KARELIA-AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma

Marko Purhonen

LOMAKE-ELEMENTIN TIETOMALLIN KEHITTÄMINEN

Opinnäytetyö
Toukokuu 2016

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Toukokuu 2016
Tietotekniikan koulutusohjelma

Karjalankatu 3
80200 JOENSUU
(013) 260 600

Tekijä(t)
Marko Purhonen

Nimeke
Lomake-elementin tietomallin kehittäminen

Toimeksiantaja
Collapick Company Oy

Tiivistelmä

Tässä opinnäytetyössä suunniteltiin sähköisen lomake-elementin tietomalli ja toteutettiin siihen entiteetti- ja palveluluokat. Työn toimeksiantaja oli Collapick Company Oy, joka tarvitsi web-sovelluksiin lomake-elementin.

Tietomallin pohjana käytettiin Concrete5:n lomake-elementin tietomallia. Työssä perehdyttiin tietomallin rakenteeseen ja toimintaan, jonka jälkeen tietomallia kehitettiin vaatimusmäärittelyjen pohjalta eteenpäin. Tietomalliin lisättiin rekursiivinen rakenne ja versiohistoria.

Työn toteutuksessa käytettiin PHP:tä ja Doctrine ORM -kirjastoa. Entiteetti- ja palveluluokat vastasivat MVC-arkkitehtuurin model-osiota. Palveluluokkien toteutuksessa käytettiin SOA:n mukaista kerrosrakennetta. Versiohistorian toteutuksessa käytettiin Doctrinen elinkaaritapahtumien kuunteluun EventListener-luokkaa. Työn tuotteena valmistui tietomallin pohjalta lomake-elementin entiteetti- ja palveluluokat. Työn tuloksia voi hyödyntää jatkossa projekteissa, joissa tarvitaan esimerkiksi rekursiota tai versiohistoriaa.

Kieli
suomi

Sivuja 29

Asiasanat
PHP, Doctrine ORM, MVC, SOA, lomake-elementti, rekursio, versiohistoria

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
May 2016
Degree Programme in Information Technology

Karjalankatu 3
80200 JOENSUU
FINLAND
+358 13 260 600

Author (s)
Marko Purhonen

Title
Development of the Data Model for the Form Block

Commissioned by
Collapick Company Oy

Abstract

In this thesis, the data model of the form block was designed, to which entity and service classes were implemented. The thesis was commissioned by Collapick Company Oy, which needed a form block for web-based applications.

The data model was based on the form block of Concrete5. In this task, the structure and behaviour of the data model were studied, after which the data model was developed further on the basis of the requirement specifications. A recursive structure and version history were added to the data model.

PHP and Doctrine ORM library were used in the implementation of the work. The entity and service classes corresponded to the model section of the MVC architecture. The service classes were implemented by using the layer structure based on SOA. In the implementation of version history, the EventListener class was used in order to listen to the Doctrine life cycle events. As a product of the work, the entity and service classes were completed on the basis of the data model. The results of this study can be utilized further, for example, in projects where recursion or version history are needed.

Language
Finnish

Pages 29

Keywords
PHP, Doctrine ORM, MVC, SOA, Form Block, recursion, version history

Sisältö

Käsitteet

Concrete5	PHP-pohjainen sisällönhallintajärjestelmä
DBAL	Doctrine Database Abstraction & Access Layer, suomeksi Doctrinen tietokannan abstrahointi- ja liityntäkerros
DQL	Doctrine Query Language, suomeksi Doctrinen kyselykieli
ERD	Entity Relationship Diagram, suomeksi entiteetti-relaatiodiagrammi
Form Block	Concrete5:n lomake-elementti
MVC	Model View Control -ohjelmistoarkkitehtuuri
ORM	Object-Relational Mapping, suomeksi olio-relaatiomappaus.
PDO	PHP Data Objects on tietokannan liityntärajapinta PHP:ssä.
SOA	Service-Oriented Architecture, suomeksi palvelukeskeinen ohjelmistorakenne

1 Johdanto

Toimeksiantaja, Collapick Company Oy, tarvitsi verkkosovelluksiinsa toimivan lomake-elementin, jolla voidaan tehdä esimerkiksi asiakaskyselyitä. Opinnäytetyössä suunniteltiin kyselylomakkeen tietomalli sekä toteutettiin PHP:llä entiteetti- ja palveluluokat. Työssä käytettiin pohjana Concrete5:n lomake-elementtiä (Form Block).

Työn toteutuksessa käytettiin Visual Paradigm-, NetBeans-, Git- ja MySQL Workbench -sovelluksia. Entiteetti- ja palveluluokat toteutettiin PHP:n Doctrine ORM -kirjastoa käyttäen. Kirjastoa käytettiin tietokannan abstrahoinnissa.

2 Toimeksianto

Toimeksiantaja halusi kehittää oman mukautetun kyselylomakkeen Concrete5-sisällönhallintajärjestelmään. Toimeksiantoon sisältyi kyselylomakkeen tietomallin kehittäminen ja tämän lisäksi tarvittavien entiteetti- ja palveluluokkien toteuttaminen. Kehitystyön pohjana käytettiin Concrete5:n lomake-elementtiä.

2.1 Rajaukset

Opinnäytetyössä keskityttiin suunnittelemaan ja toteuttamaan lomake-elementin tietomalli. Tietomallin lisäksi sovelluksesta toteutettiin ainoastaan palvelu- ja entiteettiluokat. Kontrollerit, käyttöliittymät sekä Concrete5-integraatio jätettiin työn ulkopuolelle.

2.2 Työn vaatimuslista

Toimeksiannon yhteydessä tehtiin työn tavoitteista vaatimuslista. Vaatimuslistaa täydennettiin toteutusvaiheessa palveluluokkien rakenteen osalta. Lista toimeksiannon vaatimuksista:

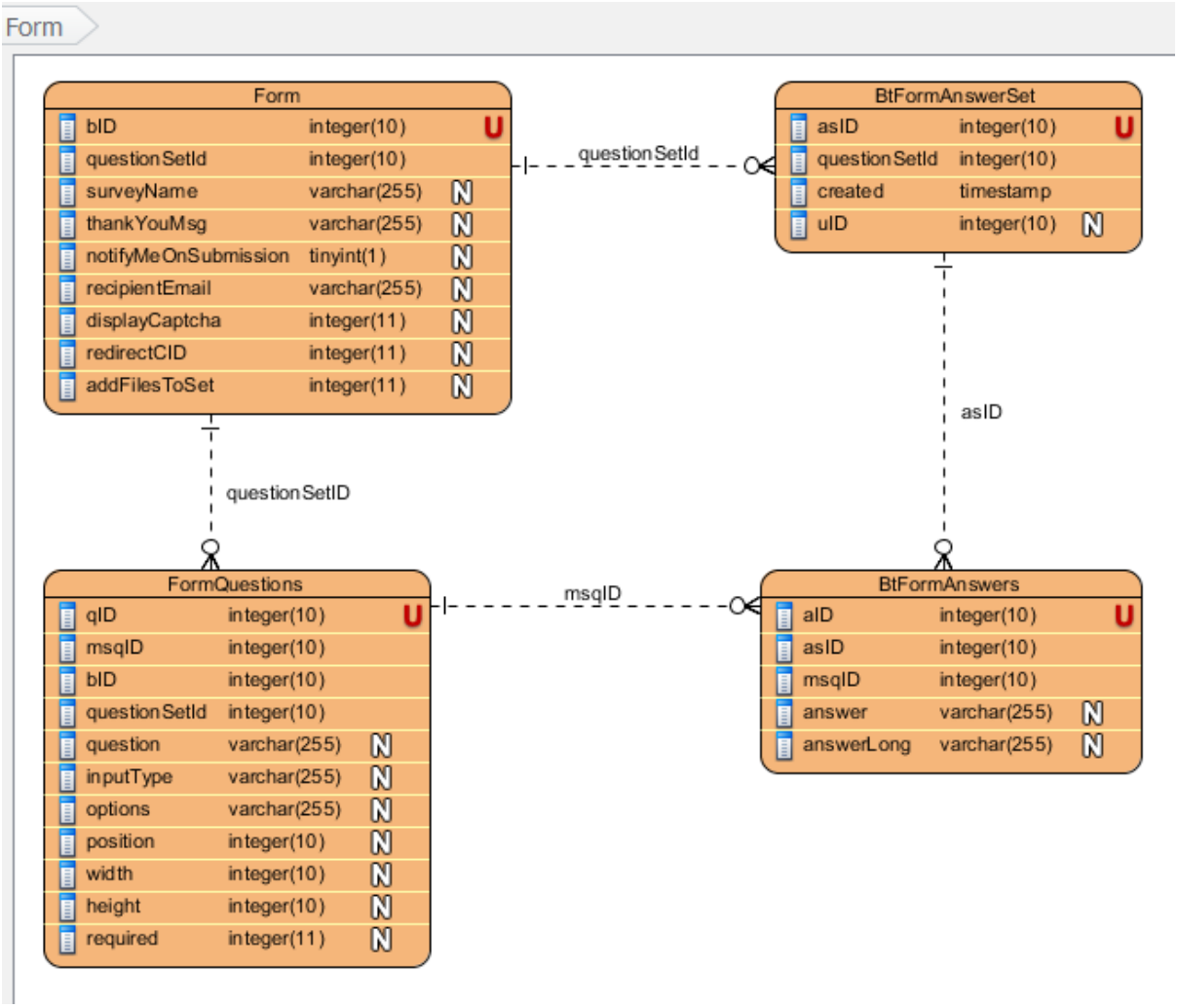
- Tietomallissa piti olla rekursiivinen tietorakenne.
- Tietomallissa piti toteuttaa kysymysten versiohistoria.
- Tietotyypit piti olla dynaamisesti valittavissa kysymystä luotaessa.
- Monivalintakysymysten vastausvaihtoehdot piti tallentaa omiin tauluihin.
- Tiedot piti tallentaa tietotyyppiä vastaavaan kenttään tai tauluun.
- Piti toteuttaa PHP:n Doctrine ORM -kirjastoa käyttäen entiteetti- ja palveluluokat.
- Palveluluokissa piti toteuttaa kerrosrakenne.

2.3 Taustaa

Toimeksiantaja tarvitsi verkkosovelluksiin toimivan lomake-elementin. Heidän käytössään oli Concrete5-sisällönhallintajärjestelmä, jossa oli valmis lomake-elementti. Concrete5:n lomake-elementissä oli kuitenkin puutteita, joista haluttiin eroon ja joiden takia se ei soveltunut sellaisenaan toimeksiantajan käyttöön.

Ensimmäinen puute oli kysymysten versiohistorian puuttuminen. Lomake-elementissä oli mahdollista muuttaa kysymyksen sisältöä ja tietotyyppiä kysymykseen vastaamisen jälkeen niin, että alkuperäisestä sisällöstä tai tietotyypistä ei jäänyt tietokantaan minkäänlaista merkintää. Tällöin oli mahdollista, että vastaus ei enää vastannut kysymystä. Tämä rikkoi tietokannan tietojen eheyden.

Toinen puute oli, että lomake-elementissä kaikkien tietotyyppien vastaukset tallennettiin merkkijonona samaan kenttään (answer) ja kysymyksen tietotyyppiä ei tallennettu vastaus-tauluun (BtFormAnswers) ollenkaan (kuva 1). Tämä puolestaan vaikeutti vastausten käsittelyä.



Kuva 1. Concrete5:n lomake-elementin tietomalli

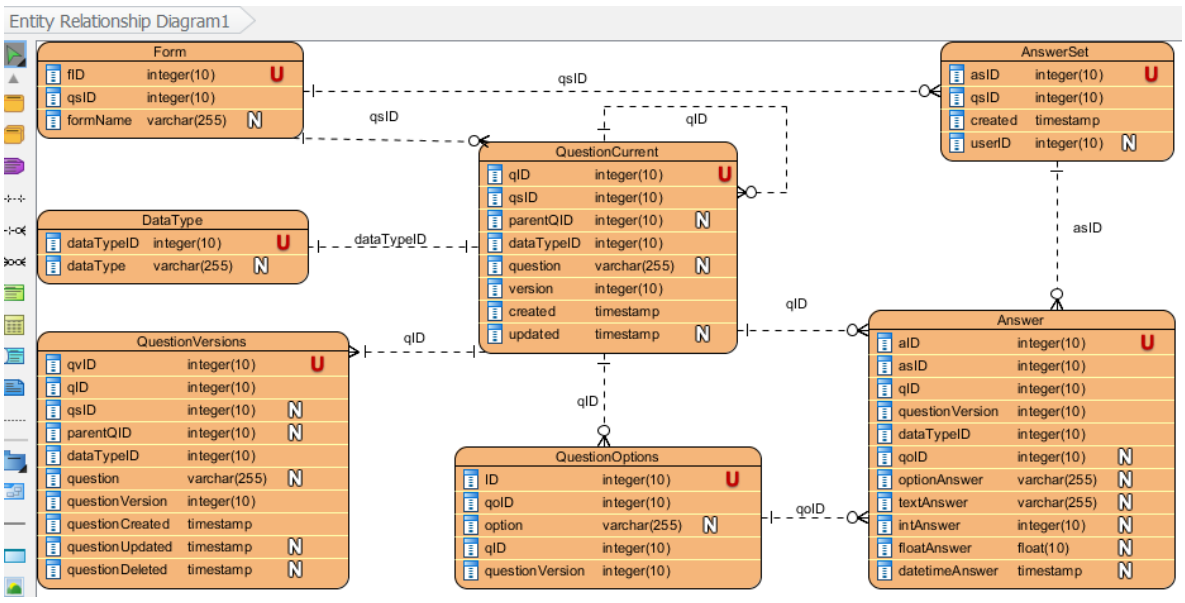
Myös monivalintakysymysten vastausvaihtoehtojen tallennustapaa haluttiin parantaa. Alkuperäisessä lomake-elementissä kaikki vastausvaihtoehdot tallennettiin kysymystaulun (FormQuestions) yhteen kenttään (options) prosenttimerkillä erotettuna. Tämä haluttiin toteuttaa niin, että vaihtoehdot tallennetaan omaan tauluunsa.

Lisäksi kysymyksille haluttiin tehdä jatkokysymyksiä. Tämän toteutukseen tarvittiin rekursiivinen rakenne kysymystauluun.

3 Tietomalli

Tietomallilla esitetään tietojen välisiä suhteita ja rakenteita [1]. Tietomallin pohjana käytettiin Concrete5:n lomake-elementtiä. Lomake-elementti antoi hyvän pohjan kehittää tietomallia eteenpäin. Tietomallin suunnittelussa käytettiin Visual Paradigm-ohjelmaa, jolla tehtiin ERD-malli. ERD on tietomalli, jota käytetään kuvaamaan tietokantaan tallennettujen entiteettien välisiä suhteita [2]. Tietomallin toteutuksessa pyrittiin mahdollisimman yksinkertaiseen toteutukseen.

Kysymysten versiohistoriaan otettiin mallia Jonblog-sivuston kirjoituksesta [3]. Kysymysten versiohistoria toteutettiin siten, että kysymyksille tehtiin kaksi taulua. Ensimmäiseen (QuestionCurrent) tuli kysymyksen viimeisin versio (kuva 2). Toiseen tauluun (QuestionVersion) tallennettiin kysymyksen kaikki versiot. Tämä mahdollisti esimerkiksi sen, että kysymyksen aikaisempia versioita QuestionVersion-taulusta oli mahdollista palauttaa QuestionCurrent-tauluun.



Kuva 2. Lomake-elementin valmis tietomalli

Tietotyypeille tuli oma taulu (DataType), josta kysymystä luotaessa haettiin saatavilla olevat tietotyypit. Tietotyypeistä esimerkiksi radiobutton vaati, että kysymykselle annetaan vastausvaihtoehdot. Näille vaihtoehdoille luotiin oma taulu

(QuestionOptions). Vastaustauluun (Answer) tehtiin jokaiselle tietotyypille omat kenttänsä, joka mahdollisti tietojen tallentamisen omina tietotyyppeinään.

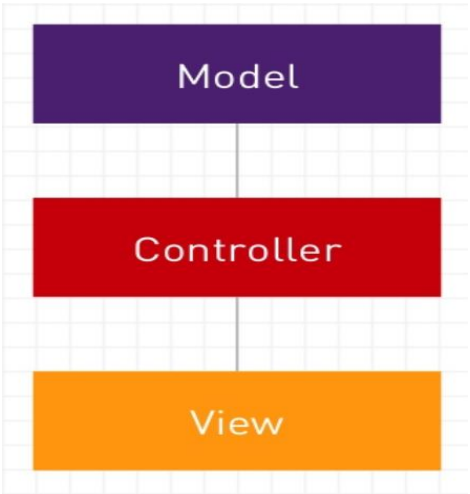
Jatkokysymykset toteutettiin rekursiolla kysymystaulussa. Käytännössä tämä toteutettiin lisäämällä kysymyksen tietueeseen parentQID-kenttä.

4 Ohjelmistoarkkitehtuuri

Työssä yhdistettiin MVC- ja SOA-mallien ohjelmistoarkkitehtuuria. Toimeksiannon mukaan tässä työssä toteutettiin vain Doctrinen entiteetti- ja palveluluokat, jotka vastaavat MVC-mallin models-osaa. Toimeksiantajan toiveesta palveluluokat toteutettiin kerrosmallin mukaan, jossa palveluluokat jaettiin SOA:n mukaisesti kerroksiin.

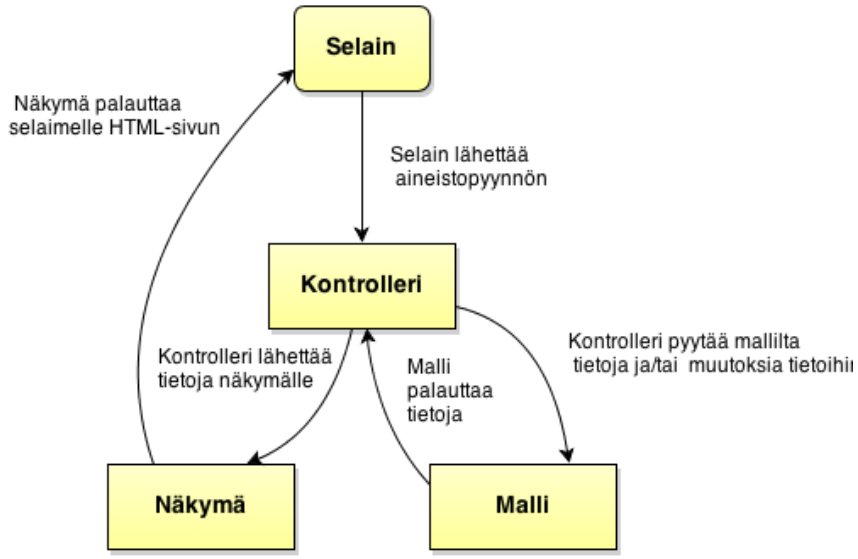
4.1 MVC

MVC-mallissa sovellus on jaettu kolmeen osaan: kontrollereihin, näkymiin ja malleihin (kuva 3).



Kuva 3. MVC-malli

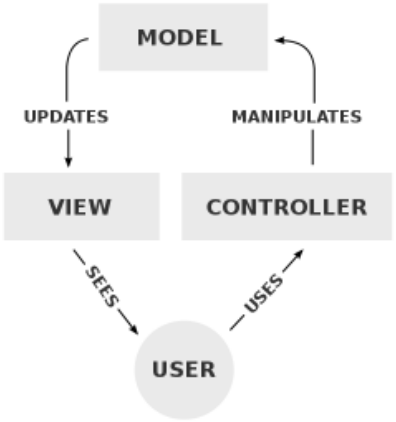
Advanced Kittenryn mukaan MVC-arkkitehtuuri tarkoittaa sitä, että koodi jaetaan kolmeen osaan: näkymiin, kontrollereihin ja malleihin (engl. views, controllers ja models). Mallit käyttävät tietokantaa ja abstrahoivat sen ohjelmointirajapinnaksi, jota muu koodi käyttää (kuva 4). Näkymät hoitavat kaiken HTML:ään liittyvän koodin ja kontrollerit ottavat vastaan käyttäjän antamat aineistopyynnöt, hakevat ja päivittävät tietoa malleja käyttäen ja välittävät tiedot näkymälle, joka näyttää sivun. [4.]



Kuva 4. MVC-mallin osien välinen kommunikaatio

Advanced Kittenryn mukaan MVC noudattaa seuraavia nyrkkisääntöjä (kuva 5):

- Malli vastaa tiedon käsittelemisestä ja abstrahoinnista ja ainoastaan siitä. Mallissa ei ikinä käsitellä HTML:ää tai käyttäjän lähettämiä pyyntöjä ja parametreja, nämä ovat näkymän ja kontrollerin tehtäviä.
- Kaikki HTML-koodi sijoitetaan näkymiin.
- Kaikki käyttäjän lähettämät GET- ja POST-paremetrit vastaanotetaan kontrollerissa.
- SQL-kyselyjä ei ikinä tehdä kontrollereiden tai näkymien puolella. [4.]



Kuva 5. MVC-mallin tiedon kulkusuunta [5]

Advanced Kittenryn mukaan [4] mallit sisältävät kaikki sovelluksen tietokannassa olevan tiedon ja käsittelyyn liittyvät toiminnot. Kaikki tietokantaa käsittelevä koodi kuuluu siis malleihin. Mallit toteutetaan keräämällä eri asioihin ja tietokohteisiin liittyviä toimintoja luokiksi ja/tai funktioiksi.

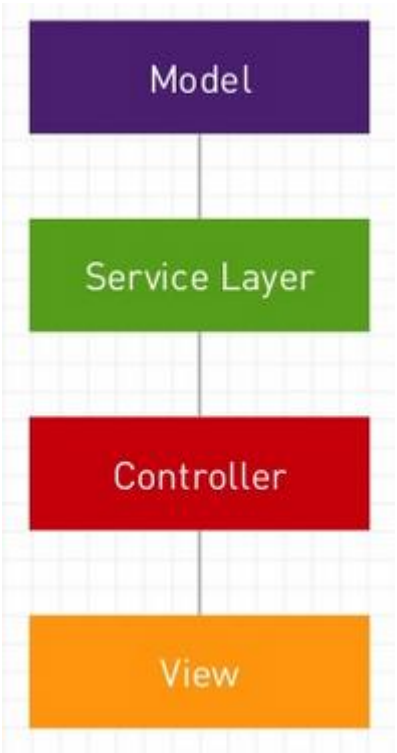
Näkymä määrittää käyttöliittymän ulkoasun ja tietojen näytön esityksen käyttöliittymässä. Tietokantasovelluksen tapauksessa käytetään nk. template-tiedostoja eli tiedostoja, joissa määritellään sivulle tuleva HTML-koodi, ja sen sekaan tulevat muuttujat. [4.]

Kontrolleri (tunnetaan myös nimellä käsittelijä tai ohjain) sitoo yhteen mallin ja näkymän ja välittää tietoa niiden välillä. Kontrolleri vastaanottaa käyttäjältä tulevat aineistopyynnöt, käyttää mallia tiedon hakemiseen ja muuttamiseen, ja lopuksi yleensä siirtää suorituksen jollekin näkymälle. Yleensä kontrolleri välittää samalla näkymälle jotain tietoa näytettäväksi, esimerkiksi listan mallin palauttamia olioita tai virheviestin. Kontrollerissa sen sijaan ei koskaan käsitellä tietokantaa suoraan, tai lähetetä selaimelle HTML:ää tai muuta sisältöä. Nämä ovat mallien ja näkymien tehtäviä. [4.]

4.2 Doctrine Project

Doctrine Projectin kotisivujen mukaan Doctrine Project on koti usealle PHP-kirjastolle, jotka ovat keskittyneet tietokantojen tiedontallennusvälineisiin ja olioiden mappaukseen. Ydinprojektit ovat Object Relational Mapper (ORM) ja Database Abstraction and Access Layer (DBAL), jonka pohjalta se on rakennettu. Doctrine ORM perustuu JAVAn Hibernate ORM-kirjastoon ja se on mukautettu PHP:lle. [6.]

Doctrinea käytettäessä perinteinen MVC-mallin models-osio on jaettu kahteen osaan - palveluihin ja malliosaan (kuva 6). Malliosa koostuu entiteettiluokista, jotka vastaavat tietokannan tauluja. Palvelukerroksessa on palveluluokkia, joilla hoidetaan tiedonvälitys tietokannan ja sovelluksen välillä.

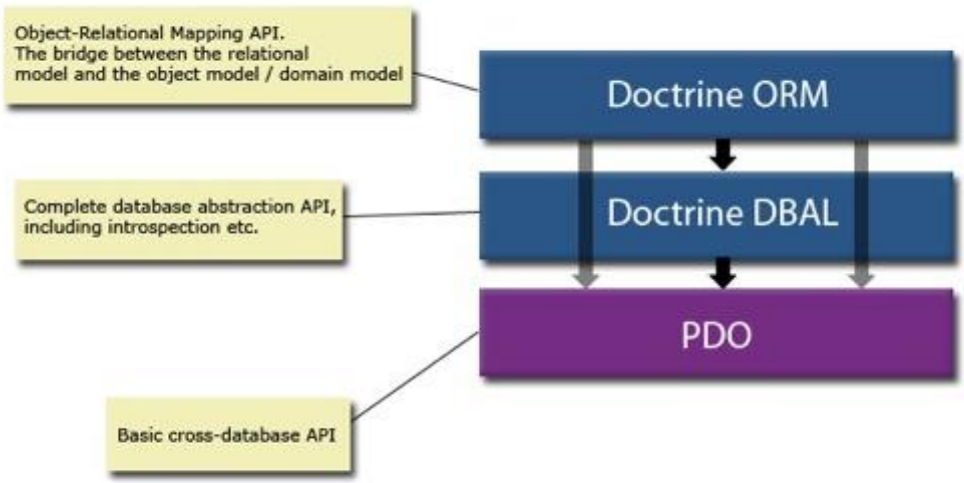


Kuva 6. MVC-malli Doctrinea käytettäessä

4.2.1 DBAL – Database Abstraction Layer

Doctrine DBAL on rakennettu PDO:n päälle ja se integroi natiivit laajennokset PDO API:in (kuva 7). DBAL tukee muun muassa seuraavia tietokantatoimittajia:

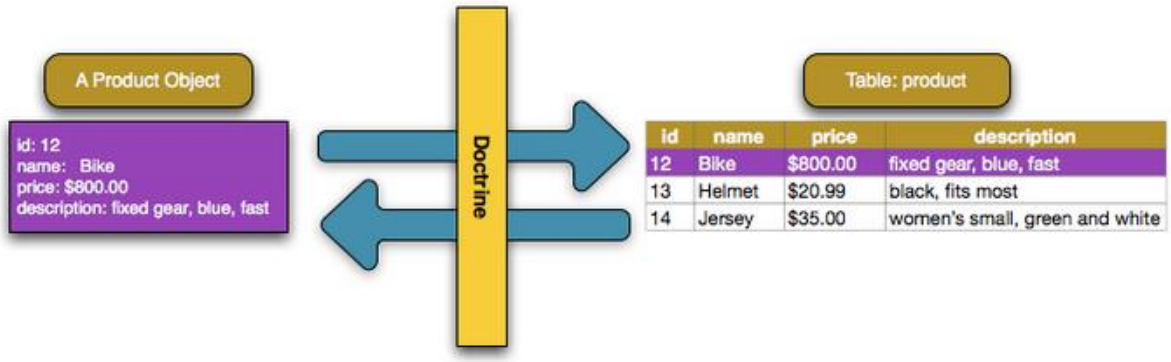
- MySQL
- Oracle
- Microsoft SQL Server
- PostgreSQL
- SAP Sybase SQL Anywhere
- SQLite
- Drizzle. [7.]



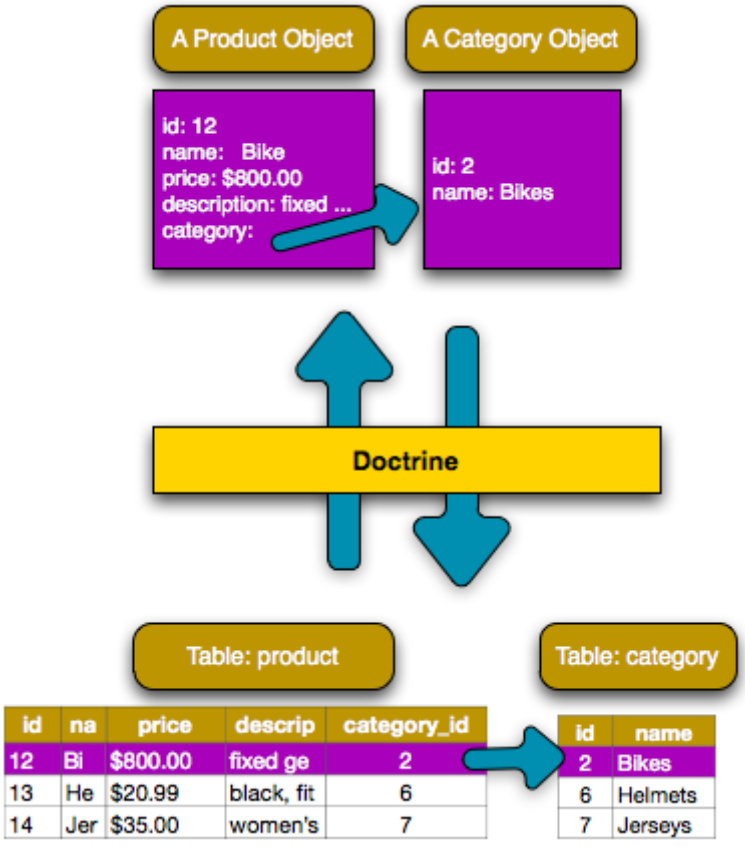
Kuva 7. Doctrine ORM:n ja DBAL:in suhde PDO:hon [8]

4.2.2 ORM – Object Relational Mapping

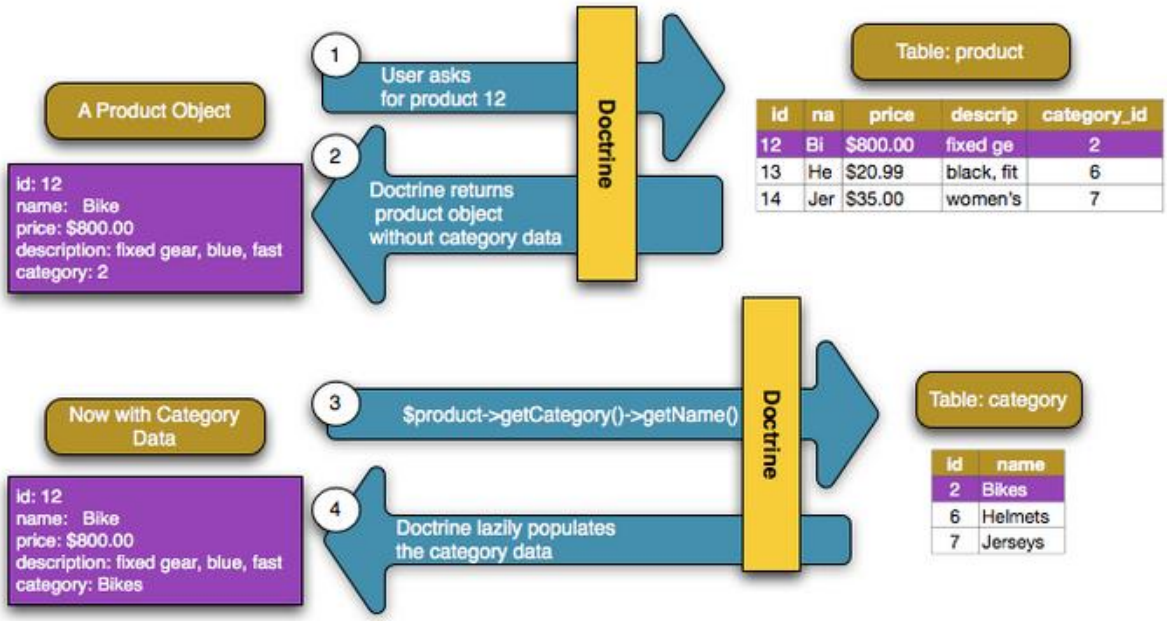
Doctrine ORM vastaa olio-relaatiomappauksesta. Se hoitaa olio- ja relaatiomallin välisen kommunikaation. Kuvissa 8, 9, ja 10 on esimerkkejä Doctrine ORM:n toiminnasta.



Kuva 8. Olio- ja relaatiomallin välinen kommunikaatio [9]



Kuva 9. Taulujen välinen liitos [9]



Kuva 10. Tiedon haku id:n perusteella [9]

4.2.3 Entity Manager ja DQL

EntityManager-luokka on yhteyspiste ORM:n toiminnallisuuksiin. Se sisältää metodit entiteettien käsittelyyn. Sen avulla voidaan luoda tietokantakyselyitä DQL -kielellä. DQL on Object Query Languagen johdannainen, joka muistuttaa Hibernate Query Languagea. DQL:ssä on kolme lauseketyyppiä: SELECT, UPDATE ja DELETE. Doctrinen käyttö ei vaadi tietokantojen toiminnan tuntemista mutta DQLa käytettäessä on hyvä tietää, kuinka tietokantojen kyselylausekkeita muodostetaan. [10.]

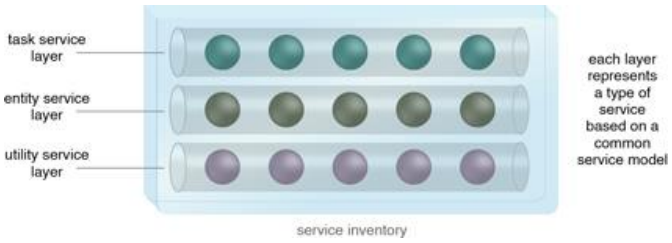
4.2.4 LifeCycle Events ja Custom Listeners

Doctrinella voidaan käyttää mukautettuja elinkaaritapahtumien (LifeCycleEvents) kuunteluun tarkoitettuja luokkia (EventListener). Näillä voidaan kuunnella Doctrinen toiminnanaikaisia elinkaaritapahtumia. Näitä tapahtumia ovat muun

muassa onFlush, prePersist, postPersist, preUpdate, postUpdate, postFlush ja postRemove. Kuuntelijaluokassa voidaan esimerkiksi kuunnella entiteetin päivitystä, jolloin Doctrine käyttää preUpdate- ja postUpdate-funktiota. [11.]

4.3 SOA

SOA on käsitteenä laaja kokonaisuus ja sitä voidaan käyttää ja tulkita monella tapaa. Tässä työssä on keskitytty soapatterns.orgin mukaiseen palveluiden kerrosrakenteeseen, jossa palveluluokat on jaettu kolmeen kerrokseen: utility service, entity service ja task service (kuva 11) [12].



Kuva 11. Palvelukerrokset [12]

Tampereen teknillisen yliopiston opetusmateriaali avaa tätä tulkintaa:

”Apupalvelu (utility service) tarjoaa toimintoja, joilla ei ole suoraa linkkiä liiketoimintavaatimuksiin. Muut palvelut ovat apupalveluiden asiakkaita.

Entiteettipalvelu (entity service) tarjoaa toimintoja, jonkin entiteetin käsittelyyn. Tämä entiteetti on sellainen, että se on käsitteenä olemassa ohjelmistomaailman ulkopuolellakin. Esim. Asiakas, Tilaus, Lasku, ...

Tehtäväpalvelu (task service) kuvaa jonkin liiketoimintaprosessin tai sen osan. Käyttää hyväkseen apu- ja entiteettipalveluita. Esim. palvelu joka luo laskutusraportin.” [13.]

5 Toteutus

5.1 Tietomallin toteutus

Projekti alkoi toimeksiannon jälkeen tutustumisella Concrete5:n lomake-elementtiin. Elementistä tutkittiin mitä tietoja käyttöliittymästä tallennetaan tietokantaan sekä tutustuttiin tietokannan rakenteeseen. Työssä keskityttiin lomake-elementin toiminnan kannalta tärkeisiin tietoihin, joten tietomallista jätettiin pois kaikki, mikä ei ollut välttämätöntä lomake-elementin toiminnan kannalta. Tietomallin suunnittelussa pyrittiin mahdollisimman yksinkertaiseen ratkaisuun.

Tietomalliin piti toteuttaa kysymysten versiohistoria. Tämä toteutettiin siten, että tietokantaan luotiin oma taulu kysymysten versioille (QuestionVersion). Versiotauluun päivitetään tiedot aina, kun kysymystauluun (QuestionCurrent) tallennetaan uutta tietoa, päivitetään olemassa olevaa tietoa tai poistetaan tietoa. Muokkauksen yhteydessä kysymyksen versionumero kasvatettiin yhdellä.

Tietotyypeille luotiin myös oma taulunsa. Tässä taulussa (DataType) olivat vain tiedot kysymysten tietotyypeistä. Kysymystä luotaessa tästä taulusta haetaan kysymykselle tietotyyppi, ja tämä tieto tallennetaan kysymystauluun. Tätä tietoa käytetään myös vastauksissa. Vastaustauluun (Answer) lisättiin kentät eri tietotyypeille. Vastausta tallennettaessa tiedot tallennetaan kyseessä olevan kysymyksen tietotyyppiä vastaavaan kenttään. Tällaisella toteutuksella vastausten analysointi helpottui, koska tiedot ovat valmiiksi oikeassa muodossa ja näin ollen niitä ei tarvitse jäsentää käsittelyn yhteydessä.

5.2 Malliosion toteutus

Ohjelmistoarkkitehtuurin malliosio toteutettiin PHP:n Doctrine ORM -kirjastoa käyttäen tietomallin pohjalta siten, että jokaista tietokannan taulua vastasi entiteettiluokka ja taulun kenttää vastasi entiteettiluokassa muuttuja. Oliomalliin luotiin AbstractEntity, johon sijoitettiin kaikille entiteeteille yhteiset muuttujat (kuva

12). Näin koodissa ei tarvinnut toistaa jokaiseen entiteettiluokkaan samoja muuttujia ja saatiin koodista rivimäärältään lyhyempi.

```
/**
 * @ORM\HasLifecycleCallbacks
 * @ORM\MappedSuperclass
 */
class AbstractEntity {

    /**
     * @ORM\Id @ORM\Column(type="integer", unique=true)
     * @ORM\GeneratedValue
     */
    protected $id;

    /**
     * @ORM\Column(type="datetime")
     */
    private $created;

    /**
     * @ORM\Column(type="datetime", nullable=true)
     */
    private $updated;

    public function __construct() {
        $this->created = new \DateTime("now");
    }

    /** @ORM\PreUpdate */
    public function updateTimeStamp() {
        $this->updated = new \DateTime("now");

    }
```

Kuva 12. Ote AbstractEntity-luokasta

5.3 Palveluluokkien toteutus

Palveluluokkien osalta toimeksiantaja toivoi, että ne toteutettaisiin kerroksittain. Tähän otettiin mallia palvelukeskeisestä ohjelmistoarkkitehtuurista. Palveluluokat jaettiin kolmeen kerrokseen: apu-, entiteetti- sekä tehtäväpalveluihin. Jokaiseen kerrokseen toteutettiin palveluluokat, jotka vastasivat entiteettiluokkia eli jokaisessa palvelukerroksessa oli palveluluokkia entiteettiluokkia vastaava mää-

rä. Tällä pyrittiin tekemään palveluluokista koodin rivimäärältä lyhyitä ja selkeitä. Osa luokista jätettiin ilman sisältöä - ne luotiin valmiiksi myöhempää käyttöä varten.

Jokaisessa palveluluokkakerroksessa käytettiin BaseService-luokkia. Näihin luokkiin sijoitettiin kyseisen kerroksen luokille yhteinen koodi. Esimerkiksi EntityBaseService sisälsi kaikki niin sanotut CRUD-funktiot (kuva 13).

```
class EntityBaseService implements EntityBaseServiceInterface {

    protected $em;
    protected $entityType;

    public function __construct(\Doctrine\ORM\EntityManager $em, $entityType = "") {
        $this->em = $em;
        $this->entityType = $entityType;
    }

    /**
     *
     * @return \Doctrine\ORM\EntityManager
     */
    public function getEntityManager() {
        return $this->em;
    }

    public function findAll() {
        $result = array();

        $qb = $this->em->createQueryBuilder()
                ->select("a")
                ->from($this->entityType, "a");

        try {
            $result = $qb->getQuery()->getResult(\Doctrine\ORM\AbstractQuery::HYDRATE_OBJECT);
        } catch (\Exception $exc) {
            echo $exc;
        }

        return $result;
    }
```

Kuva 13. Ote EntityBaseService-luokasta

Palveluluokille luotiin myös rajapinnat (interface) selkeyttämään koodia (kuva 14).

```php
<?php

namespace FormDataModel\Services\EntityServices\Interfaces;

interface EntityBaseServiceInterface {

    /**
     *
     */
    public function getEntityManager();

    /**
     * Abstract method for finding all entitys of certain type
     */
    public function findAll();

    /**
     * Abstract method for finding entity by id
     * @param type $id
     */
    public function findById($id);

    /**
     * Abstract method for finding entity by timeperiod
     * @param type $startDate
     * @param type $endDate
     */
    public function findByTimePeriod($startDate, $endDate);

    /**
     * Abstract function for saving entity
     * @param type $entity
     */
    public function save($entity);

    /**
     * Abstract function for deleting entity
     * @param type $entity
     */
    public function delete($entity);
}
```

Kuva 14. EntityBaseServiceInterface

Palveluluokille luotiin jokaiseen palvelukerrokseen ServiceFactory-luokka helpottamaan palveluluokkien käyttöä (kuva 15).

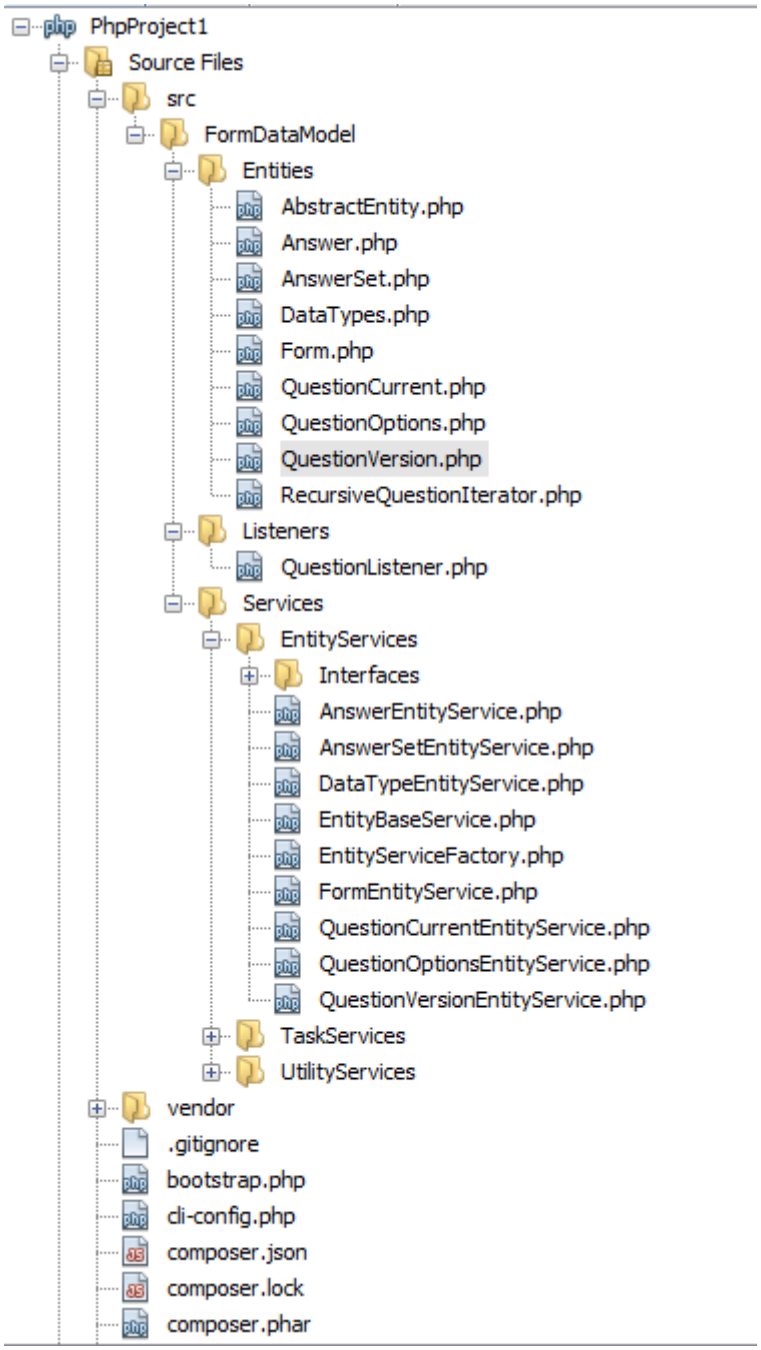
```php
final class EntityServiceFactory {

    private static $initialized = false;
    private static $entityManager = null;
    private static $answerEntityService = null;
    private static $answerSetEntityService = null;
    private static $dataTypeEntityService = null;
    private static $formEntityService = null;
    private static $questionCurrentEntityService = null;
    private static $questionOptionsEntityService = null;
    private static $questionVersionEntityService = null;

    private static function initialize() {
        if (self::$initialized) {
            return;
        }
        self::$entityManager = self::getEntityManager();
        self::$answerEntityService = new AnswerEntityService(self::$entityManager);
        self::$answerSetEntityService = new AnswerSetEntityService(self::$entityManager);
        self::$dataTypeEntityService = new DataTypeEntityService(self::$entityManager);
        self::$formEntityService = new FormEntityService(self::$entityManager);
        self::$questionCurrentEntityService = new QuestionCurrentEntityService(self::$entityManager);
        self::$questionOptionsEntityService = new QuestionOptionsEntityService(self::$entityManager);
        self::$questionVersionEntityService = new QuestionVersionEntityService(self::$entityManager);
        self::$initialized = true;
    }

    public static function getEntityManager() {
        global $entityManager;
        return $entityManager;
    }

    /**
     *
     * @return AnswerEntityService
     */
    static function getAnswerEntityService() {
        self::initialize();
        return self::$answerEntityService;
    }
```

Kuva 15. Ote EntityServiceFactory-luokasta

5.4 Nimeämiskäytäntö ja nimiavaruus

Ohjelmoinnissa käytettiin englantia nimeämisissä. Tämän lisäksi nimien piti olla niin yksiselitteisiä, ettei koodia tarvitse kommentoida erikseen. Nimet tuli kirjoittaa niin sanotulla ”Camel Case”-tyylillä siten, että luokkien nimet alkoivat isoilla ja funktioiden sekä muuttujien nimet pienillä kirjaimilla. Esimerkiksi vastausentiteettiluokan, Answer, käytöstä vastasivat palveluluokat: AnswerEntityService, AnswerTaskService sekä AnswerUtilityService.

Nimiavaruuksissa käytettiin myös ”Camel Case”-tyyliä. Nimiavaruudet muodostettiin sovelluksen kansiorakenteen mukaan (kuva 16). Esimerkiksi AnswerEntityServicen nimiavaruus oli FormDataModel\Services\EntityServices.



Kuva 16. Sovelluksen kansiorakenne

5.5 Rekursion toteutus

Tietomallin toteutuksessa kysymyksille piti pystyä tekemään alikysymyksiä. Tämä toteutettiin rekursiolla. QuestionCurrent-entiteetin rekursio toteutettiin Doctrinen omilla annotaatioilla.

Entiteettiin tuli parent- ja children-muuttujat, jotka viittasivat toisiinsa. Children-muuttujaan tuli @ORM\OneToMany-liitos ja parent-muuttujaan @ORM\ManyToOne-liitos (kuva 17). Käytännössä Doctrine tarvitsee children-muuttujaa vain olio-malliin, mutta sitä ei ole olemassa tietokannassa. Doctrine luo tietokantaan vain parent-muuttujan, johon tallennetaan children-entiteetit taulukkokokoelmana (ArrayCollection).

```php
/**
 * @ORM\Entity
 * @ORM\HasLifecycleCallbacks
 * @ORM\Table(name="questionCurrent")
 *
 */
class QuestionCurrent extends AbstractEntity {

    /**...3 lines */
    protected $version;

    /**...5 lines */
    protected $form;

    /**...5 lines */
    protected $dataTypes;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    protected $question;

    /**
     * @ORM\OneToMany(targetEntity="QuestionCurrent", mappedBy="parent", cascade={"persist"})
     */
    protected $children;

    /**
     * @ORM\ManyToOne(targetEntity="QuestionCurrent", inversedBy="children", cascade={"persist"})
     * @ORM\JoinColumn(name="parent", referencedColumnName="id")
     */
    protected $parent;

    public function __construct() {
        parent::__construct();
        $this->children = new ArrayCollection();
        $this->version = 1;
    }
```

Kuva 17. Ote QuestionCurrent-luokasta

Rekursiivisen rakenteen haku toteutettiin RecursiveQuestionIterator-luokalla, joka toteutettiin käyttäen PHP:n RecursiveIterator-luokkaa.

5.6 Event Listener

Viimeinen ongelma oli Doctrinen eventtien kuunteleminen. Tämä ongelma piti ratkaista, että pystyttäisiin päivittämään kysymysten versio-taulua samaan aikaan, kun kysymys-taulun tiedot muuttuvat. Tätä varten luotiin QuestionListener-luokka. Kuunneltavaan entiteettiin (QuestionCurrent) tuli ”@ORM\HasLifecycle Callbakcs”-annotaatio. Luokka kuunteli kuvan 18 mukaisia tapahtumia. Kuvassa 19 näkyy, kuinka postPersist-funktio luo QuestionVersion-entiteetin olion ja lisää siihen QuestionCurrent-entiteetin olion tiedot.

```
class QuestionListener {

    protected $container;

    public function onFlush(OnFlushEventArgs $eventArgs) {...63 lines }

    public function postRemove($eventArgs) {...30 lines }

    public function prePersist($eventArgs) {...44 lines }

    /** This method will be called on Doctrine postPersist event ...3 lines */
    public function postPersist(LifecycleEventArgs $args) {...15 lines }

    public function preUpdate($eventArgs) {...3 lines }

    /** This method will called on Doctrine postUpdate event ...3 lines */
    public function postUpdate(LifecycleEventArgs $args) {...27 lines }

    public function postFlush(\Doctrine\ORM\Event\PostFlushEventArgs $eventArgs) {...30 lines

}
```

Kuva 18. Ote QuestionListener-luokasta

```
/**
 * This method will be called on Doctrine postPersist event
 */
public function postPersist(LifecycleEventArgs $args) {
    $em = $args->getEntityManager();
    $entity = $args->getEntity();
    if ($entity instanceof QuestionCurrent) {
        $qv = new QuestionVersion();
        $qv->setQuestionCurrent($entity->getId());
        $qv->setVersion($entity->getVersion());
        $qv->setForm($entity->getForm());
        $qv->setDataTypes($entity->getDataTypes());
        $qv->setQuestion($entity->getQuestion());
        $em->persist($qv);
        $em->flush();
    }
}
```

Kuva 19. PostPersist-funktio

6 Tulokset

Työssä toteutuivat kaikki työlle määritellyt vaatimukset.

Jatkokysymysten toteuttamiseen käytettiin rekursiivista tietorakennetta QuestionCurrent-taulussa. Tämä toteutettiin tekemällä parent- ja children-muuttujat QuestionCurrent-entiteettiin. Doctrine muodosti näiden pohjalta QuestionCurrent-tauluun rekursiivisen rakenteen. Rekursiivisen rakenteen lukemiseen käytettiin PHP:n RecursiveIterator-luokkaa (kuva 20).

```
class RecursiveQuestionIterator  implements \RecursiveIterator{

    private $_data;

    public function __construct(Collection $data)
    {
        $this->_data = $data;
    }

    public function hasChildren()
    {
        return ( ! $this->_data->current()->getChildren()->isEmpty());
    }

    public function getChildren()
    {
        return new RecursiveQuestionIterator($this->_data->current()->getChildren());
    }

    public function current()
    {
        return $this->_data->current();
    }
```

Kuva 20. Ote RecursiveQuestionIterator-luokasta

Kysymysten versiohistorian toteutukseen käytettiin kahden taulun mallia siten, että ensimmäiseen taulun (QuestionCurrent) tietojen muuttuessa viimeisin tieto lisättiin toiseen tauluun (QuestionVersion). Ensimmäisessä taulussa säilytettiin vain viimeisintä tietoa ja toisessa taulussa säilyi koko versiohistoria. Tietojen muuttumista seurattiin kuuntelemalla tauluja vastaavien entiteettien käytön aikaisia elinkaaritapahtumia mukautetulla kuuntelijaluokalla. Kuuntelijaluokka vastasi versiotaulun päivittämisestä.

Kysymysten tietotyyppien piti olla dynaamisesti valittavissa. Tämä toteutettiin luomalla tietotyypeille oma taulu (DataType). Vastaustauluun (Answer) luotiin valittavissa olevia tietotyyppejä vastaavat kentät.

Monivalintakysymykset piti tallentaa omaan tauluun. Tämä toteutettiin luomalla QuestionOptions-taulu. Kysymykseen vastattaessa vastaustauluun tallennetaan valitun vastausvaihtoehdon id.

Tiedot piti tallentaa tietotyyppiä vastaavaan kenttään tai tauluun. Tämä toteutettiin siten, että vastaustauluun tehtiin tietotyyppejä vastaavat kentät. Omien taulujen luominen vastauksille olisi lisännyt turhaan tietomallin monimutkaisuutta.

Doctrinella piti toteuttaa entiteetti- ja palveluluokat. Tämä toteutui. Tämän lisäksi luotiin QuestionListener- ja RecursiveQuestionIterator-luokat.

Palveluihin piti toteuttaa kerrosrakenne. Tämän toteutuksessa otettiin mallia SOA:sta. Palvellut jaettiin kolmeen kerrokseen: apu-, entiteetti- ja tehtäväkerroksiin. Jokaiseen kerrokseen toteutettiin entiteettiluokkia vastaavat palveluluokat.

7 Pohdintaa

Opinnäytetyön aihetta valittaessa sain useita vaihtoehtoja, joista valitsin mielenkiintoisimman ja myös haasteellisimman aiheen. Katsoin myös, että ammatillisen kehityksen kannalta tämä aihe olisi selvästi hyödyllisin.

Työ vaati perehtymistä useisiin uusiin tekniikoihin ja teknologioihin. Tämän johdosta tiedonhakuprosessi kesti suunniteltua pidempään. Tiedonhaun sivutuotteena löysin toimeksiantajalle uusia sovelluksia ja foorumeita, joita voi hyödyntää tulevaisuudessa myös muissa projekteissa. Löysin Skipper-sovelluksen, joka muodostaa tietomallista valmiin PHP-koodin [14]. Lisäksi löysin http://ormcheatsheet.com/-sivuston, jolla kerrotaan, kuinka tietomallista muodostetaan entiteettiluokkia sekä millaisia annotaatioita entiteettien väliset liitokset vaativat.

Työtä voi jatkossa hyödyntää kyselylomakkeissa. Työn tuloksia voi myös hyödyntää muissa projekteissa, joissa tarvitaan esimerkiksi rekursiota, versiohistoriaa tai kuunnellaan Doctrinen elinkaaritapahtumia.

Lähteet

1. Laine, H. Tietomalli. Helsingin yliopisto. Tietotekniikanlaitos. Tietokantojen perusteet. Opetusmateriaali. 2003. https://www.cs.helsinki.fi/u/laine/tikape/k03/tietomalli.pdf. [Viitattu 16.5.2016.]
2. Smartdraw. Entity-relationship-diagram. 2016. https://www.smartdraw.com/entity-relationship-diagram/. [Viitattu 16.5.2016.]
3. Jonblock. Relational database versioning. 2013. http://blog.jondh.me.uk/2011/11/relational-database-versioning-strategies/. [Viitattu 11.5.2016.]
4. Advanced Kittenry – Tietokantasovellusohjeet. Arkkitehtuuri ja MVC. 2016. https://advancedkittenry.github.io/koodaaminen/arkkitehtuuri/index.html. [Viitattu 11.5.2016.]
5. Tuikka, T. MVC-arkkitehtuuri. Jyväskylän ammattikorkeakoulu. Tietojenkäsittely. Ohjelmoinnin projektiopinnot. Opetusmateriaali. 2014. http://batman.jamk.fi/~tuito/ohj_projop/main.htm#2_21. 11.1.5.2016
6. Doctrine Team. Welcome to the Doctrine Project. 2016. http://www.doctrine-project.org/. [Viitattu 11.5.2016.]
7. Doctrine. Introduction – Doctrine DBAL 2 Documentation. 2016. http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/introduction.html. [Viitattu 11.5.2016.]
8. Guzel, B. 6 CodeIgniter Hacks for the Masters. 2016. http://code.tutsplus.com/tutorials/6-codeigniter-hacks-for-the-masters--net-8308. [Viitattu 11.5.2016.]
9. Symfony. Databases and Doctrine. 2016. http://symfony.com/doc/current/book/doctrine.html. [Viitattu 11.5.2016.]
10. Doctrine. Doctrine Query Language. 2016. http://doctrine-orm.readthedocs.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html. [Viitattu 11.5.2016.]
11. Doctrine. Events – Doctrine 2 ORM Documentation. 2016. http://doctrine-orm.readthedocs.org/projects/doctrineorm/en/latest/reference/events.html. [Viitattu 11.5.2016.]
12. SOAPatterns.org. Service Layers. 2016. http://soapatterns.org/design_patterns/service_layers. [Viitattu 11.5.2016.]
13. Nieminen, A.H. Designing and realizing SOA. Tampereen teknillinen yliopisto. Tietotekniikanlaitos. Palvelupohjaiset järjestelmät. Opetusmateriaali. 2014. http://www.cs.tut.fi/~palpo/materiaali/luennot2014/luento3-2-soadesign.pdf. [Viitattu 11.5.2016.]
14. Skipper. Edit model and export CakePHP definitions. 2016. https://www.skipper18.com/en/frameworks/cakephp. [Viitattu 11.5.2016