radnik_id , ime
FROM Radnici AS A
WHERE plaća < (SELECT avg(plaća) from Radnici AS B WHERE A.odjel = B.odjel);
```

radnik_id	ime
102	Smiljka Božičković

Ovaj upit će prvo dohvatiti prosječnu plaću ovisno o nazivu odjela u dvije tablice, a zatim odabrati radnike s plaćom manjom od prosječne plaće.

*Podupit s EXISTS i NOT EXISTS**EXISTS*

Odaberimo radnike za koje postoji barem 1 odjel gdje je odjel radnika isti kao odjel u odjelima.

```
SELECT radnik_id , ime
FROM Radnici
WHERE EXISTS (SELECT 1 FROM Odjeli WHERE Odjeli.odjel = Radnici.odjel);
```

radnik_id	ime
101	Branko Ćelap
102	Smiljka Božičković

NOT EXISTS

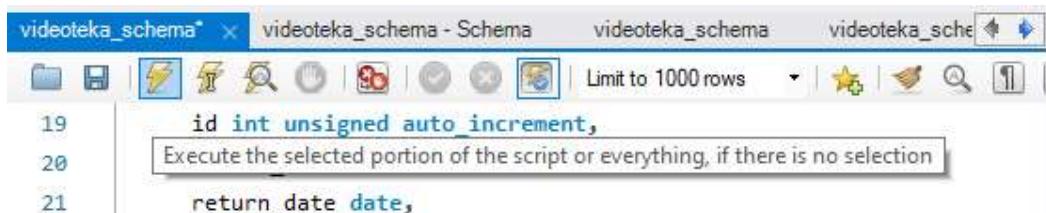
Odaberimo radnike za koje ne postoji barem 1 odjel gdje je odjel zaposlenika isti kao odjel u odjelima.

```
SELECT radnik_id , ime
FROM Radnici
WHERE NOT EXISTS (SELECT 1 FROM Odjeli WHERE Odjeli.odjel = Radnici.odjel);
```

radnik_id	ime
100	Sanja Ulipi

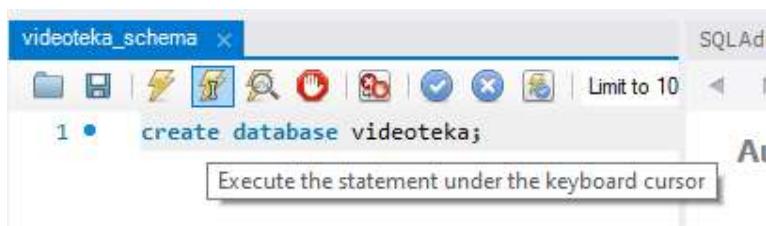
Upit dohvaća ID i ime radnika iz tablice Radnici gdje niti jedan odjel u tablici Odjeli ne odgovara odjelu radnika, provjeravajući nepostojanje takvih odjela pomoću podupita.

My SQL WorkBench načini snimanja sheme



```
videoteka_schema* | videoteka_schema - Schema | videoteka_schema | videoteka_sche | 
19 | id int unsigned auto_increment, | 
20 | Execute the selected portion of the script or everything, if there is no selection | 
21 | return_date date,
```

Izvršite odabrani dio skripte ili sve, ako nema odabira



```
videoteka_schema | videoteka_schema - Schema | videoteka_schema | videoteka_sche | SQLAd | 
1 • | create database videoteka; | 
2 | Execute the statement under the keyboard cursor |
```

Izvrši naredbu ispod kursora na tastaturi



```
videoteka_schema* | videoteka_schema - Schema | videoteka_schema | videoteka_sche | 
36 | price_list ID int unsigned | 
37 | prima | Execute the EXPLAIN command on the statement under the cursor |
```

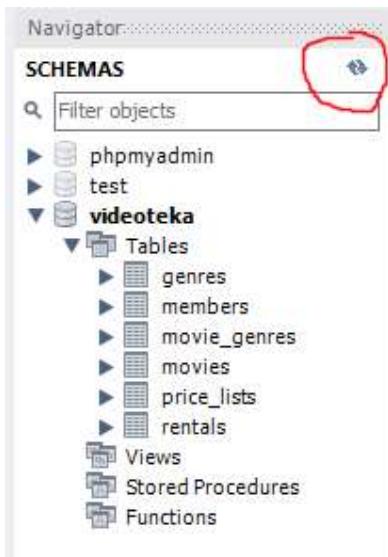
Izvrši naredbu EXPLAIN na iskazu ispod kursora

Vježba SQL – DDL ključne riječi

- Na temelju prošle vježbe kreirajte bazu podataka s pripadajućim entitetima i njihovim atributima.
- Koristite SQL - DDL ključne riječi.

`create database videoteka;`

Dok ne kliknemo na ikonu na Refresh ikonu nećemo vidjeti bazu videoteka.



ako kažemo:

```
drop database videoteka;
```

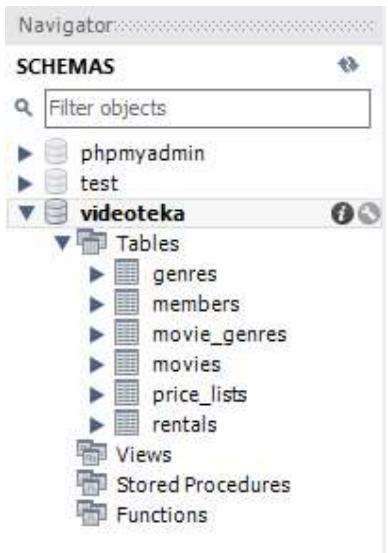
uklonit ćemo bazu. Ako to probamo još jednom, nećemo moći jer je već uklonjena.

Ako prvi red za kreiranje promijenimo u:

```
create database if not exists videoteka;
```

dobit ćemo upozorenja ako baza videoteka postoji. U pravilu kada dobijemo grešku (engl. error) skripta se prestaje izvršavati.

Kada kliknemo na info baze (ikona s desne strane imena baze), otvorit će se informacije o samoj bazi. Tu je najbitniji charset jer ako je krivo postavljen, u bazu će se upisivati krivi znakovi i kada ih budemo izvlačili iz baze, nazad ćemo dobiti hijeroglife.



Query 1 videoteka_schema videoteka x

Info Tables Columns Indexes Triggers Views Stored Procedures Functions Grants Events

MariaDB
videoteka

Schema Details

Default collation: **utf8mb4_general_ci**
 Default characterset: **utf8mb4**
 Table count: **6**
 Database size (rough estimate): **160.0 KiB**

Ovo je upravo kako treba za naš jezik. Multibajt zapis i možemo unutar teksta pohranjivati i emoji-je. Ako nam postavke ne odgovaraju, možemo kliknuti na ikonu ključa (vidi predzadnju sliku). Kada to pritisnemo, otvorit će se pokraj prozor:

Query 1 videoteka_schema videoteka videoteka - Schema x

Create a NEW TABLE or VIEW or ALTER existing table in this schema

Name: videoteka Specify the name of the schema

Rename References Refactor model, changing all references found in view, triggers,

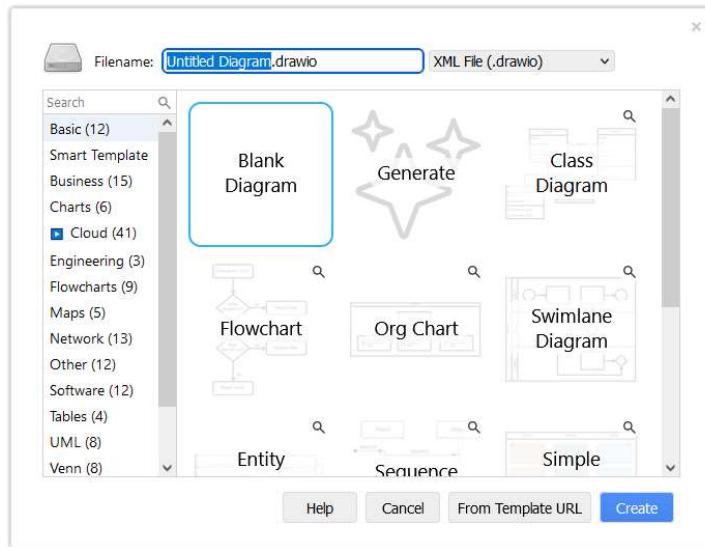
Charset/Collation: utf8mb4 utf8mb4_0900_ci The character set and its

Tu možemo promijeniti charset i treba ga potvrditi s Applay. To je moguće i napraviti kroz SQL izdefinirati:

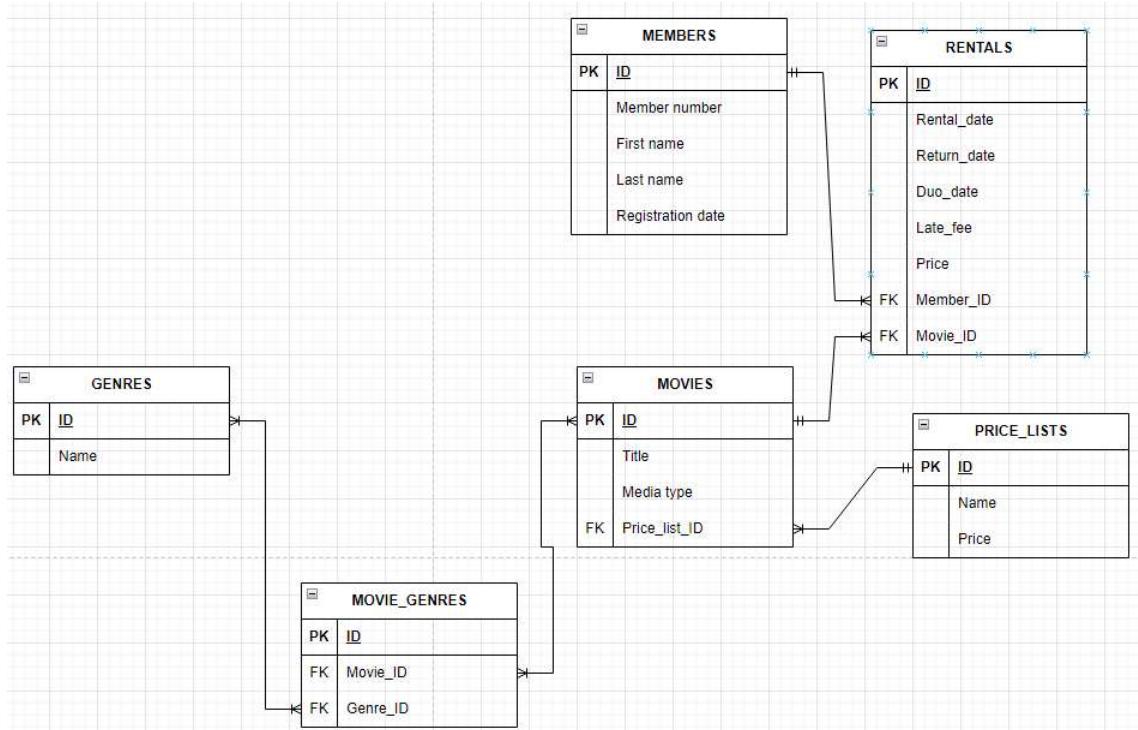
```
create database if not exists videoteka
character set utf8mb4 collate utf8mb4_general_ci;
```

Ova naredba prelomljena je u dva reda. Jedna naredba je sve do točka-zareza (;) to je delimiter koji odvaja naredbe u SQL-u.

Nakon kreiranja baze, kreiramo tablice. To ne možemo napraviti ako ne otvorimo bazu sa `use videoteka;` Vratimo se [app.diagrams.net](#) i pogledamo dijagram koji smo nacrtali.



Kada krećemo kreirati tablice, treba paziti na strane ključeve. Ne možemo kreirati tablicu `RENTALS` bez da imamo tablicu `MEMBERS`. Ne možemo kreirati tablicu `RENTALS` bez da imamo tablicu `MOVIES`. Ni tablicu `MOVIES` ne možemo kreirati dok nemamo tablicu `PRICE_LIST`. Bitan je redoslijed kako kreiramo tablice. Ako želimo automatski prilikom kreiranja tablice kreirati strane ključeve, tada smo dužni poštivati redoslijed. Drugi pristup je da kreiramo tablice a strane ključeve naknadno kreiramo s `alter table`.



Prvo ćemo kreirati tablicu `members`.

```
create table members(
```

```

        id int unsigned auto_increment,
        member_number char(10) unique not null,
        first_name varchar(50) not null,
        last_name varchar(50) not null,
        registration_date date not null,
        primary key(id)
);

```

`unsigned` uz `int` znači da želimo samo pozitivne vrijednosti. `auto_increment` govori da se automatski uvećava brojač. `char` označava znak fiksne dužine. `unique` označava da vrijednost mora biti jedinstvena. Ako je podatak obavezan onda stavimo `not null`.

možemo pisati `id int unsigned auto_increment primary key`, tako da primarni ključ deklariramo u istom redu a možemo i napisati kao iznad `id int unsigned auto_increment`, i na kraju `primary key(id)`.

Ako kliknemo na videoteka bazu i tablice s lijeve strane u donjem dijelu ispod toga trebali bi smo vidjeti

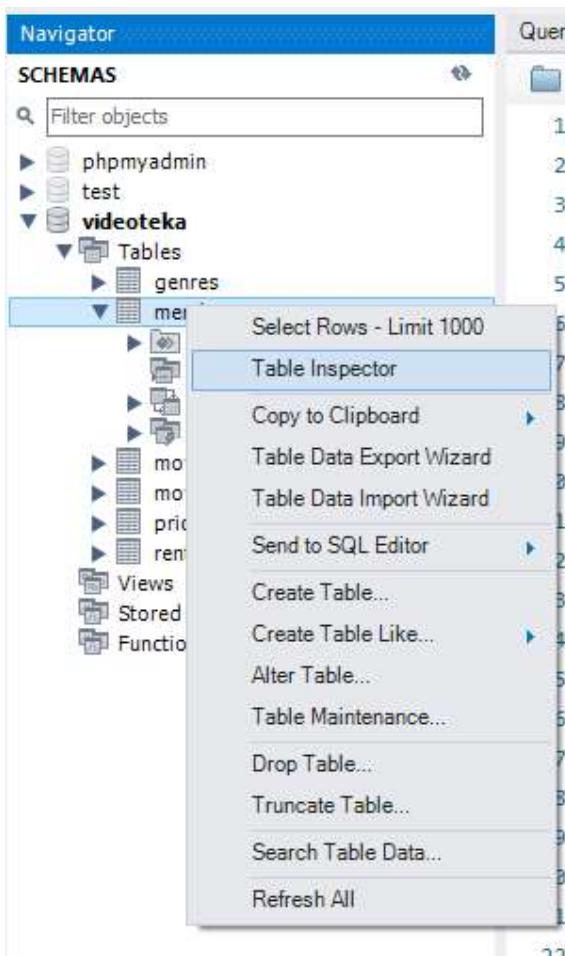
Column	Type	Default Value	Nullable	Char
id	int(10) unsigned		NO	
member_number	char(10)		NO	utf8r
first_name	varchar(50)		NO	utf8r
last_name	varchar(50)		NO	utf8r
registration_date	date		NO	

popis polja koje se nalazi u tablici.

Možemo i pokrenuti Table Inspector.

Column	Type	Default Value	Nullable	Char
id	int(10) unsigned		NO	
member_number	char(10)		NO	utf8r
first_name	varchar(50)		NO	utf8r
last_name	varchar(50)		NO	utf8r
registration_date	date		NO	

Ako je `Nullable YES`, to znači da je polje opcionalno. Da ne bi tako bilo treba dodati `not null`. To naravno ne vrijedi za polje primarnog ključa. Ako naknadno dodamo polje primarnog ključa, MySQL Workbench će ispraviti polje da ne može biti `Nullable`.



Predavač savjetuje korištenje `varchar` a ne `char` polja.

`decimal (10,2)` znači da 8 znamenki ide ispred zareza.

Ovdje je moguće pisati i komentare. Višelinjski komentar `/* */` ide kao u HTML a jednolinijski `--`.

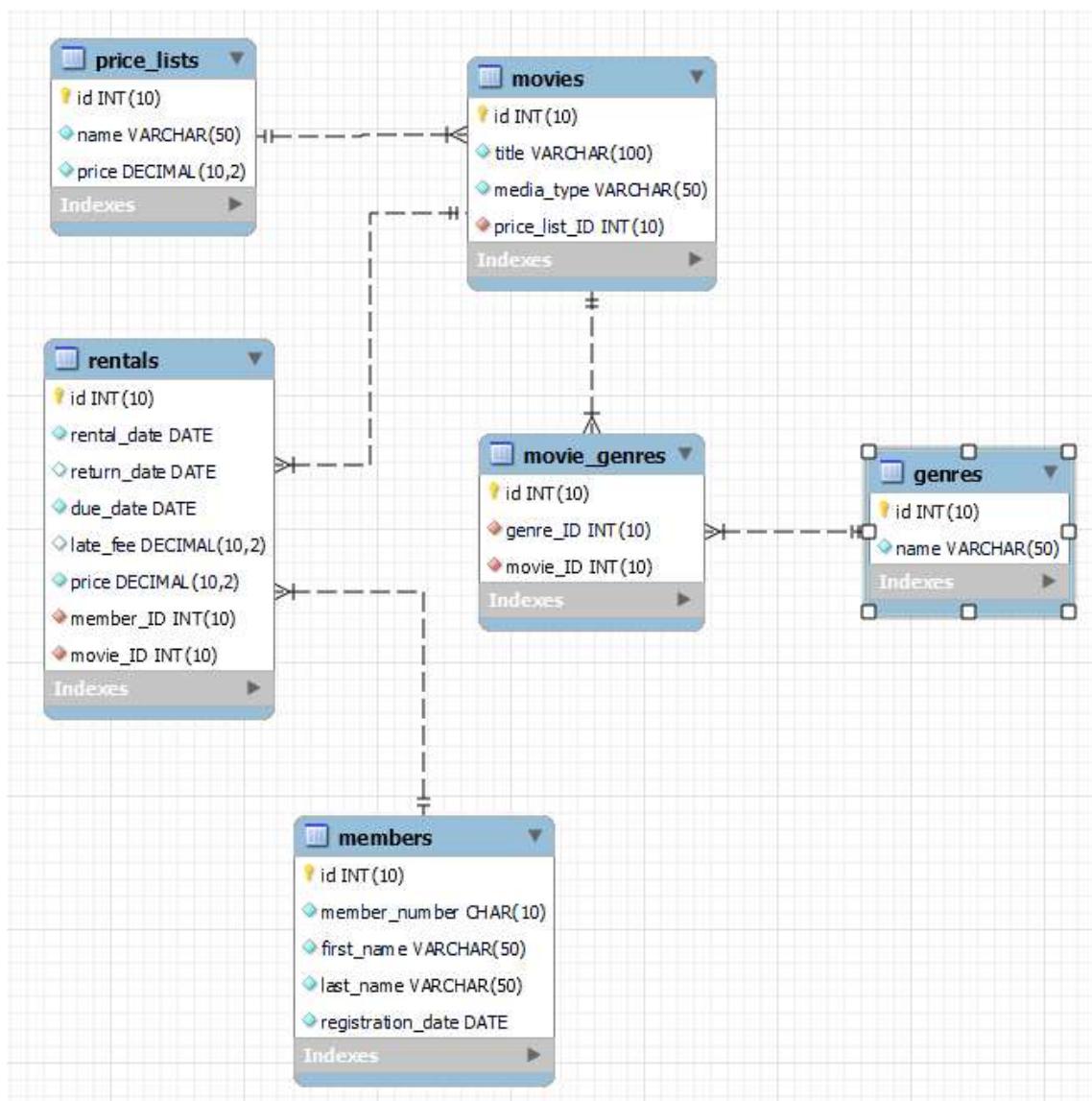
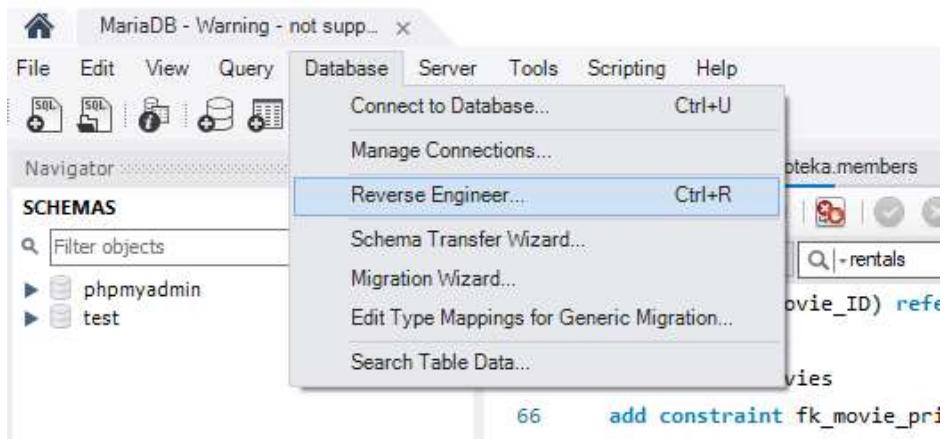
Tablice se pišu u množini. Primarni ključ je jednina (neko polje). Strani ključ je jednina, donja crta, pa njezin primarni ključ.

Kod stvaranja stranog ključa trebamo se ispravno referencirati na tablicu i njen primarni ključ:

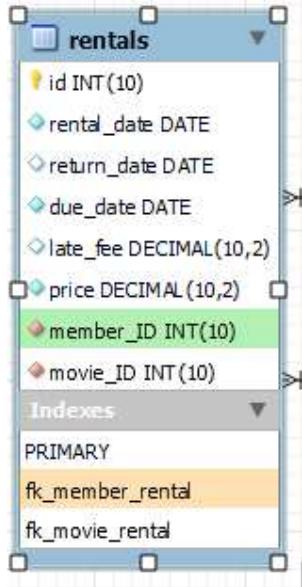
```
add constraint fk_member_rental
foreign key (members_ID) references members(id)
```

Ovo znači da strani ključ eferencira `members_ID` rpolje (stupac) na tablicu `member` u kojoj je primarni ključ polje (stupac) `id`. Preko imena navedenog u constraint, pohranjujemo ga u shemu, kako bi kasnije preko ograničenja (tog imena) mogli kontrolirati brisanje ili ažuriranje nekog podatka. Obično se navodi fk, donja crtica, pa ime tablica (u jednini). Kod imena tablica prvo ona na koju referenciramo, pa iz koje tablice.

Provjera iz SQL u dijagram shemu (Reverse Engenering)



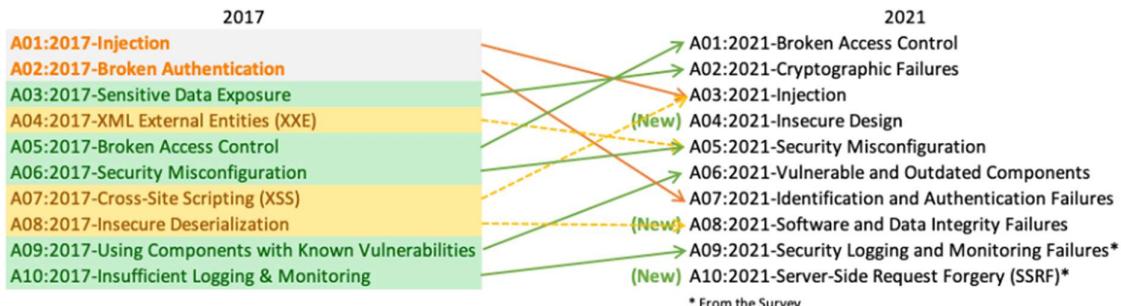
Iz ove generirane sheme vidimo indekse, strane ključeve, koji su stupci (polja) obavezni, koji nisu obavezni, itd. Kod primarnog ključa je žuti ključić. Svetloplavi rombovi pokazuju koja su polja obavezna. Romb koji ima samo obrub znači da je polje samo opcionalno. Ako je obrub narandžasti to su strani ključevi. Aklo na donjem dijelu tablice kliknemo na donji dio gdje piše Indexes vidimo indekse. Npr.:



Kako se mičemo gore dolje po PRIMARY, fk_member_rental, fk_movie_rental zelena oznaka ide po id, member_ID i movie_ID. Dakle vidimo povezanos indeksa sa stranim ključem.

Pogledajmo još jednom ograničenja (engl. constraints). Koristili smo `not null` koje ne dozvoljava da bude `null` vrijednost, tj. da je ne pošaljemo. `unique` brani da budu dva ista podatka. Ako ima previše znamenki (više od onoga što je određeno u polju) baza će odsjeći znamenke iza.

Potrebno je paziti na sigurnost, pogotovo na Cross-site Scripting tj. Injection. Na prvom mjestu je neispravna kontrola pristupa.



```
drop database if exists videoteka;

create database if not exists videoteka
character set utf8mb4 collate utf8mb4_general_ci;

use videoteka;

create table members(
    id int unsigned auto_increment,
    member_number char(10) unique not null,
    first_name varchar(50) not null,
    last_name varchar(50) not null,
    registration_date date not null,
    primary key(id)
);

create table rentals (
    id int unsigned auto_increment,
    rental_date date not null,
    return_date date,
    due_date date not null,
    late_fee decimal(10,2) default(0.00),
    price decimal(10,2) not null,
    member_ID int unsigned not null,
    movie_ID int unsigned not null,
    primary key (id)
);

create table movies (
    id int unsigned auto_increment,
    title varchar(100) not null,
    media_type varchar(50) not null,
    price_list_ID int unsigned not null,
    primary key (id)
);

create table price_lists (
    id int unsigned auto_increment,
    name varchar(50) not null,
    price decimal(10,2) not null,
    primary key (id)
);
```

```
create table genres (
    id int unsigned auto_increment,
    name varchar(50) not null,
    primary key (id)
);

create table movie_genres (
    id int unsigned auto_increment,
    genre_ID int unsigned not null,
    movie_ID int unsigned not null,
    primary key (id)
);

alter table rentals
add constraint fk_member_rental
foreign key (member_ID) references members(id),
add constraint fk_movie_rental
foreign key (movie_ID) references movies(id);

alter table movies
add constraint fk_movie_price_list
foreign key (price_list_ID) references price_lists(id);

alter table movie_genres
add constraint fk_movie_genre_genre
foreign key (genre_ID) references genres(id),
add constraint fk_movie_genre_movie
foreign key (movie_ID) references movies(id);

insert into members
(member_number, first_name, last_name, registration_date)
values
('2285566430', 'Ana', 'Anić', '2024-05-17');

insert into members
(member_number, first_name, last_name, registration_date)
values
('2285566431', 'Pero', 'Anić', '2024-05-17');

select * from members;
```

Ovo briše određeni slog koji odaberemo:

```
delete from members  
where id = 1;
```

Bez where briše sve i to je mogući problem!

Ovime ažuriramo podatak:

```
update members  
set member_number = null  
where id = 1;
```

Vježba SQL - rad sa `SELECT`

Predavač je pripremio novu bazu [adventureworkshop.sql](#). To je dump baze koji možemo uvući u PHPMyAdmin tako da kažemo Import na meniju, Choose File, pronađemo datoteku i učitamo je. Dovoljno je odabrati dugme Import. Nako nekog vremena pojavit će se baza [adventureworks](#).



Importing into the current server

File to import:

File may be compressed (gzip, bzip2) or uncompressed.
A compressed file's name must end in `.[format].[compression]`. Example: `.sql.zip`

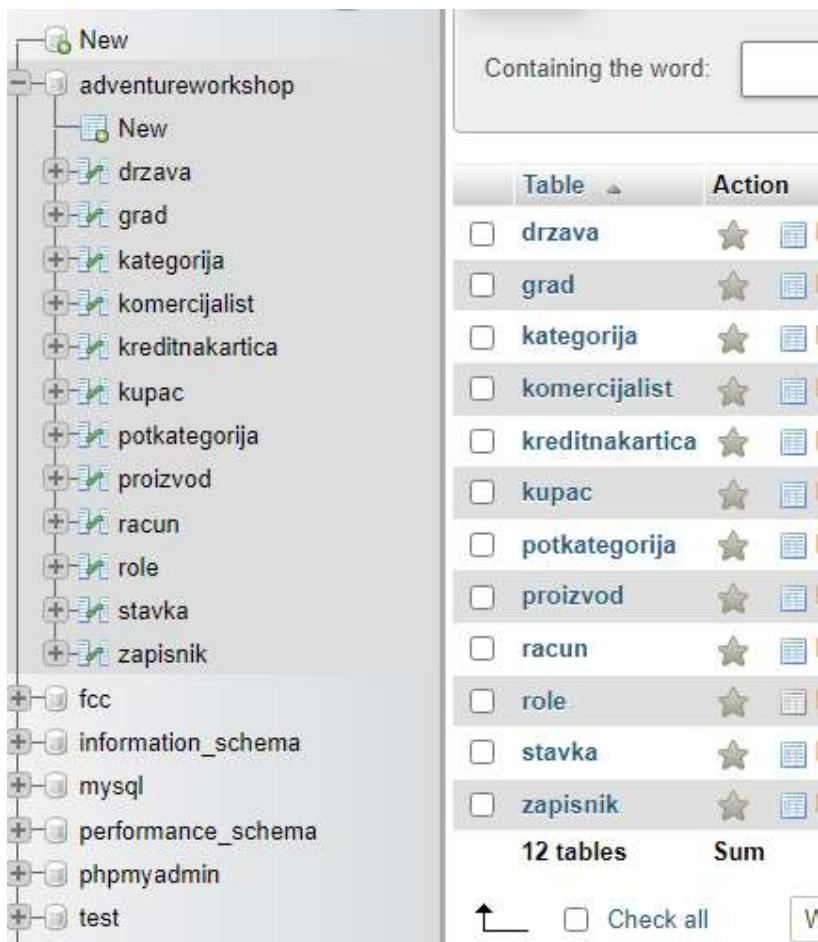
Browse your computer: (Max: 40MB)

Choose File No file chosen

You may also define character set of the imported data.

Character set of the imported data: utf-8

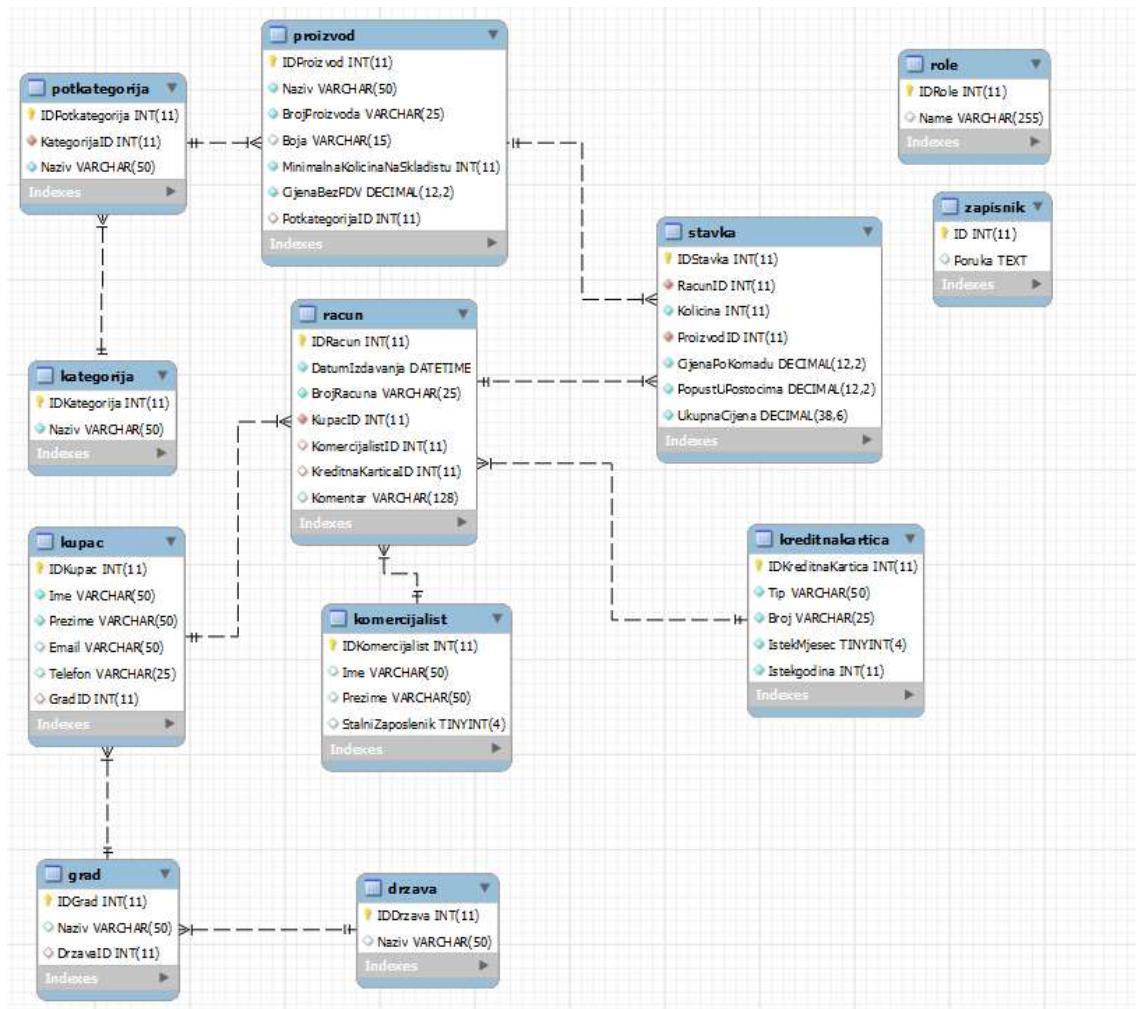
Partial import:



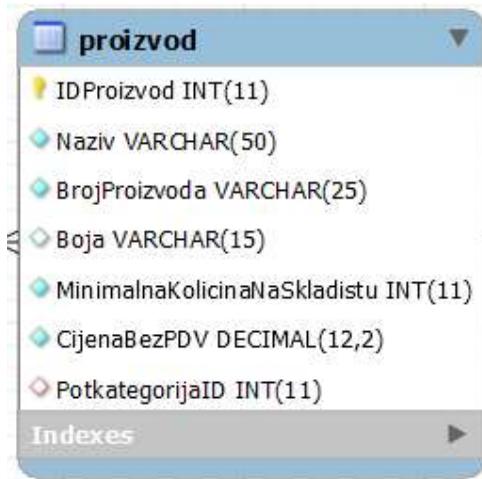
Otvorit ćemo MySQLWorkbench i odabratim skriptu za snimanje Save Script As (ili Control+Shift+S). Datoteku ćemo nazvati [basic_queryies.sql](#). Prvo što trebamo napraviti je upoznati se sa bazom. Pogledat ćemo T-SQL dijagram (sintaksnii dijagram). Za to ćemo koristiti dijagram koji možemo dobiti sa MySQL Workbench-om. Sve riječi pisane velikim slovima su rezervirane riječi. To ćemo napraviti s Reverse Engineer (ili Ctrl+R).

Iz dijagrama vidimo da postoji tablica podkategorija koja je vezana za tablicu proizvod. Tablica proizvod ima na sebi strani ključ PodkategorijaID. Ono što ćemo primjetiti je da taj strani ključ nije obavezan jer može biti proizvoda koji nemaju kategoriju.

IDProizvod	Naziv	BrojProizvoda	Boja	MinimalnaKolicinaNaSkladistu	CijenaBezPDV	Potk
1	Adjustable Race	AR-5381	NULL	750	0.00	NULL
2	Bearing Ball	BA-8327	NULL	750	0.00	NULL
3	BB Ball Bearing	BE-2349	NULL	600	0.00	NULL
4	Headset Ball Bearings	BE-2908	NULL	600	0.00	NULL
316	Blade	BL-2036	NULL	600	0.00	NULL
317	LL Crankarm	CA-5965	Crna	375	0.00	NULL
318	ML Crankarm	CA-6738	Crna	375	0.00	NULL
319	HL Crankarm	CA-7457	Crna	375	0.00	NULL
320	Chainring Bolts	CB-2903	Srebrna	750	0.00	NULL
321	Chainring Nut	CN-6137	Srebrna	750	0.00	NULL
322	Chainring	CR-7833	Crna	750	0.00	NULL
323	Crown Race	CR-9981	NULL	750	0.00	NULL
324	Chain Stays	CS-2812	NULL	750	0.00	NULL
...



To smo mogli i vidjeti i iz dijagrama koji je ljubičast što znači da se radi o stranom ključu.



Krenimo s upitima:

```
use adventureworkshop;

-- Dohvati sve proizvode
select * from proizvod;

-- Dohvati drugih 5 proizvoda
select * from proizvod limit 5;

-- Dohvati drugih 5 proizvoda
-- ili preskoči prvih 5 (offset) i uzmi idućih 5 (limit)
select * from proizvod limit 5 offset 5;

-- Dohvati drugih 5 proizvoda
-- preskoči prvih (offset) i uzmi idućih 5 (limit)
select * from proizvod limit 5,5;
```

```
-- Izbroj proizvode u svakoj podkategoriji
select PotkategorijaID, count(*) as BrojProizvoda
from proizvod
group by PotkategorijaID;
```

Ako pokrenemo prva dva reda dobit ćemo count (*) 504, dakle bez pod kategorija:

PotkategorijaID	BrojProizvoda
NULL	504

To je netočan podatak je reki proizvodi imaju kao `PotkategorijaID` vrijednost `NULL`.

Kada dodamo treći red dobivamo ispravan podatak:

PotkategorijaID	BrojProizvoda
NULL	209
1	32
2	43
3	22
4	8
5	3
6	2
7	1
8	3
9	2
10	3
11	3
12	28
13	7
14	33
15	9
16	18
17	14

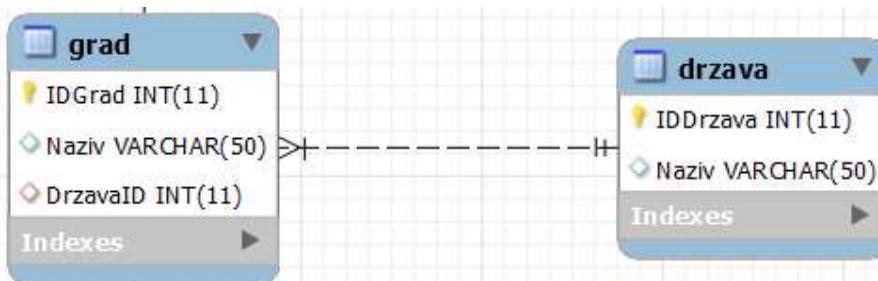
Primjetimo da je `count(*)` odmah iza `select`. U prvom redu dodan je `as BrojProizvoda` kako ne bi smo dobili naziv stupca `(*)`. Svakom stupcu moguće je dati alias sa `as`.

Izbrojmo ukupnu prodaju za svakog komercijalistu

```
select KomercijalistID, count(*) as UkupnaProdaja
from racun
group by KomercijalistID;
```

Evo primjera `CROSS JOIN`:

Imamo tablicu `grad` i tablicu `drzava`. Između njih postoji veza stranog ključa i primarnog ključa.



```
-- ispiši sve gradove s imenom države
select grad.Naziv as Grad, drzava.Naziv as Država
from grad, drzava;
```

Rezultat koji dobijemo je povezivanje svakog sa svakim i to ne daje dobre rezultate:

Grad	Država
Zagreb	Hrvatska
Zagreb	Njemačka
Zagreb	Bosna i Hercegovina
Zagreb	Madarska
Osijek	Hrvatska
Osijek	Njemačka
Osijek	Bosna i Hercegovina
Osijek	Madarska

Treba uvesti WHERE:

```
select grad.Naziv as Grad, drzava.Naziv as Država
from grad, drzava
where grad.DrzavaID = drzava.IDDrzava;
```

Sada ćemo dobiti dobar rezultat:

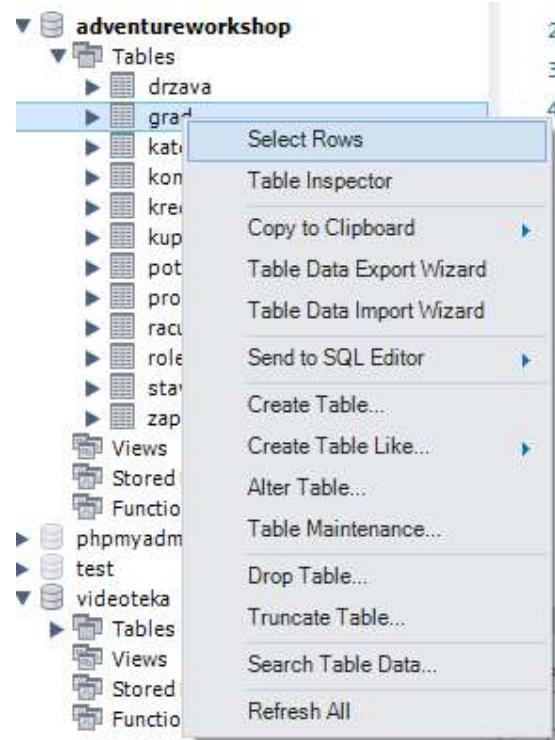
Grad	Država
Zagreb	Hrvatska
Osijek	Hrvatska
Pula	Hrvatska
Rijeka	Hrvatska
Split	Hrvatska
Bjelovar	Hrvatska
Vinkovci	Hrvatska
Koprivnica	Hrvatska

Pogledajmo ovaj isti primjer sa INNER JOIN, dakle gdje se tablice sijeku.

Pogledat ćemo što će se desiti ako je stalni ključ opcionalan. Ako ne nađe matching par kod CROSS JOINa imamo problem jer nećemo dobiti potpune podatke. Kako u tablici grad imamo sve podatke unijet ćemo jedan ručno. Na tablici grad odabratи desnom tipkom Select Rows i otvorit će se upit i ispod sadržaj tablice.

Napravimo zapis:

```
insert into grad (Naziv)
values ('Donja Motičina')
```



Na dnu smo dobili upravo poziciju koju smo željeli:

IDGrad	Naziv	DrzavaID
12	Kakanj	3
13	Derventa	3
14	Bjelovar	1
15	Vinkovci	1
16	Koprivnica	1
17	Budimpe...	5
18	Pecuh	5
19	Eger	5
20	Donja M...	NULL
NULL	NULL	NULL

Ako ponovimo zadnji upit nećemo dobiti Donju Motičinu jer nema matching para i **CROSS JOIN** ovdje pada. Isto je i sa **INNER JOIN**.

Dakle isti zadatak sa **INNER JOIN**:

```
select grad.Naziv as Grad, drzava.Naziv as Drzava
from grad
join drzava
on grad.DrzavaID = drzava.IDDrzava;
```

Nije potrebno pisati **INNER JOIN**, dovoljno je **JOIN**. Ovdje nemamo **WHERE** već koristimo **ON**. Dobili smo isti rezultat kao i prije:

Grad	Drzava
Rijeka	Hrvatska
Split	Hrvatska
Bjelovar	Hrvatska
Vinkovci	Hrvatska
Koprivnica	Hrvatska
Berli	Njemačka
Dresde	Njemačka
Frankfurt	Njemačka
Sarajevo	Bosna i ...
Zenica	Bosna i ...

U nastavku i dalje nema Donja Motičina.

Što ako želimo dohvatiti sve podatke iz jedne tablice, match-ati ih s podacima iz druge tablice. Ako ne postoji par u drugoj tablici, zadržati ono što se nalazi u prvoj tablici. Kod match-anja ostaviti vrijednost **NULL**. Npr. izvući Donja Motičinu i pod državu ispisati **NULL**. Za ovo koristimo **LEFT JOIN** i spojimo s presjekom A i B tablice, višak tablice dobacimo.

LEFT JOIN i **RIGHT JOIN** koristimo ovisno o upitu. Imamo grad koji nije vezan za državu. Tablica A (lijeva) je uvijek FROM. Desna je uvijek u **JOIN**. Kod **LEFT JOIN** će sve kod tablice grad ispisati a ono što se ne poklapa ispisaće se kao **NULL**.

```
select grad.Naziv as Grad, drzava.Naziv as Država
from grad
```

```
left join drzava
on grad.DrzavaID = drzava.IDDrzava;
```

Sada vidimo da je Donja Motičina na popisu:

Grad	Država
Mostar	Bosna i Hercegovina
Kakanj	Bosna i Hercegovina
Derventa	Bosna i Hercegovina
Bjelovar	Hrvatska
Vinkovci	Hrvatska
Koprivnica	Hrvatska
Budimpešta	Madarska
Pecuh	Madarska
Eger	Madarska
Donja M...	NULL

Kada bi na istom primjeru napravili RIGHT JOIN, opet ne bi bilo Donje Motičine:

```
select grad.Naziv as Grad, drzava.Naziv as Država
from grad
right join drzava
on grad.DrzavaID = drzava.IDDrzava;
```

Grad	Država
Dresde	Njemačka
Frankfurt	Njemačka
Sarajevo	Bosna i ...
Zenica	Bosna i ...
Mostar	Bosna i ...
Kakanj	Bosna i ...
Derventa	Bosna i ...
Budimpešta	Madarska
Pecuh	Madarska
Eger	Madarska

Kada bi obrnuli tablice opet bi vidjeli Donju Motičinu:

```
select grad.Naziv as Grad, drzava.Naziv as Država
from drzava
right join grad
on grad.DrzavaID = drzava.IDDrzava;
```

FULL OUTER JOIN u MySQL-u ne radi. U biti trebali bi dobiti spojene podatke (koji se match-aju) iz lijeve i desne strane. **FULL JOIN** se postiže ovako:

```
select grad.Naziv as Grad, drzava.Naziv as Država
from grad
left join drzava
on grad.DrzavaID = drzava.IDDrzava
union
select grad.Naziv as Grad, drzava.Naziv as Država
from grad
```

```
right join drzava  
on grad.DrzavaID = drzava.IDDrzava;
```

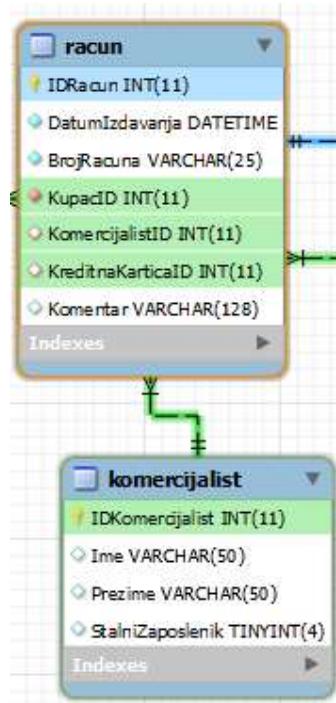
Dakle u rezultatu vidimo i podatke koji su NULL i iz jedne i iz druge tablice.

Moguće je dobiti i sve podatke koji nisu u presjeku, dakle svi podaci iz A i svi podaci iz B koji nemaju match para.

```
-- FULL OUTER JOIN with exclusion (bez presjeka A i B)  
select grad.Naziv as Grad, drzava.Naziv as Drzava  
from grad  
left join drzava  
on grad.DrzavaID = drzava.IDDrzava  
where grad.DrzavaID is null  
union  
select grad.Naziv as Grad, drzava.Naziv as Drzava  
from grad  
right join drzava  
on grad.DrzavaID = drzava.IDDrzava  
where grad.DrzavaID is null ;
```

Primjer

Ispisat ćemo sve račune s imenom i prezimenom komercijaliste. Koristit ćemo umjesto punog imena tablice skraćeno dva početna slova. Riješit ćemo to s [INNER JOIN](#). Dobivamo rezultat presjeka tablica.



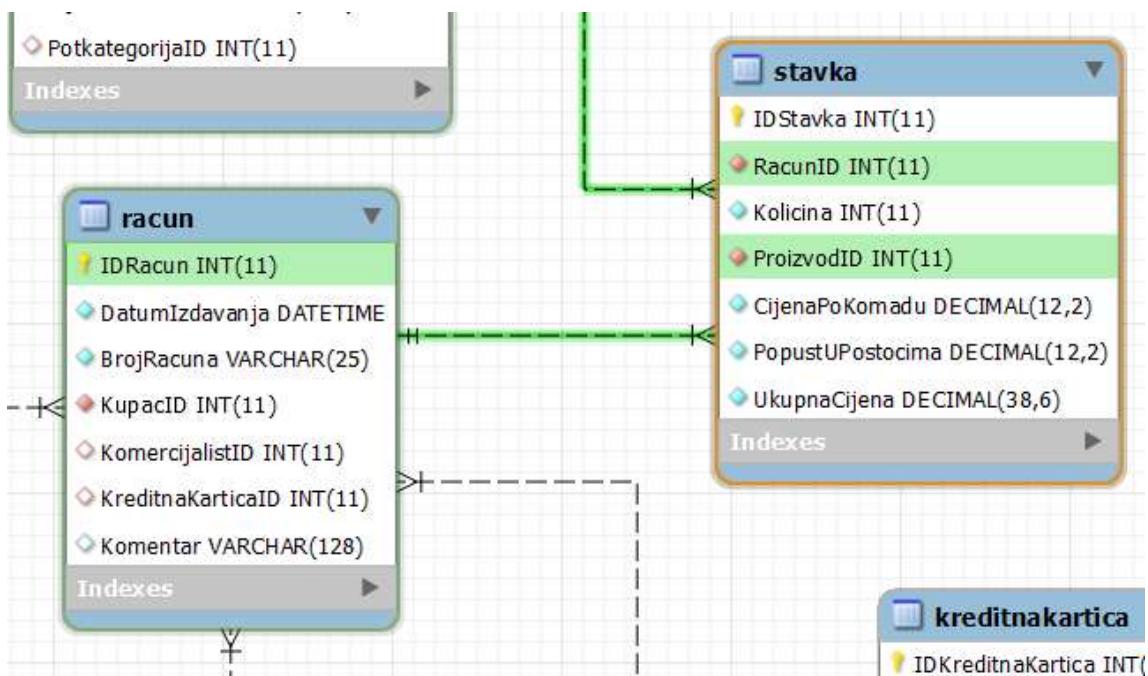
U aliasima često koristimo jedno ili dva slova početnih slova tablice. Ovdje ćemo koristiti dva slova.

```
select ra.*, ko.IMe, ko.Prezime
from racun as ra
join komercijalist as ko
on ra.KomercijalistID = ko.IDKomercijalist;
```

Možemo pisati **JOIN** umjesto **INNER JOIN**, jednako radi.

IDRacun	DatumIzdavanja	BrojRacuna	KupacID	KomercijalistID	KreditnaKarticaID	Komentar	IMe	Prezime
43849	2001-08-01 00:00:00	SO43849	371	268	14559	NULL	Miro	Mirić
44082	2001-09-01 00:00:00	SO44082	354	268	8552	NULL	Miro	Mirić
44508	2001-11-01 00:00:00	SO44508	312	268	11717	NULL	Miro	Mirić
45317	2002-02-01 00:00:00	SO45317	263	268	13468	NULL	Miro	Mirić
45526	2002-03-01 00:00:00	SO45526	354	268	8552	NULL	Miro	Mirić
45546	2002-03-01 00:00:00	SO45546	303	268	6955	NULL	Miro	Mirić
45814	2002-04-01 00:00:00	SO45814	8	268	8937	NULL	Miro	Mirić
46334	2002-06-01 00:00:00	SO46334	269	268	12698	NULL	Miro	Mirić
46369	2002-06-01 00:00:00	SO46369	213	268	12135	NULL	Miro	Mirić
46613	2002-07-01 00:00:00	SO46613	320	268	11441	NULL	Miro	Mirić

Želimo dohvatiti sve račune, te ispisati broj računa s ukupnim brojem stavki po računu.



Ispisat ćemo samo broj računa a ne i ostatak sadržaja tablice. Možemo imati više stavki za svaki račun pa ćemo ih morati grupirati jer nemamo broj stavki u tablici **racun**. Kako možemo imati račun bez stavki, moramo obuhvatiti sve račune a ne samo one koji imaju stavke, dakle ide **LEFT JOIN**. Sa **count** brojimo stavke iz stavki računa. Da bi **count** točno prebrojao moramo grupirati stavke po **ra.IDRacun**.

```
select ra.BrojRacuna, count(st.IDStavka)
from racun as ra
left join stavka as st
on ra.IDRacun = st.RacunID
group by ra.IDRacun;
```

BrojRacuna	count(st.IDStavka)
SO43659	12
SO43660	2
SO43661	15
SO43662	22
SO43663	1
SO43664	8
SO43665	10
SO43666	6
SO43667	4
SO43668	29

Ako želimo promijeniti ime stupca stavimo alias. Ako želimo sortirati po broju stavki ali od najveće ka najmanjoj:

```
select ra.BrojRacuna, count(st.IDStavka) as BrojStavki
from racun as ra
left join stavka as st
```

```
on ra.IDRacun = st.RacunID  
group by ra.IDRacun  
order by BrojStavki desc;
```

BrojRacuna	BrojStavki
SO51739	72
SO51721	72
SO53465	71
SO51160	71
SO47355	68
SO51120	67
SO57046	67
SO55297	66
SO47395	66
SO51090	66

Ako želimo prvih 5:

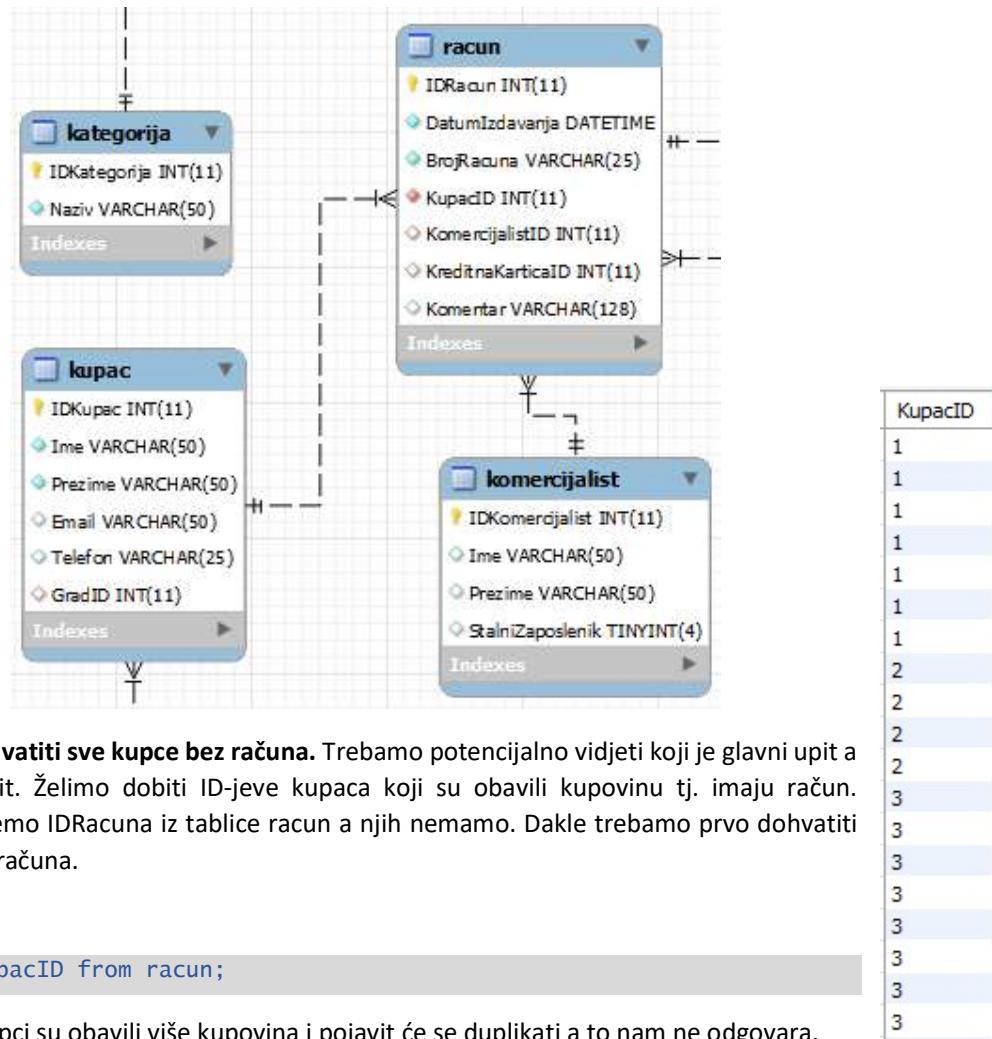
```
select ra.BrojRacuna, count(st.IDStavka) as BrojStavki  
from racun as ra  
left join stavka as st  
on ra.IDRacun = st.RacunID  
group by ra.IDRacun  
order by BrojStavki desc limit 5;
```

BrojRacuna	BrojStavki
SO51739	72
SO51721	72
SO53465	71
SO51160	71
SO47355	68

Vježba- SELECT sa podupitom

Upit se oslanja na rezultat drugog upita (podupit). Na osnovu njega dobijemo rješenje na upit.

Idemo to konkretnizirati. Gledamo odnos tablica [racun](#) i [kupac](#).



Želimo dohvatiti sve kupce bez računa. Trebamo potencijalno vidjeti koji je glavni upit a koji podupit. Želimo dobiti ID-jeve kupaca koji su obavili kupovinu tj. imaju račun. Dohvatit ćemo IDRacuna iz tablice racun a njih nemamo. Dakle trebamo prvo dohvatiti KupacID iz računa.

```
select KupacID from racun;
```

Pojedini kupci su obavili više kupovina i pojavit će se duplikati a to nam ne odgovara.

Želimo jedinstvene zapise koji nisu redundantni. To ćemo napraviti sa `distinct` da bi dobili svaki samo jednom:

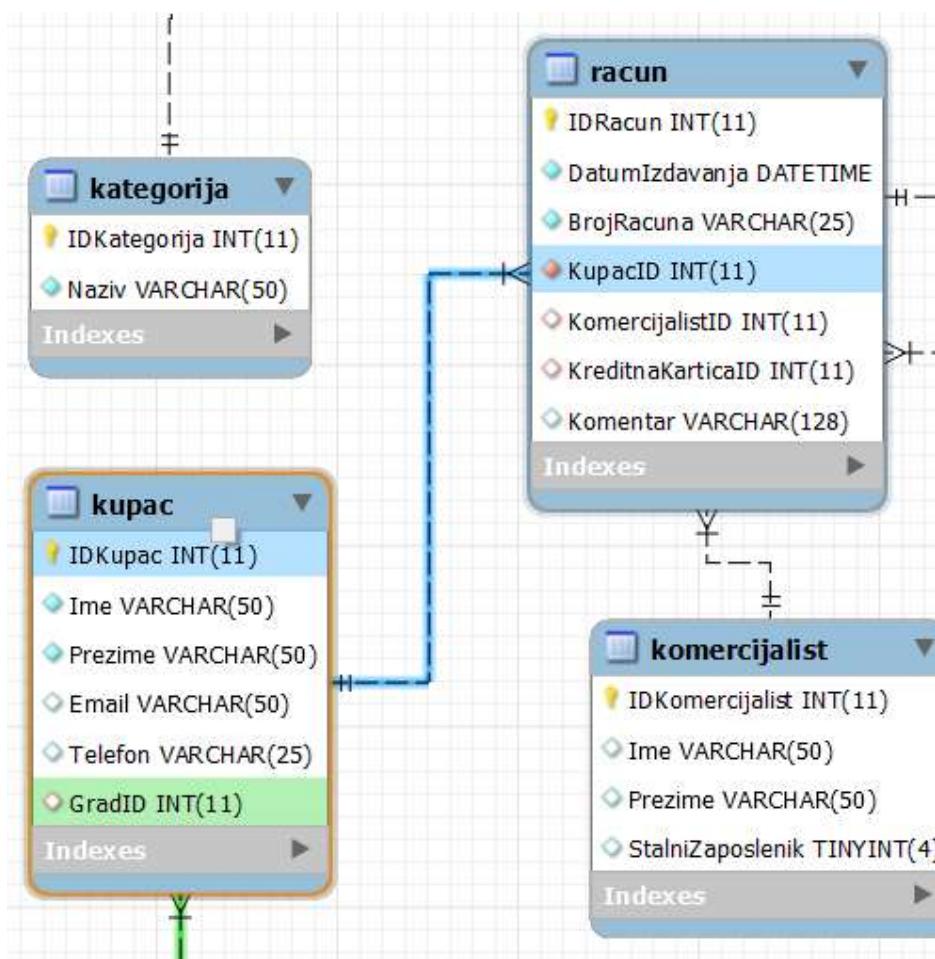
```
select distinct KupacID from racun
```

Dobili smo sve kupce koji imaju račun a trebaju nam računi koji ga nemaju. To ćemo iskoristiti kao podupit (engl. subquery) i sada možemo napisati cijeli upit. Želimo da dohvativamo sve kupce čiji se ID ne nalazi u ovom podskupu podataka koji je vratio ovaj podupit.

```
select * from kupac
where IDkupac
not in (select distinct KupacID from racun);
```

Želimo pogledati iz glavnog skupa koji kupci se ne nalaze u ovom podskupu.

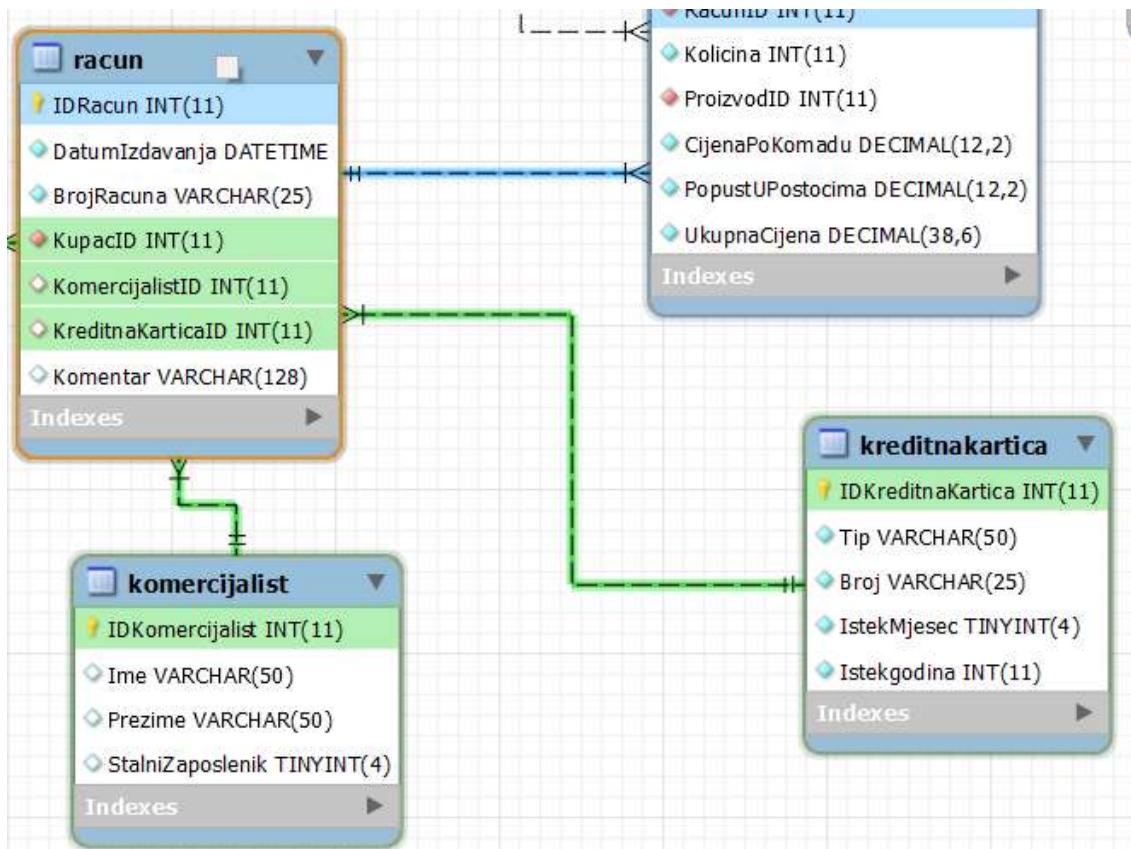
IDKupac	Ime	Prezime	Email	Telefon	GradID
16	Lili	Alameda	lili0@Adventure-works.com	1 (11) 500 555-0150	3
53	Alberto	Baltazar	alberto0@Adventure-works.com	780-555-0114	3
55	Darrell	Banks	darrell0@Adventure-works.com	816-555-0118	8
56	Angela	Barbariol	angela1@Adventure-works.com	134-555-0182	11
57	David	Barber	david9@Adventure-works.com	752-555-0115	2
63	Norma	Barrera	norma0@Adventure-works.com	1 (11) 500 555-0187	1
67	Ciro	Bauer	ciro0@Adventure-works.com	115-555-0170	7
68	Glenna	Beansto	glenna0@Adventure-works.com	1 (11) 500 555-0113	8
72	Bradley	Beck	bradley0@Adventure-works.com	1 (11) 500 555-0127	1
79	Ido	Ben-Sachar	ido0@Adventure-works.com	973-555-0112	4
84	Richard	Bentley	richard0@Adventure-works.com	911-555-0153	4
85	Maria	Berch	marian0@Adventure-works.com	640-555-0198	10
86	Kare	Berge	karen2@Adventure-works.com	813-555-0173	1
89	Kris	Bergi	kris0@Adventure-works.com	144-555-0175	1
90	Andreas	Berglund	andreas1@Adventure-works.com	795-555-0116	8
98	Mary	Bishop	mary4@Adventure-works.com	176-555-0136	4
105	Michael	Bohling	michael12@Adventure-works.com	838-555-0147	12
120	Carol	Brink	carol1@Adventure-works.com	513-555-0149	6
141	Ingrid	Burkhardt	ingrid0@Adventure-works.com	857-555-0187	7



Sve ovo možemo napisati sa **LEFT JOIN** jer želimo kupce koji nisu nešto kupili.:

```
select * from kupac as ku
left join racun ra
on ra.KupacID = ku.IDKupac
where ra.IDRacun is null;
```

Tražimo komercijaliste kod kojih je račun plaćen kreditnom karticom



Probat ćemo riješiti s podupitom. Prvo probamo napisati podupit.

```
select KomercijalistID
from racun
where KreditnaKarticaID is not null
```

Opet dobijemo duplike koje trebamo ukloniti.

```
select distinct KomercijalistID
from racun
where KreditnaKarticaID is not null
```

Dobili smo podskup i to će poslati podupit.

Tražimo da li je IDKomercijalist u podskupu.

IDKomercijalist	Ime	Prezime	StalniZaposlenik
268	Miro	Mirić	0
275	Ana	Anić	1
276	Lidija	Anić	1
277	Paško	Pašić	1
278	Ranko	Rankić	1
279	Hrvoje	Hrvić	1
280	Jurko	Jurić	1
281	Ferdo	Ferić	1
282	Tea	Tejić	1
283	Marko	Markić	1
284	Jura	Juričević	0
285	Tina	Tinić	1
286	Franka	Franjić	1
287	Garfield	Mačković	1
288	Odi	Pesić	0
289	Vinko	Vinkić	1
290	Mirka	Mirkić	1
NULL	NULL	NULL	NULL

Postavlja se pitanje ima li komercijalista koji nemaju ni jednu prodaju.

Ovdje fali dio.

Otvorit ćemo novu datoteku [advanced_queries.sql](#) i snimiti je.

Ispisat ćemo nazive gradova koji imaju više od 20 kupaca:

```
select gr.Naziv, count(ku.IDKupac) as BrojKupaca
from grad as gr, kupac as ku
where gr.IdGrad = ku.GradID
group by ku.GradID;
```

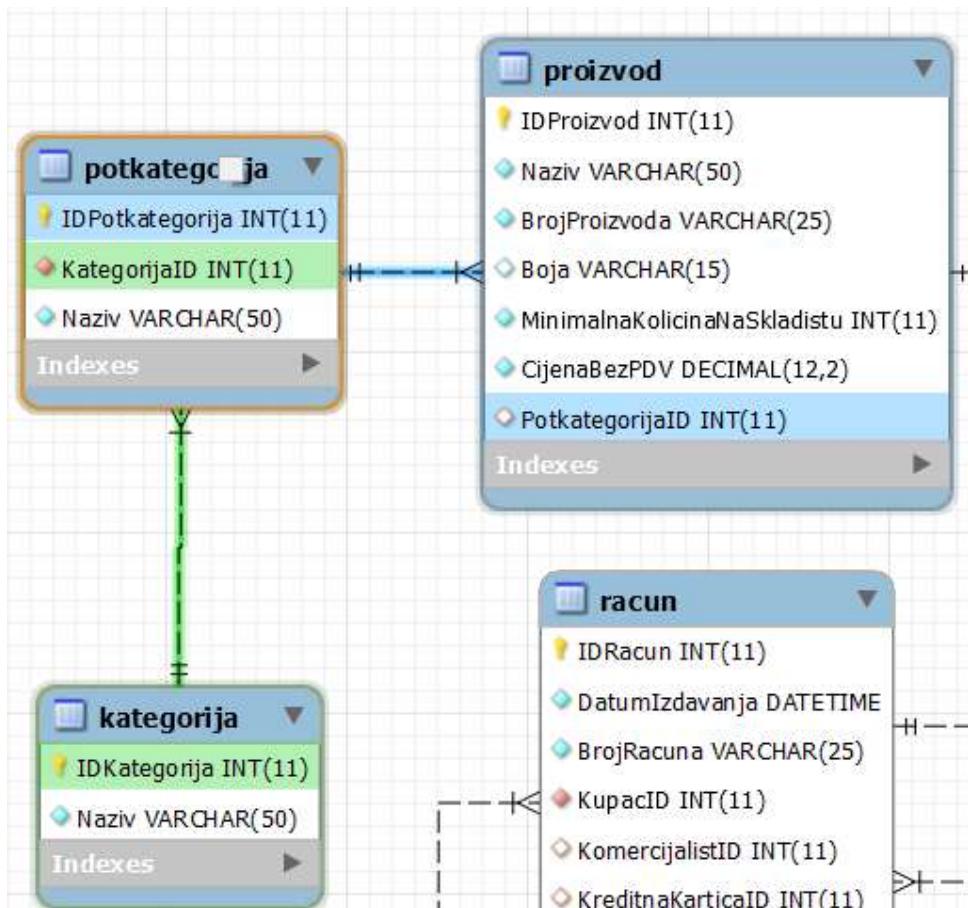
Naziv	BrojKupaca
Zagreb	1492
Osijek	1552
Pula	1506
Rijeka	1601
Split	1553
Berli	1488
Dresde	1556
Frankfurt	1497
Sarajevo	1497
Zenica	1580
Mostar	1562
Kakanj	1537
Derventa	1550

Kada želimo grupu rezultata filtrirati koristimo **HAVING**. Prethodno mora biti **GROUP BY**.

```
select gr.Naziv, count(ku.IDKupac) as BrojKupaca
from grad as gr, kupac as ku
where gr.IdGrad = ku.GradID
group by ku.GradID
HAVING BrojKupaca > 1500;
```

Naziv	BrojKupaca
Osijek	1552
Pula	1506
Rijeka	1601
Split	1553
Dresde	1556
Zenica	1580
Mostar	1562
Kakanj	1537
Derventa	1550

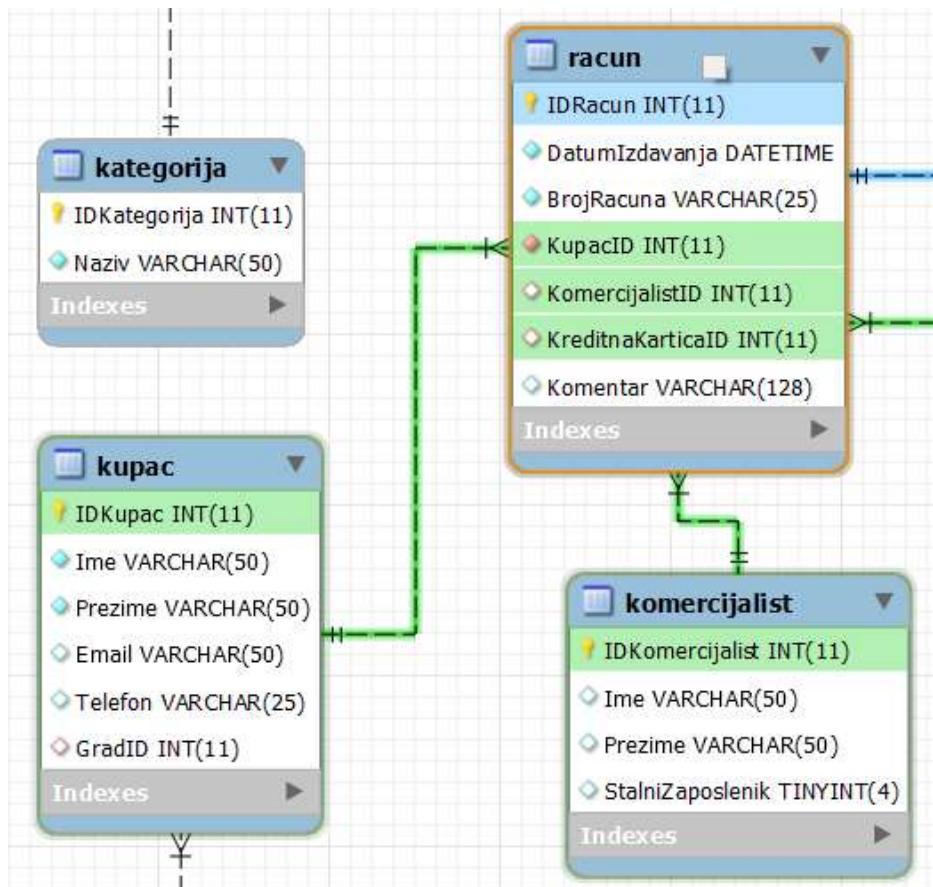
Treba ispisati sve proizvode, tj. njihov naziv s nazivom kategorije i nazivom podkategorije.



```
select
pr.Naziv as Proizvod,
pk.Naziv, ka.Naziv as Potkategorija,
ka.Naziv as Kategorija
from proizvod as pr
left join potkategorija as pk
on pr.PotkategorijaID = pk.IDPotkategorija
```

```
left join kategorija as ka  
on pk.KategorijaID = ka.IDKategorija;
```

Ispišite sve brojeve računa s imenom komercijaliste i imenom kupca.



```
select ra.BrojRacuna, ko.Ime as Komercijalist, ku.Ime as Kupac
```

```

from racun as ra
left join komercijalist as ko
on ra.KomercijalistID = ko.IdKomercijalist
left join kupac as ku
on ra.KupacID = ku.IDKupac

```

Ako želimo da se vidi i prezime komercijaliste i kupaca:

```

select
ra.BrojRacuna,
ko.Ime as KomercijalistIme,
ko.Prezime as KomercijalistPrezime,
ku.Ime as KupacIme,
ku.Prezime as KupacPrezime
from racun as ra
left join komercijalist as ko
on ra.KomercijalistID = ko.IdKomercijalist
left join kupac as ku
on ra.KupacID = ku.IDKupac

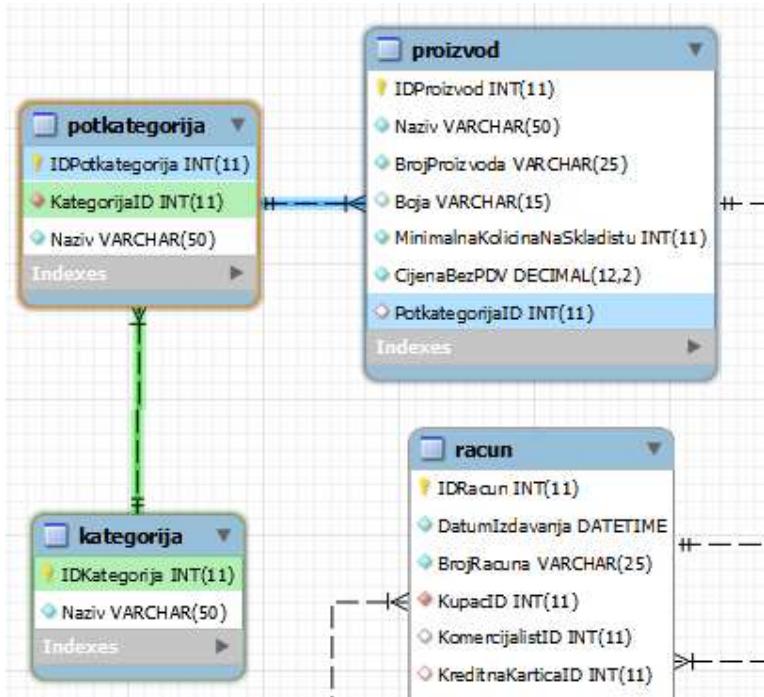
```

BrojRacuna	Komercijalist	Kupac
SO43659	Hrvoje	James
SO43660	Hrvoje	Takiko
SO43661	Tea	Jauna
SO43662	Tea	Robi
SO43663	Lidija	Jimmy
SO43664	Jurko	Sandeep
SO43665	Marko	Richard
SO43666	Lidija	Abraham
SO43667	Paško	Scott
SO43668	Tea	Rya
SO43669	Marko	Caroly
SO43670	Ana	Mae
SO43671	Marko	Peggy
SO43672	Tea	Phyllis
SO43673	Ana	Nancy
SO43674	Tea	Eric
SO43675	Paško	Valerie
SO43676	Ana	Mark
SO43677	Ranko	Brenda

BrojRacuna	KomercijalistIme	KomercijalistPrezime	KupacIme	KupacPrezime
SO43659	Hrvoje	Hrvić	James	Hendergart
SO43660	Hrvoje	Hrvić	Takiko	Collins
SO43661	Tea	Tejić	Jauna	Elso
SO43662	Tea	Tejić	Robi	McGuiga
SO43663	Lidija	Anić	Jimmy	Bischoff
SO43664	Jurko	Jurić	Sandeep	Katyal
SO43665	Marko	Markić	Richard	Bready
SO43666	Lidija	Anić	Abraham	Swearengi
SO43667	Paško	Pašić	Scott	MacDonald
SO43668	Tea	Tejić	Rya	Calafato
SO43669	Marko	Markić	Caroly	Farino
SO43670	Ana	Anić	Mae	Black
SO43671	Marko	Markić	Peggy	Justice
SO43672	Tea	Tejić	Phyllis	Thomas
SO43673	Ana	Anić	Nancy	Hirota
SO43674	Tea	Tejić	Eric	Brumfield
SO43675	Paško	Pašić	Valerie	Hendricks
SO43676	Ana	Anić	Mark	Hanso
SO43677	Ranko	Rankić	Brenda	Heaney

Zadaća

Izračunaj prosječnu cijenu proizvoda u svakoj kategoriji:



Ovdje moramo spojiti 3 tablice. Projek računamo sa `avg` a čim ga koristimo moramo grupirati proizvode s `GROUP BY`. Ignoriraju se proizvodi koji nisu u nekoj podkategoriji.

```

select ka.Naziv, avg(pr.CijenaBezPDV)
from kategorija as ka
join potkategorija as po
on ka.IDKategorija = po.KategorijaID
join proizvod as pr
on po.IDPotkategorija = pr.PotkategorijaID
group by ka.Naziv;
    
```

Naziv	avg(pr.CijenaBezPDV)
Bicikli	1586.737010
Dijelovi	469.860672
Dodaci	34.348966
Odjeća	50.991429

Napravimo isto računanje prosječnu cijenu proizvoda u svakoj kategoriji ali sa CROSS JOIN.

Stavit ćemo uslov sa `where` gdje je `ka.IDKategorija = po.KategorijaID`. Ovdje treba dodati i `and po.IDPotkategorija = pr.PotkategorijaID`. Spajamo kategorija primarni ključ, podkategorija strani ključ i opet potkategorija strani ključ.

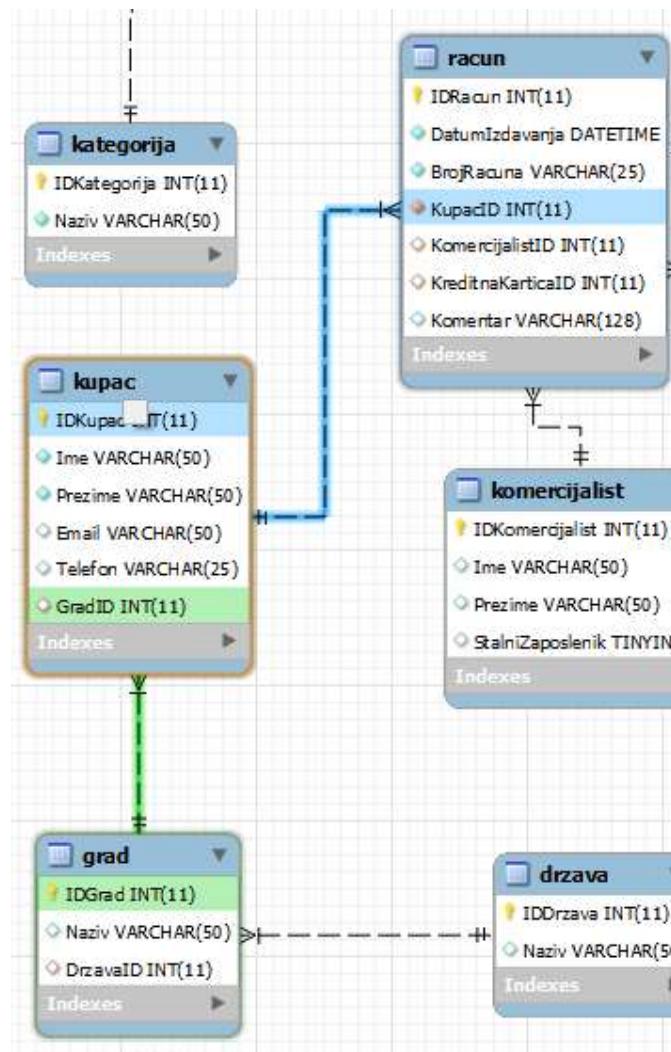
```

select ka.Naziv, avg(pr.CijenaBezPDV)
from kategorija as ka, potkategorija as po, proizvod as pr
where ka.IDKategorija = po.KategorijaID
and po.IDPotkategorija = pr.PotkategorijaID
and po.IDPotkategorija = pr.PotkategorijaID
group by ka.Naziv;
    
```

Naziv	avg(pr.CijenaBezPDV)
Bicikli	1586.737010
Dijelovi	469.860672
Dodaci	34.348966
Odjeća	50.991429

Vidimo da se prosječne cijene znatno razlikuju. Razlog tome je što **CROSS JOIN** daje više rezultata (ako uklonimo avg i group by).

Pronađi grad s najvećim brojem narudžbi (računa)



```

select gr.Naziv as Grad
from grad as gr
where gr.IDGrad =
  (select ku.GradID
   from kupac as ku
   join racun as ra
  )
  
```

```
on ku.IDKupac = ra.KupacID  
group by ku.GradID  
order by count(ra.IDRacun) desc limit 1;
```

Ovo jer rješenje sa podupitom. Podupit da 10 što odgovara Zenici.

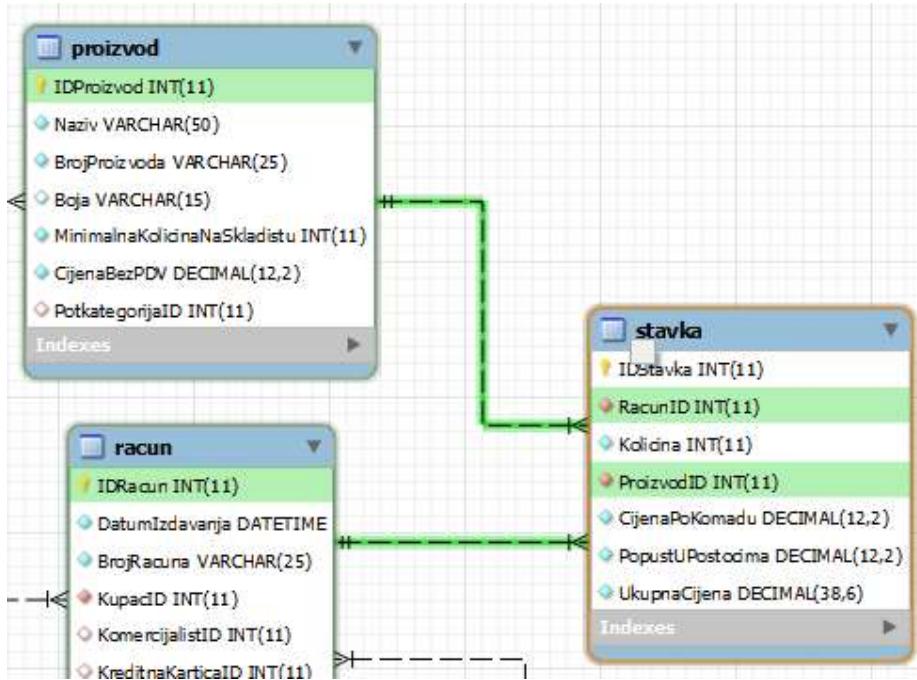
Drugo rješenje je sa višestrukim **JOIN**, bez podupita.

```
select count(ra.IDRacun) as ZbrojRacuna, gr.Naziv as Grad  
from racun as ra  
join kupac as ku  
on ra.kupacID = ku.IDkupac  
join grad as gr  
on gr.IDgrad = ku.GradID  
group by gr.IdGrad  
order by ZbrojRacuna desc limit 1;
```

Želimo dohvatiti kupce koji su obavili kupovinu u više različitih datuma.

```
select ku.Ime, ku.Prezime  
from kupac as ku  
join racun as ra  
on ku.IDKupac = ra.KupacID  
group by ku.IDKupac  
having count(distinct ra.DatumIzdavanja) > 1;
```

Prikaži razliku u količinama prodanih proizvoda između 2 najprodavanija proizvoda.



Cilj nam je kada imamo proizvod, želimo ga grupirati sa stavkama i onda napraviti grupu po proizvodu da dobijemo sve stavke vezane za proizvod i onda želimo sumirati sve proizvode.

```

select pr.Naziv, sum(st.Kolicina)
from proizvod pr
join stavka st
on pr.IDProizvod = st.ProizvodID
group by pr.IdProizvod
order by sum(st.Kolicina) desc limit 2;

```

Naziv	sum(st.Kolicina)
AWC Logo Cap	7194
Long-Sleeve Logo Jersey, L	6011

Ne treba na naziv. Prepravimo upit:

```

select pr.Naziv, sum(st.Kolicina)
from stavka st
group by pr.IdProizvod
order by sum(st.Kolicina) desc limit 2;

```

sum(st.Kolicina)
7194
6011

Rješit ćemo ovo s podupitom. U prvom upitu imat ćemo prvi zapis a u drugom upitu drugi. Evo konačnog rješenja:

```

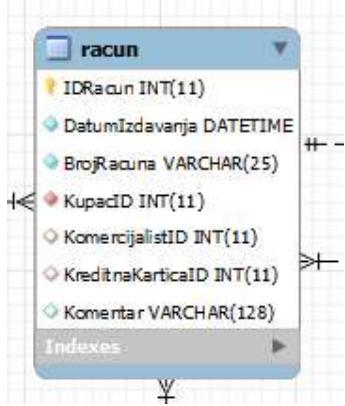
select sum(st.Kolicina) - (
    select sum(st1.Kolicina)
        from stavka st1
        group by st1.ProizvodID
        order by sum(st1.Kolicina) desc limit 1,1
) as Razlika
from stavka st

```

Razlika
1183

```
group by st.ProizvodID  
order by sum(st.Kolicina) desc limit 1;
```

Pronađi mjesec u kojem ste imali najveći volumen prodaje.



Pogledajmo prodaju po broju računa po mjesecima:

```
select month(ra.DatumIzdavanja) as Mjesec, count(ra.IdRacun) as UkupnoRacuna  
from racun as ra  
group by Mjesec  
order by UkupnoRacuna desc;
```

Mjesec	UkupnoRacuna
5	3153
6	3079
12	3014
3	2750
4	2740
2	2685
1	2483
11	2473
8	2410
9	2298
10	2282
7	2094

Pogledajmo prodaju po količini po mjesecima:

```
select month(ra.DatumIzdavanja) as Mjesec, sum(st.Kolicina) as UkupnaKolicina
```

```
from racun as ra
join stavka as st
on st.RacunID = ra.IDRacun
group by Mjesec
order by UkupnaKolicina desc;
```

Mjesec	UkupnaKolicina
8	32519
9	29405
11	26170
12	25007
7	20009
2	18730
10	18594
3	17287
4	16298
1	13798
5	11456
6	8862

Pogledajmo prodaju po prometu po mjesecima:

```
select month(ra.DatumIzdavanja) as Mjesec, sum(st.UkupnaCijena) as Promet
from racun as ra
join stavka as st
on st.RacunID = ra.IDRacun
group by Mjesec
order by Promet desc;
```

Mjesec	Promet
8	11259714.239985
11	10938760.701292
12	10464007.779045
9	9935139.516741
2	9491052.251102
3	8370268.921657
7	7410863.736317
10	6940101.173450
4	6738396.766130
5	6385440.078463
1	6075437.681933
6	4221239.351031

Uvod u MySQL Procedure

SQL procedure su skup SQL naredbi grupiranih zajedno da čine logičku jedinicu rada. One su slične funkcijama ili metodama u programskim jezicima, omogućavajući vam da složene upite i operacije sažimate u jednu jedinicu koja se može ponovno koristiti.

Procedure poboljšavaju modularnost koda, čitljivost i mogućnost održavanja, olakšavajući upravljanje i izvršavanje ponavljajućih ili zamršenih zadataka baze podataka.

- **Definicija:** MySQL procedure su pohranjene rutine koje omogućavaju ponovno korištenje SQL koda.
- **Prednosti:** Modularnost, smanjenje mrežnog prometa, poboljšanje sigurnosti.
- **Primjena:** Automatizacija redovitih zadataka, kompleksne poslovne logike, sigurnosne kontrole pristupa.
- **Kreiranje procedure:** `CREATE PROCEDURE naredba.`
- **Pozivanje procedure:** `CALL ImeProcedure();`

```
DELIMITER $$  
CREATE PROCEDURE ImeProcedure()  
BEGIN  
    -- SQL naredbe  
END $$  
DELIMITER;
```

Prvi red mijenja delimiter u `$$`. Nije obavezno u taj ali je uobičajno u MySQL-u. Proceduru počinjemo s ključnom riječi `BEGIN` a završavamo sa `END` i delimiterom. Nakon toga vratit ćemo `DELIMITER` na `;`

DETERMINISTIC ili NONDETERMINISTIC

'Rutina se smatra "determinističkom" ako uvijek proizvodi isti rezultat za iste ulazne parametre, a "nedeterminističkom" inače.' Definicije 'determinističke funkcije' iz drugih izvora također imaju "uvijek vraća isti rezultat za iste parametre".

Kada odlučujete je li zastavica `DETERMINISTIC` prikladna za pohranjenu rutinu, razmislite o tome ovako: Ako počnem s dvije identične baze podataka i izvršim svoju rutinu na obje baze podataka s istim ulaznim parametrima, hoće li moje baze podataka i dalje biti identične? Ako jesu, onda je moja rutina deterministička.

Ako izjavite da je vaša rutina deterministička iako nije, tada replike vaše glavne baze podataka možda neće biti identične originalu jer će MySQL samo dodati poziv procedure u dnevnik replikacije, a izvršavanje procedure na podređenoj bazi podataka ne daje identične rezultate.

Ako vaša rutina nije deterministička, tada MySQL umjesto toga mora uključiti pogodjene retke u dnevnik replikacije. Ako deklarirate svoju rutinu kao nedeterminističku, a nije, to neće ništa pokvariti, ali zapisnik replikacije sadržavat će sve zahvaćene redove kada bi samo poziv procedure bio dovoljan i to bi moglo utjecati na performanse.

"Deklariranje nedeterminističke rutine kao DETERMINISTIČKE može dovesti do neočekivanih rezultata uzrokujući da optimizator napravi netočne izvore plana izvršenja." Ali MySQL ne provodi niti provjerava je li vaša deklarirana deterministička rutina zapravo deterministička - MySQL vjeruje da znate što radite.

Proglasiti funkciju za `DETERMINISTIC` ili ne može imati veliki utjecaj na performanse:

```
SELECT id1, foo(id1) f1  
FROM tablica1  
ORDER BY id1  
LIMIT 1;
```

Ako je `foo()` deklariran `DETERMINISTIC`, tada se `foo()` poziva samo jednom za jedan vraćeni red.

Ako `foo()` NIJE deklariran `DETERMINISTIC`, tada se `foo()` poziva jednom za svaki red u tablici1, prije vraćanja jednog retka. Ovo može trajati dugo!

Deterministička funkcija uvijek vraća isti rezultat s istim ulaznim parametrima u istom stanju baze podataka. Npr. `POW`, `SUBSTR()`, `UCASE()`.

Nedeterministička funkcija ne vraća uvijek isti rezultat s istim ulaznim parametrima u istom stanju baze podataka. Npr. `CURDATE()`, `RAND()`, `UUID()`.

Parametri u procedurama

Parametri u procedurama mogu biti ulazni ili izlazni.

IN Parametri u Procedurama

`IN` parametri omogućavaju proslijđivanje vrijednosti proceduri kod pozivanja. Dakle procedure mogu imati ulazne parametre.

Pozivanje procedure `CALL FiltrirajPoIdu(1)` - to znači da će u `SELECT` redu `korisnikId` biti 1.

```
DELIMITER $$  
CREATE PROCEDURE FiltrirajPoIdu (IN korisnikID INT)  
BEGIN  
    SELECT * FROM korisnici WHERE id = korisnikId;  
END $$  
DELIMITER ;
```

OUT Parametri u Procedurama

- `OUT` parametri omogućavaju proceduri da vrati jednu ili više vrijednosti.
- Pozivanje procedure:

```
CALL DohvatiBrojKorisnika(@brojKorisnika);  
SELECT @brojKorisnika;
```

Za pozivanje procedure koristimo `CALL` iza kojeg slijedi naziv procedure i bilo koji ulazni parametar. Primjer iznad nema `IN` ali ima `OUT` parametar.

- `OUT` parametri vraćaju vrijednosti koje možete pohraniti u sesijske varijable.
- Sesijske varijable (npr. `@brojKorisnika`) ne zahtijevaju eksplicitnu deklaraciju pomoću `DECLARE`. One žive samo unutar sesije a onda se unište.

```
DELIMITER $$  
CREATE PROCEDURE DohvatiBrojKorisnika(OUT broj INT)  
BEGIN  
    SELECT COUNT(*) INTO broj FROM korisnici;  
END $$  
DELIMITER ;
```

Broj izlaznih parametara nije ograničen. `OUT` govori o izlaznom parametru i on se upisuje tamo. `OUT` parametre možete pohraniti u sesijske varijable.

Kod funkcije ne moramo definirati ulazne parametre. To kod funkcija moramo definirati. Funkcija može vratiti samo jedan parametar dok procedura može vratiti više parametara. Kada koristiti funkciju a kada procedure - ovisi o tome koliko rezultata vraćamo. Transakcije služe ako nešto krene po zlu da se

napravi roll back. Procedure mogu unutar sebe imati transakcije a funkcije ne mogu. Funkcije možete dodijeliti SQL upitima a procedure ne možete. Dakle proceduru nije moguće pozvati unutar upita.

Kod jednostavnih upita treba koristiti funkcije. Kod kompleksnijih procesa treba koristiti procedure.

```
DELIMITER $$

CREATE PROCEDURE CalculateSquare(
    IN input_number INT,      -- IN parametar za ulazni broj
    OUT output_square INT    -- OUT parametar za rezultat
)
BEGIN
    -- Izračunavanje kvadrata ulaznog broja i spremanje u OUT parametar
    SET output_square = input_number * input_number;
END $$

DELIMITER;
```

`DELIMITER $$` i `DELIMITER ;` mijenja završni znak naredbe (iz `;` u `$$`) kako bi se omogućilo kreiranje procedure koja može sadržavati više SQL naredbi. `IN input_number` parametar prima ulaznu vrijednost iz poziva procedure. `OUT output_square` parametar vraća izlaznu vrijednost kao rezultat procedure. `SET output_square = input_number * input_number;` Izračunava kvadrat ulaznog broja i dodjeljuje ga `OUT` parametru.

```
SET @result = 0; -- Inicijalizacija varijable za primanje rezultata
CALL CalculateSquare(5, @result); -- Pozivanje procedure
SELECT @result AS square_result; -- Provjera rezultata
```

Varijable u procedurama

Varijable se koriste za pohranu i rukovanje podacima unutar pohranjene procedure. Po potrebi se mogu deklarirati i dodijeliti im vrijednosti.

U SQL-u postoje dvije vrste varijabli. Sada ćemo pogledati svaku od njih.

Varijable sesije

Varijable sesije u MySQL-u imaju ispred simbol `@` (na primjer `@varijaba_ime`). Ove varijable pridružene su trenutnoj sesiji ili vezi i zadržavaju svoje vrijednosti tokom cijele sesije dok se eksplisitno ne promijene ili dok sesija ne završi.

```
CREATE PROCEDURE `GenerateSalesReport`(
    IN start_date DATE,
    IN end_date DATE
)
BEGIN
    SELECT @totalSales := 0;
    SELECT SUM(sales_amount) INTO @totalSales FROM sales;
```

```
SELECT @totalSales As total_sales;
END
```

Uobičajne varijable

Uobičajne varijable, također poznate kao lokalne varijable, deklariraju se pomoću **DECLARE** ključne riječi unutar opsega pohranjene procedure. Za razliku od varijabli sesije, redovne varijable nemaju prefiks **@** (na primjer **varijabla_ime**). Oni su privremene i postoje samo unutar bloka koda u kojem su deklarirane.

```
CREATE PROCEDURE `GenerateSalesReport`(
    IN start_date DATE,
    IN end_date DATE
)
BEGIN
    DECLARE totalSales INT;
    SELECT SUM(sales_amount) INTO totalSales FROM sales;
END
```

MySQL funkcije

Funkcija je slična pohranjenoj proceduri, ali vraća jednu vrijednost, a ne više skupova rezultata. Funkcije se mogu koristiti u SQL naredbama na isti način kao izrazi, a mogu se koristiti i za obavljanje zadataka kao što su provjera validacije podataka i manipulacija.

Pohranjena funkcija je slična proceduri u MySQL-u , ali ima neke razlike koje su sljedeće:

- Funkcijski parametar može sadržavati samo **IN** parametar , ali ne može dopustiti navođenje ovog parametra, dok procedura može dopustiti **IN**, **OUT**, **INOUT** parametre.
- Pohranjena funkcija može vratiti samo jednu vrijednost definiranu u zaglavlju funkcije.
- Pohranjena funkcija također se može pozvati unutar SQL naredbi.
- Možda neće dati set rezultata.

Sintaksa:

```
DELIMITER $$

CREATE FUNCTION ime_funkcije (funkcija_parametar(ri))
RETURNS tip_podataka
[NOT] {svojstva}
tijelo_funkcije;
```

Naziv parametra	Opis
ime_funkcije	To je naziv pohranjene funkcije koju želimo stvoriti u bazi podataka. Ne bi trebao biti isti kao naziv ugrađene funkcije MySQL-a.
funkcija_parametar	Sadrži popis parametara koje koristi tijelo funkcije. Ne dopušta određivanje parametara IN , OUT , INOUT .
tip_podataka	To je tip podataka povratne vrijednosti funkcije. Trebao bi biti bilo koji važeći MySQL tip podataka.

svojstva	Izraz <code>CREATE FUNCTION</code> prihvacen je samo kada su svojstva (<code>DETERMINISTIC</code> , <code>NO SQL</code> ili <code>READS SQL DATA</code>) definirane u deklaraciji.
tijelo_funkcije	Ovaj parametar ima skup SQL izraza za izvođenje operacija. Zahtijeva najmanje jednu naredbu RETURN. Kada se izvrši naredba return, funkcija će se automatski prekinuti. Tijelo funkcije dano je ispod: BEGIN -- SQL naredbe END \$\$ DELIMITER

Pogledajmo malo svojstva da bi blo jasnije. DETERMINISTIC ili NONDETERMINISTIC su objašnjeni iznad.

`READS SQL DATA` Ovo eksplisitno govori MySQL-u da će funkcija SAMO čitati podatke iz baza podataka, stoga ne sadrži upute koje mijenjaju podatke, ali sadrži SQL upute koje čitaju podatke (npr. `SELECT`).

`MODIFIES SQL DATA` Ovo označava da rutina sadrži izjave koje mogu pisati podatke (na primjer, sadrže instrukcije `UPDATE`, `INSERT`, `DELETE` ili `ALTER`).

`NO SQL` Ovo označava da rutina ne sadrži SQL izraze.

`CONTAINS SQL` Ovo označava da rutina sadrži SQL upute, ali ne sadrži izraze koji čitaju ili pišu podatke. Ovo je zadana vrijednost ako nijedna od ovih karakteristika nije navedena eksplisitno. Primjeri takvih naredbi su `SELECT NOW()`, `SELECT 10+@b`, `SET @x = 1` ili `DO RELEASE_LOCK('abc')`, koje se izvršavaju, ali niti čitaju niti zapisuju podatke.

Po defaultu, da bi `CREATE FUNCTION` izraz bio prihvacen, najmanje jedan od `DETERMINISTIC`, `NO SQL` ili `READS SQL DATA` mora biti eksplisitno specificiran. U suprotnom dolazi do greške.

Da biste riješili ovaj problem, trebamo dodati sljedeće redove naredbi nakon `RETURNS` i prije `BEGIN`:

```
READS SQL DATA
DETERMINISTIC
```

Predavač je stavljao samo `DETERMINISTIC`.

Primjer:

```
DELIMITER $$

CREATE FUNCTION DohvatiUkupnuProdaju(kupac_id INT)
RETURNS INT
BEGIN
    DECLARE ukupno INT DEFAULT 0;
    SELECT SUM(iznos) INTO ukupno FROM narudzbe WHERE kupac_id = id;
    RETURN ukupno;
END $$

DELIMITER ;
```

Pozivanje funkcije pohranjene u MySQL-u, možete koristiti ključnu riječ `SELECT` i ime funkcije. Npr.:

```
SELECT DohvatiUkupnuProdaju(1234);
```

Ovo bi pozvalo DohvatiUkupnuProdaju funkciju s ulaznim parametrom `1234`, a povratna vrijednost bi se koristila u naredbi `SELECT`.

Razlike između procedura i funkcija

Najopćenitija razlika između procedura i funkcija je da se pozivaju na različite načine i za različite svrhe:

- Procedura ne vraća vrijednost. Umjesto toga, poziva se naredbom CALL za izvođenje operacije kao što je modificiranje tablice ili obrada dohvaćenih zapisa.
- Funkcija se poziva unutar izraza i vraća jednu vrijednost izravno pozivatelju koja se koristi u izrazu.
- Ne možete pozvati funkciju s naredbom CALL, niti možete pozvati proceduru u izrazu.

Sintaksa za stvaranje rutine donekle se razlikuje za postupke i funkcije:

- Parametri procedure mogu se definirati kao samo ulazni, samo izlazni ili oboje. To znači da procedura može proslijediti vrijednosti natrag pozivatelju pomoću izlaznih parametara. Ovim se vrijednostima može pristupiti u naredbama koje slijede naredbu CALL. Funkcije imaju samo ulazne parametre. Kao rezultat toga, iako i procedure i funkcije mogu imati parametre, deklaracija parametara procedure razlikuje se od one za funkcije.
- Funkcije vraćaju vrijednost, tako da u definiciji funkcije mora postojati klauzula `RETURNS` koja označava tip podataka povratne vrijednosti. Također, mora postojati barem jedna izjava RETURN unutar tijela funkcije za vraćanje vrijednosti pozivatelju. `RETURNS` i `RETURN` se ne pojavljuju u definicijama procedura.
 - Za pozivanje pohranjene procedure koristite CALL statement. Da biste pozvali pohranjenu funkciju, pozovite se na nju u izrazu. Funkcija vraća vrijednost tokom procjene izraza.
 - Procedura se poziva korištenjem naredbe CALL i može vratiti samo vrijednosti pomoću izlaznih varijabli. Funkcija se može pozvati iz naredbe baš kao i svaka druga funkcija (to jest, pozivanjem imena funkcije) i može vratiti skalarnu vrijednost.
 - Određivanje parametra kao `IN`, `OUT` ili `INOUT` vrijedi samo za PROCEDURE. Za FUNKCIJU, parametri se uvijek smatraju `IN` parametrima.

Ako prije naziva parametra nije dana ključna riječ, to je `IN` parametar prema zadanim postavkama. Parametrima za pohranjene funkcije ne prethode `IN`, `OUT` ili `INOUT`. Svi parametri funkcije tretiraju se kao `IN` parametri.

Pohranjene rutine (odnose se i na pohranjene procedure i na pohranjene funkcije) **pridružene su određenoj bazi podataka, baš kao tablice ili pogledi**. Kada odbacite (engl. drop) bazu podataka, sve pohranjene rutine u bazi podataka također se odbacuju.

Pohranjene procedure i funkcije ne dijele isti prostor imena. Moguće je imati proceduru i funkciju s istim imenom u bazi podataka.

U pohranjenim procedurama može se koristiti dinamički SQL, ali ne u funkcijama ili okidačima.

SQL pripremljeni iskazi (`PREPARE`, `EXECUTE`, `DEALLOCATE PREPARE`) mogu se koristiti u pohranjenim procedurama, ali ne i pohranjenim funkcijama ili okidačima. Stoga pohranjene funkcije i okidači ne mogu koristiti dinamički SQL (gdje konstruirate izjave kao nizove i zatim ih izvršavate)

Još neke zanimljive razlike između FUNKCIJE i POHRANJENE PROCEDURE:

- Pohranjena procedura je unaprijed kompajlirani plan izvršenja, dok funkcije nisu. Funkcija analizirana i kompajlirana tokom izvođenja. Pohranjene procedure, Pohranjene kao pseudokod u bazi podataka, tj. kompajlirani oblik
- Pohranjena procedura ima sigurnost i smanjuje mrežni promet, a također možemo nazvati pohranjenu proceduru u bilo kojem br. aplikacija odjednom
- Funkcije se obično koriste za izračune dok se procedure obično koriste za izvršavanje poslovne logike
- Funkcije Ne mogu utjecati na stanje baze podataka (Izjave koje eksplisitno ili implicitno predaju ili se vraćaju unatrag nisu dopuštene u funkciji) Dok pohranjene procedure mogu utjecati na stanje baze podataka korištenjem commit itd.
- Funkcije ne mogu koristiti FLUSH izjave dok pohranjene procedure to mogu
- Pohranjene funkcije ne mogu biti rekurzivne, dok pohranjene procedure mogu biti. Napomena: Rekursivne pohranjene procedure onemogućene su prema zadanim postavkama, ali se mogu omogućiti na poslužitelju postavljanjem sistemske varijable poslužitelja `max_sp_recursion_depth` na vrijednost različitu od nule
- Unutar pohranjene funkcije ili okidača, nije dopušteno mijenjati tablicu koja se već koristi (za čitanje ili pisanje) naredbom koja je pozvala funkciju ili okidač

<https://stackoverflow.com/questions/3744209/mysql-procedure-vs-function-which-would-i-use-when>

Napredni MySQL

Uvod u MySQL Poglede

Pogled (engl. View) je objekt baze podataka koji predstavlja podatke koji postoje u jednoj ili više tablica. Prikazi se koriste na sličan način kao tablice, ali ne sadrže nikakve podatke. Oni samo "pokazuju" na podatke koji postoje negdje drugdje (tablice ili prikazi, na primjer)

- **Definicija:** Pogledi su virtualne tablice koje se kreiraju na temelju SQL upita.
- **Upotreba:** Koriste se za pojednostavljenje složenih upita, sigurnost podataka i poboljšanje performansi.
- **Prednosti:** Sakrivanje složenosti upita, ograničavanje pristupa podacima, olakšavanje upravljanja bazom podataka.

Kreiranje pogleda

Sintaksa:

```
CREATE VIEW ime_pogleda AS  
SELECT stupac1, stupac2  
FROM tablica  
WHERE uslov;
```

Primjer:

```
CREATE VIEW radnici_pregled AS  
SELECT ime, prezime, odjel  
FROM radnici  
WHERE aktivran =1;
```

Korištenje pogleda

Upotreba u upitima:

```
SELECT * FROM radnici_pregled;
```

Prednosti:

- jednostavnost korištenja kao obične tablice, ali bez potrebe za ponovnim pisanjem složenih upita

Ažuriranje i brisanje pogleda

Ažuriranje pogleda

```
CREATE OR REPLACE VIEW radnici_pregled AS  
SELECT ime, prezime, odjel, email
```

```
FROM radnici
WHERE aktivan = 1;
```

Brisanje pogleda

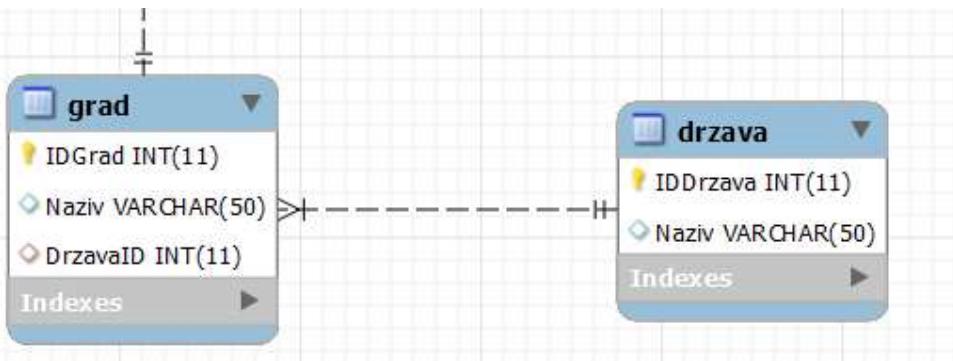
```
DROP VIEW radnici_pregled;
```

Ograničenja i najbolje prakse

- **Ograničenja:** Ne može sadržavati naredbe kao što su `ORDER BY` (kako želimo složiti rezultate koje prikazujemo), određene vrste podupita, itd.
- **Najbolje prakse:** Koristiti pogled za stabilne upite, izbjegavati previše složene upite unutar pogleda.
- **Sigurnosne mjere:** Pogledi mogu pomoći u ograničavanju pristupa osjetljivim podacima tako što će prikazati samo potrebne informacije. Kroz pogled je moguće napraviti update stvarnog podatka i zato treba biti pažljiv.

Primjer pogleda

Evo primjera:



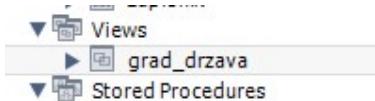
Prikazat ćemo sve gradove i u kojoj su državi. Ne trebaju nam gradovi koji nisu vezani za državu. Napravit ćemo pogled (engl. view) s `create or replace view`. Napravit ćemo standardni `select` ali sa `order` gdje ćemo složiti po državi.

```
use adventureworkshop;

create or replace view Grad_Drzava as
select gr.Naziv as Grad, dr.Naziv as Drzava
from Grad as gr
left join Drzava as dr
on gr.DrzavaId = dr.IDDrzava
```

```
order by Drzava;
```

Na Views pojavit će se:



view pokrećemo sa:

```
select * from grad_drzava order by Grad;
```

Dakle ne moramo svaki put pokrenuti select. Kao što smo već rekli, podupiti i povezivanja sa join ovdje ne dolaze u obzir.

Probat ćemo ažurirati stanje u tablici:

```
update grad_drzava set Drzava = "Hrvatska"
where Grad like "%Donja Motičina%";
```

Grad	Drzava
Berli	Njemačka
Bjelovar	Hrvatska
Budimpešta	Madarska
Derventa	Bosna i Hercegovina
Donja Motičina	NULL
Dresde	Njemačka
Eger	Madarska
Frankfurt	Njemačka
Kakanj	Bosna i Hercegovina
Koprivnica	Hrvatska
Mostar	Bosna i Hercegovina
Osijek	Hrvatska
Pecuh	Madarska
Pula	Hrvatska
Rijeka	Hrvatska
Sarajevo	Bosna i Hercegovina
Split	Hrvatska
Vinkovci	Hrvatska
Zagreb	Hrvatska
Zenica	Bosna i Hercegovina

Ovo ne prolazi (MySQL Workbench dojavi grešku) jer po defaultu nije moguće napraviti ažuriranje.



Uvod u MySQL Transakcije

- **Definicija:** Transakcija je skup operacija koje se trebaju izvesti kao jedna cjelina; ako jedna operacija ne uspije, sve ostale se poništavaju. Ove operacije mogu biti operacije kreiranja, čitanja, ažuriranja ili brisanja.

Tokom procesa transakcije, baza podataka je u nekonzistentnom stanju jer su u toku operacije koje unose promjene u bazu podataka. DB se vraća u konzistentnije stanje kada su operacije izvršene. Da bi transakcija bila uspješna, to znači da je svaka izvršena operacija izvršena.

Transakcije baze podataka vrlo su važne u osiguravanju dosljednosti vaše baze podataka kada se više operacija izvodi u isto vrijeme. Također vam daje način da vratite promjene koje su se možda dogodile zbog neuspjeha ili slučajne zlouporabe operacije.

- **ACID svojstva:**
 - **Atomarnost (engl. Atomicity):** Transakcija se izvodi u cijelosti ili se uopće ne izvodi.
 - **Dosljednost (engl. Consistency):** Transakcija prenosi bazu iz jednog valjanog stanja u drugo. Dakle podaci u određenom stupcu uvijek moraju biti istog topa.
 - **Izoliranost (engl. Isolation):** Rezultati transakcije su izolirani od drugih transakcija do njihovog završetka.

- **Trajnost (engl. Durability):** Jednom kada je transakcija završena, promjene su trajne. Koristi se zapisivanje unaprijed i zapisivanje dnevnika transakcija na disk prije bilo kakvih promjena u bazi podataka.

Početak transakcije

- **Pokretanje početka transakcije:**

```
START TRANSACTION;
```

- **Primjer upotrebe:**

- Kreiranje korisničkog računa i postavljanje početnog depozita.

```
INSERT INTO users (username, email) VALUES ('pero', 'pero@example.com');  
INSERT INTO accounts (user_id, balance) VALUES (LAST_INSERT_ID(), 100);
```

Kontrola transakcija

Commit:

Potvrda svih izmjena učinjenih u transakciji.

```
COMMIT;
```

Ovim upitom omogućuje da promjene napravljene u bazi podataka postanu trajne.

Rollback:

Poništavanje svih izmjena ako dođe do greške.

```
ROLLBACK;
```

Ovaj upit vam omogućava da poništite promjene koje ste napravili u bazi podataka, čime se baza podataka vraća u njeno prethodno (posljednje izdavanje) stanje.

Primjer:

Ako drugi `INSERT` ne uspije, koristite `ROLLBACK`; za poništavanje prvog `INSERT`.

Nivo izolacije

Nivoi izolacije:

- **READ UNCOMMITTED**

- Dopušta čitanje nespremljenih podataka, što može dovesti do "dirty reads," gdje transakcija može čitati podatke koje je druga nepotvrđena transakcija izmijenila. To znači da druge transakcije mogu promijeniti podatke koje druga transakcija trenutno čita, ali te promjene možda neće biti vidljive dok se operacija ne dovrši.

- **READ COMMITTED**
 - o Sprječava "dirty reads" dopuštajući čitanje samo podataka koje su druge transakcije već potvrdile, ali i dalje može doći do "non-repeatable reads," gdje se isti podaci čitaju dva puta unutar jedne transakcije i mogu se razlikovati. To znači da druge transakcije mogu promijeniti podatke koje transakcija trenutno čita, ali te promjene neće biti vidljive dok se druga transakcija ne izvrši.
- **REPEATABLE READ (zadano u MySQL)**
 - o Osigurava da ako se podaci jednom pročitaju unutar transakcije, svako daljnje čitanje istih podataka unutar iste transakcije vrati isti rezultat, sprječavajući "non-repeatable reads" ali ne sprječava "phantom reads," gdje se nova zapisi koji odgovaraju upitima transakcije mogu dodati od strane drugih transakcija. To znači da iako su druge transakcije izvršile promjene, ako transakcija ponovno izvrši naredbu SELECT, uvjek će vidjeti iste podatke.
- **SERIALIZABLE**
 - o Najstroža razina izolacije koja sprječava "phantom reads" i osigurava potpunu izolaciju transakcije tako da se izvršava kao da je jedina transakcija u sistemu, što može rezultirati značajnim padom performansi zbog zaključavanja resursa. Također sprječava druge transakcije da mijenjaju podatke koje transakcija čita i da dodaju nove redove koji bi bili vidljivi trenutnoj transakciji.

Primjena:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

SET upit (engl. query) omogućava da postavite vaš željeni nivo izolacije tj. da ga promjenite.

- **Utjecaj na performanse i dosljednost:**
 - o Više razine izolacije smanjuju mogućnost grešaka ali mogu utjecati na performanse.

Praktičan primjer transakcije

- **Scenarij:** Transfer novca između dva računa.

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE user_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE user_id = 2;
COMMIT;
```

U ovom primjeru napravili smo ažuriranje sa **UPDATE**. Mogli smo napraviti i/ili umetanje s **INSERT INTO**, unutar transakcije.

Sljedeći korak koji smo napravili je **COMMIT** s kojim smo potvrdili promjene kako bi smo bili sigurni da su trajne.

- **Sigurnost:** Ako bilo koji **UPDATE** nađe na problem, izvedite **ROLLBACK**; kako bi ste osigurali dosljednost podataka.

InnoDB u MySQL-u omogućava neka dodatna svojstva kao što su zaključavanje na nivou reda, ograničavanje (engl. constraints) stranog ključa i oporavak o pada (engl. crash recovery) što ga čini robusnijim i pouzdanijim.

Uvod u MySQL Okidače

MySQL okidači (engl. triggers) su slični JavaScript slušateljima događaja (engl. event listeners). Ne izvršavaju se sve dok se ne dogodi radnja koja za koju im je rečeno da je čekaju.

Okidač je imenovani objekt baze podataka koji je povezan s tablicom i koji se aktivira kada se dogodi određeni događaj na tablici.

Neke upotrebe okidača su izvođenje provjera vrijednosti koje se umeću u tablicu ili izvođenje izračuna na vrijednostima angažiranim u ažuriranje.

Okidači su korisni za automatizaciju zadataka koji se ponavljaju. Možete samo postaviti okidač za izračun nakon svake specifične akcije baze podataka. Također možete postaviti okidač za izvođenje zadataka provjere valjanosti podataka na tablici.

Okidač se pokreće kada se dogodi `INSERT`, `UPDATE` ili `DELETE` operacija na tablici baze podataka. Okidač se aktivira po redu, tako da ako se umeće ili briše više redova podataka, svaki od njih i dalje pokreće radnju koju je postavljena okidačem. Okidač se može postaviti na aktivaciju prije ili poslije akcije.

MySQL okidač je pohranjeni program (s upitima) koji se automatski izvršava kako bi odgovorio na određeni događaj kao što je umetanje, ažuriranje ili brisanje u tablici.

Ovdje ćemo govoriti kako kreirati okidače, kako ih odbaciti nakon korištenja.

Što su Okidači?

- Specijalizirane skripte unutar baze podataka
- Automatski se izvršavaju kao odgovor na određene događaje (npr. `INSERT`, `UPDATE`, `DELETE`)

Zašto koristiti Okidače?

- Automatizacija operacija nad podacima
- Održavanje integriteta podataka
- Evidencija promjena u bazi podataka

Vrste Okidača

Prema vremenu izvršenja:

- `BEFORE` (prije izvršenja akcije)
- `AFTER` (nakon izvršenja akcije)

Prema vrsti događaja:

- **INSERT**
- **UPDATE**
- **DELETE**

Kako kreirati okidač

Za kreiranje novog okidača upotrijebite naredbu **CREATE TRIGGER**. Ova naredba ima sljedeću strukturu:

```
CREATE TRIGGER okidač_ime
okidač_vrijeme
okidač_događaj
ON tablica_ime
FOR EACH ROW
okidač_tijelo
```

Evo značenja redova:

- Ključna riječ **CREATE TRIGGER** je obavezna i slijedi je naziv okidača. Koristit ćete ovo ime za pozivanje na okidač u budućnosti i za brisanje okidača ako se ikada ukaže potreba. Ovo ime treba biti jedinstveno po bazi podataka.
- **okidač_vrijeme** je promjenjiva vrijednost koja može biti samo ili **BEFORE** ili **AFTER**. Ovo određuje hoće li se okidač aktivirati prije ili nakon što se događaj dogodio.
- Druga **okidač_događaj** je varijabla koja ima ograničen broj mogućih opcija. Ova varijabla ne može biti niti jedna vrijednost osim **INSERT**, **UPDATE** ili **DELETE**. Određuje koji događaj treba slušati.
- **tablica_ime** je naziv tablice koju okidač treba promatrati. Ovo mora biti naziv postojeće tablice u vašoj bazi podataka, ali može biti i prazna tablica.
- To **FOR EACH ROW** je drugi obvezni dio definicije okidača.
- **okidač_tijelo** je SQL upit koji želite da se pokrene kada se pokrene ovaj okidač.

Za primjer kreiranja okidača, izradit ću jednostavnu tablicu **korisnici** za vježbu.

```
CREATE TABLE
korisnici (
    ime_prezime VARCHAR(120),
    email VARCHAR(120),
    korisnik VARCHAR(30),
    šifra VARCHAR(60)
);
```

Sada možemo kreirati jednostavan okidač i priložiti ga ovoj praznoj tablici. Okidač koji šifrira string šifre prije nego što se umetnu pomoću funkcije **MD5** imao bi smisla.

```
CREATE TRIGGER šifra_hasher
```

```
BEFORE
INSERT
ON korisnici
FOR EACH ROW
SET
    NEW.šifra = MD5 (NEW.šifra);
```

Ovaj primjer je prilično jednostavan i sam po sebi razumljiv. Ali postoji **NEW** ključna riječ. Ova ključna riječ vam daje pristup novim podacima koji se kreiraju i omogućavaju vam upotrebu ili izmjenu vrijednosti kako želite.

Ove vrijednosti možete mijenjati samo ako je postavljen **događaj_vrijeme** kao **BEFORE**. Ako je **događaj_vrijeme** postavljeno na **AFTER**, podaci su već pohranjeni prije nego što su došli do okidača tako da se ne mogu ponovno mijenjati.

Možete koristiti **NEW** ključnu riječ u **INSERT** i **UPDATE** događajima, ali ne i **DELETE** događaju.

Tu je i **OLD** ključna riječ koju možete koristiti u **DELETE** okidačima **UPDATE** događaja koji vam daju pristup prethodnim vrijednostima zapisa (sloga) na kojeg se odnosi. Ne možete upotrijebiti ovu ključnu riječ na **INSERT** događaju jer ne postoji prethodni zapis prije nego što se kreiraju novi podaci.

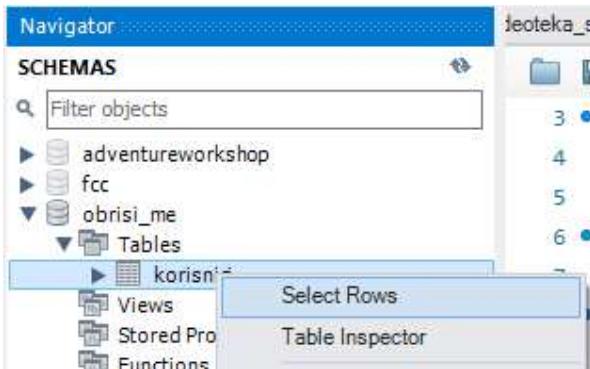
Da biste testirali ovaj okidač, umetnite red u tablicu **korisnici**:

```
INSERT INTO
    korisnici
VALUES
(
    'Branko Ćelap',
    'branko@test.com',
    'zubby1',
    'password'
);
```

Provjerite svoju tablicu za vrijednosti. Trebali biste imati nešto ovako:

ime_prezime	email	korisnik	šifra
Branko Ćelap	branko@test.com	branko1	f2bae9c62dcc5a623503ddc64acaa5c2

Iz MySQL WorkBench-a to ćemo dobiti sa **Select Rows**. Vrijednost lozinke je ispravno šifrirana.



Naravno mogli smo pogledati i sa `SELECT * FROM ime_baze.korisnici;`

Postoji 6 različitih vrsta okidača u MySQL-u: okidač prije ažuriranja (BEFORE UPDATE trigger), okidač nakon ažuriranja (AFTER UPDATE trigger), okidač prije umetanja (BEFORE INSERT trigger), okidač poslije umetanja (AFTER INSERT trigger), okidač prije brisanja (BEFORE DELETE trigger), okidač nakon brisanja (AFTER DELETE trigger).

Okidač prije ažuriranja (BEFORE UPDATE trigger),

Kao što naziv implicira, to je okidač koji se aktivira prije pokretanja ažuriranja. Ako napišemo izjavu o ažuriranju, tada će se radnje okidača izvršiti prije implementacije ažuriranja.

Kreirajmo tablice:

```
create table kupac (račun_br integer primary key,
                    kupac_prezime varchar(20),
                    raspoloživ_saldo decimal
);

create table mini_obračun (
    račun_br integer,
    raspoloživ_saldo decimal,
    foreign key(račun_br) references kupac(račun_br) on delete cascade
);
```

Ubacimo vrijednosti u njih:

```
insert into kupac values (1000, "Branko", 7000);
insert into kupac values (1001, "Sanja", 12000);
```

Sada ćemo napraviti okidač za umetanje starih vrijednosti u tablicu min_obračun (koji uključuje broj računa i prethodno dostupno stanje kao parametre) prije ažuriranja bilo kojeg sloga u tablici/slogu kupaca.

```
delimiter $$  
create trigger ažuriraj_kupca  
    before update on kupac  
    for each row  
begin  
    insert into mini_obračun values (old.račun_br, old.raspoloživ saldo);  
end $$  
delimiter ;
```

Za probu ćemo ažurirati polja u kupcu fs vidimo što se desilo:

```
update kupac set raspoloživ saldo = raspoloživ saldo + 3000 where račun_br = 1001;  
update kupac set raspoloživ saldo = raspoloživ saldo + 3000 where račun_br = 1000;
```

U kupcu su se stanja povećala za 3000, što smo i očekivali.

račun_br	kupac_prezime	raspoloživ saldo
1000	Branko	10000
1001	Sanja	15000
NULL	NULL	NULL

U tablici mini_obračun zabilježena su prethodna stanja:

račun_br	raspoloživ saldo
1001	12000
1000	7000

Okidač nakon ažuriranja (AFTER UPDATE trigger)

Kao što naziv implicira, ovaj se okidač poziva nakon ažuriranja. (tj. implementira se nakon što se izvrši naredba ažuriranja.).

Kreirajmo još jednu tablicu za zapise nakon ažuriranja:

```
create table mikro_obračun (  
    račun_br integer,  
    raspoloživ saldo decimal,  
foreign key(račun_br) references kupac(račun_br) on delete cascade  
);
```

Ubacimo još jednu vrijednost u tablicu kupca:

```
insert into kupac values (1002, "Mladen",  
4500);
```

račun_br	kupac_prezime	raspoloživ saldo
1000	Branko	10000
1001	Sanja	15000
1002	Mladen	4500
NULL	NULL	NULL

Sada napravimo okidač za umetanje novih vrijednosti brojeva računa i raspoloživog stanja nakon zapisivanja u tablicu mikro_obračun:

```
delimiter $$  
create trigger nakon_ažuriranja  
    after update on kupac  
    for each row  
begin  
    insert into mikro_obračun values(new.račun_br, new.raspoloživ saldo);  
end $$  
delimiter ;
```

Pogledajmo kako radi okidač nakon ažuriranja:

```
update kupac set raspoloživ saldo = raspoloživ saldo + 1500 where račun_br = 1002;
```

U kupcima je promijenjeno stanje:

račun_br	kupac_prezime	raspoloživ saldo
1000	Branko	10000
1001	Sanja	15000
1002	Mladen	6000
NULL	NULL	NULL

račun_br	raspoloživ saldo
1002	6000

A u tablici `mikro_obračun` je novi zapis:

Okidač prije umetanja (BEFORE INSERT trigger)

Kao što naziv implicira, ovaj okidač se poziva prije umetanja ili prije nego što se izvrši naredba umetanja.

```
create table  
    kontakti (  
        kontakt_id INT (11) NOT NULL AUTO_INCREMENT,  
        prezime VARCHAR (30) NOT NULL,  
        ime VARCHAR (25),  
        rođendan DATE,  
        datum_kreiranja DATE,  
        kreirao VARCHAR(30),  
        CONSTRAINT kontakti_pk PRIMARY KEY (kontakt_id)  
    );
```

Okidač za umetanje podataka o kontaktu kao što su ime, rođendan i datum kreiranja/korisnik u tablici kontakt prije nego što dođe do umetanja:

```
delimiter $$  
create trigger kontakti_prije_inserta  
    before insert  
    on kontakti for each row  
begin  
    DECLARE vUser varchar(50);  
    -- Pronađi korisničko ime osobe koja radi INSERT u tablicu  
    select USER() into vUser;  
    -- Ažuriraj polje datum_kreiranja na trenutni sistemski datum  
    SET NEW.datum_kreiranja = SYSDATE();  
    -- Ažuriraj polje kreirao sa korisničkim imenom osobe koja izvodi INSERT  
    SET NEW.kreirao = vUser;  
end $$  
delimiter ;
```

Ubacimo podatke u [kontakti](#):

```
insert into kontakti values (  
    1, "Nikola", "Tesla",  
    str_to_date ("10-07-1856", "%d-%m-%Y"),  
    str_to_date ("10-06-2024", "%d-%m-%Y"), "xyz");
```

Ono što je zapisano je:

kontakt_id	prezime	ime	rođendan	datum_kreiranja	kreirao
1	Nikola	Tesla	1856-07-10	2024-06-09	root@localhost

Dakle bez obzira što mi upisali tko je kreirao okidač sam uzme polja [datum_kreiranja](#) i [kreirao](#), nije ih moguće krivo upisati.

Okidač nakon umetanja (BEFORE INSERT trigger)

Kao što naziv implicira, ovaj okidač se poziva nakon implementacije umetanja.

Kreirajmo tablice:

```
create table kontakti2  
    (kontakt_id int (11) NOT NULL AUTO_INCREMENT,
```

```

        prezime VARCHAR(30) NOT NULL,
        ime VARCHAR(25), birthday DATE,
        CONSTRAINT kontakti_pk PRIMARY KEY (kontakt_id)
);

create table kontakti_kontrola (
    kontakt_id integer,
    datum_kreiranja date,
    kreirao varchar (30)
);

```

Tablicu smo nazvali `kontakt2` jer kontakt imamo u prethodnim primjerima.

Napravimo okidač za umetanje `kontakt_id`-a i datumu kreiranja i korisniku koji je kreirao slog u `kontakt_kontrola` nakon umetanja:

```

delimiter $$

create trigger kontakti_nakon_inserta
after insert
on kontakti2 for each row
begin
    DECLARE vUser varchar(50);
    -- Pronađi korisničko ime osobe koja pravi INSERT u tablicu
    SELECT USER() into vUser;
    -- Ubaci slog u kontrolnu tablicu
    INSERT into kontakti_kontrola (
        kontakt_id,
        datum_kreiranja,
        kreirao)
    VALUES
        (NEW.kontakt_id, SYSDATE(), vUser);
end $$

delimiter ;

```

Sada umetnimo nešto u tablicu `kontakt2` i time okinimo trigger `kontakti_nakon_inserta`:

```

insert into kontakti2 values (1, "Sanja", "Ulipi",
                               str_to_date("09.06.2024", "%d.%m.%Y"));

```

kontakt_id	prezime	ime	birthday
1	Sanja	Ulipi	2024-06-09
NULL	NULL	NULL	NULL

U `kontakti_kontrola` bit će zapisano:

kontakt_id	datum_kreiranja	kreirao
1	2024-06-09	root@localhost

Okidač prije brisanja (BEFORE DELETE trigger)

Kao što naziv implicira, ovaj okidač se poziva prije nego što dođe do brisanja ili prije nego što se implementira naredba brisanja.

Iskoristit ćemo tablicu **kontakti** koju već imamo iz prošlih primjera.

Kreirat ćemo kontakti_kontrola2:

```
create table kontakti_kontrola2 (
    kontakt_id integer,
    deleted_date date,
    deleted_by varchar(20)
);
```

Okidač za umetanje **contact_id** i datuma brisanja kontakta, podatka tko je obrisao **kontakti_kontrola2** zapis prije nego što dođe do brisanja:

```
delimiter $$
create trigger kontakti_prije_delete
before delete
on kontakti for each row
begin
    DECLARE vUser varchar(50);

    -- Pronađite korisničko ime osobe koja radi DELETE u tablici
    SELECT USER() into vUser;

    -- Insert record into audit table
    INSERT into kontakti_kontrola2 (
        kontakt_id,
        deleted_date,
        deleted_by)
    VALUES (
        OLD.kontakt_id,
        SYSDATE(),
        vUser
    );
end $$
delimiter ;
```

Umetnimo neki podatak a zatim ga obrišimo:

```
insert into kontakti values (3, "Tolstoj", "Lav",
    str_to_date ("07.11.1910", "%d.%m.%Y"),
    str_to_date ("09.06.2024", "%d.%m.%Y"), "xyz");
```

kontakt_id	prezime	ime	rođendan	datum_kreiranja	kreirao
1	Nikola	Tesla	1856-07-10	2024-06-09	root@localhost
2	Tolstoj	Lav	1910-11-07	2024-06-09	root@localhost
NULL	NULL	NULL	NULL	NULL	NULL

```
delete from kontakti where prezime="Tolstoj";
```

U kontakti je ovo stanje:

kontakt_id	prezime	ime	rođendan	datum_kreiranja	kreirao
1	Nikola	Tesla	1856-07-10	2024-06-09	root@localhost
2	Tolstoj	Lav	1910-11-07	2024-06-09	root@localhost
NULL	NULL	NULL	NULL	NULL	NULL

U kontakt_kontrola2 je zapisano što je obrisano:

kontakt_id	datum_brisanja	obrisao
2	2024-06-09	root@localhost

Okidač nakon brisanja (BEFORE DELETE trigger)

Kao što naziv implicira, ovaj okidač se poziva nakon što se dogodi brisanje ili nakon što je operacija brisanja implementirana.

Tablicu kontakti, ćemo koristiti od prije.

Dodat ćemo tablicu kontakti_kontrola3.

```
create table kontakti_kontrola3 (
    kontakt_id integer,
    datum_brisanja date,
    obrisao varchar(20)
);
```

Okidač za umetanje zapisuje kontakt_id i datuma brisanja, te podatak tko je obrisao slog u kontakti_kontrola3 nakon brisanja:

```
delimiter $$

create trigger kontakti_nakon_delete
after delete
on kontakti for each row
begin
    DECLARE vUser varchar(50);
```

```
-- Pronadi korisničko ime osobe koja radi DELETE u tablici
SELECT USER() into vUser;
-- Ubaci slog u kontrolnu tablicu
INSERT into kontakti_kontrola3 (kontakt_id, datum_brisanja, obrisao)
VALUES
(OLD.kontakt_id, SYSDATE(), vUser);
end $$

delimiter ;
```

Zašišimo podatak u kontakti:

```
insert into kontakti values (1, "Ruđer", "Bošković",
str_to_date ("18.05.1711", "%d.%m.%Y"),
str_to_date ("09.06.2024", "%d.%m.%Y"), "xyz");
```

kontakt_id	prezime	ime	rođendan	datum_kreiranja	kreirao
1	Bošković	Ruđer	1711-05-18	2024-06-10	root@localhost

I sad ga obrišimo da vidimo zapis u [kontakti_kontrola3](#).

```
delete from kontakti where prezime="Bošković";
```

kontakt_id	datum_brisanja	obrisao
1	2024-06-10	root@localhost

Primjeri okidača predavača

Primjer Okidača za [INSERT](#)

- Automatsko dodavanje vremena kreiranja
 - o Kada se novi red unese u tablicu, okidač dodaje trenutni vremenski zapis.

```
CREATE TRIGGER AddCreationDate
BEFORE INSERT ON employees
FOR EACH ROW
    SET NEW.creation_date = NOW();
```

- **BEFORE INSERT**: Okidač se aktivira prije dodavanja novog reda
- **NEW.creation_date**: Postavlja vrijednost stupca `creation_date` za novi red

Primjer Okidača za UPDATE

Ažuriranje evidencije promjena

- Prati promjene na stupcu plaća u tablici `radnici`.

```
CREATE TRIGGER LogPlaćaPromjene
AFTER UPDATE ON radnici
FOR EACH ROW
    WHEN (OLD.plaća <> NEW.plaća)
        INSERT INTO plaća_promjene(plaća_id, old_plaća, new_plaća, datum_promjene)
        VALUES (OLD.id, OLD.plaća, NEW.plaća, NOW());
```

- `AFTER UPDATE`: Okidač se pokreće nakon ažuriranja reda
- `WHEN`: Izvršava se samo ako se vrijednost `plaća` promijenila

Pregled i brisanje okidača

- Prikaz svih okidača: `SHOW TRIGGERS;`
- Brisanje okidača: `DROP TRIGGER IF EXISTS naziv_okidača;`

Najbolje prakse i napomene

- Testirati okidače u razvojnem okruženju
- Koristiti okidače oprezno zbog potencijalnog utjecaja na performanse
- Dobro dokumentirati sve okidače u bazi

Kako ukloniti okidač

Nakon što kreirate okidač, možda ćete poželjeti zaustaviti njegovo izvršenje iz nekog razloga. U tom slučaju možete ispustiti okidač.

Za ispuštanje okidača upotrijebite naredbu `DROP TRIGGER`. Naredba zahtijeva samo naziv okidača. Naredbu možete koristiti ovako:

```
DROP TRIGGER šifra_hasher;
```

Pokretanje ovog upita uklonit će okidač koji smo stvorili iznad i svaki zapis umetnut od sada neće imati šifriranu lozinku.

Da biste ovo testirali, umetnite isti zapis kao prije i provjerite rezultat.

Novostvoreni zapis nema šifriranu lozinku.

ime_prezime	email	korisnik	šifra
Branko Ćelap	branko@test.com	branko1	f2bae9c62dcc5a623503ddc64acaa5c2
Branko Ćelap	branko@test.com	branko1	šifra

Nešto na što trebate obratiti pažnju: ako potpuno odbacite tablicu, svi povezani okidači također se automatski odbacuju.

Kada koristiti okidače

- **Bilježenje:** možete imati okidač za automatsko pisanje u drugu tablicu pri umetanju, ažuriranju ili brisanju zapisa iz tablice.
- **Provjera ispravnosti podataka:** možete napisati okidač kako biste osigurali da su podaci određene vrste i da se po potrebi mogu postaviti točne vrijednosti.
- **Sinkronizacija podataka:** možete koristiti okidač za ažuriranje povezanih tablica. Na primjer, u tablici e-trgovine, svaki put kada se kreira zapis o prodaji, okidač može ažurirati saldo dobavljača. Ili ako je zapis dobavljača izbrisana, okidač može ukloniti sve njihove proizvode.

Primjer transakcije, okidača i procedure

Ako se dogodio uspješan update na skladištu želimo da se pokrene okidač i da u tablicu zapisnik nešto zapišemo (promjena količine za proizvod, ime proizvoda, stara količina i nova količina i datum promjene). Bilo bi bolje da tablica ima više stupaca ali iskoristit ćemo tablicu zapisnik.

Dakle trebamo tri stvari: proceduru, okidač i transakciju.

Napisat ćemo proceduru koja u sebi ima transakciju. Kada neko treba naručiti proizvod, pozvat će proceduru i poslat će podatke proceduri. Dakle krećemo s procedurom. Kao što smo rekli u poglavlju [Uvod u MySQL procedure](#), Napravimo prvo strukturu procedure:

```
DELIMITER $$  
CREATE PROCEDURE NaruciProizvod()  
BEGIN  
    -- SQL naredbe  
END $$  
DELIMITER;
```

Proceduru ćemo nazvati [NazoviProizvod](#).

Kada naručujemo proizvod želimo saznati ID proizvoda i količinu. Tko je kupac ćemo hardkodirati. Dakle trebamo kreirati ulazne parametre: [ProizvodID](#), [Kolicina](#). Izlazne parametre nećemo imati - ne vraćamo ništa van. Možemo tu ubaciti [KupacID](#), [KomercijalistID](#), [KreditnaKarticaID](#) i ostale informacije potrebne da se kreira stavka za taj račun i u konačnici račun. Mi ćemo to hardkodirati. Ulazne parametre definiramo sa [in](#) odmah u zagradama [NaruciProizvod\(\)](#).

```
delimiter $$  
create procedure NaruciProizvod(  
    in ProizvodID int,  
    in Kolicina int  
)  
begin  
...  
end
```



Prvo želimo kreirati lokalne varijable sa `declare`. Trebamo informacije o proizvodu jer moramo selektirati proizvod po ID jer moramo doći do trenutne količine proizvoda i njegove cijene jer je moramo upisati tu cijenu po komadu i shodno tome ukupnu cijenu u odnosu na količinu.

Imat ćemo dvije lokalne varijable u kojima ćemo kada napravimo `select` iz tablice `proizvod` zapisati si privremeno podatke o količini na skladištu i cijeni proizvoda.

```
begin
    declare RaspolozivaKolicina int;
    declare CijenaPoKomadu decimal;
end $$
```

delimiter ;

stavka	
IDStavka	INT(11)
RacunID	INT(11)
Kolicina	INT(11)
ProizvodID	INT(11)
CijenaPoKomadu	DECIMAL(12,2)
PopustUPostocima	DECIMAL(12,2)
UkupnaCijena	DECIMAL(38,6)
Indexes	

Primijetite točku-zarez ; iza oba `declare`.

Nakon definiranja lokalnih varijabli trebamo definirati transakciju, kao što je to definirano u poglavlju Početak transakcije, sa `start transaction`; U ovom trenutku se nešto počinje mijenjati na bazi. Te promjene neće biti odmah vidljive nego ovisno o nivou izolacije. Po defaultu ona je u `REPEATABLE READ` modu, što znači da osigurava da ako se podaci jednom pročitaju unutar transakcije, svako daljnje čitanje istih podataka iste transakcije vraća isti rezultat. Ovo ne sprječava „fantomska čitanja“, gdje se novi zapisi koji odgovaraju upitima transakcije mogu dodati od strane drugih transakcija.

Ako imate drugu transakciju koja će nešto čitati i ovu koja nešto zapisuje, s onom koja čita nećete moći pročitati informacije koja je promijenila transakcija koja zapisuje dok transakcija nije potvrđena sa `commit`. Iza `start transaction`; krenemo raditi:

```
update Proizvod
set MinimalnaKolicinaNaSkladistu = RaspolozivaKolicina - Kolicina
```

Prvo radimo `update` tablice `Proizvod`. Nova vrijednost je `RaspolozivaKolicina` (lokalna varijabla) ali tu je problem jer nemamo tu vrijednost. Moramo je dobiti sa `select` i to prije. `Kolicina` je nešto što će ući kroz ulazni parametar u proceduru.

`MinimalnaKolicinaNaSkladistu` i `CijenaBezPDV` želimo zapisati u lokalne varijable `RaspolozivaKolicina` i `CijenaPoKomadu`.

```
start transaction;

select MinimalnaKolicinaNaSkladistu, CijenaBezPDV
into RaspolozivaKolicina, CijenaPoKomadu
from Proizvod
where IDProizvod = ProizvodID;
```

proizvod	
IDProizvod	INT(11)
Naziv	VARCHAR(50)
BrojProizvoda	VARCHAR(25)
Boja	VARCHAR(15)
MinimalnaKolicinaNaSkladistu	INT(11)
CijenaBezPDV	DECIMAL(12,2)
PotkategorijaID	INT(11)
Indexes	

...

Kroz ulazni parametar u proceduru dobili smo ulazni parametar `ProizvodID` koji želimo naručiti. Želimo selektirati taj proizvod iz tablice jer želimo dobiti informacije o raspoloživoj količini i cijeni i to zapisati u lokalne varijable. To nam treba jer ih želimo kasnije koristiti u drugom upititu.

U početku transakcije imamo s upitom rješit ćemo zapisivanje podataka u lokalne varijable a nakon toga kreće proces update-a. Još uvijek nigdje ne radimo provjere.

racun	
IDRacun	INT(11)
DatumIzdavanja	DATETIME
BrojRacuna	VARCHAR(25)
KupacID	INT(11)
KomerčijalistID	INT(11)
KreditnaKarticaID	INT(11)
Komentar	VARCHAR(128)
Indexes	

Transakcija u ovom trenutku još uvijek nije potvrđena. Dakle sljedeći korak bi bio kreiranje računa.

```
insert into Racun (DatumIzdavanja, BrojRacuna,
KupacID)
values (now(), '1225877546', 1);
```

Datum izdavanja je `now()`. Broj računa je nekakav string. Prepostaviti ćemo da imamo kupca sa ID 1.

Time smo napravili insert u `racun`. Sada trebamo napraviti insert u stavku. Možemo vidjeti da nam tamo treba ID računa. A taj ID računa će te tek kreirati. Dakle on nam treba:

```
set @racunID = last_insert_id();
```

Ovim redom postavljamo sadržaj sesijske varijable kao sadržaj zadnjeg inserta. Kako sada imamo taj `racunID` možemo krenuti u proces inserta Stavke. U stavku trebamo upisati `RacunID`, `Kolicina`, `ProizvodID`, `CijenaPoKomadu`, `PopustUPostocima` (nema ga) i Ukupna Cijena. Sve imamo.

stavka	
IDStavka	INT(11)
RacunID	INT(11)
Kolicina	INT(11)
ProizvodID	INT(11)
CijenaPoKomadu	DECIMAL(12,2)
PopustUPostocima	DECIMAL(12,2)
UkupnaCijena	DECIMAL(38,6)
Indexes	

```

        set @racunID = last_insert_id();

        insert into Stavka
        (RacunID, Kolicina, ProizvodID, CijenaPoKomadu, PopustUPostocima,
UkupnaCijena)
        values
        (@racunID, Kolicina, ProizvodID, CijenaPoKomadu, 0, CijenaPoKomadu *
Kolicina);
    
```

Pogledajmo što do sada imamo (flow procedure). Definirali smo 2 lokalne varijable `RaspolozivaKolicina` i `CijenaPoKomadu`. Pokreće se transakcija i nakon toga kontroliramo sve što se događa. Radimo select i zapisujemo u lokalne varijable. Nakon toga radimo `update Proizvod`. Sa `insert` insertamo podatke u `Racun`. Izvučemo ID iz zapisa (to je red) i zapišemo ga u sesijsku varijablu. Ovdje vidimo 4 upita: select, update, insert, insert.

Pogledajmo što se može desiti. Umanjili smo količinu na skladištu a `insert into Racun` je puknuo. I integritet je narušen. Napravili smo update a narudžba nije dogodila. Moglo se desiti i da nemamo dovoljnu količinu. Rollback možemo kontrolirati tako da ako nemamo dovoljnu količinu vratimo sve.

Provjerit ćemo:

```

if RaspolozivaKolicina >= Kolicina then
    commit;
else
    rollback;
    signal sqlstate '45000'
    set message_text = "Nema dovoljno proizvoda na zalihi.";
end if;
    
```

Naredba `signal` prazni dijagnostičko područje i proizvodi prilagođenu grešku. Za prilagođene greške, preporučeni `sqlstate` je 45000.

Ovdje puno stvari treba provjeriti jer mogu krenuti po zlu. Za početak pogledajmo transakciju koja bi trebala napraviti update proizvoda.

Pokrenimo proceduru i pogledajmo tablicu `proizvod`.

IDProizvod	Naziv	BrojProizvoda	Boja	MinimalnaKolicinaNaSkladistu	CijenaBezPDV	PotkategorijaID
1	Adjustable Race	AR-5381	NULL	700	0.00	NULL
2	Bearing Ball	BA-8327	NULL	700	0.00	NULL
3	BB Ball Bearing	BE-2349	NULL	600	0.00	NULL
4	Headset Ball Bearings	BE-2908	NULL	600	0.00	NULL

Ako pogledamo `racun`, vidimo:

IDRacun	DatumIzdavanja	BrojRacuna	KupacID	KomercijalistID	KreditnaKarticaID	Komentar
75124	2024-06-05 18:33:19	1225877546	1	NULL	NULL	NULL
75123	2004-07-31 00:00:00	SO75123	4189	NULL	10084	NULL
75122	2004-07-31 00:00:00	SO75122	4781	NULL	18719	NULL

Pogledajmo i stavku:

IDStavka	RacunID	Kolicina	ProizvodID	CijenaPoKomadu	PopustUPostocima	UkupnaCijena
99107	75124	50	2	0.00	0.00	0.000000
99106	68157	1	873	2.29	0.00	2.290000
99105	68157	1	923	4.99	0.00	4.990000

Cilj nam je da za ID 1 naručimo 50 komada. Želimo vidjeti da li će se umanjiti proizvod za 50, da li će se kreirati novi račun s ID kupca 1 i nekim brojem računa, da li će se u stavci biti novi racun ID, količina koju naručujemo (to je 50), ID proizvoda, cijena po komadu, ukupan popust i ukupna cijena. To se sve treba promijeniti sa pozivanjem jedne procedure.

Ako je sve OK pogledaćemo proizvod. Umanjen je za 50.

IDProizvod	Naziv	BrojProizvoda	Boja	MinimalnaKolicinaNaSkladistu	CijenaBezPDV	Potkateg
1	Adjustable Race	AR-5381	NULL	650	0.00	NULL
2	Bearing Ball	BA-8327	NULL	700	0.00	NULL
3	BB Ball Bearing	BE-2349	NULL	600	0.00	NULL

U racun je kreiran novi račun:

IDRacun	DatumIzdavanja	BrojRacuna	KupacID	KomercijalistID	KreditnaKarticaID	Komentar
75125	2024-06-07 01:23:06	1225877546	1	NULL	NULL	NULL
75124	2024-06-05 18:33:19	1225877546	1	NULL	NULL	NULL

Vidimo novi datum izdavanja.

Pogledajmo stavku:

IDStavka	RacunID	Kolicina	ProizvodID	CijenaPoKomadu	PopustUPostocima	UkupnaCijena
99108	75125	50	1	0.00	0.00	0.000000
99107	75124	50	2	0.00	0.00	0.000000
99106	68157	1	873	2.29	0.00	2.290000

Vidimo Kolicina 50, ProizvodID, IDRacuna, CijenaPoKomadu, PopustUpostocima i UkupnaCijena.

Pogledajmo neki drugi proizvod koji ima cijenu, npr. IDProizvod 996 u proizvodu

IDProizvod	Naziv	BrojProizvoda	Boja	MinimalnaKolicinaNaSkladistu	CijenaBezPDV	PotkategorijaID
999	Road-750 Black, 52	BK-R19B-52	Crna	75	539.99	2
998	Road-750 Black, 48	BK-R19B-48	Crna	75	539.99	2
997	Road-750 Black, 44	BK-R19B-44	Crna	75	539.99	2
996	HL Bottom Bracket	BB-9108	NULL	375	121.49	5
995	ML Bottom Bracket	BB-8107	NULL	375	101.24	5

Vidimo da ima 375 komada, probat ćemo naručiti 500 kom.

```
call NaruciProizvod(996, 500);
```

Ova procedura ne smije proći - trebamo dobiti ispisano grešku u konzoli. Ne smijemo vidjeti promjenu niti na proizvod, niti na racun, niti na stavka. Upravo to treba spriječiti transakcija.

#	Time	Action	Message	Duration / Fetch
1	01:50:35	call NaruciProizvod(996,500)	Error Code: 1644. Nema dovoljno proizvoda na zalihi.	0.015 sec

Na proizvod se stanje nije promijenilo:

IDProizvod	Naziv	BrojProizvoda	Boja	MinimalnaKolicinaNaSkladistu	CijenaBezPDV	PotkategorijaID
999	Road-750 Black, 52	BK-R19B-52	Crna	75	539.99	2
998	Road-750 Black, 48	BK-R19B-48	Crna	75	539.99	2
997	Road-750 Black, 44	BK-R19B-44	Crna	75	539.99	2
996	HL Bottom Bracket	BB-9108	NULL	375	121.49	5
995	ML Bottom Bracket	BB-8107	NULL	375	101.24	5

U **racun** nema promjene. Ni u stavka. To znači da je rollback sve vratio nazad.

Probat ćemo naručiti 50 komada istog proizvoda. To treba proći.

Evo proizvoda i tamo je stanje umanjeno:

IDProizvod	Naziv	BrojProizvoda	Boja	MinimalnaKolicinaNaSkladistu	CijenaBezPDV	PotkategorijaID
999	Road-750 Black, 52	BK-R19B-52	Crna	75	539.99	2
998	Road-750 Black, 48	BK-R19B-48	Crna	75	539.99	2
997	Road-750 Black, 44	BK-R19B-44	Crna	75	539.99	2
996	HL Bottom Bracket	BB-9108	NULL	325	121.49	5
995	ML Bottom Bracket	BB-8107	NULL	375	101.24	5

U racun kreiran je broj računa (broj je hardkodiran):

IDRacun	DatumIzdavanja	BrojRacuna	KupacID	KomercijalistID	KreditnaKarticaID	Komentar
75128	2024-06-07 01:52:52	1225877546	1	NULL	NULL	NULL
75125	2024-06-07 01:23:06	1225877546	1	NULL	NULL	NULL
75124	2024-06-05 18:33:19	1225877546	1	NULL	NULL	NULL
75123	2004-07-31 00:00:00	S075123	4189	NULL	10084	NULL

U stavka vidimo ispravnu količinu i **UkupnaCijena**:

IDStavka	RacunID	Kolicina	ProizvodID	CijenaPoKomadu	PopustUPostocima	UkupnaCijena
99111	75128	50	996	121.00	0.00	6050.000000
99108	75125	50	1	0.00	0.00	0.000000
99107	75124	50	2	0.00	0.00	0.000000
99106	68157	1	873	2.29	0.00	2.290000
99105	68157	1	923	4.99	0.00	4.990000

Probat ćemo zapisati transakciju za nepostojeći ID proizvoda.

```
call NaruciProizvod(8000,50);
```

Ovo ne prolazi. Prvi select nije ništa dohvatio. Ako pogledamo racun vidjet ćemo da je upisao jer nema rollback. stavka se nije upisala. Račun se kreirao jer nemamo grešku. Ovo je problem s integritetom podataka. Ovo trebamo ispraviti tako da CijenaPoKomadu ne smije biti null. Ako jeste trebamo napraviti rollback.

Evo cijelog koda:

```
-- Procedura za narudžbu proizvoda - transakcija
delimiter $$

create procedure NaruciProizvod(
    in ProizvodID int,
    in Kolicina int
)
begin
    declare RaspolozivaKolicina int;
    declare CijenaPoKomadu decimal;

    start transaction;

    select MinimalnaKolicinaNaSkladistu, CijenaBezPDV
    into RaspolozivaKolicina, CijenaPoKomadu
    from Proizvod
    where IDProizvod = ProizvodID;

    if CijenaPoKomadu is null then
        rollback;
    else
        update Proizvod
        set MinimalnaKolicinaNaSkladistu = RaspolozivaKolicina - Kolicina
        where IDProizvod = ProizvodID;

        insert into Racun (DatumIzdavanja, BrojRacuna, KupacID)
        values(now(), '1225877546', 1);
    end if;
end;
```

```

        set @racunID = last_insert_id();

        insert into Stavka
        (RacunID, Kolicina, ProizvodID, CijenaPoKomadu, PopustUPostocima,
        UkupnaCijena)
        values
        (@racunID, Kolicina, ProizvodID, CijenaPoKomadu, 0, CijenaPoKomadu *
        Kolicina);

        if RaspolozivaKolicina >= Kolicina then
            commit;
        else
            rollback;
            signal sqlstate '45000'
            set message_text = "Nema dovoljno proizvoda na zalihi.";
        end if;
    end if;

end $$

delimiter ;

```

Napisat ćemo još okidač (engl. trigger) [AzurirajPromjenuZaliha](#) koji će kada se radi update proizvoda da se zabilježi što se dogodilo. Za to je predviđena tablica zapisnik u kojoj nema ništa osim ID i poruke. Naravno bilo bi poželjno da ima više polja (npr. datum).

Mi ćemo zapisati koja količina je bila (stara količina), nova količina i datum.

Prije poziva sam procedure treba definirati okidač.

Krenut ćemo od kreiranja okidača kao što je objašnjeno u poglavlju [Kako kreirati okidač](#):

```

CREATE TRIGGER AzurirajPromjenuZaliha
okidač_vrijeme
okidač_događaj
ON tablica_ime
FOR EACH ROW
begin
    okidač_tijelo
end $$

delimiter ;

```



Taj dio stavit ćemo iza procedura za narudžbu proizvoda (gdje je demonstrirana transakcija i funkcija) i iza poziva funkcija.

Krenut ćemo od kreiranja okidača kao što je objašnjeno u poglavlju [Kako kreirati okidač](#).

`naziv_okidača` smo odredili. Bit će to AzurirajPromjenuZaliha. `okidač_vrijeme` će biti `AFTER`, dakle okidač će se pokrenuti nakon što se događaj dogodio. `okidač_događaj` koji nas zanima je `UPDATE`. Svi okidači su vezani za tablicu `Proizvod`.

```
CREATE TRIGGER AzurirajPromjenuZaliha
after update on Proizvod
FOR EACH ROW
begin
    okidač_tijelo
end $$
```

`delimiter ;`

Idemo napisati to tijelo okidača. Imamo prije zapisivanja i staro i novo stanje sa old i new. U ovom trenutku ne postoji procedura koja pravi update količine na skladишtu ali ne znači da je neće biti sutra. Moguće je da neko napravi proceduru i napravi update procedure. Dakle treba napraviti ograničenje procedure tako da provjerimo da li je stara MinimalnaKolicinaNaSkladistu različita od nove MinimalnaKolicinaNaSkladistu, dakle prije i poslije update-a onda ćemo zapisati neke vrijednosti u tablicu zapisnik.

```
delimiter $$
```

```
create trigger AzurirajPromjenuZaliha
after update on Proizvod
for each row
begin
    if old.MinimalnaKolicinaNaSkladistu <> new.MinimalnaKolicinaNaSkladistu
        then
            insert into Zapisnik (Poruka)
            values(concat('Promijenjena količina za proizvod',
            new.Naziv,
            ', Staro stanje:',
            old.MinimalnaKolicinaNaSkladistu,
            ', Novo stanje:',
            new.MinimalnaKolicinaNaSkladistu,
            ', Datum: ',
            now()));
    end if;
end $$
```

`delimiter ;`

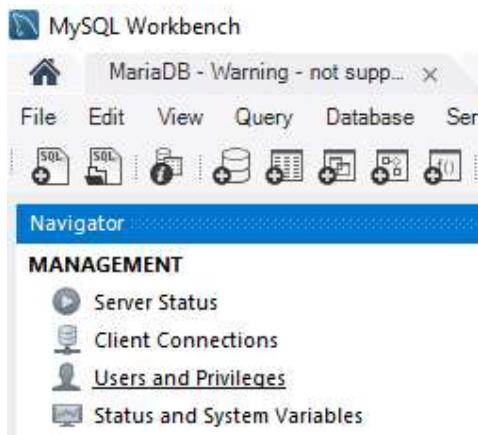
`concat` spaja stringove jer sve zapisujemo u isto polje.

```
1 Promijenjeni podaci kartice broj 77776411189527. Staro trajanje je 2008-10, novo trajanje je 2014-5
2 Promijenjena količina za proizvod Bearing Ball, Staro stanje:750, Novo stanje:700, Datum: 2024-06-05 18:33:19
3 Promijenjena količina za proizvod Adjustable Race, Staro stanje:700, Novo stanje:650, Datum: 2024-06-07 01:23:06
6 Promijenjena količina za proizvod HL Bottom Bracket, Staro stanje:375, Novo stanje:325, Datum: 2024-06-07 01:52...
NULL NULL
```

Prvi red u zapisniku je izgeneriran prilikom kreiranja tablice. Također u tablici zapisnik polje id je definirano kao `AUTO_INCREMENT` gdje počinje od 2 (jer je 1 već zauzet).

Ovime je automatiziran proces narudžbe proizvoda na nivou baze podataka.

Moguće je upravljati korisnicima i npr. ne dozvoliti korisnicima da prčkaju po bazama. User sa svim ovlastima (root) ne smije biti omogućen u produkciji.



The screenshot shows the 'MariaDB' logo and the title 'Users and Privileges'. On the left, a table lists user accounts:

User	From Host
(!) <anonymous>	%
pma	localhost
root	127.0.0.1
root	::1
root	localhost

The right panel is titled 'Details for account root@localhost' and contains tabs for 'Login', 'Account Limits', 'Administrative Roles', and 'Schema Privileges'. The 'Login' tab is selected, showing fields for 'Login Name' (root), 'Authentication Type' (Standard), 'Limit to Hosts Matching' (localhost), 'Password' (*****), and 'Confirm Password' (*****). A note says 'No password is set for this account.' Buttons at the bottom include 'Add Account', 'Delete', 'Refresh', 'Revert', and 'Apply'.

root korisnik ima sva prava

Sa Add account možemo dodati novog korisnika. Predavač preporuča da se ime logon name sastoji od imena baze (aplikacije) i u nastavku web, mobile, desktop. Tako možemo imati različite funkcionalnosti za različne korisnike. Npr. web aplikacija može glumiti backoffice aplikaciju za uređivanje sadržaja a mobilna aplikacija je čitav konzument. Naravno ne želite da korisnik mobilne aplikacije ne može promjeniti sadržaj (engl. content). Treba unijeti i Password i potvrditi ga.

Moguće je ograničiti:

Login	Account Limits	Administrative Roles	Schema Privileges
Max. Queries:	<input type="text" value="0"/>	Number of queries the account can execute within one hour.	
Max. Updates:	<input type="text" value="0"/>	Number of updates the account can execute within one hour.	
Max. Connections:	<input type="text" value="0"/>	The number of times the account can connect to the server per	
Concurrent Connections:	<input type="text" value="0"/>	The number of simultaneous connections to the server the acc	

Details for account root@localhost

Login Account Limits Administrative Roles Schema Privileges

Role	Description
<input checked="" type="checkbox"/> DBA	grants the right
<input checked="" type="checkbox"/> MaintenanceAdmin	grants rights n
<input checked="" type="checkbox"/> ProcessAdmin	rights needed
<input checked="" type="checkbox"/> UserAdmin	grants rights t
<input checked="" type="checkbox"/> SecurityAdmin	rights to mana
<input checked="" type="checkbox"/> MonitorAdmin	minimum set o
<input checked="" type="checkbox"/> DBManager	grants full righ
<input checked="" type="checkbox"/> DBDesigner	rights to creat
<input checked="" type="checkbox"/> ReplicationAdmin	rights needed
<input checked="" type="checkbox"/> BackupAdmin	minimal rights
<input checked="" type="checkbox"/> Custom	custom role

Global Privileges
<input checked="" type="checkbox"/> ALTER
<input checked="" type="checkbox"/> ALTER ROUTINE
<input checked="" type="checkbox"/> CREATE
<input checked="" type="checkbox"/> CREATE ROUTINE
<input checked="" type="checkbox"/> CREATE TABLESPACE
<input checked="" type="checkbox"/> CREATE TEMPORARY TABLES
<input checked="" type="checkbox"/> CREATE USER
<input checked="" type="checkbox"/> CREATE VIEW
<input checked="" type="checkbox"/> DELETE
<input checked="" type="checkbox"/> DROP
<input checked="" type="checkbox"/> EVENT
<input checked="" type="checkbox"/> EXECUTE
<input checked="" type="checkbox"/> FILE
<input checked="" type="checkbox"/> GRANT OPTION
<input checked="" type="checkbox"/> INDEX
<input checked="" type="checkbox"/> INSERT
<input checked="" type="checkbox"/> LOCK TABLES
<input checked="" type="checkbox"/> PROCESS
<input checked="" type="checkbox"/> REFERENCES
<input checked="" type="checkbox"/> RELOAD
<input checked="" type="checkbox"/> REPLICATION CLIENT
<input checked="" type="checkbox"/> REPLICATION SLAVE
<input checked="" type="checkbox"/> SELECT
<input checked="" type="checkbox"/> SHOW DATABASES
<input checked="" type="checkbox"/> SHOW VIEW

Revoke All Privileges

Revert Apply

MariaDB
Users and Privileges

User Accounts

User	From Host
(!) <anonymous>	%
pma	localhost
root	127.0.0.1
root	::1
root	localhost

Details for account root@localhost

Login Account Limits Administrative Roles Schema Privileges

Schema Privileges

Schema Privileges

Schema and Host fields may use % and _ wildcards.
The server will match specific entries before wildcarded ones.

Revoke All Privileges Delete Entry... Add Entry...

New Schema Privilege Definition

Select the Schema for which the user 'root' will have the privileges you want to define.

Schema

All Schema (%) This rule will apply to any schema name.

Schemas matching pattern: This rule will apply to schemas that match the given name or pattern.
You may use _ and % as wildcards in a pattern.
Escape these characters with \ in case you want their literal value.

Selected schema: Select a specific schema name for the rule to apply to.

adventureworkshop

Zašto napredno PHP programiranje?

- Potreba za naprednim tehnikama:
 - o Kako web aplikacije postaju složenije, potreba za naprednim tehnikama programiranja raste.
- Primjeri:
 - o Objektno orijentirano programiranje, obrasci dizajna, upravljanje bazama podataka.
- Cilj modula:
 - o Steći vještine potrebne za izgradnju robusnih, sigurnih i efikasnih web aplikacija.

Pregled Modula

- **Struktura gradiva:**
 - o Objektno-Orijentirano Programiranje (OOP),
 - o Imenski prostori i Autoload
 - o Iznimke i upravljanje greškama
 - o Obrasci dizajna i Arhitekture
 - o Rad s bazama podataka
 - o Testiranje
 - o Model-View-Controller (MVC)
 - o Završni pregled i parcijalni ispit
- **Metode nastave:**
 - o Kombinacija predavanja, praktičnih vježbi, i projektnog rada.
- **Očekivanja od polaznika:**
 - o Aktivno sudjelovanje, samostalno učenje i timski rad na projektima.

Resursi za Učenje

- Online dokumentacija:
 - o <https://www.php.net/manual/en/>
 - o <https://www.phptutorial.net/php-oop/>
- Preporučena literatura:
 - o [Programming PHP: Creating Dynamic Web Pages](#)
 - o [Learn PHP in One Day and Learn It Well. PHP for Beginners with Hands-on Project.](#)

Uvod u Objektno-Orijentirano Programiranje

- **Definicija OOP:**
 - o Programski pristup temeljen na objektima i klasama.
- **Prednosti OOP-a:**
 - o **Modularnost:** Omogućava razvoj aplikacija kroz modularni pristup, gdje je lakše održavati i ažurirati dijelove aplikacije neovisno.
 - o **Ponovna upotreba koda:** Nasljeđivanje i polimorfizam omogućavaju veću ponovnu upotrebu koda.
 - o **Skalabilnost:** OOP pristup olakšava upravljanje većim projektima i njihovo skaliranje.

- **Nedostaci OOP-a**

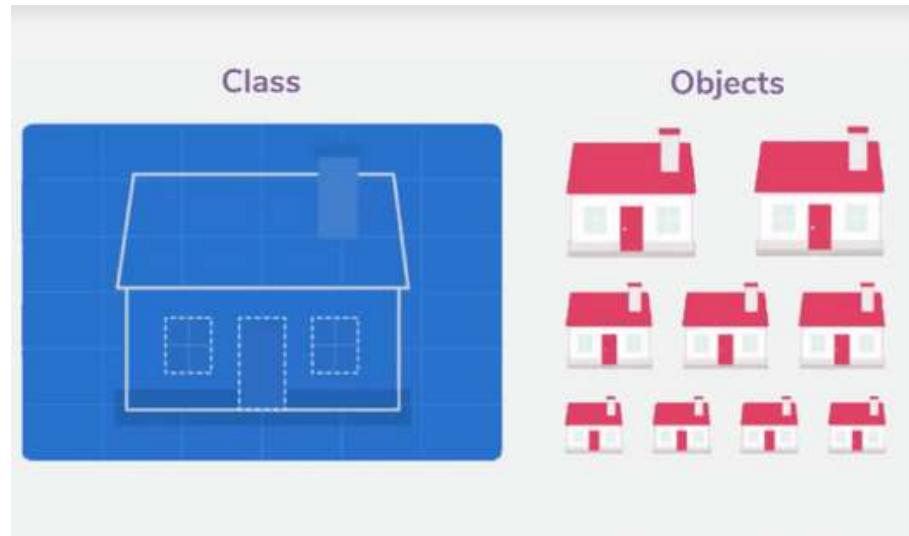
- o **Složenost:** Objektno orijentirane aplikacije mogu biti složenije za razumijevanje i razvoj.
- o **Performanse:** Objekti i klasni mehanizmi mogu zahtijevati više resursa procesora i memorije.

Osnovne karakteristike

- **Apstrakcija:**
 - o Izdvajanje esencijalnih karakteristika od sekundarnih.
- **Enkapsulacija:**
 - o Podaci unutar objekta zaštićeni su od izravnog pristupa izvana. Objekti komuniciraju putem definiranih interface-a.
- **Naslijedivanje:**
 - o Omogućava jednoj klasi da naslijedi svojstva (atribute i metode) druge klase.
- **Polimorfizam:**
 - o Mogućnost da se isto ime funkcije koristi za različite tipove. To omogućava da se ista funkcija ponaša različito ovisno o kontekstu u kojem se koristi.

Klase i objekti

- **Definicija klase:**
 - o 'Blueprint' za stvaranje objekata koji sadrži svojstva i metode.
- **Definicija objekta:**
 - o Instanca klase.



Objekti su instance klase

```
<?php

// Nacrt objekta (Razred, Klasa)
class Car{
    // Svojstva (Atributi), ispred je access modifier public, oni će nam kasnije
    // omogućiti učahurivanje
    public $brand;
    public $model;
```

Shallow copy i deep copy

Konvencija klasa nalaže da kod davanja imena svako prvo slovo riječi bude veliko slovo. Atributi se ponekad nazivaju i polja.

```
// ako probamo var_dump(Car) php ne zna što je to
// ključna riječ new pokreće distanciranje objekta
// $car je instanca klase Car
$car = new Car();
$car1 = new Car();
var_dump($car, $car1);
```

Kada izvršimo ovaj dio koda vidimo:

```
object(Car)#1 (2) {
    ["brand"]=>
    NULL
    ["model"]=>
    NULL
}
object(Car)#2 (2) {
    ["brand"]=>
    NULL
    ["model"]=>
```

Objekti su označeni sa #1 i #2. Na taj način možemo ih razlikovati po broju ali vrijednosti su im identične. Ti objekti su na gomili (engl. heap) i nisu identični. Dakle kada pogledamo sa:

```
var_dump($car, $car1);
```

Ti objekti su identični.

```
bool(true)
```

A identični su jer su objekti identični - nema ih.

Ako u jedan objekt u klasi stavimo sadržaj:

```
$car = new Car();
```

```
$car->brand = "Audi";  
  
$car1 = new Car();  
var_dump($car == $car1);
```

Više nisu identični:

```
bool(false)
```

Naravno treba zapaziti i da promjena u jednom objektu ne utječe na drugi objekt, bez obzira što su oba objekta instance iste klase.

Napraviti ćemo kopiju (klon) postojećeg objekta.

```
class Car{  
    public $brand;  
    public $model;  
  
}  
  
$car = new Car();  
$car->brand = "Audi";  
  
$car1 = new Car();  
$car2 = $car1;  
var_dump($car, $car1, $car2);
```

Dobit ćemo:

```
object(Car)#1 (2) {  
    ["brand"]=>  
    string(4) "Audi"  
    ["model"]=>  
    NULL  
}  
object(Car)#2 (2) {  
    ["brand"]=>  
    NULL  
    ["model"]=>  
    NULL  
}  
object(Car)#2 (2) {  
    ["brand"]=>  
    NULL  
    ["model"]=>  
    NULL
```

```
}
```

Pogledajmo sada cijeli kod:

```
<?php

class Car{
    // Svojstva (Atributi), ispred je access modifier public, oni će nam kasnije
    omogućiti učahurivanje
    public $brand;
    public $model;

}

$car = new Car();
$car->brand = "Audi";

$car1 = new Car();

$car2 = $car1;
$car2->model = "X5";

$car3 = $car1;
$car3->model = "X6";

var_dump($car, $car1, $car2);
var_dump($car == $car1);
var_dump($car2 == $car3);
```

Rezultat ovoga je:

```
object(Car)#1 (2) {
    ["brand"]=>
        string(4) "Audi"
    ["model"]=>
        NULL
}
object(Car)#2 (2) {
    ["brand"]=>
        NULL
    ["model"]=>
        string(2) "X6"
}
object(Car)#2 (2) {
    ["brand"]=>
```

```

NULL
["model"]=>
string(2) "X6"
}
bool(false)
bool(true)

```

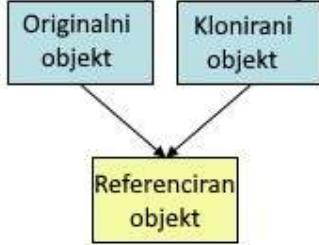
Primijetimo da su objekti `$car2` i `$car3` su na gomili (engl. heap) i pokazuju na isti objekt. Dakle nemamo novi prostor zauzet u memoriji, nego samo varijabla pokazuje na isti prostor. Promjena u `$car2` mijenja i `$car1` jer mijenjamo isti objekt.

Treba za zadaću pogledati **shallow copy** i **deep copy** u PHP-u.

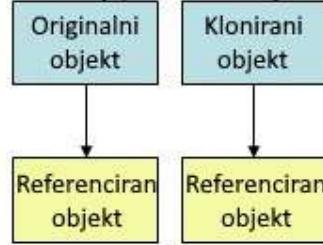
Duboke kopije dupliciraju sve. Duboka kopija, što znači da su polja dereferencirana: umjesto referenci na objekte koji se kopiraju, stvaraju se novi objekti kopije za sve referencirane objekte, a reference na njih stavljaju se u novi objekt. Duboka kopija kreira novu instancu objekta i rekursivno kopira sve objekte na koje se referira originalni objekt. Svaki objekt unutar strukture originalnog objekta se rekursivno kopira, što rezultira potpuno nezavisnom kopijom.

Plitke kopije dupliraju se što je manje moguće. Plitka kopija zbirke je kopija strukture zbirke, a ne elemenata. Uz plitku kopiju, dvije zbirke sada dijele pojedinačne elemente. Plitka kopija kreira novu instancu objekta, ali ne kopira objekte na koje se referira originalni objekt. Umjesto toga, kopira samo reference na te objekte. Kao rezultat toga, originalni objekt i kopija dijele iste referencirane objekte.

Plitko (površno, engl. Shallow) kloniranje



Duboko (engl. Deep) kloniranje



```

$obj1 = clone $obj;

```

Interno, ključna riječ `clone` kopira sve varijable iz prvog objekta u novi objekt, zatim poziva čarobnu funkciju `__clone()` za klasu koju kopira. Možete nadjačati `__clone()` ako želite, što vam daje fleksibilnost za izvođenje dodatnih radnji kada se varijabla kopira - možete je zamisliti kao konstruktor za kopirani objekt.

Evo primjera direktnom pristupu svojstvima klase. Kada su svojstva klase deklarirana kao javna tj. `public`, to je moguće.

```

<?php

// Nacrt objekta (Razred, Klasa)

```

```
class Car{
    public $brand;
    public $model;

}

$car = new Car();
$car->brand = "Audi";

$car1 = new Car();

$car2 = $car1;
$car2->model = "X5";

$car3 = clone $car1;
// ovo je potpuno novi objekt i ne utječe na objekte $car1 i $car2
$car3->model = "X6";

var_dump($car, $car1, $car2, $car3);
var_dump($car == $car1);
var_dump($car2 == $car3);
```

Rezultat je:

```
class Car#1 (2) {
    public $brand => string(4) "Audi"
    public $model => NULL
}
class Car#2 (2) {
    public $brand => NULL
    public $model => string(2) "X5"
}
2
class Car#2 (2) {
    public $brand => NULL
    public $model => string(2) "X5"
}
class Car#3 (2) {
    public $brand => NULL
    public $model => string(2) "X6"
}
bool(false)
bool(false)
```

Kloniranjem stvaramo potpuno novi objekt.

Shallow Copy: Kopira samo referencu na podobjekte. Ako promenite podobjekat u jednom objektu, ta promena će biti vidljiva i u kopiji.

Deep Copy: Rekursivno kopira svaki podobjekat. Originalni i kopirani objekat su potpuno nezavisni jedan od drugog.

Kada koristite clone u PHP-u, podrazumevano dobijate plitku kopiju. Ako želite duboku kopiju, morate ručno implementirati rekursivno kopiranje u metodi `__clone`.

Kontrola pristupa svojstava i metoda (access modifier)

- **Public:**
 - o Svojstva i metode dostupne unutar i izvan klase (u objektu).
- **Protected:**
 - o Svojstva i metode dostupne unutar klase i njenih nasljednika.
- **Private:**
 - o Svojstva i metode dostupne samo unutar klase.

Moguće je direktno pristupiti svojstvima klase ako su ona deklarirana kao javna tj. `public`. Međutim, to nije preporučena praksa u objektno orijentiranom programiranju (OOP) zbog nekoliko razloga vezanih za principe dobre prakse i održavanja koda.

Evo i razloga zašto `public` pristup nije preporučen:

Nema kontrole nad pristupom:

Kada su svojstva javna, bilo ko može da ih menja bez ikakvih ograničenja ili validacije. Ovo može dovesti do neželjenih stanja objekta.

Nema enkapsulacije:

Enkapsulacija je jedan od osnovnih principa OOP-a, koja omogućava skrivanje unutrašnje reprezentacije objekta i pruža jasno definirane metode za interakciju sa objektom.

Teže održavanje koda:

Ako direktno pristupate svojstvima, svaka promjena u strukturi klase može zahtjevati promjene u svim dijelovima koda koji koriste tu klasu.

Teže dodavanje logike:

Ako svojstva nisu privatna, teško je dodati logiku za validaciju ili bilo koju drugu logiku koja treba da se izvrši prilikom postavljanja ili dobivanja vrijednosti.

Ključna riječ \$this

Pogledajmo kako se koristi ključna riječ `$this`:

```
class Vehicle {  
    public $make;  
    protected $engine;  
    private $vin;  
    public function __construct($make, $engine, $vin)
```

```

    {
        $this->make = $make;
        $this->engine = $engine;
        $this->vin = $vin;
    }
}

```

\$this je ključna riječ koja se koristi za referenciranje trenutnog objekta instance klase.

Omogućava pristup instancama metoda i varijabli definiranim u unutar iste klase.

Evo još jednog primjera kako bi bilo jasnije:

```

<?php

class Example {
    // Svojstva klase s različitim modifikatorima vidljivosti
    public $publicProperty = 'Public Property';
    protected $protectedProperty = 'Protected Property';
    private $privateProperty = 'Private Property';

    // Javno dostupna metoda
    public function publicMethod() {
        echo "Inside publicMethod---:\n";
        echo $this->publicProperty . "\n";      // Dostupno
        echo $this->protectedProperty . "\n"; // Dostupno
        echo $this->privateProperty . "\n\n";   // Dostupno
    }

    // Zaštićena metoda
    protected function protectedMethod() {
        echo "Inside protectedMethod---:\n";
        echo $this->publicProperty . "\n";      // Dostupno
        echo $this->protectedProperty . "\n"; // Dostupno
        echo $this->privateProperty . "\n\n";   // Dostupno
    }

    // Privatna metoda
    private function privateMethod() {
        echo "Inside privateMethod---:\n";
        echo $this->publicProperty . "\n";      // Dostupno
        echo $this->protectedProperty . "\n"; // Dostupno
        echo $this->privateProperty . "\n\n";   // Dostupno
    }

    // Javni pristup zaštićenoj i privatnoj metodi
}
```

```
public function accessProtectedAndPrivate() {
    $this->protectedMethod(); // Poziv zaštićene metode unutar klase
    $this->privateMethod(); // Poziv privatne metode unutar klase
}

// Kreiranje instance klase Example
$example = new Example();

// Poziv javnog svojstva i metode
echo $example->publicProperty . "\n"; // Direktan pristup javnom svojstvu
$example->publicMethod(); // Poziv javne metode

// Pristup zaštićenim i privatnim svojstvima izvan klase (uzrokuje grešku)
// echo $example->protectedProperty; // Greška: zaštićeno svojstvo
// echo $example->privateProperty; // Greška: privatno svojstvo

// Poziv metoda koje pristupaju zaštićenim i privatnim metodama
$example->accessProtectedAndPrivate(); // Ovdje možemo pristupiti zaštićenoj i
privatnoj metodi jer se one pozivaju unutar javne metode

// Direktan poziv zaštićenih i privatnih metoda izvan klase (uzrokuje grešku)
// $example->protectedMethod(); // Greška: zaštićena metoda
// $example->privateMethod(); // Greška: privatna metoda
?>
```

Konstruktori i destruktori

- **Konstruktor:**
 - o Metoda koja se automatski poziva prilikom stvaranja objekta.
- **Destruktur:**
 - o Metoda koja se poziva kada objekt više nije potreban.

```
class Book {
public function __construct() {
    echo "A new book was created.";
}
public function __destruct() {
    echo "A book is being deleted.";
}
}
$book = new Book();
```

Setteri i Getteri

Getteri i setteri u PHP-u (kao i u drugim objektno orijentiranim jezicima) koriste se za kontrolu pristupa atributima (svojstvima) klase. Oni omogućavaju enkapsulaciju podataka, što je jedan od osnovnih principa objektno orijentiranog programiranja (OOP). Enkapsulacija pomaže u zaštiti podataka unutar objekta i pruža kontroliran način njihovog pristupa i modifikacije. Omogućavaju kontrolu pristupa kako se atributi postavljaju ili vraćaju.

- Zašto Setteri i Getteri?
 - o Omogućavaju kontroliran pristup svojstvima klase.
- Setteri:
 - o Metode koje postavljaju vrijednost svojstva.
- Getteri:
 - o Metode koje vraćaju vrijednost svojstva.

Sa getterima i setterima, možete promijeniti unutrašnju implementaciju bez mijenjanja [interface-a](#) klase.

```
class Account {  
    private $balance;  
  
    public function setBalance($amount) {  
        if ($amount >= 0) {  
            $this->balance = $amount;  
        } else {  
            echo "Balance cannot be negative.";  
        }  
    }  
  
    public function getBalance() {  
        return $this->balance;  
    }  
}  
$acc = new Account();  
$acc->setBalance(100);  
echo "The balance is: " . $acc->getBalance();
```

Evo primjera koji je radio predavač:

Uzet ćemo prvi dio definicije klase `Car` iz drugog primjera. U klasi imamo definiran konstruktor sa `$brand` i `$model` svojstvima.

```
<?php  
  
declare(strict_types=1);
```

```
class Car{
    private string $brand;
    private string $model;

    public function __construct(string $brand, string $model){
        $this->brand = $brand;
        $this->model = $model;
    }
}
```

Za `$brand` ćemo omogućiti da ga možemo i pročitati i promijeniti (trebamo getter i setter) a `$model` samo pročitati (trebamo getter).

```
// getter za brand vraća string
public function getBrand(): string{
    return $this->brand;
}

// setter za brand postavlja
public function setBrand(string $brand): void{
    $this->brand = $brand;
}

// getter za model vraća string
public function getModel(): string{
    return $this->model;
}

}

$car = new Car("Audi", "A4");
var_dump($car->getBrand()); // Audi

// Poziv setter metode za promjenu brand-a
$car->setBrand("Mercedes");
var_dump($car->getBrand()); // Mercedes

$car1 = new Car("BMW", "X5");
var_dump($car1->getBrand());
```

Getter se imenuje tako da obično ide ključna riječ i ime svojstva kojeg hvata. To je nepisano pravilo. Vidimo da će `getBrand` vratiti string (iza dvotočke). Vraća `brand`, a `brand` je string. Ključna riječ `$this` znači „u ovoj klasi“ ili „u ovom objektu“. Gledamo kontekst izvođenja gdje se `$this` izvodi. Sada je

`getBrand()` unutar klase i neće javljati grešku. Podatak je enkapsuliran (predavač kaže začahuren) u objektu i ne možemo mu pristupiti izvana osim pozivanjem metode `getBrand()`. I dalje vrijednost nije moguće promijeniti sa `$car->brand = 'Tesla';` interpreter će dojaviti grešku.

Ako želimo zapisati `$brand` moramo definirati setter. Kada ih definiramo, omogućili smo pametnu inicijalizaciju objekta. Učahurili smo podatke kao što su `$brand` i `$model`. `$brand` možemo promijeniti ali samo korištenjem settera. Ako neko želi poslati prazan string mi možemo kontrolirati da npr. ne zapišemo prazan string. Setter ne vraća ništa (pišemo `void`). Zbog stroge tipizacije nakon takve definicije ne možemo vratiti ništa.

Nasljeđivanje

- Korištenje koda iz bazne klase u izvedenoj klasi.

```
// bazna klasa
class Vehicle {
    public $make;
    protected $engine;
    private $vin;

    public function __construct($make, $engine, $vin) {
        $this->make = $make;
        $this->engine = $engine;
        $this->vin = $vin;
    }
}

// izvedbena klasa
class Car extends Vehicle {
    public function getEngine() {
        return $this->engine; // Pristupačno zbog protected
    }
    // vin svojstvu moguće je pristupiti samo pomoću gettera koji nije definiran
}

$myCar = new Car("Toyota", "V6", "123ABC");
echo $myCar->getEngine();
```

Pogledajmo što je sa konstruktorima kada imamo nasljeđivanje. Napraviti ćemo primjer `nasljeđivanje.php` s magičnom metodom.

```
<?php
```

```

class Osoba {
    // svojstvima se može pristupiti unutar klase i klasama izvedenim iz te klase
    protected $ime;
    protected $prezime;

    public function __construct($ime, $prezime){
        $this->ime = $ime;
        $this->prezime = $prezime;
    }

    // ovo je magic method koji omogućava ispis stringa
    public function __toString(){
        return $this->ime . " " . $this->prezime;
    }
}

class Student extends Osoba {
    // po ovome se Student razlikuje od Osobe, dodano svojstvo
    // jedinstvenog matičnog broja akademskog građana
    private string $jmbag;

    // doajemo još jedna konstruktor na studentu koji ima $jmbag
    public function __construct($ime, $prezime, $jmbag){
        // ne ovako, radi jer je protected ako je acces modifier private, prestaje
        raditi
        // $this->ime = $ime;
        // $this->prezime = $prezime;
        // ovako dobijamo modularnost, ova linija mora biti uvijek prva
        parent::__construct($ime, $prezime);
        $this->jmbag = $jmbag;
    }

    // ovo je polimorfizam jer je isto ime ali druga funkcionalnost
    public function __toString()
    {
        return parent::__toString() . "-" . $this->jmbag;
    }
}

// objekt iz klase student mora imati parametre
$student = new Student("Marko", "Marković", "123456789");
var_dump($student);

```

```
echo $student;
```

Rezultat je:

```
object(Student)#1 (3) {
    ["ime":protected]=>
        string(5) "Marko"
    [" prezime":protected]=>
        string(9) "Marković"
    ["jmbag":"Student":private]=>
        string(9) "123456789"
}
Marko Marković-123456789
```

U klasi `Sudent` koja je naslijeđena od klase `Osoba` dodano je još jedno `private` string `$jmbag` svojstvo. U `__construct` vidimo `parent::__construct($ime, $prezime)`. `parent` služi za **referenciranje osnovne klase (superklase) unutar izvedene klase (subklase)**. U ovom primjeru, izvedena klasa naslijeđuje osnovnu klasu koja već ima definirani konstruktor.

Simbol `::` u PHP-u označava pristup statičkim metodama ili naslijeđenim metodama iz određene klase. Kada koristimo `parent::`, dvotočke služe za pozivanje metode iz osnovne klase.

Kada izvedena klasa naslijeđuje osnovnu klasu, može ponovno koristiti metode i atributе osnovne klase. Pozivanje `parent::__construct($ime, $prezime)` osigurava da se konstruktor osnovne klase izvrši, inicijalizirajući naslijeđene atribute. Ako ne pozovemo konstruktor osnovne klase, atribute \$ime i \$prezime neće biti inicijalizirani (osim ako ih ponovno ne postavimo u izvedenoj klasi). Ovo je posebno korisno kada osnovna klasa ima složeniju logiku unutar konstruktora.

Polimorfizam

Polimorfizam je način kako različite klase mogu implementirati iste metode na različite načine. Polimorfizam je jedan od ključnih principa objektno orientiranog programiranja (OOP) koji omogućava različitim klasama da implementiraju iste metode na različite načine. U PHP-u se polimorfizam postiže korištenjem nasljeđivanja i implementacije interfejsa.

Polimorfizam kroz nasljeđivanje

Kada klasa naslijeđuje drugu klasu, može redefinisati (engl. override) metode roditeljske klase. Evo primjera:

```
<?php

class Animal {
    public function makeSound() {
        echo "Animal sound";
    }
}
```

```
class Dog extends Animal {
    public function makeSound() {
        echo "Woof!";
    }
}

class Cat extends Animal {
    public function makeSound() {
        echo "Meow!";
    }
}

function printSound(Animal $animal) {
    $animal->makeSound();
    echo PHP_EOL;
}

$dog = new Dog();
$cat = new Cat();

printSound($dog); // Ispisat će: Woof!
printSound($cat); // Ispisat će: Meow!
?>
```

U ovom primjeru, klase `Dog` i `Cat` nasleđuju klasu `Animal` i redefiniraju metodu `makeSound`. Funkcija `printSound` prihvata instancu `Animal` klase i poziva metodu `makeSound`, bez obzira na to koja je konkretna instanca u pitanju.

Interface-i

Interface je obveza („ugovor“ ili predložak) koji definira koje metode neka klasa mora implementirati on sadrži samo deklaracije metoda, bez njihove implementacije. Kada klasa implementira interace, ona mora sadržavati sve metode koje su definirane u tom interface-u.

Preko interface-a možemo znati da li neka klasa ima ili nema metodu. Ako klasa implementira interface a taj interface je obaveza za klasu da je implementira, onda smo sigurni da je ta klasa posjeduje. Upravo tada, ti interface-i nam omogućavaju da kroz te „potpisne“ znamo što naša klasa posjeduje (naravno kada se instancira u objekt).

S druge strane, ono što je kod interface-a bitno je da ne moramo strogo definirati koji ćemo objekt primiti.

Ako pogledamo primjer, imamo definiran interface `Shape`. Morat će definirati metodu koja se zove `calcArea` za računanje površine. To znači da ako imamo neku klasu koja implementira taj interface, tada je ta klasa dužna kreirati tu metodu. U ovom primjeru klasa `Circle` implementira `Shape`. Taj interface kaže: „moraš implementirati metodu `getArea` za računanje površine“. `CalcArea` upravo površinu kruga.

Imamo i klasu `Rectangle` koja isto tako kaže implementirat ču interface `Shape`. Taj interface njega obavezuje da kreira metodu koja se zove isto ali drugačije radi. Ta `getArea` metoda računa površinu kao širina x visina. Upravo u tom svojstvu možemo vidjeti polimorfizam.

Interface obavezuje i klasu `Circle` i klasu `Rectangle` da implementiraju istu metodu `calcArea`. Metode rade drugačije za različite klase. Interface je obavezao i jednu i drugu klasu da metodu implementiraju.

Kod definiranja interface-a iz MVC, Microsoft ispred stavlja `I` da bi razlikovao interface od klase. To nam nije potrebno u PHP, jer kod njega potpuno drugačije radi nasljeđivanje.

U interface-u je dozvoljeno samo pisanje imena metode, scope (`private`, `protected` ili `public`) i definicija svih metoda koje inteface treba definirati. Predavač kaže da koristi samo `public` je interface koristimo samo za javno dostupne metode. Interface ne definira kako će se metoda ponašati, zato nema tijela metode. Također zato nema vitičastih zagrada. Ovdje je samo ono što klasa treba definirati, nazava, dakle samo nabrojane metode. Za definiciju metode koje inteface treba definirati iza modifikatora pišemo ključnu riječ `function`, ime funkcije, zatim dvotočku `:` i koji tip funkcija vraća (`float`, `int`, `string`, `void`...). Na kraju ide točka-zarez `;` i iza nje druge metode ako ih ima.

Interface-i se mogu nasljeđivati. Predavač kaže da izbjegava nasljeđivanje interface-a i da to nikada ne koristi. Kaže da je tada potrebno dobro raspisati kako se koji interface definira i tko nasljeđuje koga.

Često se korištenjem `extends` krši jedan od principa ([interface segregation](#)). Iz tog razloga predavač izbjegava nasljeđivanje interface-a.

```
<?php
interface Shape {
    public function getArea(): float;
}
interface Color extends Shape{
    public function getColor(): string;
}
```

Ako ne stavimo acces modifier `public` u metode interace-a, PHP svejedno smatra da su `public`, dakle to možemo izostaviti.

Ključna riječ `implements` u PHP-u služi za implementaciju interfejsa. Interfejs je vrsta apstraktnog tipa koji definira skup metoda i/ili svojstava koje klasa mora implementirati ako želi da implementira taj interface.

Koristeći `implements`, klasa preuzima odgovornost da implementira sve metode i svojstva definirane u interfejsu. Ovo pomaže u postizanju polimorfizma i interoperabilnosti između različitih klasa.

Ključne prednosti korištenja `implements` su:

Postizanje polimorfizma: Objekti različitih klasa koje implementiraju isti interfejs mogu se tretirati na isti način.

Modularnost: Interfejsi omogućavaju definiranje ugovora između klasa (ponašanje i stanje koje klasa pruža, engl. contract), što olakšava modularno programiranje.

Testiranje: Interfejsi pomažu u testiranju klase jer se mogu lako zamijeniti mock objekti (simulirani objekti koji se koriste u testiranju softvera) koji implementiraju isti interfejs.

Korišćenje `implements` je ključno za postizanje fleksibilnosti, ponovne upotrebe koda i održavanja u objektno-orientiranom programiranju u PHP-u.

PHP podržava interfejsе, koji omogućavaju definiranje skupine metoda koje klasa mora implementirati. Ovo omogućava različitim klasama da implementiraju iste metode na različite načine. To je **polimorfizam kroz interface**.

```
<?php

// interface je obaveza za klase da implementiraju određene metode
// dozvoljeno je samo pisanje imena metode, uglavnom public
// interface ne definira kako će se metoda ponašati, nema vitičastih zagrada
// ovdje je samo ono što klasa treba definirati, nabrojane metode
interface Shape {
    public function getArea(): float;
}

class Circle implements Shape {
    private float $radius;

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

    public function getArea(): float {
        // računanje površine kruga
        return M_PI * $this->radius ** 2;
    }
}

class Rectangle implements Shape {
    private float $width;
    private float $height;

    public function __construct(float $width, float $height) {
        $this->width = $width;
        $this->height = $height;
    }

    public function getArea(): float {
        return $this->width * $this->height;
    }
}
```

```

function printArea(Shape $shape) {
    echo 'Površina: ' . $shape->getArea() . PHP_EOL;
}

// circle je nastao iz klase Circle ali je u stvari Shape
$circle = new Circle(5);
$rectangle = new Rectangle(3, 4);

printArea($circle);
printArea($rectangle);

```

Klase `Circle` implementira `Shape`. Red `public function getArea(): float` znači da u klasi `Circle` moramo imati implementiranu `getArea` metodu. Interface `Shape` obavezuje klasu `Circle` da implementira metodu `getArea`.

Otvaramo klasu `Rectange` koja implementira `Shape`. Konstruktor ovaj je primjer polimorfizma. U njoj moramo imati metodu `getArea` jer će dojaviti grešku. Interface omogućuje da možemo komunicirati prema van.

Napisali smo funkciju `printArea`. Ta funkcija ne prima `Circle`, ni `Rectangle` nego `Shape` koji je interface za komunikaciju. Ako u funkciji kažemo da ćemo poslati objekt koji ima interface `Shape` onda smo sigurni da na tom objektu postoji metoda `getArea`. Sigurni smo zato što je interface obavezao tu klasu da implementira. To je prednost interface-a.

U ovom primjeru, `Circle` i `Rectangle` klase implementiraju `Shape` interface, koji zahtijeva implementaciju metode `getArea`. Funkcija `printArea` prihvata instancu `Shape` interfejsa i poziva metodu `getArea`, bez obzira na konkretnu implementaciju.

Kada ne bi radili s interface-om, red `class Circle implements Shape` glasio bi `class Circle`. Moramo napisati ponovo funkciju `printArea` koja će printati i `Circle` i `Rectangle`. Morali bi napisati dvije nove funkcije. To nije dobra praksa jer dupliciramo kod i postoje sve složeniji.

Apstraktne klase

- **Definicija:**
 - o Klase koje ne mogu biti instancirane (dakle ne možete stvoriti instancu apstraktne klase, dizajnirane su da budu nasleđene) i koriste se kao temelj za druge klase.
- **Svrha:**
 - o Definirati temeljnu skicu za niz izvedenih klasa, prisiljavajući izvedene klase da implementiraju određene metode.

Glavne karakteristike apstraktnih klasa u PHP-u su:

- Deklariraju se ključnom riječi `abstract` ispred `class`
- Mogu sadržavati apstraktne metode bez implementacije, označene ključnom riječi `abstract` ispred `function`. Te metode su deklarirane ali bez implementacije.

- Podklase koje nasljeđuju apstraktnu klasu moraju implementirati sve njene apstraktne metode ili biti označene kao apstraktne metode
- Mogu sadržavati i konkretne (ne-apstraktne) metode i svojstva. Apstraktne klase mogu sadržavati metode sa implementacijom, koje podklase mogu koristiti ili predefinirati
- Konstruktori apstraktnih klasa se mogu pozvati prilikom instanciranja izvedenih klasa

Možemo izbjegići prisiljavanje izvedene klase da implementiraju određene metode - možemo ih pomiješati s interface-om. Apstraktna klasa može isto tako biti obvezujuća za svojeg nasljednika. Npr. u primjeru ispod klasa `makeSound()` nema nikakvu funkcionalnost, to je samo definicija. Vidi se ključna riječ `abstract`. To je metoda koja treba biti implementirana od strane klase koja će naslijediti ovu apstraktну klasu. Ostalo će moći naslijediti klasu.

Vidimo razliku od interface-a gdje su strogo definirane metode koje klasa treba definirati. Apstraktna klasa ima istu ulogu kao i interface da obvezuje ali te klase ne implementiraju apstraktну klasu nego je nasljeđuju. A kada je nasljeđuju tu apstraktну onda ih tjera da implementiraju metodu `makeSound()`.

Međutim, apstraktna klasa daje i dodatne funkcionalnosti, kao nekakav temelj koji će moći koristi svaka klasa koja implementira. **Dobra praksa je da apstraktna klasa ne implementira obvezujuće metode za nasljeđujuće klase**, nego da to sve radimo na interface-ima. Predavač apstraktnu klasu isključivo koristi samo kao temelj za neke funkcionalnosti kojima ostale klase to mogu naslijediti. Sama apstraktna klasa nema potrebe da ikada postane objekt. Dakle uvijek možemo odlučiti hoćemo li koristiti apstraktnu klasu za temelj ili ćemo je koristi za temelj i za to da natjeramo child klase da implementiraju neku metodu.

Lakše je koristi `interface`. Možemo reći `class Dog extends Animal implement neki_interface`. Onda `interface` određuje što ta klasa mora implementirati od metoda, a ne apstraktna klasa. Predavač ne koristi apstraktnu klasu za prisiljavanje naslijedene klase da nešto mora implementirati, nego samo za definiranje osnovnih funkcionalnosti.

```
abstract class Animal {
    protected $name;

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

    abstract public function makeSound();

    public function getName() {
        return $this->name;
    }
}

class Dog extends Animal {
    public function makeSound() {
        return "Bark";
    }
}
```

```

}

$myDog = new Dog("Rover");
echo $myDog->getName() . " says " . $myDog->makeSound();

```

Pogledajmo primjer kako ona radi. U primjeru ćemo onemogućiti da apstraktnu klasu samo instanciramo u objekt. To recimo možemo kod primjera iz nasljeđivanja, [nasljeđivanje.php](#). Mogli smo reći [new Osoba](#), pošaljemo ime i prezime i dobit ćemo objekt.

Kopirat ćemo klasu [Student](#) i klasu Osoba iz [nasljeđivanje.php](#) u [abstraktna_klasa.php](#).

```

<?php

class Osoba {
    // svojstvima se može pristupiti unutar klase i klasama izvedenim iz te klase
    protected $ime;
    protected $prezime;

    public function __construct($ime, $prezime){
        $this->ime = $ime;
        $this->prezime = $prezime;
    }

    // ovo je magic method koji omogućava ispis stringa
    public function __toString(){
        return $this->ime . " " . $this->prezime;
    }
}

class Student extends Osoba {
    // po ovome se Student razlikuje od Osobe, dodano svojstvo
    // jedinstvenog matičnog broja akademskog građana
    private string $jmbag;

    // doajemo još jedna konstruktor na studentu koji ima $jmbag
    public function __construct($ime, $prezime, $jmbag){
        // ne ovako, radi jer je protected ako je acces modifier private, prestaje
        // raditi
        // $this->ime = $ime;
        // $this->prezime = $prezime;
        // ovako dobijamo modularnost, ova linija mora biti uvijek prva
        parent::__construct($ime, $prezime);
        $this->jmbag = $jmbag;
    }
}

```

```
// ovo je polimorfizam jer je isto ime ali druga funkcionalnost
public function __toString()
{
    return parent::__toString() . "-" . $this->jmbag;
}
```

Dakle sada možemo reći:

```
$osoba = new Osoba("Marko", "Marković");
var_dump($osoba);
echo $osoba;
```

i sve radi. Dobijemo odgovor;

```
object(Osoba)#1 (2) {
    ["ime":protected]=>
    string(5) "Marko"
    ["prezime":protected]=>
    string(9) "Marković"
}
Marko Marković
```

Sada ćemo na klasi osoba ispred dodati ključnu riječ **abstract**. Red sada glasi **abstract class Osoba {**. Klasu Osoba više nije moguće instancirati jer **abstract** ne dozvoljava distanciranje objekta.

Sada program dojavljuje:

```
Fatal error: Uncaught Error: Cannot instantiate abstract class Osoba in ...
```

Visual Studio Code tj. Intellisense neće dojaviti ništa, grešku ćemo vidjeti tek prilikom pokretanja koda.

Apstraktna klasa treba biti naslijedena sa **extends** jer bez toga apstraktna klasa nema smisla jer ničemu ne služi. Apstraktnu klasu ne možemo istancirati u objekt. Moguće je samo da klasu koja nasljeđuje apstraktnu klasu primi sve funkcionalnosti iz nje. To je jedino moguće i to je bit apstraktne klase. U ovom primjeru ta apstraktna klasa **Osoba**. To je samo temelj, osnova kako bi druge klase to mogle koristiti da nemamo redundanciju. Npr. ime i prezime u klasi **Student**, **Vanjski_suradnik**, **Radnik**. Redundanciju kreiramo tako da kreiramo roditeljsku klasu. Ako ta roditeljska klasa nema smisla da postane objekt u našoj aplikaciji, proglašit ćemo je za apstraktnu klasu.

Apstraktna klasa se može naslijediti od neke druge klase sa **abstract class Osoba extends Test{**.

Idemo napisati tu klasu **test** iznad apstraktne klase **Osoba**.

```
<?php
class Test{
    protected string $test;
```

```
}
```

Ako na dnu koda imamo definiranu varijablu osobu kao instancu klase Student

```
$osoba = new Student("Marko", "Marković", "123456789");  
var_dump($osoba);
```

Upravo će nam `var_dump` prikazati:

```
object(Student)#1 (3) {  
    ["test":protected]=>  
    uninitialized(string)  
    ["ime":protected]=>  
    string(5) "Marko"  
    [" prezime":protected]=>  
    string(9) "Marković"  
    [" jmbag": "Student":private]=>  
    string(9) "123456789"  
}
```

Vidimo test ali nije postavljen. Dakle apstraktna klasa može naslijediti neku klasu. Iz apstraktne klase sve kreće pa je malo čudno da apstraktna klasa nasljeđuje ali čak apstraktna klasa može naslijediti i drugu apstraktну klasu. Predavač misli da to nije dobra praksa i da ne treba koristiti nasljeđivanje iz drugih klasa jer je apstraktna klasa temelj iz kojeg krećemo.

Poanta je da apstraktnu klasu razlikujemo od interface-a na način da apstraktna klasa uz definicije koje naslijeđena klasa (engl. child class, podklasa) treba implementirati nudi konkretne implementacije (možemo koristiti `__construct`, property metode i druge konkretne stvari kao i u običnim klasama). U interface-ima toga nema. Interface je sama definicija nečeg što želimo implementirati.

Ako želimo natjerati klasu Student (i sve druge klase) da implementiraju neku metodu onda možemo dati definiciju u apstraktnoj klasi sa npr. `abstract public function pozdrav();`

Većina neće to koristiti. Predavač radije koristi `interface`. zbog SOLID principa. To je skup 5 smjernica za dizajn čiji je cilj napraviti dizajn razumljivim, fleksibilnim i lakšim za održavanje. Ovi principi se široko koriste u objektno orientiranom programiranju, iako se također mogu primijeniti u drugim paradigmama. Solid označava:

- Načelo jedinstvene odgovornosti (Single Responsibility Principle- SRP)
- Princip otvoreno-zatvoreno (Open-Closed Principle - OCP)
- Liskovljev princip supstitucije (Liskov Substitution Principle - LSP)
- Načelo razdvajanja interface-a (Interface Segregation Principle - ISP)
- Načelo inverzije ovisnosti (Dependency Inversion Principle - DIP)

Klasa bi trebala imati samo jedan razlog ili svrhu kojoj služi. To znači da razred treba imati samo jednu odgovornost.

Instance bi trebale biti otvorene za proširenje, ali zatvorene za izmjene. To znači da se instance (klase, funkcije, moduli ili itd.) mogu proširiti drugom instancom, ali ne smiju dopustiti njihovu direktnu izmjenu.

To znači da bi objekti roditeljske klase (nadklase, superklase) trebali biti zamjenjivi s objektima naslijedene klase (engl. child class, podklase) bez utjecaja na ispravnost programa.

Ne bismo trebali prisiljavati instance da implementiraju interface koja ne koriste. Interface bismo trebali učiniti fleksibilnim za većinu slučajeva.

Ovo načelo ima za cilj razdvojiti module visokog nivoa (koji pružaju složenu funkcionalnost) od modula niskog nivoa (koji pružaju osnovnu funkcionalnost) uvođenjem sloja apstrakcije između njih. Ovo omogućuje modulima visokog i niskog nivoa da ovise o apstrakcijama, a ne o čvrstim implementacijama, čineći sistem fleksibilnijim, proširivim i održavanim.

<https://dev.to/devlinaung/solid-principles-in-php-363>

Razlika između korištenja apstraktnih klasa i interface-a

Predavač kaže da je ovo ponekad pitanje na razgovorima za posao. Glavne razlike između korištenja apstraktnih klasa i interfejsa u PHP-u su:

1. Apstraktne klase mogu sadržavati apstraktne i konkretnе metode, dok interface-i mogu imati samo apstraktne metode.
2. Apstraktne klase ne podržavaju višestruko nasljeđivanje, dok interfejsi to podržavaju.
3. Apstraktne klase mogu imati svojstva (variable) koja mogu biti `final`, `non-final`, `static` i `non-static`, dok interfejsi mogu imati samo `static` i `final` svojstva.
4. Apstraktne klase mogu imati `static` metode, `main` metodu i konstruktore, dok interfejsi ne mogu.
5. Apstraktna klasa može pružiti implementaciju interfejsa, dok interfejs ne može pružiti implementaciju apstraktne klase.
6. Apstraktne klase se deklariraju ključnom riječi `abstract class`, dok se interfejsi deklariraju ključnom riječi `interface`.
7. Apstraktne klase postižu djelimičnu apstrakciju (0-100%), dok interfejsi postižu potpunu apstrakciju (100%).

Ukratko, apstraktne klase su fleksibilnije jer mogu imati konkretnе metode i svojstva, ali ne podržavaju višestruko nasljeđivanje. S druge strane, interfejsi su rigidniji, ali podržavaju višestruko implementiranje i postižu potpunu apstrakciju.

Statičke Metode

- Definicija:
 - o Metode koje se mogu pozivati na razini klase, a ne na razini instance.
- Svrha:
 - o Omogućavanje pristupa funkcijama koje ne zahtijevaju podatke o specifičnoj instanci klase.

Kada želite pozvati metodu koja je definirana u klasi mora biti stvoren objekt iz te klase i tek onda na nivou te instance objekta pozvati tu metodu. Međutim, moguće je definirati metode na nivou klase

koje je moguće i pozivati na nivou klase bez da moramo stvoriti objekt. Čak i ako kreiramo objekt ta metoda neće biti dostupna u objektu. Statičke metode dostupne su samo na nivou objekta. pišu se sa ključnom riječi `static`.

To nam je potrebno za utility-je, helper klase, itd. Poanta je da ne moramo istancirati objekt nego pozovemo klasu i nad tom klasom metodu. Kako bi definirali određene patterne, potrebno je koristiti `static` metode.

```
class MathHelper {  
    public static function add($a, $b) {  
        return $a + $b;  
    }  
  
    echo MathHelper::add(5, 3);
```

PHP ima nit (thread) što drugi programski jezici nemaju. Moguće je dijeliti (engl. share) funkcionalnosti bez da morate instancirati objekte.

Za primjer predavač je definirao klasu `Utils` koja u sebi ima metodu `generateRandomString` generator slučajnih stringova, koja je definirana kao `public static`. Dužina stringa je ulazni parametar. Svaki puta kada pozovemo tu metodu, rezultat će biti različit. Metodu pozivamo na sljedeći način:

```
echo Utils::generateRandomString(10) . PHP_EOL;
```

Na globalnom nivou ne koriste se ručno napisani identifikator već `uuid` ili `guid` identifikator koji ima 16 bitnu strukturu i jedinstvena je identifikacija. Svrha ovoga nije kriptografija. Mogu se u radu koristiti i generatori koji idu od 1. Primjenjujemo ono što nam je potrebno.

Ako distanciramo klasu, nećemo imati pristup statičkoj metodi:

```
$utils = new Utils();
```

U ovom objektu metoda je nevidljiva. Nije dobra praksa imati u klasi kombinaciju statičkih metoda i običnih metoda. Zamislimo da imamo unutar klase `Utils` neku `public` funkciju. Napraviti ćemo `generateId` funkciju:

```
public function generateId(): string{  
    return Utils:: generateRandomString(10);  
}
```

Možemo je instancirati i pozvati:

```
$utils = new Utils();  
echo $utils->generateId() . PHP_EOL;
```

Objekt će vidjeti `generateId`. On poziva klasu i nad tom klasom nemamo `$this->`. U `Utils` objektu ne postoji ta metoda, što može zbuniti. Ako pokrenemo to funkcionira ali nije praksa. Također ako unutar funkcije `generateId` probamo koristiti `$this` na sljedeći način:

```
public function generateId(): string{
    return $this->generateRandomString(10);
}
```

Objektu je moguće pristupiti static metodi.

Ako unutar klase `Utils` dodamo `static` svojstvo `$test` sa `public static $test;` onda kada to svojstvo probamo pozvati van klase sa npr. `var_dump($utils->test);` `static` svojstvu neće biti moguće pristupiti i PHP će javiti grešku.

Ispred `class Utils` moguće je staviti ključnu riječ `abstract` i time smo zabranili distanciranje klase, što je dosta zbumujuće ali jedini način u PHP da se zabrani distanciranje klase u objekt i na taj način izvlačenje preko objekta. Ako to želimo, neće raditi distanciranje klase ali će i dalje raditi:

```
echo Utils::generateRandomString(10) . PHP_EOL;
```

Iskopirat ćemo `generateRandomNumber` funkciju u novu funkciju i napisati da je `public static function`:

```
public static function generirajId(int $length): string{
    return $this->generateRandomString(10);
}
```

Na dnu ćemo napisati:

```
$alati = new Utils();
echo $alati->generirajId(10) . PHP_EOL;
```

Ovo neće raditi jer smo unutar `static` metode probali koristiti `$this`. Rješenje je koristiti `self`:

```
public static function generirajId(int $length): string{
    return self::generateRandomString(10);
}
```

Ovako sve radi.

Kada radimo sa `static` metodama, sve se mogu koristiti sa `self` umjesto sa `$this`. Ne treba kombinirati različite metode. Smisao `static` je da on ne bi trebao živjeti na instanci.

`self i parent` ključne riječi za pristup statičkim članovima unutar klase

U PHP-u, ključne riječi `self` i `parent` se koriste za pristup statičkim članovima unutar klase na sljedeći način:

self

Ključna riječ **self** se koristi za referenciranje tekuće klase. Omogućava pristup statičkim metodama i varijablama unutar iste klase.

```
class NazivKlase {  
    public static $statičkaVarijabla = "vrijednost";  
  
    public static function statičkaMetoda() {  
        echo self::$statičkaVarijabla;  
    }  
}  
  
NazivKlase::statičkaMetoda(); // Ispisuje: vrijednost
```

U ovom primjeru, **self** se koristi za pristup statičkoj varijabli **\$statičkaVarijabla** i pozivu statičke metode **statičkaMetoda()** unutar iste klase **NazivKlase**.

parent

Ključna riječ **parent** se koristi za referenciranje roditelja trenutne klase. Omogućava pristup statičkim metodama i varijablama definiranim u roditeljskoj klasi.

```
class RoditeljskaKlase {  
    public static function statičkaMetoda() {  
        echo "Metoda iz roditeljske klase";  
    }  
}  
  
class NaslijedenaKlasa extends RoditeljskaKlase {  
    public static function poziviRoditelja() {  
        parent::statičkaMetoda();  
    }  
}  
  
NaslijedenaKlasa::poziviRoditelja(); // Ispis: Metoda iz roditeljske klase
```

U ovom primjeru, **NaslijedenaKlasa** nasljeđuje se od **RoditeljskaKlasa**. Metoda **poziviRoditelja()** koristi **parent** za pozivanje statičke metode **statičkaMetoda()** definiranu u roditeljskoj klasi **RoditeljskaKlasa**.

Ključne razlike:

- **self** se koristi za referenciranje trenutne klase
- **parent** se koristi za referenciranje roditeljske klase
- Obje ključne riječi se koriste za pristup statičkim članovima unutar klase

Pravilna upotreba **self** i **parent** ključnih riječi je važna za ispravno referenciranje statičkih elemenata u hijerarhiji klasa.

Operator za rješavanje opsega (engl. Scope Resolution Operator)

U PHP-u, operator `::` se naziva **Scope Resolution Operator** ili **Operator za razrješavanje opsega**.

Ovaj operator služi za:

1. Pristup statičnim članovima klase:
 - Omogućava pristup statičnim metodama i varijablama klase.
 - Primjer: `NazivKlase::statičnaMetoda()` ili `NazivKlase::$statičnaVarijabla`
2. Pristup konstantama klase:
 - Omogućava pristup konstantama definiranim unutar klase.
 - Primjer: `NazivKlase::KONSTANTA`
3. Pozivanje statičnih metoda roditelja:
 - Omogućava pozivanje statičnih metoda roditelja klase.
 - Primjer: `RoditeljskaKlase::statičnaMetoda()`
4. Pristup statičnim članovima unutar klase:
 - Omogućava pristup statičnim članovima iste klase.
 - Primjer: `self::$statičnaVarijabla` ili `self::statičnaMetoda()`
5. Pristup statičnim članovima roditelja unutar klase:
 - Omogućava pristup statičnim članovima roditelja klase.
 - Primjer: `parent::statičnaMetoda()`

Operator `::` je ključan za pristup statičnim članovima klasa u objektno orientiranom programiranju u PHP-u. Koristi se za referenciranje elemenata unutar opsega klase, bez potrebe za kreiranjem instance objekta.

Ne zaboravite na mogućnost korištenja `self` i `parent` ključnih riječi za pristup statičnim članovima unutar klase.

Kako pozvati nadjačanu statičku metodu iz osnovne klase ako je tu metodu nadjačala izvedena klasa

Pretpostavimo da kreirate dvije statičke metode u klasi `OsnovnaKlase`.

Pozvali ste prvu statičku metodu `getStaticText()` iz druge statičke metode `getStaticRezultat()`.

Sada podređena klasa `IzvedenaKlase` izvedena iz `OsnovnaKlase` i nadjačava statičku metodu `getStaticText()`.

```
<?php
Class OsnovnaKlase
{
    // Statička funkcija, čim je static nema instance
    public static function getStaticText()
    {
        return 'Pozvano iz OsnovnaKlase';
    }

    public static function getStaticRezultat()
```

```

    {
        // alternativno možemo koristiti OsnovnaKlasa::getStaticText();
        // sve static metode koriste self umjesto $this
        return self::getStaticText();
    }
}

Class IzvedenaKlasa extends OsnovnaKlasa
{
    public static function getStaticText()
    {
        return 'Pozvano iz IzvedenaKlasa';
    }
}

var_dump(IzvedenaKlasa::getStaticRezultat());

?>

```

Sada ako pozovete `getStaticRezultat()` iz `IzvedenaKlasa` – tada će rezultat `var_dump(IzvedenaKlasa::getStaticRezultat())`; biti: `Pozvano iz OsnovnaKlasa`.

Funkcija `getStaticText()` pozvana je iz `OsnovnaKlasa` jer se poziva ključnom riječi `self`.

Slučaj :

Nadjačali ste metodu `getStaticText()` u `IzvedenaKlasa`.

Također ste pozvali metodu iz `IzvedenaKlasa` –
`var_dump(IzvedenaKlasa::getStaticRezultat());`

Ovdje prepostavimo da želite, ako je bilo koja metoda nadjačana u podređenoj klasi, da je tada morate pozvati iz podređene klase. Ako je klasa koju pozivamo podređena klasa (`IzvedenaKlasa`).

- Morate pozvati `IzvedenaKlasa::getStaticRezultat()` umjesto `self::getStaticText();` iz `OsnovnaKlasa`. Ali kako netko zna dok kreira `OsnovnaKlasa` iz koje će se klase kreirati `OsnovnaKlasa`.

Kako onda to postići generički?

Ovdje dolazi poziv statičke metode s ključnom riječi `static`.

Ako koristite ključnu riječ `static` za pozivanje statičkih funkcija u klasi, tada –

- U slučaju da nema nadjačane funkcije, tada će pozvati funkciju unutar klase kao što to čini ključna riječ `self`.
- Ako je funkcija nadjačana, tada će statička ključna riječ pozvati nadjačanu funkciju u izvedenoj klasi.

U donjem kodu koristili smo `static::getStaticText()` umjesto `self::getStaticText()` metodi `OsnovnaKlase`.

```
<?php

Class OsnovnaKlase
{
    // Statička funkcija, čim je static nema instance
    public static function getStaticText()
    {
        return 'Pozvano iz OsnovnaKlasa';
    }

    public static function getStaticRezultat()
    {
        // alternativno možemo koristiti OsnovnaKlase::getStaticText();
        return static::getStaticText();
    }
}

Class IzvedenaKlasa extends OsnovnaKlase
{
    public static function getStaticText()
    {
        return 'Pozvano iz IzvedenaKlasa';
    }
}

var_dump(IzvedenaKlasa::getStaticRezultat());

?>
```

Sada će rezultat `var_dump(IzvedenaKlasa::getStaticRezultat());` biti: `Pozvano iz IzvedenaKlasa`.

Dakle ovaj puta je pozvana nadjačana funkcija `getStaticText()`. Ovako radi poziv statičke funkcije sa statičkom funkcijom.

`self` i `$parent` ključne riječi za pristup članovima klase

U PHP-u, `self` i `$this` su ključne riječi koje se koriste za pristup članovima klase, ali imaju različite svrhe i ponašanja:

`self:`

- `self` je statička ključna riječ koja se koristi za referenciranje trenutne klase.
- Omogućava pristup statičkim metodama i varijablama definiranim unutar iste klase.
- Primjer: `self::$statičkaVarijabla` ili `self::statičkaMetoda()`.

`$this:`

`$this` je ključna riječ koja se koristi za referenciranje trenutnog objekta instance klase.

Omogućava pristup instancama metoda i varijabli definiranim unutar iste klase.

Primjer: `$this->instanciranaVarijabla` ili `$this->instanciranaMetoda()`.

Ključne razlike:

`self` se koristi za pristup statičkim članovima klase, dok se `$this` koristi za pristup instancnim članovima objekta.

`self` ne zahtijeva instancu objekta, dok `$this` mora biti pozvan unutar metode instance objekta.

`self` se koristi za referenciranje same klase, dok se `$this` koristi za referenciranje trenutne instance objekta.

```
class NazivKlase {  
    public static $statičkaVarijabla = "vrijednost";  
    public $instancnaVarijabla = "vrijednost";  
  
    public static function statičkaMetoda() {  
        echo self::$statičkaVarijabla; // Pristup statičkoj varijabli  
    }  
  
    public function instanciranaMetoda() {  
        echo $this->instanciranaVarijabla; // Pristup instanciranoj varijabli  
    }  
}  
  
// Pozivanje statičke metode  
NazivKlase::statičkaMetoda();  
  
// Kreiranje instance i pozivanje instancirane metode  
$objekt = new NazivKlase();  
$objekt->instanciranaMetoda();
```

U primjeru, `self` se koristi za pristup statičkim članovima klase, dok se `$this` koristi za pristup instanciranim članovima objekta.

Final klase

U PHP-u, klasa ne može biti deklarisana kao `private`. Klase u PHP-u mogu imati samo jednu od sljedećih vidljivosti:

Public (javna) - podrazumijevano stanje kada se vidljivost ne specificira.

Abstract (apstraktna) - koristi se za klase koje ne mogu biti instancirane direktno i služe kao baze za druge klase.

Final (konačna) - koristi se za klase koje ne mogu biti naslijedene.

```
final class MyFinalClass {
    public function method() {
        echo "This is a method in a final class.";
    }
}

// Ova klasa će prouzrokovati grešku
class ChildClass extends MyFinalClass {
    // Nije dozvoljeno
}
```

Enumeratori

Prije verzije PHP 8 nisu postojali. Služe za definiranje slučajeva koji su definirani, kada želite ograničiti unos na unaprijed definirane vrijednosti.

```
enum Gender{
    case MA;
    case FE;
    case OT;
}
```

```
class User{
    private sting $name;
    private Gender $gender;

    public function __construct(string $name, Gender $gender){
        $this->name = $name;
        $this->gender = $gender;
    }

    // getter, bez njega ne radi, ovo je rješenje sa magic metodom __get
    // on se aktivira svaki puta kada pokušamo dohvatiti privatno svojstvo (čak i
public)
```

```
public function __get($name){
    return $this->$name;
}
```

```
$user = new User("Marko", Gender::MA);
var_dump($user->gender);
```

Bez getter funkcije ovo ne može raditi. Ovdje je napravljeno rješenje s magic metodom `__get` koji će se aktivirati svaki put kada pokušamo dohvatiti privatno svojstvo. Ova magic metoda će se aktivirati u `$name` i dobit ćemo property koji želimo dohvatiti. Zatim će `__get` metoda vratiti s return `$this->gender`. Magic metoda `__get` radi nad svim property-ima i na svakom idućem property-ju. Ovo je odlično ako želimo dati pristup svim svojim privatnim svojstvima u klasi. Često to nije slučaj pa moramo pisati ručno setter-e i getter-e. Kao rezultat dobijemo:

```
enum(Gender::MA)
```

Ako probamo ispisati s `echo $user->gender;` dobit ćemo grešku jer enumerator koji se smatra klasa se ne može konvertirati u string. Ako probamo ispisati vrijednost `$user->gender->value;` opet ćemo dobiti grešku jer nema vrijednosti. A nema je nemamo definirane vrijednosti u enum-u, pokraj `case`. PHP nudi Backed enumeratore. On i mogu biti ili string ili int.

```
enum Gender: string{
    case MA = "Male";
    case FE = "Female";
    case OT = "Other";
}
```

Više nema greške, dobivamo `Male`.

Imenski prostori (engl. Namespaces)

Namespaces u PHP-u služe za organizaciju koda u logične grupe i za izbjegavanje konflikata imena, što je posebno korisno u velikim projektima ili prilikom korištenja koda iz različitih biblioteka.

Što je namespace?

Namespace (prostor imena) je način grupiranja srodnih klasa, funkcija i konstanti u zasebne logičke cjeline.

Korištenje namespace-a omogućava da dvije klase ili funkcije s istim imenom koegzistiraju bez konflikta.

Kako definirati namespace?

Namespace se definira na vrhu PHP datoteke pomoću ključne riječi namespace, nakon čega slijedi ime prostora imena.

```
<?php  
namespace MojaAplikacija\Kontroleri;  
  
class KorisnikKontroler {  
    // Kod za kontroler  
}
```

Kako koristiti namespace?

Da biste koristili klase, funkcije ili konstante definirane u nekom namespace-u, možete koristiti:

Potpuno kvalificirano ime

```
$kontroler = new \MojaAplikacija\Kontroleri\KorisnikKontroler();
```

Korištenje use izjave

```
use MojaAplikacija\Kontroleri\KorisnikKontroler;
```

```
$kontroler = new KorisnikKontroler();
```

Logika i vidljivost namespace-a

Globalni prostor imena

- Kod koji nije unutar nijednog namespace bloka je u globalnom prostoru imena.
- Pristup klasama i funkcijama iz globalnog prostora imena unutar namespace-a mora biti kvalificiran s \.

Unutar namespace-a

- Sve što je definirano unutar određenog namespace-a vidljivo je unutar tog namespace-a.
- Elementi iz drugih namespace-ova nisu automatski vidljivi, osim ako nisu eksplicitno uključeni pomoću use izjave ili koristeći potpuno kvalificirana imena.

Primjer logike namespace-a

Prepostavimo da imamo dvije klase s istim imenom, ali u različitim namespace-ovima:

```
// File: src/Proizvodi/Knjiga.php  
namespace Proizvodi;  
  
class Knjiga {  
    public function getInfo() {  
        return "Ovo je knjiga iz Proizvoda.";  
    }  
}
```

```
// File: src/Biblioteka/Knjiga.php
namespace Biblioteka;

class Knjiga {
    public function getInfo() {
        return "Ovo je knjiga iz Biblioteke.";
    }
}
```

```
// File: index.php
require 'src/Proizvodi/Knjiga.php';
require 'src/Biblioteka/Knjiga.php';

use Proizvodi\Knjiga as ProizvodiKnjiga;
use Biblioteka\Knjiga as BibliotekaKnjiga;

$proizvod = new ProizvodiKnjiga();
echo $proizvod->getInfo(); // Ispisuje: Ovo je knjiga iz Proizvoda.

$biblioteka = new BibliotekaKnjiga();
echo $biblioteka->getInfo(); // Ispisuje: Ovo je knjiga iz Biblioteke.
```

Dvije klase s istim imenom `Knjiga` postoje u različitim namespace-ovima (`Proizvodi` i `Biblioteka`). Koristeći `use` i alias (`as`), možemo jednoznačno razlikovati i koristiti obje klase u istom dokumentu.

Prednosti korištenja namespace-a

- Izbjegavanje konflikata imena:** Mogućnost korištenja istih imena za različite klase, funkcije i konstante u različitim dijelovima aplikacije ili u različitim bibliotekama.
- Bolja organizacija koda:** Grupiranje srodnih klasa i funkcija u logičke cjeline.
- Lakša integracija biblioteka:** Kada koristite više biblioteka koje mogu imati klase i funkcije s istim imenima, namespace-i omogućuju njihovo koegzistiranje bez konflikt-a.

Zadatak: upravljanje događajima u PHP

Firma „EventPro“ želi razviti aplikaciju koja će olakšati organizaciju i upravljanje događajima.

Sistem treba omogućiti unos, ažuriranje, brisanje i ispis informacija o događajima i sudionicicima.

Funkcionalnosti:

Događaji: Svaki događaj ima jedinstveni identifikator, naziv, datum održavanja, vrstu događaja (konferencija, koncert, izložba), i status (planiran, potvrđen, otkazan). Događaji se mogu filtrirati po statusu i vrsti, te sortirati po datumu.

Sudionici: Svaki sudionik ima jedinstveni identifikator, ime, prezime i status sudjelovanja (prihvaćeno, odbijeno, na čekanju). Sudionici mogu biti dodani ili uklonjeni iz događaja, i moguće je ispisati listu sudionika za određeni događaj sortiranu po prezimenu.

Organizatori: Organizatori su zaduženi za upravljanje događajima. Svaki organizator ima jedinstveni identifikator, ime, prezime i listu događaja koje vodi. Organizatori mogu potvrditi ili otkazati događaje, a sustav treba automatski ažurirati status događaja.

Logika potpisivanja: Postoji viši menadžment (npr. direktor) koji može finalizirati detalje događaja. Direktor ima sposobnost da potvrdi sve aspekte događaja, uključujući datum i vrstu. Potpisivanje se može odnositi na događaje koje organizator predloži.

Implementacija:

- Kreiranje klase `Event`, `Participant`, `Organizer`, i `Director` sa relevantnim svojstvima i metodama.
- Implementacija interface-a za sortiranje, filtriranje i odobravanje.
- Demonstracija funkcionalnosti kroz hardkodirane primjere u glavnom programu, uključujući:
 - o Kreiranje i ispis nekoliko događaja.
 - o Dodavanje sudionika u događaj i njihov ispis.
 - o Ažuriranje statusa događaja od strane organizatora i direktora.
 - o Ispis svih događaja koje organizator vodi.

Prvo ćemo napraviti enumeratore:

```
<?php  
// enumerator EventType zadan po vrsti  
enum EventType: string {  
    case Conference = "Konferencija";  
    case Seminar = "Seminar";  
    case Workshop = "Radionica";  
}  
  
// enumerator EventStatus zadan po statusu  
enum EventStatus: string {  
    case Planned = "Planiran";
```

```

        case Confirmed = "Potvrđen";
        case Canceled = "Otkazan";
    }
// enumerator ParticipantStatus zadan po statusu sudjelovanja
enum ParticipantStatus: string {
    case Accepted = "Prihvaćen";
    case Declined = "Odbijen";
    case Pending = "Na čekanju";
}

```

Svaki sudionik i organizator ima ime i prezime. I sudionici i organizatori imaju jedinstveni identifikator. Nisu zajednički jer su jedinstveni. Ovdje ćemo iskoristiti nasljeđivanje. Kreirat ćemo apstraktnu klasu. Apstraktnu klasu zato što ne je ne želimo moći instancirati u objekt.

```

abstract class Person{
    protected int $id;
    protected string $firstName;
    protected string $lastName;
}

```

Imat ćemo konstruktor koji će moći napuniti. Svaki puta kada ćemo instancirati naš objekt iz klase **Participant** ili **Organizator** onda ćemo preko ovog konstruktora moći napuniti svojstva ime i prezime. Želimo dati pristup ovim svojstvima pomoću gettera i settera. Definirat ćemo konstruktor u klasi:

```

public function __construct(string $firstName, string $lastName){
    $this->firstName = $firstName;
    $this->lastName = $lastName;
}

```

Idemo napraviti klasu **Participant** iz apstraktne klase **Person**.

```

class Participant extends Person{
}

$participant = new Participant("Pero", "Perić");
echo $participant->firstName . " " . $participant->lastName;

```

Nećemo moći pristupiti **\$firstName** i **\$lastName** svojstvima jer su **protected**. To znači da su ta svojstva dostupna ili samo unutar klase **Person** ili unutar klase koja nasljeđuje klasu **Person**. To je u ovom slučaju **Participant**. Moguće im je naravno pristupiti sa getter i setter. Dakle trebamo napisati getter klasu unutar **Person** apstraktne klase. Time možemo izvan klase pristupiti svojstvu **firstName**.

```

public function getFirstName(): string{
}

```

```
    return $this->firstName;
```

Kod ispisa možemo pozvati svojstvo i dobiti informaciju:

```
echo $participant->getFirstName . " " . $participant->lastName;
```

Drugi način je da umjesto gettera i settera koristimo `__get` magičnu metodu:

```
public function __get($name){  
    return $this->$name;  
}
```

Kada pokušamo pristupiti svojstvu izvana koje je učahureno u našoj klasi aktivira se magična metoda `__get` koja to može vratiti. Kada bi vidjeli samo zadnja dva reda:

```
$participant = new Participant("Pero", "Perić");  
echo $participant->firstName . " " . $participant->lastName;
```

zaključili bi da je svojstvo `firstName` `public` i da ga možemo mijenjati. Npr. s ovim redom:

```
$participant->firstName = "Iva";
```

Kako to nije istina, dojavit će nam grešku `Undefine properties...` I morat ćemo tražiti metodu `__get` pomoću koje smo došli do svojstva. Ovo zna biti zbumujuće. Vratit ćemo se klasičnim getterima i setterima i napisati i ostale.

```
public function getLastname(): string{  
    return $this->lastName;  
}  
public function getLastname(): string{  
    return $this->lastName;  
}  
public function getId(): int{  
    return $this->id;  
}
```

Trebamo popuniti i klasu `Participant`. Trebamo generirati i jedinstveni identifikator. Želimo ga automatski generirati. Ako je ID `static` iz objekta ćemo mu pristupiti ovako:

```
echo $p1->getId();
```

Ipak, ostaviti ćemo svojstvo čistim. Želimo da se automatski aktivira kada instanciramo objekt. To znači da ćemo morati koristiti konstruktor. Počet ćemo ovako:

```
class Participant extends Person{  
    static private int $nextId = 1;  
    private ParticipantStatus $status;
```

Sada ćemo kreirati konstruktor.

```
public function __construct(string $firstName, string $lastName){
    parent::__construct($firstName, $lastName);
    $this->id = self::$nextId++;
    $this->status = ParticipantStatus::Pending;
```

Bez reda `parent::__construct($firstName, $lastName);` nećemo imati zapisano ime i prezime. parent konstruktor je zaprimio ime i prezime. Sa redom `$this->id = self::$nextId++;` zapisana vrijednost bit će u svojstvu ID. Koristili smo `self` jer se radi o statickoj vrijednosti. Radilo bi i sa `$this->id = $this->nextId++;` ali to nije dobra praksa. Benefit `static` svojstva je da zadržava svojstvo, što nam upravo treba. Prilikom instanciranja objekta svojstvo `$nextId` ne postoji nad objektom nego nad klasom. Pošto ima static ispred, zadržava prethodnu vrijednost. Svaki put kada kreiramo nove objekte, konstruktor poveća ID za jedan.

Mogli smo koristiti i `uniqid` ugrađenu funkciju.

```
private string $id;
...
...
$this->id = uniqid();
```

Za ovo su se mogli koristiti generatori ili vlastiti algoritmi. Ipak vratit ćemo na postavke od prije.

Dodat ćemo da je default svojstvo `Pending`. Dobra praksa kaže da to napravimo kod konstruktora. Ako napravimo kod definicija ispod naziva klase:

```
private ParticipantStatus $status = ParticipantStatus::Pending;
```

Bolje je u konstruktoru napisati:

```
$this->status = ParticipantStatus::Pending;
```

a kod definicije klase ostaviti:

```
private ParticipantStatus $status
```

Status sudionika može promijeniti organizator. Sudionici trebaju biti vezani uz događaj. Imat ćemo organizatora koji će imati nekakav niz događaja (events) kojima organizator upravlja, dolazi do njih i potencijalno im mijenja status. Vrijednost ID nećemo izlagati, nećemo ga mijenjati. Nismo stavili niti setttere za promjenu imena (a mogli smo). Puno je bitnije da možemo promijeniti status sudionika dakle moramo nekakav set status. Taj set status treba biti ograničen na parametar enumeratora. Ako neko pokuša poslati dio koji nije dio enumeratora, neće ga moći poslati. Na taj način štitimo i samo svojstvo.

Evo klase `Participant`:

```
class Participant extends Person{
    static private int $nextId = 1;
    private int $id;
    private ParticipantStatus $status;

    public function __construct(string $firstName, string $lastName){
```

```

        parent::__construct($firstName, $lastName);
        $this->id = self::$nextId++;
        $this->status = ParticipantStatus::Pending;
    }

    public function getStatus(): ParticipantStatus{
        return $this->status;
    }

    public function setStatus(ParticipantStatus $status): void{
        $this->status = $status;
    }
}

```

Nakon ovoga počinjemo pisati klasu `Organizer`:

```

class Organizer extends Person{

    public function __construct(string $firstName, string $lastName){
        parent::__construct($firstName, $lastName);
    }
}

```

Ponovo nasljeđujemo klasu `Person`. Znamo da tamo imamo ime i prezime. Odmah kreiramo konstruktor u ovoj klasi jer želimo pozvati roditeljski konstruktor i proslijediti ime i prezime. Kako je u zadatku ponovo potreban jedinstveni identifikator to znači da ćemo taj dio u cijelosti preuzeti iz klase `Participant`. Ubacit ćemo i getter `getId`.

```

class Organizer extends Person{
    static private int $nextId = 1;
    private int $id;
    private ParticipantStatus $status;

    public function __construct(string $firstName, string $lastName){
        parent::__construct($firstName, $lastName);
        $this->id = self::$nextId++;
    }
}

```

`getId` sada se nalazi i u `Participant` i u `Organizer`. Također `private int $id;` se nalazi u obje klase. To možemo dići na klasu `Person` i na taj način napraviti refaktoriranje koda (engl. code refactoring)² refactoring i na taj način smanjiti redundanciju. Iz klase `Participant`, uzet ćemo red `private int $id;` i zalijepiti ga u prvi red klase `Person`. Umjesto `private` upisat ćemo `protected`. Dakle cijeli red

² disciplinirana tehnika restrukturiranja postojećeg tijela kôda kojom se mijenja njegova unutarnja struktura, a da se pritom ne mijenja njegovo vanjsko ponašanje

će glasiti `protected int $id;`. Digli smo svojstvo na roditeljsku klasu. Prebacit ćemo i getter `getId()` iz `Participant` u apstraktnu klasu `Pearson`. Iz klase `Participant` možemo obrisati red `private int $id;` jer nam više ne treba. Ako pogledamo logiku, klasa `Pearson` ne generira ID-jeve ali u njoj postoji svojstvo ID i postoji getter `getId`, međutim klasu `Person` ne možemo istancirati u objekt (jer je apstraktna) i zato nas ne brine što ovdje nikada neće biti ID-jeva, nego će uvijek biti neinicijaliziran ID. U klasici `Organizer` sada trebamo ukloniti `private int $id;` red. Nakon toga trebamo i ukloniti funkciju `getId` jer se već nalazi u apstraktnoj klasici `Pearson`. Kod sada izgleda ovako:

```
<?php

// enumerator EventType zadan po vrsti
enum EventType: string {
    case Conference = "Konferencija";
    case Seminar = "Seminar";
    case Workshop = "Radionica";
}

// enumerator EventStatus zadan po statusu
enum EventStatus: string {
    case Planned = "Planiran";
    case Confirmed = "Potvrđen";
    case Canceled = "Otkazan";
}

// enumerator ParticipantStatus zadan po statusu sudjelovanja
enum ParticipantStatus: string {
    case Accepted = "Prihvaćen";
    case Declined = "Odbijen";
    case Pending = "Na čekanju";
}

interface Confirmable{
    public function confirm(): void;
    public function cancel(): void;
}

abstract class Person{
    protected int $id;
    protected string $firstName;
    protected string $lastName;

    public function __construct(string $firstName, string $lastName){
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }
}
```

```
public function getId(): int{
    return $this->id;
}

public function getFirstName(): string{
    return $this->firstName;
}

public function getLastName(): string{
    return $this->lastName;
}

class Participant extends Person{
    static private int $nextId = 1;
    private ParticipantStatus $status;

    public function __construct(string $firstName, string $lastName){
        parent::__construct($firstName, $lastName);
        $this->id = self::$nextId++;
        $this->status = ParticipantStatus::Pending;
    }

    public function getStatus(): ParticipantStatus{
        return $this->status;
    }

    public function setStatus(ParticipantStatus $status): void{
        $this->status = $status;
    }
}

class Organizer extends Person{
    static private int $nextId = 1;

    public function __construct(string $firstName, string $lastName){
        parent::__construct($firstName, $lastName);
        $this->id = self::$nextId++;
    }

    public function addEvent(Confirmable $event): void{
        $this->events[] = $event;
    }
}
```

```
public function confirmEvent(Confirmable $event): void{
    if(in_array($event, $this->events)){
        $event->confirm();
    }
}
```

Da li sve radi možemo provjeriti ovako:

```
$p1 = new Participant("Marko", "Marković");
$p1->setStatus(ParticipantStatus::Accepted);
$p2 = new Participant("Janko", "Janković");
$o1 = new Organizer("Petar", "Petrović");
var_dump($p1, $p2, $o1);
```

Rezultat:

```
object(Participant)#1 (4) {
    ["id":protected]=>
    int(1)
    ["firstName":protected]=>
    string(5) "Marko"
    ["lastName":protected]=>
    string(9) "Marković"
    ["status":"Participant":private]=>
    enum(ParticipantStatus::Accepted)
}
object(Participant)#5 (4) {
    ["id":protected]=>
    int(2)
    ["firstName":protected]=>
    string(5) "Janko"
    ["lastName":protected]=>
    string(9) "Janković"
    ["status":"Participant":private]=>
    enum(ParticipantStatus::Pending)
}
object(Organizer)#6 (4) {
    ["id":protected]=>
    int(1)
    ["firstName":protected]=>
    string(5) "Petar"
```

```
[{"lastName":protected]=>
    string(9) "Petrović"
}
```

Organizator upravlja sa više događaja (evenata), što znači da u klasi Organizer trebamo postaviti neko svojstvo, neki niz. Za sada ne znamo kakav. U drugom redu definicije klase dodat ćemo: `private array $events`. Kada instanciramo klasu i dobijemo objekt, ovaj `events` ako nije napisan on će potencijalno biti neinicijaliziran. Pretpostavljamo da imamo matricu ali je nemamo. Evo objašnjenja što se dešava. Recimo da napišemo klasu `addEvent` u klasi `Organizer` koja će dodavati događaj (poslije ćemo ukloniti tu klasu):

```
public function addEvent(string $event): void{
    $this->events[] = $event;
}
```

Tu funkciju ćemo pozvati na ovaj način:

```
$o1->addEvent("Konferencija");
var_dump($o1);
```

Kada ovo pokrenemo vidjet ćemo da radi ali je vrijednost neinicijalizirana. Ako napravimo neki getter i pokušamo preuzeti neinicijaliziranu vrijednost, dobit ćemo grešku. Rješenje je ili kod definiranja `events` izmijeniti red u `private array $events = []`; ili na kraj konstruktora dodati `$this->events = []`. Ako bi stavljali neku default vrijednost logičnije da bude u konstruktoru. Radi oboje.

Trebamo još definirati klasu `Event`. Organizatora nećemo vezati za organizatora. Moguće je da 2 organizatora rade na 1 eventu. Reći ćemo koji su sudionici na eventu.

```
class Event implements Confirmable{
    static private int $nextId = 1;
    private int $id;
    private string $name;
    private EventType $type;
    private EventStatus $status;
    private DateTime $startDate;
    private array $participants = [];
    // da li je odobrio organizator
    private bool $isApproved = false;

    public function __construct(string $name, EventType $type, DateTime
$startDate){
        $this->name = $name;
        $this->type = $type;
        $this->startDate = $startDate;
        $this->status = EventStatus::Planned;
        $this->id = self::$nextId++;
    }
}
```

```
}
```

Settere i gettere moguće je naknadno dopisati. Želimo potvrdi event. Statuse koje imamo su Zakazan, Odobren, Otkazan, Završen. Promjenit ćemo ih u Planiran, Potvrđen i Otkazan. Početni status je Planiran. Organizator da status Potvrđen. Želimo omogućiti da organizator na eventom ima mogućnost da pozove metodu koja će promijeniti status u `Confirmed`. Možemo odlučiti da osim eventa imamo npr. događaj čajanke. Čajanke isto tako mogu imati statuse. Planirana čajanka, potvrđena čajanka i otkazana čajanka. Imamo 2 različita objekta Event i Čajanku. Oba trebaju na sebi imati metodu koju će promijeniti u status potvrđeno. Toj metodi pristupat će organizator. Organizator je taj koji će nad eventom moći pozvati metodu potvrdi, a ta metoda će promijeniti status iz `Planned` koji je po defaultu.

```
public function confirm(): void{
    $this->status = EventStatus::Confirmed;
}

public function cancel(): void{
    $this->status = EventStatus::Canceled;
}
```

Ako pogledamo klasu `Organizer`, možemo imati neku matricu s eventima. Organizator može potvrditi event tako da pozove metodu `confirm`.

Kreiramo novog organizatora i event:

```
$o2 = new Organizer("John", "Doe");
$e1 = new Event("PHP Konferencija", EventType::Conference, new DateTime("2024-10-10"));
```

Ako pogledamo `$o2` ima opciju `addEvent`. Napravimo to:

```
$o2 = new Organizer("John", "Doe");
$e1 = new Event("PHP Konferencija", EventType::Conference, new DateTime("2024-10-10"));
$o2->addEvent($e1);

var_dump($o2);
```

Rezultat je:

```
enum(EventType::Conference)
["status":"Event":private]=>
enum(EventStatus::Confirmed)
["startDate":"Event":private]=>
object(DateTime)#12 (3) {
    ["date"]=>
    string(26) "2024-10-10 00:00:00.000000"
```

```

        ["timezone_type"]=>
        int(3)
        ["timezone"]=>
        string(13) "Europe/Berlin"
    }
    ["participants":"Event":private]=>
    array(0) {
    }
    ["isApproved":"Event":private]=>
    bool(false)
}
}

```

Želimo da organizator može potvrđivati evente. To možemo napraviti tako da prođe po matrici `$events` koju smo formirali pri vrhu klase `Organizer` ili tako da definiramo novu funkciju za potvrdu eventa.

```

public function confirmEvent(Confirmable $event): void{
    if(in_array($event, $this->events)){
        $event->confirm();
    }
}

```

Želimo provjeriti da li taj event postoji u matrici `$events` kako bi ga potvrdili. Možemo reći npr.

```
$e3 = new Event("Python Seminar", EventType::Seminar, new DateTime("2024-12-12"));
```

Organizator `$o1` nema u svojoj matrici `$events` ovaj event jer ga nigdje nismo dodijelili. Dakle nitko ne brani da napravimo potvrdu `$o1` iako ga on nije napravio:

```
$o1->confirmEvent($e2);
```

To se ne smije dogoditi. Problem je oko metode `confirmEvent`, ali ne u njoj direktno nego u organizatorov `confirmEvent`. Iz tog razloga ćemo u `Event` klasi promijeniti ime funkcije `confirmEvent` u `confirm`. Ta funkcija zaprimi neki event i onda nad tim eventom pozove `confirmEvent`. U klasi `Organizer` metoda `confirmEvent` sada će glasiti:

```

public function confirmEvent(Confirmable $event): void{
    $event->confirm();
}

```

Organizator `$o1` poziva metodu `confirmEvent`. Poslao je `$e2` objekt. Objekt ulazi u funkciju. Na sebi ima metodu `confirm`. To je bug jer organizator koji nema veze sa ventom može promijeniti stanje. To znači da `confirmEvent` treba drugačije napisati. Ubacili smo `if` koji provjerava da li je u matrici event koji je dobila metoda kao ulaz `$event`, da li se on nalazi u listi `events`. Ako se nalazi, potvrdi event. Ako ne nema potvrde.

```
public function confirmEvent(Confirmable $event): void{
    if(in_array($event, $this->events)){
        $event->confirm();
    }
}
```

\$o1 kada pokuša potvrditi status koji nije njegov ostat će u statusu planiran, neće biti potvrđen.

U nekom trenutku želimo proširiti naš Event manager i ubaciti klasu Cajanka. Klasa neće ekstendati Event.

```
class Cajanka{
    private EventStatus $status = EventStatus::Planned;

    public function confirm(): void{
        $this->status = EventStatus::Confirmed;
    }
}
```

Cajanka ima svoju vlastitu metodu za potvrđivanje. Organizator može promijeniti status na Cajanka. Npr.:

```
$c1 = new Cajanka();
$o1->confirmEvent($c1);
var_dump($e2);
```

Organizator na svojoj metodi očekuje event a ne čajanku. Sada imamo dvije iste metode confirm u klasi Cajanka i u klasi Event. Rješenje je da to prebacimo u interface. Ako je klasa jezgra, interface je omotač koji omogućava komunikaciju s ostatkom našeg koda.

Interface koje tjeraju sve naše klase da implementiraju metodu za potvrđivanje bio to Event, Cajanka ili nešto drugo ćemo nazvati Confirmable.

```
interface Confirmable{
    public function confirm(): void;
    public function cancel(): void;
}
```

Ove dvije metode bit će nam dovoljne da kontroliramo status. Status sada možemo promijeniti na bilo kojem eventu koji implementira Confirmable interface. Klasa Event ima status. Status će kontrolirati ove dvije metode confirm i cancel. Klasu ćemo natjerati tako da to poštuje tako da ćemo umjesto class Event{ napisati class Event implements Confirmable{ i nakon toga vidimo grešku 'Event' does not implement methods 'confirm' and 'cancel'. Tjera klasu Event to implementirati. Dakle trebamo napisati te dvije metode u klasi Event jer bez toga neće raditi.

```
public function confirm(): void{
    $this->status = EventStatus::Confirmed;
}
```

```
public function cancel(): void{
    $this->status = EventStatus::Canceled;
}
```

Više nema greške. Sada, čemo u organizatoru reći da možemo potvrditi matricu. U matricu može dobiti različite objekte event ili čajanka. Organizatora zanima da li objekt koji je dobio na potvrđivanje ima implementiran `interface Confirmable`. Interface je natjerao klasu da to ime. U klasi `Organizer` napisat ćemo:

```
public function confirmEvent(Event $event): void{
    $event->confirm()
}
```

Želimo potvrditi sve objekte koji imaju implementiran interface `Confirmable` a ne samo `Event`, tako da nam ovo ne odgovara. Ovo nazivamo **tight coupling** jer smo usko vezali `ConfirmEvent` uz `Event` klasu. Nismo mogli poslati Čajanku. Ako umjesto `Event` napišemo `Confirmable`, tj. umjesto klase napišemo interface, ova metoda kad zaprimi objekt očekuje objekt koji ima implementiran interface `Confirmable`, dakle ima sigurno implementiranu metodu `confirm()`.

```
public function confirmEvent(Confirmable $event): void{
    if(in_array($event, $this->events)){
        $event->confirm();
    }
}
```

Ako je organizator zadužen za taj event, može potvrditi. Sada lako možemo implementirati nešto što nije `Event`, nego neka druga klasa. Na organizatoru ne moramo ništa promijeniti.

U klasi `Organizer` trebamo promijeniti i funkciju `addEvent` koja ima takođe tight coupling tj. usko je vezana za `Event`. Umjesto `Event` trebamo napisati `Confirmable`. Sada funkcija glasi:

```
public function addEvent(Confirmable $event): void{
    $this->events[] = $event;
}
```

Sada organizator može dodati bilo koji objekt koji može implementirati `Confirmable` interface. Sada objekti komuniciraju sa kodom. U Čajanku dodat ćemo `Confirmable` interface., što znači da moramo implementirati `confirm` i `cancel` metode.

```
class Cajanka implements Confirmable{
    private EventStatus $status = EventStatus::Planned;

    public function confirm(): void{
        $this->status = EventStatus::Confirmed;
    }

    public function cancel(): void{
        $this->status = EventStatus::Canceled;
    }
}
```

```

    }
}

}

```

Ovime smo postigli Loose coupling tj. labavo vezivanje

Sada možemo raditi:

```

$o1 = new Organizer("John", "Doe");
$e1 = new Event("PHP Konferencija", EventType::Conference, new DateTime("2024-10-10"));
$e2 = new Event("JavaScript Radionica", EventType::Workshop, new DateTime("2024-11-11"));
$e3 = new Event("Python Seminar", EventType::Seminar, new DateTime("2024-12-12"));
$o1->addEvent($e1);
$o1->addEvent($e2);
$o1->addEvent($e3);
$o1->confirmEvent($e2);

$events = $o1->getEvents();
var_dump($events);
interface Sortable{
    public function sortBy(string $field): mixed;
}

```

`mixed` jer ne znamo što je `sortBy` funkcija vratiti, možda `int` ili `string`. Ako navedemo koje polje ograničavamo funkciju. Ovako ćemo moći koristiti sortiranje po bilo kojem polju. Kod klase `Event` možemo navesti

```
class Event implement Confirmable, Sortable{
```

Cilj je sada implementirati metodu `sortBy` na kraju funkcije:

```

    public function sortBy(string $field): mixed{
        return $this->$field;
    }

```

Zadaća ove funkcije nije je da odradi sortiranje nego da vrati polja koja želimo sortirati sa `$this->$field`; Benefit ove funkcije je da iako je ista logika kao kod settera, možemo korisititi vani ovu metodu. Da definiramo sve gettere i setter otkrili bi sva svojstva.

```

$e1 = new Event("PHP Konferencija", EventType::Conference, new DateTime("2024-10-10"));
$e2 = new Event("JavaScript Radionica", EventType::Workshop, new DateTime("2024-11-11"));
$e3 = new Event("Python Seminar", EventType::Seminar, new DateTime("2024-12-12"));
$o1->addEvent($e1);

```

```
$o1->addEvent($e2);
$o1->addEvent($e3);
$o1->confirmEvent($e1);
```

Sada kada pogledamo, organizator `$o1` ima popunjenu matricu sa 3 elementa. Ako ih probamo ispisati sa:

```
var_dump($o1->events);
```

nećemo uspjeti jer će nam reći da ne možemo pristupiti tom svojstvu zato što nema gettera. Moramo napisati u `Organizer` getter `getEvent`.

```
public function getEvents(): array{
    return $this->events;
}
```

Sada su dostupni svi eventi od tog organizatora, tj. ta matrica.

```
["name":"Event":private]=>
string(14) "Python Seminar"
[{"type":"Event":private}=>
enum(EventType::Seminar)
[{"status":"Event":private}=>
enum(EventStatus::Planned)
[{"startDate":"Event":private}=>
object(DateTime)#13 (3) {
    ["date"]=>
    string(26) "2024-12-12 00:00:00.000000"
    ["timezone_type"]=>
    int(3)
    ["timezone"]=>
    string(13) "Europe/Berlin"
}
[{"participants":"Event":private}=>
array(0) {
}
[{"isApproved":"Event":private}=>
bool(false)
}
```

Sortiranje

Za funkcije `sort`, `rsort`, `asort`, `ksort` korisit QuickSort algoritam za sortiranje. Napraviti ćemo interface `Sortable`. Potrebna je i callable funkcija. To je komparacijska funkcija koja uspoređuje vrijednosti. Na temelju te informacije usort će znati kako da promjeni pozicije u matrici. Ta funkcija će quick sort-u dati istinu ili laž tj. informaciju da li da zamjeni elementima pozicije ili ne.

Te podatke želimo sortirati. U PHP imamo za sortiranje `arsort`, `asort`, `krsort`, `ksort`, `rsort`, `sort`, `uasort` i `usort`. `sort()` treba matricu.

`usort` prima matricu i pravi referencu na nju. Dakle radi na originalnoj matrici tj. na podacima. Mi želimo sortirati izvana jer imamo organizatora koji ima niz event objekata. Te podatke želimo izvući.

```
$events = $o1->getEvents();
usort($events, fn(Sortable $a, Sortable $b) => $b->sortBy('name') <=> $a->sortBy('name'));
```

`fn` je **arrow function**. kada želite koristiti callback funkcije (funkcije koje se proslijeđuju kao argument drugoj funkciji). Ovdje služi za komparaciju dviju vrijednosti. Arrow function nema vitičaste zagrade, nema `return`, i skraćeno se piše `fn`. Callback funkcije su uglavnom anonimne. Možemo to napisati i kao:

```
function(...){
    return ...;}
```

Konkretno u ovom slučaju mogli bi napisati:

```
$events = $o1->getEvents();
usort($events, function(Sortable $a, Sortable $b){
    return $b->sortBy('name') <=> $a->sortBy('name');
});
```

Predavač tvrdi da callback ne treba koristiti za drugu svrhu, pa ih je potrebno koristiti kao anonimne funkcije. `usort` radi pomoću quick sorta.

Quick sort radi tako da se odabere „pivot“ element (slučajno) u nizu i tako podjeli elemenata u dva podmatrice, prema tome jesu li manji ili veći od „pivota“. Podnizovi se zatim sortiraju rekursivno. Gleda gdje je prvi odnosno zadnji broj u nizu u odnosu na pivot.

Nismo spomenuli `<=>` spaceshift komparatora koji gleda dvije vrednosti i vraća cijeli broj -1, 0 ili 1 ovisno o tome da li je `$a < $b`, `$a = $b` ili `$a > $b`. Problem je što objekte ne možemo komparirati jer ne znamo što je veće a što manje. Ako želimo okrenuti redoslijed sortiranja, obrnemo dio b i a dio ispred spaceshift.

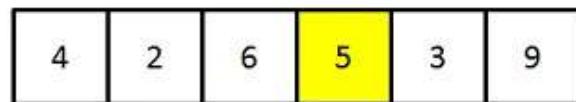
Na idućoj strani vidi se princip rada sortiranja. Ovdje vidimo različite složenosti algoritmova:

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Heapsort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Counting Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
Cubesort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

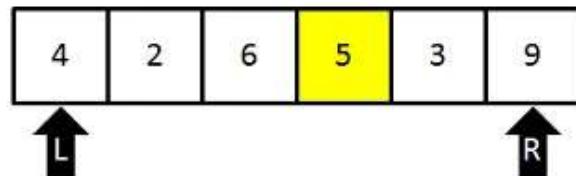
Korak 1

Odredi pivot



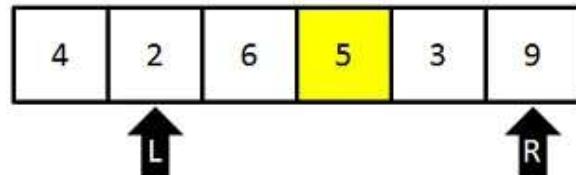
Korak 2

Pokreni pointer s lijeve i desne strane



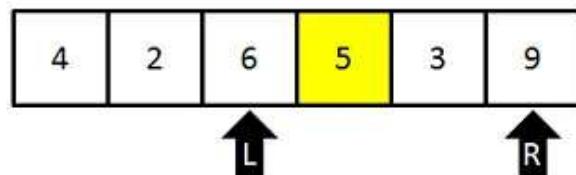
Korak 3

Budući da je $4 < 5$, pomakni lijevi pointer



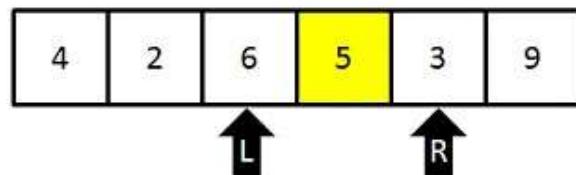
Korak 4

Budući da je $2 < 5$, pomakni lijevi pointer. Budući da je $6 > 5$, stani.



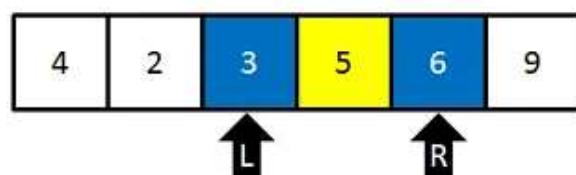
Korak 5

Budući da je $9 > 5$, pomakni desni pointer. Budući da je $3 < 5$, stani.



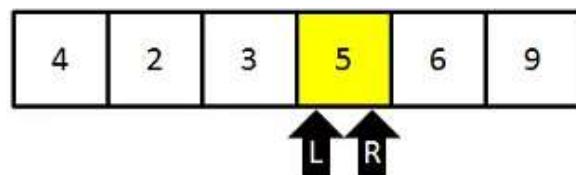
Korak 6

Zamijeni vrijednosti pointera.



Korak 7

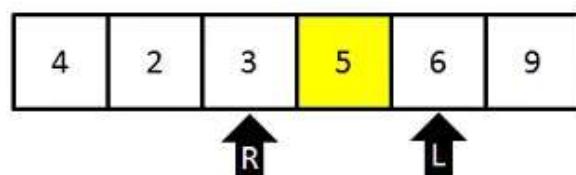
Pomakni pointere za još jedan korak



Korak 8

Budući da je $5 == 5$, pomakni pointere još jedan korak.

Stani.



PHP uvodi imenski prostor (engl. Namespace) koji rješavaju dva različita problema: omogućavaju bolju organizaciju grupiranjem klasa koji zajedno rade na obavljanju zadatka, dopuštaju da se isto ime koristi za više od jedne klase. Kao što direktorij pohranjuje povezane datoteke, slično je s imenskim prostorom koji grupira povezane klase.

Direktorij vam ne dopušta da imate dvije datoteke s istim imenom. Međutim, možete imati datoteke s istim imenima u različitim direktorijima. Isto tako, prostori imena oponašaju isti princip.

Po definiciji, imenski prostori pružaju vam način grupiranja povezanih klasa i pomažu vam da izbjegnete potencijalne kolizije imena.

Imenski prostori nisu ograničeni na grupiranje klasa. Mogu se grupirati drugi identifikatori, uključujući funkcije, konstante, varijable itd.

Da biste definirali imenski prostor, postavite `namespace` ključnu riječ iza koje slijedi naziv, npr. `App\Classes`:

Uobičajno je dodijeliti u glavnom direktoriju projekta `src` direktorij u kojem se nalaze poddirektoriji za klase `Classes`, enumeratore `Enums` i interfejs `Interfaces`. Dobra je praksa oponašati strukturu direktorija s imenskim prostorom kako biste lakše pronašli klase. Na primjer, svaka klasa unutar direktorija dobit će imenski prostor `Classes`. Na taj način izoliramo prostor od ostatka. Možemo imati više takvih prostora koje kasnije možemo povezivati. Govorili smo o doseg (engl. scope) koje imaju varijable. Isto tako je i kod imenskog prostora (engl. Namespace) gdje možemo imenovati prostor gdje klase koje imaju ista imena ali nisu u istom imenskom prostoru. Tako izbjegavamo kolizije imena.

Upravo ovaku organizaciju napravit ćemo sa aplikacijom event menager koju smo do sada pisali.

Koristit ćemo autoloader da ne bi smo ručno uključivali datoteke. Krenut ćemo od enumeratora u direktoriju `Enums`.

Potrebno je dati ime imenskog prostora gdje se nalazi enumerator. U našem slučaju mi ćemo dati `App\Enums`.

[EventStatus.php](#)

```
<?php

namespace App\Enums;

enum EventStatus: string {
    case Planned = "Planiran";
    case Confirmed = "Potvrđen";
    case Canceled = "Otkazan";
}
```

[EventType.php](#)

```
<?php

namespace App\Enums;

enum EventType: string {
    case Conference = "Konferencija";
    case Seminar = "Seminar";
    case Workshop = "Radionica";
}
```

[ParticipantStatus.php](#)

```
<?php

namespace App\Enums;

enum ParticipantStatus: string {
    case Accepted = "Prihvaćen";
    case Declined = "Odbijen";
    case Pending = "Na čekanju";
}
```

Idemo na [Interfaces](#) direktorij.

[Confirmable.php](#)

```
<?php

namespace App\Interfaces;
```

```
interface Confirmable{
    public function confirm(): void;
    public function cancel(): void;
}
```

Sortable.php

```
<?php

namespace App\Interfaces;

interface Sortable{
    public function sortBy(string $field): mixed;
}
```

Na redu su [Classes](#).

Person.php

```
<?php

namespace App\Classes;

abstract class Person{
    protected int $id;
    protected string $firstName;
    protected string $lastName;

    public function __construct(string $firstName, string $lastName){
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }

    public function getId(): int{
        return $this->id;
    }

    public function getFirstName(): string{
        return $this->firstName;
    }

    public function getLastname(): string{
        return $this->lastName;
    }
}
```

```
}
```

Event.php

```
<?php

namespace App\Classes;

use DateTime;
use App\Enums\EventStatus;
use App\Enums\EventType;
use App\Interfaces\Confirmable;
use App\Interfaces\Sortable;

class Event implements Confirmable, Sortable{
    static private int $nextId = 1;
    private int $id;
    private string $name;
    private EventType $type;
    private EventStatus $status;
    private DateTime $startDate;
    private array $participants = [];
    private bool $isApproved = false;

    public function __construct(string $name, EventType $type, DateTime
$startDate){
        $this->name = $name;
        $this->type = $type;
        $this->startDate = $startDate;
        $this->status = EventStatus::Planned;
        $this->id = self::$nextId++;
    }

    public function confirm(): void{
        $this->status = EventStatus::Confirmed;
    }

    public function cancel(): void{
        $this->status = EventStatus::Canceled;
    }

    public function sortBy(string $field): mixed{
        return $this->$field;
    }
}
```

Organizer.php

```
<?php

namespace App\Classes;

use App\Interfaces\Confirmable;

class Organizer extends Person{
    static private int $nextId = 1;
    private array $events = [];

    public function __construct(string $firstName, string $lastName){
        parent::__construct($firstName, $lastName);
        $this->id = self::$nextId++;
    }

    public function getEvents(): array{
        return $this->events;
    }

    public function addEvent(Confirmable $event): void{
        $this->events[] = $event;
    }

    public function confirmEvent(Confirmable $event): void{
        if(in_array($event, $this->events)){
            $event->confirm();
        }
    }
}
```

[Participant.php](#)

```
<?php

namespace App\Classes;

use App\Enums\ParticipantStatus;

class Participant extends Person{
    static private int $nextId = 1;
    private ParticipantStatus $status;

    public function __construct(string $firstName, string $lastName){
        parent::__construct($firstName, $lastName);
        $this->id = self::$nextId++;
        $this->status = ParticipantStatus::Pending;
    }

    public function getStatus(): ParticipantStatus{
        return $this->status;
    }

    public function setStatus(ParticipantStatus $status): void{
        $this->status = $status;
    }
}
```

S ovime smo gotovi sa sadržajem [src](#). Ostatak ćemo nazvati [index.php](#) i staviti u korijenski direktorij. U našem slučaju [Predavanje04](#).

```
<?php

require_once "vendor/autoload.php";

use App\Classes\Organizer;
use App\Classes\Event;
use App\Enums\EventType;
use App\Interfaces\Sortable;

$o1 = new Organizer("John", "Doe");
$e1 = new Event("PHP Konferencija", EventType::Conference, new DateTime("2022-10-10"));
$e2 = new Event("JavaScript Radionica", EventType::Workshop, new DateTime("2022-11-11"));
```

```
$e3 = new Event("Python Seminar", EventType::Seminar, new DateTime("2022-12-12"));
$o1->addEvent($e1);
$o1->addEvent($e2);
$o1->addEvent($e3);
$o1->confirmEvent($e1);

$events = $o1->getEvents();
var_dump($events);
usort($events, fn(Sortable $a, Sortable $b) => $b->sortBy('name') <=> $a->sortBy('name'));
var_dump($events);
```

Kada smo razdvojili kod, autoloderom ćemo pokriti direktorij u kojem se nalaze svi ti resource-i koji su obuhvaćeni namespace-om. Ovo će biti ulazna točka autoloaderu da unutra traži ono što mu treba na osnovu putanje. Pronaći će imenski prostor koji će biti vezan za strukturu direktorija. Na taj način imenskim prostorima organiziramo cijeli kod.

Da bi ovaj kod radio u bez autoloadera, u `index.php` morali bi smo navesti čitav niz `include „src...“`, za `Classes`, `Enums` i `Interfaces`. include počinje stvarati zbrku. Zato je bolje koristiti imenske prostore. Oni se u gornjem kodu već nalaze. Kasnije ćemo koristiti autoload pomoću Composer-a po PS4 standardu. Za sada ćemo raditi ručno i napraviti svoj autoloader. Na početku `index.php` datoteke ćemo uključiti taj `autoload.php` koji ćemo pozvati iz `vendor` direktorija.

```
require_once "vendor/autoload.php";
```

Ubacit ćemo imenski prostor. U svim datotekama u direktoriju `Classes` dodat ćemo red `namespace App\Classes;`, u svim datotekama u direktoriju `Enums` dodat ćemo red `namespace App\Enums;`, u svim datotekama u direktoriju `Interfaces` dodat ćemo red `namespace App\Interfaces;`.

Čim dodamo prvi `namespace`, Visual Studio će podvući greške. Npr. `App\Classes` je alias ili prefix `src` direktorija. Npr. ako gledamo `Organizer.php` (pogledaj gore), iako smo naveli `namespace App\Classes;`, u tom imenskom prostoru VSC ne vidi klasu `Person` jer red glasi:

```
class Organizer extends Person{
```

Klasa `Person` je u istom direktoriju. Ako i tamo dodjelimo `namespace App\Classes;`, klase su u istom prostoru i greška nestaje.

AKO dalje opogleđamo u `Organizer.php` i dalje imamo grešku za `Confirmable` jer on nije u istom imenskom prostoru. `Confirmable` se nalazi u `namespace App\Interfaces;`. Rješenje nije da sve bude u istom imenskom prostoru jer ne mogu biti dvije klase sa istim imenom u istom imenskom prostoru.

Rješenje za `Organizer.php` je koristiti naredbu `use App\Interfaces\Confirmable;` kako bi dali validnu putanju do imenskog prostora u kojem se nalazi interface jer nisu u istom imenskom prostoru. To je odlično iz vizure VSC ali kod i dalje neće raditi neko mora napraviti include naredbu. U ovom slučaju da to radimo ručno pisali bi smo `include_once „./Interfaces/Confirmable.php“;` Za nas će to napraviti autoloader, on će na osnovu direktive `use` i navedenog imenskog prostora `use`

`App\Interfaces\Confirmable;` pronaći datoteku i napraviti `include_once` „`../Interfaces/Confirmable.php`“. Tako nećemo dobiti grešku. Dodatni use nije potreban ako su u istom imenskom prostoru.

Radimo tako da svakoj datoteci prvo definiramo imenske prostore, snimimo a zatim ostale elemente.

Pogledamo li `Event.php`, vidimo da `DateTime` nije u imenskom prostoru `App\Classes`; Treba navesti PHP imenski prostor. Problem je što sada traži u našem imenskom prostoru. Trebamo navesti `use DateTime`; i više nema greške. Naravno, treba navesti i ostale potrebne `use`:

```
use DateTime;  
use App\Enums\EventType;  
use App\Interfaces\Confirmable;  
use App\Interfaces\Sortable;
```

Ovo je lakše rješenje nego sa `include`.

Na `index.php` također trebamo staviti klase koje koristimo:

```
use App\Classes\Organizer;  
use App\Classes\Event;  
use App\Enums\EventType;  
use App\Interfaces\Sortable;
```

U `index.php`, tj. u onome što je preostalo nije potrebno navoditi `use` za `DateTime` jer `index.php` nema svoj imenski prostor.

Sa `use` je moguće raditi aliase npr. `use App\ime_direktorija as Nešto`; Na taj način ako imamo dvije funkcije ili klase sa istim imenom možemo ih staviti u različit imenski prostor a da nema kolizije. Include nema mogućnost aliasa i nije ga moguće dobro namjestiti.

Ostalo je da još napravimo `autoloader.php`- koji ćemo staviti u direktorij `vendor`.

```
<?php  
  
spl_autoload_register(function($class){  
    $prefix = "App\\\";  
    $base_dir = __DIR__ . "/..src/";  
    $len = strlen($prefix);  
  
    if(strncmp($prefix, $class, $len) !== 0){  
        return;  
    }  
  
    $relative_class = substr($class, $len);  
    $file = $base_dir . str_replace("\\", "/", $relative_class) . ".php";
```

```
if(file_exists($file)){
    require $file;
}
});
```

... izađe iz vendora i `/src/` uđe u `src`. `require $file` zaustavlja rad a to nije dobro jer prekida program.

U index.php potrebno je naravno ubaciti red:

```
require_once "vendor/autoload.php";
```

kako bi smo pozvali autoloader i kako bi sve radilo kako treba.

Uvod u iznimke

- **Što su iznimke?**
 - o Specijalni objekti u programiranju koji upravljaju neočekivanim greškama.
 - o Omogućavaju kontrolirano upravljanje greškama umjesto zaustavljanja programa.
- **Zašto koristiti iznimke u PHP-u?**
 - o Čišći i čitljiviji kod.
 - o Lakše otkrivanje i rješavanje problema.
 - o Mogućnost reagiranja na specifične greške na specifične načine.

Iznimke su objekti koji predstavljaju neočekivane situacije ili pogreške koje se mogu pojaviti tijekom izvršavanja programa. Kada se takva situacija dogodi, program "baca" (engl. throw) iznimku, što znači da prekida normalno izvršavanje i proslijedi kontrolu do odgovarajućeg "hvatača" (engl. catch) iznimke.

Sintaksa iznimki

- **Osnovna sintaksa:**

```
try {
throw new Exception("Došlo je do greške");
} catch (Exception $e) {
echo $e->getMessage();
} finally {
echo "Ovo se uvijek izvodi.";
}
```

Glavna razlika je da `try-catch` blokovi hvataju i rukuju iznimkama, dok `throw` ručno generira iznimke. `try-catch` blok se koristi za rukovanje iznimkama koje bacaju ugrađene funkcije ili metode, dok se `throw` koristi za ručno generiranje iznimki kao odgovor na neočekivane situacije u vašem kodu.

Obično se `try-catch` blokovi koriste za rukovanje iznimkama koje mogu baciti ugrađene funkcije ili metode, dok se `throw` koristi za ručno generiranje vlastitih iznimki u vašem kodu.

`throw` ručno generiranja iznimki koristi se kod validacije ulaznih podataka

- **Ključne riječi:**

- o `try`: blok koda koji se testira za greške dok se izvodi
- o `throw`: baca iznimku kada se dogodi problem
- o `catch`: omogućuje hvatanje i obradu iznimke
- o `finally`: blok koda koji se izvodi nakon `try-catch` blokova, opcionalan

Vrste iznimki

- **Standardne iznimke:**

- o `Exception`: bazna klasa za sve iznimke. Implementira interface `Throwable`
- o `ErrorException`: koristi se za obradu PHP grešaka kao iznimki. Napravljena je iz bazne klase `Exception`.

Tvorci PHP su to odlučili razdvojiti `Exception` i `ErrorException` je lakše kontrolirati greške tj. kod. `ErrorException` je nešto što genriра PHP i što možemo uhvatiti kroz `Exception` i obrađivati.

- **Specijalizirane iznimke:**

- o `InvalidArgumentException`: baca se kada se funkciji predaju neispravni argumenti. Npr. tražite `integer` a poslan je negativan broj.
- o `LengthException`: baca se kada se funkcija suoči s problemom dužine, npr. prekratki string.
- o `BadMethodCallException`: pozvali ste metodu koja ne postoji, nema argumente itd.

Obavezno nakon toga staviti `try-catch` blok.

- **Kreiranje prilagođenih iznimki:**

```
class MyCustomException extends Exception {  
    // Prilagođene funkcije ili poruke  
}  
  
try {  
    $file = fopen("ne_postoji.txt", "r");  
    if (!$file) {  
        throw new Exception("Datoteka ne postoji.");  
    }  
} catch (Exception $e) {  
    echo "Greška: " . $e->getMessage();  
}
```

Pogledajmo primjer iznad. Ovime pravimo svoju (prilagođenu) iznimku. Kada kažete npr. `new Exception` to je generičko ime, ali ako kažete npr. `new AccountException` možemo lako prepostaviti da iznimka ima nešto sa pokušajem promjene računa bez autorizacije, da na računu nema novca ili da je račun u minusu, itd..

U primjeru iznad pokušamo otvoriti neku datoteku koja ne postoji. `throw` će baciti iznimku i prekinuti `try` blok i uhvatiti iznimku za `catch`. U `$e` se instancirao objekat `Exception`. To je **Dependency Injection** o kojem ćemo kasnije pričati. To PHP ispod haube sam radi. Tu se generira objekt sa podatkom koji je poslan konstruktoru. Ako pogledamo u helpu pod `Exception`, vidjet ćemo da u metodi `__construct` može primiti `$message` koja je pod default prazan string i `int $code = 0`.

```
/* Methods */
public __construct(string $message = "", int $code = 0, ?Throwable $previous = null)
```

To znači da je informacija koju smo poslali sa `new Exception` u našem primjeru, završila u `$message` koji je svojstvo (varijabla u klasi) a do njega možemo doći sa metodom `getMessage()`.

Razlika da smo išli sa `if (!file)` i sa `echo` je što znamo gdje se dogodila greška, možemo pomoći drugih metoda saznati u kojem redu ili u kojoj datoteci se dogodila iznimka (pogledati metode `Exception`) i to možemo zapisati. Kasnije lako možemo to analizirati i pronaći u čemu je problem.

Pogledajmo još jedan primjer:

```
function validateAge($age) {
    if ($age < 0) {
        throw new InvalidArgumentException("Starost ne može biti negativna.");
    }
    echo "Ispravna starost";
}
try {
    validateAge(-1);
} catch (InvalidArgumentException $e) {
    echo "Greška: " . $e->getMessage();
}
```

Imamo funkciju `validateAge`. Ta funkcija provjerava da li je `$age < 0` onda izbaci iznimku sa `throw` i `InvalidArgumentException` (jer tražimo neispravan argument) javi da starost ne može biti negativna. Ako neko pokuša poslati argument koji nije točan, izbacit će se iznimka, funkcija se zaustavlja (slično `die` ili `return`). Ako se funkcija pozove u `try` bloku onda će se validirati cijela funkcija na grešku. Ako je nema, funkcija će se završiti. Ako je ima, funkcija prestaje sa radom i `catch` se aktivira i hvata specifičnu grešku a to je `InvalidArgumentException`. Ako se dogodi bilo koja druga greška, to nećemo uhvatiti. Ako pogledamo kod funkcije, takva druga iznimka se ne može dogoditi. Naravno umjesto `InvalidArgumentException` mogli smo napisati i `Exception`. Neki programeri umjesto `Exception` koriste interface `Throwable`, zato što i greške mogu generirati iznimke. Time hvataju sve objekte koji imaju implementiran `Throwable`. Dakle ili klasa ili interface `Throwable`, kada se želi generalno obuhvatiti sve iznimke. Želite li iznimke i error-e koji mogu izbaciti grešku koristi se `Throwable`.

Exception Driven Development (EDD)

- Programerski pristup koji se temelji na prepostavci da će kod tokom izvođenja neizbjegno naići na greške i iznimke.

- Cilj ovog pristupa je strukturirati i napisati kod na način koji anticipira i elegantno upravlja tim greškama, umjesto da se fokusira isključivo na "sretni put" (engl. happy path) gdje se sve izvodi kako je planirano.
- Potiče programere da misle unaprijed o mogućim problemima i načinima njihove obrade, što rezultira robusnijim i pouzdanijim aplikacijama.

Ključna svojstva

1. **Anticipacija grešaka:** Programeri aktivno predviđaju moguće iznimke i greške koje mogu nastati tokom izvođenja programa.
2. **Proaktivno upravljanje iznimkama:** Kod se piše s jasno definiranim blokovima `try`, `catch` i `finally` za obradu iznimaka, što omogućava programima da nastave s radom čak i kad dođe do grešaka.
3. **Testiranje robustnosti:** Uključuje intenzivno testiranje kako bi se osiguralo da aplikacija može elegantno upravljati neočekivanim i izvanrednim situacijama.

Prednosti EDD

- Poboljšava stabilnost i pouzdanost aplikacija.
- Olakšava identifikaciju i rješavanje grešaka.
- Povećava pouzdanost softvera kroz detaljno razmatranje potencijalnih točaka neuspjeha.

U ovom primjeru vidimo da možemo imati više `catch` blokova:

```
function processFile($filename) {
    try {
        $file = fopen($filename, "r");
        if (!$file) {
            throw new Exception("Datoteka ne može biti otvorena.");
        }
        // Obradi podatke iz datoteke
        while (!feof($file)) {
            $line = fgets($file);
            // Prepostavimo da postoji neka specifična logika obrade ovdje
        }
        fclose($file);
    } catch (FileNotFoundException $e) {
        // Specifična obrada za slučaj kada datoteka nije pronađena
        echo "Greška: " . $e->getMessage();
    } catch (Exception $e) {
        // Opća obrada za sve ostale iznimke
        echo "Došlo je do greške: " . $e->getMessage();
    } finally {
        echo "Proces obrade je završen.";
    }
}
```

Vidimo propadanje. Bazni `Exception` mora biti zadnji jer on hvata sve greške. Želimo posebno obraditi npr. `FileNotFoundException` a to ne bi mogli da je `Exception` iznad.

Pogledat ćemo na jednostavnom primjeru kako iznimka radi. Napisat ćemo vlastitu iznimku i pokazati kako je uhvatiti. Poželjno je da u imenu vlastite klase uvijek bude riječ `Exception` jer je onda logično koja je svrha.

```
<?php

// cilj je imati specifični exception u svom kodu i uhvatiti i obraditi tu iznimku
class AccountException extends Exception{}

class Account{
    private float $balance = 0; // imamo balance koji je 0

    // kad nemamo return dobro je naznačiti što metoda vraća, a tu je to void jer ne
    // vraća ništa - prima amount kao neki float
    public function deposit (float $amount) : void{
        // dalje provjeri da li je iznos manji ili jednak 0
        if ($amount <= 0) {
            throw new AccountException("Amount must be grater than 0");
            // ako se dogodi da je amount = ili < 0 onda imamo (specifičnu) iznimku
        }
        $this->balance += $amount;
    }
}
```

Primjer iznimki

Sada ćemo distancirati `$account` iz klase `Account()`:

```
$account = new Account();
```

Ako krenemo po happy path, na račun ćemo za probu depozitom staviti 100 i to ispisati:

```
$account->deposit(100);
var_dump($account);
```

Ako pokrenemo program, sve radi bez problema.

Međutim, ako probamo na račun staviti -100 i probamo to ispisati:

```
$account->deposit(-100);
var_dump($account);
```

Kod pokretanja, pojavit će se greška:

```
PHP Fatal error:  Uncaught AccountException: Amount must be grater than 0 in
C:\xampp\htdocs\Algebra\Napredni PHP\Predavanje05\iznimke.php:15
Stack trace:
#0 C:\xampp\htdocs\Algebra\Napredni PHP\Predavanje05\iznimke.php(23): Account->deposit(-100.0)
```

```
#1 {main}
    thrown in C:\xampp\htdocs\Algebra\Napredni PHP\Predavanje05\iznimke.php on line
15

Fatal error: Uncaught AccountException: Amount must be grater than 0 in
C:\xampp\htdocs\Algebra\Napredni PHP\Predavanje05\iznimke.php:15
Stack trace:
#0 C:\xampp\htdocs\Algebra\Napredni PHP\Predavanje05\iznimke.php(23): Account-
>deposit(-100.0)
#1 {main}
    thrown in C:\xampp\htdocs\Algebra\Napredni PHP\Predavanje05\iznimke.php on line
15
```

`throw` je izbacio grešku ali još nema `try-catch` bloka koji bi tu grešku uhvatio. Idemo ga napisati:

```
try{
    $account = new Account();
$account->deposit(-100); // ovo ispisuje grešku koja je specifična za ovu iznimku
    // stavili smo taj kod u try block jer je poželjno da bude tu ako imamo kod
koji može generirati iznimku
    var_dump($account);
    // ako se dogodila iznimka, tj. PHP ju je uhvatio u objekt $e koji će biti
stvoren u specijalnoj iznimci
} catch(AccountException $e){
    echo "Specific exception: " . $e->getMessage();
}
```

Kada pokrenemo kod, dobivamo:

```
object(Account)#1 (1) {
    ["balance": "Account": private]=>
    float(100)
}
Specific exception: Amount must be grater than 0
```

Dakle dobijemo poruku koju smo sami kontrolirali. Ako vrijednost promijenimo u npr. `100`, greške neće biti. Rada radimo sa iznimka trebamo gledati s jedne strane da sve bude u bloku a s druge da obuhvatimo sve greške. Čak i `var_dump` može potencijalno prikazati grešku. Sa interaceom `Throwables` možemo rukovati s greškom. U uputama možemo vidjeti kako rukovati s potencijalnom greškom.

Ako u klasi `Account` napravimo funkciju `__construct()` koja može generirati neku iznimku.

```
public function __construct(bool $isSavings = false)
{
    if($isSavings){
        throw new Exception("Saving account not supported");
    }
}
```

```
}
```

Distanciramo klasu `$account = new Account();` van `try-catch` bloka i opet sve radi. Naravno jer je to opet happy path. Ako probamo `$account = new Account(true);` isto van `try-catch` bloka pojavit će se greška jer nije obuhvaćeno `try-catch` blokom imat ćemo grešku. Svaki kod treba biti u bloku. Problem je što ne obrađujemo iznimku iako je u bloku. Trebamo staviti `Exception` jer on je roditelj. Možemo dodati još jedan cache blok.

```
try{
    $account = new Account();
    $account->deposit(-100); // ovo ispisuje grešku koja je specifična za ovu iznimku
    // stavili smo taj kod u try block jer je poželjno da bude tu ako imamo kod koji može generirati iznimku
    // $account->deposit(100);
    var_dump($account);
    // ako se dogodila iznimka, tj. PHP ju je uhvatio u objekt $e koji će biti stvoren u specijalnoj iznimci
} catch(AccountException $e){
    echo "Specific exception: " . $e->getMessage();
} catch(Exception $e){
    echo "General exception: " . $e->getMessage();
}
```

Dobit ćemo:

```
object(Account)#1 (1) {
    ["balance": "Account": private]=>
    float(100)
}
Specific exception: Amount must be grater than 0
```

Sada ćemo kod malo doraditi.

Konačna verzija izgleda ovako:

```
<?php
// cilj je imati specifični exception u svom kodu i uhvatiti i obraditi tu iznimku
class AccountException extends Exception{}


class Account{
    private float $balance = 0; // imamo balance koji je 0

    public function __construct(bool $isSavings = false)
    {
        if($isSavings){
            throw new Exception("Saving account not supported");
        }
    }
}
```

```

}

// kad nemamo return dobro je naznačiti što metoda vraća
// tu je to void jer ne vraća ništa - prima amount kao neki float
public function deposit (float $amount) : void{
    // dalje provjeri da li je iznos manji ili jednak 0
    if ($amount <= 0) {
        // ako se dogodi da je amount jednak ili manji 0 onda imamo
        (specifičnu) iznimku
        throw new AccountException("Amount must be grater than 0");
    }
    $this->balance += $amount;
}
}

$account = new Account(); // kreiramo account - dodali smo ga kasnije u try block
$account->deposit(100); // to je happy path +100 = imamo balans 100
// $account->deposit(-100); // depozit ne može biti minus, za to radimo iznimku -
ako ostavimo, baca grešku na ispisu var-dump
var_dump($account); // ispisuje 100


try{
    // ako uklonimo true prikazat će Specific exception
    $account = new Account(true);
    // ovo ispisuje General exception jer se dogodila greška, ako iznad nije true
    $account->deposit(-100);
    // $account->deposit(100); // ovo je happy path
    // stavili smo taj kod u try block jer je poželjno da bude tu ako
    // imamo kod koji može generirati iznimku

    var_dump($account);
    // ako se dogodila iznimka, tj. PHP ju je uhvatio u objekt $e koji će biti
    stvoren u specijalnoj iznimci
} catch(AccountException $e){
    echo "Specific exception: " . $e->getMessage();
} catch(Exception $e){
    echo "General exception: " . $e->getMessage();
}

```

Ako pokrenemo ovaj kod dojavit će:

General exception: Saving account not supported

Ako ovaj red `$account = new Account(true);` prepravimo u `$account = new Account();`; više neće biti General exception ali će se aktivirati `Specific exception`.

`Specific exception: Amount must be grater than 0`

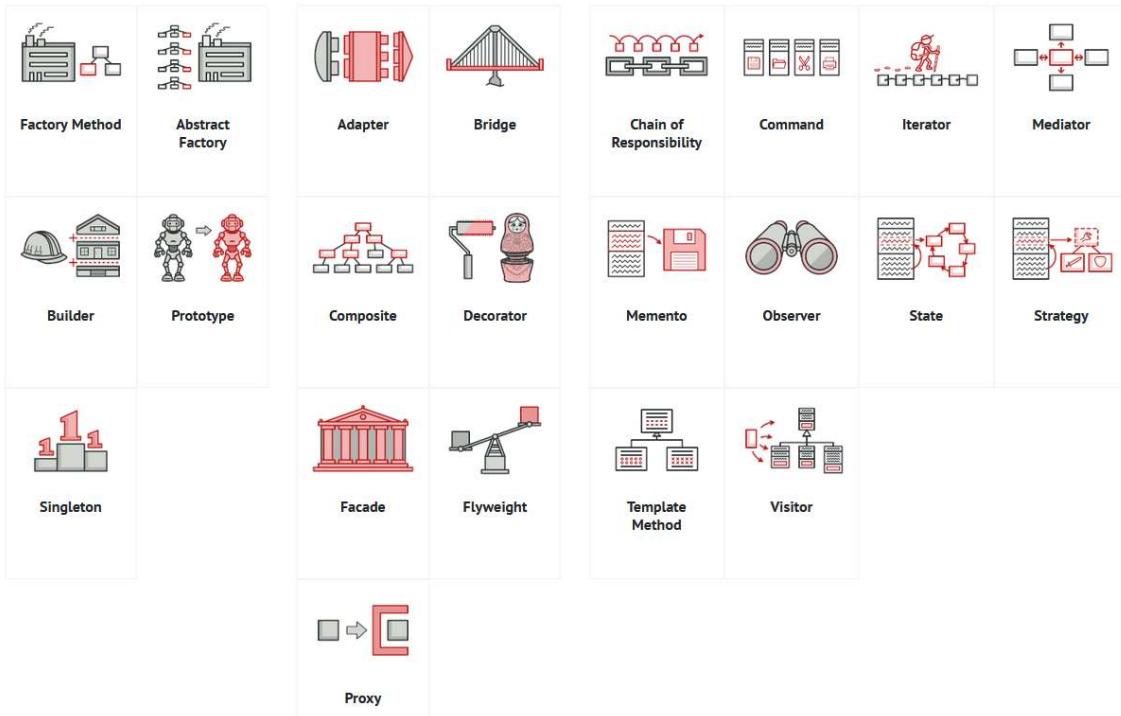
Dizajn obrasci

Što su dizajn obrasci?

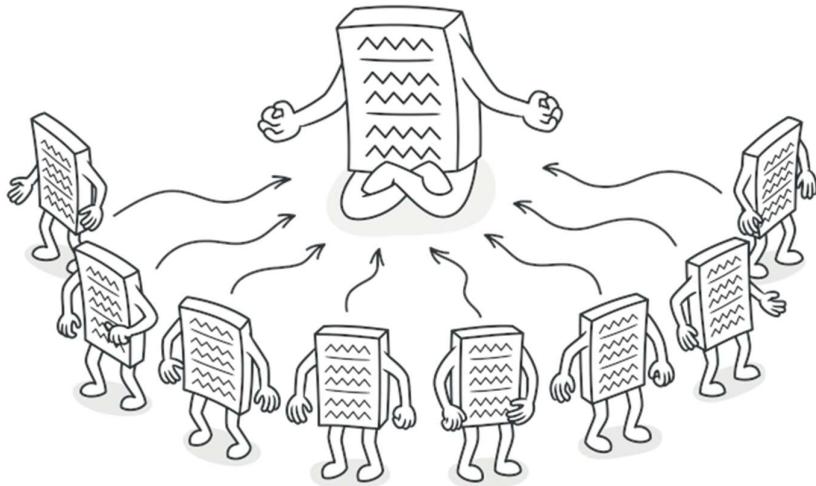
Dizajn obrasci (engl. Design Patterns) su standardizirana rješenja za česte probleme koje susrećemo u dizajnu softvera. Oni su kao predlošci koji se mogu prilagoditi za rješavanje određenih dizajnerskih problema u kodiranju.

Vrste dizajn obrazaca:

- **Kreativni (engl. Creational):** Olakšavaju stvaranje objekata. Primjeri uključuju `Singleton`, `Factory`, `Builder`, `Prototype`. Pružaju mehanizme kreiranja objekata koji povećavaju fleksibilnost i ponovnu upotrebu postojećeg koda.
- **Strukturni (engl. Structural):** Olakšavaju dizajniranje arhitekture aplikacije. Primjeri uključuju `Adapter`, `Bridge`, `Decorator`, `Facade`. Objašnjavaju kako sastaviti objekte i klase u veće strukture, a da te strukture ostanu fleksibilne i učinkovite.
- **Ponašanje (engl. Behavioral):** Usmjereni na komunikaciju između objekata i raspodjelu odgovornosti između njih. Primjeri uključuju `Observer`, `Strategy`, `Command`, `Iterator`.



Singleton obrazac osigurava da klasa ima samo jednu instancu i pruža globalnu točku pristupa toj instanci.



<https://refactoring.guru/design-patterns/singleton>

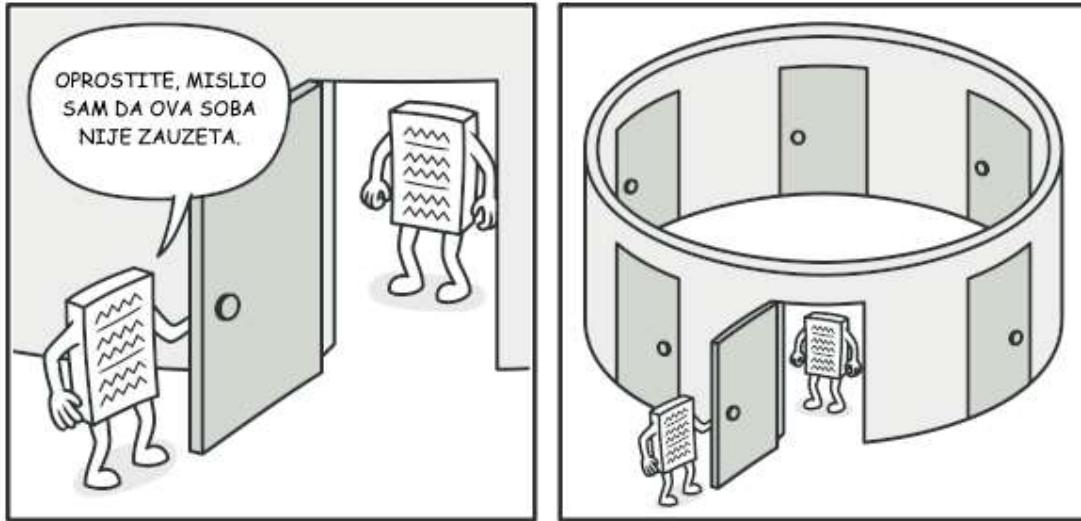
Singleton obrazac rješava dva problema u isto vrijeme, kršeći [Princip jedinstvene odgovornosti \(engl. Single Responsibility Principle\)](#).

1. **Osigurajte da klasa ima samo jednu instancu.** Zašto bi itko želio kontrolirati koliko instanci klasa ima? Najčešći razlog za to je kontrola pristupa nekom zajedničkom resursu—na primjer, bazi podataka ili datoteci.
Evo kako to funkcionira: zamislite da ste stvorili objekt, ali ste nakon nekog vremena odlučili stvoriti novi. Umjesto novog objekta, dobit ćete onaj koji ste već izradili.
Imajte na umu da je ovo ponašanje nemoguće implementirati s običnim konstruktorom budući da poziv konstruktora mora uvijek vratiti novi objekt po dizajnu.
2. **Omogućite globalnu pristupnu točku toj instanci.** Sjećate se onih globalnih varijabli koje ste (u redu, ja) koristili za pohranjivanje nekih bitnih objekata? Iako su vrlo zgodne, također su vrlo nesigurne jer bilo koji kod potencijalno može prebrisati sadržaj tih varijabli i srušiti aplikaciju.

Baš poput globalne varijable, Singleton uzorak vam omogućuje pristup nekom objektu s bilo kojeg mesta u programu. Međutim, također štiti tu instancu od prepisivanja drugim kodom.

Postoji i druga strana ovog problema: ne želite da kôd koji rješava problem broj 1 bude razbacan po vašem programu. Puno je bolje imati ga unutar jedne klase, pogotovo ako ostatak vašeg koda već ovisi o njemu.

U današnje vrijeme, Singleton obrazac je postao toliko popularan da ljudi mogu nešto nazvati *singletonom* čak i ako rješava samo jedan od navedenih problema.



Sve implementacije Singletona imaju ova dva zajednička koraka:

- Napravite zadani konstruktor privatnim kako biste spriječili druge objekte da koriste `new` operator s Singleton klasom.
- Kreirajte statičku metodu stvaranja koja djeluje kao konstruktor. Ispod haube, ova metoda poziva `private` konstruktor da kreira objekt i sprema ga u statičko polje. Svi sljedeći pozivi ovoj metodi vraćaju keširani objekt.

Ako vaš kod ima pristup klasi Singleton, tada može pozvati statičku metodu Singletona. Dakle, kad god se ta metoda pozove, uvijek se vraća isti objekt.

Vlada je izvrstan primjer Singleton obrasca. Država može imati samo jednu službenu vladu. Bez obzira na osobne identitete pojedinaca koji formiraju vlade, naslov "Vlada X" je globalna pristupna točka koja identificira grupu ljudi koji su nadležni.

Evo kako se implementira Singleton obrazac u PHP-u:

```
php
class Singleton {
    private static $instance = null;

    private function __construct() {
        // Privatni konstruktor sprječava direktno instanciranje klase
    }

    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }
}
```

```
// Ostale metode i svojstva klase
}

// Korištenje Singleton obrasca
$instance1 = Singleton::getInstance();
$instance2 = Singleton::getInstance();

// $instance1 i $instance2 će biti ista instanca
```

Svojstva

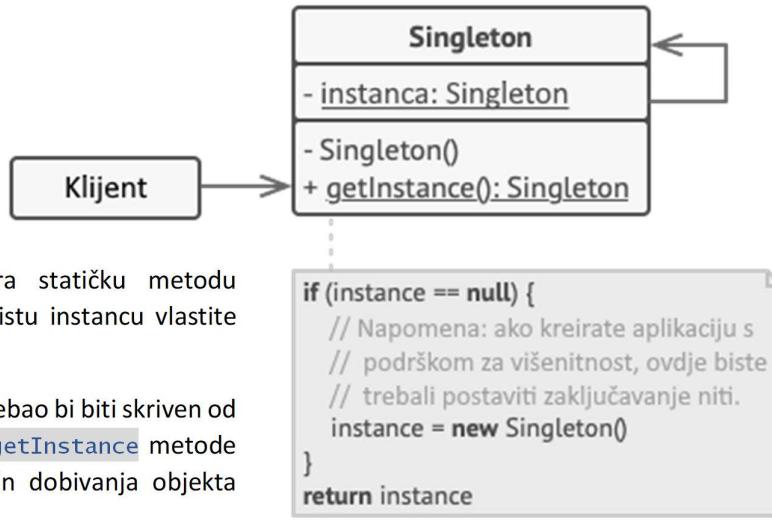
Privatni konstruktor: Onemogućava direktno instanciranje klase iz vanjskog koda.

Statična privatna varijabla instance: Čuva referencu na jedinstvenu instancu klase.

Statička metoda `getInstance()`: Pruža globalnu pristupnu točku za dohvaćanje instance klase. Ako instance ne postoji, ona se kreira i sprema u statičku varijablu.

Jedina instanca: Pozivi `getInstance()` uvijek vraćaju istu instancu klase.

Struktura



Klasa **Singleton** deklara statičku metodu `getInstance()` koja vraća istu instancu vlastite klase.

Konstruktor **Singletona** trebao bi biti skriven od koda klijenta. Pozivanje `getInstance()` metode trebao bi biti jedini način dobivanja objekta Singleton.

U ovom primjeru, klasa povezivanja baze podataka djeluje kao **Singleton**. Ova klasa nema javni konstruktor, tako da je jedini način da dobijete njen objekt da pozovete metodu `getInstance()`. Ova metoda kešira prvi kreirani objekt i vraća ga u svim sljedećim pozivima.

```
class Database {
private static $instance = null;
private function __construct() {
// Inicijalizacija konekcije na bazu
}
```

```
public static function getInstance() {  
    if (self::$instance == null) {  
        self::$instance = new Database();  
    }  
    return self::$instance;  
}  
  
public function query($sql) {  
    // Izvršavanje SQL upita  
}  
}  
  
// Primjer korištenja  
$db = Database::getInstance();  
$db->query("SELECT * FROM users");
```

Primjenjivost

Koristite Singleton obrazac kada klasa u vašem programu treba imati samo jednu instancu dostupnu svim klijentima; na primjer, jedan objekt baze podataka koji dijele različiti dijelovi programa.

Singleton obrazac onemogućuje sve druge načine kreiranja objekata klase osim posebne metode kreiranja. Ova metoda stvara novi objekt ili vraća postojeći ako je već stvoren.

Upotrijebite Singleton obrazac kada vam je potrebna stroža kontrola nad globalnim varijablama.

Za razliku od globalnih varijabli, Singleton obrazac jamči da postoji samo jedna instanca klase. Ništa, osim same klase Singleton, ne može zamijeniti keširanu instancu.

Imajte na umu da uvijek možete podesiti ovo ograničenje i dopustiti stvaranje bilo kojeg broja instanci Singletona. Jedini dio koda koji treba promijeniti je tijelo `getInstance` metode.

Kako implementirati

1. Dodajte privatno statičko polje u klasu za pohranu pojedinačne instance.
2. Deklarirajte javnu statičku metodu kreiranja za dobivanje (engl. getting) pojedinačne instance.
3. Implementirajte „lijenu inicijalizaciju“ unutar statičke metode. Trebao bi stvoriti novi objekt kod prvog poziva i staviti ga u statičko polje. Metoda bi uvijek trebala vratiti tu instancu kod svih sljedećih poziva
4. Napravite konstruktor klase privatnim. Statička metoda klase i dalje će moći pozvati konstruktor, ali ne i druge objekte.
5. Pregledajte kod klijenta i zamijenite sve direktnе pozive konstruktoru singletona pozivima njegove statičke metode kreiranja.

Za i protiv

Za	Protiv
Možete biti sigurni da klasa ima samo jednu instancu.	Krši Princip jedinstvene odgovornosti (engl. Single Responsibility Principle). Obrazac rješava dva problema u isto vrijeme.
Dobivate globalnu pristupnu točku toj instanci.	Singleton obrazac može prikriti loš dizajn, npr. kada komponente programa znaju previše jedna o drugoj.
Objekt singleton inicijalizira se samo kada se zatraži prvi put (engl. lazy initialization).	Uzorak zahtijeva poseban tretman u višenitnom okruženju kako više niti ne bi nekoliko puta stvorilo pojedinačni objekt.
	Može biti teško napraviti jedinično testiranje (engl. unit test) klijentskog koda Singletona jer se mnogi testne radne okoline oslanjaju na nasljeđivanje kada proizvode lažne objekte. Budući da je konstruktor klase singleton privatni a nadjačavanje statičkih metoda je ipak potrebno znati, morat ćete smisliti kreativan način da se ismijavate singlonu. Ili jednostavno nemoj pisati testove. Ili nemojte koristiti uzorak Singleton.

Singleton ima gotovo iste prednosti i nedostatke kao globalne varijable. Iako su super zgodni, razbijaju modularnost vašeg koda.

Ne možete samo koristiti klasu koja ovisi o Singlonu u nekom drugom kontekstu, bez prenošenja Singlonu u drugi kontekst. Većinu vremena ovo se ograničenje pojavljuje tokom izrade jediničnih testova.

Idejni (konceptualan) primjer

Ovaj primjer ilustrira strukturu uzorka dizajna Singleton i fokusira se na sljedeća pitanja:

- Od kojih se klasa sastoji?
- Koje uloge imaju te klase?
- Na koji su način povezani elementi uzorka?

Realan primjer

Singleton obrazac je poznat po tome što ograničava ponovnu upotrebu koda i komplikira testiranje jedinice (engl. unit testing). Međutim, još uvijek je vrlo koristan u nekim slučajevima. Posebno je praktičan kada trebate kontrolirati neke zajedničke resurse. Na primjer, objekt globalnog evidentiranja (engl. logging) koji mora kontrolirati pristup log datoteci. Još jedan dobar primjer: zajedničko pohranjivanje konfiguracije vremena izvođenja.

```
<?php

namespace RefactoringGuru\Singleton\RealWorld;

/**
```

```
* Ako trebate podržati nekoliko vrsta Singletona u svojoj aplikaciji, možete
* definirati osnovne značajke Singletona u osnovnoj klasi, dok stvarnu
* poslovnu logiku (kao što je sakupljanje podataka (engl. logging))
* premještate u podklase.
*/
class Singleton
{
    /**
     * Stvarna instanca singletona gotovo se uvijek nalazi unutar statičkog polja.
     * U ovom slučaju, statičko polje je matrica, gdje svaka podklasa
     * singltona pohranjuje vlastitu instancu.
     */
    private static $instances = [];

    /**
     * Singletonov konstruktor ne bi trebao biti javan. Međutim,
     * ne može biti ni privatno ako želimo dopustiti podklase.
     */
    protected function __construct() { }

    /**
     * Kloniranje i deserializacija nisu dopušteni za singleton-ove.
     */
    protected function __clone() { }

    public function __wakeup()
    {
        throw new \Exception("Nije moguće deserializirati singleton");
    }

    /**
     * Metoda koju koristite za dobivanje (engl. get) Singleton instance.
     */
    public static function getInstance()
    {
        $subclass = static::class;
        if (!isset(self::$instances[$subclass])) {
            // Primijetite da ovdje koristimo ključnu riječ "static" umjesto
            // stvarnog naziva klase. U ovom kontekstu ključna riječ "static"
            // znači "ime trenutne klase". Taj je detalj važan jer kada se
            // metoda pozove na podklasi, želimo da se ovdje kreira instanca
            // te podklase.
        }
    }
}
```

```

        self::$instances[$subclass] = new static();
    }
    return self::$instances[$subclass];
}

/**
 * Klasa za sakupljanje podataka (engl. logging class) je najpoznatija i naj-
 * hvaljenija upotreba Singleton obrasca. U većini slučajeva, potreban vam je
 * jedan objekt za sakupljanje podataka (engl. Logging object) koji zapisuje
 * u jednu log datoteku (kontrola nad zajedničkim resursom). Također vam je
 * potreban prikidan način za pristup toj instanci iz bilo kojeg konteksta
 * vaše aplikacije (globalna pristupna točka).
 */
class Logger extends Singleton
{
    /**
     * Pokazivač datoteke resursa log datoteke.
     */
    private $fileHandle;

    /**
     * Budući da se konstruktor Singletona poziva samo jednom, uvijek je
     * otvoren samo jedan resurs datoteke.
     *
     * Napomena, radi jednostavnosti, ovdje otvaramo tok konzole (engl.console
     * stream) umjesto stvarne datoteke.
     */
    protected function __construct()
    {
        $this->fileHandle = fopen('php://stdout', 'w');
    }

    /**
     * Zapiši log unos u resurs otvorene datoteke.
     */
    public function writeLog(string $message): void
    {
        $date = date('d.m.Y.');
        fwrite($this->fileHandle, "$date: $message\n");
    }

}

```

```

        * Samo zgodna prečica za za smanjenje količine koda potrebnog za
        * bilježenje poruka iz koda klijenta.
        */
    public static function log(string $message): void
    {
        $logger = static::getInstance();
        $logger->writeLog($message);
    }
}

/**
 * Primjena Singleton uzorka na pohranu konfiguracije također je uobičajena
 * praksa. Često trebate pristupiti konfiguracijama aplikacije s mnogo
 * različitih mesta u programu. Singleton vam pruža tu udobnost.
 */
class Config extends Singleton
{
    private $hashmap = [];

    public function getValue(string $key): string
    {
        return $this->hashmap[$key];
    }

    public function setValue(string $key, string $value): void
    {
        $this->hashmap[$key] = $value;
    }
}

/**
 * Kod klijenta.
 */
Logger::log("Početak!");

// Usporedite vrijednosti Logger singleton.
$11 = Logger::getInstance();
$12 = Logger::getInstance();
if ($11 === $12) {
    Logger::log("Logger ima jednu instancu.");
} else {
    Logger::log("Logger-i su različiti.");
}

```

```
// Provjerite kako Config singleton sprema podatke...
$config1 = Config::getInstance();
$login = "test_login";
$password = "test_password";
$config1->setValue("login", $login);
$config1->setValue("password", $password);
// ...i vrati ih.
$config2 = Config::getInstance();
if ($login == $config2->getValue("login") &&
    $password == $config2->getValue("password"))
{
    Logger::log("Config singleton također radi dobro.");
}

Logger::log("Završeno!");
```

Rezultat je:

```
25.06.2024.: Početak!
25.06.2024.: Logger ima jednu instancu.
25.06.2024.: Config singleton također radi dobro.
25.06.2024.: Završeno!
```

Facade obrazac

Fasada je strukturalni dizajn koji pruža pojednostavljen interface biblioteci (engl. library), framework-u ili bilo kojem drugom složenom skupu klasa.

Problem i rješenje

Zamislite da morate napraviti da vaš kod radi sa širokim skupom objekata koji pripadaju sofisticiranoj biblioteci ili framework-u. Obično biste morali inicijalizirati sve te objekte, pratiti ovisnosti, izvršiti metode ispravnim redoslijedom i tako dalje.

Kao rezultat toga, poslovna logika vaših klasa postala bi usko povezana s pojedinostima implementacije klasa trećih strana, što bi je činilo teškom za razumijevanje i održavanje.

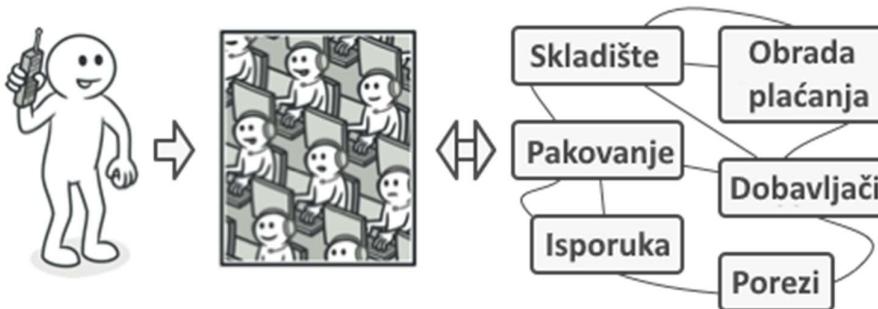
Fasada je klasa koja pruža jednostavan inteface složenom podsustavu koji sadrži mnogo pokretnih dijelova. Fasada može pružiti ograničenu funkcionalnost u usporedbi s izravnim radom s podsustavom. Međutim, uključuje samo one značajke do kojih je klijentima stvarno stalo.

Imati fasadu je zgodno kada trebate integrirati svoju aplikaciju sa sofisticiranom bibliotekom koja ima desetke svojstava, ali trebate samo maleni djelić njezine funkcionalnosti.

Na primjer, aplikacija koja prenosi kratke smiješne videozapise s mačkama na društvene medije potencijalno bi mogla koristiti profesionalnu biblioteku za konverziju videozapisa. Međutim, sve što

stvarno treba je klasa s jednom metodom encode(filename, format). Nakon što kreirate takvu klasu i povežete je s bibliotekom za video konverziju, imat ćeće svoju prvu fasadu.

Analogija u stvarnom svijetu

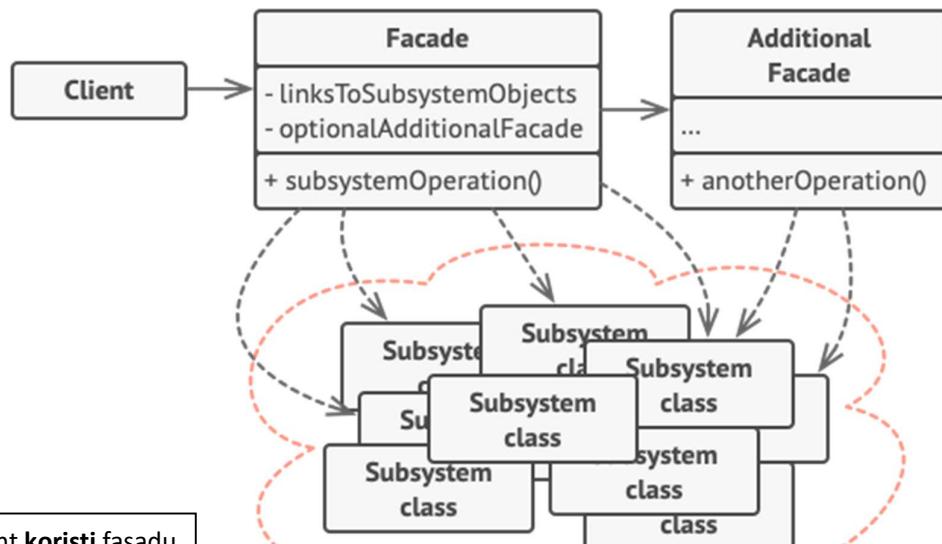


Kada nazovete trgovinu da naručite telefonom, operater je vaša fasada prema svim službama i odjelima trgovine. Operater vam nudi jednostavan glasovni interface za sistem naručivanja, pristupnike (engl. gateway) plaćanja i razne usluge dostave.

Struktura

Fasada pruža praktičan pristup određenom dijelu funkcionalnosti podsustava. Zna kamo usmjeriti zahtjev klijenta i kako upravljati svim pokretnim dijelovima.

Dodatna klasa **fasada** može se stvoriti kako bi se spriječilo onečišćenje jedne fasade nepovezanim svojstvima koje bi je moglo učiniti još jednom složenom strukturu. Dodatne fasade mogu koristiti i klijenti i druge fasade.



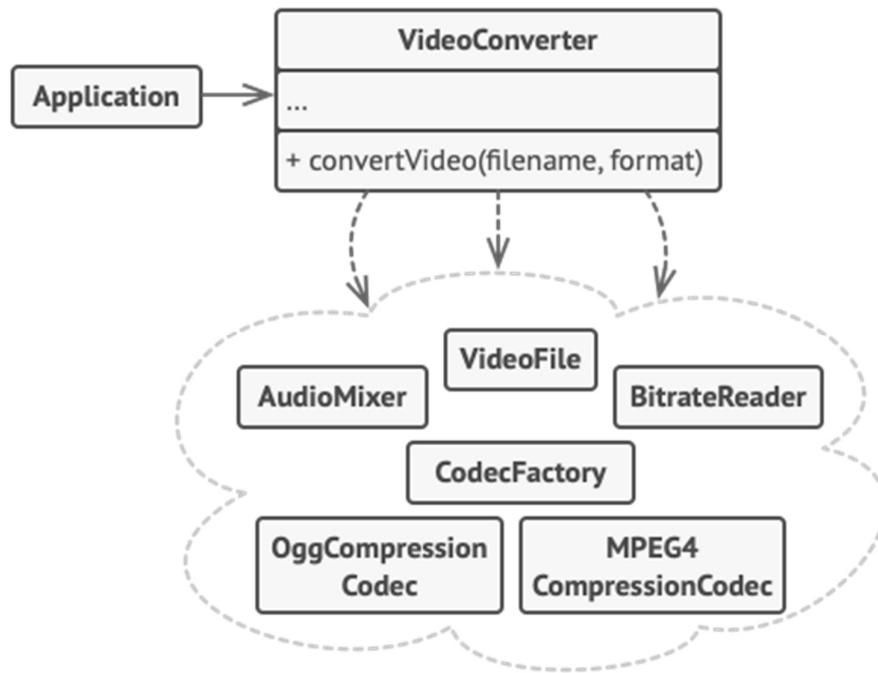
Klijent koristi fasadu umjesto direktnog pozivanja objekata podsistema.

Složeni **podsistem** sastoji se od desetaka različitih objekata. Kako bi svi učinili nešto smisleno, morate duboko zaroniti u detalje implementacije podsustava, kao što je inicijaliziranje objekata ispravnim redoslijedom i opskrbljivanje podacima u ispravnom formatu.

Klase podsustava nisu svjesne postojanja fasade. Djeluju unutar sustava i izravno surađuju jedni s drugima.

Pseudokod

U ovom primjeru, Facade uzorak pojednostavljuje interakciju sa složenim okvirom video konverzije.



Primjer izoliranja višestrukih ovisnosti unutar jedne klase fasade

Umjesto da vaš kod direktno radi s desecima klasa redne okoline, stvarate klasu fasade koja enkapsulira tu funkcionalnost i skriva je od ostatka koda. Ova struktura vam također pomaže smanjiti napor oko nadogradnje na buduće verzije okvira ili njegove zamjene drugim. Jedina stvar koju biste trebali promijeniti u svojoj aplikaciji bila bi implementacija metoda fasade.

Primjenjivost

Kako implementirati

Za i protiv

SOLID Principi

- **S: Single Responsibility Principle (SRP)**
 - Klasa bi trebala imati samo jedan razlog za promjenu, što znači da bi trebala imati samo jednu zadaću ili odgovornost.
- **O: Open/Closed Principle (OCP)**
 - Softver bi trebao biti otvoren za proširenje, ali zatvoren za izmjenu.
- **L: Liskov Substitution Principle (LSP)**

- Objekti u programu bi trebali biti zamjenjivi sinstancama njihovih podtipova bez utjecaja na točnost programa.
- **I: Interface Segregation Principle (ISP)**
 - Niti jedan klijent ne bi trebao biti prisiljen ovisiti o metodama koje ne koristi.
- **D: Dependency Inversion Principle (DIP)**
 - Visokorazinski moduli ne bi trebali ovisiti o niskorazinskim modulima. Oba bi trebala ovisiti o apstrakcijama.

Single Responsibility Principle

Princip jedinstvene odgovornosti (engl. Single Responsibility Principle) navodi da bi svaka klasa trebala imati samo jednu odgovornost ili razlog za promjenu.

Ovo znači da bi klasa trebala biti fokusirana na jednu zadaću ili funkcionalnost.

User klasa samo upravlja korisničkim podacima, dok EmailService klasa upravlja slanjem e-pošte.

```
// Loš pristup: Klasa "User" upravlja korisničkim podacima i logikom slanja e-pošte
class User {
    public $email;
    public $username;
    public function sendEmail($content) {
        // Logika za slanje e-pošte
    }
}
// Bolji pristup: Razdvajanje odgovornosti
class User {
    public $email;
    public $username;
}
class EmailService {
    public function sendEmail($user, $content) {
        // Logika za slanje e-pošte
    }
}
```

Open/Closed Principle

Princip otvorenosti/zatvorenosti kaže da bi software entiteti (klase, moduli, funkcije itd.) trebali biti otvoreni za proširenje, ali zatvoreni za izmjene.

To znači da bi trebali moći dodati nove funkcionalnosti bez mijenjanja postojećeg koda.

PaymentProcessor klasa je zatvorena za izmjene ali otvorena za proširenje kroz implementaciju PaymentGateway interfejsa.

```
interface PaymentGateway {
    public function processPayment($amount);
}
```

```

class PaypalPayment implements PaymentGateway {
    public function processPayment($amount) {
        // Proces plaćanja preko PayPal-a
    }
}

class CreditCardPayment implements PaymentGateway {
    public function processPayment($amount) {
        // Proces plaćanja kreditnom karticom
    }
}

class PaymentProcessor {
    private $paymentGateway;
    public function __construct(PaymentGateway $paymentGateway) {
        $this->paymentGateway = $paymentGateway;
    }
    public function pay($amount) {
        $this->paymentGateway->processPayment($amount);
    }
}

```

Liskov Substitution Principle

Liskovljev princip supstitucije kaže da bi objekti programa trebali biti zamjenjivi s objektima njihovih podtipova bez utjecaja na točnost programa.

To znači da bi derivirani tipovi trebali moći zamijeniti svoje bazne tipove.

U ovom primjeru, korištenje **Noj** klase može izazvati probleme jer nojevi ne mogu letjeti.

```

class Ptica {
    public function leti(){
        echo "Leti";
    }
}

class Patka extends Ptica {}

class Noj extends Ptica {
    public function leti() {
        throw new Exception("Ne može letjeti");
    }
}

```

Ovaj dizajn krši LSP jer kada instanciramo objekt iz klase **Noj** dobijemo objekt a kada pozovemo metodu iz tog objekta **leti()**, unutar metode u klasi Noj, throw omogući ovoj metodi da izbaci iznimku. U primjeru dalje nije obrađena ali objekt nije zamjenjiv s njihovim podtipovima jer utječe na točnost programa.

Interface Segregation Principle

Princip segregacije interface-a sugerira da klijenti ne bi trebali biti prisiljeni ovisiti o interface-ima koja ne koriste.

To znači da bi interface-i trebali biti specifični, a ne generički.

Ovaj dizajn krši ISP jer **RobotWorker** ne treba `eat` metodu. Bolje bi bilo razdvojiti ove interface.

```

interface Workable {
    public function work();
    public function eat();
}

class HumanWorker implements Workable {
    public function work() {
        // radi
    }
    public function eat() {
        // jede
    }
}

class RobotWorker implements Workable {
    public function work() {
        // radi
    }
    public function eat() {
        // Ova funkcija ne treba robotu
    }
}

```

Dependency Inversion Principle

Princip inverzije ovisnosti navodi da bi visoko razinski moduli ne trebali ovisiti o nisko razinski modulima. Oba bi trebala ovisiti o apstrakcijama.

FileManager ne ovisi direktno o **DatabaseStorage**, već o **Storage** interface-u, što omogućava fleksibilnost i manju povezanost komponenti.

Dependency Injection je kada imamo dva objekta a gdj jedan objekt injektiramo u drugi objekat ili kroz konstruktor. Moguće je to raditi i automatski. Logika je da imate jedan objekt koji želite koristiti unutar drugog objekta.

```

interface Storage {
    public function save($data);
}

class DatabaseStorage implements Storage {
    public function save($data) {
        // Spremi podatke u bazu
    }
}

class FileManager {
    private $storage;
    // Storage $storage je dependency injection
    public function __construct(Storage $storage) {
        $this->storage = $storage;
    }
}

```

```

    }
    public function saveFile($data) {
        $this->storage->save($data);
    }
}

```

Spajanje PHP s SQL bazama

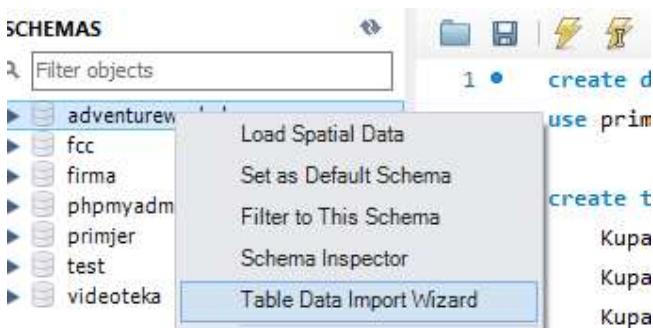
U PHP je moguće se spojiti na baze s MySQLi ili PDO_MySQL načinima. Prije, do PHP 7.0.0. postojao je `I MySQL_connect` koji više nije podržan. `mysqli_connect` oslanja se isključivo na MySQL i objektno je orijentiran. PDO_MySQL klasa reprezentira konekciju između PHP i database. Mi ćemo raditi PDO_MySQL. Može biti pitanje kako se konektirati sa `mysqli_connect` ali predavač to nije objasnio.

Za primjere ćemo koristiti `adventureworkshop`. `username` je `root` a `password` nema.

Izabrat ćemo najlakši način da se spojimo.

```
$pdo = new PDO("mysql:host=localhost;dbname=adventureworkshop; charset=utf8",
    "root", "", [PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_OBJ,
]);
```

Dakle otvaramo novi PDO objekt. Kako se radi o localhostu to i unosimo. Ime baze je `adventureworkshop`. Kodna stranica je UTF8. Nju možemo provjeriti iz MySQL Workbench-a, tako da na imenu baze podataka odaberemo Schema Inspector.



```
<?php

$pdo = new PDO("mysql:host=localhost;dbname=adventureworkshop;
charset=utf8", "root", "");

$stmt = $pdo->query("SELECT * FROM proizvod");
$res = $stmt->fetchAll(PDO::FETCH_ASSOC);
var_dump($res);
```

Prvo pripremamo izjavu (engl. statement). To je nekakav upit (engl. query), najčešće nekakav SQL upit. Ovaj upit se automatski priprema i izvršava i zbog toga je potencijalno problematičan. Generalno, nije potrebno raditi izvršavanje (engl. execute). query izvršava SQL naredbu, vraćajući skup rezultata kao PDOStatement objekt.

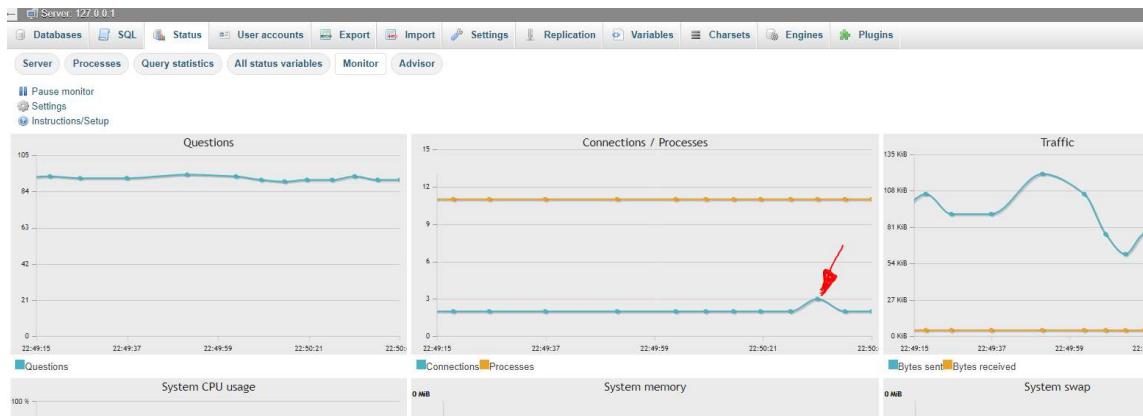
```
$stmt = $pdo->query("SELECT * FROM proizvod");
$res = $stmt->PDO::query
var_dump($res)
(PHP 5 >= 5.1.0, PHP 7, PHP 8, PECL pdo >= 0.2.0) Executes an SQL statement, returning a result set as a PDOStatement object
<?php
public function query(string $query, int|null $fetchMode = null, mixed ...$fetchModeArgs) { }

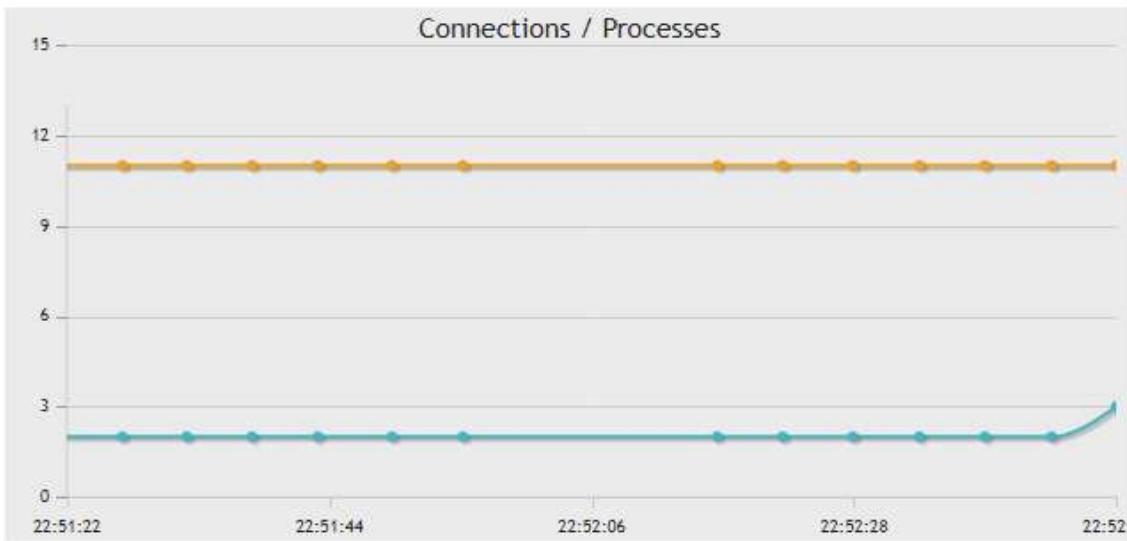
@param string $query
@param int|null $fetchMode
@param array ...$fetchModeArgs
@return \PDOStatement|false
PDO::query returns a PDOStatement object, or FALSE on failure.
```

Preko tog `$stmt` objekta kažemo `fetchAll` i dohvativamo matricu koja sadrži sve redove skupa rezultata.

Objasnit ćemo i čemu služi `PDO::FETCH_ASSOC`. Određuje da će metoda dohvaćanja vratiti svaki red kao polje indeksirano nazivom stupca kako je vraćeno u odgovarajućem skupu rezultata. Ako skup rezultata sadrži više stupaca s istim imenom, `PDO::FETCH_ASSOC` vraća samo jednu vrijednost po nazivu stupca.

Imamo funkcionalni dio koji možemo pokrenuti. Otvorit ćemo phpMyAdmin (localhost:8081/phpmyadmin u mom slučaju), otici na Status i na Monitor. Tu možemo pratiti da kada pokrenemo program s kodom iznad da broj konekcija skoči:





Nedostatak upita na PDO je mogući SQL injection jer osim pripreme i izvrši SQL upit. Npr. može napraviti `drop` tablice ili ovo:

```
$id = 1;
$stmt = $pdo->query("SELECT * FROM proizvod where IDProizvod = $id");
```

Na taj način dohvati prvi podatak:

```
array(1) {
  [0]=>
  array(7) {
    ["IDProizvod"]=>
    int(1)
    ["Naziv"]=>
    string(15) "Adjustable Race"
    ["BrojProizvoda"]=>
    string(7) "AR-5381"
    ["Boja"]=>
    NULL
    ["MinimalnaKolicinaNaSkladistu"]=>
    int(650)
    ["CijenaBezPDV"]=>
    string(4) "0.00"
    ["PotkategorijaID"]=>
    NULL
  }
}
```

Na ovaj način smo direktno injektirali (ubrzgali) podatak iz varijable u SQL, upit nema provjeru i poveže (engl. bind) vrijednost iz varijable s SQL-om preko parametara. Da smo rekli:

```
$id = "1 OR 1=1"
```

Na ovaj način možemo dobiti sve proizvode. Bolje je koristiti `prepare` umjesto `query`. Kada koristimo `prepare` nećemo ubrizgavati (injektirati) direktno podatke iz varijable nego postoje placeholder-i. Postoje pozicijski placeholder-i (upitnici, bitno je da ispravno povežete s tim upitnicima po pozicijama) i imenovani placeholder-i (ime vežemo s podatkom). Uglavnom se koriste pozicijski placeholder-i. Kod imenovanih smo ovisni o imenu, ne možemo zamjeniti pozicije.

Kako ćemo uglavnom dalje koristiti pozicijske placeholder-e, samo ćemo spomenuti da oni koriste jedinstveno ime za označavanje mesta gdje će se zamjeniti vrijednost. Ime počinje s dvotočkom (`:`) i može sadržavati slova, brojeve i donje crtice.

```
$stmt = $pdo->prepare("SELECT * FROM users WHERE id = :user_id AND email =
:email");
$stmt->bindValue(':proizvod_id', $aktiv);
$stmt->bindValue(':email', $email);
$stmt->execute();
```

Vratimo se na primjer. Ovdje trebamo napraviti `execute` i prilikom izvršavanja šaljemo matricu podataka placeholder-ima. Ovdje se događa tzv. **bind**. PDO ima čak mogućnost `bindColumn` (veže se za pozicijske parametre), `bindValue` (veže se za imenovane parametre), `bindParam`, . Kod `execute` nije potrebno koristiti dodatnu metodu iako je moguće vezati vrijednost i parametrima. Ako direktno u `prepare` gurnemo vrijednost isto će biti problem. Sama `prepare` metoda ne sprječava da propustite SQL injection. Ako na temelju placeholder-a prilikom `execute` povežemo vrijednost (engl. value bind), tada će PDO ispod haube odraditi sanitizaciju.

PDO statement ponovo na sebi ima metode `fetch`, `fetchAll` (različite kombinacije `PDO::FETCH_DEFAULT`, `PDO::FETCH_COLUMN`, `PDO::FETCH_CLASS`), `FETCH_OBJ`, itd.

Od dohvata podataka želimo dobiti asocijativnu matricu.

```
<?php

$pdo = new PDO("mysql:host=localhost;dbname=adventureworkshop;
charset=utf8", "root", "");
$id = "1 OR 1=1";
$stm = $pdo->prepare("SELECT * FROM proizvod where IDProizvod = ?");
$stm->execute([$id]);
$res = $stm->fetchAll(PDO::FETCH_ASSOC);
var_dump($res);
```

S ovime ćemo dobiti prvi proizvod.

```
array(1) {
[0]=>
array(7) {
["IDProizvod"]=>
int(1)
["Naziv"]=>
string(15) "Adjustable Race"
```

```
["BrojProizvoda"]=>
string(7) "AR-5381"
["Boja"]=>
NULL
["MinimalnaKolicinaNaSkladistu"]=>
int(650)
["CijenaBezPDV"]=>
string(4) "0.00"
["PotkategorijaID"]=>
NULL
}
```

Ako umjesto `PDO::FETCH_ASSOC` stavimo `PDO::FETCH_OBJ` nećemo kao rezultat dobiti asocijativnu matricu nego `stdClass` objekt u kojem se nalazi matrica objekata:

```
array(1) {
[0]=>
object(stdClass)#3 (7) {
    ["IDProizvod"]=>
    int(1)
    ["Naziv"]=>
    string(15) "Adjustable Race"
    ["BrojProizvoda"]=>
    string(7) "AR-5381"
    ["Boja"]=>
    NULL
    ["MinimalnaKolicinaNaSkladistu"]=>
    int(650)
    ["CijenaBezPDV"]=>
    string(4) "0.00"
    ["PotkategorijaID"]=>
    NULL
}
}
```

Predavač kaže da mu je kasnije lakše manipulirati sa asocijativnom matricom, to je stvar osobnog izbora.

Da ne bi morali svaki puta ponavljati red :

```
$res = $stm->fetchAll(PDO::FETCH_OBJ);
```

ili

```
$res = $stm->fetchAll(PDO::FETCH_ASSOC);
```

u red gdje smo definirali PDO možemo na kraju dodati još jedan parametar.

To je matrica nekakvih opcija. Kada to dodamo, možemo maknuti podatke sa `fetchAll`.

```
<?php

$pdo = new PDO("mysql:host=localhost;dbname=adventureworkshop;
charset=utf8", "root", "", [
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_OBJ
]);
$id = "1 OR 1=1";
$stm = $pdo->prepare("SELECT * FROM proizvod where IDProizvod = ?");
$stm->execute([$id]);
$res = $stm->fetchAll(); //dohvaća sve zapise
var_dump($res);
```

Sada možemo kontrolirati PDO.

Moguća sintaksa mogla je biti:

```
$stm = $pdo->prepare("SELECT * FROM proizvod where IDProizvod = ? AND Aktivan =
?");
$stm->execute([$id, 1]);
```

Kako ovdje imamo dva placeholder-a, svaki podatak ulazi na svoje mjesto jer je pozicijski.

`fetch` dohvaca prvi zapis a `fetchAll` dohvaca sve zapise. Predavač kaže da uvijek koristi `fetchAll`. `fetchAll` uvijek gradi asocijativnu matricu, iako rezultat može biti samo jedan element.

```
array(1) {
    [0]=>
    object(stdClass)#3 (7) {
        ["IDProizvod"]=>
        int(1)
        ["Naziv"]=>
        string(15) "Adjustable Race"
        ["BrojProizvoda"]=>
        string(7) "AR-5381"
        ["Boja"]=>
        NULL
        ["MinimalnaKolicinaNaSkladistu"]=>
        int(650)
        ["CijenaBezPDV"]=>
        string(4) "0.00"
        ["PotkategorijaID"]=>
        NULL
    }
}
```

Međutim, ako koristimo `fetch`, uvijek će se vratiti samo prvi zapis. To znači da ako ih dohvate više, ispisat će se samo jedan zapis. Da, ovo je bio happy path. `try-cache` blokova nema, a trebalo bi ih biti.

```
object(stdClass)#3 (7) {
    ["IDProizvod"]=>
        int(1)
    ["Naziv"]=>
        string(15) "Adjustable Race"
    ["BrojProizvoda"]=>
        string(7) "AR-5381"
    ["Boja"]=>
        NULL
    ["MinimalnaKolicinaNaSkladistu"]=>
        int(650)
    ["CijenaBezPDV"]=>
        string(4) "0.00"
    ["PotkategorijaID"]=>
        NULL
}
```

SQL ne želimo pisati ručno i želimo imati funkcionalnosti u radu s bazom. Sve exception-e želimo kontrolirati na jednom mjestu. Zato ćemo napisati funkcionalnost gdje želimo upravljati greškama i napisat ćemo funkcionalnost za dohvaćanje podataka iz baze i zapisivanje podataka. Napraviti ćemo ORM. ORM, ili Object Relational Mapper, je dio softvera dizajniran za prevođenje između prikaza podataka koje koriste baze podataka i onih koji se koriste u objektno orijentiranom programiranju. Naravno nećemo pisati novi Laravel.

Iz perspektive razvojnog programera, ORM vam omogućava rad s podacima koji se temelje na bazi podataka koristeći iste objektno orijentirane strukture i mehanizme koje biste koristili za bilo koju vrstu internih podataka. ORM-ovi omogućavaju da se ne morate oslanjati na posebne tehnike ili rukovati s SQL-om kako biste bili produktivni sa svojim podacima.

Općenito, ORM-ovi služe kao sloj apstrakcije između aplikacije i baze podataka. Pokušavaju povećati produktivnost programera uklanjanjem potrebe za standardnim kodom i izbjegavanjem upotrebe neugodnih tehnika.

Imat ćemo kako kaže predavač „code first pristup“. Taj pojam nije uobičajen u kontekstu PHP-a. Umjesto toga, u PHP-u se češće koristi pristup „database first“ ili „model first“ pri radu s bazama podataka. Međutim, postoji sličan koncept u PHP-u koji se naziva "Active Record" ili "Object-Relational Mapping" (ORM). Ovo je pristup gdje se objekti u kodu preslikavaju na tablice u bazi podataka, a interakcija s bazom podataka se obavlja kroz te objekte, umjesto direktno pisanjem SQL upita.

Mi nećemo pisati direktno SQL u prepare izrazu. Želimo to olakšati tako da kreiramo klasu preko koje možemo stvoriti objekt i onda preko te klase dohvatiti podatke. Poanta je da kako pišemo 3,4 linije

koda i to moramo ponavljati više puta i naravno održavati. Probati ćemo to staviti u klase i malo to proširiti na idućem projektu (MVC pattern). Time ćemo dobiti intro u Laravel i time ćemo vidjeti benefite framework-a.

Isprogramirati ćemo funkcionalnosti koje nisu usko vezane za našu aplikaciju. To možemo koristiti na svim aplikacijama kojim želimo. Poanta je korištenja framework-a je da to ima gotovo i da to samo koristimo. Cilj ovoga je dobiti uvid kako to radi, kako bi kada dođemo na Laravel, razumijemo koncepte. Predavač želi da se shvate principi.

Pogledat ćemo prvo konekcije na bazu bez singleton obrasca u Database.php datotekci.

Unutar `__construct` funkcije napisat ćemo:

```
$dsn = "mysql:host=localhost;dbname=adventureworkshop;charset=utf8";
        $username = "root";
        $password = "";
        $options = [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        ];

        try {
            $this->connection = new PDO($dsn, $username, $password, $options);
        } catch (PDOException $e) {
            die("Connection failed: " . $e->getMessage());
        }
    }
```

Za razliku od prije, ubacili smo i `PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION`. Želimo da greške koje se dogode kod dohvata podataka (sve radimo sa PDO), želimo da baca iznimke ako dođe do problema i upravljamo s njima. `$this->connection` je sada puno jednostavnije složiti jer to možemo s varijablama koje smo hardkodirali. Pošto smo rekli da želimo raditi sa exception-ima, stavit ćemo to otvaranje u `try-catch` blok. Zaustaviti ćemo izvođenje koda ako nema konekcije.

Postoje aplikacije koje kako bi bile funkcionalne ako je baza pala ili ako je pala konekcija, to su popularne progresivne aplikacije koje vam omogućavaju da radite u off-line modu. Kada se konekcija pojavi napravi se sinkronizacija.

Ovime stvaramo novu instancu `$database` kako bi mogli raditi s bazama.

```
$database = new Database()
```

Ako napravimo više instanciranja, za svako instanciranje klase u `$database` objekt, instancira se PDO i kreirira se nova konekcija i raste broj otvorenih konekcija. Dakle pitanje je kada će konekcija pući.

Rješenje je singleton. Ne želimo da svaki put kada se instancira `$database` objekt da provjerimo da li već postoji, ako postoji koristi postojeći.

Kod singletona, ne smijemo dozvoliti da neko instancira objekt izvana. Dakle ne smije nitko moći reći izvana `new Database`. Mi želimo kontrolirati mehanizam instanciranja. To je smisao singletona.

Kada radimo sa singleton-om, ne smijemo dozvoliti instancirati objekta. To je prvi korak. Kako je instanciranje zadužen `__construct` i `public` je može ga bilo tko pozvati. Trebamo ga promijeniti u `private`.

Time smo zabranili instanciranje `$database` u objekt. Ako sada probamo pokrenuti program, PHP će dojaviti grešku. Sada to želimo kontrolirati. Da bi smo pristupili nečemu unutar klase (ne objekta), ta metoda mora biti `static` tipa. Komplet dio iz `__construct` ćemo prebaciti u novu `private` funkciju `connect()`. Ono što je bilo u konstruktoru sada je u privatnoj funkciji `connect()`. Iz konstruktora pozvat ćemo tu novu funkciju `connect`.

```
private function __construct()
{
    $this->connect();
}

private function connect()
{
    $dsn = "mysql:host=localhost;dbname=adventureworkshop;charset=utf8";
    $username = "root";
    $password = "";
    $options = [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
    ];

    try {
        $this->connection = new PDO($dsn, $username, $password, $options);
    } catch (PDOException $e) {
        die("Connection failed: " . $e->getMessage());
    }
}
```

Sada ćemo napisati `public static` metodu `getInstance`. Želimo dobiti `$database` objekt na taj način da kažemo `$database = Database::getInstance();` Ova metoda koja je `static` je moguće pozvati:

```
$database = Database::getInstance();
```

Napisat ćemo je:

```
public static function getInstance(): Database
{
    if (self::$instance === null) {
        self::$instance = new Database();
    }
}
```

```

        return self::$instance;
    }
}

```

Ako je `self::$instance` jednako `null`, to znači da nemamo instancu `Database` objekta i da je trebamo kreirati (dakle instancirati) sa `self::$instance = new Database()`. To možemo pozvati jer smo unutar klase `Database`. Konstruktor je privatан па je možemo pozvati samo unutar klase (a mi jesmo unutar klase). Pozivamo konstruktor koji poziva `connect()` i `connect()` pravi konekciju. Metoda vraća `Database` (u definiciji metode). Metoda vraća tu instancu sa `return self::$instance`. Prvi puta kada neko dođe i pozove `getInstance()`, `$instance` je `null`, i stvori se instanca i vrati nazad. Drugi puta kada se pozove `getInstance()`, sada više `$instance` nije više `null` i vraća se isti objekat. Upravo to je singleton.

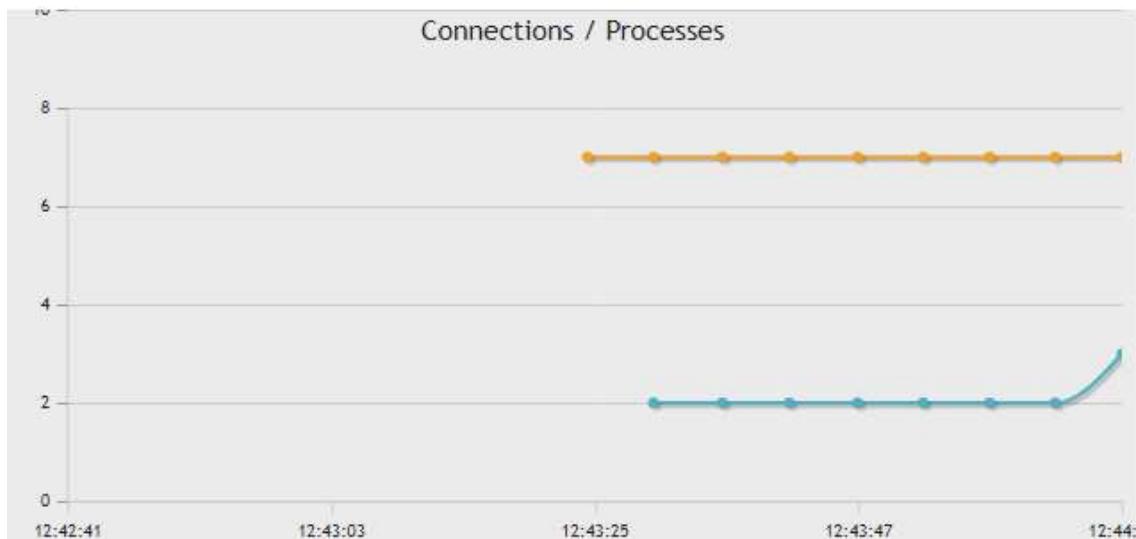
Ako sada probamo napraviti:

```

$database = Database::getInstance();
$db1 = Database::getInstance();
$db2 = Database::getInstance();
$db3 = Database::getInstance();

```

Konekcija se otvara samo prvi puta i to možemo vidjeti. Trebali bi smo vidjeti preostale tri konekcije, ali sa singletonem to nije slučaj:



Na svakom idućem pozivu `getInstance()` nije seinstanciralo.

Kada bi smo provjerili da li se radi o istoj konekciji i objektu, zaključili bi da je upravo takva slučaj:

```

var_dump($database);
var_dump($db1);

```

Dobit ćemo:

```

object(Database)#1 (3) {
    ["connection":"Database":private]=>
    object(PDO)#2 (0) {
}

```

```
}

["table":"Database":private]=>
uninitialized(string)
["columns":"Database":private]=>
string(1) "*"
["where":"Database":private]=>
array(0) {
}
}

object(Database)#1 (3) {
    ["connection":"Database":private]=>
    object(PDO)#2 (0) {
    }
    ["table":"Database":private]=>
    uninitialized(string)
    ["columns":"Database":private]=>
    string(1) "*"
    ["where":"Database":private]=>
    array(0) {
    }
}
```

Ako napišemo:

```
$db4 = $db1;
var_dump($db4==$db1);
```

Dobit ćemo referencu tj. isti objekt kao i u prethodnim slučajevima. Rezultat je:

```
bool(true)
```

Međutim ako dodamo:

```
$db5 = clone $db1;
var_dump($db5);
```

Rezultat je:

```
object(Database)#3 (3) {
    ["connection":"Database":private]=>
    object(PDO)#2 (0) {
    }
    ["table":"Database":private]=>
    uninitialized(string)
    ["columns":"Database":private]=>
```

```

string(1) "*"
["where":"Database":private]=>
array(0) {
}
}

```

Primjetit ćemo da je to potpuno drugi objekt `$database`. PDO je ostao isti. Klonirao se `$database` objekt i sada `$db5` nije isto što i `$db1`. `$db5` je novi objekt ali je PDO ostao isti. Kloriranje nije pokrenulo novo instanciranje i nije se pokrenuo PDO.

Ovo znači da trebamo zabraniti kloniranje. To ćemo napraviti tako da dodamo magic metodu `__clone` (magic metoda kao i `__construct`). Dovoljno je da dodamo i praznu metodu i kloniranje neće biti moguće. Time smo zaštitili našu `$database` objekt.

```
private function __clone() {}
```

Nakon ovoga kloniranje više ne radi, dojavljuje grešku.

Kada imamo instancu, želimo razviti neka svojstva (engl. features). Kada napravimo `$database` objekt nemamo pristup ničemu. Ne možemo napraviti upit na bazu jer nam treba pristup `$connection` gdje je PDO objekt koji može raditi `prepare` izjavu i `execute`. Da bi smo to mogli moramo napraviti getter metodu.

```

public function getConnection(): PDO
{
    return $this->connection;
}

```

Sada možemo:

```

$conn = $database->getConnection();
$conn->prepare("SELECT * FROM proizvod")->execute();

```

Direktno smo napravili `execute`, jer PHP stvori PDO statement objekt a na tom PDO statement objektu pozovemo `execute`.

Radit ćemo to ulančavanje (engl. chaining) gdje jedna metoda kad pozovemo vrati objekt, ponovo možemo pozvati drugu metodu pa treću itd. Ta metoda se koristi u [Builder obrascu](#). Na taj način se gradi (eng. build) objekt. Mi gradimo SQL upit (engl. query) na način da pozivamo neke metode sa kojima ćemo u `$database` objektu modificirati taj upit koji ćemo na kraju izvršiti. To je cilj.

tabase.php

PDO::prepare

(PHP 5 >= 5.1.0, PHP 7, PECL pdo >= 0.1.0) Prepares a statement for execution and returns a statement object

```
<?php
public function prepare(string $query, array $options = []):
PDOStatement|false { }

@param string $query
@param array $options
// $db5 [optional] This array holds one or more key=>value pairs to set attribute values for the PDOStatement object that this method returns. You would most commonly use
$conn = this to set the PDO::ATTR_CURSOR value to PDO::CURSOR_SCROLL to request a
$conn->prepare("SELECT * FROM proizvod")->execute();
```

Probati ćemo dobiti podatke sa ulančanim izrazom:

```
$conn = $database->getConnection();
$res = $conn->prepare("SELECT * FROM proizvod")->execute()->fetchAll();
var_dump($res);
```

Ovo ne radi jer `execute` vraća `true` ili `false`.

// \$db5 = clone \$db1; // ne možemo klonirati zbog private function __clone() {}
// var_dump(\$db5);
\$conn = \$database->getConnection();
\$res = \$conn->prepare("SELECT * FROM proizvod")->execute()->fetchAll();
var_dump Expected type 'object'. Found 'bool'. intelephense(P1006)
Codeium: Explain Problem

```
$res = PDOStatement::execute
->t
->s
->w <?php
->w public function execute(array|null $params = null): bool { }
->g @param array|null $params
[optional] An array of values with as many elements as there are bound parameters in the SQL statement being executed. All values are treated as PDO::PARAM_STR.
@return bool — TRUE on success or FALSE on failure.
@throws PDOException On error if PDO::ERRMODE_EXCEPTION option is true.
```

Moramo promijeniti kod:

```
$conn = $database->getConnection();
$stmt = $conn->prepare("SELECT * FROM proizvod");
$stmt->execute();
$res = $stmt->fetchAll();
var_dump($res);
```

Htjeli bi smo da `$database` objekt u sebi sadrži neke metode koje će graditi SQL i koji će pozvati bazu i vratiti rezultat. Svaka metoda će imati svoju ulogu. Željeli bi smo dohvati rezultat na sljedeći način:

```
$res = $database->get();
```

Mi želimo da naša buduća metoda `get` uzme sve što smo prije podesili (iz koje tablice, iz kojeg stupaca, where itd). Cilj je da prethodno izgradimo upit kako bi get dobio upit od izgrađenih opcija i izvratio što treba (zbog prepare statementa i bindanja vrijednosti na placeholder-e) i izvršio ga s `execute`.

Želimo napisati metodu `get`. Želimo da to nije hard kodirano, već da možemo odrediti koje stupce želimo selektirati, iz koje tablice želimo selektirati, imamo li `where` uslov itd.

```
public function get()
{
    $sql = "SELECT $this->columns FROM $this->table";
    $stmt = $this->connection->prepare($sql);
    $stmt->execute();
    return $stmt->fetchAll();
}
```

iznad na početku klase definirat ćemo `private string $table` svojstvo i to ćemo pozivati iz upita u `get()` metodi. Treba nam setter. Reći ćemo:

```
$res = $database->table('proizvod')->get();
```

Nad `$database` želimo pozvati metodu `table`, čiji je zadatak da postavi svojstvo `table` na proizvod.

Napravimo metodu `table`:

```
public function table(string $table): Database
{
    $this->table = $table;
    return $this;
}
```

`Database` je objekt koji smo dobili preko `getInstance()`. Na tom objektu imamo pristup metodi `table` koja prima argument kroz svoj parametar koji zapiše u svojstvo `$this->table`. Trenutno, prije toga, `$table` nije bio definiran. Nakon poziva metode `table`, svojstvo `table` ima zapisan `proizvod`. nakon toga kažemo `return $this`. Dakle vrati se modificiran objekt iz `$database->table('proizvod')`. Kada pozovemo metodu `get()` ona će izvući to svojstvo i dobit ćete `SELECT * FROM proizvod`.

```
public function get()
{
    $sql = "SELECT $this->columns FROM $this->table";

    if (!empty($this->where)) {
        $whereParts = [];
        foreach ($this->where as $condition) {
            $whereParts[] = "{$condition[0]} {$condition[1]} ?";
        }
    }
}
```

```

        $sql .= " WHERE " . implode(" AND ", $whereParts);
    }

    $stmt = $this->connection->prepare($sql);
    $values = array_column($this->where, 2);
    $stmt->execute($values);

    return $stmt->fetchAll();
}

```

Dosadašnji kod ćemo modificirati da prilikom korištenja `$database` objekta, da pozivom metode `get()` da umjesto `*` u `SELECT` iskazu to netko modificira.

Dodat ćemo `private string $columns = "*"` što znači kada želimo sve dohvati ostaje tako.

U `get()` metodi promijenit ćemo red u:

```
$sql = "SELECT $this->columns FROM $this->table"
```

Property ne modificiramo jer je `*` default.

```
$res = $database->table('proizvod')->get();
```

Međutim, napravit ćemo metodu `select()` koja će nam omogućiti da to modificiramo:

```
$res = $database->table('grad')->select()->get();
```

Ovo je moguće napraviti na 2 načina. Kako smo kod `table` definirali parametar tako možemo i kod `select`. Dakle, jedan način je da pošaljemo string:

```
$res = $database->table('grad')->select('Naziv')->get();
```

Drugi način je da ako želimo da pošaljemo više stupaca npr. `(['IDgrad', 'Naziv'])` onda šaljemo matricu.

Kako sad pomiriti da `select` može primiti dva različita tipa. Način je da prilagodimo metodu `select`.

```

public function select(string|array $columns) : Database
{
    $this->columns = is_array($columns) ? implode(", ", $columns) : $columns;
    return $this;
}

```

Definirali smo da može biti ili string ili matrica. Ako je mix onda može ući bilo što. Želimo vidjeti da li smo dobili matricu na ulazu. Jer ako jesmo umjesto zvjezdice trebamo imati `IDgrad`, Naziv uz ove ulazne parametre. `implode` spoji elemente matrice. Ako nije matrica onda šaljemo string.

Ovo nije pravi builder patern jer praktično gradimo upit (engl. query). Metode `table` i `get` nisu nešto komplikirane.

Ako pozovemo:

```
$res = $database->table('grad')->select(['IDgrad', 'Naziv'])->get();
$res = $database->table('drzava')->get();
```

Javit će se greška u drugom redu.

U gornjem `select` je modificiran objekt, a kako u idućem redu koristimo isti objekt, unutra su ostali podaci od prethodnog upita. Na selectu je napravljena promjena a u idućem redu ga ne koristimo. Da smo koristili `select`, problem se ne bi pojavio. Nakon svakog poziva get metode trebamo resetirati svojstva. Napraviti ćemo promjene na funkciji get:

```
public function get()
{
    $sql = "SELECT $this->columns FROM $this->table";

    if (!empty($this->where)) {
        $whereParts = [];
        foreach ($this->where as $condition) {
            $whereParts[] = "{$condition[0]} {$condition[1]} ?";
        }

        $sql .= " WHERE " . implode(" AND ", $whereParts);
    }

    $stmt = $this->connection->prepare($sql);
    $values = array_column($this->where, 2);
    $stmt->execute($values);

    return $stmt->fetchAll();
}
```

možemo poslati `where('IDGrad', 1)` a drugi puta `where('IDGrad', 1)`. Na pozicijama u argumentima jednom će doći operator a drugi puta vrijednost. Ako na drugoj poziciji dođe operator onda na trećoj poziciji dođe vrijednost. PHP ne podržava preopterećenje metoda (engl. overload, funkciju sa istim imenom i različitim brojem parametara). Upravo smo to postigli.

```
public function where(string $column, mixed $operator, mixed $value = null):
Database
{
    if(func_num_args() === 2){
        $value = $operator;
        $operator = "=";
    }

    $this->where[] = [$column, $operator, $value];
    return $this;
}
```

```

        }
    }
}
```

Možemo poželjeti imati višestruke `WHERE`, ne smiju pregaziti jedan drugog. `$this->where[]` je višedimenziona matrica.

Evo cijelog koda:

```
<?php

class Database
{
    private static ?Database $instance = null;
    private PDO $connection;
    private string $table;
    private string $columns = "*";
    private array $where = [];

    private function __clone() {}
    private function __construct()
    {
        $this->connect();
    }

    private function connect()
    {
        $dsn = "mysql:host=localhost;dbname=adventureworkshop;charset=utf8";
        $username = "root";
        $password = "";
        $options = [
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
        ];

        try {
            $this->connection = new PDO($dsn, $username, $password, $options);
        } catch (PDOException $e) {
            die("Connection failed: " . $e->getMessage());
        }
    }

    public static function getInstance(): Database
    {
        if (self::$instance === null) {
            self::$instance = new Database();
        }
    }
}
```

```
    }

    return self::$instance;
}

public function getConnection(): PDO
{
    return $this->connection;
}

public function get()
{
    $sql = "SELECT $this->columns FROM $this->table";

    if (!empty($this->where)) {
        $whereParts = [];
        foreach ($this->where as $condition) {
            $whereParts[] = "{$condition[0]} {$condition[1]} ?";
        }

        $sql .= " WHERE " . implode(" AND ", $whereParts);
    }

    $stmt = $this->connection->prepare($sql);
    $values = array_column($this->where, 2);
    $stmt->execute($values);

    return $stmt->fetchAll();
}

public function table(string $table): Database
{
    $this->table = $table;
    return $this;
}

public function select(string|array $columns) : Database
{
    $this->columns = is_array($columns) ? implode(", ", $columns) : $columns;
    return $this;
}

public function where(string $column, mixed $operator, mixed $value = null): Database
```

```
{  
    if(func_num_args() === 2){ // saznaj koliki broj argumenata imaš u funkciji  
        $value = $operator;  
        $operator = "=";  
    }  
  
    $this->where[] = [$column, $operator, $value];  
    return $this;  
}  
}  
  
$database = Database::getInstance();  
$db1 = Database::getInstance();  
$db2 = Database::getInstance();  
$db3 = Database::getInstance();  
  
var_dump($database);  
var_dump($db1);  
  
$db4 = $db1;  
var_dump($db4==$db1);  
// $db5 = clone $db1; // ne možemo klonirati zbog private function __clone() {}  
// var_dump($db5);  
$conn = $database->getConnection();  
$stmt = $conn->prepare("SELECT * FROM proizvod");  
$stmt->execute();  
$res = $stmt->fetchAll();  
var_dump($res);  
  
  
$res = $database  
->table('grad')  
->select(['IDGrad', 'Naziv'])  
->where('IDGrad', 1)  
->where('Naziv', 'Zagreb')  
->get();  
  
var_dump($res);
```

Composer

Provjera imamo li instaliran Composer

Prvo čemo pogledati imamo li instaliran Composer. Potrebno je otvoriti Git Bash i ukucati `composer`.

Instalacija Composera

Ako nam se javi, sve je u redu. Ako ne, potrebno napraviti instalaciju sa:

<https://getcomposer.org/download/>

Ažuriranje (update) Composera

Ažuriranje (update) možemo napraviti sa naredbom:

```
$ composer self-update
```

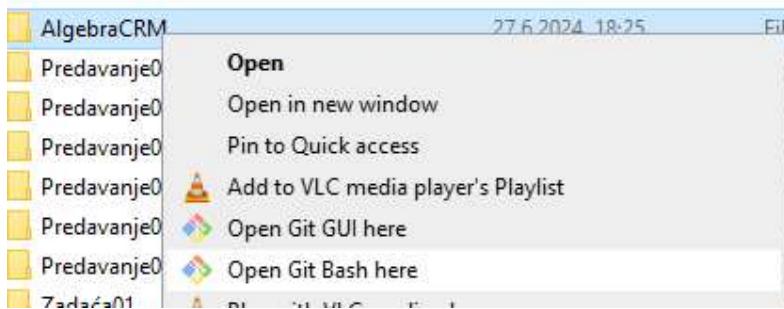
Ovaj packet manager omogućava upravljanje paketima. Benefit toga je da radite s kodom, dužni ste ga održavati. Treba voditi brigu da je kod kompatibilan s novim verzijama. Nove verzije PHP ponekad nisu 100% kompatibilne unazad i kod treba održavati. Kod treba biti kompatibilan sa novim verzijama PHP, ponekad treba nešto zamijeniti ili riješiti sigurnosne propuste.

Kada koristimo tuđi kod, on je obično pakiran. Kada je „out of date“, vlasnik koda održava kod i izdaje nove zagrpe. Ako instaliramo pakete trebamo provjeriti vjerodostojnost paketa.

Pakegist na pakegist.org je glavni repozitorij za Composer tj. ovaj packet manager.

Podešavanje Composera na projektu (datoteka `composer.json`)

Desnom tipkom miša otvorimo meni na AlgebraCRM direktoriju i odaberemo Git Bash. Naredba `composer init` koristi se za interaktivno kreiranje nove `composer.json` datoteke u korijenskom (engl. root) direktoriju vašeg PHP projekta.



Composer nam daje mogućnost da upravljamo paketima i koristimo njegov autoloader umjesto da pišemo svoj. Daje nam pristup PSR-4 standardu. Bez korištenja frameworka (u tzv. vanilla PHP-u), također je poželjno koristiti Composer zbog lakšeg upravljanja, autoloader-a i namespace-ova.

Ova naredba pokreće čarobnjaka koji te vodi kroz postupak postavljanja osnovnih informacija i konfiguracija za tvoj projekt. Evo korak-po-korak što se događa kada pokrenemo `composer init`.

```
MINGW64:/c/xampp/htdocs/Algebra/Napredni PHP/AlgebraCRM
Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/Napredni PHP/AlgebraCRM
$ composer init

Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [administrator/algebra-crm]:
```

Čarobnjak pita za osnovne informacije o projektu. Ovdje traži `vendor/` ime paketa. Kako razvijamo vlastiti paket, ostavit ćemo predložene (engl. default) vrijednosti. Razvijamo projekt a ne paket, pa nam je svejedno. Čarobnjak će redom pitati za osnovne informacije o projektu:

```
MINGW64:/c/xampp/htdocs/Algebra/Napredni PHP/AlgebraCRM
Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/Napredni PHP/AlgebraCRM
$ composer init

Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [administrator/algebra-crm]:
Description []:
Author [Branko Celap <branko.celap@gmail.com>, n to skip]:
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []: project|
```

Nakon toga traži opis paketa (Description) gdje možemo dati kratak opis projekta. Pod Author (Author), možemo upisati informacije o autoru (ime i mail). Možemo to preskočiti. Pod Minimum Stability možemo upisati koja je minimalna stabilnost paket. Npr. `stable`, `beta`, `alpha`, `dev`. U tip paketa unijeli smo `project`. Najčešće možemo još unijeti `project`. Nakon ovoga pita za licencu. Često ćemo vidjeti `MIT` licencu.

```
raCRM
$ composer init

Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [administrator/algebra-crm]:
Description []:
Author [Branko Celap <branko.celap@gmail.com>, n to skip]:
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []: project
License []: MIT

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]?
? no
Add PSR-4 autoload mapping? Maps namespace "Administrator\AlgebraCrm" to the entered relative path. [src/, n to skip]:
```

Nakon toga pita želimo li definirati ovisnosti, (dependencies) i ovisnosti za razvoj (dev dependencies). Preskočit ćemo ova dva koraka i dodati ih kasnije. Za sada `no`. Mogli smo unijeti ovisnosti koje će naš projekt koristiti (npr. `monolog/monolog`, odnosno ovisnosti koje će biti korištene samo tokom razvoja. Na primjer, `phpunit/phpunit`)

Pita da li da doda PSR-4 autoload mapiranje i da li da mapira namespace Administrator\AlgebraCRM na relativnu putanju direktorija `/src`. Pritisnemo samo Enter i mapira se na direktorij `src/`. Potvrdimo sa `yes`. Kreirana je struktura poddirektorija u direktoriju `AlgebraCRM`.

```

MINGW64:/c/xampp/htdocs/Algebra/Napredni PHP/AlgebraCRM
Would you like to define your dependencies (require) interactively [yes]? no
Would you like to define your dev dependencies (require-dev) interactively [yes]? no
Add PSR-4 autoload mapping? Maps namespace "Administrator\AlgebraCrm" to the entered relative path. [src/, n to skip]:
{
    "name": "administrator/algebra-crm",
    "type": "project",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "Administrator\\AlgebraCrm\\": "src/"
        }
    },
    "authors": [
        {
            "name": "Branko Celap",
            "email": "branko.celap@gmail.com"
        }
    ],
    "require": {}
}
Do you confirm generation [yes]? |

```

Sada možemo vidjeti konfiguracijsku datoteku. Javio nam je da koristimo „namespace `Administrator\AlgebraCRM;`“ u `src/`. To ćemo kasnije promjeniti. Također nam je javio da imamo pristup autoloader-i pomoću: `require 'vendor/autoload.php'`;

Možemo ručno uređivati `composer.json` ako želimo dodati ili izmijeniti bilo koju postavku.

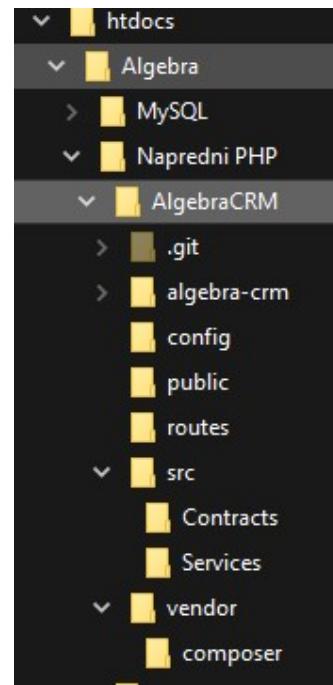
Disk (C:) > xampp > htdocs > Algebra > Napredni PHP > AlgebraCRM		
Name	Date modified	Type
src	27.6.2024. 18:39	File folder
vendor	27.6.2024. 18:39	File folder
composer.json	27.6.2024. 18:39	JSON Source File

Nakon inicijalizacije, možemo koristiti ostale Composer naredbe za upravljanje ovisnostima i daljnju konfiguraciju projekta.

Direktorij `composer` ne smijemo izgubiti. Direktorij `vendor` možemo obrisati. Uvijek možemo reći `composer install` iz direktorija `AlgebraCRM` i ponovo rekonstuirati `vendor` direktorij. `vendor` direktorij osim što sadrži autoload, kasnije može sadržavati ovisnosti (engl. dependence) tj. pakete koje instaliramo da bi smo ih koristili u našem projektu. Moramo na-mapirati namespace-ove. Datoteku `composer.json` ne smijemo izgubiti.

`composer.json` u `root` direktoriju projekta direktoriju treba izgledati ovako:

```
{
    "name": "administrator/algebra-crm",
    "type": "project",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    },
    "authors": [
        {
            "name": "Branko Ćelap",
            "email": "branko.celap@gmail.com"
        }
    ],
    "require": {
        "php": ">=8.2"
    }
}
```



U redu `"name": "administrator/algebra-crm"` se definira ime paketa/projekta. Obično je u formatu vendor/paket, gdje kod nas `administrator` označava organizaciju ili korisnika, a `algebra-crm` naziv projekta.

Ovdje se definira autoload za projekt. Kao što smo već spomenuli, PSR-4 je standard za automatsko učitavanje klasa u PHP-u. Klase koje pripadaju namespace-u `App\` će se nalaziti u direktoriju `src/`. Na primjer, u neka moguća klasa `App\Controller\HomeController` bi trebala biti smještena u `src\Controller/HomeController.php`.

Izbacio sam red: `"Administrator\\AlgebraCrm\\": "src/"` i umjesto njega napisao `"App\\": "src/"`

Ako pogledamo sliku konfiguracije, `require` smo ostavili prazno ali tu možemo natjerati da provjeri korisnikovu verziju PHP-a. To je dobro jer kad prenosimo projekt, možemo odrediti koja je najniža verzija koja radi i spriječimo da radi s nekompatibilnom verzijom. Dodao sam redove:

```
"require": {
    "php": ">=8.2"
}
```

U ovom slučaju, projekt zahtijeva PHP verziju 8.2 ili noviju. Ako korisnik ima manju verziju a proba napraviti `composer install`, pojavit će se greška.

Composer koristi `composer.json` za instalaciju i upravljanje ovisnostima definiranim u `require` sekciji. Na temelju autoload sekcije, Composer generira datoteku `vendor/autoload.php` koja automatski učitava klase prema PSR-4 standardu. Informacije o imenu, tipu, licenci i autorima koriste se za dokumentaciju i distribuciju paketa. Obično, kada pokrenete `composer install` ili `composer update`, Composer čita `composer.json` i instalira potrebne ovisnosti u `vendor` direktorij, generirajući `composer.lock` za zaključavanje verzija tih ovisnosti.

Važno je napomenuti da bi `composer.json` trebao biti u korijenskom direktoriju projekta, dok `vendor` direktorij sadrži instalirane ovisnosti, uključujući Composerov vlastiti kod i sve biblioteke (engl. libraries) koje su navedene kao ovisnosti.

Composer naredbe

Instalacija ovisnosti

Prvi put kada postavite projekt s `composer.json`, trebate pokrenuti

`composer install`

Ova naredba preuzima i instalira sve ovisnosti navedene u `composer.json` i stvara `vendor` direktorij.

Dodavanje novih ovisnosti

Kada dodaješ novu ovisnost, koristite:

`composer require ime/ovisnosti`

Ovo će ažurirati `composer.json`, preuzeti novu ovisnost i dodati je u `vendor` direktorij.

Ažuriranje postojećih ovisnosti

Kada želiš ažurirati sve ovisnosti na najnovije verzije koje zadovoljavaju ograničenja navedena u `composer.json`, koristite:

`composer update`

Ova naredba ažurira sve ovisnosti, ažurira `composer.lock` datoteku i preuzima nove verzije ovisnosti u `vendor` direktorij.

Ažuriranje specifične ovisnosti

Ako želiš ažurirati samo određenu ovisnost, možeš koristiti:

`composer update ime/ovisnosti`

Ovo će ažurirati samo tu ovisnost i odgovarajuće zapise u `composer.lock`.

Uklanjanje ovisnosti

Ako želiš ukloniti neku ovisnost, koristiš:

```
composer remove ime/ovisnosti
```

Ovo će ukloniti ovisnost iz `composer.json`, ažurirati `composer.lock` i ukloniti je iz `vendor` direktorija.

Provjera novih verzija ovisnosti

Composer omogućava provjeru dostupnosti novih verzija ovisnosti koristeći:

```
composer outdated
```

Ova naredba prikazuje popis ovisnosti koje imaju novije verzije dostupne.

Inicijalizacija novog GIT repozitorija

Sljedeći korak je postavljanje GIT-a:

```
git init
```

```
$ git init
Initialized empty Git repository in C:/xampp/htdocs/Algebra/Napredni PHP/Algebra
CRM/.git/
Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/Napredni PHP/Algeb
raCRM (master)
$ |
```

Da se podsjetimo `git init` omogućava:

1. Verzioniranje:
 - Omogućava praćenje promjena u datotekama tijekom vremena. Svaka promjena može biti spremljena kao commit, koji čuva stanje koda u tom trenutku.
2. Povijest promjena:
 - Svaki commit stvara trajni zapis o promjenama koje su napravljene, omogućavajući povratak na prethodna stanja koda.
3. Kolaboracija:
 - Git omogućava suradnju među više programera. Korištenjem udaljenih repozitorija (npr. na GitHubu), programeri mogu raditi na istom projektu, spajati promjene i rješavati konflikte.
4. Eksperimentiranje:
 - Grane omogućavaju eksperimentiranje s novim svojstvima ili popravcima bez utjecaja na glavni kod. Nakon što su promjene testirane, mogu biti spojene natrag u glavnu granu.

`git init` je osnovna Git naredba koja inicijalizira novi Git repozitorij u trenutnom direktoriju. Omogućava praćenje verzija, kolaboraciju i eksperimentiranje unutar projekta. Nakon inicijalizacije, možeš dodavati datoteke u praćenje, kreirati commitove i koristiti ostale Git funkcionalnosti za upravljanje i verzioniranje koda.

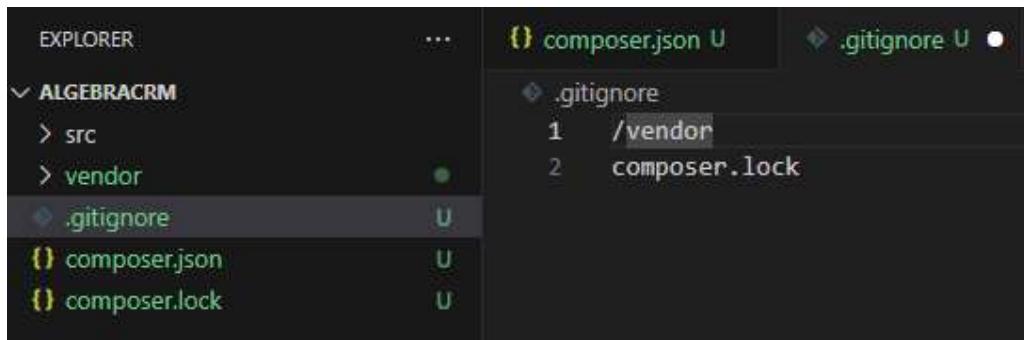
Time smo se pripremili da ignoriramo `composer.lock` i `vendor`. Kada nemamo vendor mapu izvuče je iz cache-a. `composer.lock` se koristi kod upgrade-a za usporedbu.

`composer.json` je inicijalna datoteka. `composer.lock` nije nužan i može izazvati probleme.

```
$ composer update
Loading composer repositories with package information
Updating dependencies...
Nothing to modify in lock file
Installing dependencies from lock file (including require-dev)
Nothing to install, update or remove
Generating autoload files
No installed packages - skipping audit.

Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/Napredni PHP/AlgebraCRM (master)
$ |
```

`composer.lock` će probati instalirati ali ako ga obrišemo svatko će na svom kompjuteru praviti update.



Želimo ignorirati direktorij `vendor` jer ne smije biti na git-u i `composer.lock`. Napravićemo `.gitignore` datoteku.

PSR-4

PSR-4 opisuje specifikaciju za automatsko učitavanje klase iz putanja datoteka. Ovaj PSR također opisuje gdje smjestiti datoteke koje će se automatski učitati prema specifikaciji.

Izraz "klasa" odnosi se na klase, interface-e, traits i druge slične strukture.

Potpuno kvalificirano ime klase ima sljedeći oblik:

```
\<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
```

- Potpuno kvalificirano ime klase MORA imati imenski prostor (engl. namespace) imena najvišeg nivoa, također poznat kao "vendor namespace".
- Potpuno kvalificirano ime klase MOŽE imati jedno ili više imena pod-namespace-ova.
- Potpuno kvalificirano ime klase MORA imati završno ime klase.

- Crtice za podvlačenje nemaju posebno značenje ni u jednom dijelu potpuno kvalificiranog naziva klase.
- Abecedni znakovi u potpuno kvalificiranom nazivu klase MOGU biti bilo koja kombinacija malih i velikih slova.
- Sva imena klasa MORAJU biti navedena na način koji razlikuje velika i mala slova.

Prilikom učitavanja datoteke koja odgovara potpuno kvalificiranom nazivu klase ...

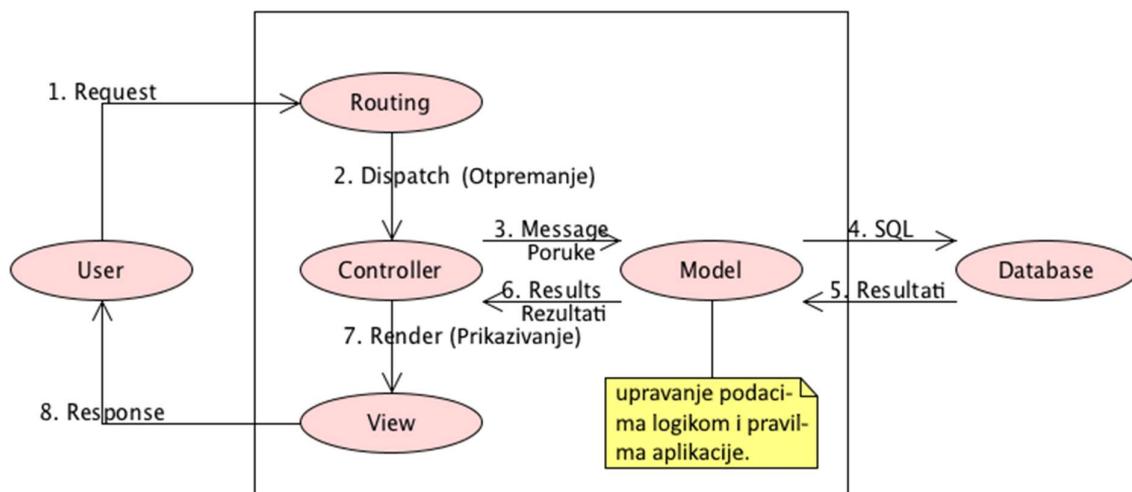
- Neprekinuti niz jednog ili više vodećih namespace-ova i pod-namespace-ova, ne uključujući namespace separator, u potpuno kvalificiranom nazivu klase ("prefiks prostora imena") odgovara najmanje jednom "osnovnom direktoriju".
- Granična pod-namespace nakon "prefiksa prostora imena" odgovaraju poddirektoriju unutar "osnovnog direktorija", u kojem separatori namespace predstavljaju separatore direktorija. Naziv poddirektorija MORA odgovarati velikim i malim slovima imena prostora pod-namespace-a.
- Završni naziv klase odgovara nazivu datoteke koji završava na .php. Naziv datoteke MORA odgovarati velikim i malim slovima imena završne klase.

Implementacije automatskog učitavanja (autoloader) NE SMIJE bacati iznimke, NE SMIJE pokretati pogreške bilo kojeg nivoa i NE SMIJE vraćati vrijednost.

MVC arhitektura

Dolazimo do MVC arhitekture.

U modelu se događa poslovna logika. Model može raditi na dva načina. Jedan je da s modela ide direktno na pogled.



MVC dizajn uzorak je uzorak softverske arhitekture koji razdvaja aplikaciju u tri dijela. **Model, Pogled (engl. View) i Kontroler (engl. Controller)**, što olakšava upravljanje i održavanje kodom. Također omogućuje ponovnu upotrebu komponenti i promovira modularniji pristup razvoju softvera.

Prvo, browser šalje zahtjev kontroleru. Nemamo jedan kontroler nego njih više. Ako imamo jedan kršimo Single Responsibility solid princip. Kako znamo kojem kontroleru trebamo poslati zahtjev je usmjerenje (engl. routing) aplikacije. Sistem na temelju URL-a zna kojem kontroleru poslati zahtjev. Sistem hvata zahtjeve zato što ćemo na serveru reći da sve zahtjeve preusmjeravamo na jednu ulaznu točku tj. datoteku. Svi dolazni zahtjevi će pristizati Routeru i on će ih odašiljati (engl. dispatch). To je cilj. MVC kao takav teško da može raditi na drugačiji način. Bilo koja aplikacija je izgrađena na ovaj način jer je najbolji.

Zatim Kontroler stupa u interakciju s modelom radi slanja i primanja podataka.

Kontroler zatim stupa u interakciju s Pogledom kako bi prikazao podatke. Pogled se brine samo o tome kako prezentirati informacije, a ne o konačnoj prezentaciji. Bit će to dinamička HTML datoteka koja prikazuje podatke na temelju onoga što joj Kontroler šalje.

Konačno, Pogled će poslati svoju konačnu prezentaciju Kontroleru, a Kontroler će poslati te konačne podatke korisničkom izlazu.

Važno je da Pogled i Model nikad ne komuniciraju jedan s drugim. Jedina interakcija koja se odvija između njih je putem Kontrolora.

To znači da logika aplikacije i interface-a nikad ne djeluju jedno na drugo, što olakšava pisanje složenih aplikacija.

Kada korisnik unese URL u browser, Fasada prosljeđuje zahtjev na odgovarajuću Rutu. Ruta zatim određuje koji Kontroler treba obraditi zahtjev. Kontroler poziva odgovarajući Model da dohvati potrebne podatke, a zatim poziva Pogled da generira HTML koji će se prikazati u browseru.

Suradnja među MVC komponentama

Koordinacija između komponenti odvija se na sljedeći način:

Evo detaljnijeg objašnjenja toka rada aplikacije algebra-crm, uključujući moguće kontrolere, metode routera i objašnjenje uloge fasade:

```

algebra-crm/
├── .gitignore
├── composer.json
├── composer.lock
├── index.php
└── public/
    └── index.php
controllers/
    ├── ClientController.php
    ├── Controller.php
    ├── PositionController.php
    └── UserController.php
models/
    ├── ClientModel.php
    ├── Model.php
    ├── PositionModel.php
    └── UserModel.php
routes/
    └── web.php
views/
    ├── clients/
    │   ├── create.php
    │   ├── edit.php
    │   └── index.php
    ├── positions/
    │   ├── create.php
    │   ├── edit.php
    │   └── index.php
    ├── users/
    │   ├── create.php
    │   ├── edit.php
    │   └── index.php
    └── layout.php

```

1. Korisnik šalje zahtjev
 - Korisnik šalje zahtjev putem URL-a (npr. `/clients/create`)
2. Apache preusmjerava zahtjev
 - `.htaccess` datoteka preusmjerava sve zahtjeve na public/index.php
3. Glavna ulazna datoteka `index.php`
 - učitava potrebne konfiguracije i pokreće aplikaciju
4. Autoloading i routing
 - autoload.php omogućava automatsko učitavanje klasa
 - URI se dijeli na segmente i prosljeđuje `Router` klasi
5. Router parsira zahtjev
 - `Router` klasa parsira URI i određuje odgovarajući `Controller` i metodu
 - Prosljeđuje parametre za akciju
6. Fasada i rute
 - `App` fasada prosljeđuje zahtjev na `Route` klasu
 - `Route` klasa mapira URL na odgovarajući `Controller` i metodu
 - Primjer rute: `Route::get('/clients/create', 'ClientController@create');`
7. Kontroler obrađuje zahtjev
 - Odgovarajući `Controller`, npr. `ClientController`, prihvaca zahtjev
 - Analizira URL i određuje koju akciju treba izvršiti
8. Kontroler dohvaca podatke
 - `Controller` dohvaca potrebne podatke iz Model klase
9. `View` generira HTML
 - Controller šalje pripremljene podatke View klasi
 - View klasa generira HTML stranicu
10. Kontroler šalje odgovor
 - `Controller` prima generirani HTML od View klase
 - Šalje HTML kao odgovor korisniku

U ovom detaljnem toku rada, ključne komponente su:

`Router` klasa - parsira URI i određuje odgovarajući `Controller` i metodu

`Route` klasa - mapira URL na `Controller` i metodu

`App` fasada - prosljeđuje zahtjev na `Route` klasu

`Controller` klase - obrađuju zahtjev, dohvacaču podatke i šalju view-u

`Model` klase - upravljaju interakcijom s bazom podataka

`View` klase - generiraju HTML stranicu

Ovakva organizacija koda i toka rada omogućava bolju modularnost, održivost i testiranje aplikacije algebra-crm.

Ovaj tok kontrole i razmjena podataka između komponenti omogućuje jasnou podjelu odgovornosti i fleksibilnu arhitekturu. Promjene u jednoj komponenti ne utječu na druge, što olakšava održavanje i proširivanje aplikacije.

Ovaj primjer prikazuje osnovnu strukturu MVC arhitekture u kontekstu web browsera. U stvarnim projektima, ova arhitektura može biti puno složenija, uključujući dodatne slojeve, servise, repozitorije i druge komponente. Međutim, ključni koncepti ostaju isti: Kontroler upravlja protokom, Model rukuje podacima, a Pogled prikazuje informacije korisniku.

Organizacija MVC arhitekture

To je način kako bismo mogli organizirati MVC arhitekturu u PHP-u (na sličan način kao u Laravelu):

1. Fasade (Facades):

- U vanilla PHP-u možemo implementirati vlastite fasade kao statičke klase koje pružaju jednostavan pristup ključnim komponentama aplikacije.
- Ove fasade mogu služiti kao "proxy" instance za pristup servisima poput autentifikacije, baze podataka, slanja e-pošte i sl.
- Primjer fasade za pristup bazi podataka:

```
class DB {  
    public static function query($sql) {  
        // Logika za izvršavanje SQL upita  
    }  
}
```

2. Routing:

- U vanilla PHP-u možemo implementirati vlastiti jednostavan sustav usmjeravanja.
- Definirali bismo rute i mapirali ih na odgovarajuće kontrolere.
- Primjer:

```
$routes = [  
    '/' => 'HomeController@index',  
    '/users/{id}' => 'UserController@show'  
];
```

3. Kontroleri:

- Kontroleri bi bili zasebne PHP datoteke ili klase koje obrađuju korisničke zahtjeve.
- Kontroleri bi pozivali Modele za dohvaćanje podataka i proslijedivali ih Pogledima.
- Kontroleri se nalaze u direktoriju controllers. Oni sadrže klase poput ClientController i PositionController koje obrađuju korisničke zahtjeve, dohvaćaju podatke iz modela i pripremaju ih za prikaz.

4. Modeli:

- Modeli bi bili zasebne PHP klase koje predstavljaju entitete i upravljaju podacima.
- Modeli bi sadržavali logiku za pristup i manipulaciju podacima.
- Nalaze se u direktoriju app/Models i sadrže klase poput [ClientModel](#) i [PositionModel](#)

5. Pogledi:

- Pogledi bi bili zasebne PHP datoteke koje generiraju HTML prikaz.
- Pogledi bi koristili podatke dobivene od Kontrolera.

- nalaze direktno u `controllers` direktoriju, a ne u zasebnom views direktoriju. Oni sadrže HTML template datoteke poput `clients.index` i `clients.create`.

`index.php` ulazna točka

Otvorit ćemo `public` direktorij. Ovaj direktorij je izložen (engl. exposed). To je bitno za sigurnost. Kada uzmete server imate root, i unutra su osjetljive informacije. Mi svjesno izložimo prema van određeni direktorij. To radi i XAMPP. Kada pokrenemo Apache i pokrenemo `localhost:8081` prema van se izloži samo jedan direktorij i to `xampp\htdocs`. Bio je zadatok gdje smo u konfiguraciji Apache-a `httpd.conf` mijenjali root document. Isto tako ćemo mi izlažiti u `public` direktoriju samo jednu datoteku.

To će biti `index.php`. To je ulazna točka u našu aplikaciju. `index.php` će imati dva zadatka: da uključi `autoloader.php` i da uključi router. Svi zahtjevi dolaze u `index.php` jer ćemo to reći Apache-u i svi će biti redirektani na public direktorij i taj `index.php`. Napraviti ćemo unutar `index.php` Router koji će otpremati (engl. dispatch) te zahtjeve (engl. request) prema kontroleru. Tako je omogućeno da browser stavlja direktno zahtjev (engl. request) na kontroler. Browser šalje zahtjev (engl. request) na server. Server redirekta na `index.php`. Mi u `index.php` upalimo Router i kažemo pročitaj putanju sa koje je došao zahtjev, kada otkrijemo koja je, preusmjerimo na tu rutu a na rutu kažemo zadužen je određeni Kontroler. Sada uzmemo podatke iz zahtjeva i pošaljemo ih kontroleru. Kontroler dalje radi svoje.

Mora biti `.php` datoteka i ne može biti html datoteka, jer u HTML ne možemo napraviti `include` niti `require`.

```
require __DIR__ . '/../vendor/autoload.php';

echo $_SERVER['REQUEST_URI'];
```

Ovo nije moguće napraviti u `.html` datoteci.

Prvo ćemo definirati potrebne `require`. `require` je identičan kao i `include` osim što će u slučaju neuspjeha proizvesti fatalnu pogrešku nivoa `E_COMPILE_ERROR`. Drugim riječima, zaustavit će skriptu, dok uključi samo emitira upozorenje (`E_WARNING`) koje omogućuje nastavak skripte.

`__DIR__` je magična konstanta u PHP-u koja sadrži apsolutnu putanju direktorija u kojem se trenutna PHP skripta nalazi. Dohvaćanje apsolutne putanje direktorija: `__DIR__` vraća apsolutnu putanju direktorija u kojem se trenutna PHP skripta nalazi. Ova putanja ne uključuje naziv same skripte, samo direktorij.

Korištenje u uključivanju datoteka: Ovdje se `__DIR__` koristi u kombinaciji s `require` za uključivanje drugih PHP datoteka. To omogućava portabilnost koda jer ne ovisi o relativnoj putanji datoteka.

`require` je ključna riječ u PHP-u koja se koristi za uključivanje i izvršavanje vanjske PHP datoteke unutar trenutne skripte. `index.php` je u `public` direktoriju. `'/../vendor/autoload.php'` znači izadi iz `public`, uđi u `vendor` i učitaj `autoload.php`. Time će se paliti `autoloader` u `autoload.php`.

Informaciju o URL-u dobivamo sa `echo $_SERVER['REQUEST_URI']`, to iščita ono što je gore u address baru.

.htaccess Apache datoteka

Apache nudi nešto što se zove `.htaccess` datoteka što omogućava „on fly“ promjenu konfiguracije. Želimo da se server ponaša na određen način. Treba upaliti rewrite mod i rewrite pravilo. Predavač je poslao datoteku, koju je pripremio:

```
<IfModule mod_rewrite.c>
RewriteEngine On // uključi RewriteEngine On

# Zaustavi obradu ako se već nalaziš u /public direktoriju
RewriteRule ^public/ - [L]

# Statički resursi ako postoje
RewriteCond %{DOCUMENT_ROOT}/public/$1 -f
RewriteRule (.+) public/$1 [L]

# Usmjeri (Route) sve ostale zahtjeve na index.php
RewriteRule (.*) public/index.php?route=$1 [L,QSA]
</IfModule>
```

Uključi `mod_rewrite`:

Ova `.htaccess` konfiguracija koristi Apache-ov modul `mod_rewrite` za preusmjeravanje URL-ova u web aplikaciji.

```
RewriteEngine On
```

Ova linija uključuje `mod_rewrite` modul, što omogućava da se ostala pravila preusmjeravanja primijene.

Zaustavi obradu ako se već nalaziš u `/public` direktoriju:

```
RewriteRule ^public/ - [L]
```

Ovo pravilo odgovara bilo kojem URL-u koji počinje s `public/` i zaustavlja daljnju obradu preusmjeravanja pomoću `[L]` zastavice, koja znači "Last" (posljednje). Ovo sprječava rekuziju i osigurava da se zahtjevi koji su već u `public` direktoriju ne preusmjeravaju ponovno.

Posluži statičke resurse ako postoje:

```
RewriteCond %{DOCUMENT_ROOT}/public/$1 -f
RewriteRule (.+) public/$1 [L]
```

Ove linije provjeravaju postoji li traženi resurs (datoteka) u `/public` direktoriju. Ako postoji, zahtjev se preusmjerava tako da poslužuje datoteku direktno iz `public` direktorija. `RewriteCond` direktiva postavlja uvjet koji provjerava postoji li datoteka, a `RewriteRule` direktiva provodi stvarno preusmjeravanje ako je uvjet ispunjen. `[L]` zastavica opet zaustavlja daljnju obradu ako se ovo pravilo podudara.

Preusmjeri sve ostale zahtjeve:

```
RewriteRule (.*) public/index.php?route=$1 [L, QSA]
```

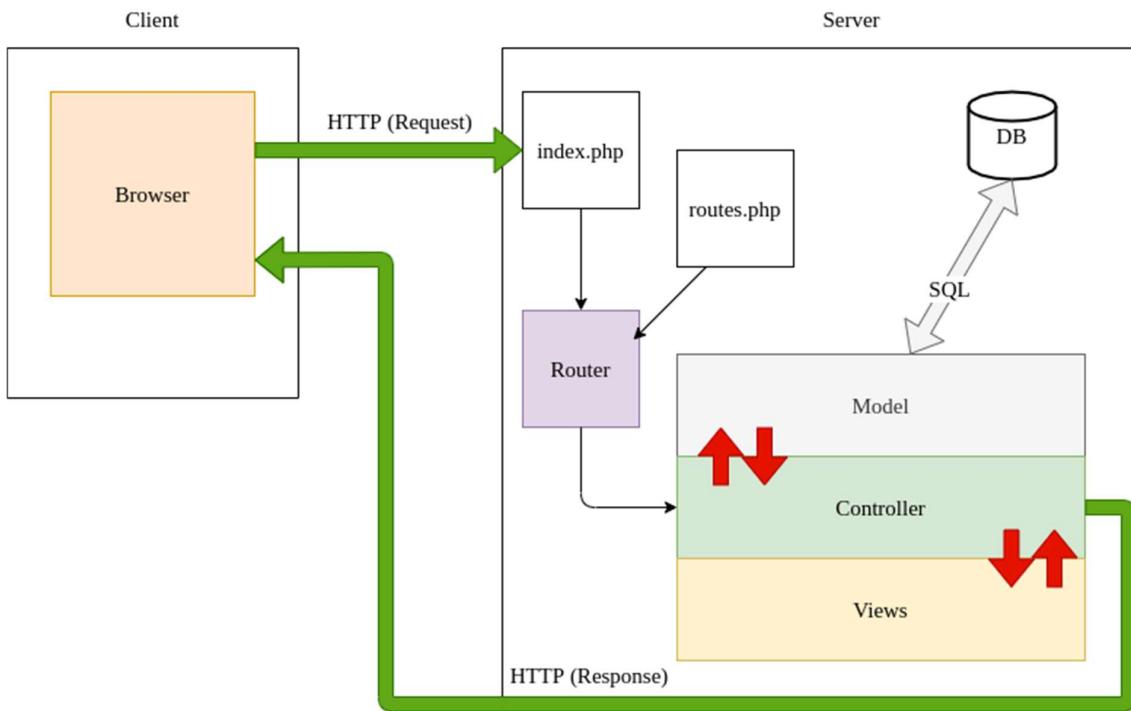
Ovo završno pravilo hvata sve preostale zahtjeve i preusmjerava ih na `public/index.php`, prosljeđujući izvorni traženi URL kao parametar upita nazvan route. `[L]` zastavica zaustavlja daljnju obradu, a `[QSA]` zastavica osigurava da se postojeći niz upita dodaje novom nizu upita.

Zahtjevi za statičkim datotekama u `/public` direktoriju poslužuju se direktno ako datoteke postoje. Svi ostali zahtjevi preusmjeravaju se na `public/index.php` s originalnom putanjom dodanom kao parametar upita, što omogućava PHP skripti da internu rukuje preusmjeravanjem.

Ako probamo pristupiti projektu preko browsera i adresi `localhost:8081/Algebra/Napredni PHP/AlgebraCRM` vidjet ćemo kompletnu strukturu direktorija i datoteka (`composer.json`, `composer.lock`, `public`, `src` i `vendor` direktorije). Međutim, ako u root projekta stavimo `.htaccess` datoteku, korisnik neće vidjeti strukturu nego sadržaj `index.php`, koji je u ovom trenutku još prazan. Sada imamo tunel kroz koji možemo preusmjeriti sve zahtjeve.

Router

Sada možemo raspisati Router tako da kontroliramo zahtjev i čitamo koji je URL došao.



Otpremanje (engl. dispatch) ćemo kontrolirati tako da pročitamo URL sa `echo $_SERVER['REQUEST_URI']` i na temelju toga ćemo znati koji kontroler je zadužen. Na temelju toga poslat ćemo zahtjev. Dakle uloga Routera je da otpremi zahtjeve Kontrolerima.

Ubacit ćemo konfiguracijske datoteke u direktoriju `config`. Unutra ćemo napraviti `app.php`.

```
<?php  
  
define('APP_NAME', 'Algebra CRM');
```

```
define('APP_ROOT', dirname(__DIR__));
define('APP_PUBLIC', APP_ROOT . '/public');
define('APP_URL', 'http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM');
```

To su podaci koji će završiti na GitHub-u i tu ćemo definirati manje osjetljive podatke. Direktorij `config` je van dohvata verzioniranja.

`define('APP_NAME', 'Algebra CRM')` definira konstantu `APP_NAME`. `APP_NAME` sadrži naziv aplikacije, u ovom slučaju `Algebra CRM`.

Konstanta `APP_ROOT` sadrži putanju do korijenskog direktorija aplikacije. `dirname(__DIR__)` vraća putanju do roditeljskog direktorija trenutnog direktorija (gdje je `app.php` smješten). Ova konstanta je korisna za definiranje absolutnih putanja unutar aplikacije. Moguće je konfigurirati i sa:

```
define('APP_ROOT', __DIR__ . '/../');
```

Uzima direktorij gdje je `app.php` i izlazi iz nje. To je root i stavi ga u konstantu.

Konstanta `APP_PUBLIC` sadrži putanju do `public` direktorija aplikacije. Kombinira vrijednost `APP_ROOT` s poddirektorijem `/public` da bi se dobila puna putanja do javnog direktorija. Ova konstanta se može koristiti za pristup datotekama i resursima koji su dostupni javnosti, kao što su slike, CSS datoteke, JavaScript datoteke itd. Konstanta `APP_URL` sadrži osnovni URL aplikacije. Ovaj URL se koristi za generiranje absolutnih URL-ova unutar aplikacije, što pomaže u pravilnom povezivanju stranica i resursa.

Na kraju `app.php` ćemo potencijalno staviti kasnije konfiguraciju baze.

Završavamo `index.php`

Sljedeći korak je da se vratimo u `index.php` u `public` direktoriju i dodamo konfiguraciju:

```
require __DIR__ . '/../config/app.php';
```

Sada nakon autoloadera učitavamo konfiguraciju iz `app.php`. Sada će nam konstante biti dostupne svuda, jer smo u globalnom doseg (engl. scope).

Zadnji korak je da u root direktoriju formiramo direktorij `routes` i datoteku `web.php`. Dakle uključit ćemo rute. Bitno je da je autoloader prvi učitan jer će nam trebati funkcionalnosti iz direktorija `src`.

Dodat ćemo zadnji red u `index.php` koji je u public direktoriju:

```
require __DIR__ . '/../ROUTES/web.php';
```

Sada cijeli `index.php` izgleda ovako:

```
require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../config/app.php';
require __DIR__ . '/../ROUTES/web.php';

echo $_SERVER['REQUEST_URI'];
```

Definirajmo Router

Idemo definirati Router,. Prije nego što izgradimo funkcionalnost, zapitamo se što nam je sve potrebno. Za Router ćemo definirati u `src` direktoriju `Services` poddirektorij i u njemu `Router.php`

Definirat ćemo u njemu `namespace App\Services`. Taj `App` je bitan jer PSR-4 iz `composer.json` u dijelu koji kaže da to uzme

```
"autoload": {
    "psr-4": {
        "App\\": "src/"
    }
},
```

iz namespace App i pogleda u `src` direktorij, dakle ako navedemo u `Router.php`, `namespace App\Services` on pogleda u direktorij `src`, i u poddirektorij `Services`, tamo bi trebala biti klasa `Router` koju ćemo napisati.

Želimo da Router funkcionira na sljedeći način: Svaki puta kada `index.php` uključi `web.php` da `web.php` uz pomoć `Router.php` napravi matricu (engl. array) u kojoj će biti definirane sve rute a svaka ruta će imati neki kontroler i metodu na tom kontroleru koja će biti zadužena za taj zahtjev (engl. request).

Podsjetimo se kako zahtjevi pristići na server. Metodom `GET` i metodom `POST`., o čemu smo više puta govorili.

Razlike između `GET` i `POST`

	Metoda <code>GET</code>	Metoda <code>POST</code>
Način slanja podataka	Podaci se šalju kao dio URL-a, odnosno kroz query string. Na primer, URL može izgledati ovako: <code>http://example.com/page?param1=value1&param2=value2</code> .	Podaci se šalju u tijelu HTTP zahtijeva. URL izgleda ovako: <code>http://example.com/page</code> , a podaci se šalju odvojeno u tijelu (engl. body) zahtjeva.
Vidljivost podataka	Podaci su vidljivi u adresnoj traci browsera i mogu biti sačuvani u povijesti pretraga, logovima servera ili dijeljeni preko URL-a.	Podaci nisu vidljivi u adresnoj traci browsera i ne čuvaju se u povijesti pretraga ili logovima servera na isti način kao <code>GET</code> parametri.
Ograničenje veličine	Postoji ograničenje veličine podataka koje može da se pošalje. To ograničenje zavisi od browsera i servera, ali obično je oko 2048 karaktera.	<code>POST</code> zahtjevi nemaju striktno ograničenje veličine podataka kao <code>GET</code> zahtjevi. Server i browser mogu postaviti neka ograničenja, ali ona su obično mnogo veća od onih za GET zahtjeve.
Idempotencija	<code>GET</code> je idempotentna metoda, što znači da više ponovljenih poziva sa istim URL-om neće promijeniti stanje servera. Idealna je za preuzimanje podataka bez ikakvog utjecaja na server.	POST nije idempotentna metoda. Višestruki pozivi istog <code>POST</code> zahtjeva mogu dovesti do različitih rezultata (npr. kreiranje više resursa na serveru). Koristi se za slanje podataka koji mogu promijeniti stanje servera.
Keširanje	GET zahtjevi se često keširaju od strane browsera i proxy servera, što	POST zahtjevi se obično ne keširaju jer se koriste za operacije koje mijenjaju stanje servera.

	može poboljšati performanse aplikacije.	
--	---	--

Kada koristiti koju metodu:

POST:

- Kada želite da dobijete ili pretražite informacije sa servera.
- Kada podaci nisu osjetljivi i nije problem ako se vide u URL-u.
- Kada želite da omogućite keširanje odgovora za poboljšane performanse.

POST:

- Kada šaljete podatke koji mijenjaju stanje servera, kao što su podaci iz formulara za prijavu, registraciju ili slanje komentara.
- Kada su podaci veći ili sadrže osjetljive informacije koje ne želite da se vide u URL-u.
- Kada želite da izbjegnete keširanje zahtjeva.

Završimo Router

Želimo nekako dobiti sljedeći slučaj (engl. case): URI treba biti ključ a u njemu je matrica koja ima ime kontrolera i metodu.

```
{  
    [  
        'URI' => ['Controller', 'Method'],  
        'URI' => ['Controller', 'Method'],  
        'URI' => ['Controller', 'Method']  
  
    ];  
}
```

Želimo da klasa Router kreira ovaku matricu a da to u `web.php` to napišemo na jednostavan način, dakle da na jednostavan način to definiramo uz pomoć koda a ne hardkodiramo.

U monolitnim aplikacijama sa Laravelom postoji problem kada zahtjev (engl. request) ide iz HTML je HTML forme ne podržavaju direktno metode `PUT`, `PATCH` i `DELETE`. Laravel koristi `_method` polje za simulaciju ovih metoda u okviru `POST` zahtjeva, ali o tome kasnije. Kod API nema takvih problema. Kod API centrične aplikacije Backend je razvijen kao RESTful ili GraphQL API, dok frontend može biti odvojen i razvijan korištenjem frontend frameworka kao što su Vue.js, React ili Angular.

U biti iz reda:

```
'URI' => ['Controller', 'Method'],
```

`Method` je funkcija na kontroleru. Metodu ćemo morati dodatno naznačiti jer je potencijalno moguće isti `URI` ali jedan URI može biti poslan metodom `GET` a drugi zahtjev metodom `POST`. Dakle i tu informaciju moramo negdje umetnuti.

Kako je ta matrica cilj, onda će `Router` imati:

```
namespace App\Services  
class Router
```

```
{
    private array $routes = [];
}
```

Postavlja se pitanje želimo li za svakog korisnika koji dođe na aplikaciju kreirati njegovu klasu `Router` (za simultane korisnike) ili ćemo htjeti samo jedan objekt. Kada se `Router` objekt kreira nije potreban drugi, treći, četvrti objekt jer su rute iste, nema potrebe da budu različite rute. Pojedinu rutu možemo štiti - da li pojedini korisnik ima prava da bude na toj ruti ali to je nešto drugo. U biti svi trebaju isti set ruta. Trebamo postići da uvijek ima samo jedan objekt. Već znamo od prije da to možemo postići sa [singleton obrascem](#). Ovdje ne bi koristili puno resursa više da i ne stavimo singleton obrazac ali želimo optimizirati obrazac.

Dodajemo u Router:

```
private static ?Router $router = null;
```

Ovim redom definiramo privatnu statičku varijablu `$router` sa tipom koji može biti instanca `Router` ili `null`. Varijabla je dostupna samo unutar klase i ne može joj se pristupiti izvan klase. Varijabla pripada klasi a ne instancama klase. Svi objekti te klase dijele istu instancu ove varijable. Može joj se pristupiti koristeći `self::$router` unutar klase. Da nije private mogli bi pristupiti i van klase sa `Router::$router`. `?` ispred tipa govori da je tip nullable, tj. da može primiti vrijednost `null`. Dakle, `?Router` označava da varijabla `$router` može biti instanca interface-a `Router` ili `null`. `= null` označava da je početna vrijednost varijable `null` dok se ne postavi na instancu koja implementira `Router`.

Nadalje, pravimo private funkciju `__construct()`. Kloniranje isto tako želimo zabraniti.

```
private function __construct(){}
private function __clone(){}
```

Da bi mogli napraviti singleton, napravit ćemo `getInsance` metodu koja će provjeriti da li je u svojstvu `$router` zapisana instanca ili nije. Ako je zapisana, vratit će je. Ako nije, stvorit će je. Evo dosadašnjeg koda:

```
class Router
{
    private array $routes = [];
    private static ?Router $router = null;

    private function __construct(){}
    private function __clone() {}

    public static function getInstance(): Router
    {
        if (self::$router === null) {
            self::$router = new Router();
        }
    }
}
```

```
        return self::$router;
    }
```

Sljedeći korak je da kreiramo metodu čiji je zadatak da puni matricu sa svim mogućim routama. Dakle, mora biti dostupna izvana. Nazvat ćemo je `addRoute`. Kako bi imali potpunu sliku trebamo imati `URI` na kojem se dogodio zahtjev, trebamo metodu da li se zahtjev dogodio putem `GET`-a ili `POST`-a, trebamo kontroler na koji će biti preusmjeren zahtjev (engl. request) kada korisnik napravi zahtev na taj `URI` i trebamo metodu zaduženu za obradu tog zahtjeva. Dakle imamo četiri elementa koja ova `addRoute` metoda mora primiti. Nećemo slati cijeli `URI` nego ćemo slati samo nastavak nad onim što je definirano u `app.php`, a to je `http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM`, nadošukturat (konkatenirat) ćemo dio.

```
public function addRoute(string $path): void
{
    }

}
```

Kako u linku iznad nema kose crte na kraju, ovu metodu pozivat ćemo sa:

```
addRoute('/login', 'AuthController', 'login', 'POST');
```

Ako želimo na home page, napisat ćemo `'/'`. Kako ovdje vidimo drugi parametar bi trebao isto biti string `$controller`, zatim ime metode nad tim kontrolerom string `$action`, string method (`GET` ili `PUT`).

Logika je kada Router otpremi (engl. dispatch) zahtjev prema funkciji, ona se pokrene i router preusmjeri zahtjev. Unutra smo napisali neku funkcionalnost i na taj način smo preusmjerili zahtjev prema tom kontroleru.

```
public function addRoute(string $path, string $controller, string $action,
string $method): void
{
    $uri = APP_URL . $path;
    $this->routes[$method][$uri] = [
        'controller' => $controller,
        'action' => $action
    ];
}
```

Htjeli bi smo izbjegići takav način pozivanja. Željeli bi smo pozivati sa:

```
get('/login', 'AuthController', 'showLogin');
```

ili

```
post('/login', 'AuthController', 'login');
```

Definirat čemo metodu `get`:i post u kojima čemo hardkodirati taj zadnji \$method parametar.

```
public function get(string $path, string $controller, string $action): void
{
    $this->addRoute($path, $controller, $action, 'GET');
}

public function post(string $path, string $controller, string $action): void
{
    $this->addRoute($path, $controller, $action, 'POST');
}
```

Dakle iz obje rute pozivamo `addRoute`. Kako ne želimo da se addRoute poziva van klase, umjesto public napisat čemo `private function addRoute`. Sada je možemo koristiti samo unutar klase. get i post su izložene prema van. Ono što definira da neka klasa mora definirati neke funkcionalnosti je interface. Sa interface prema van, Router isključivo će davati get i post prema van. U interface čemo staviti `get` i `post` metode i kada radimo instanciranje sa `getInstance` nećemo vratiti `Router`nego čemo vratiti objekt koji ima implementiran interface RouterInterface. Sada će definicija metode glasiti:

```
public static function getInstance(): RouterInterface
```

U src direktoriju otvorit čemo poddirektorij `Contracts`. Često čemo vidjeti umjesto `Interfaface`, `Contracts` jer se smatra da ste potpisali ugovor s interface-om, kao što se u ugovoru kaže da će se obavezati, tako i u interface-u. Ovdje u `src\Contracts` čemo otvoriti `RouterInterface.php` datoteku:

```
<?php

namespace App\Contracts;

interface RouterInterface
{
    public function get(string $path, string $controller, string $action): void;
    public function post(string $path, string $controller, string $action): void;
    public function dispatch(): void;
}
```

`get` i `post` su ovako izloženi (engl. expose) prema van jer čemo preko Router-a izložiti samo `get` i `post` prema van. Kada kažemo na Routeru da implementira taj inteface, potpisali su ugovor. Router mora poštivati što on definira i to implementira. S druge strane to daje potpis objektu, tako da kad ga izložimo prema van, na RouterInterface imamo `get` i `post`.

Vratimo se u `Router.php` i promijenimo deklaraciju klase Router. Sada čemo imati:

```
class Router implements RouterInterface
```

On sada ne zna gdje je, pa mora reći `use App\Contracts\RouterInterface`

Nakon toga ako kliknemo iznad `RouterInterface`, vidjet ćemo da nismo implementirali metodu post.

Ako pogledamo što klasa vraća, to je iz metode `getInstance`, `self::router`. Ne želimo u potpisu imati klasu nego njezinu apstrakciju. Apstrakcija te klase je RouterInterface i promijenit ćemo u .

```
public static function getInstance(): RouterInterface
```

Sada smo zaštitili metode jer mi šaljemo interface a ne samu klasu za komunikaciju s drugima.

`get` i `post` su izloženi prema van uz pomoć interface a one se oslanjaju na metodu `addRoute` koja se nalazi unutar same klase `Router`.

Sada trebamo napisati addRoute. U `$uri` konkateniramo postojeću putanju i `$path` koji dolazi iz poziva. Nakon toga vežemo za jedan ključ i za drugi ključ.

GitHub Copilot predložio je ovaku metodu koja nije dobra:

```
private function addRoute(string $path, string $controller, string $action,
string $method): void
{
    $uri = APP_URL . $path;
    $this->routes[$uri] = [
        'controller' => $controller,
        'action' => $action
        'method' => $method
    ];
}
```

Ovime bi smo dobili dva ista ključa u višedimenzionoj matrici:

```
[

    '/login' => [
        'controller' => 'AuthController',
        'action' => 'login',
        'method' => 'GET'
    ],
    '/login' => [
        'controller' => 'AuthController',
        'action' => 'login',
        'method' => 'POST'
    ]
]
```

Oni gaze jednu rutu preko druge.

Treba napraviti ovako:

```
private function addRoute(string $path, string $controller, string $action,
string $method): void
{
    $uri = APP_URL . $path;
```

```
$this->routes[$method][$uri] = [
    'controller' => $controller,
    'action' => $action
];
}
```

Sada ćemo za svaki od njih puniti ovako:

```
[
    '/GET' => [
        '/login' => [
            'controller' => 'AuthController',
            'action' => 'login'
        ]
    ],
    '/POST' => [
        '/login' => [
            'controller' => 'AuthController',
            'action' => 'login'
        ]
    ]
]
```

Sada ovo funkcioniра. Prvo ide metoda kao ključ. /GET je jedna dimenzija a /POST je druga.

Ovo je primjer kako Copilot pogriješio.

Evo dosadašnjeg cijelog [Router.php](#):

```
<?php

namespace App\Services;

use App\Contracts\RouterInterface;
use App\Exceptions\NotFoundException;

class Router implements RouterInterface
{
    private array $routes = [];
    private static ?RouterInterface $router = null;

    private function __construct(){}
    private function __clone(){}

    public static function getInstance(): RouterInterface
```

```
{  
    if (self::$router === null) {  
        self::$router = new Router();  
    }  
  
    return self::$router;  
}  
  
private function addRoute(string $path, string $controller, string $action,  
string $method): void  
{  
    $uri = APP_URL . $path;  
    $this->routes[$method][$uri] = [  
        'controller' => $controller,  
        'action' => $action  
    ];  
}  
  
public function get(string $path, string $controller, string $action): void  
{  
    $this->addRoute($path, $controller, $action, 'GET');  
}  
  
public function post(string $path, string $controller, string $action): void  
{  
    $this->addRoute($path, $controller, $action, 'POST');  
}  
  
public function dispatch(): void  
{  
    $method = $_SERVER['REQUEST_METHOD'];  
    $uri = APP_ROOT_URL . explode('?', $_SERVER['REQUEST_URI'])[0];  
  
    if (! isset($this->routes[$method][$uri])) {  
        throw new \Exception('Route not found', 404);  
    }  
  
    $controller = $this->routes[$method][$uri]['controller'];  
    $action = $this->routes[$method][$uri]['action'];  
  
    $controllerInstance = new $controller();  
    $controllerInstance->$action();  
}  
}
```

Kreirali smo logiku Router-a kojim ćemo moći definirati sve rute u našoj aplikaciji i Router će ih zapisati u tu matricu. Prilikom dolaska prvog korisnika na aplikaciju stvorit će se Router objekt. Napunit će se podacima o svim rutama i svi drugi korisnici kada dolaze koriste isti objekt koji je napunjen rutama. To je benefit singltona.

Pokazat ćemo u `web.php` kako će se puniti Router.

Napraviti ćemo `web.php` u direktoriju `routes`:

```
<?php

use App\Services\Router;

$router = Router::getInstance();
$router->get('/', 'App\Controllers\HomeController', 'index');
$router->get('/contcts', 'App\Controllers\ContactsController', 'index');
$router->post('/contacts/create', 'App\Controllers\ContactsController', 'store');

echo '<pre>';
var_dump($router);
```

Ovdje imamo :: što je [operator za rješavanje opsega \(engl. scope resolution operator\)](#). Kao što smo već govorili, Koristi se za pristup statičkim metodama, konstantama ili svojstvima iz klase, kao i za pristup metodama i svojstvima iz instanci klase kada se koristi unutar konteksta klase.

Ovdje se poziva statička metoda `getInstance` iz klase `Router`. Ova metoda koju smo već objasnili implementira Singleton obrazac, gdje se osigurava da postoji samo jedna instanca klase Router tokom trajanja aplikacije. Dakle, `getInstance` vraća tu jedinu instancu.

Ovdje metoda `get` iz instance Router postavlja rutu za HTTP GET zahtjev. To znači da kada korisnik posjeti osnovnu URL putanjу /, pozvat će se metoda `index()` iz klase `HomeController`.

Kako još nemamo kontrolere, nešto smo iznad upisali.

Kada pokrenemo <http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM/>, vidjet ćemo:

```
C:\xampp\htdocs\Algebra\Napredni PHP\AlgebraCRM\routes\web.php:11:
object(App\Services\Router) [4]
  private array 'routes' =>
    array (size=2)
      'GET' =>
        array (size=2)
          'http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM/' =>
            array (size=2)
              ...
              'http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM/contcts' =>
                array (size=2)
                  ...
                  'POST' =>
```

```
array (size=1)
    'http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM/contacts/create'
=>
    array (size=2)
        ...
/Algebra/Napredni%20PHP/AlgebraCRM/
```

Vidimo `GET` i `POST`, vidimo `$path`, `$controller` i `$action`. Dakle funkcionira matrica iz `web.php`.

Dispatcher

Kada se napuni matrica sa svim rutama, zadatak dispatcher-a će biti da pokupi request URI iz super globalne varijable `$_SERVER`, pokupi metodu (možemo saznati i to) i u ovoj matrici na temelju URI i metode pronađe stavku u kojoj se nalaze kontroler i action. Dispatcher kada pronađe što treba (engl. match) pozove tu metodu na kontroleru. Dispatcher će instancirati kontroler u objekt, na tom objektu će pozvati metodu.

Dakle sada na Routeru trebamo napisati dispatcher-a koji će moći instancirati `Controller` i pozvati metodu na kontroler.

Skidanje sa GitHub-a

Predavač je ovo stavio na github.com/tkelcec-algebra/algebra-crm. Možemo skinuti ZIP ali moramo biti pozicionirani u direktorij AlgebraCRM. Još bolje je otici u Git Bash a zatim provjeriti da lis smo u pravom direktoriju s `pwd`. Nakon toga potrebno je isprazniti direktorij sa `rm -rf *`. Sada možemo klonirati GitHub repozitorij:

```
git clone https://github.com/tkescec-algebra/algebra-crm .
```

Točkica na kraju znači da klonirani repozitorij bude smješten u tekući direktorij.

Kada raspakujemo (ili kloniramo) treba pokrenuti `composer install` jer nema na GitHubu direktorija `vendor`. To znači na nema ni u `composer` direktoriju `autoloader.php`. Bez toga aplikacija neće raditi.

Po commitevima je moguće pratiti što se mijenjalo.

Ponovimo ukratko što smo do sada radili. Klijent radi na browser-u zahtjev (engl. request), to dolazi na Apache server. U `.htaccess` napravili smo redirekciju i hvatamo sve zahtjeve, tj. redirektamo sve prema `index.php`. `index.php` uključi autoloader, uključi konfiguraciju i Router gdje su naše rute (`web.php`). Kada napravimo dump ruta iz `web.php`, vidimo matricu sa rutama. Vidimo da su razdjeljene na `GET` i `POST` metode. Idući korak bio bi da Router optpremi (engl. dispatch) zahtjev koji je pristigao prema nekom kontroleru. Idući korak bi bio u `web.php` pozivanje `$router` objekt i na njemu pozivanje dispatch:

```
$router->dispatch();
```

Za sada još nemamo metodu `dispatch`. U prvom koraku dispatchera trebamo saznati koja metoda je u pitanju. `dispatch` ćemo pisati u `Router.php`. U `RouterInterface.php` dodat ćemo da postoji taj dispatcher kojeg možemo pozvati izvana, dakle mora biti public.

```
public function dispatch(): void;
```

`dispatch` neće vraćati ništa ali će njegova zadaća biti da na temelju matrice pronađe informaciju koji kontroler i koja metoda su zaduženi za request koji se dogodio. `dispatch` na temelju requesta koji je pristigao na server, uzeće ih i pokušati pronaći u matrici. Ako ih pronađe, otpremiti (engl. dispatch) će ih Kontroleru. Kontroler je ništa drugo nego klasa. Dakle instancirat će klasu u objekt. Kada imamo objekt, možemo pozvati na tom objektu metodu. To je cilj.

Kod objektnog programiranja, kada imamo neku klasu imamo dva načina: da instanciramo objekt ili da instanciramo `static` metodu. Mi ćemo u našem slučaju, ići pristupom da su Kontroleri svi klase sa metodama koje se mogu instancirati u objekt i onda se mogu pozivati te metode.

Cilj je napisati dispatch u `RouterInterface.php` jer se `Router` u `Services` direktoriju buni. Idemo u `Router.php` u `Services` napisati metodu `dispatch`. Ovaj PHP kod predstavlja jednostavan mehanizam za rutiranje (routing) HTTP zahtjeva na odgovarajuće Kontrolere i akcije (metode) u aplikaciji. Evo detaljnog objašnjenja:

```
public function dispatch(): void
{
    $method = $_SERVER['REQUEST_METHOD'];
    $uri = APP_ROOT_URL . explode('?', $_SERVER['REQUEST_URI'])[0];
```

`$method`: Ova linija dohvaća HTTP metodu zahtjeva (npr. `GET`, `POST`, itd.) iz `$_SERVER` superglobalne varijable `$_SERVER`. Ta informacija je bitna zato što imamo pristup u `Router.php` rutama. Na temelju ove `$method` možemo reći npr. `var_dump($this->routes[$method])`. Time iz zahtjeva uhvatimo ime metode i postavimo ga kao ključ u matricu rute kako bi vratio vrijednosti koje su vezane za taj ključ.

Vidimo odvojene `GET` i `POST` u matrici. Kada pristigne neki zahtjev (engl. request), u `web.php` u Router-u imamo `$router->dispatch();` i ako se zahtjev dogodio na `/` ili na `/contacts` ili na `/contacts/create`, poanta je da `Router.php` uhvati metodu a to će znati tako da `put`, odnosno metoda `get` zapiše `$path`, `$controller` i `$action`.

`$uri`: Ovo je drugi korak. Ova red dohvaća URI zahtjev (Uniform Resource Identifier) pomoću superglobalne varijable `$_SERVER`. `$_SERVER['REQUEST_URI']` sadrži cijeli URI koji je klijent (browser) posao serveru tj. u mom slučaju `/Algebra/Napredni%20PHP/AlgebraCRM/` bez `http://localhost:8081` što možemo lako nadodati. Međutim, ako otvorimo ovakav link `http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM/?test=test` dobijemo `/Algebra/Napredni%20PHP/AlgebraCRM/?test=test`. Želimo se riješiti dijela iza `?`. Da se toga riješimo koristit ćemo `explode('?', $_SERVER['REQUEST_URI'])[0]` kako bi se izdvojio samo dio URL-ja prije upita (query stringa) ako postoji. Dakle string razdvajamo u elemente matrice. Prvi argument je separator. Pod nultim indeksom će biti informacija o našem URI. Sada u našem URI nema te informacije, dok u linku postoji. Ovo je bitno zato što ako probamo npr.

```
var_dump($this->routes[$method][$uri]);
```

dobit ćemo grešku. Problem je ako pogledamo `APP_URL` definiran u `app.php`:

```
define('APP_URL', 'http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM' );
```

koji je hardkodiran. Taj podatak se treba dinamički mijenjati i uklonit ćemo ga za sada. To se može riješiti kroz `.env` datoteku (Pohranjeni a pohranu konfiguracijski podaci u obliku parova ključ-vrijednost). Sada red glasi:

```
define('APP_URL', '/Algebra/Napredni%20PHP/AlgebraCRM' );
```

Sada ako pogledamo `http://localhost:8081/Algebra/Napredni%20PHP/AlgebraCRM/?test=test` radi.

Da smo ostavi redi sa `http://localhost:8081`, onda bi kod `$uri` ispred trebali dodati `„http://localhost:8081“` i ostatak spojiti sa `..` i opet bi trebalo raditi.

Još bolje rješenje je da u `app.php` stavimo sljedeće:

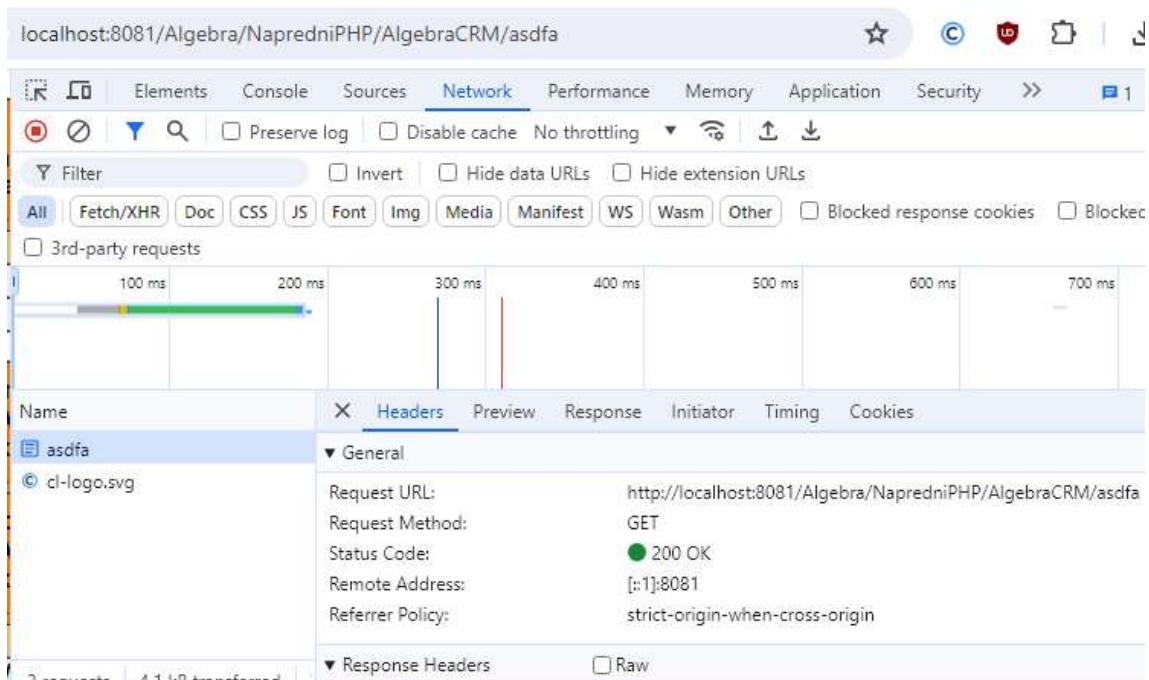
```
define('APP_ROOT_URL', 'http://localhost:8081');
define('APP_URL', APP_ROOT_URL . '/Algebra\NapredniPHP\AlgebraCRM');
```

Sada se u [Router.php](#) možemo pozvati na [APP_ROOT_URL](#).

Zatim se dodaje [APP_ROOT_URL](#), što je konstanta ili varijabla koja sadrži osnovni URL aplikacije.

Sada želimo iz te matrice svih ruta koje smo napunili u [web.php](#), da PHP pronađe informaciju o kontroleru, o akciji i metodi. To ustvari želimo napraviti sa dispatcher-om. Ako unesemo neku rutu koju nema, program javlja grešku: [Undefined array key](#). Ako nemamo niti jednu [get](#) metodu, onda će već i prije puknuti. Napravit ćemo sljedeće:

```
if (! isset($this->routes[$method][$uri])) {  
    echo '404';  
    exit;  
}
```



Vidimo da je Status kod [200](#) a dogodio se [404](#) - Stranica nije nađena. Dakle imamo krivi status, bez obzira što ispišemo. Status kodove treba razumjeti - treba potražiti na Google http status codes. Najzgodnija stranica je možda [HTTP response status codes](#) od Mozilla.

Pod [404](#) piše:

„Server ne može pronaći traženi resurs. U browzeru to znači da URL nije prepoznat. U API-ju to također može značiti da je krajnja točka važeća, ali sam resurs ne postoji. Serveri također mogu poslati ovaj odgovor umjesto odgovora [403 Zabranjeno za gledanje](#) kako bi sakrili postojanje resursa od neovlaštenog klijenta. Ovaj kod HTTP odgovora vjerojatno je najpoznatiji jer se često pojavljuje na webu.“

Ne bi trebalo biti ni dozvoljeno otvoriti link koji je predviđen za [POST](#) metodu, [GET](#) metodom. Tu bi npr. trebali dobit grešku [405 Method Not Allowed](#):

„[HyperText Transfer Protocol \(HTTP\) 405](#) Metoda nije dopuštena označava da server prepoznaće metodu zahtjeva (engl. request), ali ciljni resurs ne podržava ovu metodu.

Server mora generirati polje zaglavlja s [405](#) statusom koda. Polje mora sadržavati popis metoda koje ciljni resurs trenutno podržava.“

The screenshot shows the Network tab in the Chrome DevTools. A single request named 'create' is listed. The 'Headers' tab is selected. The 'General' section shows the following details:

- Request URL: <http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/contacts/create>
- Request Method: GET
- Status Code: 200 OK
- Remote Address: [::1]:8081
- Referrer Policy: strict-origin-when-cross-origin

The 'Response Headers' section shows:

- Connection: Keep-Alive
- Content-Length: 2669
- Content-Type: text/html; charset=UTF-8
- Date: Mon, 08 Jul 2024 14:26:17 GMT
- Keep-Alive: timeout=5, max=100

At the bottom left, it says '2 requests | 4.1 kB transferred'.

I ovdje se generira statusni kod 200, što je također pogrešno. Mi prvo gledamo [\\$method](#) a ne URI.

Treba napisati drugačije:

```
if (! isset($this->routes[$method][$uri])) {
    throw new \Exception('Route not found', 404);
}
```

Vidimo da [Exception](#) prima [string \\$message](#) i [int \\$code](#).

The screenshot shows a code editor with the file 'Router.php' open. The cursor is positioned over the word 'Exception'. A tooltip provides the following information:

Exception
Exception is the base class for all Exceptions.
<?php
class Exception implements Throwable { }
@link <https://php.net/manual/en/class.exception.php>

The code in the editor shows the implementation of the [__construct](#) method:

```
public function __construct(string $message = "", int $code = 0, Throwable|null $previous = null) { }
```

```

if (! isset($this->routes[$method][$uri])) {
    throw new HttpNotFoundException();
}

```

Provjera rute: Ovdje se provjerava postoji li definirana ruta za trenutni HTTP metod i URI. Ako u definiranim rutama (`$this->routes`) ne postoji ruta za određeni metod i URI, baca se iznimka `HttpNotFoundException`. Ova iznimka se obično koristi kada aplikacija ne može pronaći odgovarajuću rutu za zadani zahtjev.

```

$controller = $this->routes[$method][$uri]['controller'];
$action = $this->routes[$method][$uri]['action'];

```

Dohvaćanje kontrolera i akcije: Ako je ruta pronađena, dohvaćaju se Kontroler i akcija povezani s tom rutom iz `$this->routes`. Svaka ruta je definirana sa svojim kontrolerom i akcijom koja se izvršava kada se ruta poklapa s primljenim zahtjevom.

```

$controllerInstance = new $controller();
$controllerInstance->$action();

```

Instanciranje kontrolera i pozivanje akcije: Stvara se nova instanca kontrolera koristeći dobiveni naziv kontrolera (`$controller`). Zatim se poziva odgovarajuća metoda na tom kontroleru (`$action`). Ova linija izvršava funkcionalnost specifičnu za tu rutu, obično obrađujući i generirajući odgovor za klijenta.

Ovaj kod implementira osnovni mehanizam rutiranja zahtjeva u PHP-u, gdje se na temelju HTTP metode i URI-ja određuje koji kontroler i koja akcija će biti pozvani za obradu zahtjeva.

Evo cijelog koda:

```

public function dispatch(): void
{
    $method = $_SERVER['REQUEST_METHOD'];
    $uri = APP_ROOT_URL . explode('?', $_SERVER['REQUEST_URI'])[0];

    if (! isset($this->routes[$method][$uri])) {
        throw new HttpNotFoundException();
    }

    $controller = $this->routes[$method][$uri]['controller'];
    $action = $this->routes[$method][$uri]['action'];

    $controllerInstance = new $controller();
    $controllerInstance->$action();
}

```

`dispatch()` radi za sve rute koje smo ubacili u `$router` matricu. Router kada se pokrene, distancira `$router` objekt. On je singleton. On puni matricu sa podacima. `dispatch()` pokreće priču tako da iz zahtjeva izvuče informacije, provjeri u matrici, instancira kontroler i pozove metodu na kontroler.

Kontroler

Idemo otvoriti direktorij `Controllers` i datoteku `Controllers.php`. Baza (početak) će biti `abstract`. Ne želimo moći ovu klasu instancirati. Cilj je da svaki kontroler koji gradimo, naslijedi ovu klasu. Zato dignemo nadređenu (engl. parent) klasu koja je kontroler a podređene (engl. child) klase je nasljeđuju. Dakle ovu klasu ne možemo instancirati u objekt.

Otvorit ćemo u direktoriju Controllers datoteku `HomeController.php`.

```
<?php

namespace App\Controllers;

class HomeController extends Controller
{
    public function index()
    {
        echo 'Hello, word!';
    }
}
```

Kada pogledamo, zahtjev (engl. request) dolazi na Apache server. Apache spusti na `index.php`. `index.php` upali Router tj. `web.php` rute. Napunimo matricu (engl. array) sa svim rutama. Nakon toga pozovemo dispatcher. Dispatcher uzme ulazni zahtjev (engl. request), ekstrahira podatke iz njega, provjeri u matrici podudaranje. Ako se podudara s nečim izvuče kontroler, instancira ga u objekt i nad tim objektnom pozove metodu. Metodu isto izvuče iz matrice. Sada kontroler preuzima ulogu koja mu je dodijeljena.

Sada kada smo ovo uspjeli povezati ostali su nam Pogledi (engl. View) i Model za obraditi MVC.

Prije nastavka, pogledat ćemo još jednom `web.php` i `$router` da ljepeše izgleda:

Predavač kaže da mu se ne sviđa ovakvo rješenje u `web.php` jer ne izgleda lijepo sa `$router`. i da ćemo to popraviti. Ne želi raditi sa instanciranjem routera.

Obrazložio je i zašto. Recimo da odlučimo promijeniti namespace kontrolerima. Pojavit će se problem.

```
$router = Router::getInstance();
$router->get('/', 'App\Controllers\HomeController', 'index');
$router->get('/contcts', 'App\Controllers\ContactsController', 'index');
$router->post('/contacts/create', 'App\Controllers\ContactsController', 'store');
```

Prva stvar, čak i IDE (engl. Integrated development environment, tj. integrirana razvojna okolina) kao što je PHP Strom neće razriješiti taj problem (na jednom mjestu smo promijenili ime pa će sam IDE promijeniti na drugom). Isto tako neće razriješiti promjenu u stringovima, kao što je ovdje slučaj. Često puta neće pronaći u stringu informaciju i napraviti rename. Vi mijenjate ime namespace-a a ovo morate ručno.

PHP ima rješenje za ovo. To je **Fully qualified name (FQN)**, tj. **potpuno kvalificirano ime**. je način referenciranja klase, funkcija ili konstanti koristeći njihov puni naziv, uključujući potpuni namespace. To omogućava preciznu identifikaciju i izbjegavanje sukoba imena kada se koristi više klase, funkcija ili konstanti s istim imenom ali iz različitih namespace-ova. Potpuno kvalificirana imena uvijek se razrješuju u ime bez vodećeg namespace separatora. Na primjer, `\A\B` se razrješuje u `A\B`. Ovo je identifikator sa namespace separatorom koji počinje namespace separatorom, kao što je `\Foo\Bar`. Namespace `\Foo` također je potpuno kvalificirano ime.

On želi da možemo reći:

```
$router->get('/', HomeController::class, 'index');
```

`HomeController::class`: Ovo je specijalna konstanta koja vraća puni naziv klase `HomeController` kao string. Ova sintaksa se često koristi kod definiranja ruta ili sličnih operacija gdje je potrebno proslijediti naziv klase. Također koristi `::` za pristup konstanti `class`. Ova naredba znači da svaka klasa u PHP može vratiti FQN tako da pozove `::class` skupa sa namespace. Ovdje smo pozvali statičku metodu `get` iz klase `Route`. Drugi parametar `HomeController::class` koristi operator `::` za pristup specijalnoj konstanti `class`, koja vraća puni naziv klase `HomeController` kao string.

Operater `::` se u PHP-u koristi na nekoliko načina, uključujući:

- pristupanje statičkim metodama (`KlasaIme::staticMetoda()`)
- pristupanje statičkim svojstvima (`KlasaIme::$staticSvojstvo`)
- pristupanje klasnim konstantama (`Klasaime::KONSTANTA`)
- pristupanje posebnoj `class` konstanti koja vraća puni naziv klase (`KlasaIme::class`)

Ovaj kod koristi operater `::` za pristupanje statičkoj metodi `getInstance` i posebnoj konstanti `class`.

Korištenje FQN osigurava da PHP točno zna koju klasu, funkciju ili konstantu treba koristiti, čime se izbjegavaju sukobi imena. Ovakva organizacija pomaže u boljoj organizaciji koda, omogućavajući jasno razdvajanje klase, funkcija i konstanti po namespace-ovima. Ako promijenite namespace-ove, morate ažurirati sve `use` izjave i reference u kodu kako bi odražavale nove namespace-ove.

Ovo možemo riješiti pomoću [Fasade obrazca](#) (engl. pattern). Fasada prema van će biti ono što koristi programer ovdje iznad a ono što je učahureno je ono što je na vrhu stranice prije. Daje se pristup funkcionalnostima preko Facade. Ovaj primjer implementacije Facade uzorka u PHP-u demonstrira kako Facade može pojednostaviti složene operacije pozivom na više različitih klasa. Facade će kreirati objekt Routera i onda će izložiti (engl. expose) preko statičke metode `get` pristup Router objektu i njegovoj metodi `get`. Isto vrijedi i za `post` metodu. Na taj način želimo napuniti svoju matricu sa svim rutama. Rute će biti preglednije i bolje će se razumjeti. Težimo pisanju čistog koda. U ovom slučaju Facade obrazac će nam to omogućiti.

Sve funkcionalnosti u Laravelu, u koru Laravel-a su dostupne u aplikaciji bilo gdje preko fasada. Oni su ubrizgali (engl. inject) sve te servise u service container tako da su unutra upucali fasade koji imaju pristup servisima. U Laravelu je sve dostupno upravo preko fasada.

Fasade

U `src` poddirektoriju otvorit ćemo poddirektorij `Facades` otvorit ćemo `Route.php`.

Ovdje ćemo definirati 3 metode: `get`, `post` i `dispatch`, upravo one iz `web.php` u direktoriju `routes`.

Krenimo redom:

```
<?php

namespace App\Facades;
use App\Services\Router;

class Route
{
    public static function get(string $path, string $controller, string $action): void
    {
        //
    }
}
```

Unutar metode možemo napisati:

```
Router::getInstance()->get($path, $controller, $action);
```

ili

```
Router::getInstance()->...func_get_args();
```

Razlika između njih leži u načinu na koji se poziva metoda `get` na instanci klase `Router`.

U prvom redu, poziva se metoda `get` na instanci klase `Router` direktno, proslijedjujući joj tri argumenta: `$path`, `$controller` i `$action`. Tu metodu šaljemo Routeru, to je fasada. Na fasadi pozovemo metodu `get`, fasada poziva stvarnu funkcionalnost Routera, stvara objekt i nad tim objektom poziva metodu `get`.

Drugi red, koristi magičnu metodu `__call` za poziv metode `get`. Metoda `__call` se automatski poziva kada se pokuša pozvati metoda koja ne postoji ili nije dostupna u vidljivom kontekstu. `func_get_args()` vraća niz svih argumenata proslijedjenih metodi `get` u klasi `Route`. `...func_get_args()` je ugrađena (core) funkcija koja sadrži listu argumenata tj. gleda gdje je ta funkcija pozvana. Ona pokupi argumente pristigle u parametre i stavi ih u matricu. Zapakuje tu matricu i proslijedi svaki element kao pojedinačni argument metodi `get` na instanci `Router`. `...` uzima elemente matrice i predaje ih po parametrima. Dakle oba pristupa rade. U ovom slučaju je jasno koji parametri su pozvani pa ćemo koristiti prvi red.

Vratimo se na `web.php`. Sada izgleda čitkije:

```
<?php

use App\Controllers\HomeController;

Route::get('/', HomeController::class, 'index');
Route::get('/contcts', 'App\Controllers\ContactsController', 'index');
Route::post('/contacts/create', 'App\Controllers\ContactsController', 'store');
Route::dispatch()
```

Idemo nazad na `Route.php`. Napisat ćemo metodu `post` koja je ista kao `get`:

```
public static function post(string $path, string $controller, string $action): void
{
    Router::getInstance()->post($path, $controller, $action);
}
```

Evo i metode `dispatch`:

```
public static function dispatch(): void
{
    try {
        Router::getInstance()->dispatch();
    } catch (HttpNotFoundException $e) {
        header($_SERVER['SERVER_PROTOCOL'] . ' 404 Not Found');
        echo $e->getMessage();
    }
}
```

Ovdje smo rješili iznimku jer `dispatch` može baciti iznimku. To nije generalna iznimka već vlastita iznimka `HttpNotFoundException`, znamo da je to 404 kod.

Mi pozovemo `Route::get` iz klase `Route`, metoda `get` a ona ispod haube pozove `Router`, stvori instancu `getInstance()` i u toj instanci pozove `get`.

U `web.php` trebamo dodati `use App\Facades\Route;`.

Ako sada pokrenemo sve radi i vidjet ćemo Hello, World!

Preko fasade dali smo pristup nekoj našoj funkcionalnosti u Routeru. Router sada možemo proširiti. Fasada daje pristup dijelovima kojima mi želimo za komunikaciju s drugim djelovima. Interface je potpis koji kaže ovo su metode koje možeš koristiti za komunikaciju s drugima a onda fasada to iskoristi to za jednostavnije korištenje tih funkcionalnosti jer nema instanciranja.

Fasade se inače koriste u kompleksnim servisima, kao u Laravelu, što ćemo i vidjeti.

Pogledajmo još jednom tko koga zove. Nakon zahtjeva, idemo na `index.php` koji ide na `web.php`. U `web.php` poziva se fasada `Route`, pozove metodu `get`. `get` pozove `Router` pa `getInstance`. `getInstance` vrati instancu `$router` i imamo objekt. Nad njom pozovemo metodu `get` (u `Route.php`). Metoda `get` primi podatke o putanji, kontroleru i akciji i pozove metodu `addRoute`. Mi nemamo izloženo prema van `addRoute`. Zaštitali smo je sa pristupnim modifikatorom `private`. Također, niti interface ne daje informaciju da router ima tu metodu, niti sama fasada može pristupiti toj metodi jer

je ona privatna. Iz `get` metode pozovemo `addRoute` napunimo informacijama našu matricu, nakon što smo je napunili sa svim mogućim routama naše aplikacije, ponovo pozovemo ponovo fasadu `dispatch` koji kaže pozovi `Router::getInstance() -> dispatch()`. `dispatch()` iz `Router.php` uzme `$method` uzmi URI, pitaj da li postoji `$method` iz `$uri`. Ako ga nema baci iznimku. Ako ga ima dohvati `$controller`, dohvati `$action`, pričamo o Full qualified class name. To nam je potrebno kako bi mogli instancirati kontroler i pozvati `$action`. `HomeController` je instanciran i pozvana je metoda `index()`.

Nemamo nigdje da smo ručno pozvali `new HomeController` pa `index()`, sve se događa dinamički, kroz Router. Router digne sve rute i pokrene dispatch.

Naš `dispatch()` iz `Route.php` potencijalno može generirati iznimke, počeli smo pričati oko toga. Prvo želimo prilagoditi izniku. Sve iznimke nasleđuju exception ali miželimo vlastiti exception. Za to ćemo otvoriti poddirektorij Exceptions i datoteku `HttpNotFoundException.php`.

Vidjeli smo kod generiranje iznimke da smo poslali 404 kod. Kada personiziramo Exception onda je logično da bude 404. Ako pogledamo servis `Router.php` kako smo to do sada napravili:

```
if (! isset($this->routes[$method][$uri])) {
    throw new \Exception('Route not found', 404);
}
```

Ovdje se možemo osloniti na ovaj kod ali u `catch` bloku moramo svaki puta reagirati (za 404, za 401 itd. i tako svaki puta sa `switch` ili `if-else` blokom svaki puta reagirati). Zato je bolje napisati `HttpNotFoundException.php` i automatski postaviti `$message` da je 'Not Found' a `$code` 404. Kada pozovemo `throw` možemo postaviti default poruku i kod, kao što je u kodu ispod. Prosljeđuje se roditeljskoj klasi a to je bazna klasa `Exception`, koja uvijek mora kod klase ovakvog tipa biti `extends`.

```
<?php

namespace App\Exceptions;

use Exception;

class HttpNotFoundException extends Exception
{
    public function __construct(string $message = 'Not Found', int $code = 404)
    {
        parent::__construct($message, $code);
    }
}
```

Poanta je da ne moramo reći `new \Exception` i poslati kod. U servisu `Router.php` izmjenimo kod i sada glasi:

```
if (! isset($this->routes[$method][$uri])) {
    throw new HttpNotFoundException();
}
```

Vidimo da nismo morali slati dodatne podatke.

U toj datoteci sada nedostaje use koji moramo dodati:

```
use App\Exceptions\HttpNotFoundException;
```

Sada kada imamo prilagođeni \Exception, doći ćemo u Route fasadu, gdje imamo definiranu metodu `dispatch()` kao `public static function dispatch()` jer želimo iz web.php nad Route pozvati `dispatch()` sa `Route::dispatch()`

Metoda sada glasi:

```
public static function dispatch(): void
{
    Router::getInstance()->dispatch();
}
```

Metoda sada glasi:

```
public static function dispatch(): void
{
    try {
        Router::getInstance()->dispatch();
    } catch (HttpNotFoundException $e) {
        header('HTTP/1.1 404 Not Found');
        echo $e->getMessage();
    }
}
```

Znamo da `dispatch` može generirati iznimku i to 404. Umjesto da to samo ispišemo, bitno je da kao odgovor umjesto status koda 200 vratimo 404. Da bi to napravili trebamo u header staviti:

```
header(' 404 Not Found');
```

Ovo će promijeniti zaglavje odgovora.

Upravo to možemo vidjeti iz Chroma - status kod se promijenio u 404 Not Found. Iduća slika je uvećan detalj.

Već smo prije spomenuli da se funkcija `header` u PHP-u se koristi za slanje HTTP zaglavlja (headers) direktno iz PHP skripte na klijent (obično web browser).

The screenshot shows the Google Chrome Network tab with the URL `localhost:8081/Algebra/NapredniPHP/AlgebraCRM/asdfaas`. The tab title is "Not Found". The Network tab is selected, showing a timeline from 200 ms to 1400 ms. A single request is listed under "Headers":

| Name | Value |
|-----------------|---|
| Request URL | <code>http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/asdfaas</code> |
| Request Method | GET |
| Status Code | 404 Not Found |
| Remote Address | [::1]:8081 |
| Referrer Policy | strict-origin-when-cross-origin |

Below the Headers tab, the "General" section is expanded:

| Name | Value |
|-----------------|---|
| Request URL | <code>http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/asdfaas</code> |
| Request Method | GET |
| Status Code | 404 Not Found |
| Remote Address | [::1]:8081 |
| Referrer Policy | strict-origin-when-cross-origin |

The "Response Headers" section is also expanded:

| Name | Value |
|----------------|-------------------------------------|
| Cache-Control | no-store, no-cache, must-revalidate |
| Connection | Keep-Alive |
| Content-Length | 9 |
| Content-Type | text/html; charset=UTF-8 |

Ako ostavimo ovako, vidjet ćemo u zaglavlju da se desio Status Code 404. To je dobra praksa kada dođe do situacije da nemamo Router koji zna obraditi zahtjev koji je pristigao iz URI, poanta je da se to preusmjeri na 404. To smo iskontrolirali na jednostavan način. Naš servis Router je rekao ako nisam uspio naći metodu i URI u našoj matrici svih ruta, izbacim iznimku `HttpNotFoundException`. Naša fasada pozove `Route::dispatch()` nad Fasadom. Fasada kaže znam da servis `dispatch()` može baciti iznimku.

U tom `if`-u treba dodati još:

```
echo $e->getMessage();
```

Koko još nismo radili Poglede (engl. View) nećemo ulaziti u detalje. Dalo bi se to još urediti. Laravel ima funkciju 404: `abort(404)` koji upravo to radi.

Potencijalni problem je protokol koji je hardkodiran sa: [HTTP/1.1](#).

HTTP (HyperText Transfer Protocol) je protokol koji se koristi za prijenos podataka na webu. Postoji nekoliko verzija HTTP protokola koje su evoluirale tijekom vremena kako bi poboljšale performanse, sigurnost i funkcionalnost. On može biti: HTTP/0.9, HTTP/1.0, HTTP/1.1, HTTP/2. HTTP/3 je u razvoju. Kombinacija HTTP-a s SSL/TLS protokolom za osiguranje komunikacije. HTTPS enkriptira podatke kako bi se zaštitili od presretanja i manipulacije. HTTP/1.1 preko TLS (HTTPS) je de facto standard za sigurnu komunikaciju na webu.

Moguće je namjesti da ne hardkodinamo ovu varijablu sa servera, nego je povučemo sa superglobalne `$_SERVER`. Evo konačne verzije metode `dispatch()` u `Route.php`:

```
public static function dispatch(): void
{
    try {
        Router::getInstance()->dispatch();
    } catch (NotFoundException $e) {
        header($_SERVER['SERVER_PROTOCOL'] . ' 404 Not Found');
        echo $e->getMessage();
    }
}
```

Ovdje nemamo problem sa promjernom servera i protokola, ne moramo mijenjati kod.

Odime smo napravili čitku arhitekturu. Početnici ne razumiju dobru praksu i ne drže se praila iako kod izgleda logično. Za par mjeseci nije moguće to raspetljati i budemo pogubljeni u svom kodu. Što je kompleksnije početnici imaju više `if-else` uslova. Ovakvom organizacijom razumljiva je organizacija datoteka i direktorija.

Pogledi

Trenutno u `HommeController.php` imamo samo metodu `index()` koja ispiše ‘Hello, World!’. Htjeli bi tu vratiti neki sadržaj (engl. content) ali u obliku nekog HTML sadržaja.

To bi trebalo biti u obliku:

```
public function index()
{
    $this->render('home');
}
```

Želimo metodu koja se zove render. Prvi argument će biti ime Pogleda a drugi argument će biti potencijalni podaci koji se trebaju prikazati na tom Pogledu ili ćemo pitati Model da nam trebaju podaci i on će ih dohvatiti. Te podatke ćemo vratiti kao nekakav objekt ili matricu. Prvi argument je obavezan. Drugi argument je opcionalan, možemo tražiti pogled bez da pošaljemo podatke. Pogled vraćamo korisniku. Ovu render metodu tj. Poglede pišemo najčešće u poddirektorij `src` ali možemo ih izvući kao odvojen poddirektorij `views`.

Ti pogledi su obične php datoteke koje u sebi imaju HTML kod. Cilj je da pogled pomoću `render` metode uključi neki od tih pogleda. U tom `views` poddirektoriju otvorit ćemo `home.php`. Poanta je da se ime stringa koji poziva `render()` metoda slaže s imenom pogleda u `views` direktoriju. U Laravelu se u navodnicima navodi npr. ‘`home.index`’ gdje home označava direktorij a `index` datoteku `index.php`.

Idemo napraviti taj `home.php`:

```
<h1>
    <?php echo $title; ?>
</h1>
```

Zadaća kontrolora Render je da na temelju imena uključi ovu PHP datoteku. To je odgovor korisniku.

Da nemamo autoloader, da nemamo `.htaccess` (ili da ga isključimo) koji preusmjerava stvari, do `home.php` mogli bi doći tako da kažemo `/views/home.php` tj.

<http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/views/home.php>.

Mi ne želimo da to ide tako. Naš `HomeController.php` mora uključiti tu datoteku. Sam server ne može vratiti tu datoteku. Metodu render moramo napisati u roditeljskoj klasi jer želimo da svi kontroleri imaju pristup toj metodi.

Otići ćemo u nadređenu klasu `Controller.php` u direktoriju `Controllers`.

```
<?php

namespace App\Controllers;

abstract class Controller
{
    protected function render(string $view, array $data = []): void
    {
        try {
            extract($data);
            require_once APP_ROOT . "/views/{$view}.php";
        } catch (\Throwable $e) {
            header($_SERVER['SERVER_PROTOCOL'] . ' 400 Bad Request');
            echo "View not found.";
        }
    }
}
```

Ova klasa je apstraktna, što znači da ne može biti instancirana direktno. Njena svrha je da bude naslijedena od strane drugih klasa koje će implementirati specifične funkcionalnosti.

Metoda render je zaštićena (protected), što znači da je mogu koristiti samo klase koje nasleđuju ovu apstraktну klasu. Ova metoda služi za prikazivanje (renderiranje) pogleda (view-a). Prima dva parametra:

- `string $view`: ime `view` datoteke koja treba biti učitana.
- `array $data`: asocijativna matrica podataka koji će biti ekstrahiran u varijable dostupne unutar view datoteke. Podrazumijevana vrijednost je prazna matrica.

Funkcija `extract` uzima asocijativnu matricu `$data` i kreira varijable od njegovih ključeva sa vrijednostima iz matrice. `require_once` učitava i izvršava `view` datoteku smještenu u direktoriju definiranom konstantom `APP_ROOT` (to je root definiran sa `define('APP_ROOT', dirname(__DIR__)`) iz `app.php`), poddirektorijem `views`, i imenom datoteke koje je navedena u `$view` parametru (`home.php`). Ako `view` datoteka ne postoji ili se dogodi neka druga greška, izvršit će se blok `catch`. U slučaju greške (`\Throwable`) obuhvaća sve vrste grešaka, uključujući `Exception`). Ne radi s `Exception`. HTTP odgovor će biti postavljen na 400 Bad Request i poruka "View not found." bit će prikazana

korisniku. U stvarnoj aplikaciji ne bi ovo ispisivali nego bi to negdje i pohranili. Funkcija `header` se koristi za slanje HTTP zaglavla klijentu. HTTP zaglavla daju informacije o odgovoru koji server šalje klijentu. U ovom slučaju, koristimo header da pošljemo specifičan statusni kod klijentu. Kada se dogodi greška u `try` bloku (npr. ako `view` datoteka ne postoji), kod ulazi u `catch` blok. Greška je uhvaćena kao instanca klase `\Throwable`, koja obuhvaća sve vrste grešaka i izuzetaka. `HomeController` nasljeđuje `Controller` (`class HomeController extends Controller`) i ima pristup `render` metodi, šalje joj home u prvom parametru. Mi uzimamo taj podatak i kažemo:

```
require_once APP_ROOT . "/views/{$view}.php";
```

`$_SERVER['SERVER_PROTOCOL']` vraća verziju HTTP protokola koji server koristi (npr. HTTP/1.1 ili HTTP/2.0).

Ovaj protokol se kombinira sa statusnim kodom 400 Bad Request da formira kompletan statusni odgovor. 400 Bad Request je HTTP statusni kod koji označava da je klijent poslao neispravan zahtjev, što je prikladno za situaciju kada tražena view datoteka ne može biti pronađen ili postoji neka druga greška u zahtjevima klijenta.

Sada ne možemo pristupiti direktno pozivanje mape `views` i `home.php` tada bi korisnik mogao pokretati što hoće. Ovako to nije moguće. Mi to kontroliramo.

Funkcija `extract`, kao što smo već govorili, uzima asocijativnu matricu i kreira varijable iz njegovih ključeva sa odgovarajućim vrijednostima. Matrica `$data` je asocijativna matrica koji se proslijeđuje funkciji `render`. Ključevi u matrici predstavljaju imena varijabli, a vrijednosti predstavljaju vrijednosti tih varijabli. Ako pogledamo `HomeController.php` u `Controllers` poddirektoriju gdje smo pozvali render, možemo u drugi argument staviti `['title' => 'Home Page']`:

```
$this->render('home', ['title' => 'Home Page']);
```

Primjetit ćete da je ključ u matrici s podacima identičan imenu varijable u `home.php` u `views`. To je `$title`. Kada bi promijenili ime ključa ili ime varijable, program ne bi radio i dojavio bi grešku. Ovaj ključ je završio na pogledu upravo sa `extract`.

Na taj način modeliramo podatke na Pogledu.

Za sada nam fali još Model.

Sada bez problema možemo napraviti novu stranicu. Ako pogledamo naše rute u `web.php` dodamo rutu za probu sa metodom contacts:

```
Route::get('/contacts', HomeController::class, 'contacts');
```

Promijenili smo rutu i dodamo metodu `contacts()` u `HomeController.php` uz postojeću metodu `index()`.

```
public function contacts()
{
    $this->render('home', ['title' => 'Contacts Page']);
}
```

Potrebito je još kreirati Pogled u `views` poddirektoriju `contacts.php`:

```
<h1>
<?php echo $title; ?>
```

```
</h1>

<form>
    <input type = "text" name="name" placeholder="Name">
    <input type = "email" name="email" placeholder="Email">
    <textarea name="message" placeholder="Message"></textarea>
    <button type="submit">Send</button>
</form>
```

Kada izgradimo MVC izgradnja aplikacije na web-u je jednostavna. Ne gubimo vrijeme na funkcionalnosti koje nisu vezane sa funkcionalnostima. Upravo je to prednost framework-a.

Modeli

Modeli su zaduženi za komunikaciju sa komunikaciju s bazama. Predavač je gledao da ne bude prekompleksno a da opet odradimo komunikaciju s bazom. Treba nam cijelo svojstvo (engl. feature) kojeg ćemo koristiti za rad s bazom a Modeli u radu s bazama podataka. Iskoristit ćemo već postojeći servis. Predavač je dodao par metoda `find`, `get` i `insert`. U biti to je postojeći database malo dorađen. Radi se s PDO, logika je ista. U primjeru database smo hardkodirali podatke a ovdje nećemo. Predavač je dao sljedeći kod:

```
<?php

namespace App\Services;

use PDO;
use PDOException;

class Database
{
    private static ?Database $instance = null;
    private PDO $conn;
    private string $table;
    private string $columns = "*";
    private array $where = [];

    private function __construct() {
        $this->connect();
    }

    private function __clone() {}

    private function connect() {
```

```

$dsn =
DB_DRIVER." :host=".DB_HOST.";dbname=".DB_NAME.";charset=".DB_CHARSET;
$options = [
    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
];

try{
    $this->conn = new PDO($dsn, DB_USER, DB_PASS, $options);
} catch (PDOException $e) {
    echo $e->getMessage();
}
}

private function reset(): void {
    $this->columns = "*";
    $this->where = [];
}

public static function getInstance(): Database {
    if (self::$instance === null) {
        self::$instance = new Database();
    }
    return self::$instance;
}

public function getConnection(): PDO {
    return $this->conn;
}

public function table(string $table): Database {
    $this->table = $table;
    return $this;
}

public function select(string|array $columns): Database {
    $this->columns = is_array($columns) ? implode(", ", $columns) : $columns;
    return $this;
}

public function where($column, $operator, $value = null): Database {
    if (func_num_args() === 2) {
        $value = $operator;
        $operator = "=";
    }
}

```

```

        }

        $this->where[] = [$column, $operator, $value];
        return $this;
    }

    public function get() {
        $sql = "SELECT $this->columns FROM $this->table";

        if (!empty($this->where)) {
            $whereParts = [];
            foreach ($this->where as $condition) {
                $whereParts[] = $condition[0] . " " . $condition[1] . " ?";
            }

            $sql .= " WHERE " . implode(" AND ", $whereParts);
        }

        $stmt = $this->conn->prepare($sql);

        $params = array_map(function($condition){
            return $condition[2];
        }, $this->where);

        $stmt->execute($params);

        $this->reset();

        return $stmt->fetchAll();
    }

    public function find($id, $column = "id"): ?array {
        return $this->where($column, $id)->get()[0] ?? null;
    }

    public function first(): ?array {
        return $this->get()[0] ?? null;
    }

    public function insert(array $data): bool|int
    {
        $this->columns = implode(", ", array_keys($data));
        $placeholders = implode(", ", array_fill(0, count($data), "?"));
        $sql = "INSERT INTO $this->table ($this->columns) VALUES ($placeholders)";
    }
}

```

```

$stmt = $this->conn->prepare($sql);
$this->reset();

if ($stmt->execute(array_values($data))) {
    return $this->conn->lastInsertId();
}

return false;
}
}

```

To će biti `Services\Database.php`.

Vidjet ćemo da se dijelovi koda crvene (tj podvlači ih Visual Studio Code). To su konstante koje ćemo definirati nakon objašnjenja kako ovo radi.

Idemo ažurirati `app.php` s konstantama i vratiti se analizi `Database.php`:

```

<?php

define('APP_NAME', 'Algebra CRM');

define('APP_ROOT', dirname( __DIR__ ) );
define('APP_PUBLIC', APP_ROOT . '/public');
define('APP_ROOT_URL', 'http://localhost:8081');
define('APP_URL', APP_ROOT_URL . '/Algebra/NapredniPHP/AlgebraCRM');

# Database configuration
define('DB_DRIVER', 'mysql');
define('DB_HOST', 'localhost');
define('DB_NAME', 'algebra_crm');
define('DB_USER', 'root');
define('DB_PASS', '');

```

Dodali smo konstante vezane za bazu podataka jer ih koristimo u `Database.php`, koju je pripremio predavč.

PHP kod iz `Database.php` predstavlja jednostavnu implementaciju klase za rad s bazom podataka koristeći PDO (PHP Data Objects). Klasa Database koristi Singleton dizajn uzorak, što znači da se samo jedna instanca klase može stvoriti i koristiti u cijeloj aplikaciji.

Datoteka `Database.php` implementira singleton klasu za rukovanje vezom prema bazi podataka i pruža osnovne metode za rad s bazom podataka. Ovo uključuje selekciju, umetanje i pretraživanje podataka. Pogledajmo detaljno kako funkcioniра ova klasa.

Ovim se definiraju namespace i potrebne klase iz PDO ekstenzije:

```

namespace App\Services;
use PDO;

```

```
use PDOException;
```

Ovi redovi uvoze PDO i PDOException klase iz PHP-a koje se koriste za rad sa bazom podataka i rukovanje greškama.

Ovaj red definira privatnu statičku varijablu koja čuva jedinu instancu klase **Database**:

```
private static ?Database $instance = null;  
private PDO $conn;  
private string $table;  
private string $columns = "*";  
private array $where = [];
```

\$instance: Sadrži jedinstvenu instancu Database klase (za singleton obrazac).

\$conn: PDO instanca za vezu prema bazi podataka.

\$table, **\$columns**, **\$where**: Koriste se za kreiranje SQL upita.

Konstruktor je privatna metoda koja je tako definirana kako bi se spriječili direktno stvaranje instanci. Poziva se **connect()** za uspostavljanje veze s bazom podataka. Kloniranje je onemogućeno s privatnom metodom **__clone**:

```
private function __construct() {  
    $this->connect();  
}  
  
private function __clone() {}
```

Metoda **connect** koristi parametre definirane konstantama (definirane su u **app.php** i iznad ovog objašnjenja) kako bi se spojila na bazu podataka:

```
private function connect() {  
    $dsn = DB_DRIVER."":host=".DB_HOST.";dbname=".DB_NAME.";charset=".DB_CHARSET;  
    $options = [  
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,  
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC  
    ];  
  
    try {  
        $this->conn = new PDO($dsn, DB_USER, DB_PASS, $options);  
    } catch (PDOException $e) {  
        echo $e->getMessage();  
    }  
}
```

Uspostavlja vezu s bazom podataka koristeći PDO. Konfigurira PDO da baca izuzetke i koristi asocijativne matrice kao zadani način dohvaćanja rezultata.

\$dsn će izgledati ovako: mysql:host=localhost;dbname=algebra_crm;charset=utf8

PDO: Inicijalizuje se pomoću \$dsn, DB_USER i DB_PASS, uz dodatne opcije koje definiraju način rukovanja greškama i način dohvaćanja rezultata.

Metoda reset vraća postavke upita na početne vrijednosti. Resetira varijable \$columns i \$where na njihove početne vrijednosti.

```
private function reset(): void {
    $this->columns = "*";
    $this->where = [];
}
```

Ova metoda osigurava da postoji samo jedna instanca klase Database (Singleton):

```
public static function getInstance(): Database {
    if (self::$instance === null) {
        self::$instance = new Database();
    }
    return self::$instance;
}
```

getInstance() vraća jedinstvenu instancu Database klase. Ako instanca ne postoji, kreira je.

getConnection(): Vraća PDO vezu:

```
public function getConnection(): PDO {
    return $this->conn;
}
```

Metode table i select omogućavaju definiranje tablice i stupaca koje se koriste u upitu:

```
public function table(string $table): Database {
    $this->table = $table;
    return $this;
}

public function select(string|array $columns): Database {
    $this->columns = is_array($columns) ? implode(", ", $columns) : $columns;
    return $this;
}
```

Dakle `table` postavlja naziv tablice i vraća trenutnu instancu klase radi ulančavanja metoda. `select` postavlja stupce koje treba odabrat i vraća trenutnu instancu klase.

Metoda `where` dodaje uvjete za filtriranje podataka:

```
public function where($column, $operator, $value = null): Database {
    if (func_num_args() === 2) {
        $value = $operator;
        $operator = "=";
    }
    $this->where[] = [$column, $operator, $value];
    return $this;
}
```

`where` dodaje WHERE uslove u SQL upit i vraća trenutnu instancu klase:

```
public function where($column, $operator, $value = null): Database {
    if (func_num_args() === 2) {
        $value = $operator;
        $operator = "=";
    }
    $this->where[] = [$column, $operator, $value];
    return $this;
}
```

`get` metoda sastavlja (kreira) i izvršava `SELECT` upit, te vraća rezultate kao matricu:

```
public function get() {
    $sql = "SELECT $this->columns FROM $this->table";

    if (!empty($this->where)) {
        $whereParts = [];
        foreach ($this->where as $condition) {
            $whereParts[] = $condition[0] . " " . $condition[1] . " ?";
        }

        $sql .= " WHERE " . implode(" AND ", $whereParts);
    }

    $stmt = $this->conn->prepare($sql);

    $params = array_map(function($condition) {
        return $condition[2];
    }, $this->where);
```

```

$stmt->execute($params);

$this->reset();

return $stmt->fetchAll();
}

```

Metode `find` i `first` koriste `where` i `get` metode kako bi pronašle specifičan zapis ili prvi zapis u rezultatu. `find`: Pronalazi zapis po ID-u (ili drugom stupcu), vraća prvi rezultat ili `null`. `first` vraća prvi rezultat upita ili `stup`.

```

public function find($id, $column = "id"): ?array {
    return $this->where($column, $id)->get()[0] ?? null;
}

public function first(): ?array {
    return $this->get()[0] ?? null;
}

```

Metoda `insert` sastavlja i izvršava `INSERT` upit, te vraća ID zadnje umetnute stavke ili false u slučaju neuspjeha:

```

public function insert(array $data): bool|int {
    $this->columns = implode(", ", array_keys($data));
    $placeholders = implode(", ", array_fill(0, count($data), "?"));
    $sql = "INSERT INTO $this->table ($this->columns) VALUES ($placeholders)";

    $stmt = $this->conn->prepare($sql);
    $this->reset();

    if ($stmt->execute(array_values($data))) {
        return $this->conn->lastInsertId();
    }

    return false;
}

```

Ova klasa pruža jednostavan način za interakciju s bazom podataka pomoću PDO-a, omogućavajući osnovne operacije kao što su `SELECT`, `INSERT`, i pretragu. Singleton uzorak osigurava da postoji samo jedna instanca klase koja se koristi tijekom cijelog životnog ciklusa aplikacije, što može biti korisno za:

- Učinkovitost: Spajanjem na bazu podataka samo jednom, smanjuje se opterećenje na serveru i ubrzava izvršavanje upita.
- Centralizacija pristupa bazi: Sva interakcija s bazom podataka ide preko jedne instance, što olakšava upravljanje vezom i upitima.
- Sigurnost i održavanje koda: Budući da je pristup bazi centraliziran, promjene se mogu provesti na jednom mjestu, čime se smanjuje mogućnost pogrešaka i povećava sigurnost koda.

Primjeri korištenja

Evo nekoliko primjera kako možemo koristiti ovu klasu:

- Dobivanje svih zapisu iz tablice:

```
$db = Database::getInstance();
$users = $db->table('users')->get();
```

- Dobivanje određenih stupaca:

```
$users = $db->table('users')->select(['id', 'name'])->get();
```

- Primjena where uvjeta:

```
$user = $db->table('users')->where('id', 1)->first();
```

- Umetanje novog zapisa:

```
$data = [
    'name' => 'John Doe',
    'email' => 'john@example.com'
];
$userId = $db->table('users')->insert($data);
```

Ovaj pristup omogućava jednostavnu, čitljivu i održivu manipulaciju s bazom podataka, smanjujući količinu koda potrebnu za obavljanje osnovnih operacija.

Otvaramo u direktoriju `Models` datoteku `Model.php`. Želimo imati neku roditeljsku klasu nad kojom će živjeti dva svojstva: instanca `Database` servisa `$db` (definiranog u `Database.php`) i `$table`. `$table` će diktriatiti svaki model koji extenda, npr. user model extend model gdje ćemo onda postaviti table je user.

```
<?php

namespace App\Models;

use App\Services\Database;

abstract class Model
{
```

```

protected Database $db;
protected string $table;

public function __construct()
{
    $this->db = Database::getInstance();
}
}

```

`models.php` sadrži `Model` klasu koja implementira `ModelInterface` i koristi `Database` klasu.

Definira prostor imena `App\Models`, što organizira klase unutar direktorija `Models`. Uvoze se `ModelInterface` iz `App\Contracts` namespace-a i `Database` klasa iz `App\Services` namespace-a. Ovo omogućava korištenje tih klasa unutar `Model` klase.

Ovim se definira apstraktna klasa `Model` koja implementira `ModelInterface`. Definira dva zaštićena (engl. `protected`) svojstva:

- `$db`: instanca `Database` klase, korištena za rad s bazom podataka. Možemo odlučiti da podređeni modeli ne smiju mijenjati `Database $db`, onda moraju biti `private`, ali onda moramo imati getter metodu. Ostaviti ćemo `protected` jer možda ćemo htjeti otvoriti neku drugu konekciju ili na neki drugi način, želimo moći stvoriti više database objekata. To sve možemo raditi sa setterom ali neka ostane ovako.
- `$table`: ime tablice povezane s modelom.

Konstruktor ove klase inicijalizira instancu `Database` klase koristeći Singleton uzorak. Ovime u apstraktnoj klasi `Model` dolazimo svaki puta do `Database` instance koja će se nalaziti u zaštićenom `$db` svojstvu.

Metoda `all()` koristi `$db` instancu za dohvati svih zapisa iz tablice specificirane u `$table` svojstvu.

`Model` klasa služi kao baza za sve modele u aplikaciji, pružajući osnovnu funkcionalnost za rad s bazom podataka. Implementira `ModelInterface`, što osigurava da sve klase koje nasleđuju `Model` imaju definirane metode koje su specificirane u tom interface-u.

Otvorit ćemo u `Models` datoteku `Model.php`:

```

<?php

namespace App\Models;

use App\Services\Database;
use App\Contracts\ModelInterface;

abstract class Model implements ModelInterface
{
    protected Database $db;
    protected string $table;
}

```

```

public function __construct()
{
    $this->db = Database::getInstance();
}

public function all(): array
{
    return $this->db->table($this->table)->get();
}
}

```

Klasa `Model` je apstraktna što znači da se ne može instancirati direktno. Treba je naslijediti i koristiti u konkretnim modelima. Implementacija interfejsa `ModelInterface` osigurava da sve metode deklarirane u interfejsu budu prisutne u klasi.

Konstruktor se poziva prilikom instanciranja objekta klase. Postavlja `$db` atribut na instancu `Database` klase koristeći Singleton pattern (`Database::getInstance()`).

Metod `all` vraća sve zapise iz tabele koja je definirana u `$table` atributu.

`table($this->table)`: Postavlja ime tabelice.

`get()`: Dohvaća sve zapise iz te tabelice.

Singleton pattern osigurava da postoji samo jedna instance klase `Database` tokom životnog vijeka aplikacije. Ovo je korisno za dijeljenje jedne konekcije prema bazi podataka među različitim dijelovima aplikacije.

Idemo na `User.php` koji se također nalazi u poddirektoriju `Models`:

```

<?php

namespace App\Models;

class User extends Model
{
    protected string $table = 'users';

    public function create(array $data): bool|int
    {
        $data['password'] = password_hash($data['password'], PASSWORD_BCRYPT);

        return
            $this->db
                ->table($this->table)
                ->insert($data);
    }
}

```

```

    }

    public function findByEmail(string $email): ?array
    {
        return
            $this->db
                ->table($this->table)
                ->where('email', $email)
                ->first();
    }
}

```

Datoteka `User.php` predstavlja konkretan model koji se koristi za rad sa korisnicima u bazi podataka. Ova klasa nasljeđuje osnovnu funkcionalnost iz Model klase i dodaje specifične metode za rad sa korisnicima.

User klasa nasljeđuje Model klasu. Na taj način dobiva sve metode i atribute Model klase, uključujući i konekciju sa bazom podataka.

Red `protected string $table = 'users'` definira ime tabele u bazi podataka koju ovaj model predstavlja. U ovom slučaju, tabela se zove `users`.

Metoda `create` prima asocijativnu matricu (engl. array) podataka koji će biti umetnuti u tablicu.

Prije umetanja, lozinka korisnika se hešira koristeći `password_hash` funkciju sa `PASSWORD_BCRYPT` algoritmom. Ovo osigurava sigurnost šifre.

Metoda zatim koristi `Database` servis da umetne podatke u tablicu i vraća rezultat umetanja, što može biti `true` ili `false` u slučaju neuspjeha ili ID novog zapisa u slučaju uspjeha.

Metoda `findByEmail` prima email adresu kao argument.

Koristi `Database` servis da pronađe prvi zapis u tablici `users` koji ima odgovarajući email.

Vraća podatke o korisniku kao asocijativnu matricu ili `null` ako korisnik sa danim emailom ne postoji.

Kako User model koristi Database servis

1. Umetanje korisnika (create metod):
 - Metoda `create` priprema podatke, hešira lozinku i koristi `Database` servis za umetanje podataka u `users` tablicu.
2. Pronalaženje korisnika po emailu (findByEmail metod):
 - Metoda `findByEmail` koristi `Database` servis da pronađe korisnika sa određenim emailom.

Kako bi se ove metode ispravno izvodile, oslanjaju se na funkcionalnosti `Database` servisa definiranih u `Database.php`. Ključni metodi koji se koriste su:

- **table(\$table)**: Postavlja trenutnu tabelu za upit.
- **insert(\$data)**: Umeće podatke u postavljenu tabelu.
- **where(\$column, \$value)**: Dodaje uslov za upit.
- **first()**: Vraća prvi rezultat upita.

`User.php` je konkretni model koji nasljeđuje osnovnu funkcionalnost iz Model klase i dodaje specifične metode za rad sa korisnicima. Korištenjem `Database` servisa, omogućava sigurno umetanje i pretragu korisnika u `users` tablici. Ovakva struktura olakšava upravljanje podacima o korisnicima i osigurava da se svi upiti prema bazi podataka vrše na konzistentan i siguran način.

Definiranje baze podataka

Sada ćemo definirati bazu koja ide uz ovaj projekt.

```
DROP TABLE IF EXISTS `users` ;
CREATE TABLE `users` (
  `id` int(11) NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `email` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
ALTER TABLE `users`
  ADD PRIMARY KEY (`id`);
ALTER TABLE `users`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT;
COMMIT;
```

Imamo tablicu `users` sa 4 polja: `id`, `name`, `email` i `password`.

U projektnoj grupi moguće je bazu distribuirati kao migracije. Migracije baza podataka su način za verzioniranje i upravljanje shemom baze podataka. One omogućavaju programerima da prate promjene u strukturi baze podataka kroz vrijeme, što olakšava rad na različitim verzijama aplikacije, kolaboraciju među članovima tima i implementaciju promjena u različitim okruženjima (razvoj, testiranje, produkcija). Kasnije ćemo objasniti Laravel migracije koje omogućavaju

- Verzioniranje baze podataka:** Migracije omogućavaju praćenje promjena u shemi baze podataka kao što su dodavanje, brisanje ili izmjena tablica i stupaca.
- Automatizacija promjena:** Omogućavaju automatsko primjenjivanje promjena u bazi podataka bez potrebe za ručnim pisanjem SQL naredbi.
- Kolaboracija:** Olakšavaju rad timova na istom projektu, omogućavajući svakom članu da primjeni iste promjene na svojim lokalnim bazama podataka.
- Povratne migracije:** Omogućavaju povratak na prethodnu verziju baze podataka ako nešto pođe po zlu.

"Code First" pristup u kontekstu migracija baze podataka odnosi se na razvoj baze podataka iz koda aplikacije. Umjesto da prvo ručno dizajnirate bazu podataka i zatim pišete kod koji koristi tu bazu, u "Code First" pristupu prvo pišete kod (npr. PHP klase koje predstavljaju entitete), a zatim koristite alat za migracije da automatski generira i ažurira shemu baze podataka.

Provjera radi li Model

Da bi smo vidjeli kako ovo radi izmjenit ćemo `HomeController` klasu, tako da privremeno ubacimo 3 reda od `$model` do `var_dump`:

```
class HomeController extends Controller
{
    public function index()
    {
        $model = new \App\Models\User();
        $users = $model->all();
        var_dump($users);

        $this->render('home', ['title' => 'Home Page']);
    }
}
```

Metoda `index()` unutar `HomeController` klase u našem PHP kodu služi za dohvaćanje svih korisnika iz baze podataka i prikazivanje rezultata, te za prikazivanje odgovarajućeg Pogleda (view). Evo detaljnog objašnjenja kako ova metoda radi:

Instancirali smo `User()` model (ili drugim riječima kreirali novu instancu) i User klasa nasljeđuje Model klasu koja je apstraktna klasa. Ova instanca omogućava pristup metodama definiranim u Model klasi, uključujući metodu `all()` koja bi trebala ako pogledamo u `Model.php` i metodu `all()` reći tablica koja je users napravi `get()`.

Pozivamo metodu `all()` iz Model klase preko `User` modela. Ova metoda koristi `Database` servis za dohvaćanje svih zapisa iz tablice users, tj. dohvaćanje svih korisnika. Ovaj korak podrazumijeva sljedeće:

- Database klasa koristi singleton obrazac za osiguranje jedne instance konekcije prema bazi.
- Metoda `all()` u Model klasi poziva metodu `get()` na Database instanci, koja izvodi SQL upit `SELECT * FROM users` i vraća rezultate kao asocijativnu matricu.

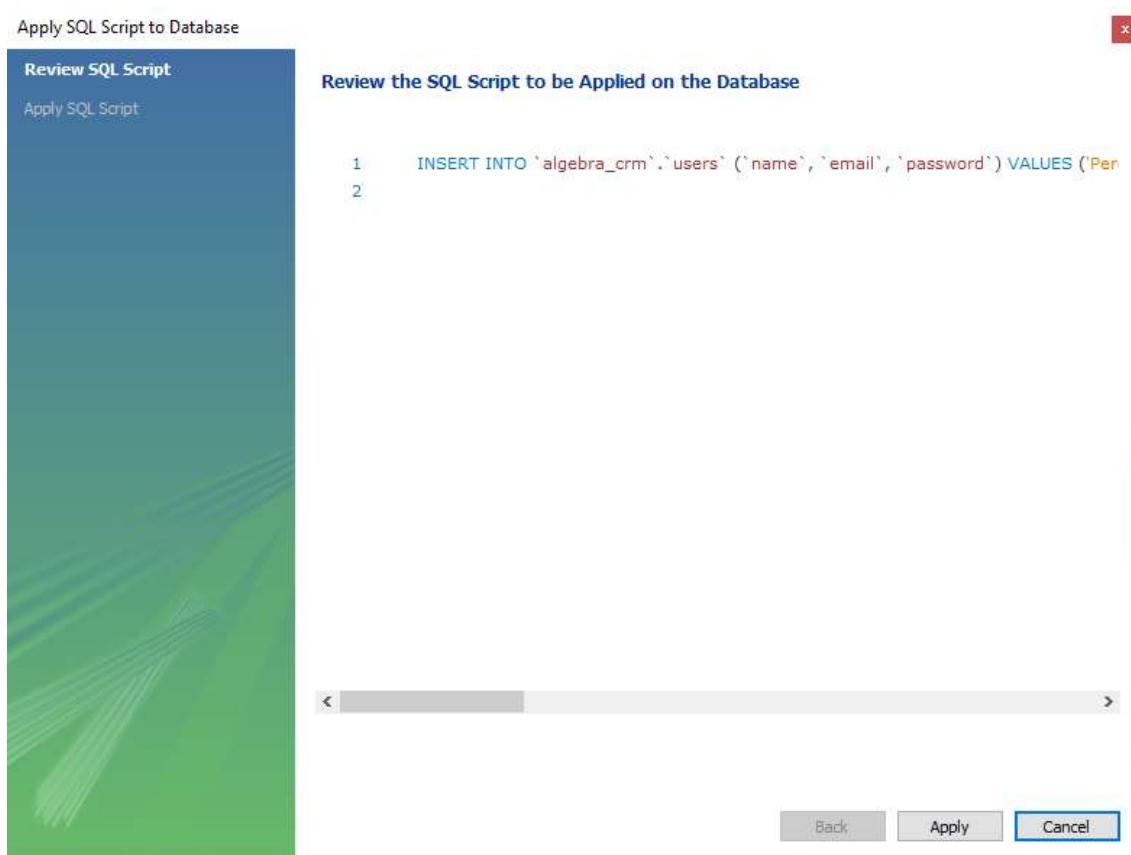
Za ispis rezultata koristimo red s `var_dump($users)` gdje ispisujemo sadržaj varijable `$users`, koja će prikazati strukturu i vrijednosti niza korisnika. Zato smo i dodali ova tri reda da bi smo mogli eventualno raditi debuggiranje i provjeriti da li sve u redu..

Za prikazivanje podataka pozivamo metodu `render()` (naslijедenu iz `Controller` klase) kako bismo prikazali određeni pogled (view). U ovom slučaju, prikazuje se home pogled i proslijeduje se asocijativna matrica s naslovom Home Page. Taj red bio je i prije.

Trebamo i ažurirati bazu podataka. Hardkodirat ćemo podatke da imamo podatke u bazi:

	id	name	email	password
1	NULL	Pero	pero@pero.com	12345678
2	NULL	NULL	NULL	NULL

Obavezno treba napraviti Applay jer neće zabilježiti promjenu u bazu:



Pogledajmo da li je povukao podatke:

The screenshot shows a browser window with the URL 'localhost:8081/Algebra/NapredniPHP/AlgebraCRM/'. The page content displays the results of a database query:

```
C:\xampp\htdocs\Algebra\NapredniPHP\AlgebraCRM\src\Controllers\HomeController.php:11:
array (size=1)
  0 =>
    array (size=4)
      'id' => int 1
      'name' => string 'Pero' (Length=4)
      'email' => string 'pero@pero.com' (Length=13)
      'password' => string '12345678' (Length=8)
```

Home Page

Vidimo da model radi uspješno i da je dohvatio podatke.

Instancirali smo User, dobili smo objekt User. U tom objektu svojstvo `$table` je postavljeno na `'users'`. Pošto smo klasu model naslijedili a klasa `Model` ima u sebi metodu `all()`. Kada smo pozvali `all()`, ona je pozvala `$this->db`, a `db` svojstvo u sebi sadrži `Database::getInstance()` jer je definirano u `__construct()`, s kojim možemo otići na bazu. Konstruktor će se aktivirati kada se izvrši iz metode `index()` red `$model = new \App\Models\User()`. Kako radimo new a klasa `User` nema svoj konstruktor i shodno tome koristimo konstruktor roditelja iz klase `Model`. Tamo u klasi Model dobivamo pristup Database servisu sa `$this->db = Database::getInstance()`. Nakon toga, vraćamo se u idući red u `index()` koji je u `HomeController.php`. Tu imamo pristup metodi `all()` koja je ispod haube pozvala Database servis i postavila `table($this->table)->get()`. Database je odradio svoje. U metodi `table`, `$this->table` postavlja svojstvo `$table` i vraća nazad objekt, a `get()` kaže „`SELECT $this->columns FROM $this->table`“. kako nemamo parametara i `where`, radi `fetchAll()`.

Sada ćemo to moći raditi sve to preko Modela tj. preko `index()`. i preko njega možemo lako doći do podataka, napraviti update ili obrisati.

Kako je ovo bio samo primjer, ubaćena 3 reda ćemo obrisati. Služila su da provjerimo da li model radi.

Kreiranje sistema autentifikacije

Ne smijemo dozvoliti da je password u plain tekstu, on mora biti hashiran.

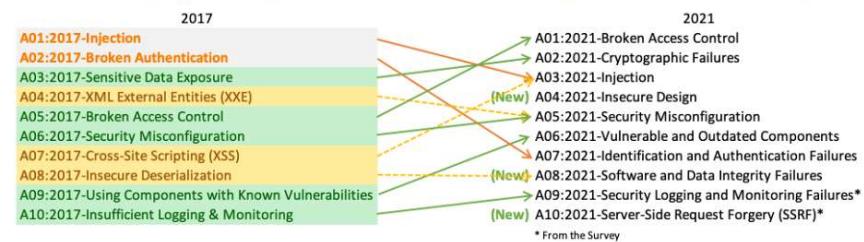
Želimo omogućiti da se korisnik može rezistrirati i prijaviti. Kada govorimo o autentifikaciji govorimo o username i password-u. Ako govorimo o dvostrukoj autentikaciji (dvofaktorska autentifikacija) - moramo imati nešto što znamo i što posjedujemo. Ono što znamo je username i password. Obično je to što posjedujemo mobitel. Autentifikatori obično traže autentifikacijski kod koji obično daju aplikacije na mobitelima. Taj kod je vremenski ograničen (obično 30 sekundi). Ovo nećemo implementirati na ovim predavanjima.



Tvrtke bi trebale usvojiti ovaj dokument i započeti proces osiguravanja da njihove web aplikacije minimiziraju ove rizike. Korištenje OWASP Top 10 možda je najučinkovitiji prvi korak prema promjeni kulture razvoja softvera unutar vaše organizacije u onu koja proizvodi sigurniji kod.

10 najvećih sigurnosnih rizika web aplikacije

Postoje tri nove kategorije, četiri kategorije s promjenama u nazivima i opsegu te neke konsolidacije u Top 10 za 2021.



- **A01:2021-Broken Access Control** pomică se s pete pozicije; 94% aplikacija testirano je na neki oblik neispravne kontrole pristupa. 34 uobičajena popisa slabosti (CWE) preslikana na Broken Access Control imala su više pojavljivanja u aplikacijama od bilo koje druge kategorije.

Registracija korisnika

U Laravelu validator ima 50-ak kombinacija koje se mogu koristiti. Za sada ćemo to preskočiti. Validiranje je bitno iako frontend može validirati podatke. Bez obzira na to potrebno je na backendu raditi validaciju. Validacija i sanitizacija su različite. Sanitizaciju trebamo napraviti nakon zahtjeva jer ti podaci ulaze u bazu. Sanitizacijom ne bi smjeli dopustiti da prođe maliciozni kod u bazu. To je obično JavaScript kod čiji ulazak trebamo spriječiti. Ako to priđe, korisnik će učitati te podatke, što ne želimo.

Želimo imati nekakav pojednostavljeni mehanizam koji će omogućiti da dođemo do podataka u zahtjevu. Možemo koristiti post ili get metodu i poslati podatke kroz URL. Pitanje je kako to imati na jednom dijelu i jednostavno skupiti. Laravel to radi jednostavno (u smislu logike), tako da kada se dogodi request, prije nego što dođemo na kontroler, popuni request sa hrpsom informacija. Praktično se u tom request objektu nalaze svi podaci koje smo koristili u našem primjeru (npr. server protokol, superglobalne varijable, podaci koji su pristigli pomoću URL-a, podaci koji su pristigli form data kanalima). Laravel popuni request objekt sa podacima i onda ih injektira u metodu koja se poziva na kontroleru i tamo imamo sve podatke.

Ovo je pojednostavljeno. Kako svaki kontroler u nekom trenutku treba doći do podataka iz requesta nije rađen poseban objekt i negdje punjen prije nego što dođe u kontroler. U apstraktnom **Controller**-u je napravljena metoda `getrequestData`:

```
<?php

namespace App\Controllers;

abstract class Controller
{
    protected function render(string $view, array $data = []): void
    {
        // ...
    }
}
```

```

        try {
            extract($data);
            require_once APP_ROOT . "/views/{$view}.php";
        } catch (\Throwable $e) {
            header($_SERVER['SERVER_PROTOCOL'] . ' 400 Bad Request');
            echo "View not found.";
        }
    }

protected function getRequestData(?array $fields = null): array
{
    $data = [];

    if ($fields) {
        foreach ($fields as $field) {
            $data[$field] = filter_input(INPUT_POST, $field,
FILTER_SANITIZE_SPECIAL_CHARS);
        }
    } else {
        foreach ($_POST as $key => $value) {
            $data[$key] = filter_input(INPUT_POST, $key,
FILTER_SANITIZE_SPECIAL_CHARS);
        }
    }

    return $data;
}

protected function redirect(string $path): void
{
    header("Location: " . APP_URL . "{$path}");
    exit;
}
}

```

Metoda `getRequestData` na početku provjerava da li polje `$fields` proslijeđeno metodi. Ako su specificirana polja, prolazi kroz svaki element u matrici `$fields`.

Nećemo gledati da li je zahtjev došao putem `POST` ili `GET` metode, bitna nam je samo superglobalna varijabla. Ako su podaci pristigli kroz POST metodu onda su u superglobalnoj `$_POST` i `$_GET`. Superglobalna varijabla `$_REQUEST` u PHP-u je asocijativna matrica koja sadrži podatke iz `$_GET`, `$_POST` (i `$_COOKIE` koji nas ovom prilikom ne zanima) superglobalnih varijabli. Ovo znači da se `$_REQUEST` koristi za prikupljanje podataka poslanih putem `HTTP GET` i `POST` metoda. Pitanje je želimo li hvatati jedne i druge podatke ili ne. Npr. možemo još nešto poslati kroz URL ako želimo.

Želimo Strana 452 od 517

omogućiti da u metodi `getrequestData` imamo matricu `$fields` tako da možemo poslati koja polja želimo. Možemo predati polja koja želimo a ne sva. U drugom koraku želimo sanitizirati input za svako polje. Dakle radimo s `filter_input` ugrađenom funkcijom. Ova funkcija može filtrirati različite vrste podataka - `INPUT_GET`, `INPUT_POST`, `INPUT_COOKIE`, `INPUT_SERVER` i `INPUT_ENV`. Nama je dovoljno `FILTER_SANITIZE_SPECIAL_CHARS` jer pretvara opasne znakove u specijalne znakove. Ti specijani znakovi prohranjuju se kao stringovi. Npr. napadač može pokušati pohraniti cookie. Najgore što se ovde može dogoditi je krađa identiteta. Moguće je npr. da ukrade kopiju osobne. Trebamo onemogućiti `<script></script>` tagove kako napadač ne bi mogao izvršiti JavaScript kod. Ako browser dobije `<script>`; to će tretirati kao običan tekst a ne kao tag za skriptu. Potencijalni tekst kod unosa koji može biti problematičan ovako će biti onesposobljen tj. sanitiziran.

Ako polja nisu specificirana, ide iteracija kroz sve POST podatke. Ovdje se filtriraju, dohvaćaju i sanitiziraju svi podaci iz `$_POST` superglobalne varijable. Kada pogledamo `$field`, to će biti matrica polja (tj. asocijativna matrica) s dohvaćenim i sanitiziranim podacima provućena kroz `foreach` petlju. Kada završimo posao, vraćamo `$data` nazad. Dakle, nikada ne smijemo podatke uzeti iz zahtjeva (engl. request) i gurnuti ih u bazu. **Ovime imamo SQL Injection zaštitu.** To ne sprječava Cross-side scripting tj. da neko gurne Java skriptu i mi je spremimo. Sa redom:

```
$data[$key] = filter_input(INPUT_POST, $key, FILTER_SANITIZE_SPECIAL_CHARS);
```

rješavamo to djelomično. Moguće je još da nam se desi Session Hijacking gdje netko gurne JavaScript kod a mi smo logirani u neku aplikaciju. Imamo otvorenu sesiju s kojom se predstavljamo (autoriziramo) serveru. Skripta ukrade sesijski token, pokupi ga sa Java skriptom i pošalje sebi. Taj sesijski token stavi u zahtjev (engl. request) i server ga pusti jer sesija vrijedi. Neko se autorizira dok smo mi još u sesiji. Tada na primjer može promijeniti username i password i ode račun.

Ovima ubijamo sve moguće probleme koji dolaze iz POST-a.

Ubacit ćemo ovdje i `render` i `redirect` metode, koju do sada nismo imali.

Recimo o njima koju rječ pa ćemo se vratiti na novi kontroler koji trebamo napraviti.

Metoda `render` je odgovorna za prikazivanje Pogleda (engl. View) unutar MVC arhitekture. `extract($data)`: Ova funkcija uzima elemente matrice `$data` i kreira varijable s imenima koja odgovaraju ključevima u matrici. Na primjer, ako `$data sadrži ['title' => 'Home Page']`, funkcija `extract` će kreirati varijablu `$title` s vrijednošću `'Home Page'`.

```
require_once APP_ROOT . "/views/{$view}.php";
```

Uključuje odgovarajući `view` file na temelju naziva view-a proslijedenog kao argument `$view`. Ako je `$view 'home'`, uključit će se datoteka `APP_ROOT . "/views/home.php"`.

```
catch (\Throwable $e):
```

Hvata bilo koju iznimku ili grešku koja se može dogoditi prilikom uključivanja view file-a.

- `header($_SERVER['SERVER_PROTOCOL'] . ' 400 Bad Request')`: Postavlja HTTP statusni kod na 400 (Bad Request).
- `echo "View not found."`: Prikazuje poruku "View not found."

Metoda `redirect` je odgovorna za preusmjeravanje korisnika na određeni URL.

- `header("Location: " . APP_URL . "{$path}")`: Koristi PHP funkciju header za postavljanje HTTP zaglavlja `Location`, koje preusmjerava korisnika na zadani URL. Varijabla `APP_URL` je definirana kao baza URL-a aplikacije, a `$path` je relativna putanja unutar aplikacije.
Na primjer, ako je `APP_URL http://localhost/myapp` i `$path '/dashboard'`, zaglavlj će biti postavljeno na `Location: http://localhost/myapp/dashboard`.
- `exit`: Prekida izvršavanje skripte odmah nakon slanja zaglavlja preusmjeravanja kako bi se osiguralo da se preusmjeravanje izvrši odmah i da se daljnji kod ne izvršava.

Napravit ćemo novi kontroler jer nećemo gurati sve u `HomeController.php`. Napravit ćemo novi kontroler koji će biti za autentifikaciju i nazvat ćemo ga `AuthController.php`. Stavit ćemo ga s ostalim kontrolerima u direktorij `Controller`. Doći ćemo do svih podataka iz svih metoda iz baznog kontrolera `Controller` i `render` i nove `getrequestData` tako da ćemo u `AuthController.php` definirati klasu `AuthController` kao `class AuthController extends Controller`.

Prvo želimo korisnika registrirati, što znači da nam treba Model. U konstruktoru ćemo to instancirati. Poanta ubrizgavanja (engl. injection) je da pošljete već u konstruktor objekt a ne da ga morate stvarati.

To bi glasilo ovako:

```
<?php

namespace App\Controllers;

use App\Models\User;

class AuthController extends Controller
{
    private User $userModel;

    public function __construct(User $userModel)
    {
        $this->userModel = $userModel;
    }
}
```

`$userModel` se instancira u konstruktoru. Poanta je da se to odradi u konstruktoru a ne kao ovdje da se šalje.

To se zove **injekcijska ovisnost** (engl. **Dependency Injection - DI**). Naš `AuthController` ovisi o `$userModel`-u i mi ga injektiramo ovdje. Nad kontrolerom koji se negdje instancira, gdje mi pozivamo konstruktor, a to je u našem Router-u, trebali bi razriješiti tu priču tako da sustav radi. Mi nećemo ovdje raditi cijeli sustav za inject-anje. U `Router.php` imamo `new $controller()` ali ne znamo koji. Morali bi imati međukorak da znamo što injektirati u kontroler u metodi `dispatch()`.

Zato mi ovdje nećemo imati injekcijsku ovisnost (engl. dependence injection), nego ćemo nad konstruktorom umjesto da smo inject-ali neki objekt, mi ćemo ga stvoriti. Više o [dependency injection moguće je pročitati u odvojenom poglavljju](#).

Napravit ćemo metodu `showRegister()` koja će pozvati metodu `render`:

```
<?php

namespace App\Controllers;

use App\Models\User;

class AuthController extends Controller
{
    private User $userModel;

    public function __construct()
    {
        $this->userModel = new User();
    }
}
```

To znači da želimo iscrtati neki Pogled. Moramo iscrtati neki obrazac.

```
public function showRegister()
{
    $this->render('auth/register', ['title' => 'Register']);
}
```

Nismo podesili rute pa trebamo i njih srediti. Trebamo znati koji zahtjev će doći do `showRegister()` u `AuthController`. Idemo u `web.php` u direktoriju `routes`. Nismo podesili rute, trebamo podesiti rutu, dati URI, dati kontroler i dati metodu na tom kontroleru. Dodat ćemo red među ostale rute:

```
Route::get('/register', AuthController::class, 'showRegister');
```

Sada možemo pozvati tu metodu `showRegister`. Ona je u `AuthController.php` u klasi `AuthController`. Možemo je vidjeti na prošloj stranici.

Sada kada imamo rutu koja će moći doći do naše metode, trebamo Pogled. U tom Pogledu trebamo imati formu (obrazac).

Kako u `showRegister()` metodi pozivamo iz `auth` poddirektorija `register`, idemo to napraviti. Dakle u `views` poddirektoriju otvaramo `auth` poddirektorij i u njemu `register.php`:

```
<h1>
    <?php echo $title; ?>
</h1>

<form action="register" method="post">
    <div>
        <label for="name">Name</label>
        <input type="text" name="name" id="name">
    </div>
    <div>
```

```

<label for="email">Email</label>
<input type="email" name="email" id="email">
</div>
<div>
    <label for="password">Password</label>
    <input type="password" name="password" id="password">
</div>
<button type="submit">Register</button>
</form>

<a href="login">Login</a>

```

Ovaj kôd predstavlja jednostavnu stranicu za registraciju korisnika s navigacijom na stranicu za prijavu.

Tu je dakle forma `register` ali metoda je `post`. Ako pogledamo rute u `web.php` vidjet ćemo da get zove metodu register ali to nije ta metoda jer request pristiže metodom `post`. Bitno je da će on namapirati na imena `name`, `email` i `password`. To su ujedno i polja u bazi, tako da ne moramo raditi posebne mapere. Ključevi u bazi su isti kao imena polja.

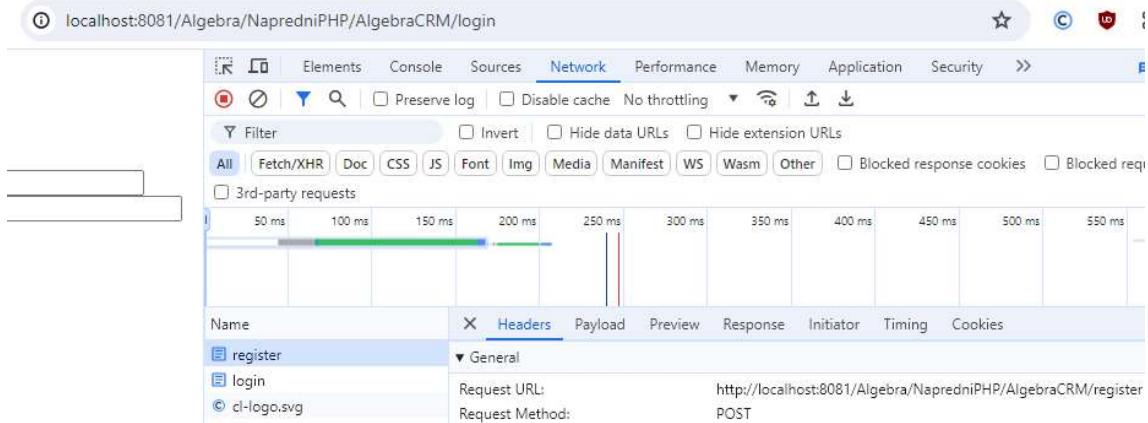
Ovaj kôd predstavlja jednostavnu registracijsku formu koja prikuplja ime, email i lozinku korisnika i šalje ih na URL `register` pomoću POST metode. Pored toga, uključuje link za preusmjeravanje korisnika na stranicu za prijavu. PHP dio u ovom kôdu koristi se za dinamički prikaz naslova stranice.

Pogledajmo kako ovo radi:

The screenshot shows a web browser window with the following details:

- Address Bar:** localhost:8081/Algebra/NapredniPHP/AlgebraCRM/register
- Title Bar:** Register
- Form Fields:**
 - Name: [Empty input field]
 - Email: [Empty input field]
 - Password: [Empty input field]
- Buttons:**
 - A large red "Register" button.

Trenutno nemamo validator za Register dugme. Ako ga pritisnemo dojavit će Not Found kao da path nije dobar. To je zato jer kada pritisnemo dugme Login, Request Method je POST. Metoda se promjenila i nema ništa pod `POST`, imamo samo pod `GET`.



Poanta je da nas vrati na Routh. `action` atribut u `register.php` mora biti bez forward slash-a, tj. `/`.

Ovo znači da trebamo dodati novu rutu za `POST` u `web.php` u `routes`.

```
Route::post('/register', AuthController::class, 'register');
```

Vidimo da imamo isti URI a to je zbog različitih metoda kojima pristigne request.

Vratimo se u `AuthController.php` i napišimo metodu `register` koju smo predviđeli u rutama.

Prvi korak u `register()` je da dohvativamo podatke `$data = $this->getrequestData()`. Ovo očekuje da pošaljemo barem jedan akument. Ako pogledamo logiku `getrequestData` vidimo da petlja očekuje podatke. Opet očekujemo da kada ne imenujemo podatke, da dobijemo sve. Ako napišemo:

```
$data = $this->getrequestData['name', 'email', 'password']);
```

Dobit ćemo te podatke.

Nismo osigurali da ne možemo poslati ništa tj. da nema argumenata. `getrequestData()` očekuje barem jedan podatak u `array`.

```
public function register():
{
    $data = $this->getrequestData();
    echo '<pre>':
    var_dump($data);
}
```

Idmo sada u `Controller.php` izmeniti metodu `getrequestData` tako da matrica može biti `nullable`, tj. ispred tipa govori da je tip `nullable`, tj. da može primiti vrijednost `null` a da default vrijednost bude `null`.

```
protected function getrequestData(?array $fields = null): array
```

Ako mi ne posaljemo ništa sada će `$fields` biti `null`. Napravićemo da samo ako imamo `$fields` prolazimo kroz petlju.

```
protected function getrequestData(?array $fields = null): array
```

```

{
    $data = [];

    if ($fields) {
        foreach ($fields as $field) {
            $data[$field] = filter_input(INPUT_POST, $field,
FILTER_SANITIZE_SPECIAL_CHARS);
        }
    } else {
        foreach ($_POST as $key => $value) {
            $data[$key] = filter_input(INPUT_POST, $key,
FILTER_SANITIZE_SPECIAL_CHARS);
        }
    }

    return $data;
}

```

Sada ne moramo provjeravati da li je `$fields` matrica.

Ako sada pošaljemo `$data = $this->getRequestData()` bez podataka moramo dodati neki else koji kada nemamo polja možemo izvlačiti po superglobalnoj varijabli `$_POST`. Vrijednosti nam ovdje ne trebaju.

```

protected function getRequestData(?array $fields = null): array
{
    $data = [];
    if ($fields) {
        foreach ($fields as $field) {
            $data[$field] = filter_input(INPUT_POST, $field, FILTER_SANITIZE_SPECIAL_CHARS);
        }
    } else {
        foreach ($_POST as $key => $value) {
            $data[$key] = filter_input(INPUT_POST, $key, FILTER_SANITIZE_SPECIAL_CHARS);
        }
    }

    return $data;
}

```

Kod registracije korisnika sanitizirati password nije baš dobra ideja. Provjerili smo sanitizaciju i `FILTER_SANITIZE_SPECIAL_CHARS` nije sanitizirao podatke. To znači da `getRequestData` moramo još jednom primijeniti.

```

protected function getRequestData(?array $fields = null): array
{
    $data = [];

```

```

if ($fields) {
    foreach ($fields as $field) {
        $data[$field] = htmlspecialchars($_POST[$field]);
    }
} else {
    foreach ($_POST as $key => $value) {
        $data[$key] = htmlspecialchars($value);
    }
}

return $data;
}

```

Kako ovo i dalje ne radi moguće je da je greška u var_damp koji nam prikazuje podatke. Jedino rješenje za provjeru je da u Name napišemo u Name <script>alert('HAHAHA')</script>. Kako se ovo nije pojavilo znači da ipak radi i ova i prethodna verzija.

Dakle to su načini kako spriječiti Cross-side scripting. Što se tiče passworda, možemo ga sanitizirati.

Vratimo se malo iza i pogledajmo što se desilo od pozivanja forme na `/register`. Imamo za unijeti Name, Email i Password. Kada korisnik pritisne dugme Register, podaci će se ako gledamo našu formu, proslijediti metodom `POST` u `register()` URI a Router će to obraditi (predavač kaže odhendlati) tako da imamo metodu `post` koja će preusmjeriti `AuthController` i metodu `register`. Tamo prvo što radimo skupljamo podatke iz `getrequestData` i sanitiziramo ih

Sada ćemo u `register()` metodi koja je pozvala `getrequestData()` dobiti u `$data` sanitizirane podatke. Metoda `getrequestData` se nalazi u `AuthController` nad apstraktnom klasom `Controller` koja je dostupna u svakom kontroleru kada iz request-a želimo doći do podataka koji će biti odmah i sanitizirani.

Sada ih trebamo poslati Modelu preko `Database` servisa.

```

public function register(): mixed
{
    try {
        $data = $this->getrequestData();
        $this->userModel->create($data);

        return $this->redirect('/login');

    } catch (\Exception $e) {
        echo 'An error occurred. Please try again.';
    }

    return $this->redirect('/register');
}

```

Red `$this->userModel->create($data)` poziva metodu `create` na instanci `User` modela (`$userModel`), dakle kreira user-a. Podatke koje smo prikupili pomoću Modela šaljemo na bazu. Metoda `create` će uzeti podatke iz matrice `$data` (koji uključuju `username`, `email` i `password`), hashirati lozinku i zatim ubaciti korisnika u bazu podataka.

Kontroler kontrolira zahtjev, nema nekakvu poslovnu logiku. Prikupi podatke i šalje Modelu. Kada Model završi s pohranom ili vratimo odgovor korisniku ili redirektamo.

Ako je korisnik uspješno kreiran, korisnik će biti preusmjeren na `/login` stranicu. Ako se dogodi neka greška (izuzetak) tokom procesa registracije, uhvatice se `Exception` i ispisat će se poruka `"An error occurred. Please try again."`. Poruka je namjerno generička jer korisnik ne smije dobiti poruku što to ne valja. Razlog za to je sigurnost. Kako greške govore nama o kodu, tako govore i napadaču. Ako je došlo do greške ili izuzetka, korisnik će biti preusmjeren nazad na `/register` stranicu.

Metoda `register` je dizajnirana da jednostavno i efikasno obradi registraciju novog korisnika, unese njegove podatke u bazu i preusmjeri ga na odgovarajuću stranicu, bilo da je registracija uspješna ili nije. Sve dok je `User` model pravilno postavljen i postoji ispravna veza s bazom podataka, proces bi trebao funkcionirati bez problema.

`userModel` nalazi se u direktoriju `Models` u `User.php`:

```
<?php

namespace App\Models;

class User extends Model
{
    protected string $table = 'users';

    public function create(array $data): bool|int
    {
        $data['password'] = password_hash($data['password'], PASSWORD_BCRYPT);

        return
            $this->db
                ->table($this->table)
                ->insert($data);
    }

    public function findByEmail(string $email): ?array
    {
        return
            $this->db
                ->table($this->table)
                ->where('email', $email)
                ->first();
    }
}
```

```
}
```

`User` klasa nasljeđuje `Model` klasu. Varijabla `$table` je zaštićena (protected) i sadrži naziv tablice u bazi podataka koju ova klasa predstavlja, u ovom slučaju `users`.

Ova metoda prihvata matricu podataka `$data` kao argument.

Hashirali smo lozinku korištenjem `password_hash` funkcije sa `PASSWORD_BCRYPT` algoritmom:

```
$data['password'] = password_hash($data['password'], PASSWORD_BCRYPT);
```

Prije umetanja, lozinka korisnika se hešira koristeći `password_hash` funkciju sa `PASSWORD_BCRYPT` algoritmom.

Vraćamo `return $this->db->table($this->table)->insert($data)`. Iznad modela postoji `$this->db` jer smo ga instancirali, gdje pozivamo `table`, šaljemo tablicu i pozivamo metodu `insert`. Metoda `insert` se nalazi u `Database.php`. Ona će pokušati te podatke staviti u bazu.

Metoda `findByEmail` prihvata email kao argument. Pretražuje bazu podataka u tabelici `users` koristeći `where` metodu za filter. Nakon toga vraća prvi rezultat pretrage koristeći `first` metodu.

Ako se pronađe korisnik, vraća se kao matrica. Ako se ne pronađe, vraća `null`.

`User` klasa nasljeđuje `Model` klasu koja pruža osnovnu funkcionalnost za rad s bazom podataka.

`create` metoda ubacuje nove korisnike u bazu, hashira lozinku prije ubacivanja.

`findByEmail` metoda pretražuje korisnika po emailu i vraća rezultat kao matricu ili `null` ako korisnik nije pronađen.

Pogledajmo tu metodu `insert` u `Database.php`. Želimo da nam podaci koje ubacujemo budu u ovom obliku:

```
insert into users (name, email, password) values (?, ?, ?)
```

```
public function insert(array $data): bool|int
{
    $this->columns = implode(", ", array_keys($data));
    $placeholders = implode(", ", array_fill(0, count($data), "?"));
    $sql = "INSERT INTO $this->table ($this->columns) VALUES ($placeholders)";

    $stmt = $this->conn->prepare($sql);
    $this->reset();

    if ($stmt->execute(array_values($data))) {
        return $this->conn->lastInsertId();
    }

    return false;
}
```

`array_keys($data)` vraća sve ključeve matrice `$data` (npr. `[name, email, password]`). Dakle ključevi postaju vrijednosti.

`implode(", ", array_keys($data))` stvara string od stupaca razdvojenih zarezima (npr. `name, email, password`).

`array_fill(0, count($data), "?")` stvara matricu placeholdera za pripremljenu izjavu (npr. `['?', '?', '?']`).

`implode(", ", array_fill(0, count($data), "?"))` stvara string placeholdera razdvojenih zarezima (npr. `?, ?, ?`). Ovo će generirati samo upitnike, na temelju podataka iz `$data`.

```
$sql = "INSERT INTO $this->table ($this->columns) VALUES ($placeholders);"
```

Taj table dobijamo iz `User` modela. To je u Database došlo jer je metoda `create` proslijedila. `insert` se nalazi u `Database.php`. U svojstvu `table` se nalazi informacija o tablici. `($this->columns)` je isto tako u `Database.php`, gdje generiramo neke stupce. To je svojstvo nad `Database`. Ti stupci su izvučeni sa `$this->columns`. tj s `implode(", ", array_keys($data))`. Ako pogledamo pristigli `$data` na početku `insert`, ključ u matrici ima ista imena i zove identično kao stupac u bazi. To je tako jer smo u formi `register` u `registrovati.php` već nazvali ulazna polja(engl. input fields) `name, email, password`. Ovo nam olakšava stvari jer praktično u ključevima imamo imena stupaca. Upit se formira tako da umetne vrijednosti u tablicu u stupce specificirane varijablom `$this->columns`. Ovako uvijek imamo ispravan SQL upit.

`prepare` metoda PDO objekta priprema SQL upit za izvršenje. `reset` metoda vraća vrijednosti `columns` i `where` na njihove podrazumijevane vrijednosti. Kada u `$this->columns` zapišemo vrijednost prvi puta u jednom pozivu Database servisa koji je singleton. To znači da ga koristimo svuda, uvijek isti objekt. To znači ako u jednom dijelu koda nešto napunimo u stupac, to smo napunili u objekt i pri drugom pozivu te vrijednosti su unutra. Zato je to bitno resetirati. `$columns` je po default-u `*`, tako da kada netko idući puta dođe, `$columns` mora biti na default-u. To je zato što metoda `select` ne mora poslati `$columns`, jer ni obavezno pozvati taj `select`. Zato je bitno da metoda `reset` resetira taj `$columns` i `$where`. Resetiramo svojstva koja imaju default vrijednost jer u singletonu imamo objekt koji se jedanput instancira. Kako nešto radimo vrijednosti u klasi se mijenjaju. Neko je pozvao `select` i nešto je promijenio. Kada dođe neko drugi vrijednosti `$columns` i `$where` nisu resetirane. Bile bi resetirane kada bi ponovo instancirao objekt. Kako se radi o singletone objektu to nije moguće ali smo dužni napraviti reset svaki puta kada završimo s nekom operacijom.

U uslovu `execute` metoda izvršava pripremljeni upit sa vrijednostima iz `$data`. `array_values($data)` vraća sve vrijednosti matrice `$data` u matricu i vežu za placeholdere prilikom izvršavanja. Jako je bitno da između `array_keys` i `array_values` ne promjenimo redoslijed polja u matrici jer neće raditi pretraga s placeholderima. Ako izvršenje upita uspije, vraća se ID posljednjeg unosa (`lastInsertId`). Ako upit ne uspije, vraća se `false`. `lastInsertId` živi nad PDO objektom i vraća ID koji je baza dodijelila. Pojnta je da možemo nešto raditi s tim ID:

Metoda `insert`: prima matricu podataka za umetanje, formira SQL upit koristeći stupce i placeholdere, priprema upit, izvršava upit sa vrijednostima iz matrice podataka. Ako umetanje uspije, vraća ID posljednjeg unosa; inače vraća `false`. Ovo omogućava jednostavno i sigurno umetanje podataka u bazu iz aplikacije.

Vratimo se na `userModel` nalazi se u direktoriju `Models` u `User.php`.

Ubacujemo podatke u bazu koristeći `insert` metodu. `table` metoda postavlja tablicu u koju se ubacuju podaci. `insert` metoda ubacuje podatke u tablicu i vraća ovisno o uspjehu ili `false` (ovisno o uspjehu) ili ID unosa. Mi možemo nad tim nešto raditi. Upitno je u kojem dijelu aplikacije želimo upravljati iznimkama. Da li je to Kontroler ili Model. Predavač kaže da je bolje u Kontroleru. Pogledajmo Database.php. Tamo može biti iznimaka čim radimo s PDO moguće da će baciti neki exception. U connectu već rukujemo sa PDOException i pravimo ispis. Ako time ne želimo rukovati u `Database.php` tj. da se ne uhvati u Database, možemo to prebaciti na `Controller`. To je dobra ideja ako želimo sve držati na istom mjestu. Možda ćemo u kontroleru AuthController samo uhvatiti iznimku i dati korisniku informaciju da je nešto krenulo po zlu. Možemo tamo gdje se iznimka dogodila umjesto da mu damo informaciju o tome, da to zapišemo u nekakav log. Ovdje ga nemamo ali to se može napraviti. Nakon toga možemo ponovo vratiti `throw $e`.

```
private function connect() {
    $dsn =
DB_DRIVER." :host=".DB_HOST.";dbname=".DB_NAME.";charset=".DB_CHARSET;
    $options = [
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
        PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
    ];

    try{
        $this->conn = new PDO($dsn, DB_USER, DB_PASS, $options);
    } catch (PDOException $e) {
        // Zapiši iznimku u log datoteku
        throw $e;
    }
}
```

Kada se dogodi iznimka cilj nam je u loger zapisati zašto se dogodila iznimka i proslijediti je dalje. Kako iznimka ide dalje, negdje na Database, gdje smo recimo pozvali `create($data)` na User modelu (klasa `User extends Model`). User model je pozvao `insert` na `Database`. `insert` se oslanja na cijeli objekt. Kontroler (`AuthController`) će isto tako dobiti tu iznimku.

Mi možemo kontrolirati iznimke na više različitih mjesta, što je dobra praksa. Moguće je i upravljati s jedne pozicije kao što je Kontroler.

Do `PDOException` iznimke može doći do i kod `execute`, možemo to staviti u try-cache blok i pokušati to iskontrolirati. Potencijalno možemo pustiti Exception do Kontrolera a u Kontroleru (`AuthController.php`) napraviti promjene kao što smo ih već napravili na početku Registracije korisnika.

Pogledajmo još metodu `findByEmail`, koja je također objašnjena ali ne i metoda `first` koja se nalazi u klasi `Database` (u poddirektoriju `Services` u `Database.php`):

```
public function first(): ?array {
    return $this->get()[0] ?? null;
}
```

Odmah iz reda `return` se poziva `get` metoda klase `Database` koja izvršava SQL upit i vraća sve rezultate u obliku matrice. [0] označava prvi element matrice rezultata. `?? null` koristi PHP operator za spajanje sa nultom vrijednošću (null coalescing operator) koji vraća `null` ako prvi element niza ne postoji.

Ako upit vrati rezultate, `first` metoda vraća prvi rezultat kao matrica. Ako upit ne vrati rezultate, `first` metoda vraća `null`.

Kriptiranje passworda

Ostalo je da još rješimo da password ne bude u plain tekstu, vidljiv svima.

Iz AuthController u metodi register napravimo `$data = $this->getRequestData` što prikupi podatke, šalje ih modelu kako bi ih insertao: `$this->userModel->create($data)` i Model prima informacije o potencijalnom novom korisniku (pogledati klasu `User` u `User.php`) i šalje ih pomoću servisa na bazu. Kao odgovor vraća `bool` ili `int`, što znači da je u AuthController u tom trenutku u redu `$this->userModel->create($data)` i \$id vraća ID novog korisnika u bazi. Vidimo da u `Database.php` kod insert ako je uspješno izvršen upit `($stmt->execute(array_values($data))` vraća `lastInsertId()`. Model vrati ono što vraća metoda `insert`. Onda Kontroler (AuthController tj metoda register) ima tu informaciju i sada s tim podatkom može kreirati novi zapis. Npr. imate tablicu koja ima relaciju na usera i treba joj ID usera.

Praksa kaže da napravimo hash. Hash algoritmi i kriptografski algoritmi su dvije potpuno različite stvari.

Hashiranje plain teksta (tekst u običnom formatu) je proces pretvaranja tog teksta u fiksnu dužinu niza koristeći hash funkciju. Hash funkcije su algoritmi koji uzimaju ulaz (i našem slučaju password) i vraćaju hash vrijednost ili `hash` koji je jedinstven za dani ulaz. Hashiranje je jednokratni proces: to znači da je vrlo teško (teoretski nemoguće) dobiti originalni tekst iz hash vrijednosti, što ga čini korisnim za pohranu osjetljivih podataka kao što su lozinke. To znači da nema dehash. Hashing algoritmi su MD5 (potpuno zastarjelo), SHA-1 (zastarjelo), SHA-256 (pitanje), SHA-512, bcrypt, itd. Moderni standardi preporučuju korištenje SHA-512 ili bcrypt za sigurnost. Hash algoritmi uvijek za isti set znakova daju uvijek isti hash. Zato postoji solt tj. sol i zasoljavanje, gdje dajete određeni slučajni set znakova prije passworda koji se skupa sa passwordom hashira. Tako da ako više korisnika ima isti password, zbog solta neće imati isti hash. Tim „soljenjem“ spriječavate kada neko ima jednostavan password da potencijalno izhashira i sa nekim crypterom i decrypterom otkrije password. Problem je naravno gdje je solt zapisan.

Password	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz	p4s5w3rdz
Salt	-	-	et52ed	ye5sf8
Hash	f4c31aa	f4c31aa	1vn49sa	z32i6t0

Pogledajmo kriptiranje. Kada kriptiramo podatke trebamo ih i dekriptirati. Postoje simetrične i asimetrične kriptografije tj. sistemi koji se oslanjaju na jedan ili na dva ključa.

Simetrična kriptografija koristi jedan ključ za kriptiranje i dekriptiranje podataka. Ovaj ključ mora biti tajan i poznat samo pošiljatelju i primatelju podataka. **Simetrični algoritmi su: AES (Advanced Encryption Standard, danas najsigurniji), DES (Data Encryption Standard), 3DES (Triple DES).**

Prednosti simetrične kriptografije su brzina jer simetrični algoritmi su brži od asimetričnih i jednostavnost jer koriste jednostavnije matematičke operacije. Nedostatak je sigurno dijeljenje ključeva između pošiljatelja i primatelja.

Asimetrična kriptografija koristi dva ključa: javni ključ za kriptiranje i privatni ključ za dekriptiranje. Javni ključ može biti slobodno distribuiran, dok privatni ključ mora ostati tajan. Asimetrični algoritmi su **RSA (Rivest-Shamir-Adleman, najčešće korišten)** i **ECC (Elliptic Curve Cryptography, raste popularnost)**. Prednosti asimetrične kriptografije su sigurna distribucija ključeva jer nema potrebe za sigurnim dijeljenjem ključa i digitalni potpisi jer omogućava autentifikaciju i integritet podataka. Nedostaci asimetrične kriptografije su brzina jer su asimetrični algoritmi su sporiji od simetričnih i složenosti.

U SHA-2 je implementiran pomoću hash funkcije. bcrypt je podržan sa funkcijom kao i argon2.

Predavač kaže da nema smisla raditi kriptiranje passworda, dovoljan je hash.

Predavač kaže da koga zanima početak kriptiranja neka pogleda Cezarovu šifru (engl. Ceaser Cipher) i Cezarova kutija (engl. Ceaser Box Cipher).

Vratimo se na PHP i hash-ranje password-a. Metoda koju PHP ima ugrađenu zove se `password_hash`. Podržani su bcrypt algoritam i Argon2. Praktično se možemo osloniti na default, dobit ćemo i solt i radit će decrypt.

U User.php u klasi `Model`, u `Create`, prije vraćanja dodali smo red:

```
$data['password'] = password_hash($data['password'], PASSWORD_BCRYPT);
```

Prije umetanja, lozinka korisnika se uzme iz `$data['password']` i hešira se koristeći `password_hash` funkciju sa `PASSWORD_BCRYPT` algoritmom. Solt je dodan pa ne moramo stavljati cost.

Unijet ćemo za probu 2 korisnika sa istim password-om ali ne i s istim imenom. U bazi vidimo:

	id	name	email	password
▶	1	test	test@test.com	\$2y\$10\$IEY1ivrjbp8EwouOPSiGhetjhFZ3J1r01L...
	2	test1	test1@test.com	\$2y\$10\$7pyNjsejaZPmyy1szAlibusMrtNsHaTbz...
*	NULL	NULL	NULL	NULL

Dakle heširani password zbog soli ne izgleda isto, skoro u potpunosti je različit. Kako ćemo sada prilikom logina provjeriti da li se podudara s onim što je unio korisnik. Napravit ćemo verifikaciju s `password_verify` kojem damo ono što je korisnik unio i usporedi s onim što je u bazi. Kako to radi ispod haube:

`$2y$` označava da je korišten `bcryt`. To je algoritam koji je korišten.

`10$` označava broj iteracija (cost faktor).

Iduća 22 znaka su salt.

Zadnjih 31 je hashirani rezultat lozinke.

Redirekcija registriranog korisnika

Nakon uspješne registracije korisnika, trebamo ga preusmjeriti na login stranicu. Prijavimo ga odmah u sustav i preusmjerimo na neku nadzornu ploču (engl. dashboard) odakle kreći prijavljeni korisnici. Prvo ćemo vidjeti na koji način korisnika negdje preusmjeriti. Idemo u `AuthController.php` u klasi `User` vidjeti metodu `register`. Nad kontrolerom ćemo kreirati metodu i reći: `$this->redirect('/login')`. Pogledajmo ima li smisla raditi ovo sa `return` ili ne. Ova metoda `register` se poziva negdje u donjem dijelu Router-a. Zamislimo da u donjem dijelu metode `register` dodamo `$this->redirect('/register')`. Evo te metode za registraciju korisnika:

```
public function register(): mixed
{
    try {
        $data = $this->getRequestData();
        $this->userModel->create($data);
        $this->redirect('/login');

    } catch (\Exception $e) {
        echo 'An error occurred. Please try again.';
    }

    $this->redirect('/register');
}
```

Donja linija koda `$this->redirect('/register')` izvršit će se nezavisno da li je bio uspješan `try` i da li je bio `catch`. Ako dođemo na `try` i on recimo pukne na redu `$this->userModel->create($data)`, on preskoči u `catch`, odradi grešku i nastavi do reda `$this->redirect('/register')`. Logika je ovdje da probamo registrirati korisnika, ako ga uspješno registriramo u bazi, korisnika redirektamo na `login` stranicu. Međutim, ako dođe do problema, uhvatimo s `catch Exception` i korisniku ispišemo grešku i redirektamo ga nazad na `register` stranicu. U stvarnoj produkciji ne bi smo ispisali s `echo` već bi poruku stavili u neki `session`, kako bi `session` živio kada dođemo na drugu stranicu i kako bi iz tog `session`-a isčupali poruku. Ovako kako je sada napisano, nakon poruke odmah ide `redirect` i korisnik možda uopće neće vidjeti poruku jer će ići brzo. Kako ćemo korisnika preusmjeriti na register, korisnik neće ni primijetiti što se desilo. Dakle trebamo drugačiji mehanizam da korisnik vidi što se stvarno desilo.

Može biti i `echo` ali poanta je da kod ispisa poruka to radimo na samome. Pogledu na početku tako da bitamo da li negdje u `session`-u postoji neka poruka i ako postoji da se ispiše korisniku. Mi onda kontroliramo taj podatak u `session`-u. Nismo još otvorili session pa ćemo o tome pričati. To se inače zovu **flash poruke** (engl. **flash message**). Laravel ima isti takav mehanizam.

Vratimo se na `try` blok koji ako se ne desi `redirect`, potencijalno ide `catch` i nastavlja se kod. Ako stavimo `return` ispred jednog i drugog `redirect`-a:

```
public function register(): mixed
{
    try {
        $data = $this->getRequestData();
        $this->userModel->create($data);
```

```

        return $this->redirect('/login');

    } catch (\Exception $e) {
        echo 'An error occurred. Please try again.';
    }

    return $this->redirect('/register');
}

```

Sada zaustavljamo djelovanje funkcije, dakle ako nije bilo greške ide na `login` a ako je greške bilo ide na `register`. `redirect` modificira zaglavlje requesta tj. zahtjeva. Postoje dva načina na koji radi `redirect`, i za to se koriste status kodovi.

Status kod	Značenje	Opis
301	Temporary Redirect	Ovaj kod statusa označava da je traženi resurs definitivno premješten na URL koji su dali zaglavljaju lokacije. Browser preusmjerava na novi URL, a pretraživači (engl. search engines) ažuriraju svoje veze na resurs.
302	Found	Ovaj kod statusa označava da je zatraženi resurs privremeno premješten na URL koji daje zaglavlje lokacije. Browser preusmjerava na ovu stranicu, ali pretraživači (engl. search engines) ne ažuriraju svoje veze na resurs (u 'SEO-govoru', kaže se da se 'link-juice' ne šalje na novi URL).

Tu metodu `redirect` ćemo raditi na baznom kontroleru `Controller`.

```

protected function redirect(string $path): void
{
    header("Location: " . APP_URL . "{$path}");
    exit;
}

```

Redirect koristi `header`, koji omogućava da zamijenimo određene stvari. Idući red je `exit`. `redirect` vraća `void`.

Probat ćemo upisati test2 korisnika na `register` strani.

localhost:8081/Algebra/NapredniPHP/AlgebraCRM/login

Login

Email
Password
 [Register](#)

Network tab in developer tools showing a POST request to 'register' took 300 ms.

Name	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
register	General			Request URL: http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/register Request Method: POST Status Code: 302 Found Remote Address: [::1]:8081			
login							
cl-logo.svg							

```
http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/register
POST
● 302 Found
[::1]:8081
strict-origin-when-cross-origin
```

Vidimo da se desio 302 Found.

Probat ćemo upisati isti mail na register stranici dva puta. Preusmjerenje ide toliko brzo da ništa nismo stigli vidjeti.

PDO Exeption se desio u Database u `insert` metodi.

Međutim da budemo sigurni možemo u catch dio staviti iza komentara echo '`An error occurred. Please try again.`' red sa `die($e->getMessage())` i tada ćemo biti sigurni da kada unesmo isti mail da program ispada van. Ali tada vidimo grešku.

```
Algebra CRM An error occurred. Please try again.SQLSTATE[23000]: Integrity
constraint violation: 1062 Duplicate entry 'test@test.com' for key
'idx_users_email'
```

Ovo odgovara korisniku jer tako može otkriti koji je mail već registriran. I zato greška ne smije biti tako dojavljena.

>Login korisnika

Trebamo dodati rutu u `web.php`. Jedna metoda ide metodom GET za prikaz forme za prijavu i druga koja ide metom POST koja ide za stvarnu provjeru i prijavu korisnika

```
Route::get('/login', AuthController::class, 'showLogin');
Route::post('/login', AuthController::class, 'login');
```

Dodat ćemo još dvije rute:

```
Route::get('/logout', AuthController::class, 'logout');
Route::get('/admin/dashboard', AdminController::class, 'dashboard');
```

Morat ćemo ili napraviti kontroler za dashboard ili ga zakomentirati.

U `AuthController` napisat ćemo `showLogin`:

```
public function showLogin()
{
    $this->render('auth/login', ['title' => 'Login']);
}
```

U `register.php` ispod forme dodat ćemo link za Login:

```
<a href="login">Login</a>
```

Iskopirat ćemo `register.php` i od njega napraviti `login.php`. Uklonit ćemo dio s `name` i promijenit ćemo `action` u `login`. Dugme neće biti na `Regiser` nego `Login` a link će biti `Register`:

```
<h1>
    <?php echo $title; ?>
</h1>

<form action="login" method="post">
    <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email">
    </div>
    <div>
        <label for="password">Password</label>
        <input type="password" name="password" id="password">
    </div>
    <button type="submit">Login</button>
</form>

<a href="register">Register</a>
```

Sada se možemo prebacivati sa Login i Switch između stranica. Imamo unesenog jednog korisnika sa password-om 123456789. Želimo još jedom vidjeti kuda će podaci biti poslani ako se probamo logirati. U formi je `action="login"`. U ruti vidimo da ako pristigne s metodom post jer je metoda u formi podešena na `method="post"`, šalje se na `AuthController` metodu `login`. Idemo napisati tu metodu u `AuthController`:

```
public function login(): mixed
{
    try {
        [$email, $password] = array_values(
            $this->getRequestData(['email', 'password'])
        );
    }
```

```

$user = $this->userModel->findByEmail($email);

if(!$user || !password_verify($password, $user['password'])) {
    return $this->redirect('/login');
}

$_SESSION['user'] = $user;

} catch (\Exception $e) {
    echo 'An error occurred. Please try again.';
}

return $this->redirect('/admin/dashboard');
}

```

Metodom `getrequestData` koristimo se za dohvaćanje podataka iz POST zahtjeva. Možemo dohvatiti sve podatke ali nas zanimaju `email` i `password`. `array_values` vraća numeričku indeksiranu matricu tj. listu, što omogućava jednostavno automatsko raspakiranje u `$email` i `$password` varijable. To je destrukcija matrice i zgodno je što to nismo morali raditi s petljama. Destrukciju matrice nismo mogli napraviti kao:

```
[$email, $password] = $this->getrequestData(['email', 'password'])
```

Prvi korak je pronaći `$user` iz baze podataka preko email-a. `findEmail` ako ne pronađe niti jednog korisnika dobit ćemo `null` a ako je pronašao dobit ćemo matricu.

```
$user = $this->userModel->findByEmail($email);
```

Napravili smo i provjeru:

```

if(!$user || !password_verify($password, $user['password'])) {
    return $this->redirect('/login');
}

```

Ovdje se provjerava da li je korisnik pronađen (`$user` nije `null`). `password_verify` provjerava da li je unešena lozinka (`$password`) ispravna u odnosu na hashiranu lozinku (`$user['password']`) iz baze podataka. Ako bilo koji uslov nije ispunjen (korisnik ne postoji ili je lozinka netočna), korisnik se preusmjerava natrag na stranicu za prijavu.

Ako je autentifikacija uspješna, korisnički podaci se pohranjuju u PHP sesiju. Ovo omogućava da aplikacija prepozna korisnika tokom sljedećih zahtjeva:

```
$_SESSION['user'] = $user;
```

Superglobalnu varijablu `$_SESSION` možemo koristiti kada napravimo session start, što nismo napravili. To znači da to moramo napraviti u `index.php` u prvom redu:

```
session_start();
```

Tek sada možemo pristupiti superglobalnoj varijabli `$_SESSION`.

Ako se dogodi neka iznimka unutar `try` bloka, ispisuje se poruka o grešci.

Ovo je jednostavna obrada greške koja informira korisnika da je došlo do problema.

Ostalo je još da napravimo preusmjeravanje korisnika na dashboard:

```
return $this->redirect('/admin/dashboard');
```

Ako je autentifikacija uspješna i nema iznimke, korisnik se preusmjerava na dashboard. Metoda `redirect` postavlja HTTP zaglavlje za preusmjeravanje i prekida izvršavanje skripte.

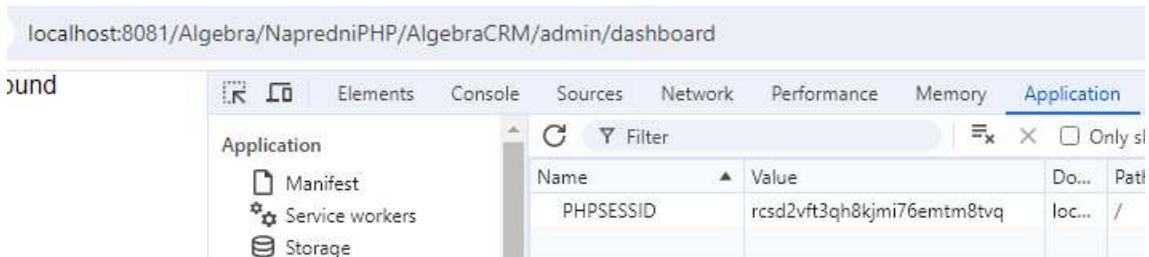
Metoda login u AuthController.php provodi osnovnu autentifikaciju korisnika provjeravajući unesene podatke s onima u bazi podataka. Ako su podaci ispravni, korisnik se prijavljuje i preusmjerava na `dashboard`; inače, preusmjerava se natrag na `login` stranicu za prijavu.

Odjavljivanje korisnika

Uspješno smo odradili autentifikaciju korisnika.

`session` živi skroz dok je otvoren browser ili dok korisnik ne prođe neki `logout` proces. To znači ako napravimo `die` da će sesija ostati otvorena.

Sada ne želimo u aplikaciji na više mjesta raditi autentifikaciju. Želimo znati da je korisnik prijavljen i pustiti ga. To ćemo moći upravo preko podatka u superglobalnoj varijabli `$_SESSION`. Ako unutra postoji neki podatak, tj. 'user', to znači da je user autoriziran od samog servera.



To je zato što je taj user dobio svoj svoj `PHPSESSID`. Ako ga ima to znači da je `$_SESSION['user']` dopuštena i unutra se nalaze neki podaci o korisniku. Mi nećemo raditi usporedbu da li korisnik ima neka prava ili ne (administrator ili slično). Želimo zaštiti dashboard tako da vidimo da li je gost (anonimni korisnik) ili prijavljeni korisnik. Korisnik koji nije prijavljen je gost. Superglobalna varijabla `$_SESSION` živi u memoriji možemo joj dati rok trajanja i pohraniti je na disk. Možemo koristiti i cookie. Mi ćemo ostati na superglobalnoj u memoriji.

Brute force koji se može dogoditi je da neko ode na formu i pogleda kod. Vidi da ide post metodom i vidi da se pohranjuju mail i name. Moguće je da napravi Java skriptu kod i gađa Email i Password. Ako zna Email gađat će Password i suprotno.

Za onemogućavanje brute force koristi se straddling mehanizam. Kroz određeni broj pokušaja blokira se pristup s cool down timer-om. Pauza se eksponencijalno podiže u slučaju ignoriranja. Postoji trashhold koji ne da da brojač bude na istom vremenu.

Kreiranje sistema autorizacije

Korisnik napravi zahtjev na `/admin/dashboard`. Ne smije biti ovdje ako nije autentificiran tj. prijavljen u sistem. Nećemo svaki puta tražiti autentifikaciju nego ćemo ga autorizirati pomoću session-a. Treba složiti mehanizam koji će vršiti provjeru na jednostavan način. `Middleware` će biti od Routra prema Kontroleru u konstruktor. Svaki Kontroler kada se instancira, aktivira se njegov konstruktor. U tom konstruktoru ćemo tražiti da se korisnik autorizira uz pomoć `Middleware`-a. Zahtjev (engl. request) će prije nego što završi na nekoj metodi, tipa `Login` ili register u `Auth`. Prvo ćemo u konstruktoru pitati da li je korisnik prijavljen ili gost. Ako je gost, nema tu što raditi, treba ga redirect-ati na login stranicu. Isto tako ćemo napraviti i na showLogin stranici jer nema što tu raditi ako je prijavljen. Možemo to raditi na nivou Kontrolera, na nivou metoda, ovisno o tome kako ćemo složiti sistem. Napravit ćemo to jednostavno kao `AuthMiddleware` s dvije metode s koje ćemo provjeravati status korisnika.

U `web.php` dodat ćemo dio koji će služiti za administriranje. Ako korisnik ili administrator nije prijavljen redirektat ćemo ga na logiranje. Ako je prijavljen, napravit ćemo redirect na dashboard.

Middleware, ili kako se nekad nazivaju filteri, služe za zaštitu kontrolera i metoda u njemu. U `src` direktorij dodat ćemo poddirektorij `Middleware` i nazvat ćemo datoteku `Auth.php`.

Napravit ćemo metodu `check()` kojom ćemo provjeriti da li je korisnik prijavljen. Ako je prijavljen želimo da ga redirektamo tako da bi trebali dodati else blok ali bolje je razdvojiti u dvije metode zbog single responsibility principa. Promijenit ćemo metodu `check()`:

```
public static function check(): void
{
    if (! isset($_SESSION[''])) {
        header('');
    }
}
```

Dakle napisat ćemo još `isAuthenticated()` metodu.

Idemo u `Controllers` poddirektoriju napisati `AdminController.php`.

Prvo ćemo aktivirati rutu u `routes` direktoriju u `web.php`:

```
Route::get('/admin/dashboard', AdminController::class, 'dashboard');
```

Ovoj ruti trebaju moći pristupiti svi registrirani korisnici. Ako korisnik nije registriran, redirektat ćemo ga na login stranicu. I suprotno ako je registrirani korisnik na login stranici redirektat ćemo ga na `/admin/dashboard`. Najispravnija praksa je korištenje Middleware-a ili filtera. Laravel je u ranim verzijama to također nazivao filterima, koji su preusmjeravali. Uz pomoć Middleware-a želimo zaštititi

cijeli Kontroler ili pojedinu metodu unutar kontrolera. To ćemo riješiti tako da u poddirektorij dodamo još jedan poddirektorij s nazivom `Middleware` i dodamo datoteku koju ćemo nazvati `Auth.php`.

Kao što smo već rekli, metode u ovoj klasi koristit ćemo da bi smo osigurali da samo ovlašteni korisnici mogu pristupiti određenim dijelovima aplikacije. Možda bi bilo moguće sve napraviti s jednom metodom. Tada bi `check()` metodi morali dodati `else` i u njemu staviti `header('Location: '. APP_URL . '/admin/dashboard')`. Ipak je bolje to razdvojiti u različite metode (zbog single responsibility principa).

Metoda `check()` provjerava da li je korisnik prijavljen ili nije. Ako korisnik nije prijavljen (nema `'user'`-a u sesiji), preusmjerava ga na stranicu za prijavu. Ako je korisnik prijavljen, preusmjerava ga na nadzornu ploču `/admin/dashboard`. Dakle kada god pozovemo `check()` ili ide na `/login` ili na `/admin/dashboard`. To je mogući potencijalni problem jer svaki puta imamo preusmjeravanje.

`isGuest()` metoda provjerava da li korisnik nije prijavljen (tj. ako je gost). Ako korisnik nije prijavljen, preusmjerava ga na stranicu za prijavu.

`isAuthenticated()` metoda provjerava da li je korisnik prijavljen. Ako je korisnik prijavljen, preusmjerava ga na nadzornu ploču `'/admin/dashboard'`.

Sve metode koriste superglobalnu varijablu `$_SESSION` za provjeru da li postoji korisnički podatak `'user'` koji znači da je korisnik prijavljen. Koristeći ugrađenu `header()` funkciju, metode preusmjeravaju korisnika na odgovarajuće stranice. Ovaj postupak završava s `exit` kako bi se osiguralo da ostatak koda ne bude izvršen nakon preusmjeravanja.

```
<?php

namespace App\Middleware;

class Auth
{
    public static function check(): void
    {
        if (! isset($_SESSION['user'])) {
            header('Location: '. APP_URL . '/login');
            exit;
        } else {
            header('Location: '. APP_URL . '/admin/dashboard');
        }
    }

    public static function isGuest(): void
    {
        if (! isset($_SESSION['user'])) {
            header('Location: '. APP_URL . '/login');
            exit;
        }
    }
}
```

```

    }

    public static function isAuthenticated(): void
    {
        if (!isset($_SESSION['user'])) {
            header('Location: ' . APP_URL . '/admin/dashboard');
            exit;
        }

    }
}

```

Metode se koriste za kontrolu pristupa razlicitim dijelovima aplikacije. Na primjer:

`check()`: Ova metoda je korisna za stranice koje zahtijevaju autentifikaciju.

`isGuest()`: Ova metoda je korisna na stranicama kao što je stranica za registraciju, gdje bi bilo neologično dopustiti pristup korisnicima koji su već prijavljeni.

`isAuthenticated()`: Ova metoda se koristi za zaštitu stranica na koje samo prijavljeni korisnici smiju imati pristup.

Ove metode pomažu u osiguravanju da samo ovlašteni korisnici imaju pristup određenim resursima i da gosti budu usmjereni prema stranici za prijavu.

U `Controllers` napraviti ćemo novu datoteku `AdminController.php`. Ova klasa koristi `Auth` middleware za autentifikaciju korisnika i pruža metodu za prikaz administratorskog panela i korisničkih informacija. Cilj nam je zaštiti kompletan Kontroler i sve metode koje su na tim Kontrolerima.

```

namespace App\Controllers;
use App\Middleware\Auth;

```

Prvi red specificira da se `AdminController` nalazi u `App\Controllers` namespace-u, što pomaže organizaciji koda i izbjegava konflikte s drugim klasama koje mogu imati isti naziv. Idući red koristi `Auth` klasu iz `App\Middleware` namespace-a, koja se koristi za autentifikaciju i autorizaciju.

Metoda `dashboard` prikazuje jednostavan HTML sadržaj. U ovom slučaju, prikazuje poruku `Admin Dashboard` i link za odjavu. Koristi se `echo` za ispis poruke i HTML elemenata. Link za odjavu vodi na `/logout`, što je ruta za izlaženje iz sesije i odjavu korisnika.

```

public function dashboard()
{
    echo 'Admin Dashboard', '  
' , '<a href=".../logout">Logout</a>';
}

```

Metoda `showUsers` također koristi echo za ispis poruke `Show Users`.

```
public function showUsers()
{
    echo 'Show Users';
}
```

Idemo u `routes` popraviti rute u `web.php`.

```
Route::get('/admin/dashboard', AdminController::class, 'dashboard');
Route::get('/admin/users', AdminController::class, 'showUsers');
```

Ovdje ćemo zaštititi cijeli kontroler, niti jedna ruta neće biti aktivna za goste.

Do sada smo mogli ukucati linkove

<http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/admin/dashboard> ili
<http://localhost:8081/Algebra/NapredniPHP/AlgebraCRM/admin/users> i otvoriti te stranice bez problema. To ćemo zaštiti upravo korištenjem Middleware-a koji je posrednik prije nego što se desi neko djelovanje (engl. action). Kada pričamo o Middleware-u, Middleware se koristi za filtriranje zahtjeva (engl. request). Sa Middleware je moguće filtrirati zahtjeve ka kontroleru. Mi to nismo radili, mi smo samo gledali da napravimo zaštitu tko može pristupati određenoj stranici.

Ako želimo zaštiti cijeli `AdminController`, koristit ćemo iz `Auth.php`, `isGuest` metodu. Želimo vidjeti da li se radi o gostu - jer je svakog gosta 5 sekundi dosta...

Vratimo se u `AdminController`. Metoda `__construct()` se automatski poziva kada se stvara nova instanca `AdminController` klase.

Metoda `Auth::isGuest()` provjerava da li je korisnik prijavljen. Ako nije prijavljen, preusmjerava ga se na stranicu za prijavu `/login`. Ako je korisnik već prijavljen, može nastaviti koristiti tekuće funkcionalnosti. Ova metoda osigurava da samo autentificirani korisnici mogu pristupiti stranicama.

```
public function __construct()
{
    Auth::isGuest();
}
```

Sada bi trebalo raditi ako probamoći sa `/admin/dashboard` ili sa `/admin/users`.

Evo cijelog koda `AdminController.php`:

```
<?php

namespace App\Controllers;

use App\Middleware\Auth;

class AdminController extends Controller
```

```
{
    public function __construct()
    {
        Auth::isGuest();
    }

    public function dashboard()
    {
        echo 'Admin Dashboard', '<br>', '<a href="../logout">Logout</a>';
    }

    public function showUsers()
    {
        echo 'Show Users';
    }
}
```

Ako se probamo logirati i onda opet otići na `/login` uspjjet ćemo, što nije dobro. Nema potrebe da prijavljen korisnik dođe na tu stranicu. To nije sigurnosni problem ali nije dobra praksa.

Otići ćemo na `AdminController.php` i u konstruktoru provjeriti da li je korisnik autentificiran.

```
public function __construct()
{
    Auth::isAuthenticated();
    $this->userModel = new User();
}
```

Kada korisnik proba pristupiti tom logout-u, bit će preusmjeren. Pokuša li korisnik na `/login` kao prije bit će preusmjeren na `/admin/dashboard`. Ako probamo `/register`, također će biti preusmjeren na `/admin/dashboard`. To radi kako treba. Međutim imamo problem. U AdminController u metodi `dashboard` napravili smo `Logout` jer nam treba.

Ako odemo na `Logout` on će aktivirati `AdminController`, i metodu `Logout` koju trenutno nemamo. Na `/admin/dashboard` vidimo link za `Logout`. Metoda `Logout` nevedena je u rutama. Ako kliknemo na `Logout` ostajemo i dalje na `/admin/dashboard` ne javlja grešku a trebao bi. Problem je u `AdminController.php` u konstruktoru, tj. u metodi `Auth::isAuthenticated()` koja se poziva. Kada kreiramo rutu koja se zapisala u matricu, kada napravimo `dispatch()`, kada `dispatch()` distancira `AdminController`, kod instanciranja se aktivira konstruktor koji provjeri da li je korisnik prijavljen. Ako je redirekta ga na `/admin/dashboard`. Dakle svaki puta kada kliknemo na link `Logout` vratimo se na `/admin/dashboard`. Praktički nikada action neće završiti na `Logout`. Ako isključimo metodu `Auth::isAuthenticated()` iz `AdminController.php` i pokrenemo `/admin/dashboard` i sada dojavljuje grešku jer i dalje nemamo metodu.

Moguće rješenje je da proširimo `isAuthenticated()`, da mu možemo poslati za provjeru određenih metoda, da recimo ne vršimo provjeru za određene metode (npr. za `Logout`). To znači da bi proširivali

Middleware i da bi poslao kompleksniji. Ili jednostavnije rješenje, koje je malo redundantno, da u svakoj metodi u `AuthController.php` stavimo `Auth::isAuthenticated()`.

```
<?php

namespace App\Controllers;

use App\Middleware\Auth;
use App\Models\User;

class AuthController extends Controller
{
    private User $userModel;

    public function __construct()
    {
        $this->userModel = new User();
    }

    public function showRegister()
    {
        Auth::isAuthenticated();
        $this->render('auth/register', ['title' => 'Register']);
    }

    public function register(): mixed
    {
        Auth::isAuthenticated();
        try {
            $data = $this->getRequestData();
            $this->userModel->create($data);

            return $this->redirect('/login');

        } catch (\Exception $e) {
            echo 'An error occurred. Please try again.';
        }

        return $this->redirect('/register');
    }

    public function showLogin()
    {
        Auth::isAuthenticated();
    }
}
```

```

        $this->render('auth/login', ['title' => 'Login']);

    }

    public function login(): mixed
    {
        Auth::isAuthenticated();
        try {
            [$email, $password] = array_values(
                $this->getRequestData(['email', 'password'])
            );

            $user = $this->userModel->findByEmail($email);

            if(!$user || !password_verify($password, $user['password'])) {
                return $this->redirect('/login');
            }

            $_SESSION['user'] = $user;

        } catch (\Exception $e) {
            echo 'An error occurred. Please try again.';
        }

        return $this->redirect('/admin/dashboard');
    }

    public function logout()
    {
        Auth::isGuest();
        session_destroy();
        return $this->redirect('/login');
    }
}

```

Ovdje smo napisali i `logout()` metodu. Čim se desio `session_destroy()` dešava se nova sesija. Nakon toga ide redirekcija na `/login`.

Trajanjem sesije moguće je upravljati tako da prije `session_start()` postavimo `session_set_cookie_params()`, ako je `3600` to znači da da svaki klijent treba zapamtititi da session id traje **točno** 1 sat. Ako želimo da server zadrži podatke o sesiji najmanje jedan sat koristit ćemo `ini_set('session.gc_maxlifetime', 3600)`.

Dependency Injection (DI) u PHP-u

Dependency Injection (DI) je dizajnerski obrazac u kojem objekt prima svoje ovisnosti od vanjskog izvora, umjesto da ih sam stvara. Ovaj obrazac povećava modularnost i testabilnost koda jer ovisnosti mogu biti lako zamijenjene ili modificirane bez promjene samog objekta koji ih koristi.

Zašto koristiti Dependency Injection?

1. **Razdvajanje ovisnosti:** Kôd postaje manje zavisan o specifičnim implementacijama.
2. **Lakše Testiranje:** Korištenjem DI, možete lakše zamijeniti stvarne ovisnosti s [mock](#) ili stub³ (zamjenskim programskim) objektima za testiranje.
3. **Povećana Fleksibilnost:** Ovisnosti se mogu jednostavno mijenjati bez potrebe za izmjenom koda u kojem se koriste.

Vrste Injekcije ovisnosti

1. **Konstruktorska injekcija:** Ovisnosti se prosljeđuju kroz konstruktor objekta.
2. **Setter-ska injekcija:** Ovisnosti se prosljeđuju kroz setter metode objekta.
3. **Metodna Injekcija:** Ovisnosti se prosljeđuju direktno u metode objekta.

Primjer Dependency Injection-a u PHP-u

Konstruktorska Injekcija

Konstruktorska injekcija je najčešće korištena metoda, gdje se ovisnosti prosljeđuju putem konstruktora klase.

```
class Database {  
    // Pretpostavimo da Database klasa ima potrebne metode  
}  
  
class UserRepository {  
    private $database;  
  
    // Database instanca se prosljeđuje preko konstruktora  
    public function __construct(Database $database) {  
        $this->database = $database;  
    }  
  
    public function getUser($userId) {  
        // Koristi $this->database za dohvaćanje korisnika iz baze  
    }  
}
```

³ U kontekstu testiranja softvera, stub je jedan od nekoliko vrsta testnih duplikata (engl. test doubles) koji se koriste za testiranje jedinica (unit testing). Stubovi su jednostavne implementacije ili zamjene za stvarne komponente ili module koje omogućavaju kontrolu nad testnim okruženjem. Oni se koriste za simulaciju ponašanja stvarnih komponenti (funkcija, metoda ili objekta) na jednostavan način. Cilj korištenja stubova je omogućiti izolaciju dijela koda koji se testira, kako bi se mogao testirati u kontroliranim uslovima bez potrebe za oslanjanjem na stvarne implementacije koje mogu biti kompleksne, nepouzdane ili teško dostupne tokom testiranja.

Setterska Injekcija

Setterska injekcija koristi metode za postavljanje ovisnosti nakon što je objekt kreiran.

```
class UserRepository {  
    private $database;  
  
    public function setDatabase(Database $database) {  
        $this->database = $database;  
    }  
  
    public function getUser($userId) {  
        // Koristi $this->database za dohvaćanje korisnika iz baze  
    }  
}  
  
// Korištenje setterske injekcije  
$userRepo = new UserRepository();  
$userRepo->setDatabase(new Database());
```

Metodna Injekcija

Ovisnosti se prosljeđuju direktno u metodu kada je potrebno.

```
class UserRepository {  
    public function getUser($userId, Database $database) {  
        // Koristi $database za dohvaćanje korisnika iz baze  
    }  
}  
  
// Korištenje metodne injekcije  
$userRepo = new UserRepository();  
$userRepo->getUser(1, new Database());
```

Dependency Injection Container (DI Container)

DI Container je alat koji automatizira proces injekcije ovisnosti. On kreira i upravljainstancama objekata i njihovih ovisnosti.

Evo primjera jednostavnog DI kontejnera:

```
class Container {  
    private $instances = [];  
  
    public function set($name, $resolver) {  
        $this->instances[$name] = $resolver;  
    }  
}
```

```

public function get($name) {
    if (isset($this->instances[$name])) {
        return $this->instances[$name]($this);
    }

    throw new Exception("No entry found for {$name}");
}

// Definiranje ovisnosti
$container = new Container();

$container->set(Database::class, function() {
    return new Database();
});

$container->set(UserRepository::class, function($container) {
    return new UserRepository($container->get(Database::class));
});

// Dohvaćanje UserRepository instance s automatski injektiranim Database ovisnošću
$userRepo = $container->get(UserRepository::class);

```

Dependency Injection je ključna tehnika za stvaranje modularnog, testabilnog i fleksibilnog koda. Korištenjem različitih vrsta injekcije ovisnosti i DI Containera, možete efikasno upravljati ovisnostima unutar vaših PHP aplikacija.

```

Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/Napredni PHP/Algeb
raCRM (master)
$ git remote add origin git@github.com/tkescec-algebra/algebra-crm.git

```

```

$ git remote -v
origin  git@github.com/tkescec-algebra/algebra-crm.git (fetch)
origin  git@github.com/tkescec-algebra/algebra-crm.git (push)

Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/Napredni PHP/Algeb
raCRM (master)
$ 

```

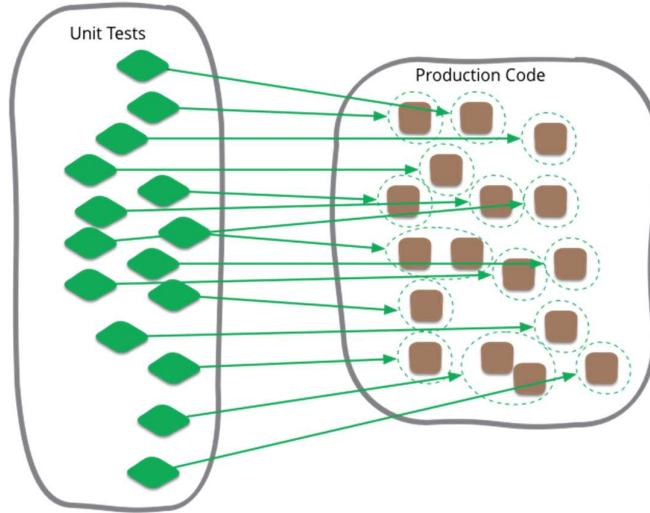
Testiranje

Vrste testova

Postoji nekoliko vrsta testova koje se često koriste u razvoju softvera, a Evo pregleda različitih vrsta testova:

Jedinični testovi (Unit tests)

- **Opis:** Provjeravaju pojedinačne komponente (jedinice) koda, kao što su funkcije, metode ili klase, izolirano od ostatka aplikacije.
- **Primjer:** Provjera povratnih vrijednosti, stanja objekata ili iznimki unutar jedinica koda.
- **PHPUnit podrška:** PHPUnit je izvrstan alat za pisanje i izvršavanje jediničnih testova u PHP-u. Pruža različite assertivne metode kao što su `assertEquals`, `assertTrue`, `expectException` i druge za provjeru očekivanog ponašanja jedinica koda.



Integracijski testovi (Integration tests)

- **Opis:** Provjeravaju kako se različite komponente ili cijeli sustav ponašaju kada su integrirane zajedno. Laravel ih naziva Testovi mogućnosti (engl. Feature tests)
- **Primjer:** Testiranje integracije između različitih servisa, baza podataka, API-ja ili frontend-backend integracija.
- **PHPUnit podrška:** PHPUnit podržava pisanje integracijskih testova gdje možete testirati složenije scenarije koji uključuju više komponenti vaše aplikacije.

Funkcionalni testovi (Functional tests)

- **Opis:** Testiraju funkcionalnosti aplikacije na visokoj razini, često simulirajući stvarne korisničke interakcije.

- **Primjer:** Testiranje korisničkog interface-a (UI testiranje), provjera korisničkih scenarija, testiranje API-ja ili interakcija s vanjskim servisima.
- **PHPUnit podrška:** Možete koristiti PHPUnit za funkcionalno testiranje, na primjer integrirajući ga s Selenium WebDriver-om za testiranje web aplikacija ili za provjeru API-ja.

End-to-end testovi (E2E tests)

- **Opis:** Testiraju cijelokupan tok kroz aplikaciju, od ulaza korisnika do izlaza bez simuliranja pojedinačnih komponenti.
- **Primjer:** Automatizirano testiranje korisničkih scenarija od početka do kraja, uključujući sve komponente sustava.
- **PHPUnit podrška:** PHPUnit nije primarno alat za end-to-end testiranje, ali se može kombinirati s drugim alatima kao što su Selenium, Codeception ili Cypress za izvršavanje end-to-end testova.

Performansijski testovi (Performance tests)

- **Opis:** Provjeravaju brzinu, otpornost i skalabilnost aplikacije pod određenim opterećenjem.
- **Primjer:** Testiranje brzine odgovora API-ja pod različitim opterećenjima ili provjera maksimalnih kapaciteta sustava.
- **PHPUnit podrška:** PHPUnit se ne koristi izravno za performansijsko testiranje. Za to su bolji alati poput Apache JMeter, Loader.io ili slični specijalizirani alati.

Sigurnosni testovi (Security tests)

- **Opis:** Provjeravaju sigurnosne ranjivosti u aplikaciji ili sustavu.
- **Primjer:** Testiranje XSS ranjivosti, SQL injection, autentifikacije i autorizacije.
- **PHPUnit podrška:** PHPUnit nije specijaliziran za sigurnosno testiranje. Za to su potrebni alati i tehnike specifične za sigurnosno testiranje.

Regresijski testovi (Regression tests)

- **Opis:** Provjeravaju da li nove promjene u kodu nisu uzrokovale pogoršanje postojećih funkcionalnosti.
- **Primjer:** Ponovno testiranje funkcionalnosti nakon izmjene koda ili integracije novih značajki.
- **PHPUnit podrška:** PHPUnit se može koristiti za pisanje regresijskih testova kako bi se osiguralo da nova funkcionalnost ne narušava postojeće funkcionalnosti.

PHPUnit se uglavnom koristi za jedinično testiranje i integracijsko testiranje u PHP projektima. To znači da je najčešće korišten za testiranje pojedinačnih dijelova koda (jedinica) i za testiranje integracije između različitih dijelova aplikacije ili sustava. Za druge vrste testova, poput funkcionalnih, end-to-end, performansijskih ili sigurnosnih testova, možda će biti potrebno koristiti druge alate ili kombinirati PHPUnit s drugim alatima koji su specijalizirani za te vrste testiranja.

Vrste integracijskih testova

Sistemski integracijske testovi:

Ovi testovi testiraju integraciju cijelog sustava, uključujući sve njegove komponente, odnosno kako se cijela aplikacija ili servis ponaša kada se pokrene kao cjelina. To može uključivati integraciju između više servisa, baza podataka, vanjskih sustava itd.

API integracijski testovi:

Ovi testovi provjeravaju integraciju između različitih API-ja (Application Programming Interface). Na primjer, testiranje kako se vaš frontend aplikacije integrira s backend API servisom, provjera ispravnosti podataka koji se razmjenjuju putem API-ja, provjera statusnih kodova odgovora itd.

Baza podataka integracijski testovi

Ovi testovi fokusiraju se na provjeru kako se podatci pohranjuju, čitaju i mijenjaju unutar baze podataka. Mogu uključivati testiranje SQL upita, provjeru transakcijskog ponašanja, integraciju s ORM (Object-Relational Mapping) alatima itd.

Jedinični testovi se smatraju osnovnom vrstom testiranja i obično se izvode na najmanjoj razini, fokusirajući se na provjeru jedinica koda neovisno o njihovoj integraciji s ostalim dijelovima sustava. Stoga, iako jedinični testovi mogu biti važni u okviru testiranja softvera, oni se često ne spominju eksplicitno kao integracijski testovi jer se usredotočuju na ispravnost pojedinačnih dijelova koda izoliranih od ostatka aplikacije.

Postoje dva pristupa kod testiranja.

Prvi je isprogramirati funkcionalnost a onda za tu funkcionalnost napisati testove. Drugi pristup je Test-Driven Development (TDD). To je metoda razvoja softvera u kojoj se testovi pišu prije samog koda. Osnovna ideja je napisati test za određenu funkcionalnost, vidjeti kako taj test pada (jer funkcionalnost još nije implementirana), zatim napisati minimalni kod potreban da test prođe i na kraju refaktorirati kod uz osiguranje da testovi i dalje prolaze.

Test-Driven Development (TDD) i njegova primjena u PHP-u pomoću PHPUnit-a

Što je TDD?

Test-Driven Development (TDD) je metoda razvoja softvera u kojoj se testovi pišu prije samog koda. Osnovna ideja je napisati test za određenu funkcionalnost, vidjeti kako taj test pada (jer funkcionalnost još nije implementirana), zatim napisati minimalni kod potreban da test prođe i na kraju refaktorirati kod uz osiguranje da testovi i dalje prolaze. Ovaj ciklus se često sažima kao Crveno-Zeleno-Refaktoriši.

Prednosti i nedostaci TDD-a

TDD donosi brojne prednosti:

Poboljšana kvaliteta koda: Budući da se testovi pišu prvi, osigurava se da kod ispunjava zahtjeve. To smanjuje broj grešaka i pomaže u održavanju visoke kvalitete koda.

Poboljšanje dizajna: Pisanje testova prvo pomaže programerima da bolje razumiju zahtjeve i potiče pisanje odvojenog i modularnog koda. Ovo se često slaže sa SOLID principima objektno orijentiranog dizajna, promovirajući bolju arhitekturu softvera.

Dokumentacija: Testovi služe kao oblik dokumentacije koji objašnjava što bi kod trebao raditi.

Prevencija regresija: Kada se naprave promjene, postojeći testovi mogu brzo potvrditi da nijedna postojeća funkcionalnost nije pokvarena.

Nedostatak je što je ponekad potrebno puno vremena jer kod refaktoriranja ponovo testiramo i testovi moraju proći. U ovom trenutku zadnja verzija je 11.2.

PHPUnit

PHPUnit je framework za jedinično testiranje (unit testing) u PHP jeziku. Razvijen je za olakšavanje automatiziranog testiranja PHP aplikacija, omogućavajući programerima da pišu testove za svoj kod i provjeravaju njegovu funkcionalnost na automatiziran način. Potreban nam je

Glavna svojstva PHPUnit-a

1. **Jednostavnost integracije:** PHPUnit se integrira s Composer-om, što olakšava njegovu instalaciju i upotrebu u PHP projektima.
2. **Asertivni izrazi:** Pruža različite metode asertivnih izraza kao što su `assertEquals`, `assertTrue`, `assertFalse`, `assertEmpty`, `assertContains`, itd., što omogućuje provjeru očekivanih rezultata u testovima.
3. **Podrška za različite vrste testova:** Osim jediničnih testova (unit test), podržava i integracijske testove, funkcionalne testove, testiranje API-ja, testiranje korisničkog interface-a (UI), itd.
4. **Reportiranje rezultata:** Generira detaljne izvještaje o izvođenju testova, uključujući statistike o prolaznim i neuspješnim testovima, te detalje o svakom neuspješnom testu.
5. **Podrška za CI/CD:** Može se integrirati s alatima za kontinuiranu integraciju i isporuku (CI/CD) kako bi se automatiziralo izvršavanje testova tijekom razvojnog ciklusa.
6. **Fleksibilnost:** Omogućuje postavljanje specifičnih konfiguracija za testiranje, kao i korištenje raznih dodataka i ekstenzija za proširenje njegovih mogućnosti.

PHPUnit je popularan zbog svoje funkcionalnosti, pouzdanosti i aktivne podrške zajednice. Koristi se kao standardni alat za testiranje PHP aplikacija, pomažući u osiguranju kvalitete koda i smanjenju grešaka pri razvoju softvera.

Instalacija PHPUnit-a

1. Instalacija pomoću Composer-a:

PHPUnit se obično instalira pomoću Composer-a, popularnog alata za upravljanje PHP paketima, koji većimam instaliran.

Možemo i instalirati ovako:

```
composer require --dev phpunit/phpunit
```

Ova naredba će instalirati PHPUnit kao razvojni zahtjev (`--dev`), što znači da će PHPUnit biti dostupan samo u vašem razvojnoj okolini.

sa `--dev` kažemo da ga ne smjesti u require područje nego u require-dev područje. To je bitno zato što ono što je u require mora biti zadovoljeno a ono što je u `require-dev` ne mora.

```
$ composer require --dev phpunit/phpunit
./composer.json has been updated
Running composer update phpunit/phpunit
Loading composer repositories with package information
Updating dependencies
Lock file operations: 26 installs, 0 updates, 0 removals
- Locking myclabs深深-copy (1.12.0)
- Locking nikic/php-parser (v5.1.0)
- Locking phar-io/manifest (2.0.4)
- Locking phar-io/version (3.2.1)
- Locking phpunit/php-code-coverage (11.0.5)
- Locking phpunit/php-file-iterator (5.0.1)
- Locking phpunit/php-invoker (5.0.1)
- Locking phpunit/php-text-template (4.0.1)
- Locking phpunit/php-timer (7.0.1)
- Locking phpunit/phpunit (11.2.8)
- Locking sebastian/cli-parser (3.0.2)
- Locking sebastian/code-unit (3.0.1)
- Locking sebastian/code-unit-reverse-lookup (4.0.1)
- Locking sebastian/comparator (6.0.1)
- Locking sebastian/complexity (4.0.1)
```

...

```
- Installing sebastian/complexity (4.0.1): Extracting archive
- Installing sebastian/code-unit-reverse-lookup (4.0.1): Extracting archive
- Installing phpunit/php-code-coverage (11.0.5): Extracting archive
- Installing phar-io/version (3.2.1): Extracting archive
- Installing phar-io/manifest (2.0.4): Extracting archive
- Installing myclabs深深-copy (1.12.0): Extracting archive
- Installing phpunit/phpunit (11.2.8): Extracting archive
4 package suggestions were added by new dependencies, use 'composer suggest' to
see details.
Generating autoload files
24 packages you are using are looking for funding.
Use the 'composer fund' command to find out more!
No security vulnerability advisories found.
Using version ^11.2 for phpunit/phpunit

Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/NapredniPHP/ATgebr
aCRM (master)
```

Možemo primijetiti i da se poddirektorij `phpunit` i još poneki pojavili u u poddirektoriju `vendor`.

C:\ Disk (C:) > xampp >htdocs > Algebra > NapredniPHP > AlgebraCRM > vendor			
Name	Date modified	Type	Size
bin	21.7.2024. 16:40	File folder	
composer	21.7.2024. 16:40	File folder	
myclabs	21.7.2024. 16:40	File folder	
nikic	21.7.2024. 16:40	File folder	
phar-io	21.7.2024. 16:40	File folder	
phpunit	21.7.2024. 16:40	File folder	
sebastian	21.7.2024. 16:40	File folder	
theseer	21.7.2024. 16:40	File folder	
autoload.php	20.7.2024. 16:39	PHP Source File	1 KB

Atomatski se ubacio dio u [composer.json](#):

```
{  
    "name": "administrator/algebra-crm",  
    "type": "project",  
    "license": "MIT",  
    "autoload": {  
        "psr-4": {  
            "App\\": "src/"  
        }  
    },  
    "authors": [  
        {  
            "name": "Branko Ćelap",  
            "email": "branko.celap@gmail.com"  
        }  
    ],  
    "require": {  
        "php": ">=8.2"  
    },  
    "require-dev": {  
        "phpunit/phpunit": "^11.2"  
    }  
}
```

Možemo instalirati PHPUnit globalno iz terminala ali to nećemo raditi:

```
composer global require phpunit/phpunit
```

Ova naredba će globalno instalirati PHPUnit na vaš sistem, omogućavajući vam da ga koristite bilo gdje.

Prvo je potrebno odrediti što ćemo testirati. To će biti `Router.php`. Znamo kako Router treba raditi i treba napisati testove koji će nam reći da Router ne radi kako treba. Pokrenemo kod da nešto radi i onda radimo provjeru (engl. assert) da li se taj komad koda ponaša u skladu s očekivanjem, da li je vratio očekivani odgovor, da li je vratio Exception.

Napraviti ćemo test funkcije za zbrajanje. Napišemo funkciju kojoj pošaljemo dva argumenta i ona vraća rezultat. Test mora pokriti i ako pošaljemo string ili dva stringa ili matricu. Ako u rezultatu ne dobijemo očekivano ponašanje assert će doživjeti neuspjeh (engl. fail). Test ćemo napraviti tako da testiramo očekivano ponašanje, ne samo pozitivno, nego i ono kada funkcija ne radi. Što kasnije otkrijemo grešku to dovodi do bug-a.

2. Verifikacija instalacije:

Nakon instalacije, provjerite da li je PHPUnit uspješno instaliran pomoću:

```
phpunit --version
```

Trebali biste vidjeti verziju PHPUnit-a koju ste instalirali.

Testiranje

Testiranje s PHPUnit-om u PHP-u uključuje nekoliko osnovnih koraka kako bi se osiguralo da vaš kod radi ispravno i očekivano. Evo detaljnog vodiča kako vršiti testiranje s PHPUnit-om:

Pokretanje testova

Da biste pokrenuli testove, pozicionirajte se u korijenski direktorij vašeg projekta i izvršite PHPUnit:

```
./vendor/bin/phpunit
```

Direktorij `vendor` je standardno mjesto gdje Composer instalira sve pakete i njihove izvršne skripte. Izvršna skripta PHPUnit koja se nalazi u direktoriju `bin` unutar `vendor` direktorija. PHPUnit se instalira kao Composer paket, i njegova izvršna skripta (`phpunit`) se nalazi u poddirektoriju `bin` unutar `vendor` direktorija.

Ova naredba je posebno korisna jer omogućava izvršavanje PHPUnit testova iz bilo kojeg direktorija u vašem projektu, osiguravajući da koristite pravu verziju PHPUnit-a koja je instalirana za vaš projekt pomoću Composer-a.

PHPUnit će automatski pronaći sve testove u direktoriju `tests` (i poddirektorijima `unit` i `integration`) i izvršiti ih. Rezultati testova će biti prikazani u terminalu, zajedno s informacijama o prolazu ili neuspjehu svakog testa.

```
./vendor/bin/phpunit --bootstrap vendor/autoload.php tests
```

Priprema vašeg projekta

Prije nego što započnete s testiranjem, važno je da imate dobro organiziran projekt koji uključuje izvore (engl. source code) koje želite testirati i PHPUnit konfiguriran za vaš projekt. Potrebno je otvoriti spomenute poddirektorije `tests` u korijenskom direktoriju projekta i u njemu `unit` i `integration`.

Struktura testova

PHPUnit očekuje da vaši testovi budu organizirani u odgovarajućoj strukturi direktorija. Standardna praksa je da se testovi nalaze u direktoriju `tests` unutar korijenskog direktorija vašeg projekta.

Primjer strukture direktorija:

```
projekt/
└── src/
    └── (vaš PHP originalni kod)
└── tests/
    └── integration/
        └── unit/
            └── PrimjerTest.php
```

Pisanje testova

Nakon što imate strukturu, možete početi pisati svoje testove. PHPUnit koristi određene konvencije za pisanje testova.

Ovdje smo napisali novu PHP datoteku unutar `tests` direktorija, npr. `CalculatorTest.php`.

```
<?php
use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase {
    public function testAdd() {
        $calculator = new Calculator();
        $result = $calculator->add(1, 2);
        $this->assertEquals(3, $result);
    }
}
```

Testovi jedinica pišu se koristeći PHPUnit API. Svaki test je metoda klase koja nasljeđuje `PHPUnit\Framework\TestCase`.

Razumijevanje rezultata

Kada PHPUnit završi izvršavanje testova, možete vidjeti detaljne informacije o svakom testu:

- **Prolazni testovi:** Ako su svi asertivni izrazi u testu istiniti, test se smatra prolaznim.
- **Neuspješni testovi:** Ako bilo koji asertivni izraz u testu nije istinit, test se smatra neuspješnim. PHPUnit će vam pružiti informacije o tome koji asert je bio neuspješan i očekivane vs. stvarne vrijednosti.

Koncept PHPUnit-a

PHPUnit nudi mnoge dodatne mogućnosti za testiranje, uključujući mock-ove, data providers, i različite vrste asertivnih izraza. Preporučljivo je istražiti PHPUnit dokumentaciju za potpunu preglednost svih funkcionalnosti koje nudi.

1. Struktura testne klase

- Svaka testna klasa nasljeđuje TestCase klasu iz PHPUnit-a.
- Testne metode moraju počinjati s test ili koristiti @test anotaciju.

2. Asertivni izrazi

- Koristite asertivne metode za provjeru očekivanih rezultata, kao što su `assertEquals`, `assertTrue`, `assertFalse`, itd.
- Razumijevanje asertivnih izraza pomaže u pisanju učinkovitih testova.

3. Data Providers

- Korištenje data providers omogućuje vam testiranje jedne metode s različitim skupovima podataka.

4. Mock-ovi

- Mock-ovi se koriste za simulaciju objekata i njihove interakcije u testovima.

Ukratko, PHPUnit je moćan alat za jedinično testiranje u PHP-u koji omogućuje automatizirano testiranje vašeg koda kako biste osigurali njegovu ispravnost i pouzdanost. Integrirajte ga u svoj razvojni proces kako biste smanjili greške i olakšali održavanje vaše aplikacije.

Dodatne vještine i alati

1. Continuous Integration (CI)

- Integracija PHPUnit-a s CI alatima kao što su GitHub Actions, Travis CI ili Jenkins omogućuje automatsko pokretanje testova pri svakoj promjeni koda.

2. Code Coverage

- Alati kao što je Xdebug mogu se koristiti za mjerjenje pokrivenosti koda testovima. To pomaže u razumijevanju koliko je vaš kod testiran.

```
./vendor/bin/phpunit --coverage-html coverage
```

3. Debugging

- Naučite kako koristiti debuggere kao što su Xdebug za otkrivanje i ispravljanje grešaka u testovima.

Testiranje naše aplikacije unit testovima

Vratimo se testiranju naše aplikacije. Otvorit ćemo u unit poddirektoriju `RouterTest.php` datoteku.

Sva imena datoteka trebaju završavati na `Test` da ne bi smo morali ručno PHPUnitu govoriti gdje je test.

Bitno je da sve Test klase se nasljeđuju od `TestCase`. `TestCase` dolazi PHPUnit frameworka i možemo ga vidjeti u `vendor\phpunit\src\Framework\TestCase.php`. Svi naši testovi se mogu zavrtjeti. `TestCase` ekstenda `Assert`.

Datoteku treba nazvati po unitu koji testiramo, to može biti klasa ili pojedina metoda i na kraju `Test`. Svaki test može imati `setUp` metodu. Ova metoda se automatski poziva pre svakog testa. Prilikom `setUp` možemo podesiti predefinirane vrijednosti koje će se izvršiti. Na to možemo gledati kao na neki konstruktor. Vratit ćemo se `setUp` metodi i napisati je kasnije.

Pogledajmo kako bi mogli testirati Router. Router je singleton. Možemo testirati da li `getInstance` daje uvijek isti objekt, po tome ćemo znati da li je `Router` singleton.

```
class RouterTest extends TestCase
```

Idemo napisati `testGetInstance()` koji će testirati da li imamo singleton. To ćemo napraviti na sljedeći način:

```
public function testGetInstance()
{
    $router1 = Router::getInstance();
    $router2 = Router::getInstance();

    $this->assertSame($router1, $router2);
}
```

Napravimo dva objekta i provjerimo da li `getInstance()` metoda vraća isti objekt svaki puta tj. da li imamo singleton obrazac. Ako obje varijable referenciraju na isti objekt sigurni smo da imamo singleton. Koristimo `assertSame()` da proveri da li su dva objekta identična (ist tip i vrijednost).

Da bi smo pokrenuli ovaj test dovoljno je unijeti u Git Bash:

```
./vendor/bin/phpunit tests/unit/RouterTest.php
```

Kasnije ćemo to riješiti da ne moramo ručno unositi. Testiranje će dojaviti da je prošlo.

```
$ ./vendor/bin/phpunit tests/unit/RouterTest.php
PHPUnit 11.2.8 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.12
Configuration: C:\xampp\htdocs\Algebra\NapredniPHP\AlgebraCRM\phpunit.xml

...
1 / 1 (100%)

Time: 00:00.055, Memory: 8.00 MB
```

```
OK (1 tests, 1 assertions)

Administrator@DESKTOP-4J5DF41 MINGW64
/c/xampp/htdocs/Algebra/NapredniPHP/AlgebraCRM (master)
$
```

Recimo da u nekom trenutku promijenimo Router i testiranje ne prođe jer nije singleton.

```
public static function getInstance(): RouterInterface
{
    self::$router = new Router();
}
```

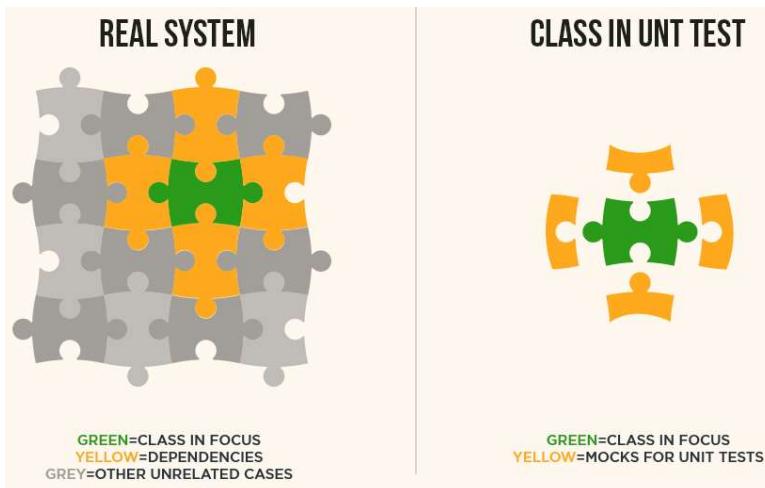
Ako sada probamo testirati test će dojaviti F što označava failure. Test je pukao jer nije singleton. Mi možemo zaboraviti da je trebalo biti singleton. Ako dođe zahtjev da se promjeni Router, vidjet ćemo kasnije da smo iz nekog razloga test napravili da testira da li je metoda singleton. Nedostatak Unit testa je da neće reći gdje to neće raditi, zato su tu Integracijski testovi. Unit test može ponekad dati krive ili nepotpune informacije.

Kada vratimo na singleton test će proći i bit će sve u redu.

```
public static function getInstance(): RouterInterface
{
    if (self::$router === null) {
        self::$router = new Router();
    }
}
```

Ovo nije dovoljno. Moram vidjeti što se dešava s ostalim stvarima, radi li npr. `dispatch()`. Kod ove metode nije jasno kako ćemo dobiti request u superglobalnoj varijabli bez stvarnog zahtjeva.

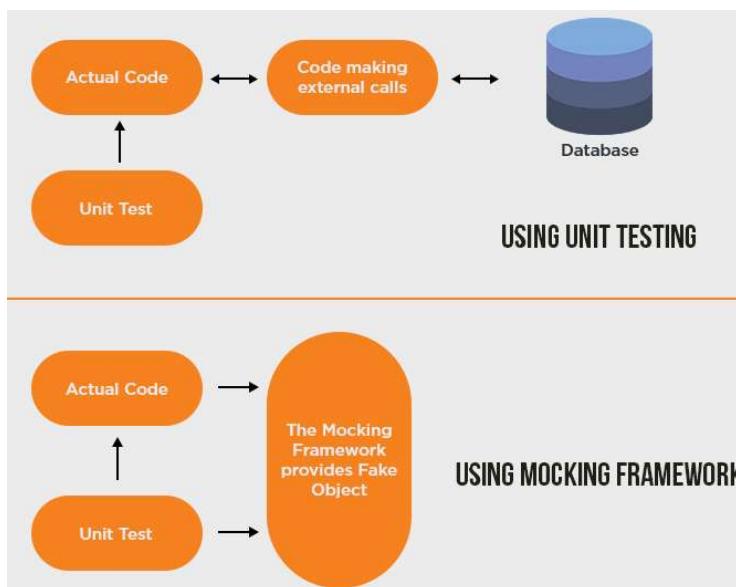
```
$method = $_SERVER['REQUEST_METHOD']
```



Moramo simulirati te uslove. Moramo napraviti [mock-ove](#). To su setovi potrebnih informacija koje bi imali da se dogodi stvarni zahtjev. Ako je testirana metoda `get`, očekivano ponašanje je s GET ključem zapiše rutu. To znači da ako testiramo `dispatch`, za tu metodu `get`, onda je očekivano da `REQUEST_METHOD`, bude GET. Onda ćemo to sami izdefinirati tj. mock-ati. To znači da superglobalnu varijablu `$_SERVER` moramo fake-ati tj. stavit je u mock. Upravo to ćemo napraviti `setUp()` metodi.

```
protected function setUp(): void
{
    $_SERVER['REQUEST_METHOD'] = '';
    $_SERVER['REQUEST_URI'] = '';
}
```

Postavili smo ih prazne.



Da bi smo testirali `dispatcher()` trebamo Kontroler. Nećemo testirati pravi kontroler nekog ga trebamo mock-ati.

```
class TestController
{
    public function testAction()
    {
        echo 'Test Action';
    }
}
```

Napraviti ćemo `testDispatchSuccess`.

```
public function testDispatchSuccess()
{
    $_SERVER['REQUEST_METHOD'] = 'GET';
    $_SERVER['REQUEST_URI'] = APP_PATH . '/test';

    $router = Router::getInstance();
    $router->get('/test', TestController::class, 'testAction');

    $this->expectOutputString('Test Action');
    $router->dispatch();
}
```

Ova metoda postavlja `REQUEST_METHOD` na `GET` i `REQUEST_URI` na `APP_PATH . '/test'`.

Nakon toga instancira Router, u Router gurne rutu `/test` koja instancira na `TestController` klasu i `testAction` metodu. Testiramo `dispatch()` ali ujedno i `get()` metodu. `get()` mora ispravno umetnuti informaciju u matricu koju `dispatch()` kada uhvati treba usporediti request URI s ovime što je ovdje. Instancirati `TestController` da li radi. Ako kaže `expectOutputString`, očekujemo kao odgovor `Test Action`. Možemo reći da je dispatch uspješno odradio test.

Ako test sada pokrenemo, dobit ćemo grešku da nije definirana konstanta „`App\Services\APP_URL`“. Tek konstante nemamo u `RouterTest.php`. Pogledajmo u `Router`. U metodi `get` zovemo `addRoute`, u `addRoute` imamo `APP_URL`. Kako nismo napravili stvarni request nego smo ga simulirali mock-om, konstanta `APP_URL` nije inicijalizirana u `app.php`. Zbog toga u `RouterTest.php` u `setUp()` trebamo dodati tu konstantu kao `define('APP_URL', 'http://localhost:8081')`. Ako pokrenemo sada opet će javiti da nedostaje `APP_ROOT_URL`. Moramo dodati i taj red: `define('APP_ROOT_URL', 'http://localhost:8081')`. Ako od početka ne napravimo testiranje, morat ćemo trošiti puno vremena što diže ukupan trošak.

Ako sada pokrenemo testove, opet dobijemo grešku `HttpNotFoundException: Not Found` a li ne dobijemo poruku `Test Action`. To znači da `dispatch` je odradio `throw new HttpNotFoundException()` i nije pronašao u rutama za metodu `get()` ispravan URI. Problem je sistem

kojim smo ubacivali konstante. Trebamo pripaziti na testu da integriramo nešto što ćemo izvući ovdje kao `APP_PATH`:

```
define('APP_ROOT_URL', 'http://localhost:8081');
define('APP_PATH', '/Algebra/NapredniPHP/AlgebraCRM');
define('APP_URL', APP_ROOT_URL . APP_PATH);
```

Metoda `setUp` u `RouterTest.php` je potencijalni problem. Kada se pokrene prvi test pokrene se `setUp` metoda i definiramo konstantu `APP_ROOT_URL`, kada pokrenemo drugi test dojavljujemo da je ta konstanta već definirana ali samo kao upozorenje (engl. warning). Na žalost od PHPUnit-a ne dobijemo detalje.

```
./vendor/bin/phpunit --debug tests/unit/RouterTest.php
```

Stare verzije koriste su `--verbose` umjesto `--debug`.

S ovom naredbom vidimo kako phpunit radi:

```
PHPUnit Started (PHPUnit 11.2.8 using PHP 8.2.12 (cli) on WINNT)
Test Runner Configured
Test Suite Loaded (2 tests)
Event Facade Sealed
Test Runner Started
Test Suite Sorted
Test Runner Execution Started (2 tests)
Test Suite Started (RouterTest, 2 tests)
Test Preparation Started (RouterTest::testGetInstance)
Before Test Method Called (RouterTest::setUp)
Before Test Method Finished:
- RouterTest::setUp ←
Test Prepared (RouterTest::testGetInstance)
Test Passed (RouterTest::testGetInstance)
Test Finished (RouterTest::testGetInstance)
Test Preparation Started (RouterTest::testDispatchSuccess)
Test Triggered PHP Warning (RouterTest::testDispatchSuccess)
Constant APP_ROOT_URL already defined
Test Triggered PHP Warning (RouterTest::testDispatchSuccess)
Constant APP_PATH already defined
Test Triggered PHP Warning (RouterTest::testDispatchSuccess)
Constant APP_URL already defined
Before Test Method Called (RouterTest::setUp)
Before Test Method Finished:
- RouterTest::setUp ←
Test Prepared (RouterTest::testDispatchSuccess)
Test Passed (RouterTest::testDispatchSuccess)
Test Finished (RouterTest::testDispatchSuccess)
Test Suite Finished (RouterTest, 2 tests)
Test Runner Execution Finished
Test Runner Finished
PHPUnit Finished (Shell Exit Code: 0)
```

Vidimo da trenutno postoje dva testa. Također vidimo da je problem što smo `APP_ROOT_URL` i `APP_URL` probali definirati dva puta.

Ovo ćemo riješiti tako da maknemo ove konstante iz `RouterTest.php` metode `setUp` ili možemo staviti `if`:

```
if (!defined('APP_ROOT_URL')){
    define('APP_ROOT_URL', 'http://localhost:8081');
}
```

Bolja praksa je da izdvojimo ove konstante u poddirektorij `test` u `bootstrap.php`:

```
<?php

define('APP_ROOT_URL', 'http://localhost:8081');
define('APP_PATH', '/Algebra/NapredniPHP/AlgebraCRM');
define('APP_URL', APP_ROOT_URL . APP_PATH);
```

Mi ne želimo dati putanju do naših testova. Želimo pozvati samo `./vendor/bin/phpunit` a ona da starta apsolutno sve testove. Prije startanja testova želimo da `phpunit` pogleda u `bootstrap` datoteku.

Cilj je da ne testiramo ručno testove. Benefit ovoga je da se `bootstrap` datoteka samo jednom učita.

Ako probamo pokrenuti pojedinačni test bez `--debug` neće raditi jer test ne vidi iz `bootstrap.php` `APP_PATH`. Nigdje nismo rekli kada se pokreće test, da je `bootstrap.php` početna točka.

```
./vendor/bin/phpunit --debug tests/unit/RouterTest.php
```

Direktno smo pokrenuli test.

Ako ovo želimo automatizirati proces pokretanja testova tako da ako pokrenemo `./vendor/bin/phpunit` moramo kreirati xml shemu. Tamo je definiran plan izvršavanja testova. Tamo piše da je `bootstrap.php` ulazna točka (engl. entry point), i gdje se nalaze svi testovi (PHPUnit to naziva suits tj. ugađanje). Sada su to `integration` i `unit`, moguće ih je definirati i više (iako to nije praksa, moguće je). To radimo u root-u projekta. Dakle trebamo napraviti `phpunit.xml`:ili ga preuzeti iz dokumentacije: <https://docs.phpunit.de/en/11.2/organizing-tests.html#composing-a-test-suite-using-xml-configuration>.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/|version|/phpunit.xsd"
    bootstrap="tests/bootstrap.php">

<testsuites>
    <testsuite name="unit">
        <directory>tests/unit</directory>
    </testsuite>

    <testsuite name="integration">
        <directory>tests/integration</directory>
    </testsuite>
</testsuites>
</phpunit>
```

Konfiguracija nam potpuno odgovara. `bootstrap.php` je definiran u poddirektoriju `tests` kao kod nas a u direktorijma unit i integration se nalaze `testsuite`-ovi. Ako Nemamo konstante nije potrebno ni definirati `bootstrap.php`.

Pokrenemo li sada testiranje, izvršit će se svi testovi:

```
./vendor/bin/phpunit
```

Rezultat je:

```
$ ./vendor/bin/phpunit
PHPUnit 11.2.8 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.12
Configuration: C:\xampp\htdocs\Algebra\NapredniPHP\AlgebraCRM\phpunit.xml

...
2 / 2 (100%)

Time: 00:00.059, Memory: 8.00 MB

OK (2 tests, 2 assertions)

Administrator@DESKTOP-4J5DF41 MINGW64
/c/xampp/htdocs/Algebra/NapredniPHP/AlgebraCRM (master)
$
```

Do sada smo gledali kada smo uspješno dodali rutu i kada smo dispatch-ali zahtjev da li je dispatch uspješno distancirao neki Kontroler i da li smo dobili neki očekivani string. U našem slučaju kontroler je vratio `Test Action` string.

Napisat ćemo još jedan test `testDispatchFail()` koji želimo da ne funkcioniira:

```
public function testDispatchFail()
{
    $this->expectException(HttpNotFoundException::class);

    $_SERVER['REQUEST_METHOD'] = 'GET';
    $_SERVER['REQUEST_URI'] = APP_PATH . '/nonexist';

    $router = Router::getInstance();
    $router->get('/test1', TestController::class, 'testAction');

    $router->dispatch();
}
```

Ovdje kreiramo rutu `/test1` a u `REQUEST_URI` šaljemo `/nonexist`. Ta ruta ne postoji. Očekivano ponašanje našeg Router-a da vrati grešku a ne prazan string. Ako pogledamo `dispatch()` metodu u RouterTest vidjet ćemo da ako ne pronađe metodu, da baca s throw `HttpNotFoundException()`. To je očekivano ponašanje našeg Routera kada nema rute.

Zato s `$this->expectException(HttpNotFoundException::class)` očekujemo iznimku kada nema ispravne rute.

Ostatak testa postavlja `REQUEST_METHOD` na `GET` i `REQUEST_URI` na `APP_PATH . '/nonexist'` koji naravno ne postoji. Inicijalizira ruter i registrira putanju `/test1` koja mapira na `TestController` klasu i `testAction` metodu. Poziva `dispatch()` metodu, što baca izuzetak jer ruta `/nonexist` ne postoji.

Kada pokrenemo vidimo da je sve u redu.

To znači da promjena u našem Router-u znači da će doći do neke greške. Ako promjenimo u `dispatch()` throw i on sada glasi `throw new \Exception()` testiranje će prijaviti grešku. Dakle testovi moraju biti usklađeni.

Kada bismo testirali `register()` u AuthController neko ne napravio request, znači da ga i mi moramo mock-ati. Prvo trebamo simulirati `isAuthenticated()`, ako pogledamo `public` metoda `register()` koristi private svojstvo `$userModel`. Kako bi izbjegao sukob oko private access modifier-a napravimo privremeni objekt tj. napravimo mock s kojim ćemo raditi. `isAuthenticated()` provjerava da li je korisnik prijavljen u sistemu. Ako jeste, preusmjeri ga na `/admin/dashboard`.

Za `isAuthenticated()` možemo napisati dva testa. Prvi test je da korisnik nije prijavljen, očekuje se da krene dalje, morat ćemo to razgraditi. Drugi test je vidjeti kako se `/register` ponaša ako je korisnik prijavljen, da li će korisnik stvarno biti preusmjeren. Ako imamo validatore, moramo vidjeti kako oni rade, ima li emaila, ima li passworda itd. U priču ulazi Middleware, modeli, request tj. više stvari koje su u `/register`. Promjena može izazvati neočekivano ponašanje što nije error nego bug. To je područje integracijskih testova. Za to ćemo koristiti [Github Actions](#). To je sistem za stvaranje pipeline-ova za kontinuiranu integraciju i kontinuiranu implementaciju (engl. deployment) koji će olakšati proces podizanja na testne i kasnije produkcijske servere. Njihova je zadaća da izvrte testove i da onda kod negdje implementiraju. Poanta je da to automatizirate: spajanje negdje sa SSH, povlačenje sa GitHub-a, povlačenje promjena itd.



Postoji i Travis koji ima istu svrhu kao i Github Actions. Postoji i Jenkins. Jenkins može raditi i s JIR-om.



Bitbucket ima svoje pipelines i moguće je koristiti njegov workflows.

YAML je je jezik za serijalizaciju podataka čitljiv ljudima. Obično se koristi za konfiguracijske datoteke i aplikacije u kojima se podaci pohranjuju ili prenose.



Podržava ga Github Actions.

Što su mock-ovi?

Mock-ovi su objekti koji simuliraju ponašanje stvarnih objekata u kontroliranim načinima. Oni su ključan alat u jediničnom testiranju jer omogućavaju testiranje dijelova koda izolirano od vanjskih ovisnosti, poput baza podataka, API-ja ili drugih servisa. Mock-ovi pomažu u:

- Izolaciji koda: Omogućavaju testiranje specifične jedinice koda bez uključivanja cijelog sustava.
- Kontroli okruženja: Simuliraju različite uvjete i odgovore, omogućavajući testiranje kako će kod reagirati u različitim situacijama.
- Ubrzavanju testiranja: Smanjuju vrijeme potrebno za postavljanje i izvršavanje testova jer nema potrebe za pristupom stvarnim resursima.

Kako se definiraju mock-ovi u PHPUnit-u?

PHPUnit pruža podršku za stvaranje mock-ova putem ugrađenih metoda. Evo osnovnih koraka za korištenje mock-ova s PHPUnit-om:

1. Kreiranje mock objekta:

- Koristite metodu `createMock` ili `getMockBuilder` za stvaranje mock objekta.

2. Postavljanje očekivanja:

- Definirajte očekivano ponašanje mock objekta, kao što su povratne vrijednosti metoda, broj poziva metoda, itd.

3. Korištenje mock objekta:

- Uključite mock objekt u testiranu jedinicu koda.

Data Providers

Data providers u PHPUnit-u omogućavaju da jedna testna metoda bude izvršena više puta s različitim skupovima podataka. Ovo je korisno kada želite testirati istu funkcionalnost s različitim ulaznim vrijednostima i očekivanim rezultatima.

Kako se koristi data providers u PHPUnit-u

1. Definiranje data provider metode:

- Metoda koja pruža podatke treba vraćati niz nizova. Svaki podniz predstavlja skup ulaznih podataka za jedan poziv testne metode.

2. Povezivanje data provider metode s test metodom:

- Koristite `@dataProvider` anotaciju iznad test metode da biste je povezali s data provider metodom.

Primjer korištenja data providers u PHPUnit-u

Prepostavimo da imamo klasu `Calculator` s metodom `add` koja sabira dva broja. Želimo testirati ovu metodu s različitim parovima brojeva.

```
<?php
```

```
class Calculator {
    public function add($a, $b) {
        return $a + $b;
    }
}
```

Sada ćemo napisati test za ovu metodu koristeći data provider.

```
<?php
use PHPUnit\Framework\TestCase;

class CalculatorTest extends TestCase {
    /**
     * @dataProvider additionProvider
     */
    public function testAdd($a, $b, $expected) {
        $calculator = new Calculator();
        $this->assertEquals($expected, $calculator->add($a, $b));
    }

    public function additionProvider() {
        return [
            [1, 2, 3],
            [0, 0, 0],
            [-1, 1, 0],
            [100, 200, 300],
        ];
    }
}
```

Definiranje data provider metode:

`additionProvider` metoda vraća matricu matrica. Svaka podmatrica sadrži tri elementa: dva ulazna broja i očekivani rezultat.

Ovdje definiramo četiri seta podataka: [1, 2, 3], [0, 0, 0], [-1, 1, 0], [100, 200, 300].

Povezivanje data provider metode s test metodom:

`@dataProvider additionProvider` anotacija iznad `testAdd` metode povezuje ovu testnu metodu s `additionProvider` metodom.

Kada PHPUnit izvršava test, `testAdd` metoda će biti pozvana četiri puta, svaki put s jednim od matrica podataka iz `additionProvider` metode.

Testna metoda:

testAdd metoda prima tri argumenta: dva ulazna broja (\$a i \$b) i očekivani rezultat (\$expected).

Unutar testne metode, kreiramo instancu `Calculator` klase i pozivamo `add` metodu s ulaznim brojevima, a zatim koristimo `assertEquals` da provjerimo je li rezultat jednak očekivanom.

Prednosti korištenja data providers

Pojednostavljenje koda: Umjesto pisanja više testnih metoda za različite ulazne vrijednosti, možete koristiti jednu metodu s data providerom.

Povećanje pokrivenosti: Možete jednostavno testirati funkcionalnost s različitim kombinacijama podataka.

Čitljivost: Testne metode su kraće i jasnije jer se podaci definiraju odvojeno.

Data providers su moćan alat u PHPUnit-u koji vam omogućuje efikasno testiranje s različitim ulaznim podacima. Korištenje data providers poboljšava održavanje testova i osigurava veću pokrivenost vašeg koda.

Asertivni izrazi u PHPUnit-u

Asertivni izrazi su metode koje koristimo u PHPUnit-u za provjeru očekivanih rezultata u testovima. One pomažu u određivanju da li je određeni test prošao ili nije, bazirano na usporedbi stvarnih i očekivanih vrijednosti. Evo pregleda različitih vrsta asertivnih izraza koje možete koristiti u PHPUnit-u:

1. `assertEquals($expected, $actual)`

Provjerava da li su dvije vrijednosti jednake.

2. `assertTrue($condition)`

Provjerava da li je uslov `true`.

```
$this->assertTrue(1 < 2);
```

3. `assertFalse($condition)`

Provjerava da li je uslov `false`.

```
$this->assertFalse(1 > 2);
```

4. `assertNull($actual)`

Provjerava da li je vrijednost `null`.

```
$this->assertNull(null);
```

5. `assertNotNull($actual)`

Provjerava da li vrijednost nije `null`.

```
$this->assertNotNull(5);
```

6. `assertSame($expected, $actual)`

Provjerava da li su dvije vrijednosti identične (ist tip i vrijednost).

```
$this->assertSame(3, 3);
$this->assertSame('abc', 'abc');
```

7. `assertNotSame($expected, $actual)`

Provjerava da li dvije vrijednosti nisu identične.

```
$this->assertNotSame(3, '3');
```

8. `assertContains($needle, $haystack)`

Provjerava da li matrica sadrži određenu vrijednost.

```
$this->assertContains(2, [1, 2, 3]);
```

9. `assertNotContains($needle, $haystack)`

Provjerava da li niz ne sadrži određenu vrijednost.

```
$this->assertNotContains(4, [1, 2, 3]);
```

10. `assertCount($expectedCount, $haystack)`

Provjerava broj elemenata u matrici.

```
$this->assertCount(3, [1, 2, 3]);
```

11. `assertEmpty($actual)`

Provjerava da li je matrica prazna.

```
$this->assertEmpty([]);
```

12. `assertNotEmpty($actual)`

Provjerava da li matrica nije prazna.

```
$this->assertNotEmpty([1, 2, 3]);
```

13. `assertInstanceOf($expected, $actual)`

Provjerava da li objekt pripada određenoj klasi ili implementira određeni interface.

```
$this->assertInstanceOf(MyClass::class, $myObject);
```

14. `assertStringContainsString($needle, $haystack)`

Provjerava da li string sadrži podstring.

```
$this->assertStringContainsString('foo', 'foobar');
```

15. `assertStringStartsWith($prefix, $string)`

Provjerava da li string počinje određenim prefiksom.

```
$this->assertStringStartsWith('foo', 'foobar');
```

16. `assertStringEndsWith($suffix, $string)`

Provjerava da li string završava određenim sufiksom.

```
$this->assertStringEndsWith('bar', 'foobar');
```

Asertivni izrazi su osnovni alati u PHPUnit-u za provjeru ispravnosti koda tokom testiranja. Korištenjem različitih asertivnih metoda možete provjeriti razne uvjete i osigurati da vaš kod radi kako je očekivano. Razumijevanje i pravilna upotreba ovih metoda ključni su za učinkovito testiranje i održavanje kvalitete koda.

Stub u PHP-u

Stub u PHP-u može biti implementiran na različite načine, ali osnovna ideja je zamijeniti stvarne objekte ili njihove metode sa simuliranim (stubiranim) verzijama koje vraćaju unaprijed definirane vrijednosti. Ovo je korisno kada želite izolirati dio koda koji testirate i eliminirati utjecaj drugih dijelova sistema.

Primjer korištenja stubova u PHPUnit-u

Pretpostavimo da imate klasu `UserService` koja koristi objekt `UserRepository` za dohvati korisničkih podataka iz baze. Kada testirate `UserService`, ne želite se oslanjati na stvarnu bazu podataka. Umjesto toga, koristite stub za `UserRepository`. Definirajmo klasu:

```
// UserRepository.php
namespace App\Repositories;

class UserRepository {
    public function find($id) {
        // Ovdje bi obično bio upit prema bazi podataka
    }
}

// UserService.php
namespace App\Services;

use App\Repositories\UserRepository;

class UserService {
    protected $userRepository;

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

    public function getUser($id) {
        return $this->userRepository->find($id);
    }
}
```

Kreirajmo stub i test:

```
// UserServiceTest.php
namespace Tests;

use PHPUnit\Framework\TestCase;
use App\Services\UserService;
use App\Repositories\UserRepository;

class UserServiceTest extends TestCase {
    public function test GetUser() {
        // Kreirajte stub za UserRepository
        $userRepositoryStub = $this->createStub(UserRepository::class);
```

```

    // Konfigurirajte stub da vrati određene vrijednosti za određene metode
    $userRepositoryStub->method('find')
        ->willReturn(['id' => 1, 'name' => 'John Doe']);

    // Injektirajte stub u UserService
    $userService = new UserService($userRepositoryStub);

    // Testirajte getUser metodu
    $result = $userService->getUser(1);

    // Assercija (provjera) rezultata
    $this->assertEquals(['id' => 1, 'name' => 'John Doe'], $result);
}
}

```

```
$userRepositoryStub = $this->createStub(UserRepository::class);
```

Ovime kreiramo stub za klasu `UserRepository`. Nakon ovoga ide konfiguiriranje stub-a;

```
$userRepositoryStub->method('find')->willReturn(['id' => 1, 'name' => 'John Doe']);
```

Podesili smo stub da metoda `find` uvijek vraća određenu vrijednost. Sada ćemo riješiti injektiranje stuba:

```
$userService = new UserService($userRepositoryStub);
```

Ovo kreira instancu `UserService` koristeći `stub` umjesto stvarnog `UserRepository` objekta.

Sada idemo na testiranje metode:

```
$result = $userService->getUser(1);
```

Ovo poziva metodu `getUser` i koristi stub za dohvaćanje podataka.

Na kraju ćemo napraviti provjeru rezultata:

```
$this->assertEquals(['id' => 1, 'name' => 'John Doe'], $result);
```

Ovo provjerava da li metoda `getUser` vraća očekivani rezultat.

Stubovi omogućavaju kontrolirano i predvidljivo testiranje, eliminirajući zavisnosti od stvarnih baza podataka ili drugih vanjskih servisa. To omogućava fokusiranje na testiranje logike unutar metode ili klase koju testirate.

*Primjer Unit testova**Podešavanje okoline za testove*

Potrebno je postaviti okruženje za testiranje i napraviti nekoliko unit testova za testiranje zadane klase u datoteci `ProductManager.php`:

```
<?php

use Exception;

class ProductManager
{
    private $products = [];

    /**
     * Adds a product to the inventory.
     *
     * @param string $id Unique identifier for the product
     * @param string $name Name of the product
     * @param float $price Price of the product
     */
    public function addProduct($id, $name, $price)
    {
        if (!isset($this->products[$id])) {
            throw new Exception("Product with this ID already exists.");
        }

        $this->products[$id] = [
            'name' => $name,
            'price' => $price
        ];
    }

    /**
     * Updates an existing product.
     *
     * @param string $id Product identifier
     * @param string $name New name of the product
     * @param float $price New price of the product
     */
    public function updateProduct($id, $name, $price)
    {
        if (!isset($this->products[$id])) {
            throw new Exception("Product not found.");
        }
    }
}
```

```
        $this->products[$id] = [
            'name' => $name,
            'price' => $price
        ];
    }

/**
 * Removes a product from the inventory.
 *
 * @param string $id Product identifier
 */
public function removeProduct($id)
{
    if (!isset($this->products[$id])) {
        throw new Exception("Product not found.");
    }

    unset($this->products[$id]);
}

/**
 * Retrieves a product by its ID.
 *
 * @param string $id Product identifier
 * @return array|null
 */
public function getProduct($id)
{
    if (!isset($this->products[$id])) {
        return null;
    }

    return $this->products[$id];
}

/**
 * Returns all products.
 *
 * @return array
 */
public function getAllProducts()
{
    return $this->products;
```

```
    }  
}
```

Ova PHP klasa `ProductManager` definira osnovnu logiku za upravljanje proizvodima unutar inventara. Koristi se za dodavanje, ažuriranje, brisanje i dohvaćanje proizvoda tj. osnovne CRUD operacije (Create, Read, Update, Delete) za upravljanje proizvodima u inventaru. `use Exception` omogućava korištenje PHP-ove ugrađene klase `Exception` za rukovanje greškama. Ja sam svoj direktorij nazvao `UnitTests` i pozicionirao se sa Git Bash-om. Prvi korak je instalirati composer sa `composer init`. `private array $products = []` definira privatno svojstvo koje pohranjuje proizvode kao asocijativnu matricu, gdje je ključ jedinstveni identifikator proizvoda `id`, a vrijednost je matrica sa nazivom i cijenom proizvoda.

Metoda `addProduct` dodaje novi proizvod u inventar. Provjerava postoji li već proizvod s istim ID-em. Ako postoji, baca `Exception`. Dodaje proizvod u matricu `$products` s `id`, `name`, i `price`.

Metoda `updateProduct` ažurira postojeći proizvod. Provjerava postoji li proizvod s danim ID-em. Ako ne postoji, baca `Exception`. Ažurira ime i cijenu proizvoda.

Metoda `removeProduct` Uklanja proizvod iz inventara. Provjerava postoji li proizvod s danim ID-em. Ako ne postoji, baca `Exception`. Uklanja proizvod iz niza `$products`.

Metoda `getProduct` dohvaća proizvod po ID-u. Ako proizvod ne postoji, vraća `null`. Ako postoji, vraća informacije o proizvodu.

Metoda `getAllProducts` vraća sve proizvode. Vraća matricu svih proizvoda iz `$products`.

```
Welcome to the Composer config generator

This command will guide you through creating your composer.json config.

Package name (<vendor>/<name>) [administrator/unit-tests]:
Description []:
Author [Branko Celap <branko.celap@gmail.com>, n to skip]:
Minimum Stability []:
Package Type (e.g. library, project, metapackage, composer-plugin) []: project
License []:

Define your dependencies.

Would you like to define your dependencies (require) interactively [yes]?
Search for a package: phpunit/phpunit

Found 13 packages matching phpunit/phpunit

[0] phpunit/phpunit
[1] phpunit/phpunit-mock-objects Abandoned. No replacement was suggested.
[2] phpunit/phpunit-selenium
[3] ergebnis/phpunit-slow-test-detector
[4] ergebnis/data-provider
[5] phpunit/phpunit-dom-assertions
[6] phpunit/phpunit-story Abandoned. Use behat/behat instead.
[7] phpunit/phpunit-skeleton-generator Abandoned. No replacement was suggested.
[8] phpunit/phpunit-mink-trait Abandoned. No replacement was suggested.
[9] teqnears/phpunit Stopwatch
[10] localheinz/phpunit-framework-constraint Abandoned. Use ergebnis/phpunit-f
ramework-constraint instead.
[11] ergebnis/phpunit-framework-constraint Abandoned. No replacement was suggested.
[12] phpunit/phpunit-tideways-listener Abandoned. No replacement was suggested.

Enter package # to add, or the complete package name if it is not listed: 0
Enter the version constraint to require (or leave blank to use the latest version):
A connection timeout was encountered. If you intend to run Composer without connecting to the internet, run the command again prefixed with COMPOSER_DISABLE_N
ETWORK=1 to make Composer run in offline mode.
https://repo.packagist.org could not be fully loaded (curl error 28 while downlo
ading https://repo.packagist.org/p2/phpunit/phpunit.json: Resolving timed out af
ter 10005 milliseconds), package information was loaded from the local cache and
may be out of date
Using version ^11.2 for phpunit/phpunit
```

```
Using version ^11.2 for phpunit/phpunit
Search for a package:
Would you like to define your dev dependencies (require-dev) interactively [yes]?
?
Search for a package:
Add PSR-4 autoload mapping? Maps namespace "Administrator\UnitTests" to the entered relative path. [src/, n to skip]:
{
    "name": "administrator/unit-tests",
    "type": "project",
    "require": {
        "phpunit/phpunit": "^11.2"
    },
    "autoload": {
        "psr-4": {
            "Administrator\\UnitTests\\": "src/"
        }
    },
    "authors": [
        {
            "name": "Branko Celap",
            "email": "branko.celap@gmail.com"
        }
    ]
}
Do you confirm generation [yes]? yes
Would you like to install dependencies now [yes]? |
```

Nakon ovoga phpunit se instalira tj. instaliraju se različite ovisnosti.

U `composer.json` imam sljedeću konfiguraciju:

```
{
    "name": "administrator/unit-tests",
    "type": "project",
    "require": {
        "phpunit/phpunit": "^11.2"
    },
    "autoload": {
        "psr-4": {
            "Administrator\\UnitTests\\": "src/"
        }
    },
    "authors": [
        {
            "name": "Branko Ćelap",
            "email": "branko.celap@gmail.com"
        }
    ]
}
```

phpunit možemo pokrenuti sa `./vendor/bin/phpunit`.

Sljedeći korak je dodati direktorij `tests`. Ako nemamo puno testova, ne moramo ih razgranati. Ako trebamo automatizaciju trebamo napraviti `phpunit.xml` u root-u projekta. Konfiguraciju pokupimo iz dokumentacije, [Organizing Tests->Composing a Test Suite Using XML Configuration](#):

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/|version|/phpunit.xsd"
        bootstrap="tests/bootstrap.php">
    <testsuites>
        <testsuite name="unit">
            <directory>tests/unit</directory>
        </testsuite>

        <testsuite name="integration">
            <directory>tests/integration</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Ako nam što ne treba, to izbacimo.

U ovom zadatku ne trebamo `bootstrap.php` niti `integration` direktorij jer ćemo samo praviti `unit` test. `unit` će biti u `tests` direktoriju.

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="https://schema.phpunit.de/|version|/phpunit.xsd">
    <testsuites>
        <testsuite name="unit">
            <directory>tests</directory>
        </testsuite>
    </testsuites>
</phpunit>
```

Podešavanje namespace i probnog testa

Ovdje smo dobili `ProductManager.php` kojeg ćemo postaviti u `src` direktorij. U direktoriju `tests` kojeg smo otvorili napraviti ćemo `ProductManagerTests.php`. Počet ćemo ga odmah pisati:

```
<?php

use PHPUnit\Framework\TestCase;

class ProductManagerTest extends TestCase
{
```

```

private $productManager;

protected function setup(): void
{
    $this->productManager = new ProductManager();
}

}

```

Ovo je struktura koja nam treba svaki put kada počinjemo pisati.

Podešavamo `productManager` da bi ga mogli koristiti. Pitanje da li će potencijalno biti problem taj `ProductManager`. Kada smo u prošlom primjeru pisali `RouterTest.php` morali smo ureći `use App\Services\Router`. Kada se vratimo u naš kod `ProductManagerTest.php`, Visual Studio Code nas ne obavljaštava da ne može pronaći `ProductManager()` klasu. On vidi gdje se klasa nalazi (kliknemo na Ctrl i lijevom tipkom miša na klasu). Pitanje je kako to radi bez `namespace`?

Ako pokrenemo `./vendor/bin/phpunit --debug` vidjet ćemo da nema testova.

Napisat ćemo neki bazni test (unutar klase `ProductManagerTest`) koji ćemo poslije ukloniti:

```

public function testBase()
{
    $this->assertTrue(true);
}

```

Ovime hardkodiramo `true` jer nema provjere da li je `assertTrue` uslov ispravan. Ispravan je sigurno.

Želimo debugirati `setup()`. Želimo vidjeti da li nas Visual Studio Codevara da će biti dostupan `ProductManager()` ili je u pravu. Ako je u pravu, želimo znati zašto je u pravu.

Ponovit ćemo `./vendor/bin/phpunit --debug` vidjet da klasa „`ProductManager`“ nije pronađena. Dojavio je grešku iako VSC je javljaо drugačije. Testovi ne rade jer `ProductManager` nema namespace. Također trebamo vidjeti kako je na autoloaderu (composer.json) definiran taj namespace. Kod mene je to `Administrator\\UnitTests\\` a kod predavača `Tkescec\\Vjezba\\`.

To znači da u `ProductManager.php` trebamo staviti `namespace Administrator\UnitTests`. Sada da se ne bi bunili `Exception`-i niže (pocrvenili su), potrebno je dodati `use Exception`. Vidimo da sada `Exception` nije crven. Sada je crven `ProductManager()`! To je zato što `ProductManager()` nije imao namespace. To znači da sada u `ProductManagerTest.php` moramo reći `use Administrator\UnitTests\ProductManager`. Ako sada pustimo test, vidjet ćemo da je `setUp` sada dobar. Možemo nastaviti pisati test.

```

<?php

use PHPUnit\Framework\TestCase;
use Administrator\UnitTests\ProductManager;

class ProductManagerTest extends TestCase

```

```
{
    private $productManager;

    protected function setUp(): void
    {
        $this->productManager = new ProductManager();
    }

    public function testBase()
    {
        $this->assertTrue(true);
    }
}
```

Pojedinačni (Unit) testovi primjera

Evo prva dva testa:

```

public function testGetAllProducts()
{
    $this->productManager->addProduct('A1', 'Product 1', 10.0);
    $this->productManager->addProduct('A2', 'Product 2', 20.0);
    $products = $this->productManager->getAllProducts();

    $this->assertCount(2, $products);
    $this->assertArrayHasKey('A1', $products);
    $this->assertArrayHasKey('A2', $products);
}

public function testAddProduct()
{
    $this->productManager->addProduct('A1', 'Product 1', 10.0);

    $this->assertArrayHasKey('A1', $this->productManager->getAllProducts());
    $this->assertIsArray($this->productManager->getProduct('A1'));

}

```

U metodi `testGetAllProducts()` dodat ćemo dva proizvoda s ID-ovima `'A1'` i `'A2'`. Dohvatit ćemo sve proizvode s `getAllProducts` da dobije sve elemente matrice tj. proizvode. Trebalo bih biti dva jer smo dva ubacili.

Provjere:

`$this->assertCount(2, $products)`: Provjerava da matrica `$products` ima točno 2 elementa.

`$this->assertArrayHasKey('A1', $products)`: Provjerava da matrica `$products` sadrži ključ `'A1'`.

`$this->assertArrayHasKey('A2', $products)`: Provjerava da matrica `$products` sadrži ključ 'A2'.

Metoda `testAddProduct()` dodaje proizvod s ID-om 'A1'.

Provjere:

`$this->assertArrayHasKey('A1', $this->productManager->getAllProducts())`: Provjerava da matrica sa svim proizvodima sadrži ključ 'A1'.

`$this->assertIsArray($this->productManager->getProduct('A1'))`: Provjerava da metoda `getProduct('A1')` vraća matricu (što znači da proizvod postoji) sa ključem i vrijednošću.

Svi unit testovi međusobno testiraju više metoda i tako pokrivaju više slučajeva. Tako smo sigurni da metode rade. Npr. u `testGetAllProducts()` i `testAddProduct()` smo pokrili gotovo sve metode.

Napravimo još `testAddProductWithExistingId()`:

```
public function testAddProductWithExistingId()
{
    $this->expectException(Exception::class);
    $this->expectExceptionMessage('Product with this ID already exists.');

    $this->productManager->addProduct('A1', 'Product 1', 10.0);
    $this->productManager->addProduct('A1', 'Product 2', 20.0);
}
```

Red `$this->expectException(Exception::class)` govori PHPUnit-u da očekuje da se tokom izvršavanja testa baci izuzetak tipa `Exception`. Ako izuzetak ne bude bačen, test će biti označen kao neuspješan.

Red `$this->expectExceptionMessage('Product with this ID already exists.')` postavlja očekivanu poruku izuzetka. PHPUnit će proveriti da li poruka izuzetka koji je bačen odgovara ovoj poruci. Ako izuzetak ima drugačiju poruku, test će biti označen kao neuspješan.

Red `$this->productManager->addProduct('A1', 'Product 1', 10.0)` dodaje prvi proizvod sa ID-jem 'A1', imenom 'Product 1', i cijenom 10.0. Očekujemo da se ovaj proizvod uspješno dodaje u sistem.

Red `$this->productManager->addProduct('A1', 'Product 2', 20.0)` pokušava dodati drugi proizvod sa istim ID-jem 'A1', ali sa drugačijim imenom i cijenom. U ovom trenutku očekujemo da će se desiti problem, jer ID 'A1' već postoji u sistemu, i sistem bi trebao da baci izuzetak.

Pokrenut ćemo testove još jednom:

```
$ ./vendor/bin/phpunit
PHPUnit 11.2.8 by Sebastian Bergmann and contributors.

Runtime:      PHP 8.2.12
Configuration: C:\xampp\htdocs\Algebra\NapredniPHP\UnitTests\phpunit.xml

...
3 / 3 (100%)

Time: 00:00.050, Memory: 8.00 MB

OK (3 tests, 7 assertions)

Administrator@DESKTOP-4J5DF41 MINGW64 /c/xampp/htdocs/Algebra/NapredniPHP/UnitTe
sts
$ |
```

Kontejneri

Kontejneri (engl. containers) u kontekstu softverskog inženjerstva i razvoja softvera obično se odnose na **kontejnerizaciju** ili **kontejnere**, kao što su Docker kontejneri. Evo ključnih pojmljiva i koncepta vezanih za kontejnere:

1. **Docker:** Popularna platforma za kontejnerizaciju koja omogućava pakiranje, distribuciju i izvršavanje aplikacija unutar kontejnera. Kontejneri su izolirani okružaji koji uključuju aplikacijski kod, zavisnosti i konfiguraciju potrebnu za pokretanje aplikacije.
2. **Kontejnerizacija:** Tehnologija koja omogućava pakiranje aplikacija i njihovih zavisnosti u lagan, prenosiv kontejner koji može raditi na bilo kojoj platformi koja podržava kontejnerizaciju.
3. **Kontejnerska tehnologija:** Omogućava automatsko upravljanje infrastrukturom, skaliranje aplikacija i olakšava implementaciju mikroservisne arhitekture.
4. **Kubernetes:** Orkestracijski alat otvorenog koda koji se koristi za upravljanje i automatsko održavanje aplikacija koje se izvode u kontejnerima.
5. **Prednosti kontejnera:** Ubrzan razvoj, dosljedno okruženje za razvoj, testiranje i proizvodnju, izolacija aplikacija, smanjenje resursa potrebnih za virtualizaciju.

Kubernetes i **OpenShift** su dva popularna alata koji se koriste za upravljanje kontejneriziranim aplikacijama.

Kubernetes

Kubernetes je open-source sistem za automatsko upravljanje aplikacijama koje se izvršavaju u kontejnerima. Ključne karakteristike uključuju:

1. Orkestracija kontejnera: Kubernetes omogućava automatsko raspoređivanje, skaliranje i upravljanje aplikacijama koje su pakirane u kontejnerima.

2. Declarative configuration: Konfiguracija se definiše kao YAML datoteke koje opisuju željeno stanje aplikacije, što olakšava upravljanje aplikacijama kroz njihov životni ciklus.
3. Servisi i networking: Kubernetes automatski upravlja mrežnim komunikacijama između kontejnera i omogućava definisanje servisa koji izlažu aplikacije prema van.
4. Skaliranje: Mogućnost horizontalnog i vertikalnog skaliranja aplikacija prema zahtjevima.

OpenShift

OpenShift je enterprise-ready platforma za kontejnerizaciju koja je bazirana na Kubernetes-u, razvijena od strane Red Hat-a. Glavne karakteristike uključuju:

1. Dodatne komponente: OpenShift dodaje dodatne komponente i alate za upravljanje, sigurnost, automatsko razmještanje aplikacija i integraciju s drugim alatima.
2. Build, deploy, manage: Omogućava kompletne CI/CD (Continuous Integration/Continuous Deployment) pipeline unutar platforme.
3. Developer friendly: Pruža jednostavne alate za razvoj i upravljanje aplikacija, podržava razne programerske jezike i tehnologije.
4. Multi-cloud: Mogućnost upravljanja aplikacija na više različitih cloud platformi.

Razlike između Kubernetes-a i OpenShift-a

- OpenShift je platforma bazirana na Kubernetes-u: OpenShift dodaje dodatne funkcionalnosti i alate, dok je Kubernetes više generički sistem za orkestraciju kontejnera.
- Upravljanje: OpenShift nudi dodatne mogućnosti za upravljanje i sigurnost, dok Kubernetes pruža osnovni set funkcionalnosti koje se mogu proširiti prema potrebama.
- Korporativna primjena: OpenShift je više usmjeren na korporativno okruženje, dok Kubernetes može biti korišten u raznim scenarijima, uključujući i enterprise.

- Podrška i integracija: OpenShift pruža integraciju s različitim alatima i platformama iz Red Hat ekosistema, dok Kubernetes ima širu podršku i veću zajednicu.

Ukratko, Kubernetes je moćan sistem za orkestraciju kontejnera, dok je OpenShift platforma koja dodaje dodatne alate i funkcionalnosti za enterprise upotrebu. Odabir između ova dva alata zavisiće od specifičnih zahtjeva vašeg projekta, potreba za podrškom, sigurnošću i integracijama s postojećim infrastrukturnim komponentama.

Želimo napraviti novu aplikaciju od postojećeg primjera. Pogledat ćemo što nam je potrebno a što ne.

Otvorit ćemo direktorij Test i nakon toga otići u Git Bash. Pokrenut ćemo `git init` ili kopirati `composer.json` ako nam to odgovara. Otvorimo Visual Studio Code. src direktorij je spremam. Promijenit ćemo

```
„psr-4“: {  
    „App\\“: „src/“  
}
```

Otvorimo poddirektorije: `config`, `public`, `routes`, `src`, `tests` i `vendor` koji postoji.

Da bi smo dobili sve zahtjeve u istoj točki. Na primjer, ne želimo raditi s routerom. U `public` otvorimo `index.php`. Kasnije se `public` expose-a ili možemo XAMPP natjera da gleda u tu mapu. Ako odemo na <http://localhost:8081/Algebra/NapredniPHP/Test/public> i pokrenut će se index.php. Napravit ćemo index.php.

Iskopirat ćemo u `config` poddirektorij sadržaj app.php pod istim imenom. Prebacit ćemo i `Services`: `Database.php`.