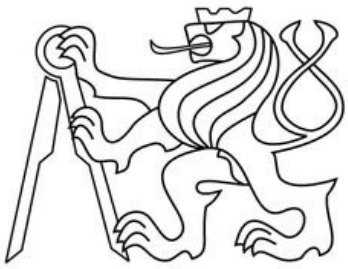
author email

Databáze EMS podacích lístků

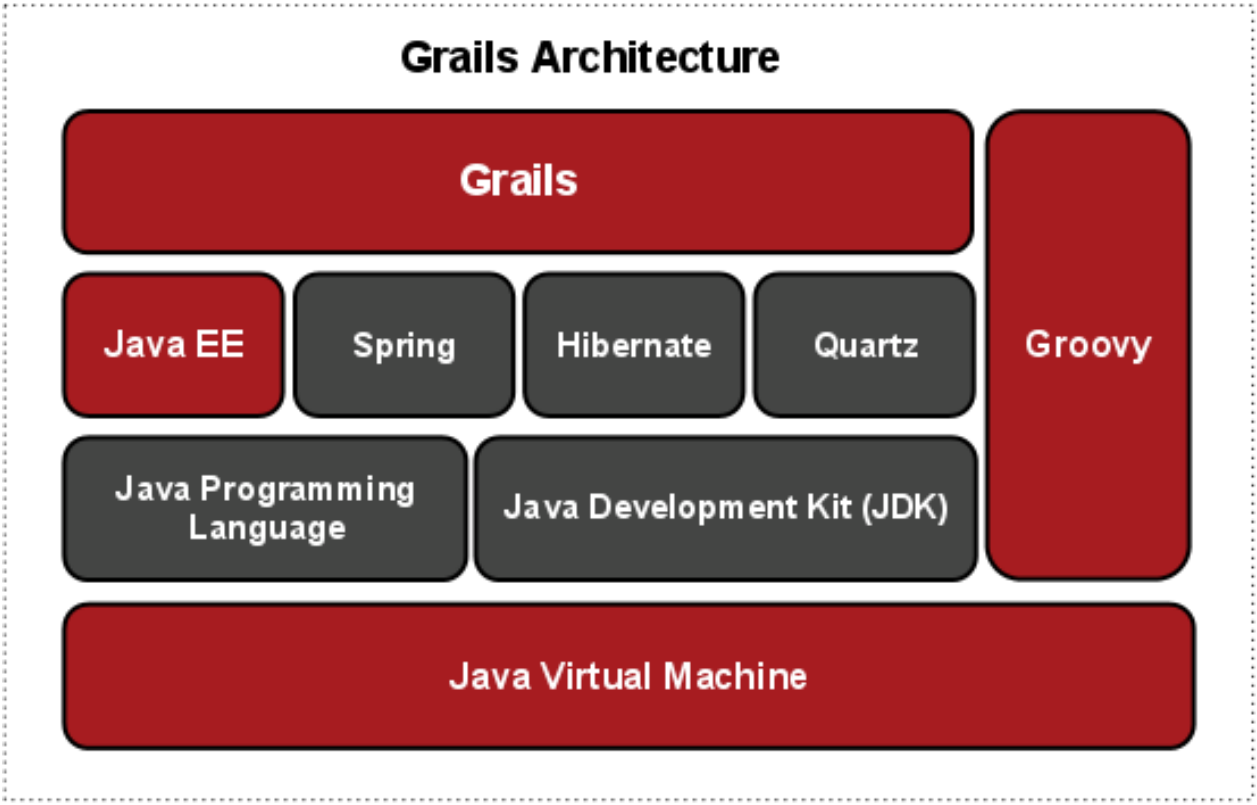
Semestrální práce



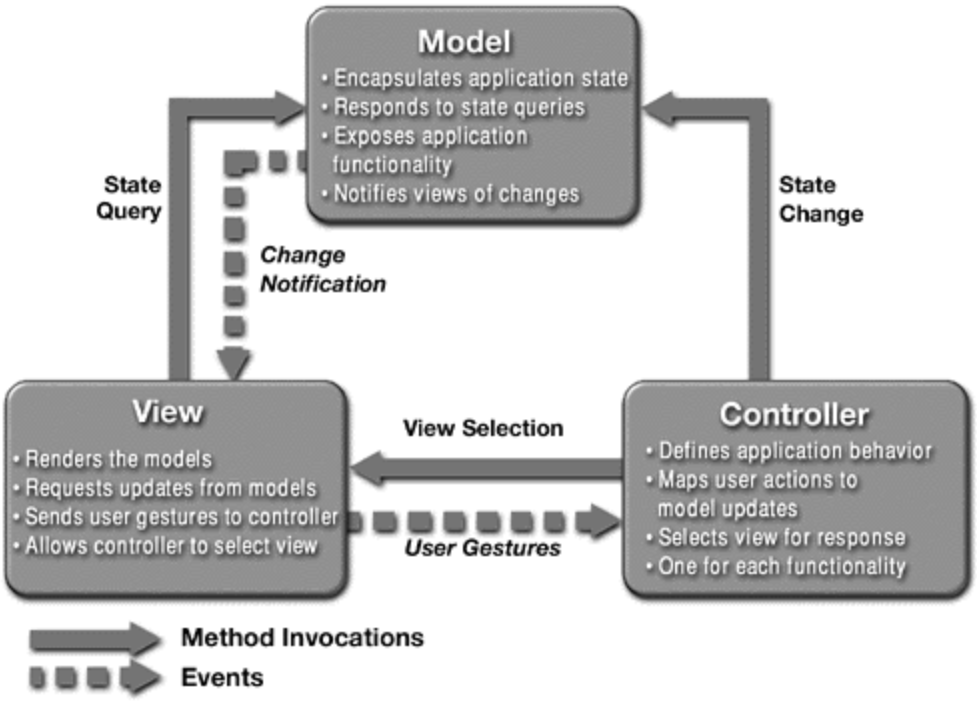
Obsah

Záměr projektu

Cílem semestrálního projektu byla tvorba webové aplikace Databáze EMS podacich lístků, pomocí dvou odlišných architektur. První architektura byla zvolena aplikace typu klient-server s pomocí frameworku Grails na operačním systému Ubutnu 12. V druhém případě se také jednalo o aplikaci typu klient-server, ale byl použit programovací jazyk PHP. Rozdíl spočívam v samotné struktuře. Aplikace implementovaná pomocí Grails je tří vrstvá a jedná se o enterprise aplikaci. Druhá varianta je naivní PHP aplikace, implementovaná do dvou vrstvé architektury. Na obr. č. 1 jsou vidět hlavní technologie použité frameworkem Grails a na obr. č. 2 je vidět obecný princip MVC architektury.

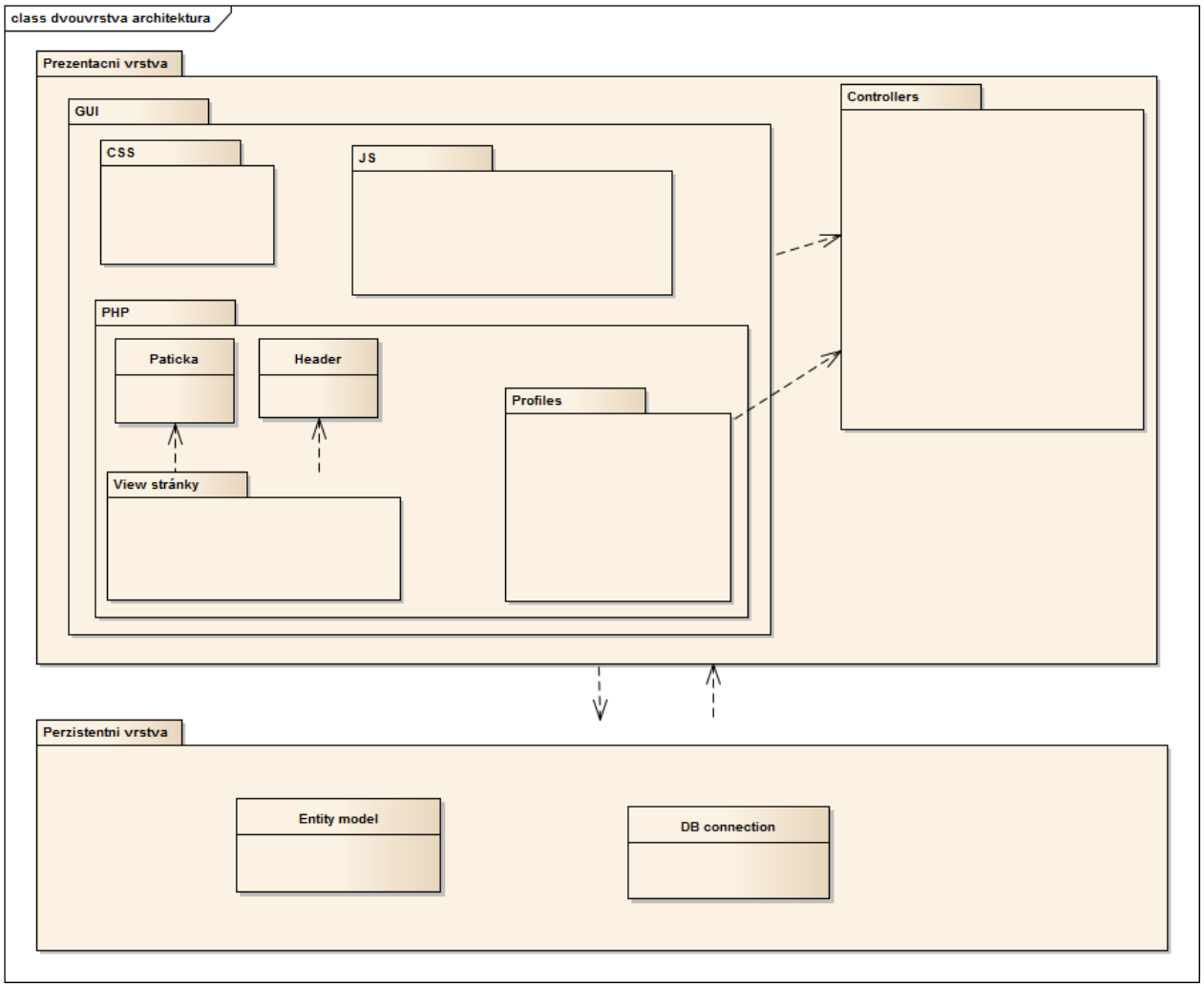


Obr. č. 1 – grails architektura



Obr. č. 2 – MVC architektura
(převzato z http://pratapreddypilaka.blogspot.cz/2011/09/view-controller-in-mvc.html)

Na obr. č. 3 je znázorněna dvouvrstvá architektura PHP aplikace. Jak je na obrázku vidět, je zde sloučena vrstva controller a view do prezentační. Ve vrstvě modelu naopak je navíc logika připojování do databáze.



Obr. č. 3 – dvouvrstvá architektura

Aplikace v obou případech je určena pro správu EMS podacích lístků. EMS neboli v překladu Express mail service je služba České pošty pro přepravu dokumentů a zboží do hmotnosti 20 kg po celém území České republiky. Mnoho firem tuto agendu nutně potřebuje.

Uživatele

Systém bude podporovat 3 typy uživatelů:

- host
- user
- admin

Host

Host se může pouze registrovat a přihlásit se.

User

Přihlášený uživatel již může vytvořit EMS záznam, upravit vlastní záznam, mazat vlastní záznam a nechat si vypsat všechny záznamy.

Admin

Admin může dělat totéž co přihlášený uživatel, ale editovat a mazat může jakýkoli záznam.

Katalog požadavků

Funkční požadavky

Následující katalog obsahuje funkční požadavky aplikace. Pro obě varianty je katalog stejný.

- Registrace
- Login
- Vytvořit záznam
- Upravit vlastní záznam
- Smazat vlastní záznam
- Výpis všech záznamů se stránkováním
- Upravit jakýkoli záznam
- Smazat jakýkoli záznam

Nefunkční požadavky

Následující katalog obsahuje nefunkční požadavky aplikace. Pro obě varianty je katalog stejný, krome posledního bodu, který je oddělený.

- Aplikace musí mít webové rozhraní optimalizované v prohlížečích Firefox, Chrome, Opera
- Aplikace by měla chránit svá data před nahráním nesprávných dat a to jak na straně aplikace tak na straně databáze
- “User-friendly” GUI
- Možnost rozšíření aplikace v budoucnu
- Uložení všech dat v centralizované databázi (MySQL)

Grails nefunkční požadavek

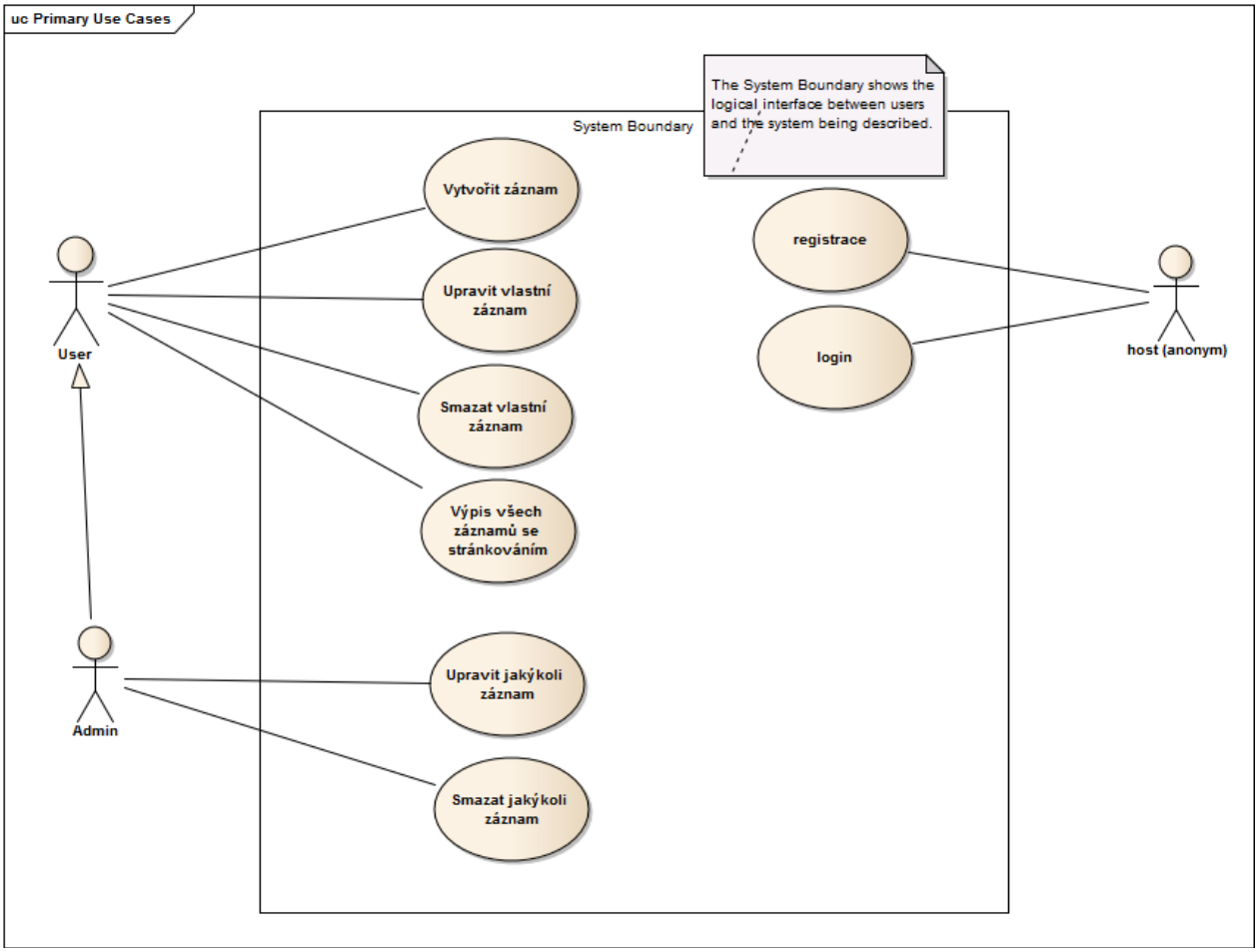
- Aplikace bude vyvíjena ve frameworku Grails (Java EE, Groovy, GORM...)
- View aplikace bude implementováno pomocí GSP (grails server pages)

PHP nefunkční požadavek

- Aplikace bude vyvíjena v PHP

Use case

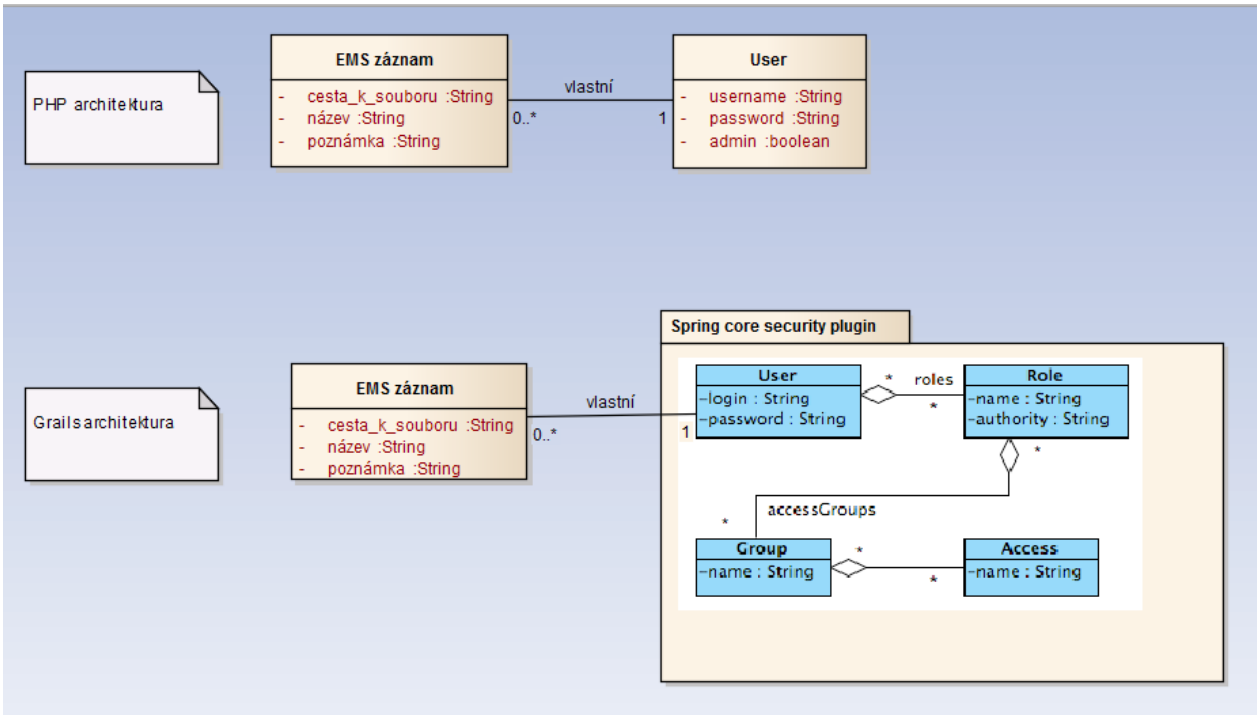
Na obr. č. 4 je use case diagram, který obsahuje jednotlivé případy užití aplikace.



Obr. č. 4 – use case diagram

Konceptuální datový model

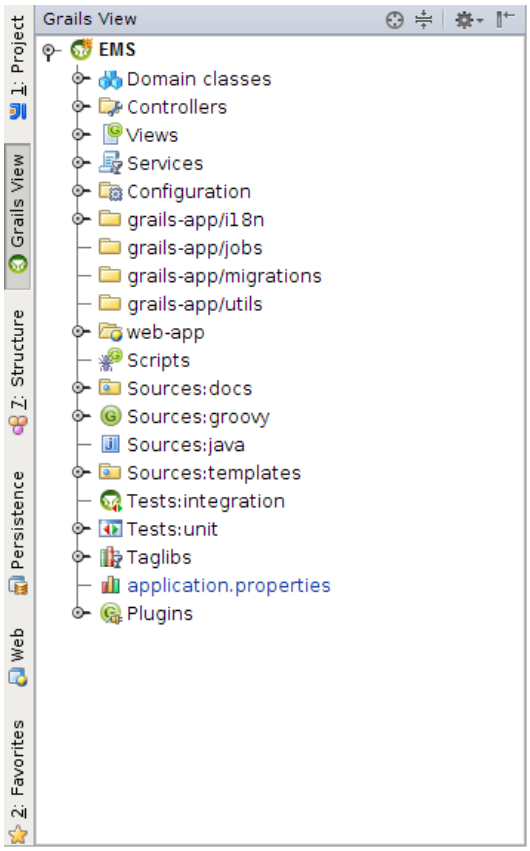
Tato část popisuje konceptuální datový model. Tento model v tomto projektu závisí na implementaci, protože v architektuře Grails je velmi výhodné používat pluginy. Ty šetří čas a již mají navrženou nejakou strukturu viz obr. č. 5. Daným pluginem je známý a velmi používaný plugin spring core security. Pro bezpečnost aplikace se také požívá pro jeho dobrou škálovatelnost. U PHP aplikace stačí pouze tabulka User, která obsahuje proměnnou boolean admin. Obecně budeme potřebovat entitu EMS záznamu a uživatele. Uživatele, ale potřebujeme nějak rozlišovat podle dvou rolí.



Obr. č. 5 – konceptuální datový model pro obě architektury

Architektura 1 - Grails

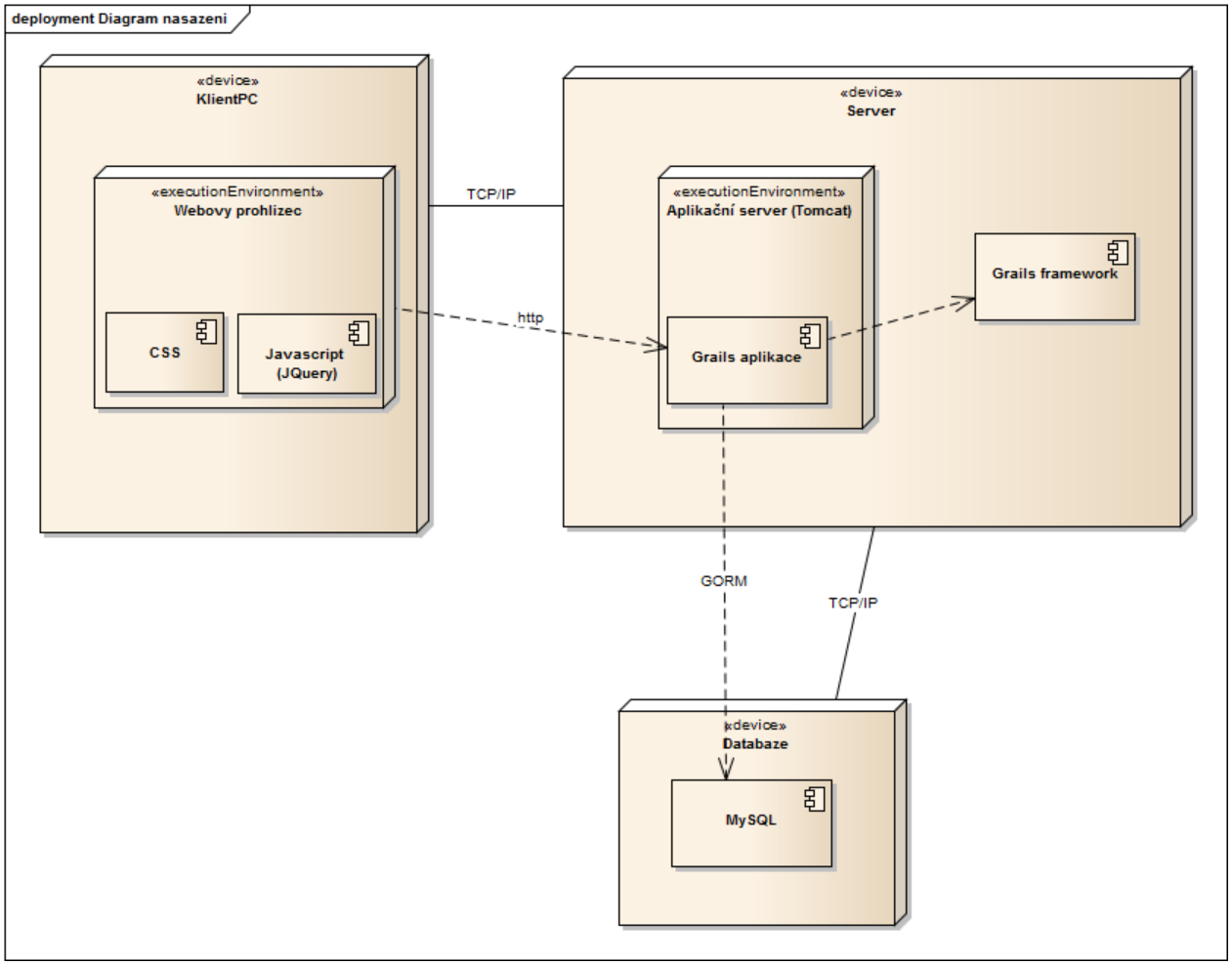
První architektura byla zvolena klient-server na frameworku Grails. Aplikační server byl zvolen Tomcat pro jeho jednoduší a rychlejší provoz než například JBoss nebo Glassfish. Na obr. č. 6 je vidět struktura projektu. Složka Domain classes obsahuje model aplikace, Controller složka obsahuje controllery, View složka obsahuje gsp soubory. Složka Services obsahuje servisy, které nebyly použity. Služba je vlastně třída které vykonává nejakou činnost a pak je volána v daném controlleru a tím zpřehledňuje kód controller, což je důležité. Poslední důležitou složkou je Configuration. Zde jsou všechny xml deskriptory projektu a různé jiné xml soubory s nastavením. Také zde najdeme například Boostrap, což je soubor ve kterém můžeme při spuštění aplikace vytvořit nějaká testovací data.



Obr. č. 6 – struktura projektu Grails

Diagram nasazení

Pro spuštění aplikace musí být na serverovém stroji nainstalována Java JDK 1.4 nebo vyšší, Grails framework a databáze MySQL. Dále je potřeba nastavit proměnné JAVA_HOME a GRAILS_HOME . Diagram nasazení je zobrazen na obr. č. 7.



Obr. č. 7 – diagram nasazení Grails aplikace

Ukázka kódu

Na obr. č. 8 je znázorněno nastavení databáze v Grails. Je zde možnost nastavení pro vývojové účely a zároveň poté pro účely nasazení (ty se zpravidla liší). Velmi důležitý řádek je url databáze co používáme, defaultní nastavení je H2 databáze, která se neukládá na disk. H2 se ukládá pouze do RAM a po vypnutí aplikace se vytvoří nová. Proto je nevhodná na produkční verzi a využívá se zpravidla jen pro vývoj.

DataSource.groovy

```
dataSource {
    pooled = true
    driverClassName = "org.h2.Driver"
    username = "sa"
    password = ""
}
hibernate {
    cache.use_second_level_cache = true
    cache.use_query_cache = false
    cache.region.factory_class = 'net.sf.ehcache.hibernate.EhCacheRegionFactory'
}
// environment specific settings
environments {
    development {
        dataSource {
            dbCreate = "create-drop" // one of 'create', 'create-drop', 'update', 'validate', ''
            url = "jdbc:h2:mem:devDb;MVCC=TRUE;LOCK_TIMEOUT=10000"
        }
    }
    test {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:mem:testDb;MVCC=TRUE;LOCK_TIMEOUT=10000"
        }
    }
    production {
        dataSource {
            dbCreate = "update"
            url = "jdbc:h2:prodDb;MVCC=TRUE;LOCK_TIMEOUT=10000"
            pooled = true
            properties {
               maxActive = -1
```

1 : 9 / 44 [1268] c 99 $0063 Text UNIX Kódování: UTF-8

Obr. č. 8 – nastavení databáze

Model v Grails aplikacích se skládá z doménových tříd. Taková třída je na obr. č. 9. Obsahuje nejen atributy, ale také tzv. contraints. Jedná se o omezení atributů v databázi. Nejen, že se nastaví sloupečky v databázi, ale také podle těchto omezení se pak provádí validace v controlleru (pomocí funkce validate()), kterou nemusíme dělat zbytečně ručně. Typu omezení existuje mnoho od maxSize po email. Na obr. č. 9 jsou omezení blank: false, což zamezuje aby EMS záznam měl prázdný string atribut a nullable: true, který povoluje hodnotu atributu na null.

```
package emscontrol

import java.util.Date;

class EMS {

    String kod
    String zeme
    Date datum
    Boolean smazany = false
    String path
    String name
    String saveName //date + id + extension

    static constraints = {
        kod blank:false
        zeme blank:false
        datum blank:false
        path nullable:true
        name blank:false
    }
}
```

Obr. č. 9 – model (doménová třída) EMS záznamu

Po modelu je potřeba vytvořit příslušné controllery a to většinou ke každé doménové třídě. Ukázka controlleru k EMS doménové třídě viz obr. č. 10. Zde přichází síla Grailsu ve spojení s některými IDE. Např. IDE IntelliJ IDEA od JetBrains dokáže vygenerovat jak základní controller k doménové třídě (s akcemi show, edit, create, save, update, delete) tak základní gsp soubory pro view aplikace. Na obr. č. 11 je podrobnější metoda save, která byla trochu komplikovanější.

```
package emscontrol

import java.util.Date;
import org.codehaus.groovy.grails.web.context.ServletContextHolder

import org.springframework.dao.DataIntegrityViolationException
import org.springframework.web.multipart.MultipartHttpServletRequest

class EMSController {

    static allowedMethods = [save: "POST", update: "POST"]//smazano , delete: "POST" kvuli linku delete z view

    def index() {
        redirect(action: "list", params: params)
    }

    def list() {
        params.max = Math.min(params.max ? params.int('max') : 10, 100)
        // query by example
        def example = new EMS(smazany: "false")
        [EMSInstanceList: EMS.findAll(example), EMSInstanceTotal: EMS.count()]

        //[EMSInstanceList: EMS.list(params), EMSInstanceTotal: EMS.count()]
    }

    def create() {
        [EMSInstance: new EMS(params)]
        //rozepsane
        //[EMSInstance: new EMS( kod: params.kod, zeme: params.zeme, datum: params.datum, smazany : false )]
    }
```

Obr. č. 10 – controller EMS záznamů

```
def save() {
    def f = request.getFile('myFile')
    if (f.empty) {
        flash.message = 'file cannot be empty'
        render(view: 'create')
        return
    }
    def EMSInstance = new EMS(params)
    EMSInstance.saveName = ''
    if (!EMSInstance.save(flush: true)) {
        render(view: "create", model: [EMSInstance: EMSInstance])
        return
    }
    flash.message = message(code: 'default.created.message', args: [message(code: 'EMS.label', default: 'EMS'), EMSInstance.id])
    def currentDate = new Date()
    def formattedDate = g.formatDate(date:currentDate, format: 'yyyy-MMM-dd')

    //zjisteni koncovky
    String extension = "";
    String fileName = f.originalFilename.toString()
    int i = fileName.lastIndexOf('.')
    if (i > 0) {
        extension = fileName.substring(i+1)
    }
    def servletContext = ServletContextHolder.servletContext
    EMSInstance.path = servletContext.getRealPath('testUkladani')
    EMSInstance.saveName = formattedDate + '_' + EMSInstance.id + '.' + extension//koncovka
    if (!EMSInstance.save(flush: true)) {
        render(view: "create", model: [EMSInstance: EMSInstance])
        return
    }
    f.transferTo(new File(EMSInstance.path+ File.separator +EMSInstance.saveName))
    redirect(action: "show", id: EMSInstance.id)
}
```

Obr. č. 11 – controller controller EMS záznamů (akce ukládání)

Poslední část je vytvořit view aplikace. Z pohledu architektury je to jen jedna část. Ale pro uživatele se jedná o tu nejdůležitější část. Pomocí IntelliJ IDEA lze vygenerovat nějaké základní rozhraní. Jedná se o gsp soubry k základním akcím z controlleru viz obr. č. 12, kde je edit formulář. Gsp soubory jsou napsány pomocí xhtml, css a mohou obsahovat i javascipt (např. známý Jquery), ale hlavně obsahují gsp tagy. Např. gsp tag datepicker je pouze na jednom řádku a vytvoří vše co potřebzujeme. V php verzi je potřeba toto vytvořit pomoc Jquery, což je kód navíc oproti této verzi. Soubory také obsahují expression jazyk, který se obaluje pomocí znaků ${ a }. Tyto nástroje pomáhají vytvářet logiku stránek, brát data z objektů bean, předávat parametry, vyrenderovávat i18n jazykové modifikace apod. Většina věci, které se dají napsat pomocí gsp tagů, se dají napsat i pomocí expression jazyka.

```
<%@ page import="emscontrol.EMS" %>


<div class="fieldcontain ${hasErrors(bean: EMSInstance, field: 'kod', 'error')} required">
    <label for="kod">
        <g:message code="EMS.kod.label" default="Kod" />
        <span class="required-indicator">*</span>
    </label>
    <g:textField name="kod" required="" value="${EMSInstance?.kod}"/>
</div>

<div class="fieldcontain ${hasErrors(bean: EMSInstance, field: 'zeme', 'error')} required">
    <label for="zeme">
        <g:message code="EMS.zeme.label" default="Zeme" />
        <span class="required-indicator">*</span>
    </label>
    <g:textField name="zeme" required="" value="${EMSInstance?.zeme}"/>
</div>

<div class="fieldcontain ${hasErrors(bean: EMSInstance, field: 'datum', 'error')} ">
    <label for="datum">
        <g:message code="EMS.datum.label" default="Datum" />

    </label>
    <g:datePicker name="datum" precision="day" value="${EMSInstance?.datum}" noSelection="['': '']" />
</div>

<div class="fieldcontain ${hasErrors(bean: EMSInstance, field: 'name', 'error')} ">
    <label for="name">
        <g:message code="EMS.name.label" default="Name" />

    </label>
```

Obr. č. 12 – gsp soubor reprezentující formulář editace záznamu

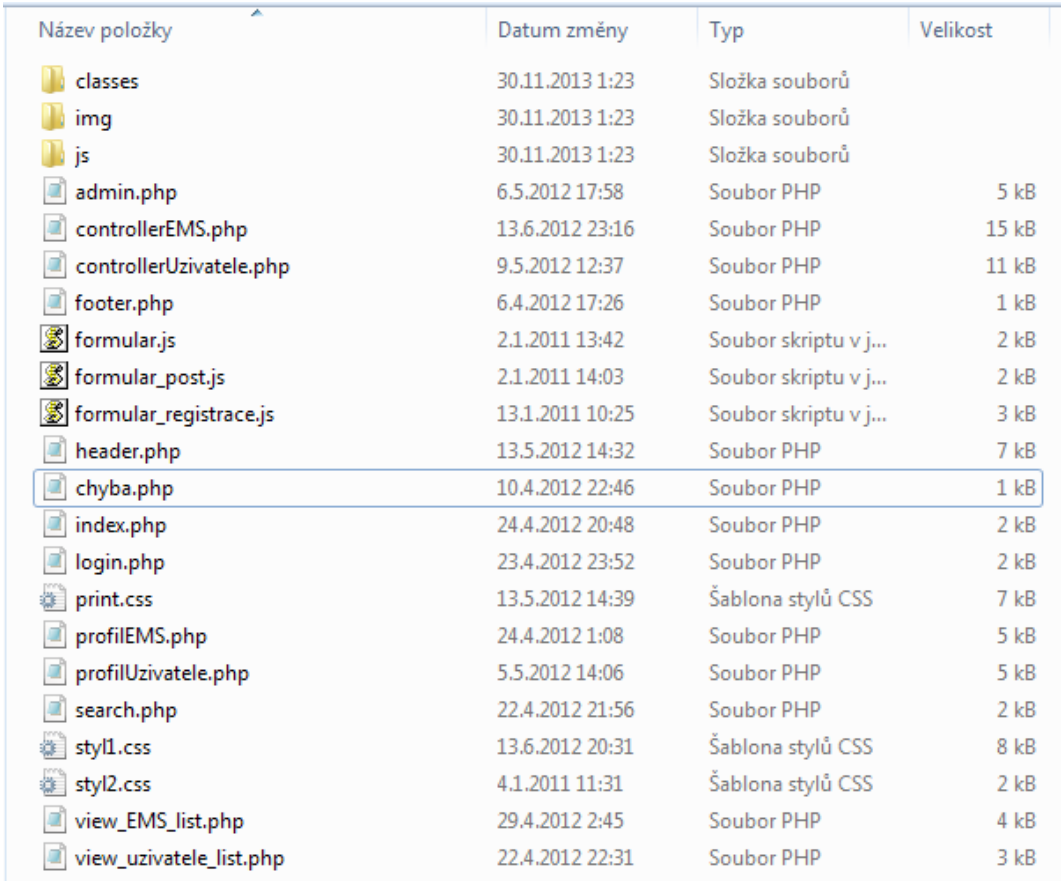
Na obr. č. 13 je kus gsp douboru, který se stará o bezpečnost aplikace. Využívá přímo plugin spring core security. Je vidět, že význam gsp tagů je velký. Bezpečnost aplikace samozdřejmě nestojí jen na straně view, ale také v controlleru se dají akce, ale i celé třídy zabezpečit pomocí anotací @Secured(role="ROLE_ADMIN"). Při složitějších nastavení zabezpečení například, kdybychom chtěli mít nejen role ale i skupiny oprávnění tak musíme spring core security nastavit v xml souboru v složce configuration.

```
<sec:ifLoggedIn>
    <div class="user-login-state" userWatchDogId="${sec.loggedInUserInfo(field: 'id')}" >

    <g:message args="[sec.loggedInUserInfo(field: 'fullName'), sec.loggedInUserInfo(field: 'username')]" code="user.loginState.label"/>
    <g:link controller="logout"> <g:message code="userLoginState.singOut.label" /> </g:link>

    </div>

</sec:ifLoggedIn>
<sec:ifNotLoggedIn>
    <div class="user-login-state">
    <g:message code="userLoginState.notLogged.label" />
    </div>
</sec:ifNotLoggedIn>
```

Obr. č. 13 – ukázka použití spring core security v gsp soubrou

Architektura 2 - PHP

Druhá architektura byla zvolena klient-server jako nativní aplikace PHP. Aplikační server byl zvolen Apache, jedná se o velmi rozšířenou kombinaci pro webové aplikace. Na obr. č. 14 je vidět struktura projektu. Oproti první architektuře byla aplikace vyvíjena v PSPadu. Snaha byla zachovávat OOP přístup proto je zde složka classes, kde se nachází používané třídy (jedná se o celou perzistentní vrstvu), složka img obsahuje obrázky, složka js obrahuje javascript. Ostatní složky jsou vedle a to jak php soubory tak css soubory a js soubory, které kontrolují vstupy.

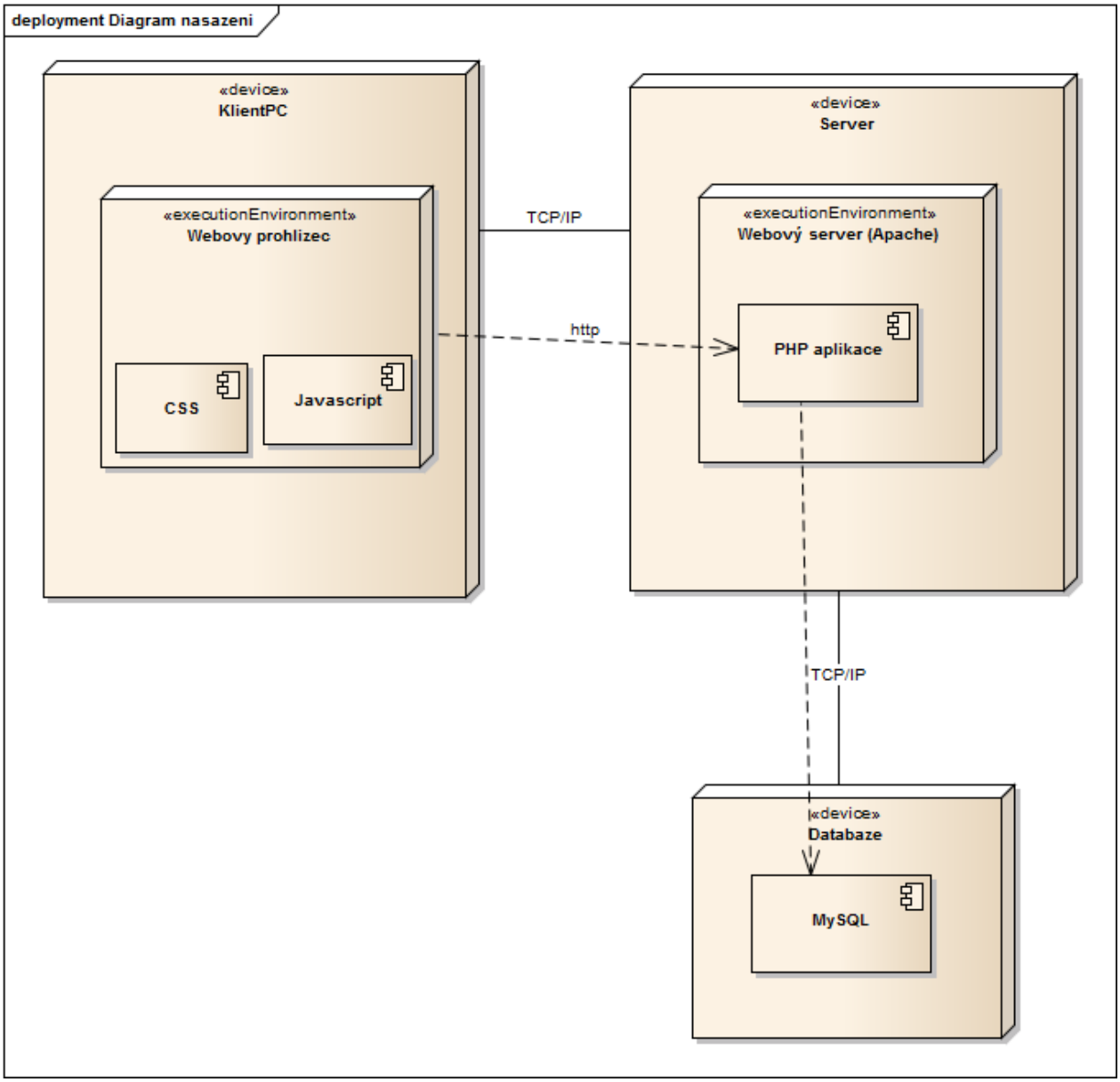


Název položky	Datum změny	Typ	Velikost
classes	30.11.2013 1:23	Složka souborů	
img	30.11.2013 1:23	Složka souborů	
js	30.11.2013 1:23	Složka souborů	
admin.php	6.5.2012 17:58	Soubor PHP	5 kB
controllerEMS.php	13.6.2012 23:16	Soubor PHP	15 kB
controllerUzivatele.php	9.5.2012 12:37	Soubor PHP	11 kB
footer.php	6.4.2012 17:26	Soubor PHP	1 kB
formular.js	2.1.2011 13:42	Soubor skriptu v j...	2 kB
formular_post.js	2.1.2011 14:03	Soubor skriptu v j...	2 kB
formular_registrace.js	13.1.2011 10:25	Soubor skriptu v j...	3 kB
header.php	13.5.2012 14:32	Soubor PHP	7 kB
chyba.php	10.4.2012 22:46	Soubor PHP	1 kB
index.php	24.4.2012 20:48	Soubor PHP	2 kB
login.php	23.4.2012 23:52	Soubor PHP	2 kB
print.css	13.5.2012 14:39	Šablona stylů CSS	7 kB
profilEMS.php	24.4.2012 1:08	Soubor PHP	5 kB
profilUzivatele.php	5.5.2012 14:06	Soubor PHP	5 kB
search.php	22.4.2012 21:56	Soubor PHP	2 kB
styl1.css	13.6.2012 20:31	Šablona stylů CSS	8 kB
styl2.css	4.1.2011 11:31	Šablona stylů CSS	2 kB
view_EMS_list.php	29.4.2012 2:45	Soubor PHP	4 kB
view_uzivatele_list.php	22.4.2012 22:31	Soubor PHP	3 kB

Obr. č. 14 – struktura projektu

Diagram nasazení

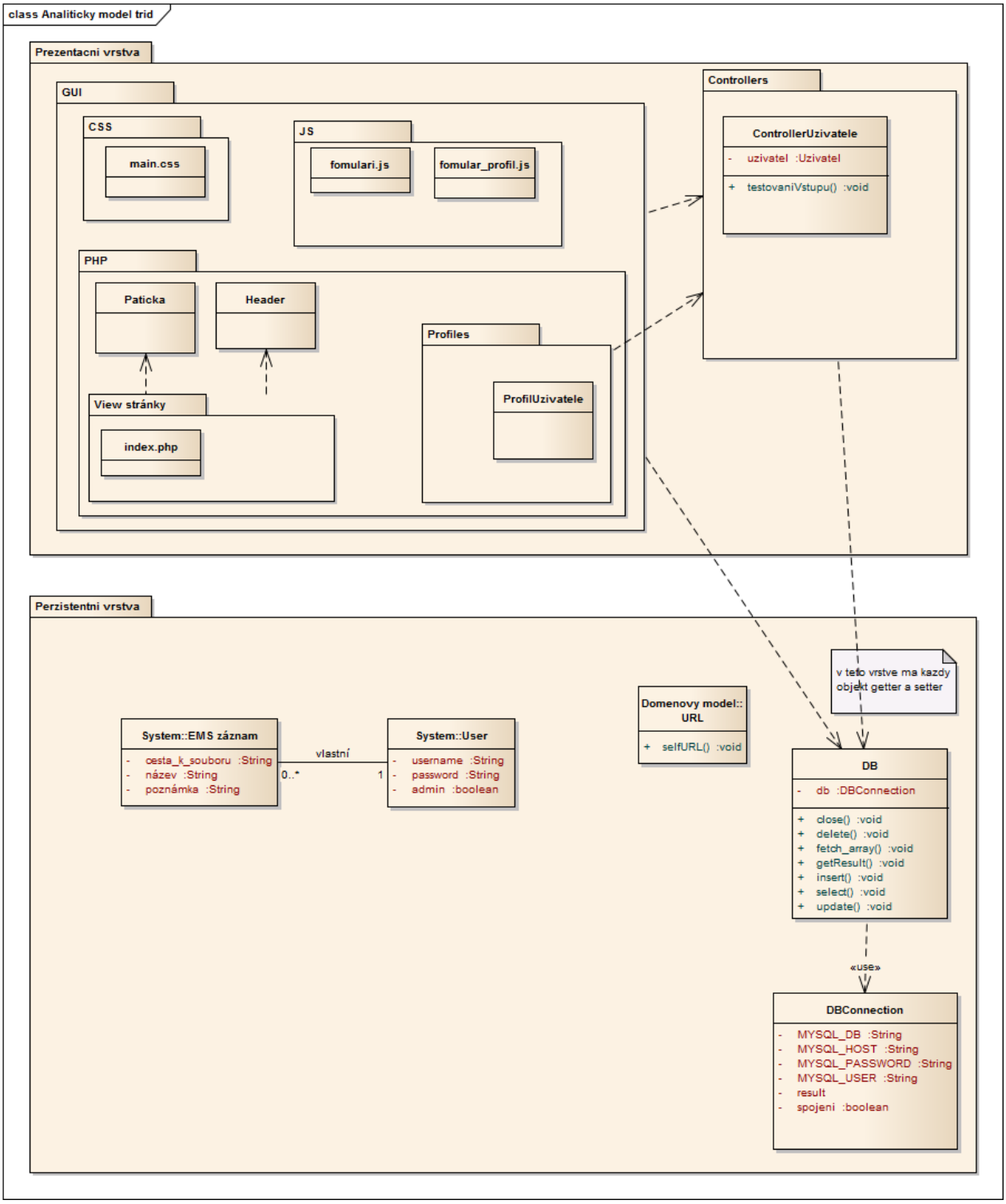
Pro spuštění aplikace musí být na serverovém stroji nainstalována Apache s MySQL. Existují distrubuce které obsahují vše jako například XAMP. A to je vše, není potřeba nastavovat zádné proměnné prostředí. Diagram nasazení je zobrazen na obr. č. 15.



Obr. č. 15 – diagram nasazení PHP aplikace

Analytický model tříd

PHP architektura se liší od Grails architektury. Zde byla použita dvouvrstvá architektura, ta obsahuje pouze prezentační a perzistentní vrstvu viz obr. č. 16. Vrstva controlleru se zde da doimplenetovat v rámci prezentační vrstvy a logiku přístupu do databáze se většinou implementuje přímo do perzistentní vrstvy. Další skupinou souboru jsous Profiles, který slouží jako view z controlleru. Podle session proměnné pozná jestli má zobrazit show, edit, create. V perzistentní vrstvě je ještějedna třída a to URL. Ta se stará o linkování na stránku, na které uživatel byl předtím než se dostal tam kde je. Jedná se o náhradu toho, aby uživatel stisknul stačítko zpět nahoře v prohlížeči. V takovém případě totiž ztratí všechna data co zadal do jakéhokoli formuláře. Bohužel v PHP nemáme beany.



Obr. č. 16 – analytický model tříd

Ukázka kódu

Na obr. č. 17 je znázorněna třída DBConnection, která reprezentuje spojení s databází. Technologie byla použita mysql. Jedná se o poměrně starou technologii. Existují i lepší jako mysqli nebo prepared statements. Mysql bylo použito kvůli jednoduchosti. Tato třída má na starosti nejnižší komunikaci s databází a je využita třídou DB.

```php
<?php
/**
* Třída DBConnection
*
*  Představuje přímou reprezentaci spojení s databází. Použité jsou funkce MySQL. Implementuje rozhraní DBLayer.
*  @author Jiří Šebek
*  @project EMS databáze
*/
require 'DBLayer.php';
class DBConnection implements DBLayer{

private $spojeni;
private $result;
private $MYSQL_HOST =[redacted];
private $MYSQL_USER =[redacted];
private $MYSQL_PASSWORD =[redacted];
private $MYSQL_DB = [redacted]
  /**
  * Kontruktor třídy DBConnection nastavuje proměnné spojení a result na false.
  */
public function __construct(){
          $this->spojeni = false;
          $this->result = false;
}

public function connect(){
        $this->spojeni = mysql_connect($this->MYSQL_HOST, $this->MYSQL_USER, $this->MYSQL_PASSWORD) or die ('Nemohu se připojit. Zkontrolujte prosím připojení k
}

public function selectDB(){
        mysql_select_db($this->MYSQL_DB, $this->spojeni) or die(mysql_error($this->spojeni));
}

public function query($sql){
        $this->result = mysql_query($sql, $this->spojeni)or die(mysql_error($this->spojeni));
}
```

Obr. č. 17 – třída pro připojení do databáze

Třída DB již oproti DBConnection reprezentuje třídu se specifickými dotazy jako select, select s limitou, insert apod viz obr. č. 18. Oproti grails projektu jsou obě tyto třídy nadbytečné a zdržují vývojáře od programování důležitějších částí. Na druhou stranu je to nezbytnost, protože psát poté všude v kódu připojení k databázi proceduralně by bylo redundantní a zvětšovalo by to objem kódu.

```
class DB {
private $db;
function __construct(){
      $this->db = new DBConnection();
      $this->db->connect();
      $this->db->selectDB();
    }
   public function getSpojeni(){
        return $this->db->getSpojeni();
   }
   function close(){
      $this->db->close();
    }
    function getResult(){
      return $this->db->getResult();
    }

    public function insert($kam, $co, $hodnoty){
          $sql = 'INSERT INTO '.$kam
                    .'('.$co.')
                VALUES ('.
                   $hodnoty .' )';
     $this->db->query($sql);
    }
    public function select($co, $odkud, $podminka){
           $sql = 'SELECT '.
                     $co
                . ' FROM '.
                     $odkud
                .' WHERE '.
                     $podminka;
     $this->db->query($sql);
    }
    public function selectLimit($co, $odkud, $podminka, $limit){
           $sql = 'SELECT '.
                     $co
```

Obr. č. 18 – třída obsluhující dotazy do databáze

Na obr. č. 19 je již třída Uživatele. Jedná se stále o třídy, které patří do perzistentní vrstvy. Třída obsahuje atributy entity, dané gettery a settery. V kontruktoru je vidět trik jak se v PHP řeší přetížení metod. Přetížení jako takové zde není a proto se v parametru metody dají hodnoty NULL. Když parametr poté nezadáme, vyplní se NULL. Jiné řešení pro PHP zatím není.

```php
<?php
class Uzivatel {

private $IDU;
private $uzivatelskeJmeno;
private $heslo;
private $role;

   function __construct($uzivatelskeJmeno = NULL, $heslo = NULL, $role = NULL){
         $this->uzivatelskeJmeno=$uzivatelskeJmeno;
         $this->heslo=$heslo;
         $this->role=$role;
   }

   //settery
   function setID($IDU){
         $this->IDU = $IDU;
   }
   function setUzivatelskeJmeno($uzivatelskeJmeno){
         $this->uzivatelskeJmeno = $uzivatelskeJmeno;
   }
   function setHeslo($heslo){
         $this->heslo = $heslo;
   }

   function setRole($role){
         $this->role = $role;
   }

   //gettery
   function getID(){
         return $this->IDU;
```

Obr. č. 19 – třída modelu uživatele

Dalším krokem je vytvořit controllery k modelům podobně jako v architektuře 1. Udržování stavu a předávání informací je řešeno pomocí session viz obr. č. 20. Na obr. č. 20 je jen kus kódu, ale je vidět, že kód při horším navržení může být nepřehledný a je pracnější. Je zde vidět snaha o podobný systém jako v Grails, ale místo samostatných akcí (metod) se zde kontroluje stav session proměnné „action“.

```php
<?php
/**
 * Skrip na zpracovani zmen s uctem od prihlaseni po upravu.
 */
session_start();
require 'classes/DB.php';
require 'classes/EMS.php';
require 'classes/Uzivatel.php';
$db = new DB();

if (isset($_SESSION['upozorneni'])) unset($_SESSION['upozorneni']);

if (isset($_SESSION['uzivatel'])) {
  $uzivatel = unserialize($_SESSION['uzivatel']);
} else {
  $uzivatel = null;
}
if ( isset($_REQUEST['action']) && ($_REQUEST['action'] == 'Přidat' )) {
        if ($uzivatel->getRole()==1) {
            if (isset($_POST['kod']) && isset($_POST['zeme']) && isset($_POST['datum']) )
              {

          $current_image=$_FILES['image']['name'];
          $typ = substr(strrchr($current_image, '.'), 1);
          if (!(($typ == "pdf") ))
          {
             $_SESSION['upozorneni']="Špatný formát souboru!";
             header("Location: admin.php");
             exit();
          }
```

Obr. č. 20 – EMS controller

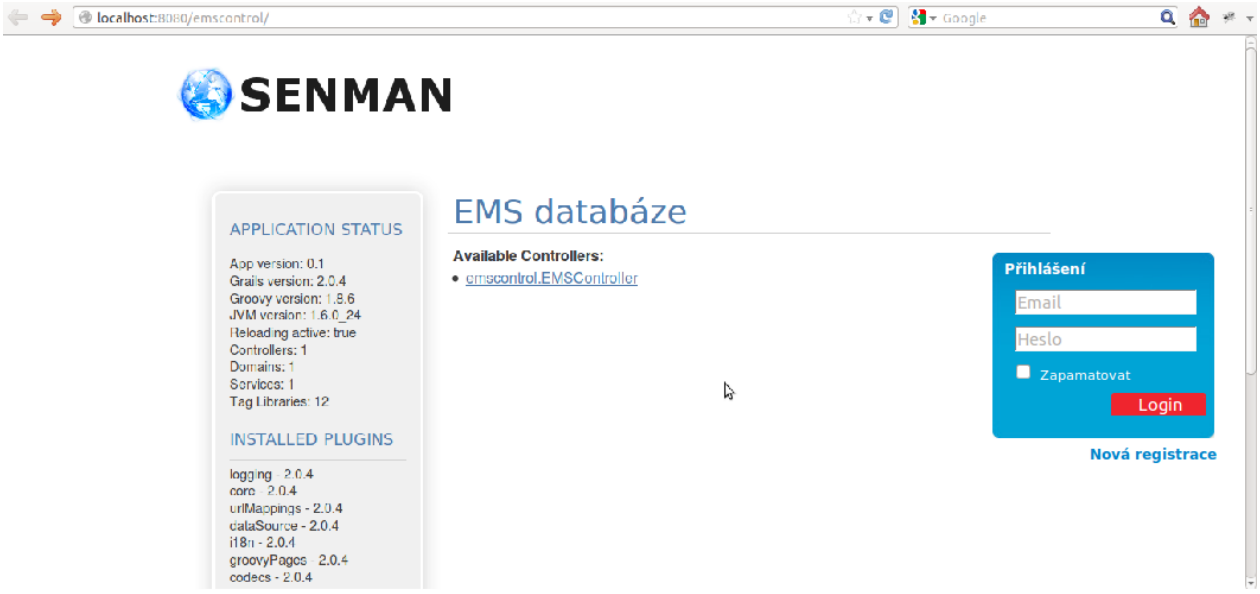
View část mě přisla z tohoto projektu jako nejméně přehledná. Zde vidím velký problém PHP. Míchání PHP kódu s výpisama html kódu a čístým html kódem mezi tím může mást. Na obr. č. 21 je kus kódu pro fomulář pro přidání EMS záznamu. Dále také je potřeba všude testovat přítomnost proměnných pomocí funkce isset(). Když na to vývojář zapomene a aplikace nedostane proměnnou, tak aplikace spadne.

```php
<?php
/**
 * Skrip na zpracovani zmen s uctem od prihlaseni po upravu.
 */


require_once 'header.php';

echo '<div id="hlavni">';
if ( isset($_REQUEST['action']) && ($_REQUEST['action'] == 'Přidat' )) {
 echo '<div class="shadow">' . $_REQUEST['action'] . '<hr/></div> ';
   ?>
 <form method="post" action="controllerZbozi.php" name="formular" onsubmit="return kontrola(formular)" enctype="multipart/form-data">

  <label for="kod">Kód:</label><br>
    <input type="text" name="kod" id="kod" maxlength="100" value="<?php if(isset($nazev)) {echo $kod; }?>"/></p>
  <label for="zeme">Země:</label><br>
    <input type="text" name="zeme" id="zeme" maxlength="100" value="<?php if(isset($nazev)) {echo $zeme; }?>"/></p>
  <label for="datum">Datum:</label><br>
    <input type="text" name="datum"  id="SelectedDate" readonly onClick="GetDate(this);" /></p>


   <input type="hidden" id="MAX_FILE_SIZE" name="MAX_FILE_SIZE" value="300000">
    <label for="image">File:</label><br>
   Poporované formáty souboru jsou PDF<br>
   maxim. povolený velikost 300 MB <br>

   <input type="file" name="file"><br>
```

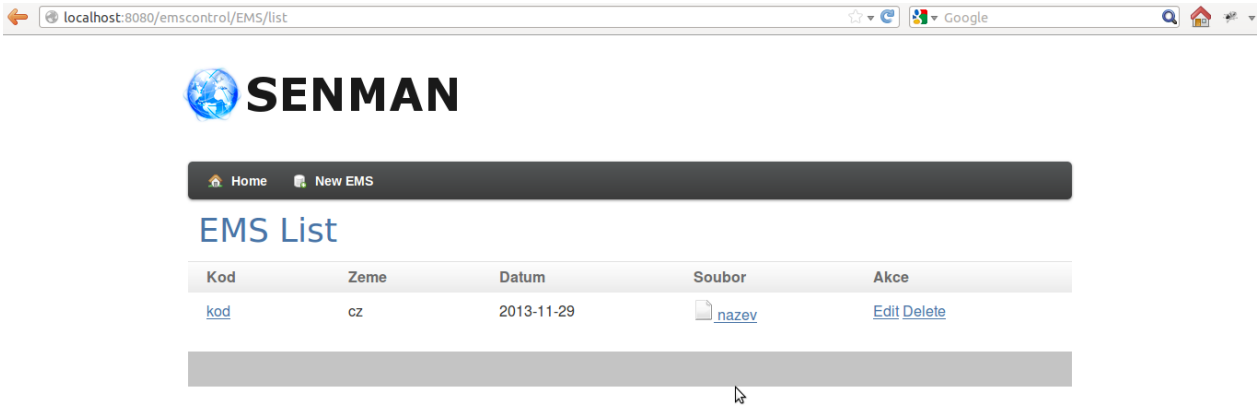
Obr. č. 21 – view formulář

Ukázka GUI

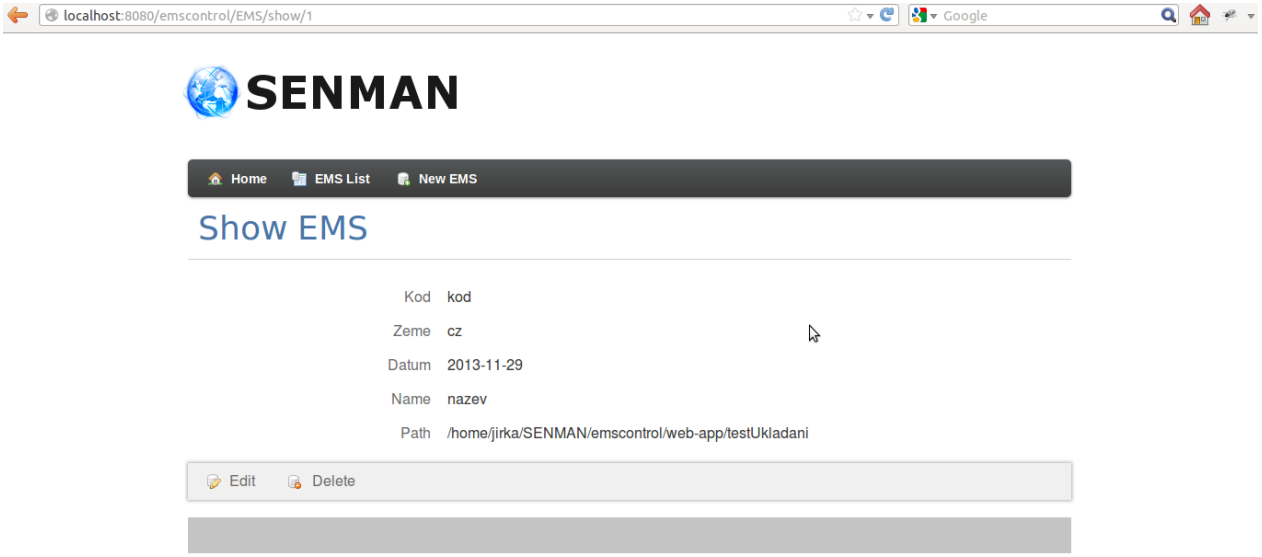
GUI aplikace je v obou případech stejné, protože se jednou vytvořil vzhled a poté se css styly a stejné prvky použily i v druhé architektuře. Velmi dobrou pomoc představoval pro vzhled framework Grails, který právě základní šablonu vygeneruje a ta se poté může modifikovat viz obr. 22 – 26, kde je konečná podoba aplikace.



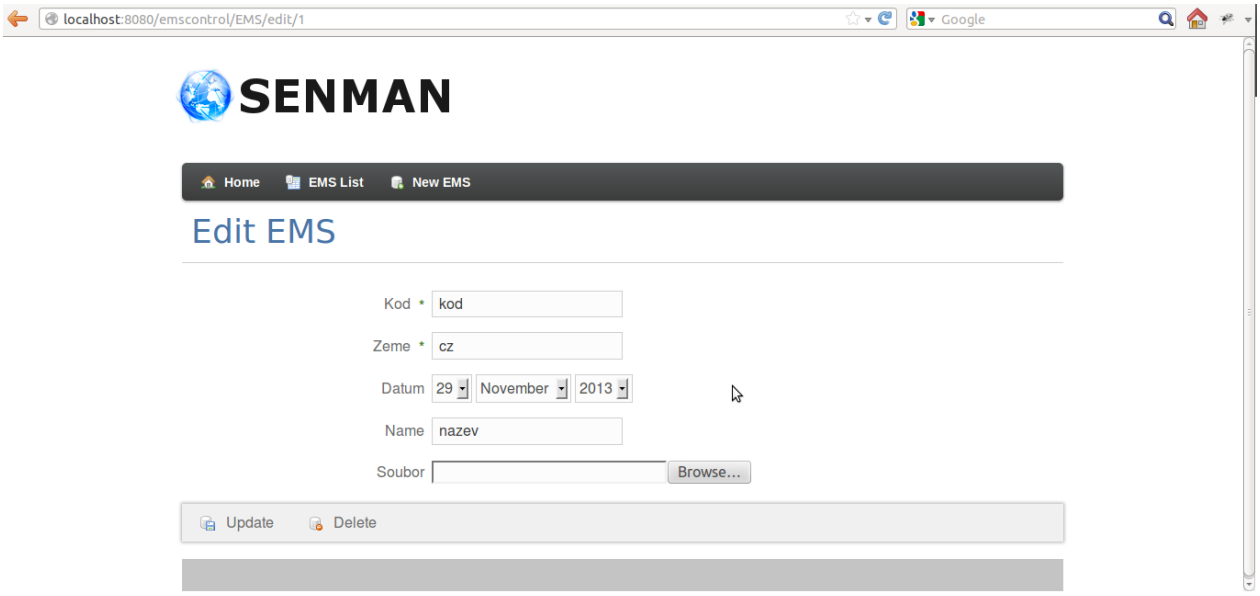
Obr. č. 22 – úvodní obrazovka



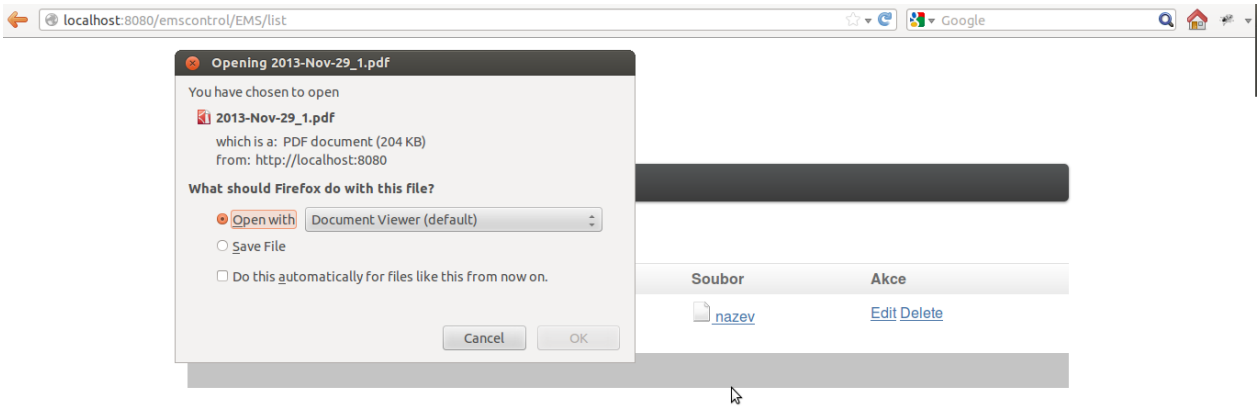
Obr. č. 23 – vylistování jednotlivých EMS záznamů



Obr. č. 24 – zobrazení jednoho EMS záznamů



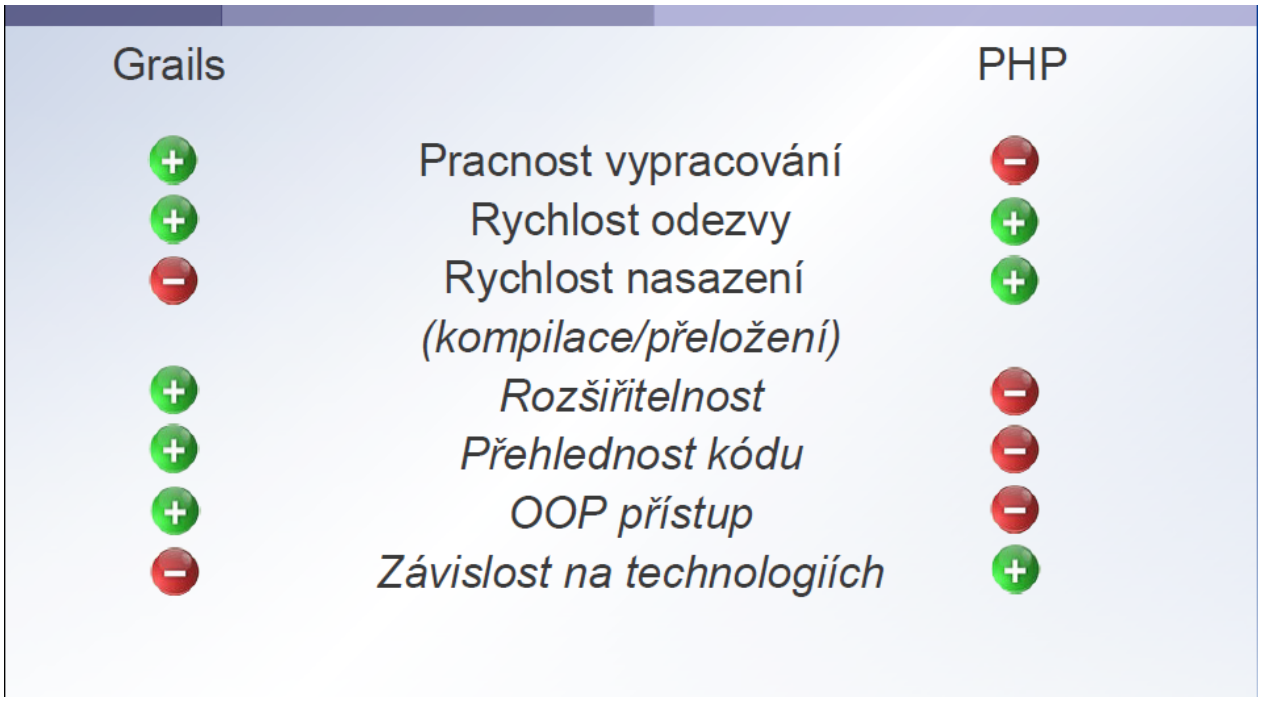
Obr. č. 25 – editace jednoho záznamu



Obr. č. 26 – funkční odkaz na EMS pdf soubor

Porovnání obou přítupů

Na obr. č. 27 je přehledně zobrazeno porovnání obou přístupů. Výsledek mě nijak nepřekvapil. Obecně se dá říci, že když vývojář začíná v Javou EE v jakékoli podobě, tak se jedná o velmi pomalý vývoj. Jak se jeho znalosti, ale zdokonalují a ví kde co má nastavit, tak vývoj naopak extrémně rychlejší než v PHP aplikaci. Framework grails ulehčuje ve všech částech programátorovi práci. V PHP architektuře je sice vždy jasné co máme udělat, ale kvantita kódu je větší. Z obr. č. 27 je ale vidět, že takto malá aplikace měla stejnou rychlost odezvy jak v PHP tak v Grails verzi. V bodě nasazení bylo php lepší a to také v závislosti na technologiích. Když totiž tvůrci Grails, springs a jiných technologií co používáme prostě změní danou službu co používáme jako black box, tak nám dále s velkou pravděpodobností nepudou. Z vlastní zkušenosti se to často nestává a člověk v takovém případě může zůstat na stávající verzi frameworku a nemusí ho to nijak trápit, ale přesto jsem to zahrnul jako mínus pro Grails.



Obr. č. 27 – Porovnání obou architektur

Závěr

Tato semestrální práce měla za cíl navrhnout a naimplementovat dvě aplikace. Tyto aplikace měly být funkčně stejné, ale při tom naimpementovány pomocí jiných architektur. První architektura byla enterprise aplikace implementovaná pomocí frameworku Grails. Druhá architektura byla naivni PHP aplikace, která využívala dvouvrstvou architekturu. Obě zvolené architektury se podařilo naimplementovat. Cílem práce také bylo vytvoření této dokumentace. Tato práce mi pomohla uvědomit si rozdíly mezi jednotlivými technikami programování a také že návrhář musí přemýšlet dopředu, než danou aplikaci implementuje, protože když pak přijde na velký problém tak už ho většinou nevyřeší.