LAHDEN AMMATTIKORKEAKOULU
Lahti University of Applied Sciences

OLIO-RELAATIO-MALLINNUS WWW-PALVELUSSA

LAHDEN
AMMATTIKORKEAKOULU
Tekniikan ala
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Opinnäytetyö
Kevät 2012
Antti Korhonen

Lahden ammattikorkeakoulu
Tietotekniikan koulutusohjelma

KORHONEN, ANTTI: Olio-relaatio-mallinnus WWW-palvelussa

Ohjelmistotekniikan opinnäytetyö, 49 sivua

Kevät 2012

TIIVISTELMÄ

Opinnäytetyö tehtiin ohjelmistotalo Korpimedia Oy:lle. Opinnäytetyön kohteena oli toteuttaa tietokannan olio-relaatio-mallinnus PHP-kielellä toteutettuun avainasiakasjärjestelmään. Mallinnus toteutettiin käyttämällä Doctrine ORM -kirjastoa. Ohjelmointiympäristönä käytössä oli Netbeans-ohjelma ja WAMP-palvelinkokonaisuus.

Doctrine toimii tietokanta-abstraktiokerroksen avulla. Doctrine on rakennettu Active Record, Data Mapper- ja Meta Data Mapping -suunnittelumallien päälle. Keskeisessä asemassa on Doctrine_Record-luokka, josta peritään tietokannan tauluja vastaavat malliluokat.

Doctrine voidaan asentaa palvelimelle käyttäen joko Subversion-versionhallintaa tai PEAR-asennusohjelmaa tai lataamalla Doctrinen PEAR-paketti. Toimiakseen Doctrine vaatii PHP:n version 5.2.3 tai uudemman.

Doctrineen on mahdollista konfiguroida esimerkiksi yhteyden toiminta käyttämällä Doctrine_Manager::setAttribute-metodia. Doctrine_Connection-luokka toimii tietokantayhteyden säilövänä luokkana. Sen avulla voidaan luoda ja tuhota tietoja tietokannasta. Tietokannan mallintavat malliluokat, jotka voidaan luoda joko malligeneraattorilla tai manuaalisesti. Mallit ladataan joko niin, että malli ladataan vain tarvittaessa, tai niin, että kaikki mallit ladataan muistiin. Käytettävä lataustapa voidaan konfiguroida. Malleihin tietokannan sarakkeet määritellään setTableDefinition-metodissa käyttäen DoctrineRecord::hasColumn-metodia. Metodissa määritellään sarakkeen nimi, tietotyyppi ja pituus. Suhteet määritellään setUp-metodissa käyttäen joko hasOne- tai hasMany-metodeja.

DQL, eli Doctrine Query Language, on oliopohjainen kyselykieli. DQL:n avulla voidaan tehdä select-, update- tai delete-kyselyitä. Kysely luodaan käyttäen Doctrine_Query::create-metodia. Doctrinella on myös mahdollista validoida tietueet ja sivuttaa suuri joukko tietueita.

Avainasikasjärjestelmässä käytettiin hyväksi automaattista mallienluontia, Doctrine_Record-malliluokkia tiedon käsittelyyn, Doctrine_Query-olioita tiedon hakemiseen ja Doctrine_Pager-oliota tietuejoukkojen sivutukseen. Parannettavaa sovelluksessa olisi ollut transaktioiden käyttö ja sivutuksen osittaminen. Sovellus on koekäytössä, ja sen kehitystyö jatkuu.

Asiasanat: tietokannat, relaatiotietokannat, mallit, tietomallit, kyselykielet, PHP, DQL, Doctrine

Lahti University of Applied Sciences
Degree Programme in Information Technology

KORHONEN, ANTTI: Object relational mapping in a web application

Bachelor's Thesis in software engineering 49 pages

Spring 2012

ABSTRACT

This Bachelor's Thesis deals with mapping a database as relational objects in a PHP-application meant for key customers of a company. The mapping was implemented using the Doctrine ORM library. The development environment consisted of a Netbeans IDE program and WAMP-server configuration.

The Doctrine ORM library works above a database abstraction layer. Doctrine is based on Active Record, Data Mapper and Meta Data Mapping design patterns. A central class in Doctrine is the Doctrine_Record class, which is responsible for the tables of a database.

Doctrine can be installed using either Subversion or with a PEAR-package or PEAR installer. In order to work, Doctrine requires PHP version 5.2.3 or newer.

It is possible to configure Doctrine using the Doctrine_Manager::setAttribute method. This can be used e.g. to configure database connections. Connections are encapsulated in the Doctrine_Connection class. It can be used for creating and deleting records from the database. The model classes used for modeling the database can be generated either automatically or manually. Database columns are defined in the setTableDefinition method using the Doctrine_Record::hasColumn method. This method takes in the column name, datatype and length as parameters. Relations are defined in the setUp method using the hasOne and hasMany methods.

Doctrine has an object oriented query language called DQL, i.e. Doctrine Query Language. With DQL it is possible to execute select, update or delete queries. Queries are created using the Doctrine_Query::create method. It is possible to validate records with Doctrine or divide a large collection of records into pages.

When creating the application for Esa Print, automatic model generation was used for models and Doctrine_Record models were used in handing data objects and Doctrine_Query objects were used to query records from the Database. Doctrine_Pager object was used to display a large number of records in separate pages. Possible improvement would have been using transactions and partitioning paging into sets of pages. The application is in its trial stage and the development of the application continues.

Key words: databases, relational databases, models, data models, query languages, PHP, DQL, Doctrine

SISÄLLYS

1 JOHDANTO

Tietokantaa käytetään ohjelmistosovelluksissa persistenttien tietojen ja tilojen tallentamiseen luotettavasti istuntojen välissä. Jotta tietoja voidaan käyttää hyväksi oliopohjaisissa sovelluksissa, täytyy tavalla tai toisella yhdistää tieto sovellusalueen logiikasta tietokannan tauluihin. Tämä onnistuu käyttämällä jotain saatavilla olevista ORM-työkaluista (Object-Relational-Mapper). Jotta kyseisistä työkaluista saadaan kaikki irti, täytyy kuitenkin niiden toiminta tuntea hyvin. (Miller 2009.)

Opinnäytetyö tehtiin lahtelaiselle Korpimedia Oy:lle, joka on digimediatoimisto ja ohjelmistotalo, joka perustettiin vuonna 2010. Korpimedia tuottaa verkkomediaa, räätälöityjä sovelluksia ja graafista suunnittelua (Korpimedia Oy 2012). Yrityksen liikevaihto vuonna 2010 oli 199 000 euroa ja yrityksessä työskenteli neljä työntekijää (Fonecta Oy 2011).

Opinnäytetyön kohteena ollut avainasiakasjärjestelmä-ohjelmisto toimitettiin Esa Print Oy:lle, joka on myös lahtelainen yritys. Esa Print on painotalo, jolla on lähes satavuotinen kokemus painoalalta. Esa Print panostaa ympäristöystävällisyyteen. (Esa Print 2012.)

Ohjelmointiympäristössä käytössä oli Netbeans-ohjelma. Ohjelmaan asennettiin lisäosaksi PHP-kehitystyökalut, jotka tarjosivat koodimalleja, refaktorointityökalut, parametrivihjeet ja älykkään koodin täydennöksen (Netbeans 2012). Sivusto toimi kehitysympäristössä WAMP-palvelinkokonaisuuden avulla. Lyhenne WAMP tulee sanoista Windows-Apache-MySQL-PHP, eli siis palvelinympäristö Windows-käyttöjärjestelmässä, jossa käytetään WWW-palvelimena Apache-ohjelmaa, tietokantajärjestelmänä MySQL-ohjelmaa ja skriptikielenä PHP-kieltä (Webopedia 2012). Työ aloitettiin kesällä 2011.

Työn tavoitteena oli toteuttaa avainasiakasjärjestelmään tietokannan olio-relaatio-mallinnus PHP-ohjelmointikielellä. Avainasiakasjärjestelmä oli rakennettu Zend-ohjelmistokehyksen päälle. Zend-ohjelmistokehys on MVC-arkkitehtuuriin

pohjautuva PHP-ohjelmistokehys (Vaswani 2010, 3). Tutkimusongelmana oli miten toteuttaa yksinkertaisesti ja ylläpidettävästi olio-relaatio-mallinnus kyseiseen avainasikasjärjestelmään.

Työssä keskityttiin tietokannan olio-relaatio-mallinnukseen Doctrine ORM -kirjaston avulla. Työstä rajattiin pois mallinnuksen näkökulma Zend-ohjelmistokehyksen kannalta ja muiden ORM-työkalujen käsittely, koska yritykselle oli jo tehty opinnäytetyö Zend-ohjelmistokehyksestä ja Doctrine ORM on yrityksessä käytössä useassa ohjelmistoprojektissa.

Luvussa kaksi käsitellään Doctrinen käyttöönoton vaatimukset, konfigurointi, malliluokkien luonti ja käyttö sekä Doctrinen oma kyselykieli. Luvussa kaksi selitetään myös, mitä olio-suhde-mallinnus on ja miten ORM-työkalut liittyvät mallinnukseen. Kolmannessa luvussa kerrotaan, kuinka Doctrinen avulla toteutettiin tietokannan olio-suhde-mallinnus sekä mitä ja miten Doctrinen ominaisuuksia käytettiin hyväksi.

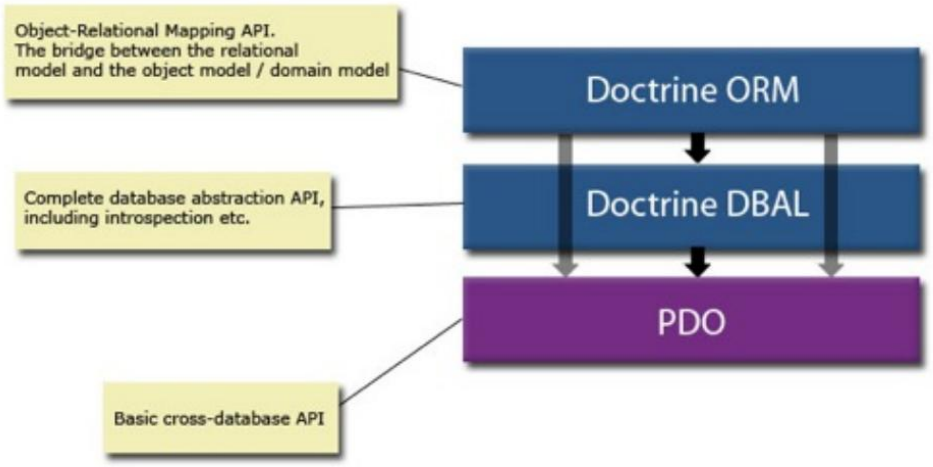
2 DOCTRINE

2.1 Doctrine ja ORM

Olio-relaatio-mallinnus on tekniikka, jota käytetään yhdistämään relaatiotietokannan tietotyypit ohjelmalogiikan tietotyyppeihin. Tämän ansiosta ohjelmoijalla on käytössään virtuaalinen oliotietokanta, jota voidaan käyttää hyväksi ohjelman suorituksessa. (Blanco, Borschel, Vesterinen & Wage 2010, 27.)

Doctrine on ORM-työkalu (Object Relational Mapper), eli olio-relaatio-mallintaja, joka on rakennettu toimimaan PHP-version 5.2.3 tai uudemman kanssa. Se käyttää hyväkseen Doctrinen tietokanta-abstraktiokerrosta , joka taas käyttää hyväkseen PDO:ta (PHP Data Objects) (KUVIO 1). Doctrine sisältää oman kaupallisen olio-orientoituneen SQL-kielen – DQL:n – kyselyjen rakentamiseen (Doctrine Query Language). (Blanco ym. 2010, 27.)

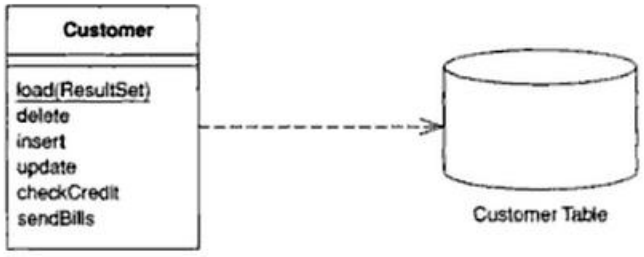
Tietokannan funktiokutsujen abstrahointiin Doctrine käyttää PDO:ta (PHP Data Objects) (Blanco ym. 2010, 28). Versiosta 5.1 lähtien PHP:n mukana on tullut tietokantojen käsittelyä varten oliopohjainen PDO, joka yhtenäistää PHP:n tukemien tietokantamoottorien rajapinnat (Ohjelmointiputka 2009).



KUVIO 1. Doctrinen rakenne (Blanco ym. 2010, 29)

Doctrine ORM on suurimmilta osin rakennettu Active Record, Data Mapper ja Meta Data Mapping -suunnittelumallien pohjalta. Tyypillinen Active Record -rajapinta, eli tallennus, poisto ja muut toiminnot, saadaan laajentamalla tiettyä Doctrine_Record-nimistä kantaluokkaa. Valtaosa työstä kuitenkin uudelleen ohjataan muille luokille, kuten Doctrine_Table-luokalle. Tämä luokka toimii Data Mapper -rajapinnan metodeille kuten esimerkiksi createQuery(), find(id), findAll() ja findBy(). Tämä tarkoittaa, että ActiveRecord-kantaluokan avulla Doctrinen on mahdollista hallinnoida tietueita, kun taas kartoittaminen toteutetaan muualla. (Blanco ym. 2010, 29–30.)

Active Record -suunnittelumallissa jokainen Active Record -luokka on vastuussa datan tallennuksesta ja lataamisesta sekä kaikesta sovellusalueen logiikasta, joka koskee dataa. Active Recordin -rakenteen tulee vastata tietokantaa, eli luokan attribuutti vastaa tietokannan saraketta (KUVIO 2). (Fowler 2003, 35.)



KUVIO 2. ActiveRecord-luokan luokkakaavio ja suhde tietokannan tauluun (Fowler 2003, 35)

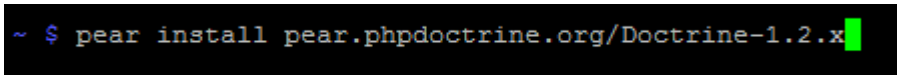
2.2 Käyttöönotto

2.2.1 Asennus ja vaatimukset

Ennen Doctrinen käyttöönottoa tarvitsee tarkistaa, että Doctrine toimii onnistuneesti palvelimella. Tämä pystytään tarkistamaan tekemällä pieni PHP-skripti nimeltään phpinfo.php ja kopioimalla se palvelimelle sijaintiin, josta sen voi avata verkkoselaimella. Skripti koostuu yhdestä rivistä – phpinfo(); – kun mukaan ei lasketa PHP:n aloittavaa ja lopettavaa tägiä. Suoritettaessa skripti verkkoselaimella tulee ruutuun näkyviin tietoa palvelimen PHP-konfiguraatiosta.

Tiedoista tulee näkyä, että PHP-versio on 5.2.3 tai uudempi ja PDO-ajurit ovat asennettuna. (Blanco ym. 2010, 35–36.)

Doctrinen voi asentaa Subversion-versionhallintaa tai PEAR-asennusohjelmaa käyttäen tai lataamalla Doctrinen PEAR-paketti. Blanco ym. suosittelevat käyttämään Subversion-versionhallintaa Doctrinen lataamiseen, sillä tämä tekee Doctrinen päivittämisestä helppoa. Doctrinen eri versiot voi ladata Doctrine-projektin SVN-palvelimelta: http://svn.doctrine-project.org. Doctrine-projekti tarjoaa PEAR-palvelimen Doctrinen asennusta ja päivittämistä varten. PEAR-paketin asennus onnistuu komentoriviltä (KUVIO 3). Jos ei ole mahdollista käyttää PEAR-asennusohjelmaa, paketin voi ladata myös Doctrine-projektin sivuilta. (Blanco ym. 2010, 36–39.)



KUVIO 3. Käsky Doctrinen PEAR-paketin asentamiseen (Fowler 2003, 38)

Jotta Doctrine toimisi, sovellus tarvitsee Doctrine.php-tiedoston, joka sisältää ydinluokan. Doctrinen esilataamiseksi tarvitaan bootstrap.php-tiedosto (KUVIO 4). Tiedostossa tehdään myös Doctrinen konfigurointi. Doctrine.php-tiedoston lataus tehdään PHP:n require_once-määreellä. Tarve on myös rekisteröidä luokkien automaattiseen lataamiseen tarkoitettu autoloader-funktio. Luodessa Doctrine-olioista ilmentymiä autoloader-funktio lataa automaattisesti luokan nimen perusteella oikean PHP-tiedoston. Tämä nopeuttaa skriptin suoritusta, kun kaikkia luokkia ei tarvitse ladata etukäteen, vaan ainoastaan tarvittavat luokat ladataan. Lopuksi luodaan singleton-tyyppinen Doctrine_Manager-instanssi Doctrinen hallinnoimista varten. (Blanco ym. 2010, 39–41.)

```
2
3
4  // bootstrap.php
5  /**
6    * Bootstrap Doctrine.php, register autoloader specify
7    * configuration attributes and load models.
8    */
9
10 require_once(dirname(__FILE__) . '/lib/vendor/doctrine/Doctrine.php');
11 spl_autoload_register(array('Doctrine', 'autoload'));
12 $manager = Doctrine_Manager::getInstance();
13
14
15
```

KUVIO 4. Koodiesimerkki Doctrinen esilataamisesta (Blanco ym. 2010, 42)

2.2.2 Yhteyksien avaaminen

Uuden tietokantayhteyden avaaminen käy helposti alustamalla uusi PDO-olio (KUVIO 5). Jotta Doctrinella olisi käytettävissä yhteyden salasana ja käyttäjäntunnus tietokantojen luontia ja tuhoamista varten, asetetaan manuaalisesti $conn-oliolle username- ja password-optiot, kuten kuviossa 5 riveillä 9 ja 10. Tietokantayhteyden avaus suoritetaan aina vain tarvittaessa. Tämä siksi, että sivusto, joka käyttää Doctrinea, ei tarvitse aina tietokantaa ja tietokantayhteyden avaus on raskas operaatio. Jos ei haluta käyttää PDO-oliota, on myös yksinkertaisempi tapa avata tietokantayhteys Doctrinelle. Tämä toteutetaan antamalla Doctrine_Manager::connection-metodille parametrina validi osoite tietokantaan kuten: 'mysql://username:password@localhost/test'. (Blanco ym. 2010, 48–49.)

```
1
2  // bootstrap.php
3  // ...
4  $dsn = 'mysql:dbname=testdb;host=127.0.0.1';
5  $user = 'dbuser';
6  $password = 'dbpass';
7  $dbh = new PDO($dsn, $user, $password);
8  $conn = Doctrine_Manager::connection($dbh);
9  $conn->setOption('username', $user);
10 $conn->setOption('password', $password);
11
```

KUVIO 5. Koodiesimerkki yhteyden muodostamisesta (Blanco ym. 2010, 48–49)

2.3 Konfigurointi

Doctrinen ominaisuuksien ja toiminnallisuuden konfigurointi onnistuu käyttämällä attribuutteja. Konfiguraatiot ovat kolmessa portaassa tärkeysjärjestyksessä: globaali-, yhteys- ja taulukohtaiset konfiguraatiot. Tämä tarkoittaa sitä, että jos sama attribuutti on asetettu ylemmällä ja alemmalla tasolla, ylemmän tason arvo on aina käytössä. (Blanco ym. 2010, 53.)

Attribuutit asetetaan setAttribute-metodilla, joka tarvitsee kaksi parametria. Ensimmäinen parametri kertoo, mitä attribuuttia ollaan käsittelemässä ja toinen parametri kertoo kyseisen attribuutin arvon (KUVIO 6). Kun setAttribute-metodia kutsutaan Doctrinen manager-oliosta, vaikuttaa attribuutti globaalilla tasolla ja sillä on siis vaikutus kaikkiin yhteyksiin ja kaikkiin tauluihin. Kun metodia kutsutaan connection-oliolle, vaikuttaa attribuutti kyseiseen yhteyteen ja kaikkiin kyseisen yhteyden tietokannan tauluihin. Kun taas metodia kutsutaan table-oliolle, vaikuttaa attribuutti vain kyseiseen tauluun. (Blanco ym. 2010, 53–54.)

```
1
2  // bootstrap.php
3  // ...
4  $manager->setAttribute(Doctrine_Core::ATTR_VALIDATE,
5                         Doctrine_Core::VALIDATE_ALL);
6
```

KUVIO 6. Validointinnin konfigurointi (Blanco ym. 2010, 54)

Eri tietokannanhallintajärjestelmät toimivat eri tavoin. Esimerkiksi jotkin tietokannat pakottavat nimeämään taulujen nimet niin, että nimet alkavat isolla alkukirjaimella, toiset taas niin, että kaikki taulujen nimet ovat pienellä. Tämä hankaloittaa sovelluksen siirtämistä tietokantahallintajärjestelmien välillä. Doctrinessa on joukko konfiguraatio-optioita helpottamaan siirrettävyyttä. Siirrettävyyteen liittyvät optiot ovat bittikohtaisia, joten niitä voidaan yhdistellä käyttämällä bittioperaatioita (KUVIO 7). (Blanco ym. 2010, 55.)

```
1
2  $conn->setAttribute(Doctrine_Core::ATTR_PORTABILITY,
3                      Doctrine_Core::PORTABILITY_FIX_CASE |
4                      Doctrine_Core::PORTABILITY_RTRIM);
5
```

KUVIO 7. Siirrettävyyden konfigurointi käyttäen bittioperaatiota (Blanco ym. 2010, 57)

Oletuksena Doctrine ylikirjoittaa paikalliset muutokset olioihin tapauksessa, jossa suoritetaan kysely, joka palauttaa saman olion, johon muutokset oli tehty. Täten jos esimerkiksi $user-olion username-attribuuttia on muutettu ja suoritetaan kysely, joka palauttaa kyseisen $user-olion, ylikirjoittuu kyseinen muutos. Jos halutaan, että muutokset säilyvät kyselyiden välissä, voidaan asettaa ATTR_HYDRATE_OVERWRITE-attribuutti epätodeksi, jolloin saadaan haluttu toimintatapa. (Blanco ym. 2010, 58–59.)

Doctrine antaa käyttäjän konfiguroida tavan, jolla eri tietokantaan liittyvät elementit, kuten taulut ja indeksit, nimetään. Tämä toimii kumpaankin suuntaan, eli kun luokat luodaan olemassaolevasta tietokannasta tai kun luokista luodaan tietokantataulut. Esimerkkinä ATTR_IDXNAME_FORMAT-attribuutti määrää indeksien nimeämistavan, ATTR_SEQNAME_FORMAT-attribuutti sekvenssien nimeämistavan, ATTR_TBLNAME_FORMAT-attribuutti taulujen nimeämistavan ja ATTR_DBNAME_FORMAT-attribuutti tietokantojen nimeämistavan. (Blanco ym. 2010, 61–63.)

Doctrine mahdollistaa validointiprosessin konfiguronnin. Validointi konfiguroidaan Doctrine_Core::ATTR_VALIDATE-attribuutilla. Samoin kuin siirrettävyys attribuuttien kohdalla, ATTR_VALIDATE-attribuutin arvot voidaan yhdistää bittioperaatioilla. Käyttämällä tai-operaatiota VALIDATE_LENGTHS- ja VALIDATE_TYPES-attribuuteille validoidaan sekä kenttien pituudet että kenttien tyypit. (Blanco ym. 2010, 63–64.)

2.4 Lisää yhteyksistä

Doctrine_Connection on luokka, joka toimii tietokantayhteyden säilövänä luokkana. Yhteys on yleensä instanssi PDO:sta, mutta Doctrine sallii omien PDO:n toimintaa matkivien tietokantasovittimien käytön. Doctrine_Connection-luokan vastuulla on pitää kirjaa Doctrine_Table-olioista, tietueista ja päivitettävistä tai poistettavista tietueista. Luokka tekee varsinaiset kyselyt tietokantaan, kun kyseessä on insert-, update- tai delete-tyyppinen kysely. Luokka myös ohjaa DQL-kyselyt varsinaisen tietokannan kyselyiksi. Vaihtoehtoisesti Doctrine_Connection-luokka voi käyttää transaktioiden validointiin Doctrine_Validator-luokkaa. (Blanco ym. 2010, 234.)

Doctrine on suunniteltu toimimaan useamman tietokantayhteyden kanssa. Doctrine oletuksena käyttää aina viimeisintä yhteyttä kyselyiden suorittamiseen, jos ei toisin ole konfiguroitu. Doctrine_Manager-luokalla on staattinen metodi, Doctrine_Manager::connection, joka avaa uusia yhteyksiä. Metodin ensimmäiseksi parametriksi annetaan tietokannan osoite ja toiseksi parametriksi yhteyden nimi (KUVIO 8). Jos metodille ei anneta parametreja, palauttaa se tämän hetkisen yhteyden eli viimeksi avatun yhteyden. Aktiivista yhteyttä voidaan vaihtaa kutsumalla Doctrine_Manager::setCurrentConnection-metodia. Metodi ottaa parametrina yhteyden nimen. (Blanco ym. 2010, 67–68.)

```
1
2  $conn = Doctrine_Manager::connection(
3      'mysql://testuser:testpwd@localhost/testdb',
4      'connectionName');
5
```

KUVIO 8. Yhteyden avaus käyttäen Doctrine_Manager::connection()-metodia

Yhteyksien joukon ylitse voidaan iteroida antamalla manager-olio PHP:n foreach-rakenteelle. Tämä toimii, koska Doctrine_Manager toteuttaa PHP:n IteratorAggregate rajapinnan, jota tarvitaan olioiden iterointiin. (Blanco ym. 2010, 69.)

Manager-oliolla on joukko metodeja, joilla voidaan manipuloida yhteyksien joukkoa. Yhteyden nimen saa selville kutsumalla manager-olion getConnectionName-metodia ja antamalla sille parametrina kyseinen yhteysolio. Yhteyden voi sulkea ja poistaa yhteysrekisteristä kutsumalla manager-oliolle closeConnection-metodia parametrina yhteysolio. Jos yhteys halutaan sulkea, mutta säilyttää yhteysrekisterissä, voidaan yhteysoliolle kutsua close-metodia. Doctrine_Manager::getConnections-metodi palauttaa taulukon rekisteröidyistä yhteyksistä. Tämä on vaihtoehtoinen tapa iteroida yhteydet manager-olion käytön sijasta. Doctrine_Manager-luokka toteuttaa myös Countable-rajapinnan, joten manager-olio voidaan antaa count-metodille, jolloin saadaan selville yhteyksien määrä. (Blanco ym. 2010, 69–71.)

Tietokantayhteyksen avulla on mahdollista luoda ja tuhota tietokantoja käyttäen Doctrinea. Tämä tapahtuu yksinkertaisesti käyttäen näihin toimintoihin tehtyjä metodeja Doctrine_Manager- tai Doctrine_Connection-luokista. Kutsuttaessa dropDatabases- tai createDatabases-funktioita manager-oliolle iteroidaan kaikki luodut yhteydet lävitse ja kutsutaan näille joko dropDatabase- tai createDatabase-metodia. Jos halutaan luoda tai tuhota tietyltä yhteydeltä tietokanta, kutsutaan kyseisen yhteyden createDatabase- tai dropDatabase-funktioita. (Blanco ym. 2010, 71–72.)

On mahdollista, että eteen tulee tilanne, jossa on tarve tehdä oma yhteysluokka. Esimerkkinä tarve laajentaa MySQL-rajapinnan toimintaa. Tämä on mahdollista toteuttaa tekemällä uusi Doctrine_Connection_Nimi-luokka, joka periytyy Doctrine_Connection_Common luokasta ja luokka Doctrine_Adapter_Nimi-luokka, joka implementoi Doctrine_Adapter_Interface-rajapinnan. Uusi yhteys rekisteröidään Doctrine_Manager-luokalle registerConnectionDriver-metodilla, jolloin se on openConnection-metodin käytettävissä (KUVIO 9). (Blanco ym. 2010, 72.)

```
1
2  $manager->registerConnectionDriver(
3     'nimi',
4     'Doctrine_Connection_Nimi');
5  $conn = $manager->openConnection(
6     'nimi://username:password@localhost/dbname');
7
```

KUVIO 9. Oman yhteysajurin rekisteröinti ja hyödyntäminen (Blanco ym. 2010, 72)

2.5 Mallit

Doctrine mallintaa tietokannan sisäistä tietomallia joukolla PHP-luokkia, jotka määräävät mallin rakenteen ja toiminnan. Nämä luokat periytyvät Doctrine_Record-luokasta. Jokainen lapsiluokka voi sisältää yhden setTableDefinition-metodin ja setUp-metodin. SetTableDefinition-metodia käytetään sarakkeiden, indeksien ja muun yleisen taulun rakenteeseen liittyvän informaation määrittämiseen. SetUp-metodi on tarkoitettu Doctrine-Record-lapsiluokkien välisten suhteiden määrittämiseen. (Blanco ym. 2010, 75–76.)

2.5.1 Mallien luonti

Doctrinen mukana tulee malligeneraattori. Tämä generaattori pystyy lukemaan olemassaolevaa tietokantaa ja luomaan joukon luokkia sen tauluista automaattisesti (KUVIO 10). Generaattoria käytetään PHP-skriptitiedostosta, joka suoritetaan komentoriviltä. (Vaswani 2010, 113.)

Automaattisesti luotuja luokkia käytetään esimerkiksi silloin, kun on tarve luoda olemassaolevan sovelluksen tauluista malliluokat. Automaattisesti luodut luokat eivät yleensä kuitenkaan toimi halutulla tavalla, vaan niitä täytyy modifioida tarpeeseen sopivaksi. Metodi, joka on vastuussa malliluokkien luomisesta, on Doctrine_Core-luokan generateModelsFromDb-metodi. Metodi ottaa ensimmäiseksi parametrikseen kansion nimen, jonne luodut Doctrine_Record-lapsiluokat luodaan. Toinen parametri on taulukko tietokantayhteyksien nimistä, joista mallit luodaan. Kolmas parametri on taulukko vaihtoehtoja, joita käytetään

mallien rakennuksessa. Kuviossa 10 generateTableClasses-optio tarkoittaa, että mallien lisäksi jokaiselle malliluokalle luodaan myös tyhjä Doctrine_Table-luokka, jota voidaan laajentaa. Generaattori luo, esimerkiksi author-taulusta, BaseAuthor nimisen abstraktin luokan ja tästä luokasta periytyvän Author-luokan (KUVIO 11). (Blanco ym. 2010, 76–79.)

```
1
2  Doctrine_Core::generateModelsFromDb('models',
3                                      array('doctrine'),
4                                      array('generateTableClasses' => true));
5
```

KUVIO 10. Malliluokkien luonti olemassa olevasta kannasta (Blanco ym. 2010, 77)

```
1
2  abstract class BaseAuthor extends Doctrine_Record
3  {
4      public function setTableDefinition()
5      {
6          $this->setTableName('author');
7          $this->hasColumn('lname', 'string', 255, array('type' => 'string',
8              'length' => 255, 'primary' => true, 'autoincrement' => false));
9          $this->hasColumn('fname', 'string', 255, array('type' => 'string',
10             'length' => 255, 'primary' => true, 'autoincrement' => false));
11         $this->hasColumn('byear', 'integer', 4, array('type' => 'integer',
12             'length' => 4));
13     }
14 }
15
16 class Author extends Doctrine_Record
17 {
18
19 }
20
```

KUVIO 11. Esimerkki Doctrinen generoimasta malliluokasta

Vaihtoehtoisesti luokat voidaan myös luoda kirjoittamalla luokan PHP-koodi manuaalisesti. Tällöin luokka täytyy kirjoittaa Doctrinen syntaksin mukaisesti. (Blanco ym. 2010, 83.)

2.5.2 Mallien lataus automaattisesti

Doctrine lataa mallit joko konservatiivisesti tai aggressiviisesti. Oletuksena Doctrine käyttää aggressiivista mallien latausta, joten jos tarve on käyttää

aggressiivista lataustyyppiä, niin sitä ei tarvitse asettaa erikseen. Konservatiivinen lataus ei tarvitse aluksi PHP-tiedostoa, vaan polku luokan nimeen tallennetaan ja tätä polkua käytetään kutsuttaessa Doctrine_Core::modelsAutoload-metodia. Jotta voidaan käyttää Doctrinen mallien latausta, täytyy automaattinen lataaja rekisteröidä komennolla: spl_autoload_register(array('Doctrine_Core', 'modelsAutoload')). Konservatiivinen lataustapa on hyvä vaihtoehto tuotantoympäristössä, koska tämä säästää resursseja. Tämä lataustapa vaatii kuitenkin sen, että jokainen tiedosto sisältää vain yhden luokan ja luokan nimen täytyy vastata tiedoston nimeä. Jotta voidaan käyttää konservatiivista mallien latausta, täytyy Doctrinelle konfiguroida Doctrine_Manager-luokan setAttribute-metodia käyttäen ATTR_MODEL_LOADING-attribuutti arvoon MODEL_LOADING_CONSERVATIVE (KUVIO 12). Nyt loadModels-metodia kutsuttaessa löydetyt luokat varastoidaan sisäiseen välimuistiin, jotta automaattinen lataaja voi ladata ne myöhemmin. (Blanco ym. 2010, 83–85.)

```
1
2  $manager->setAttribute(Doctrine_Core::ATTR_MODEL_LOADING,
3                  Doctrine_Core::MODEL_LOADING_CONSERVATIVE);
4
```

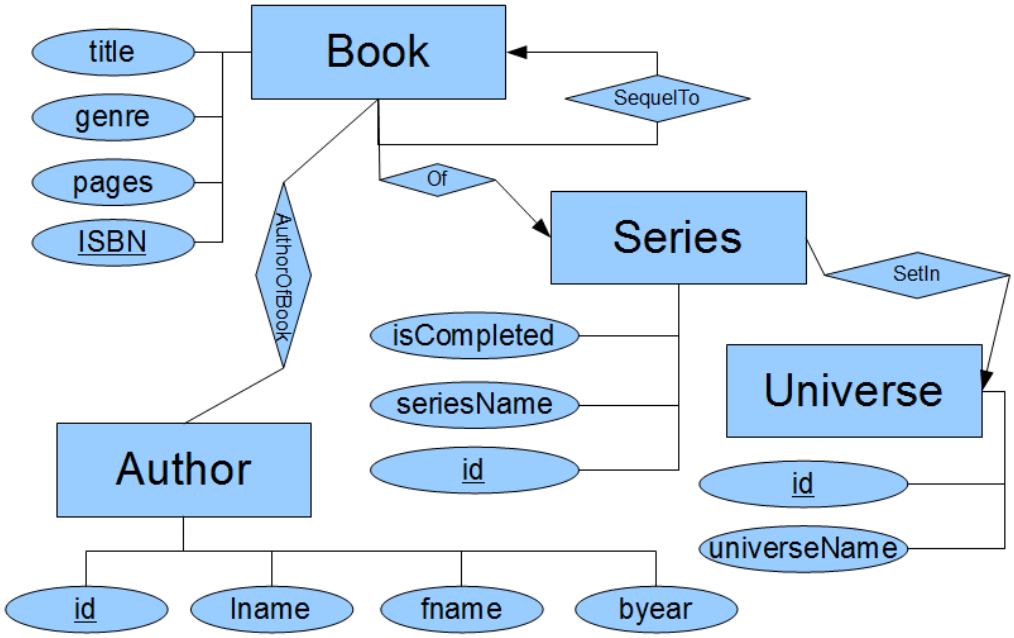
KUVIO 12. Konservatiivisen mallien lataustavan konfigurointi (Blanco ym. 2010, 84)

Aggressiivinen mallien lataus on siis oletuslataustapa. Aggressiivisessä lataustavassa etsitään kaikki tiedostot, joilla on .php-pääte, ja ladataan ne muistiin. Koska Doctrine ei ota huomioon perintää, pitää muista malleista perityillä luokilla ottaa huomioon, ettei automaattinen lataaja tiedä, missä järjestyksessä luokat kuuluu ladata. Aggressiivisessa lataustavassa on mahdollista olla useita luokkia samassa tiedostossa ja tiedoston nimen ei tarvitse vastata luokan nimeä. Huono puoli aggressiivisessa lataustavassa on se, että jokainen PHP-tiedosto latadaan jokaisella sivupyynnöllä. (Blanco ym. 2010, 84–85.)

2.5.3 Sarakkeiden määritys

Sarakkeet määritellään malliluokan setTableDefinition-metodissa. Määrittely tapahtuu kutsumalla Doctrine_Record-luokan hasColumn-metodia (KUVIO 14).

Havainnollistavana tietokantana seuraavissa osioissa käytetään kuvion 13 kuvaamaa tietokantaa. (Blanco ym. 2010, 92.)



KUVIO 13. Esimerkki tietokannan E/R-kaavio

```
1  class Author extends Doctrine_Record
2  {
3      public function setTableDefinition()
4      {
5          $this->setTableName('author');
6          $this->hasColumn('id', 'integer', 11, array(
7                   'primary' => true,
8                   'autoincrement' => true
9              )
10         );
11     }
12 }
13
```

KUVIO 14. Sarakkeen määrittely malliluokassa

Yksi ongelma, joka tulee vastaan tietokantoja käsiteltäessä, on se että useat tietokantaohjelmat eroavat toisistaan siinä, miten ne käsittelevät kenttien nimiä. Doctrine tarjoaa ongelmaan läpinäkyvän ratkaisun. Tämä tarkoittaa sitä, että jos

luodaan Doctrine_Record-luokan perivä luokka, voidaan luokassa määritellä esimerkiksi isCompleted niminen kenttä. Tähän kentään päästään käsiksi isCompleted nimisen attribuutin kautta (KUVIO 15) huolimatta siitä, onko käytössä MySQL, Postgres vai Oracle tietokanta. Doctrinessa kenttien nimien kirjainkoolla on kuitekin väliä käytettäessä kenttien nimiä kyselyssä. (Blanco ym. 2010, 87.)

```
1
2  $series = Doctrine_Core::getTable('Series')->find(1);
3  $series->isCompleted;
4  $series['isCompleted']; // samanarvoinen kuin edellä
5
```

KUVIO 15. Kaksi tapaa päästä käsiksi tietueen kenttään

Jotta tietotyypit saataisiin yksinkertaisiksi ja yhteensopiviksi kaikkien tietokantajärjestelmien kanssa, Doctrineen on sovittu joukko perustietotyyppejä. Teoriassa sovellukset voivat käyttää näitä tietotyyppejä välittämättä siitä, minkä tietokantajärjestelmän pohjalle sovellus on rakennettu. Sarakkeiden pituudet määritetään Doctrinessa kokonaislukuarvolla. Jotkin saraketyypit määrittyvät myös siirrettävyystyypin lisäksi sarakkeen pituuden mukaan. Esimerkiksi jos sarakkeen tyyppi on string ja pituus on 1000, tällöin MySQL-tietokannoissa tietotyypiksi tulee TEXT. (Blanco ym. 2010, 90–91.)

Pituusmääre tulkitaan eri tavoin riippuen sarakkeen tyypistä. Integer-tyypin tapauksessa pituus määrää, kuinka monta tavua kokonaisluku vie. String-tyypissä pituusmääre määrää, kuinka monta merkkiä merkkijonoon mahtuu. Float- ja decimal-tyyppisissä sarakkeissa pituus merkitsee merkkien määrää pois lukien desimaalipiste. Enum-tyypissä pituusmäärettä ei oteta huomioon, jos natiivi toteutus tukee enum-tyyppiä, mutta jos enum-tyyppi emuloidaan, pituus tarkoittaa merkkijonon pituutta. (Blanco ym. 2010, 88.)

Doctrinella on mahdollista antaa sarakkeille vaihtoehtoisia nimiä sekä oletusarvoja kaikille tietotyypeille. Kun oletusarvo on määritelty sarakkeelle, käytetään oletusarvoa luotaessa uusi tietue tauluun. Jos määritelmästä luodaan

tietokantaan uusi taulu, on oletusarvo mukana taulun määritelmässä. (Blanco ym. 2010, 88.)

Tietotyyppejä voidaan määrittää tarkemmin käyttäen esimerkiksi notnull-, length-, default-, unsigned- (KUVIO 16) ja fixed length -määreitä. Esimerkiksi notnull-määre asettaa tietokannan NOT NULL lipun todeksi tai epätodeksi. Jos pituusmääre jätetään pois, käyttää Doctrine tietotyypille maksimaalista pituutta, josta voi tulla ongelmia joidenkin tietotyyppien kohdalla. Paras käytäntö on asettaa aina kaikille sarakkeille pituusarvo. (Blanco ym. 2010, 91.)

```
1  class Author extends Doctrine_Record
2  {
3      public function setTableDefinition()
4      {
5          $this->setTableName('author');
6          $this->hasColumn('id', 'integer', 11, array(
7                  'primary' => true,
8                  'autoincrement' => true
9              )
10         );
11         $this->hasColumn('fname', 'string', 255, array(
12                 'notnull' => true
13             )
14         );
15         $this->hasColumn('lname', 'string', 255, array(
16                 'notnull' => true
17             )
18         );
19         $this->hasColumn('byear', 'integer', 4, array(
20                 'unsigned' => true,
21                 'fixed' => true
22             )
23         );
24     }
25 }
```

KUVIO 16. Lisämääreiden käyttö sarakkeen määrittelyssä

Boolean-tietotyyppi voi saada vain kaksi mahdollista arvoa, joko true tai false. Tietotyypin tallennustapa riippuu käytetystä tietokantajärjestelmästä. Integer- eli kokonaislukutyyppi on sama kuin PHP:n kokonaislukutyyppi. Se voi sisältää niin

suuria kokonaislukuja kuin käytettävä tietokanta pystyy tukemaan. Tämän tietotyypin kanssa voidaan käyttää unsigned-määrettä, mutta kaikki tietokantajärjestelmät eivät tue tätä ominaisuutta. Todellisen tietokannan sarakkeen tyyppi riippuu pituusmääreestä. Float-tyyppinen sarake voi säilöä liukulukuja. Tämä tietotyyppi soveltuu suurten lukujen tallentamiseen, jotka eivät tarvitse suurta tarkkuutta. Luvun maksimisuuruus ja tarkkuus riippuvat käytetystä tietokannasta. Decimal-tietotyypillä voidaan säilöä kiinteän tarkkuuden desimaalilukuja eli tämä tietotyyppi käy isoa tarkkuutta vaativien lukujen käsittelyyn. Desimaaliluvun pituus voidaan asettaa samoin kuin muissakin tietotyypeissä ja tyypille voidaan myös asettaa mittakaava. Mittakaavalla tarkoitetaan, monenko desimaalin tarkkuudella tieto talletetaan tietokantaan. Esimerkki mittakaavan määrittämisestä on esitetty kuviossa 17 rivillä 43. (Blanco ym. 2010, 91–94.)

```
32
33  class Book extends Doctrine_Record
34  {
35      public function setTableDefinition()
36      {
37          $this->setTableName('book');
38          $this->hasColumn('isbn', 'string', 14, array(
39                   'primary' => true
40              )
41          );
42          $this->hasColumn('pages', 'desimal', 5, array(
43                   'scale' => 1
44              )
45          );
46          $this->hasColumn('title', 'string', 255);
47      }
48  }
49
```

KUVIO 17. Kirjan kuvaava luokka

String- eli tekstitietotyypille voidaan määrittää pituus kahdella eri tavalla. Ensimmäisessä tavassa pituus on eksplisiittisesti määritetty arvo ja toisessa tavassa pituus on tietokannan sallima maksimipituus. Ensimmäistä tapaa suositellaan, koska se on tehokkaampi näistä kahdesta tavasta. Toinen tapa

mahdollistaa hyvin pitkät tekstikentät, mutta saattaa estää null-arvojen käytön tai kyseisen kentän lajittelun. (Blanco ym. 2010, 95.)

Array-tyyppi on sama kuin PHP:n taulukkotyyppi. Doctrine tukee myös oliotyyppisiä sarakkeita. Doctrine serialisoi automaattisesti olion, kun tietue tallennetaan, ja palauttaa olion, kun sarakkeen arvo luetaan. Sekä array- että object-tyypit serialisoidaan, kun tiedot persistoidaan tietokantaan ja puretaan takaisin taulukoksi tai olioksi, kun tietoa haetaan tietokannasta. (Blanco ym. 2010, 95–96.)

Blob- ja Clob-tietotyypit (Binary Large OBject ja Character Large OBject) on tarkoitettu määrittämättömän pituisen ja tiedon, joka on liian iso mahtumaan tekstityyppiseen tallennuspaikkaan, tallentamiseen tietokantaan. Yleensä kyseessä on esimerkiksi tiedosto. Clob-tietotyyppi on tarkoitettu säilömään dataa, joka koostuu ASCII-merkeistä, kun taas blob-tietotyyppi on tarkoitettu säilömään mielivaltaista dataa. (Blanco ym. 2010, 97.)

Doctrinesta löytyy tietysti myös aikaan liittyvät tietotyypit, kuten timestamp, time ja date. Time-tietotyyppi vastaa tiettyä kellonhetkeä. Kellonaika tallennetaan merkkijonona ISO-8601-standardin mukaisesti eli HH:MI:SS, jossa HH vastaa tunteja 00:sta 23:een ja MI ja SS vastaavat minuutteja ja sekunteja. Date-tietotyyppi vastaa kalenteripäivää. Myös date-tietotyyppi tallennetaan merkkijonona ISO-8601-standardin mukaisesti eli YYYY-MM-DD, jossa YYYY vastaa vuotta, MM vastaa kuukautta 01:stä 12:een ja DD on päivämäärä 01:stä 31:een. Timestamp-tietotyyppi on yhdiste date- ja time-tietotyypeistä. Timestamp on siis muotoa YYYY-MM-DD HH:MI:SS. (Blanco ym. 2010, 98–99.)

Enum-tietotyyppi on enumeroitu lista. Mahdolliset arvot voidaan määrittää Doctrine_Record::hasColumn-metodissa, kuten kuviossa 18 rivillä 42. Jos halutaan käyttää tietokantajärjestelmän natiivia enum-tyyppiä, täytyy ATTR_USE_NATIVE_ENUM-attribuutti asettaa todeksi. (Blanco ym. 2010, 100.)

```
32
33  class Book extends Doctrine_Record
34  {
35      public function setTableDefinition()
36      {
37          $this->setTableName('book');
38
39          //...
40
41          $this->hasColumn('genre', 'enum', null,
42              array('values' => array('scifi', 'fantasy', 'romance'))
43          );
44          $this->hasColumn('universeId', 'integer', 11, array(
45                  'notnull' => false
46              )
47          );
48      }
49  }
50
```

KUVIO 18. Book-luokkaan enumeroidun tietotyypin määrittely

2.5.4 Suhteet

Doctrinessa suhteet asetetaan joko Doctrine_Record::hasMany- tai Doctrine_Record::hasOne-metodeilla (KUVIO 20). Doctrine tukee lähes kaikkia mahdollisia tietokantarelaatioita kuten yhden suhde yhteen ja itseviittausta. Toisin kuin sarakemääritteet, Doctrine_Record::hasMany- ja Doctrine_Record::hasOne-metodeja käytetään setUp-metodissa (KUVIO 20). Metodit ottavat kaksi parametria. Ensimmäinen parametri on merkkijono, joka määrää suhteeseen kuuluvan luokan nimen. Parametrille voi antaa vaihtoehtoisen nimen, kuten kuviosta 20 riviltä 77 näkyy. SetUp- ja setTableDefinition-metodeja kutsutaan suorituksen aikana Doctrinen toimesta. (Blanco ym. 2010, 104–105.)

series

id	isCompleted	seriesName	universeId
1	TRUE	The Icewind Dale Trilogy	1

universe

id	universeName
1	Forgotten Realms

KUVIO 19. Esimerkki Series- ja Universe-tyyppisistä tietueista kannassa, joilla on yhden suhde moneen -relaatio

```
50  class Series extends Doctrine_Record
51  {
52      public function setTableDefinition()
53      {
54          $this->setTableName('series');
55          $this->hasColumn('id', 'integer', 11, array(
56                  'primary' => true,
57                  'autoincrement' => true
58              )
59          );
60          $this->hasColumn('isCompleted', 'boolean');
61          $this->hasColumn('seriesName', 'string', 255, array(
62                  'notnull' => true
63              )
64          );
65          $this->hasColumn('universeId', 'integer', 11, array(
66                  'notnull' => false
67              )
68          );
69      }
70
71      public function setUp()
72      {
73          $this->hasOne('Universe', array(
74              'local' => 'universeId',
75              'foreign' => 'id')
76          );
77          $this->hasMany('Book as Books', array(
78              'local' => 'id',
79              'foreign' => 'seriesId')
80          );
81      }
82  }
```

KUVIO 20. Suhteiden asettaminen Doctrine_Record::setUp-metodissa Series-luokalle

Yhden suhde yhteen -relaatiot ovat yksinkertaisimpia relaatioita. Yhden suhde yhteen -relaatiot toteutetaan käyttämällä hasOne-metodia. Tästä esimerkki kuviossa 20 rivillä 73. Tässä tapauksessa local-määre vastaa omistettavaan luokkaan viittaavaa parametria, kun taas foreign-määre viittaa omistettavan luokan yksilöivään parametriin. Tämä ei eroa yhden suhde moneen -relaation hasOne-määreestä. Esimerkki vastaavista tietuesta on kuviossa 19. Relaation nimeksi tulee omistavan luokan nimi. Omistavaan luokkaan tulee muuten

samanlainen määre, mutta local- ja foreign-määreet vaihtavat paikkaa ja relaation nimeksi tulee omistettavan luokan nimi. (Blanco ym. 2010, 110–112.)

```
65  class Book extends Doctrine_Record
66  {
67      public function setTableDefinition()
68      {
69          $this->setTableName('book');
70          //...
71          $this->hasColumn('seriesId', 'integer', 11, array(
72                  'notnull' => false
73              )
74          );
75          $this->hasColumn('sequelToIsbn', 'string', 14, array(
76                  'notnull' => false
77              )
78          );
79      }
80
81      public function setUp()
82      {
83          $this->hasOne('Series', array(
84              'local' => 'seriesId',
85              'foreign' => 'id')
86          );
87          $this->hasOne('Book as SequelTo', array(
88              'local' => 'sequelToIsbn',
89              'foreign' => 'id')
90          );
91          $this->hasMany('Book as Sequels', array(
92              'local' => 'id',
93              'foreign' => 'sequelToIsbn')
94          );
95      }
96  }
97
```

KUVIO 21. Luokan Book suhteet mukaan lukien itseviittaus

Yhden suhde moneen ja monen suhde yhteen -relaatioiden toteutukset ovat hyvin samankaltaisia kuin yhden suhde yhteen -relaation toteutus. Luokka, jolla on siis suhde moneen toisen luokan instanssiin, määritellään setUp-metodissa Doctrine_Record::hasMany-metodilla. Foreign-määreeseen tulee sarakkeen nimi, joka suhteen toisessa luokassa viittaa määriteltävään luokkaan, ja local-määreeseen tulee luokan identifioiva sarake, kuten kuviossa 20 riveillä 77–79. Relaation toiseen luokkaan tulee samanlainen hasOne-metodikutsu kuin edellä yhden suhde yhteen -relaatiossa. (Blanco ym. 2010, 113–114.)

Puurakenne on itseviittaava relaatio. Kuviossa 21 luokalla on sekä hasMany-metodikutsu (rivi 91) että hasOne-metodikutsu (rivi 83), jotka kummatkin viittaavat kutsuvaan luokkaan. (Blanco ym. 2010, 114–115.)

2.5.5 Monen suhde moneen -relaatiot

```
100 class AuthorOfBook extends Doctrine_Record
101 {
102     public function setTableDefinition()
103     {
104         $this->setTableName('authorofbook');
105         $this->hasColumn('authorId', 'integer', 11, array(
106                 'primary' => true
107             )
108         );
109         $this->hasColumn('bookIsbn', 'string', 14, array(
110                 'primary' => true
111             )
112         );
113     }
114 }
```

KUVIO 22. Liitostaulu Book-luokan ja Author-luokan välillä

Monen suhde moneen -relaatioissa on tarve käyttää ylimääristä liitostaulua (KUVIO 22). Molemmilla luokilla on hasMany-metodikutsu, joka viittaa suhteen toisen luokan instansseihin. Metodissa kuitenkin tarvitaan myös refClass-määrettä, joka viittaa liitostauluun (KUVIO 23). Parametrit ’local’ ja ’foreign’ viittaavat liitostaulun kenttiin. (Blanco ym. 2010, 116–117.)

```
 6  class Author extends Doctrine_Record
 7  {
 8      //...
 9
10      public function setUp()
11      {
12          $this->hasMany('Book as Books',
13              array(
14                  'local' => 'authorId',
15                  'foreign' => 'bookIsbn',
16                  'refClass' => 'AuthorOfBook'
17              )
18          );
19      }
20  }
```

KUVIO 23. Monen suhde moneen -relaation määrittely

2.5.6 Olioiden luonti ja haku

Toisistaan riippuviin tietueisiin pääsee Doctrinessa käsiksi käyttäen määriteltyjä suhteita. Esimerkiksi yhden suhde yhteen -tapauksessa, jos user-oliolla on email-olio, pääsee user-olion email-olioon käsiksi seuraavalla käskyllä: $user->Email. Jos kyseistä oliota ei ole olemassa, luo Doctrine automaattisesti sen. Näin olion arvoja muutettaessa ja kutsuttaessa save-metodia tallentuu uusi tietue tietokantaan. Jos relaatio on monen suhde moneen, palauttaa ->Email kutsu Doctrine_Collection-olion. Doctrine_Collection-olion yli voidaan iteroida foreach-rakenteella tai siihen voidaan lisätä uusia tietueita ja tallentaa ne tietokantaan (KUVIO 24). Myös Doctrine_Core::getTable-metodia voidaan käyttää tietyn taulun käsittelyyn, jolloin voidaan käyttää Doctrine_Table-luokan metodeita kuten find-metodia alkoiden hakemiseen (KUVIO 25). (Blanco ym. 2010, 152–155.)

```
22
23  $author1 = $book->Authors[];
24  $author1->lname = 'Brandon';
25  $author1->fname = 'Sanderson';
26  $author1->byear = '1975';
27  $author2 = $book->Authors[];
28  $author2->lname = 'Jordan';
29  $author2->fname = 'Robert';
30  $author2->byear = '1948';
31  $book->save();
32
```

KUVIO 24. Doctrine_Collection-olion käyttöesimerkki

```
32
33  $book = Doctrine_Core::getTable('Book')->find('978-0765326355');
34
```

KUVIO 25. Kirjan, jonka isbn tunnetaan entuudestaan, haku tietokannasta (Blanco ym. 2010, 105)

Edellä esitettyjä tapoja yksinkertaisempi ja tehokkaampi tapa hakea joukko olioita tietokannasta on käyttää DQL-kyselyitä (Blanco ym. 2010, 155). DQL-kielen ominaisuuksia käsitellään luvussa 2.6. Doctrine_Record- ja Doctrine_Collection-luokilta löytyy myös toArray- ja fromArray-metodit taulukoiden kanssa työskentelyyn (Blanco ym. 2010, 172–173).

2.5.7 Doctrine_Table-luokan periyttäminen

Joskus on tarpeellista laajentaa Doctrine_Table-luokkaa, esimerkiksi kun on tarve käyttää omaa kyselyä kyseisessä luokassa. Tällöin pitää Doctrine_Core::getTable-metodin palauttaa uusi laajennettu luokka Doctrine_Table-luokan sijasta. Tämä onnistuu kutsumalla setAttribute-metodia joko connection- tai manager-oliolle parametrillä Doctrine_Core::ATTR_TABLE_CLASS ja laajennetun luokan nimellä. Nyt getTable-metodin palauttamalle oliolle voidaan kutsua laajennetun Doctrine_Table-luokan uusia metodeja. Myös kyselyistä vastaavaa kantaluokkaa voidaan laajentaa tarvittaessa. Tällöin laajennetaan Doctrine_Query-luokkaa ja asetetaan Doctrine_Core::ATTR_QUERY_CLASS-attribuutti. Yksittäiselle Doctrine_Record-luokan lapsiluokalle voidaan myös toteuttaa oma

Doctrine_Table-toteutus. Nimeämiskäytäntönä, jos Record-luokan nimi on Author, tulee Table-luokan nimeksi AuthorTable. Nyt, jos AuthorTable-luokkaan toteutetaan hasBooks-metodi, voidaan metodia kutsua komennolla: Doctrine_Core::getTable('Author')->hasBooks(). (Blanco ym. 2010, 59–60.)

2.6 DQL

DQL (Doctrine Query Language) on oliopohjainen kyselykieli. Se on rakennettu auttamaan kompleksisten olioiden haussa. Sen avulla voidaan hakea yksittäisiä olioita tai kokonaisia oliopuita käyttäen luokkien nimiä, luokan kenttien nimiä ja luokkien välisiä suhteita. Tehokkuuden kannalta kannattaa aina käyttää joko DQL-kyselyitä tai puhdasta SQL-kielistä kyselyä. DQL:n hyvänä puolena on, että se toimii samalla lailla eri tietokantaohjelmien välillä. Puhtaan SQL-kyselyn saa käyttämällä Doctrine_RawSql-luokkaa. Koska DQL on hyvin samantapainen SQL-kielen kanssa, on sen omaksuminen suhtellisen helppoa, jos osaa valmiiksi SQL-kieltä. Erona SQL:llään on se, että DQL:n valintalauseet kohdistetaan luokkiin eikä tauluihin ja DQL on suunniteltu palauttamaan joukko tietueita eli olioita. DQL myös ymmärtää asetetut suhteet automaattisesti, joten yhdisteen ehtoja ei tarvitse erikseen määritellä kyselylle (KUVIO 26). Näin kappaleessa 2.5.4 määriteltyjen suhteiden mukaan voidaan yhdistää luokkia. (Blanco ym. 2010, 31, 177–179, 273.)

```
34
35  $q = Doctrine_Query::create()
36      ->from('Book b')
37      ->leftJoin('b.Authors a');
38
```

KUVIO 26. Esimerkki DQL-kyselystä

2.6.1 Select-kyselyt

Uusi kysely saadaan luotua Doctrine_Query::create-metodilla (KUVIO 26). Metodi palauttaa Doctrine_Query-olion, jolla on kyselyn rakentamiseen tarvittavat metodit. Select-metodi on tarkoitettu datan hakemiseen yhdestä tai

useammasta komponentista. Metodille annetaan parametrina merkkijono, joka koostuu pilkulla erotetuista haettujen kenttien nimistä tai koostefunktioista. Jokaisessa select-metodissa pitää olla vähintään yksi nimetty kenttä tai vaihtoehtoisesti *-merkki, joka tarkoittaa kaikkien kenttien hakua. Doctrine ei kuitenkaan käytä *-merkkiä DQL-kyselyä käännettäessä SQL-kyselyksi, vaan tehdäkseen tehokkaamman kyselyn, käyttää se tässä tapauksessa kaikkia sarakkeiden nimiä SQL-kyselyssä. Select-metodia voi kutsua useamman kerran samalle Doctrine_Query-oliolle. (Blanco ym. 2010, 180–186.)

Doctrine_Query-luokan from-metodia käytetään kertomaan, miltä komponenteilta tietueita haetaan. From-metodille annetaan parametrina merkkijono pilkulla eroteltuja luokkien eli komponenttien nimiä. Nimelle voidaan antaa alias erottamalla alias välilyönnillä luokan nimestä, esimerkkinä kuvio 27 rivi 41. (Blanco ym. 2010, 182–183, 189.)

Query-luokan where-metodille ilmoitetaan ehdot, jotka tietueen pitää täyttää, jotta se tulee valituksi kyselyssä. Metodin parametrin tulee olla lause, joka on tosi kaikille valituille tietueille. Jos Doctrine_Query-oliolle ei kutsuta where-metodia, valitaan kaikki mahdolliset tietueet. Metodissa voidaan käyttää kaikkia DQL:n tukemia funktioita ja operaattoreita paitsi koostefunktoita. Having-metodia voidaan käyttää yhdessä koostefunktoiden kanssa (KUVIO 27). (Blanco ym. 2010, 184, 194–195, 212–213.)

```
38
39  $q = Doctrine_Query::create()
40      ->select('b.title, COUNT(a.id) as yhteensa')
41      ->from('Book b')
42      ->leftJoin('b.Authors a')
43      ->where('b.genre = "scifi"')
44      ->having('yhteensa > 1');
45
```

KUVIO 27. Koodiesimerkki DQL-kyselyn rakentamisesta

Limit- ja offset-metodeja voidaan käyttää rajoittamaan haettujen tietuieden määrää. Metodit ottavat parametrinaan kokonaisluvun. Query-luokan orderBy-

metodia voidaan käyttää tulosten järjestämiseen. Metodille annetaan parametrina järjestämiseen käytettävän kentän tai kenttien nimet ja joko ASC-määre käytettäessä kasvavaa järjestystä tai DESC-määre käytettäessa laskevaa järjestystä. (Blanco ym. 2010, 185, 215–218.)

2.6.2 Update-kyselyt

Update- eli päivityskysely luodaan Doctrine_Query::create-metodilla samoin kuin select-kyselyn tapauksessa. Kysely päivittää olemassa olevien tietueiden kenttien tiedot ja palauttaa arvonaan päivitettyjen tietueiden lukumäärän. Kyselystä saadaan update-kysely kutsumalla Doctrine_Query::update-metodia (KUVIO 28), joka ottaa parametrinaan päivitettävän komponentin eli luokan nimen. Doctrine_Query::set metodi ottaa vastaan kaksi parametria. Ensimmäinen parametri vastaa päivitettävien kenttien nimiä. Toinen parametri on arvo tai arvot, jotka päivitettävät kentät saavat. Where-metodikutsulla rajataan, mitä kaikkia tietueita päivitys koskee. Kyselyssä voi myös käyttää orderBy-metodikutsua, joka määrä tietueiden päivitysjärjestyksen. Limit-metodilla voidaan rajoittaa päivitettävien tietueiden määrää. Update-kysely suoritetaan kutsumalla kyselyolion execute-metodia, kuten kuviossa 28 rivillä 28. Tämä metodi siis palauttaa tietueiden määrän, joihin kysely vaikutti. (Blanco ym. 2010, 187–189.)

```
23
24  $q = Doctrine_Query::create()
25      ->update('Author a')
26      ->set('a.byear', $byear)
27      ->where('u.id = ?', $authorId);
28  $q->execute();
29
```

KUVIO 28. Update-kysely

Käytettäessä DQL-kyselyä tietueiden päivittämiseen, olioita ei ladata kyselyn välillä tietokannasta vaan kysely käsittelee suoraan tietokantaa. Jos päivitettävään tietueeseen liittyvä olio on valmiiksi alustettuna, voidaan oliolle kutsua Doctrine_Record::save-metodia tietueen päivittämiseen. (Blanco ym. 2010, 246.)

2.6.3 Delete-kyselyt

Delete- eli poistokyselylle luodaan kyselyolio käyttäen Doctrine_Query::create-metodia samoin kuin select- ja update-kyselyn tapauksissa. Kysely tuhoaa halutut tietueet ja palauttaa poistettujen tietuiden lukumäärän. Delete-tyyppinen kysely luodaan käyttämällä delete-metodia ja tämäntyyppisen kyselyn kanssa voidaan käyttää from-, where-, orderBy- ja limit-metodeja. Metodien toiminta vastaa update-kyselyn vastaavia metodikutsuja. Jos where-metodia ei kutsuta, poistetaan kaikki from-metodissa määritetyt tietueet. Kysely suoritetaan execute-metodilla, joka palauttaa poistettujen tietueiden määrän. (Blanco ym. 2010, 188–189.)

Tietueita voidaan poistaa myös kahdella eri tavalla käyttäen Doctrine_Record-luokkaa tai Doctrine_Collection-luokkaa. Ensimmäinen tapa on kutsua Doctrine_Record-olion delete-metodia, joka poistaa kyseisen Doctrine_Record-olion osoittaman tietueen. Jos on olemassa Doctrine_Collection-kokoelma Doctrine_Record-olioita, kutsuttaessa Doctrine_Collection::delete-metodia kutsutaan kaikkien kokoelmaan kuuluvien olioiden delete-metodia. (Blanco ym. 2010, 250–251.)

2.6.4 DQL-lauseiden vakiot, määreet ja alikyselyt

Tässä luvussa käsiteltävät määreet koskevat pääosin Query-olioiden where-metodin parametriksi annettavia DQL-lauseita. DQL-lause voi sisältää merkkijono-, kokonaisluku-, liukuluku-, boolean- ja enum-vakioita. Merkkijonovakiot määritellään heittomerkkien sisällä. Kokonaisluku- ja liukulukuvakioille pätee PHP:n vastaava syntaksi. Boolean-vakio voi saada joko arvon true tai arvon false. Enumeroidut arvot määritellään yhtäläisesti merkkijonovakioiden kanssa. (Blanco ym. 2010, 195–199.)

Käytettäessä muuttuvia parametreja käytetään kyselyssä parametrin tilalla kysymysmerkkiä. Parametrit annetaan where-metodille taulukkona. Kysymysmerkit korvataan kyselyssä järjestyksessä taulukossa annetuilla parametreilla. Kun muuttuvia parametreja on vain yksi, voidaan toisena parametrina antaa pelkkä skalaarivakio taulukon sijasta, kuten kuviossa 29 rivillä

54. Nimetyt parametrit annetaan metodille assosiaatiotauluna ja ne alkavat kyselyssä aina kaksoispisteellä, kuten kuviossa 29 rivillä 55. (Blanco ym. 2010, 199–200.)

```
50
51  $q = Doctrine_Query::create()
52      ->from('Book b')
53      ->leftJoin('b.Author a')
54      ->where('b.genre = ?', 'fantasy')
55      ->andWhere('a.fname = :fname AND a.lname = :lname',
56          array(':fname' => 'Brandon', ':lname' => 'Sanderson'));
57
```

KUVIO 29. Koodiesimerkki muuttuvasta parametrista ja nimetystä parametrista

DQL:llä on mahdollista tehdä alikyselyitä from-, select- ja where-lauseista. Alikysely voi pitää sisällä mitä tahansa avainsanoja, jotka on sallittu tavallisessa select-kyselyssä. Alikyselyn avulla kyselyn eri osat on mahdollista järjestellä niin, että ne ovat erillään toisistaan. Alikysely myös yksinkertaistaa kyselyitä, jotka muuten vaatisivat monimutkaisia liitoksia tai unioneita. (Blanco ym. 2010, 207–208, 211.)

IN- ja EXIST-operaatioita käytetään alikyselyn kanssa (KUVIO 30). Jos alikysely palauttaa joukon alkioita, joihin kuuluu IN-operaation operandi, tulee IN-operaation arvoksi tosi. EXIST-lauseke palauttaa arvon tosi, jos alikysely palauttaa yhden tai useamman tietueen. Muussa taupauksessa operaatio palauttaa arvon epätosi. (Blanco ym. 2010, 201, 203.)

```
58
59  $q = Doctrine_Query::create()
60      ->from('Book b')
61      ->where('b.isbn IN (SELECT bo.isbn
62              FROM Book bo
63              INNER JOIN bo.Series s
64              WHERE s.id = ?)', $seriesId);
65
```

KUVIO 30. IN-operaatio ja alikysely

LIKE-operaatiolla voidaan verrata merkkijonoa malliin. Mallissa alaviiva (_) tarkoittaa yksittäistä merkkiä ja prosenttimerkki (%) tarkoittaa mielivaltaista joukkoa merkkejä. Jos verrattava merkkijono on arvoltaan NULL on LIKE-lausekkeen tulos tuntematon. (Blanco ym. 2010, 202.)

ALL-lauseke on ehdollinen lauseke, joka palauttaa arvon tosi, jos sen kanssa käytetty vertailuoperaattori palauttaa arvon tosi kaikkien alikyselyn palauttamien arvojen kanssa. Lause palauttaa arvon epätosi tapauksessa, jossa vertailuoperaattori palauttaa vähintään yhden epätoden arvon alikyselyn palauttamien arvojen kanssa. ANY-lauseke on ehdollinen lauseke, joka palauttaa arvon tosi, kun vertailuoperaatio palauttaa ainakin kerran arvon tosi ja epätosi, jos alikysely on tyhjä tai vertailu ei palauta yhtään arvoa tosi. Vertailuoperaattoreita =, <, <=, >, >= ja <> voidaan käyttää ALL- ja ANY-lausekkeissa. (Blanco ym. 2010, 206.)

2.6.5 Nimetyt kyselyt

Doctrinen nimetyt kyselyt on tehty auttamaan mallien ja kyselyjen ylläpitäimisessä. Jos tietokantamalli muuttuu, täytyy kyselytkin päivittää vastaamaan uutta mallia. Nimetyllä kyselyllä voidaan tehdä Doctrine_Query-kyselyitä ja käyttää niitä uudelleen, jolloin muutokset täytyy tehdä vain yhteen paikkaan. Tuen nimetyille kyselyille tarjoaa Doctrine_Query_Registry-luokka. Luokka on tarkoitettu nimettyjen kyselyjen rekisteröintiin ja nimeämiseen. Kyselyt lisätään add-metodilla. Metodi tarvitsee kaksi parametria: kyselyn nimen, joka koostuu luokan nimestä ja kyselyn nimestä, ja itse DQL-kyselyn (KUVIO 31). Kun kysely on luotu, pääsee siihen käsiksi Doctrine_Table::find-metodilla. Parametriksi metodille annetaan nimetyn kyselyn nimi. (Blanco ym. 2010, 221–222.)

```
65
66  $r = Doctrine_Manager::getInstance()->getQueryRegistry();
67  $r->add('Book/all', 'FROM Book b');
68  $bookTable = Doctrine_Core::getTable('Book');
69  //find all books
70  $books = $bookTable->find('all');
71
```

KUVIO 31. Nimetyn kyselyn rekisteröinti ja käyttö

Nimetty kysely voidaan luoda myös yksittäiseen Doctrine_Record-luokkaan liittyvässä Doctrine_Table-luokassa eli esimerkiksi Author-luokkaan liittyvässä AuthorTable-luokassa. Nimetty kysely määritellään Doctrine_Table-luokan construct-metodissa (KUVIO 32). Kyselyyn päästään käsiksi hakemalla kyseinen AuthorTable-luokka Doctrine_Core::getTable-metodilla ja kutsumalla saadulle oliolle createNamedQuery-metodia. Samoin kuin edellä, metodille annetaan parametrinä nimetyn kyselyn nimi. Kysely voidaan nyt suorittaa kutsumalla sille execute-metodia. (Blanco ym. 2010, 221–223.)

```
1
2  class AuthorTable extends DoctrineTable
3  {
4      public function construct()
5      {
6          //Named Query defined using DQL string
7          $this->addNamedQuery('get.by.id', 'SELECT a.fname, a.lname FROM Author a
8                                WHERE a.id = ?');
9
10         // Named Query defined using Doctrine_Query object
11         $this->addNamedQuery(
12             'get.by.similar.names', Doctrine_Query::create()
13                 ->select('a.id, a.fname, a.lname')
14                 ->from('Author a')
15                 ->where('LOWER(a.fname) LIKE LOWER(?)')
16         );
17     }
18 }
19
20 $authors = Doctrine_Core::getTable('Author')
21     ->createNamedQuery('get.by.similar.names')
22     ->execute(array('%bran%'));
23
```

KUVIO 32. Nimetyn kyselyn määritteleminen Doctrine_Table-luokassa (Blanco ym. 2010, 156)

2.7 Datavalidointi

Doctrine tarjoaa työkalut tietokantataulujen sarakkeiden tietojen validoinnille. Rajoite määritellään Doctrine_Record-lapsiluokan hasColumn-metodissa (KUVIO 33). Jos käyttäjä yrittää syöttää tietokantaan tietoja, jotka rikkovat asetettuja rajoitteita, antaa Doctrine virheviestin. Virhe tapahtuu, vaikka annettu arvo olisi mallissa asetettu oletusarvo. Tietueen validius voidaan tarkistaa kutsumalla Doctrine_Record::isValid-metodia. (Blanco ym. 2010, 297, 300–301.)

```
23
24  $this->hasColumn('isbn', 'string', 14, array(
25      'minlength' => 10));
26
```

KUVIO 33. Koodiesimerkki sarakemäärittelystä, jolle on käytetty validointia

Notnull-rajoite varmistaa, ettei kenttä voi saada arvoa null. Eli siis sarake, jolle on määritetty notnull-rajoite, ei ota vastaan null arvoja. Email-rajoite katsoo, että sarakkeen arvoksi tulee validi sähköpostiosoite. Not blank -validaattori muistuttaa not null -validaattoria. Erona on, että not blank -validaattori antaa virheen jos arvo on tyhjä merkkijono tai merkkijono, jossa on vain välimerkkejä (whitespace). No space -validaattori tarkistaa, ettei tallennettavassa arvossa ole yhtään välilyöntejä. Date-määre tarkistaa, onko annettu arvo validi päivämäärä. Past-rajoite katsoo, että annettu arvo on validi päivämäärä menneisyydessä ja future-rajoite katsoo, onko annettu arvo validi päivämäärä tulevaisuudessa. Minlength-rajoittella määrätään kentän minimipituus. Country-rajoite tarkistaa, onko annettu arvo validi maakoodi. Ip-validaattori tarkistaa, onko annettu arvo validi IP-osoite. Validaattori kelpuuttaa sekä IPv4- että Ipv6-osoitteet (Boyd 2010). Htmlcolor-määre tarkistaa, onko arvo validi HTML-väri annettuna heksadesimaalilukuna. Range-määreelle annetaan taulukkona ala- ja yläraja, joiden välistä arvon tulee löytyä. Jos taulukko on assosiaatiotaulu ja avain on nolla, määrää arvo alarajan. Jos avain on ykkönen, määrää arvo ylärajan. Unique-rajoitteella voidaan määrätä, että tiettyä arvoa omaavia rivejä saa olla vain yksi kappale tietokannan taulussa. Rivit, joiden arvo on null, katsotaan olevan keskenään uniikkeja. Regexp-määreellä voidaan rajoittaa sarakkeen arvot vastaamaan tiettyä säännöllistä lauseketta. Creditcard-validaattori validoi luottokorttien numeroita. Readonly-

määreella voidaan pakottaa kenttä vain luettavaksi. Lopuksi unsigned-validaattori validoi etumerkittömiä kokonaislukuja. (Blanco ym. 2010, 298, 299–318.)

2.8 Transaktiot

Transaktio on joukko operaatioita, jotka ovat atomisia, konsistentteja, eristettyjä ja kestäviä (ACID – Atomic, Consistent, Isolated, Durable). Joukko operaatioita on atominen, jos joko kaikki operaatiot onnistuvat tai jos jokin operaatioista epäonnistuu, kumotaan jo suoritettujen komentojen vaikutus. Konsistenttiudella tarkoitetaan, että operaatiot eivät muuta datan konsistenttiutta. Yleinen esimerkki on tilanne, jossa tililtä toiselle siirretään rahaa. Rahansiirto-operaation jälkeen tilien saldojen summan täytyy olla pysynyt muuttumattomana. Joukko operaatioita on eristettyjä, jos transaktion operaatiot eivät näy suoritusvaiheessa muille käyttäjille. Muutokset näkyvät vasta, kun koko transaktio on suoritettu. Transaktio on kestävä, kun onnistuneesti läpivietynä transaktion vaikutukset eivät häviä tietokannasta. Jos transaktio epäonnistuu, tietokannan tulee säilyä muuttumattomana. (Microsoft 2012.)

Doctrinessa oletuksena kaikki operaatiot suoritetaan transaktioissa. Doctrinen transaktoissa pitää ottaa huomioon, että Doctrine suorittaa aina insert-, update- ja delete-kyselyt transaktion viimeisenä. Ensimmäisenä suoritetaan lisäykset, jonka jälkeen päivitykset ja viimeisenä poistot. Doctrine osaa myös optimoida poistoja niin, että kaikki samaan komponenttiin liittyvät poistot rakennetaan yhteen ja samaan kyselyyn. Transaktiot aloitetaan Doctrine_Connection::beginTransaction-metodia käyttäen ja lopetetaan kutsumalla Doctrine_Connection::commit-metodia (KUVIO 34). (Blanco ym. 2010, 426.)

```
26
27  $conn->beginTransaction();
28  $author = new Author();
29  $author->fname = 'Brandon';
30  $author->lname = 'Sanderson';
31  $author->byear = 1975;
32  $author->save();
33  $authorCollection = Doctrine::getTable('Author')
34      ->findByDql('lname = "Jordan"');
35  $authorCollection[0]->byear = 1948;
36  $authorCollection[0]->save();
37  $conn->commit();
38
```

KUVIO 34. Esimerkki Doctrinen transaktio toiminnon käytöstä

2.9 Sivutus

Yleinen tapaus reaalimaailmassa on se, että tietokannasta tarvitsee listata taulujen sisältöä ja taulussa on mahdollisesti tuhansia rivejä. Tämä voi johtaa muistia tuhlaaviin ja käytettävyyden kannalta huonoihin ratkaisuihin. Tähän hyvänä ratkaisuna on sivutus. Doctrinen avulla voidaan jakaa listaus sivuihin sekä kontrolloida sivulinkkien tyyliä. Doctrine_Pager-luokka toimii periaatteessa lähes samalla tavalla kuin Doctrine_Query-luokka. Pager-luokan konstruktorille annetaan parametrina ensin itse tietueet hakeva Doctrine_Query-tyyppinen kysely, kuten kuviossa 35 rivillä 47. Toisena parametrina annetaan tämän hetkinen sivu ja kolmantena parametrina annetaan tietueiden määrä yhdellä sivulla. Kolmannen parametrin voi jättää pois, jolloin se on oletuksena 25. Kun Doctrine_Pager-luokalle kutsutaan execute-metodia, palauttaa metodi Doctrine_Collection-olion, jossa on kyseisen sivun tietueet. (Blanco ym. 2010, 475–476.)

```
39
40 $currentPage = $_POST['page'];
41 $resultsPerPage = 24;
42 $q = Doctrine_Query::create()
43     ->from('Book b')
44     ->leftJoin('b.Authors a')
45     ->leftJoin('b.Series s');
46
47 $pager = new Doctrine_Pager($q, $currentPage, $resultsPerPage);
48 $result = $pager->execute();
49
```

KUVIO 35. Sivutus Book-taulun sisällöstä

Joskus yksinkertaiset sivutukset eivät riitä, jos halutaan esimerkiksi tulostaa linkit eri sivuille. Doctrine_Pager antaa tähän mahdollisuuden. Kun Pager-oliolta kutsutaan getRange-metodia ensimmäisenä parametrinä joko 'sliding' tai 'jumping' ja toisena parametrinä sivutusalueen pituus, palauttaa metodi Doctrine_Pager_Range-olion. Oliolla on metodi rangeAroundPage, joka palauttaa listan aktiivisen sivun ympärillä olevista sivuista. Sliding-tyylisen osituksen kanssa alue liikkuu sulavasti aktiivisen sivun kanssa, niin että aina kuin mahdollista, aktiivinen sivu on keskimmäisenä. Jumping-tyylisessä osituksessa sivulinkit ovat kiinteitä. Esimerkiksi 1–5, 6–10 ja niin edelleen. (Blanco ym. 2010, 479–480.)

3 OLIO-RELAATIO-MALLINNUS AVAINASIAKASJÄRJESTELMÄSSÄ

3.1 Avainasiakasjärjestelmä

Avainasiakasjärjestelmä toteutettiin lahtelaiselle painotalolle. Järjestelmä on sivusto, joka on tarkoitettu palvelemaan vakioasiakkaita ja tarjoamaan heille kanava yhteydenpitoon painotalo kanssa. Ennen järjestelmän ohjelmoinnin aloittamista sivuston ulkonäön olivat suunnitelleet Lahden ammattikorkeakoulun muotoiluinstituutin opiskelijat. Heiltä saatiin layout-kuviot, joiden pohjalta sivuston ulkonäkö toteutettiin ja ominaisuudet suunniteltiin.

3.2 Tietokanta

Sovelluksen tietotana toimii MySQL-relaatiotietokantaohjelmisto. Tietokantaa käytettiin tallentamaan sovelluksessa persistenttinä pysyvät tiedot. Tietokannasta löytyy myös taulu lokitiedon keräämiseen sovelluksen tuottamista virheviesteistä.

Monen suhde moneen -tapauksia oli kaksi. Kummassakin tapauksessa monen suhde moneen –tapaukset ilmenivät laajennuksina kehityksen loppuvaiheessa. MySql-tietokantamäärittelyssä ei käytetty viiteavainmäärittelyä, koska Doctrinelle oli määritelty taulujen väliset suhteet. Tämä yksinkertaisti tietokannan ylläpitämistä hieman.

3.3 Mallien luonti sovelluksessa

Muutamasta ensimmäisestä tietokannan taulusta luotiin malliluokat automaattisesti, mutta tietokannan kehittyessä huomattiin Doctrine_Record-lapsiluokkien kirjoittaminen manuaalisesti helpommaksi ja tehokkaammaksi tavaksi.

Doctrine olisi antanut mahdollisuuden luoda kirjoitetuista malliluokista tietokannan taulut, kuten luvussa 2.4 kerrottiin. Tietokanta ja tietokannan taulut luotiin kuitenkin käyttäen PhpMyAdmin-käyttöliittymää ja luotujen taulujen perusteella kirjoitetettiin Doctrine_Record-luokat. PhpMyAdmin on ilmainen

ohjelmisto työkalu, joka on kirjoitettu PHP:lla ja tarkoitettu MySQL-tietokannan hallinointiin verkkokäyttöliittymän ylitse (PhpMyAdmin 2012).

3.4 Doctrinen mallien käyttö Zend-ohjelmistokehyksessä

Doctrine_Record-lapsiluokat sijaitsivat Zend-ohjelmistokehyksen hierarkiassa models-kansiossa, eli ne toimivat MVC-arkkitehtuurin malliosiossa. Doctrine initialisoitiin Zend-projektin bootstrap.php tiedostossa (KUVIO 36). Tietokantayhteys avattiin luvun 2.2.2 ohjeiden mukaisesti Doctrine_Manager::connection-metodilla. Doctrine konfiguroitiin lataamaan mallit aggressiivisesti ja lataamaan taulut automaattisesti (luku 2.3). Tietokantayhteyteen liittyvät vakiot, kuten mallien sijainti ja tietokannan osoite, määriteltiin application.ini-konfiguraatiotiedostossa.

```
1
2  protected function _initDoctrine() {
3      $manager = Doctrine_Manager::getInstance();
4      $manager->setAttribute(
5          Doctrine::ATTR_AUTO_ACCESSOR_OVERRIDE,
6          true);
7      $manager->setAttribute(
8          Doctrine::ATTR_MODEL_LOADING,
9          Doctrine_Core::MODEL_LOADING_AGGRESSIVE);
10     $manager->setAttribute(
11         Doctrine::ATTR_AUTOLOAD_TABLE_CLASSES,
12         true);
13     $manager->setAttribute(
14         Doctrine_Core::ATTR_DEFAULT_TABLE_CHARSET,
15         'utf8');
16     $doctrineOptions = $this->getOption('doctrine');
17     Doctrine::loadModels($doctrineOptions['models_path']);
18     $conn = Doctrine_Manager::connection(
19         $doctrineOptions['dsn'],
20         'doctrine');
21     $conn->setCharset('utf8');
22     $conn->setAttribute(
23         Doctrine::ATTR_USE_NATIVE_ENUM,
24         true);
25     return $conn;
26 }
27
```

KUVIO 36. Doctrinen initialisointi Zend-projektin bootsrap.php-tiedostossa

Kun malliolioiden väliset suhteet oli määritelty (luku 2.2.4), pystyi niitä käyttämään helposti isojen tietomäärien hakemiseen ilman, että tarvitsi kirjoittaa monimutkaisia kyselylauseita. Esimerkiksi, jos saatavilla oli User-olio, pystyttiin Doctrine_Record::hasMany-metodilla määritetyn suhteen avulla hakemaan kaikki kyseiselle käyttäjälle kuuluvat viestit.

Doctrinea käytettiin kyselyjen rakentamiseen, joilla haettiin esimerkiksi käyttäjälistaus tietokannasta tai käyttäjän antamien lomaketietojen tallentamiseen tietokantaan. Kirjautuneen käyttäjän tiedot haettiin kirjautumisvaiheessa ja tallennettiin Doctrine_Record-oliona välimuistiin. Näin välimuistista voitiin kysyä tämän hetkisen käyttäjän tiedot ja sen perusteella tulostaa hänelle kuuluvat tiedot sivulle.

Vähintään yksi Doctrine-kysely, Doctrine_Record-olion luonti tai muokkaus tapahtui jokaisessa sivuston osiossa. Doctrine oli sovelluksessa kaikkialla läsnäoleva.

Zend-ohjelmistokehyksessa on mahdollista luoda lomake-olioita, joita voidaan dynaamisesti lisätä sivuille (Vaswani 2010, 50–51). Lomake-olioihin lisättiin dynaamisesti vaihtoehdot esimerkiksi kirjautuneen käyttäjän perusteella. Vaihtoehdot saatiin käyttämällä Doctrine_Query-kyselyä.

Eri tietuieden luonti- ja muokkaussivujen lomakkeita käsittelevässä koodiosiossa joutui usein käyttämään täysin samanlaisia DQL-kyselyrakenteita. Nämä kyselyt olisi voitu toteuttaa mallien Doctrine_Table-luokkaan.

Sivuttaessa isoa määrää tietoa käytettiin Doctrinen Pager-luokkaa. Luokan palauttamat arvot tulostettiin näkymään ja sivulinkit tulostettiin Pager-luokan antaman viimeisen sivun indeksin perusteella. Sivulinkkien tulostukseen olisi luultavasti kannattanut käyttää luokan tarjoamaa skaalaus mahdollisuutta (luku 2.9). Eräässä sivuston osiossa ei voitu käyttää hyväksi Doctrine_pager-luokkaa, koska tiedot töistä sijaitsivat ulkoisessa järjestelmässä, johon oli XML-rajapinta. Tästä syystä tuotantolistauksen sivutuslogiikka rakennettiin perustumaan

laskennallisesti. Tämä teki tuotantolistauksesta suhteelisen raskaan, koska kaikki tuotannot työt täytyi hakea kerralla, koska XML-rajapinta ei tarjonnut mahdollisuutta rajoittaa palautetun työlistauksen määrää kappalemääräisesti.

3.5 Mallien laajennus

Luvussa 2.5.7 suositeltiin toteuttamaan usein käytetyt kyselyt Doctrine_Table-luokan laajennuksena. Suosituksesta poiketen usein käytetyt kyselyt toteutettiin sovelluksessa Doctrine_Record-luokan staattisina metodeina. Tämä vähensi ylläpidettävien tiedostojen määrää, mutta lisäsi koodin monimutkaisuutta

3.6 Monen suhde moneen -relaatioden toteutus sovelluksessa

Monen suhde moneen -relaatiot toteutettiin luvun 2.5.5 tavasta poiketen käyttämättä refClass-määrettä. Tämä siitä syystä, että kummatkin sovelluksen monen suhde moneen -relaatiot olivat alun perin yhden suhde yhteen -relaatioita. Muutospyynnön tullessa, jotta muutokset saatiin nopeasti sovellukseen, lisättiin sovellukseen liitostaulu ja liitostaulun malliluokkaan hasOne()-määritykset (luku 2.5.4) osoittamaan liitostaulun liittämiin luokkiin. Tämän muutoksen ansiosta lähdekoodi ei kyselyjen kannalta muuttunut merkittävästi.

Monen suhde moneen -suhteeseen kohdistuvissa kyselyissä kutsuttiin leftJoin()-metodia liitostaululle ja rajattiin joukko haluttuihin tietueisiin lisäämällä where-määre. Esimerkiksi kun haluttiin tulostaa tietylle käyttäjälle kuuluvat tietueet, liitettiin ensin tietueiden tauluun liitostaulu leftJoin-käskyllä. Palautetut arvot rajattiin tietyn käyttäjän tietueisiin where-määreellä, jossa ehtona oli, että liitostaulun viittaaman taulun viiteavain on sama kuin haetun käyttäjän tunnus.

3.7 Validointi

Validointiin sovelluksessa käytettiin Zend-ohjelmistokehyksen validointityökaluja. Tämä oli järkevämpää kuin Doctrinen validointityökalujen käyttö, koska Zendissä validointityökalut olivat rakennettu valmiiksi Zendin

lomakkeiden yhteyteen. Jos käytössä olisi ollut Doctrinen validointityökalut, olisi koodi pilkkoutunut ja koodista tullut vaikeammin ylläpidettävää.

4 YHTEENVETO

Tavoitteena oli toteuttaa tietokannan olio-relaatio-malinnus avainasiakasjärjestelmään, joka oli rakennettu toimimaan Zend PHP-ohjelmistokehyksen päälle. Ongelmaan ratkaisuna oli Doctrine PHP-kirjasto. Periyttämällä Doctrine_Record-luokasta malliluokat tietokannan tauluille, voitiin näitä luokkia käyttää sovellusalueen logiikan yhdistämiseen tietokannan tauluihin. Kun malliluokat oli luotu, voitiin Doctrine_Query-oliota käyttää hyväksi kyselyjen tekemiseen tietokantaa vastaan. Kyselyn rakentamiseen käytettiin Doctrinen tarjoamaa DQL-kieltä, joka yhdessä Doctrine_Query-olion kanssa palautti Doctrine_Collection-olion, joka sisälsi tarvitut tietueet. Isojen tietuejoukkojen sivuttamiseen käytettiin Doctrine_Pager-luokan tarjoamia työkaluja.

Parannettavaa avainasikasjärjestelmässä olisi esimerkiksi sivutuksessa. Sivutuksen sivunnumeroinnin logiikan ohjelmointiin olisi ollut mahdollista käyttää Doctrinen sivutusta helpottavia työkaluja kuten osittaista sivunumerointia. Tästä ei olisi näkynyt konkreettista hyötyä vielä kehitysvaiheessa. Vasta kun järjestelmä oli ollut tietyn aikaa asiakkaalla testikäytössä, oli dataa kertynyt tarpeeksi, niin että osittaisen sivutuksen hyödyt tulivat näkyviin. Avainasiakasjärjestelmän kehitystyö jatkuu tulevaisuudessa.

LÄHTEET

Blanco, G., Borschel, R., Vesterinen, K. & Wage, J. 2010. Doctrine ORM for PHP: Guide to Doctrine 1.2. Sensio Labs.

Boyd, J. 2010. Doctrine_Validator_Ip fails on IPV6 IP addresses [viitattu 13.3.2012]. Saatavissa: http://www.doctrine-project.org/jira/browse/DC-458

Esa Print Oy. 2012. Etusivu [viitattu 9.3.2012]. Saatavissa: http://www.esaprint.fi/

Fonecta Oy 2011. Tunnuslukutiivistelmä [viitattu 8.9.2011]. Saatavissa: http://www.finder.fi/Mainostoimistoja/Korpimedia%20Oy/LAHTI/taloustiedot/1612766

Fowler, M. 2003. Patterns of enterprise application architecture. Addison-Wesley.

Korpimedia Oy. 2012. Korpimedia [viitattu 9.3.2012]. Saatavissa: http://www.korpimedia.fi/

Microsoft. 2012. What is a transaction? [viitattu 13.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx

Miller, J. 2009. Design Patterns for Data Persistence [viitattu 10.3.2012]. Saatavissa: http://msdn.microsoft.com/en-us/magazine/dd569757.aspx

Netbeans. 2012. NetBeans IDE 7.1 Features, PHP Development [viitattu 10.3.2012]. Saatavissa: http://netbeans.org/features/php/

Ohjelmointiputka. 2009. PDO – hyvä tapa käsitellä tietokantoja [viitattu 11.3.2012]. Saatavissa: http://www.ohjelmointiputka.net/koodivinkit/vinkki.php?id=25206

PhpMyAdmin. 2012. PhpMyAdmin, About [viitattu 15.3.2012]. Saatavissa: http://www.phpmyadmin.net/home_page/

Vaswani, V. 2010. Zend Framework: A Beginner’s Guide. McGraw-Hill.

Webopedia. 2012. WAMP [viitattu 11.3.2012]. Saatavissa: http://www.webopedia.com/TERM/W/WAMP.html