Comenius University in Bratislava
Faculty of mathematics, physics and informatics

Assignments Workflow in Learning Management System

Diploma thesis

2016

Bc. Tomáš Trungel

Comenius University in Bratislava
Faculty of mathematics, physics and informatics

Assignments Workflow in Learning Management System

Diploma thesis

Study programme:	Applied Informatics
Field of study:	2511 Applied Informatics
Department:	Department of Applied Informatics
Supervisor:	RNDr. Martin Homola, PhD.
Consultant:	doc. RNDr. Zuzana Kubincová, PhD.

Bratislava, 2016
Bc. Tomáš Trungel

39605477

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Meno a priezvisko študenta:	Bc. Tomáš Trungel
Študijný program:	Applied Computer Science (Single degree study, master II. deg., full time form)
Študijný odbor:	9.2.9. Applied Informatics
Typ záverečnej práce:	Diploma Thesis
Jazyk záverečnej práce:	English
Sekundárny jazyk:	Slovak

Title: Assignments Workflow in Learning Management System

Abstrakt:

Dátum odovzdania:

....
Študent

By this I declare that I wrote this master thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.

. . . .

Acknowledgment: I would like to say thank You to my master thesis supervisor Martin Homola for his help and professional and valuable advices while I was writing this thesis.

Abstract

English abstract: TODO

Keywords: one, two, three

Abstrakt

Slovenský abstrakt: TODO

Kľúčové slová: jedno, druhé, tretie

Contents

Listings

List of Figures

Introduction

TODO

Part I

Background

Chapter 1

MVC and HMVC design patterns

For software to be usefult, it has to interact with something. This could be another computer or process, or the interaction could be with people. So, there are interfaces. In modern applications, more effort often goes into development of usable interface than in other tasks. On the other side, our software needs to acces and store its data and perform operations on them. Purpose of this chapter is, at first, to explain MVC design patter, which is a simple and widespread way for solving this problem, then we will focus on extension of this pattern, HMVC. Both patterns are crucial to understand for purposes of this work.

1.1 Model-View-Controller

In an Object oriented programming (OOP) programmers often run into problems with application complexity and maintanance of code. As application becomes larger and more robust, it is ofter harder to add or modify existing features. Modifying of complex classes and unclear implementation can cause a lot of software bugs. On the other way, design patterns aim to provide reusable soulutions of common problems in software engineering and reduce this complexity by splitting functionality into specific classes with a single purpose. One of the most important desing patterns is Model View-Controller, which is the performs as the core of the CodeIgniter framework, which we will discuss later.

Model-View-Controller (MVC) design pattern was first created in 70's by SmallTalk programmers. Since that time, MVC design idiom has become a commonplace, especially in object oriented systems [3]. Motto of this pattern is specified as to "Stop data, program logic and presentation layers being mixed up together".

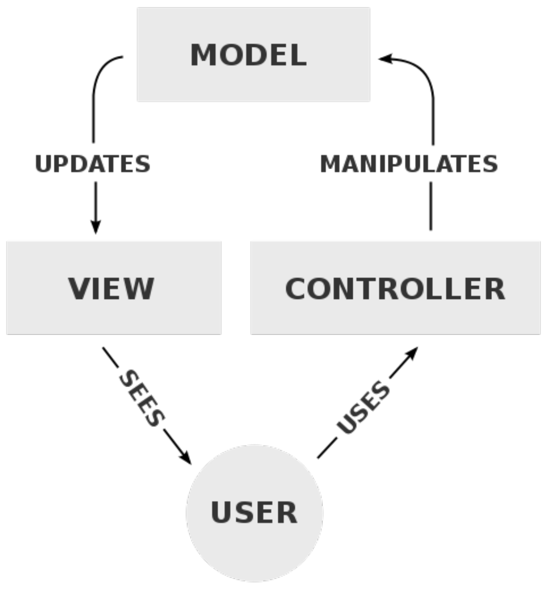


Figure 1.1: Model-View-Controller

To do so, MVC has to split application functionality into three layers: models, views and controllers. Each part plays specific, distinctive, non-overlapping function. This also enables programmers to easily replace these layers and extend functionality. Now, let's look at each layer.

1.1.1 Model

The model is where all the business logic of an application is kept [9]. This can mean anything from usage of third-party services via APIs to database access in order to fulfill applications business requirements. If applications needs to read or write any data, all the operations must be performed in this layer. For purposes of this work, models will mostly contain CRUD (Create-Read-Update-Write) operations with Courses database tables in PosgreSQL.

1.1.2 View

The view, as opposed to model, is where all of the user interface elements are kept [9]. Very important rule is that models are just templates without any logic whatsoever. For purposes of our work, views will be only PHP templates, which directly generates HTML, CSS and JavaScript responses.

1.1.3 Controller

And finally, the controller is the component that connects models and views together [9]. Over-simplifying a bit, controller knows all the enviroment settings (operating system, display resolution, ...) and according to these loads data from models, performs operations with them and then the results are sent to a view. Controller knows the views but views does not. The same applies to models.

1.2 HMVC design pattern

Using MVC design pattern is excellent for building small applications. Hovewer, huge application building on pure MVC pattern brings also some disadvantages [2]. For example, large sets of views, models and controllers could be a point of confusion and failure for many programmers. In case of larger projects, it is very needed to bring order to MVC. Solution to this problem can be Hierarchical Model-View-Controller design pattern (HMVC) [7].

Hierarchical Model-View-Controller design pattern (HMVC) is a direct extension of the MVC pattern that aims to solve mainly scalability issues mensioned earlier. HMVC was first described in a blog post entitled HMVC: The layered pattern for developing strong client tiers on the JavaWorld web site in July 2000 [7]. In summary, HMVC is a collection of MVC triads operating as one application. Each of this triads is treated as individual and can be rendered on it's own. This resolves into greater scalability, code reusability and most importantly, resolves the problem with hudge views in large projects.

To successfuly implement HMVC, it is crucial for application to be broken into systems [7]. Each system gets it's own views, models and controllers and is fully in control of it's own part and nothing else. In Courses 2 system, these systems are called modules, which we will discuss later. One of the advantages of using HMVC design pattern is, that we can focus on one module without changing implementation in others.

As understanding of design patterns is needed but not considered main focus of this work, we leave any further explanations to Jakub Culik's work [2].

Chapter 2

CodeIgniter framework

CodeIgniter is an Application Development Framework - a toolkit - for people who build web sites using PHP [5]. One of it's goals is to provide exceptional performance of web applications along with minimum complexity and fast development. CodeIgniter was born from ExpressionEngine [4] in 2006 and since then, it increasingly gained popularity among PHP developers. In 2008 CodeIginiter became industry leader in an enviroment saturated with PHP frameworks [4] and in 2014 CodeIgniter v 2.2 was released.

This framework is licensed under custom Elislab license, which is basicaly an open source license with only a few restrictions [6]. We provide a copy of this license in medium attached to this work.

In this chapter, we will present a short guide of CodeIgniter request and response processing. Full user-guide and documentation can be found on CodeIgniter official website [5].

2.1 Request and response flow

To understand how CodeIgniter works, let's look at processing of HTTP requests in this framework. CodeIgniter uses Apache2 module `mod rewrite`. This allows to be firstly processed `index.php` file as seen on image 2.1. Purpose of this script is to load configuration of whole application, load routes, cache and other classes.

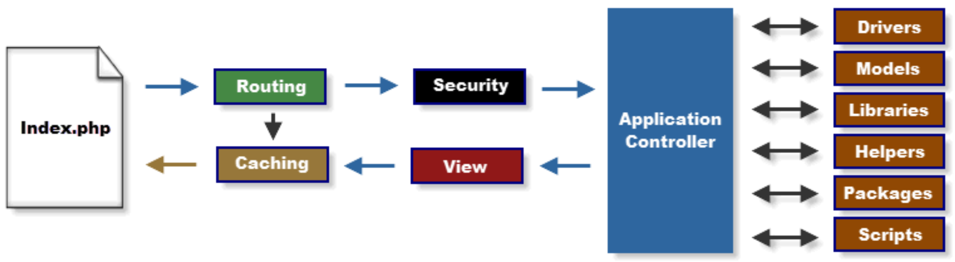


Figure 2.1: CodeIgniter request flow

It is provided by CodeIgniter framework and should not be changed (only for configuration purposes).

To process HTTP request, router must decide correct operation over it. If a cache file exists, it is sent directly to the browser, bypassing the normal system execution [5]. Before the application controller is loaded, the HTTP request and any user submitted data is filtered for security.

In the next step, routing takes place. Each module of application must have it's own routes defined as follows:

Listing 2.1: CodeIgniter routing

```
$route[course/(:num)] = "courses/$1";
```

The application takes list of all routes and applies the one of which regexp pattern matches the request URL. After this operation, proper controller (that must extend `CI_ CONTROLLER` class) is selected, instanciated and method `run()` of this controller is run.

This controller loads data from models, can perform some operations over them and then data are sent directly to view. The finalized View is rendered then sent to the web browser to be seen. If caching is enabled, the view is cached first so that on subsequent requests it can be served instantly.

2.2 Configuration and loading

CodeIgniter also adds support configuration classes. These classes are located in `application/config` folder with possibility of creating multiple application enviroments. We have defined three of these enviroments, each for a different purpose: development(`localhost`), test (`devcourses.matfyz.sk`) and production (`courses.matfyz.sk`) . In this configs, all settings for database access, Oauth, PHP interpreter and other enviroment dependent variables must be located.

Listing 2.2: Database configuration for development enviroment

```
$db['default']['hostname'] = 'localhost';
$db['default']['username'] = 'root';
$db['default']['password'] = 'root';
$db['default']['database'] = 'courses';
$db['default']['dbdriver'] = 'mysql';
$db['default']['dbprefix'] = '';
```

Another important feature of CodeIgniter is autolading of resources. Major drawback of development in PHP is that application must be instantiated, interpreted and run each time for every request. For optimalization purposes, we do not want each time to instantiate all of the application classes, just the ones we need. Also, it is often annoying to import many classes in controllers or models. Elegant solution for these problems is resource autoloading, which is provided as a CodeIgniter core functionality. In `config/autoload.php` we can specify all classes or paths we want to autoload and CodeIgniter will import them when application needs them.

2.3 Demonstration of Model-View-Controller

Now that we have enough knowledge of CodeIgniter background processing, we can focus on development. In this section, we will provide examples and demonstrate development in this framework.

2.3.1 Model

We cannot start explaining models without writing about active records. CodeIgniter uses a modified version of the Active Record Database Pattern. This pattern allows information to be retrieved, inserted, and updated in your database with minimal scripting. In some cases only one or two lines of code are necessary to perform a database action. CodeIgniter does not require that each database table be its own class file. It instead provides a more simplified interface. [5] This interface might seem similar to many ORM mappers.

Listing 2.3: Article model

```
class Article_model extends CI_Model
{

    function __construct()
    {
        parent::__construct();
    }

    function get_article($id)
    {
        return $this->db->from('article')
            ->where('id', $id)
            ->get()
            ->result_row();
    }
}
```

As we can see on listing 2.3, each model has to extend `CI_Model` class, which is the base class for each model and provides active records functionality. We also can not forget to call `parent::__construct()` in our constructor.

This is just an example of very simplified model, whose only purpose is to load one article with specified `id` from database. This file should be located in folder `application/models` and named `article_model.php`. Now, let's focus on views.

2.3.2 View

Listing 2.4: Article view

```
<html>
  <head>
    <title>Article page</title>
  </head>
  <body>
    <h1>Article says:</h1>
    <div>
      <?php $article->content ?>
    </div>
  </body>
</html>
```

All views are placed in `application/views` folder. Views are mostly HTML files with some PHP formatting code. We do not put any logic inside views. As we said, this view is just HTML with one PHP variable inside. How can we fill this variable? This is the time, when controller steps in.

2.3.3 Controller

Controllers must extend `CI_CONTROLLER` class. It is a base class which provides basic functionality for application management, such as view and models loading, routing or request/response management. To put our example to work, we have to write a controller like this:

Listing 2.5: Article controller

```php
<?php

class Article extends MY_Controller
{
    function __construct()
    {
        parent::__construct();
    }

    function show()
    {
        $id = $this->uri->segment(3);
        $article = $this->article_model->get_article($id);
        $this->layout->set_content('views/article_view', array('article' => $article)
            ;
    }
}
```

Now, let's explain what we did in listing 2.5. At first, we had to make a `__construct` method with `parent` call. This will instantiate controller basic functionality. In method called `show()` we at first got an ID of article. This ID comes from accessed URL. Full URL route to this controller would be `domain.com/article/show/:id`, where `:id` serves as specified ID. In the next step we used call from our model, which is autoloaded and then we filled this data to our view. And voila, building applications with CodeIgniter is simple!

Chapter 3

Courses 2 LMS

Now, that we have understaning of how CodeIgniter works, we can continue with description of Courses 2 system, as it was developed by Jakub Culiks work [2]. Motivation of this chapter is to describe this system as it was implemented before this work, explain motivation for creating a new learning management system (LMS) and bring some technical details of Courses 2 LMS. This chapter is considered as an introduction to our work and may be used to differentiate our work from previous works.

3.1 Motivation

Courses 2 system was developed during spring and winter of 2015 with development team led by Jakub Culik. This team was able to create a simple, customizable and modular learning management system which is already used at our faculty. Until the end of summer semester of 2016, 32 courses with total of 200 students were using this system.

Need for this system started in 2010, when a new aproach of the learning was introduced in the web design course at our faculty [2]. Each student had to publish a multiple blog posts which where then evaluated by his classmates and teachers of the course. Ratings were then projected into his final grade from this course. This approach led students to study more about topics they were interested in and were related to this subject. With this approach, students were also forced to discuss these problems and got greater insight into blog topics.

Used blog portal `blog.matfyz.sk` was also developed at our faculty in 2008 by Martin Rejda [10], so there were many options improve this system and to modify it to our needs. For example, blog portal was modified to allow students to implement and submit their own blog layout in XSLT. Hovewer, there were still some missing features like system for reviewing of these blog articles and discussions so the need for another system arose [2]. As a solution, new system, Courses was created [1].

This system was developed as a part of bachelor's thesis in 2013 [1] and was running 4 courses for one year [2]. After this time, it seemed clear that this system has to be completely redesigned and implemented from scratch. A new system had to be modular, scalable and well designed. And so, Courses 2 was born.

3.2 Semantics

Courses 2 system architecture can be divided into two parts: system part and modules. This division is required for maintaining this system modular and scalable. Now, let's look at each part.

3.2.1 System

System part is mostly responsible for "background stuff". Models and controllers extend standard CodeIgniter classes and there is only a few standalone libraries. All files of this part are located in `application/` folder and comply with standard CodeIgniter MVC flow. All libraries of system part are loaded once on each request.

Main goal of this part is to share all data to every component. For example, after user login, user information have to be avaible for each module. Since Course and Layout information are shared too, it is placed here as well. This part is also responsible to decide which modules has to be run on specific request and printing their outputs on the correct place in application layout.

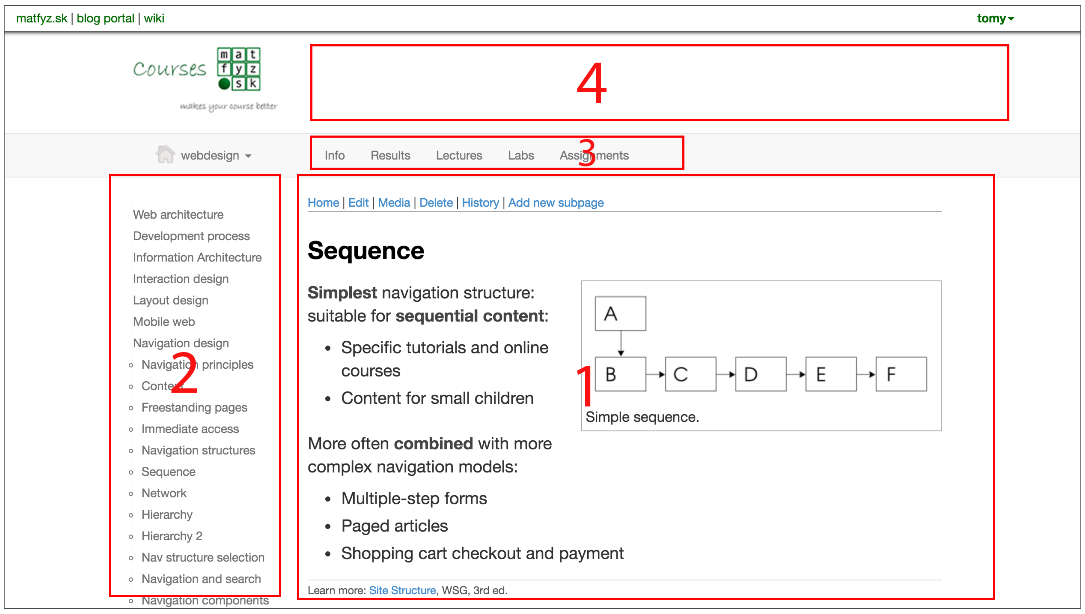
3.2.2 Modules

After system initialized all required classes, correct modules has to be run. Currently, there are 5 modules installed: `Core`, `Assignments`, `Quiz`, `Notes`, `Results`. Each module has different purpose and we will later explain `Assignments` throughoutly as it is main focus of this work. Other modules work similar and are not referenced by us, so we do not consider their description important for this work. It can, hovewer, be found in previous works [2].

3.2.3 Layout

Layout is managed by system part of this application. CodeIgniter default layout system is not sufficient for modularity, so it had to be replaced by a new one. Main problem was that each part of layout can be rendered by different module, which is decided at run-time. So page structure was divided into 4 different parts, each with its own rendering.

Courses 2 is build to offer one, root layout for all of the pages. It is not needed to create multiple different layouts in learning management system such as this. So there is just one, root layout, which is located in `application/views` directory of the project. This layout can be easily changed but this is not main focus of this work.



Sequence

Simplest navigation structure: suitable for **sequential content**:

- Specific tutorials and online courses
- Content for small children

Simple sequence.

More often **combined** with more complex navigation models:

- Multiple-step forms
- Paged articles
- Shopping cart checkout and payment

Learn more: Site Structure, WSG, 3rd ed.

Figure 3.1: Courses 2 screenshot

As we can see on figure 3.1, root layout was split into 4 different parts, each with specific purpose and rendered independently. Each numbered part is rendered by selected module, the rest is rendered by system. This screenshot was rendered using `Notes` module as it is showing content of a lecture. Part number 4 is reserved for notifications, such as new assignment was created or so.

3.3 Implementation

technicke detaily - databaza a sracky

3.3.1 HMVC extension in CodeIgniter

Now, let's focus on hierarchical model-view-controller design pattern again. We indicated before this pattern plays the main role in implementation of Courses 2. It is done so that each module has it's own MVC flow which is then merged by System part. This approach si not directly supported by CodeIgniter, so changes had to be made.

To do so, in our predecessive work CodeIgniter Modular Extensions [11] was selected. This extension is licenced as open-source and can be freely used and distributed. One of the reasons for usage of this extensions was that in CodeIgniter we can't call more than 1 controller per request. Therefore, to achieve HMVC, we have to simulate controllers [11].

Modular extensions allow us to call subcontrollers in controllers, but these subcontrollers must extend `MX_Controller` class instead of `MI_Controller` class. Another difference is that all autoload files has to be specified in subcontroller and are no longer autoloaded by default. This differences are shown on listing 3.1.

Listing 3.1: Module controller

```php
<?php
class Module_controller extends MX_Controller
{
    $autoload = array(
        'helper'     => array('escape', 'form'),
        'libraries'  => array('email'),
    );
}
```

With this extension, all modules has are placed in `application/modules` directory and can be called by base CodeIgniter controller. To some degree, they can also communicate with each other, but it is not a good practice since modules aims to be separate.

3.3.2 Oauth

3.4 Assignments Module

Now, we will describe Assignments module, whose development aims to be core of this work. We will provide description as it was, before publishing this work and before any change to module was made. We consider this section to be important for diferentiating our work from what has been done.

Part II

Implementation

Chapter 4

Peer review and improved submissions

4.1 Introduction

TODO [8]

Chapter 5

Team projects

5.1 Introduction

ss

Chapter 6

Assignments mirroring

6.1 Introduction

dfsdf

Chapter 7

Other work

TODO

End

TODO

Bibliography

[1] Jakub Culik. *Integrovany LMS system.* Bachelor's thesis, Faculty of mathematics physics and informatics, Comenius University, 2013.

[2] Jakub Culik. *Learning Management System for Open Web-based Learning.* Master's thesis, Faculty of mathematics physics and informatics, Comenius University, 2015.

[3] John Deacon. Model-view-controller (mvc) architecture. *[Online][Visited at: 25. March 2015.] http://www.jdl.co.uk/briefings/MVC.pdf*, 2009.

[4] Elislab. A brief history of codeigniter, 2014.

[5] Elislab. Codeigniter framework user guide, 2014.

[6] Elislab. Codeigniter license agreement, 2014.

[7] Sam Freysnett. Scaling web applications with hmvc. 2010.

[8] Martin Homola, Zuzana Kubincová, Jakub Čulík, and Tomáš Trungel. Peer review support in a virtual learning environment. In *State-of-the-Art and Future Directions of Smart Learning*, pages 351–355. Springer, 2016.

[9] Chris Pitt. *Pro PHP MVC.* Apress, 2012.

[10] Martin Rejda. *Redesign of blog portal.* Master's thesis, Faculty of mathematics physics and informatics, Comenius University, 2008.

[11] wiredesignz. Modular extensions - hmvc, 2014.