Objektno relaciono mapiranje

Object Relational Mapping

Šta je ORM?

- eng. *Object Relational Mapping*
- Mehanizam koji omogućava rad sa klasama i objektima na aplikativnom nivou, uz automatski rad sa bazom podataka
  - tzv. POPO: *Plain Old Php Object*
- Skup klasa/funkcija koje generišu odgovarajući SQL kod
  - dodavanje redova
  - stvaranje veza
  - dovlačenje podataka iz DB (prosti i složeni upiti)
  - modifikovanje baze
- Obično, 1 tabela -> 1 klasa (entitet)
- Entiteti, domenske klase = klase koje ORM perzistira (održava) u bazi podataka

Zašto ORM?

- Principi razvoja:
  - DRY: *Don't Repeat Yourself*
    - Bez ORM: kod sadrži dosta ponavljanja (otvaranje konekcije ka bazi, formiranje upita, preskakanje vrednosti koje se ugrađuju u upit, dohvatanje rezultata,...)
  - *Don't Reinvent The Wheel*
    - Može se pasti u iskušenje da se piše kod koji je „čist“, bez ponavljanja i možda objektno orijentisan
    - To već postoji, testirano je i korišćeno.
      Vreme upotrebiti za razvoj aplikacije.

Zašto ORM?

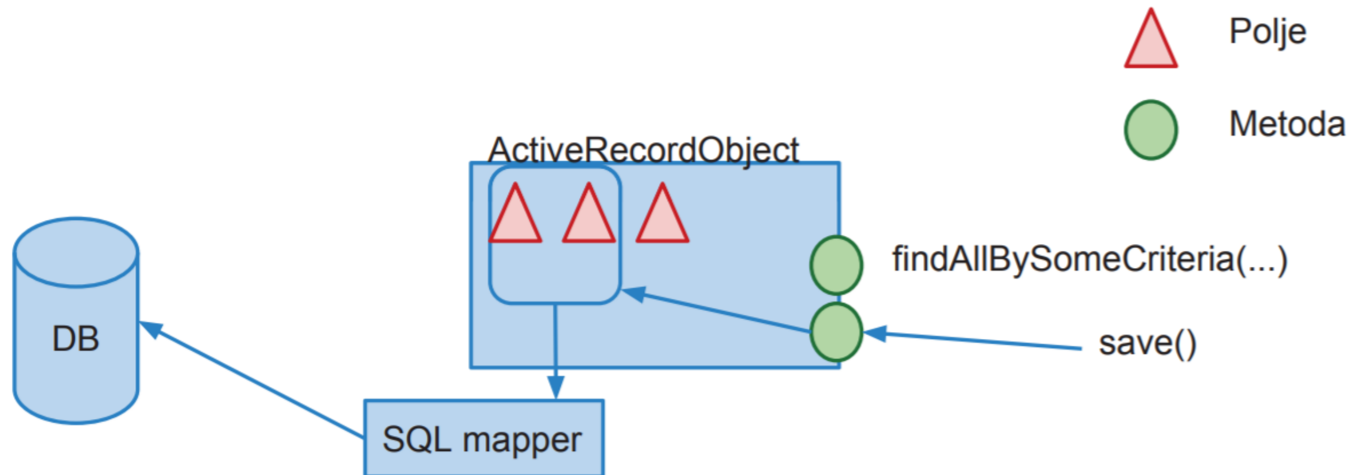
- Piše se na visokom nivou apstrakcije
- Radi se sa objektima
- Moguće lakše testiranje koda poslovne logike (nezavisno od baze podataka)
- Moguća laka promena baze (npr. MySQL na PgSQL). ORM prelazak na drugu bazu sređuje sam!
- Nema SQL stringova rasutih po čitavom kodu
  - jedna od loših praksi, široko raširenih u PHP programiranju

Zašto ORM?

- Zaštita od SQL napada (*SQL injection*)
  - umesto mysqli::real_escape_string() ili PDO::quote(), koji se mora pozivati za svaki parametar koji želimo da ugradimo u upit
    - npr. da nam neko ne bi ubacio DROP DB_name i sve obrisao
- Zato što je to standard u industriji
  - Svaka tehnologija ima ORM radne okvire: Java, Python, Ruby (on Rails), Microsoft .Net,...
  - I za jednu tehnologiju postoji dosta različitih rešenja
  - Veb rešenja mahom koriste neki ORM, danas se retko radi bez toga
- Pitanje kompleksnosti i zahtevnosti (memorija, brzina) više nisu problem

ORM varijanta: *Active Record*

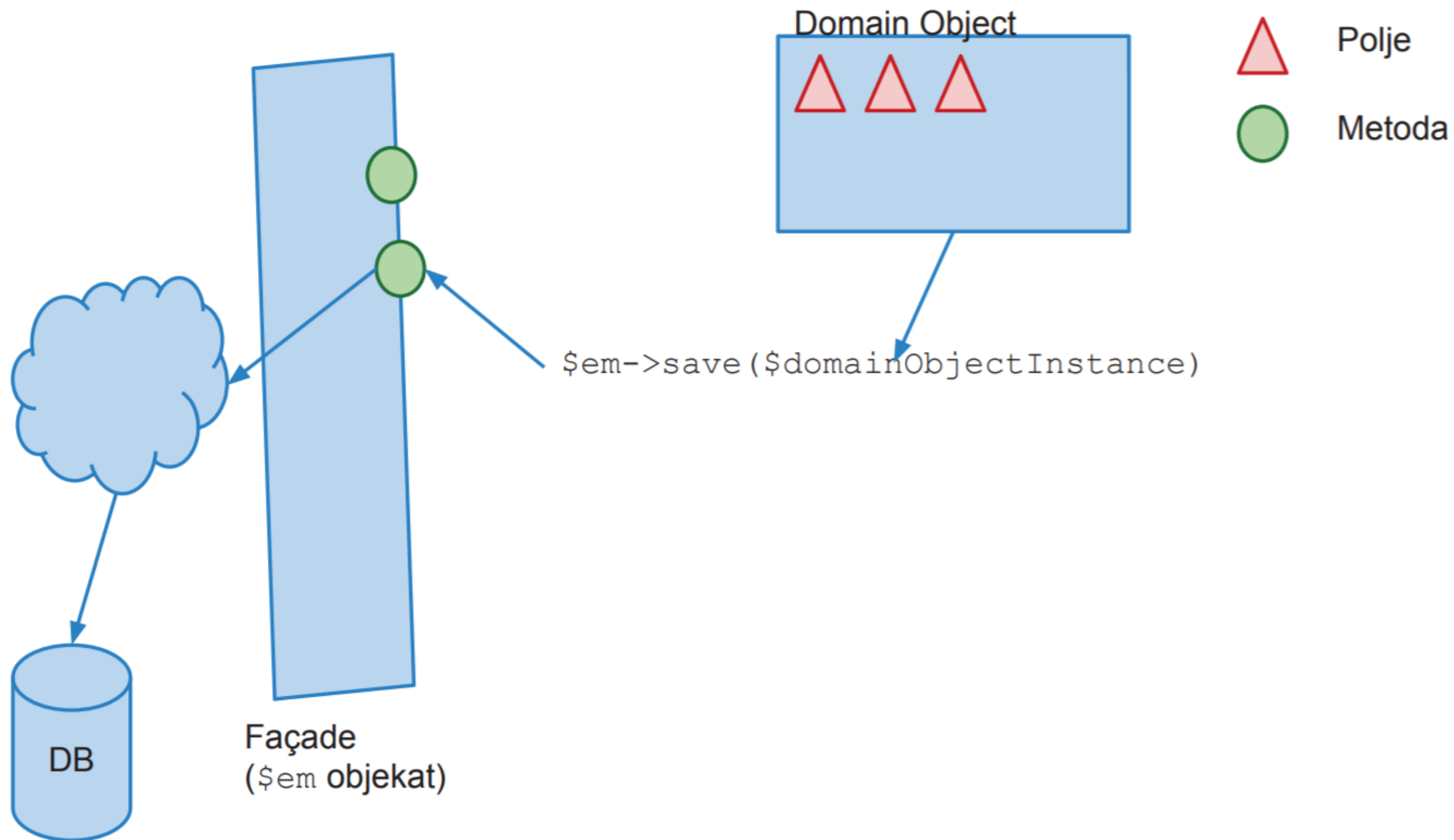
- Aktivni zapis (eng. *Active Record*)
- Objekat koji sam pruža metode, čijim se pozivanjem izmene nad njim perzistiraju (šalju u bazu i time čine trajnim): save, delete,...
- Nudi metode koje rade perzistiranje određenih polja - nije POPO, svestan je da služi za perzistiranje



ORM varijanta: *Data Mapper*

- Takođe jako popularan
- Može biti osnova za *Active Record* (da stoji u pozadini)
- Skup klasa, dat jednom fasadom (*Entity Manager*) koja na osnovu objekata (entiteta) koji se predaju izvodi operacije nad bazom
- Domenski objekti su POPO - samo objekti koji čuvaju podatke; ne znaju za postojanje baze, niti da su perzistirani (*database agnostic*)

ORM varijanta: Data Mapper



ORM varijante - pristup

- Pišemo domenske klase, generišemo bazu
  - U domenskim klasama, kroz anotacije, definišemo kako želimo da izgleda baza, alat analizira klase, generiše DDL
- Pravimo bazu, generišemo klase
  - alat posmatra bazu, i odatle generiše odgovarajuće klase
- Pišemo XML/YAML konfiguracione fajlove, generišemo i bazu i klase
  - kroz konfiguracioni fajl specificiramo izgled baze, alat generiše DDL i generiše domenske klase
- Kako izgleda baza:
  - šta želimo da imamo u entitetima (nazivi polja u klasama - kolona u tabelama, tabela, tipovi...)
  - kako su entiteti međusobno povezani (npr. firma referencira svoje zaposlene)

DOCTRINE 2

Doctrine 2

- Projekat sa skupom PHP biblioteka za skladištenje podataka i objektno mapiranje
- http://www.doctrine-project.org
- *Data mapper* implementacija
- Trenutna verzija 2.7.2 (sledeće: 2.8 i 3.0)

- Instalacija:
  - korišćenjem *Composer* alata dodaje se *Doctrine* u projekat
    - `composer require doctrine/orm`
  - Preuzeti sa GIT repozitorijuma

Doctrine 2

- Konfiguracija:
  - Kreira se biblioteka *Doctrine*
    - U okviru nje se specifiraju parametri konekcije, koji način mapiranja se koristi, definiše se putanja do entiteta i proxy-a i kreira se *EntityManager*
    - Dohvataju se parametri iz konfiguracionih fajlova i prosleđuju *Doctrine*-u
  - Definiše se *Doctrine* biblioteka kao deljeni servis
    - Omogućava kreiranje deljene instance biblioteke
  - Kreira se CLI konfiguracioni fajl
    - Omogućava pozivanje *Doctine* alata iz komandne linije
    - U okviru njega se kreira instanca *EntityManager*-a i *Doctrine CommandLine HelperSet*-a koju su neophodni za ispravan rad *Doctrine* alata

Doctrine alat

- Komande se izvršavaju pozivanjem *batch* fajla na sledeći način:
  - vendor/bin/doctrine.bat
- Sve komande se potrebno pozivati iz *root* direktorijuma

- Lista svih komandi se nalazi u dokumentaciji
  - https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/tools.html

Doctrine komande (1)

- `list`
  - Prikazuje listu svih komandi
- `orm:schema-tool:create`
  - Kreiranje tabela u bazi podataka iz entiteta
  - Pozivanjem komande sa opcijom --dump-sql, ne kreiraju se tabele već se samo prikazuje generisani kod
- `orm:schema-tool:drop`
  - Brisanje tabela iz baze podataka

Doctrine komande (2)

- `orm:convert:mapping --force --from-database --namespace="App\Models\Entities\" annotation .`
  - Kreiranje entiteta iz baze podataka.
  - Generiše ispravno oko 70-80 % koda, ali potrebno je izmeniti deo koda. Generisane anotacije nisu potpune kod asocijacija.
  - Ovom komandom je moguće kreirati entitete i iz XML ili YAML fajla.
  - Pozivanjem komande sa opcijom --force, sadržaje postojećih fajlova se brišu i generišu se novi fajlovi.

- orm:generate:entities --generate-annotations="true" .
  - Generiše *getter* i *setter* metode i anotacije za postojeće klase.
- orm:generate:proxies
  - Generiše *proxy* klase za sve entitete.

Anotacije

- Kroz dokumentacione komentare (phpDoc)
  - Dokumentacioni komentar započinje sa dve zvezdice
    - Pomaže IDE-u da prikaže namenu funkcije, očekivane tipove parametara, povratne vrednosti
  - Anotacije imaju prefiks @
- To su metapodaci: podaci o podacima
- Doctrine analizira tekst klasa, izvlači anotacije, i koristi ih za pravljenje DDL izraza
- Sve što se želi "poručiti" Doctrine, može se učiniti preko anotacija
  - To se može učiniti i preko XML i YAML fajlova
- Lista svih anotacija se nalazi u dokumentaciji
  - https://www.doctrine-project.org/projects/doctrine-orm/en/2.7/reference/annotations-reference.html

Anotacije

- Anotacije polja:
  - `@ORM\Column(type="tip")`
    - Mapira se polje u klasi na kolonu u tabeli.
    - Može se definisati vrednost atributa: name, nullable,…
  - `@ORM\Id`
  - `@ORM\GeneratedValue(strategy="strategija")`

- Anotacije klasa:
  - @ORM\Entity
    - Klasa se posmatra kao entitet.
    - Može se definisati vrednost atributa *repositoryClass*.
  - @ORM\Table(name=”naziv_tabele")
    - Definiše se na koju tabelu se mapira data klasa.

Anotacije - Primer

```
/**
 * @ORM\Table(name="vest”)
 * @ORM\Entity
 */
class Vest
{
    /**
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="IDENTITY")
     */
    private $id;
    /**
     * @ORM\Column(name="naslov", type="string",
                    length=50, nullable=false)
     */
    private $naslov;
}
```

Anotacija (→ `@ORM\Column`)

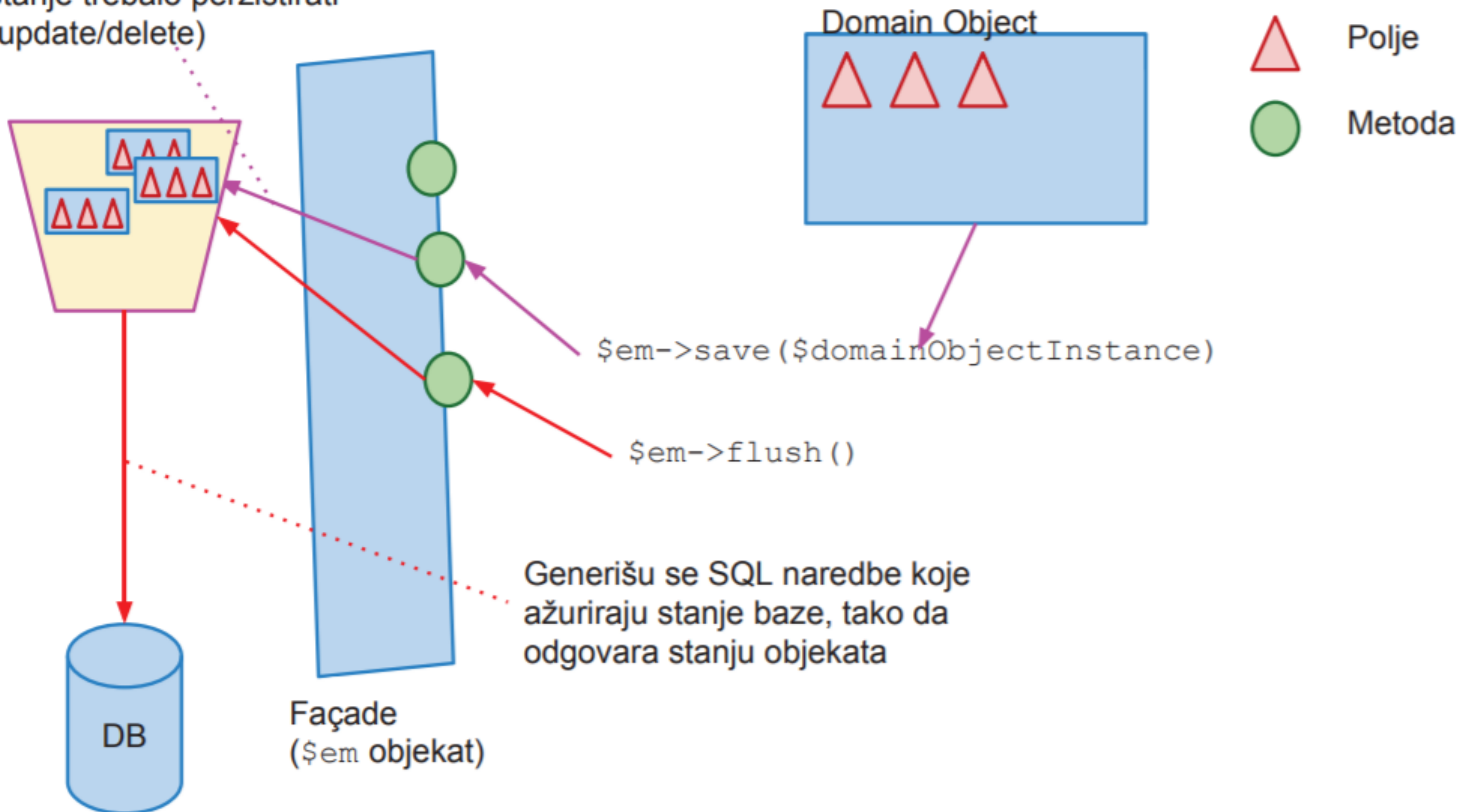
Parametrizovana anotacija (→ `name="id", type="integer"`)

ENTITY MANAGER I RAD SA OBJEKTIMA

Entity Manager (1)

- Radi transformaciju entiteta u zapise u bazi, i obratno.
- Izvršava upite i naredbe koje kreiraju, ažuriraju i uklanjaju zapise iz baze, tako da se odraze promene načinjene nad objektima.
- Funkcioniše tako što sve *Insert*, *Update* i *Delete* naredbe koje mora da izvrši skuplja, tj. ne izvršava ih istog momenta jednu po jednu.
- Izvrši ih paketski, sve „skupljene“ u okviru jedne transakcije
  - Ako se desi problem (npr. referencijalni integritet) transakcija vraća podatke u prethodno stanje (*rollback*), i ništa iz tog paketa ne ostane izvršeno.

Entity Manager i Unit Of Work



Entity Manager (2)

- *Doctrine* biblioteka kreira *Entity manager* i čuva ga kao polje klase
- Unutar kontrolera *Doctrine* biblioteka se dohvata kao servis
  - `$doctrine = \Config\Services::doctrine();`

- Pozivanjem `$em->persist()` kome je prosleđen novi objekat anotiran sa `@Entity`, *Entity Manager* sačuva informaciju da treba da kreira novi objekat
- Pozivanjem `$em->flush()` izvršavanju se sve *Insert*, *Update* i *Delete* akcije nad bazom koje bi trebalo da sačuvaju tekuće stanje perzistiranih objekata
  - Ako ne pozovemo `$em->flush()`, izmene neće biti sačuvane u bazi podataka

Entity Manager - Primer 1

```
$nick = new Person();
$nick->setName('Nikola Tesla');

$mick = new Person();
$mick->setName('Mihajlo Pupin');

$jo = new Person();
$jo->setName('Jovan Cvijic');

$entityManager->persist($nick);
$entityManager->persist($mick);

$entityManager->flush();

$entityManager->persist($jo);
```

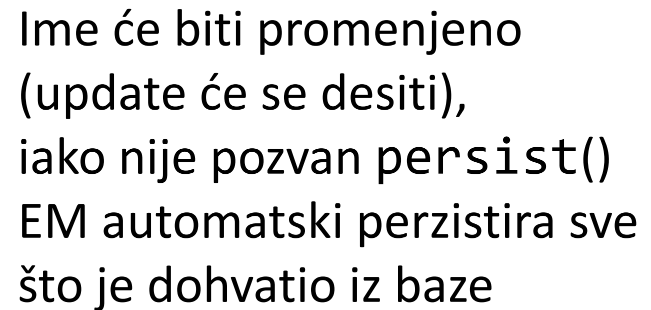
Kreirani su novi objekti (ne potiču iz baze)
EM ne zna za njihovo postojanje sve dok se objekti ne perzistiraju

EM saznaje za postojanje objekata

Biće sačuvani samo `$nick` i `$mick`, jer su dostavljeni EM-u pre `flush`().
`$jo` će biti sačuvan u bazi, ako se u kodu koji dolazi, javi poziv `flush`() metode

Entity Manager - Primer 2

```
$p1 = $entityManager->find("Person", 1);
$p1->setName("Novo ime");
$entityManager->flush();
```



Entity Manager (3)

- *Entity Manager* sadrži skup svih objekata koji su dovučeni iz baze podataka.
- Kada se radi $flush$, proveravaju se svi objekti koji su doneti iz baze
- Ako je neki izmenjen (*dirty*), *Entity Manager* generiše *Update* naredbu

- Koristi se uzorak *Identity map*
- Kada zahtevamo entitete sa istim identifikatorom, na više različitih mesta u kodu, dobijamo reference ka istom objektu!

ASOCIJACIJE

Referisanje

- Moguće je da se referiše jedan objekat iz drugog, a da u bazi podataka postoji strani ključ koji to oslikava
  - Radi se samo sa objektima i kolekcijama objekata
- Tipovi veza
  - 1-1,
  - 1-*,
  - *-1,
  - *-*
- Vlasnik veze (*owner*)
- Strane u asocijaciji
  - *Mapped by*
  - *Inverted by*

Tipovi veza

- **Unidirekcione** - samo u jednoj klasi postoji referenca ka objektu druge
- **Bidirekcione** - u obe klase postoje reference ka drugoj strani

- **Multiplikativnost** - sa koliko objekata na suprotnom kraju je posmatrani entitet u vezi

Veza - primer

- Autor je glavni autor više vesti, pa je sa strane `Autor` to veza tipa 1-*, tj. *OneToMany*.

- Vest ima samo jednog glavnog autora, dok takvih vesti može biti više, pa je sa strane `Vest` to veza tipa * -1, tj. *ManyToOne*.

Definisanje asocijacija

- Putem anotacija
  - `@ManyToOne(targetEntity="...",inversedBy="...")`
  - `@OneToMany(targetEntity="...", mappedBy="...")`
  - `@ManyToMany(targetEntity="...", inversedBy="..")`

- Atributi `mappedBy` i `inversedBy` u anotacijama moraju sadržati naziv polja u `targetEntity`, koji sadrži referencu ka objektu posmatranog entiteta.

Primer veze 1-*

```
class Vest
{
    ...
    /**
     * @ORM\ManyToOne(targetEntity="Autor", inversedBy="mojeVesti")
     * @ORM\JoinColumns({
     *   @ORM\JoinColumn(name="autor", referencedColumnName="korime")
     * })
     */
    private $autor;
    ...
}

class Autor
{
    ...
   /**
    * @ORM\OneToMany(targetEntity="Vest", mappedBy="autor",
                orphanRemoval=true)
    */
    private $mojeVesti;
    ...
}
```

Primer veze *-*

```
class Vest
{
    ...
    /**
     * @ORM\ManyToMany(targetEntity="Autor", inversedBy="koautorVesti")
     * @ORM\JoinTable(name="koautor_vesti",
     *   joinColumns={ @ORM\JoinColumn(name="id_vest",
                referencedColumnName="id") },
     *   inverseJoinColumns={ @ORM\JoinColumn(name="koautor",
                referencedColumnName="korime") }
     * )
     */
    private $koautori;
}

class Autor
{
    ...
   /**
    * @ORM\ManyToMany(targetEntity="Vest", mappedBy="koautori")
    */
    private $koautorVesti;
}
```

Programsko rukovanje asocijacijama

- Zavisi od ORM radnog okvira
  - Neki sve srede iza kulisa (npr. *Hibernate*, *JPA*,...)
  - Kod nekih drugih dosta toga mora ručno (npr. *Doctrine*)
    - *Doctrine*: Pri stvaranju asocijacije kod bidirekcione veze, ručno se mora definisati strana koja referiše kod unidirekcione veze.

- *Lazy loading*
  - Objekti koji su u asocijaciji se dovlače po potrebi (pri prvom pristupu)
  - Podrazumevano se koristi ovakav pristup, ali je moguće podesiti i da se objekti u asocijaciji dovuku odmah
  - Implementira se pomoću uzorka *Proxy*
    - Moguće je da u nekim situacijama dođe do N+1 fetch problema.

Vlasnik (*owner*) asocijacije (1)

- Vlasnik asocijacije
- *Doctrine*: Čuvanje izmena kod vlasnika asocijacije dovodi do perzistiranja u izmenama asocijacije (dodavanje nove veze, uklonjene postojeće,...)
  - U slučaju da se samo inverzna (ne-vlasnik) strana izmeni, asocijacija neće biti prezistirana.

- Nasuprot vlasnikuu nalazi se inverzna strana asocijacije
- Kod vlasnika, atributom inversedBy se specificira polje preko kog inverzna strana referiše svog vlasnika (ili više njih, zavisno od kardinalnosti).

Vlasnik (*owner*) asocijacije (2)

- Kod 1-* i *-1 veze, vlasnik može biti samo jedna strana
  - Vlasnik je uvek * strana, jer se kod nje nalazi strani ključ

- Kod *-* veze, obe strane mogu biti vlasnici
  - Tada se logički bira koja strana je odgovorna za rukovanje asocijacijom
    - Da li je vest odgovorna da čuva asocijacije sa osobama koje su koautori ili je osoba odgovorna za to?
  - Svejedno je, jer će se asocijacija perzistirati kad god se odabrani vlasnik izmeni, međutim logički gledano nije svejedno.

Veze kod Doctrine (1)

- Neophodno ručno postaviti sve reference objektima, a *Doctrine* će to smestiti kako treba u bazu.
- Primer:

```
Class Autor {
…
    public function addMojeVesti($mojaVest)
    {
          if (!$this->mojeVesti->contains($mojaVest)) {
              $this->mojeVesti[] = $mojaVest;
              $mojaVest->setAutor($this);
          }
          return $this;
    }
}

class Vest{
    ...
    public function setAutor($autor = null)
   {
          $this->autor = $autor;
          return $this;
    }
}
```

Dodaje se vest u listu autorovih vesti i poziva se izvršavanje metode i sa *owner* strane

Postavlja se primarni autor – *owner* referenca sređena

Veze kod Doctrine (2)

- Realizacija dovlačenja objekta koji su u asocijaciji po potrebi realizovana je pomoću uzorka *Proxy*
  - Sa svaku *Entity* klasu generiše se po jedna *Proxy* klasa koja predstavlja omotač oko postojeće *Entity* klase
  - Prilikom upravljanja asocijacijama radi se sa objektima *Proxy* klase, međutim to programeru nije vidljivo
  - U okviru definisane *Doctrine* biblioteke, podešava se putanja i *namespace* na kojoj će se nalaziti *Proxy* klase
  - *Proxy* klase se automatski generisu na osnovu *Entity* klasa
    - Pozivom alata iz komandne linije se generišu sve *Proxy* klase
      - `orm:generate:proxies`
    - Prilikom izvršavanja aplikacije u razvojnom (*Dev*) režimu, *Proxy* klase se generišu po potrebi

Veze kod Doctrine (3)

- Može se podesiti kaskadno ponašanje prilikom rada sa poljima koja učestvuju u formiranju asocijacije
  - podrazumevano je isključeno;
  - Tada nije neophodno ručno podešavati sadržaj kolekcija.
  - Ipak se entiteti dovlače u memoriju, pa može doći do veće upotrebe resursa.
  - Primer:

```
@ORM\OneToMany(targetEntity="Vest",
mappedBy="autor",cascade={"persist", "remove"})
```

- Može se podesiti da ukoliko se izbace elementi iz dovučenog niza u memoriju, da se on automatski obriše i iz baze podataka
  - podrazumevano je isključeno;
  - Primer:

```
@ORM\OneToMany(targetEntity="Vest",
mappedBy="autor",orphanRemoval=true)
```

UPITI KOD ORM

Repository (1)

- Omogućava razdvajanje logike upita od modela
- Uzorak *Repository*
- Svaki entitet ima svoj *Repository* koji sadrži predefinisane upite

- Primer:

```
$em->getEntityManager()
  ->getRepository('Vest')->findAll();
$em->getEntityManager()
  ->getRepository(Vest::class)
  -> findOneBy(array('naslov'=>$naslov));
```

Repository (2)

- Podrazumevani *Repository* se može proširiti dodatnim metodama
  - Potrebno je da klasa proširuje *EntityRepository*
  - Dodati anotaciju *Entity* klasi, kako bi znala da ne treba da koristi podrazumevani *Repository*, već zadati
    - `@ORM\Entity(repositoryClass="VestRepository")`

DQL upiti (1)

- ORM obično „donosi“ svoju varijantu upitnog jezika
  - HQL, JPQL,…
- DQL - *Doctrine Query Language*
- U upitu figurišu domenske klase i koriste se njihova polja
- Može se ići „dublje“ u pristupu poljima

- Primer:

```
$dql = "SELECT v, a FROM Vest v join v.autor a";
$query = $entityManager->createQuery($dql);
$query->setMaxResults(30);
$books = $query->getResult();
```

DQL upiti (2)

- Imenovani parametri
  - Definišu se sa : pre naziva parametra
  - Primer:

```
$dql = "SELECT v, a FROM Vest v join v.autor a
  where v.datum>:pocetak and v.datum<:kraj";
$results = $entityManager->createQuery($dql)
  ->setParameter('pocetak', '2020-04-01')
  ->setParameter('kraj', '2020-05-01’)->getResult();
```

- Pozicioni parametri
  - Definišu se sa ? pre rednog broja parametra
  - Primer:

```
$dql = "SELECT v, a FROM Vest v join v.autor a
  where v.datum>?1 and v.datum<?2";
$results = $entityManager->createQuery($dql)
  ->setParameter(1,'2020-04-01')
  ->setParameter(2,'2020-05-01’)->getResult();
```

DQL upiti (3)

- *Join* predstavlja jednostavan način da se postave upiti koji vraćaju jednu od strana koje učestvuju u asocijaciji
- *Join* zahteva da se podatak nalazi i u drugoj tabeli,
  dok korišćenjem *left join*-a mogu se dohvatati i podaci koji se ne nalaze u drugoj strani asocijacije

- Primer:

```
$dql = "SELECT v FROM Vest v
  join v.autor
  left join v.koautori";
```

Query Builder (1)

- *Query Builder* objekat se kreira, a zatim se njemu „dodaju“ delovi upita
  - `$qb=$em->createQueryBuilder()`
- Uzorak *Builder* (Graditelj)
- Pruža način da se SQL konstrukcije dodaju, pozivom odgovarajućih metoda
- Upit se samo konstruiše programski; izvršavanje se pokreće eksplicitno
- Moguće formirati kompleksne upite, sa opcionim delovima, na osnovu onoga što korisnik unosi na formi, koristeći *if* strukturu i uslovnim pozivanjem metoda
  - sa DQL bi se konkatenacija stringova obavljala uslovno, što je loše i naporno

Query Builder (2)

- Primer:

```
$qb=$em->createQueryBuiIder();
$qb->select('a')
    ->from('Autor' ,'a')
    ->where('a.korime like :korime') ;
if ($admin) {
    $qb->andWhere('admin=1');
}
$qb->orderBy( 'u.korime' );
$qb->setParameter(korime ,'%'.$korime.'%');
$result=$qb->getQuery()->getResult();
```

Query Builder - Expr class

- Delovi upita kao što su poređenja i slično mogu se ostvariti programski, koristeći Expr klase
  - U potpunosti se izbegava pisanje DQL/SQL koda
- Primer:

```
$qb->select ('a')->from ('Autor','a');
$qb->where($qb->expr()->orx(
    $qb->expr()->like('u.ime',':pretraga'),
    $qb->expr()->like('u.prezime',':pretraga')
);
```

Criteria expression

- Za dohvatanje entiteta mogu se koristiti i *Criteria* upiti

- Primer:

```
$criteria = Criteria::create();
$criteria
   ->orWhere(Criteria::expr()
          ->contains('ime',$pretraga))
   ->orWhere(Criteria::expr()
          ->contains('prezime',$pretraga));
$vesti= $em->gtRepository(Autor::class)
    ->matching($criteria);
```