Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

УДК 007.51

Михалькевич Александр Викторович

Разработка web-ориентированного приложения на php-фрэймворке Laravel с использованием шаблона проектирования HMVC

Диссертация
на соискание степени магистра техники и технологии
по специальности 1-39 81 01 Компьютерные технологии проектирования
электонных систем

Научный руководитель
канд.техн.наук, доцент
АЛЕКСЕЕВ Виктор Фёдорович

Минск 2024

Автореферат

Наименование диссертационной работы Разработка web-ориентированного приложения на php-фрэймворке Laravel с использованием шаблона проектирования HMVC

Автор А.В.Михалькевич

Характеристика работы

Цель диссертации состоит в разработке электронного ресурса учебных дисциплин на основе php-фрэймворка Laravel с помощью архитектурного шаблона проектирования HMVC.

В процессе разработки web-приложения были выполнены следующие задачи: произведен анализ технологий разработки web-приложений, сравнены ядра операционных систем Linux, Mac и Windows, обоснован выбранный шаблон проектирования — HMVC; разработан алгоритм клиент-серверного взаимодействия, а также разработаны маршруты web-приложения.

С итоговой работой можно познакомиться по адресу http://erud.by

Анотация

-

Anotations in English

-

Ключевые слова диссертация, Laravel, HMVC, дисциплина, ООП,

Количество символов 972196

Количество изображений 12

Количество таблиц 2

Содержание

Введение
1 Анализ технологий разработки Web-приложений
    1.1 Сравнение ядер операционных систем Linux, Mac и Windows
    1.2 Сравнительный анализ одностраничных, интеграционных и мульти-интеграционных Web-приложений
    1.3 Обоснование выбора шаблона проектирования Web-приложения HMVC
    1.4 Выводы по главе 1
2 Операционная среда разработки
    2.1 Системный инструментарий
    2.2 Серверный инструментарий
    2.3 Конфигурирование операционной среды разработки
    2.4 Выводы по главе 2
3 Этапы разработки клиент-серверного web-приложения
    3.1 Макет главной страницы web-приложения
    3.2 Программирование серверной части web-приложения
    3.3 Программирование клиентской части web-приложения
    3.4 Размещение web-приложения в сети интернет
    3.5 Выводы по главе 3
Заключение
Список использованных источников
Приложения

Введение

Процесс разработки web-приложения включает в себя несколько этапов. Для достижения наилучшего результата все этапы произведены в строгой последовательности.

Первый этап – определение целей создания сайта и его позиционирования. На этом этапе необходимо определить вид сайта, для чего он создается и каких целей необходимо достичь с его помощью. Также определяются ключевые требования.

На втором этапе осуществляется создание дизайн - макета сайта. Производится выбор цветовой схемы сайта, определяется оформление и расположение элементов на страничке, подготавливается шаблон.

Разрабатывается визуальное оформление сайта.
На третьем этапе происходит верстка и программирование сайта. Создаются шаблоны страниц, внедряются интерактивные сервисы. Также на этом этапе разрабатывается структура сайта.

На четвертом этапе готовый сайт наполняется разнообразным содержимым: текстовые материалы, графические материалы, видео, аудиозаписи и т.д. Информация, размещенная на сайте, становится доступной для просмотра.

На пятом этапе происходит тестирование сайта и выкладка в сеть Интернет. Готовый сайт необходимо протестировать, на наличие ошибок. После того, как веб-страничка успешно прошла тестирование, производится выбор и настройка сервера. Когда сервер подготовлен и настроен, сайт можно выкладывать в сеть.

Работа над сайтом не заканчивается размещением ресурса в сети Интернет. Необходимо проводить работу по продвижению сайта, а также улучшать и обновлять его. Можно даже отметить, что разработка сайта - это бесконечный процесс: до тех пор, пока пользователи на него заходят, работа над ресурсом должна вестись.

Цель диссертации состоит в разработке электронного ресурса учебных дисциплин на основе php-фрэймворка Laravel с помощью архитектурного шаблона проектирования HMVC.

В процессе разработки web-приложения были выполнены следующие задачи: произведен анализ технологий разработки web-приложений, сравнены ядра операционных систем Linux, Mac и Windows, обоснован выбранный шаблон проектирования — HMVC; разработан алгоритм клиент-серверного взаимодействия, а также разработаны маршруты web-приложения.

Разработка велась на операционной системе Ubuntu, и были использованы следующие технологии: серверный язык программирования PHP, web-сервер Apache2 и сервер баз данных MySQL. В качестве основного фрэймврока был выбран php-фрэймворк Laravel [1].

С итоговой работой можно ознакомиться по адресу http://erud.by — электронный ресурс учебных дисциплин.

1 Анализ технологий разработки Web-приложений

Сообществом программистов принято разделять два направления *web*-разработки: *FrontEnd* (или разработка на стороне клиента) и *BackEnd* (или разработка на стороне сервера). Причем, ранее, на заре зарождения web, ожидалось, что основную нагрузку на себя возьмет серверная часть, т.к. сервера будут мощнее компьютеров пользователей. Оказалось же, что компьютеры

пользователей по характеристикам ничем не отличаются от серверов. Поэтому нагрузка решений между сервером и клиентом распределена примерно поровну.

Рассмотрим три группы технологий: для *Backend* (серверные технологии), для *Frondend* (клиентские технологии) и общий инструментарий, который может быть использован как на сервере, так и на клиенте [2].

По сути, в понятие *FrontEnd* входит всего три технологии: *HTML*, *CSS* и *JavaScript*. Но перечень инструментов применяемых на стороне клиента значительно шире [3]. Это:

– язык разметки *HTML*;

– таблица стилей *CSS*;

– формат данных *JSON*;

– формат данных *XML*;

– язык программирования *JavaScript*;

– библиотека *jQuery*;

– библиотека классов *Bootstrap 3, API Bootstrap* – для адаптивной верстки;

– фрэймворк *React*, реализующий структуру приложения на стороне клиента в формате архитектурных шаблонов, таких как *MVC* (модель-вид-контроллер), *HMVC* (иерархические модель-вид-контроллер), *MV-VM* (модель-вид-пользователь-вид-модель) и *MVW* (модель-вид-'что-то еще');

– *HTML5* и *API HTML5*, а именно: видео и аудио, холст, перетаскивание, работа с файлами, геолокация (в том числе работа с *GOOGLE* и *YANDEX*-картами), web-хранилища, взаимодейтсвие с сервером. Вспомогательный инструментарий [4];

– генератор микроданных schema.org предоставляет микроформат данных для *HTML* [5].

Далее рассмотрим используемые серверные технологии, или *BackEnd* [6]:

– операционная система *Xubuntu*. *Xubuntu* (*Ubuntu, Debian, Mint*…) по сути, являются не просто операционными системами, но серверами.

– локальный сервер *Apache2*;

– web-интерфейс для *Apache – Webmin*;

– язык программирования *PHP*;

– язык программирования *Java*;

– язык программирования *Python*;

– язык программирования *Node*;

– база данных *MySQL*;

– система управления базой данных *PHPMyAdmin*;

– система управления виртуальными хостами *Webmin*;

– менеджер зависимостей для *PHP Composer*;

- контейнеры Docker;

– php-фрэймворк *Laravel*. [7]

Существует огромное количество инструментария для программирования как на стороне сервера, так и на стороне клиента, как платных, так и бесплатных. Воспользуемся бесплатным дистрибьютивом (за исключением *PHPStorm*, который хоть и является платным, но для преподавателей и студентов БГУИР предоставляется бесплатно). [8]

– Интегрированная среда разработки *PHPShtorm*;

– Система контроля версий *Git;*

– Программная платформа *Node.js;*

– Браузеры *Firefox* и *Chrome*;

– Отладчик кода на стороне клиента *Firebug*;

- Программное обеспечение для автоматизации развёртывания и управления приложениями в средах с поддержкой контейнеризации - Docker.

– *FTP*-клиент FireFTP.

Для разработки *web*-приложения были установлены как серверные, так и клиентские технологии, а также подобран соответствующий инструментарий [2].

1 .1 Сравнение ядер операционных систем Linux, Mac и Windows

Существует единая система, на которой базируются ядра таких операционных системы, как *Linux*, *Mac* и *Windows*, и др. — это *Unix* [9].

С целью выбора наиболее оптимальной системы для разработки произведем аналитический обзор данных операционных систем. В таблице 1 отображены результаты сравнения операционных систем Linux, Mac и Windows.

Таблица 1 – Сравнение операционных систем *Linux*, *Mac* и *Windows*.

Критерии сравнения	**Linux**	**Windows**	**Mac**
Стоимость	Бесплатно	Платное программное обеспечение	Платное железо
Системные требования	Достаточно одноядерного процессора, 256 Мб оперативки и любой видеокарты.	Для стабильной работы понадобится процессор с двумя ядрами, 1 Гб оперативки и хорошей видеокартой.	Система закрыта, и однозначного вывода сделать не получится. Теоретически Mac получится запустить с 512 Мб оперативки, одноядерным процессором с частотой 1 ГГц и 9 свободными Гб памяти на жестком диске.
Слежка за пользователями	Нет	Да	Да
Расширения	Любые	.exe	Понятие расширений отсутствует. Установка программ осуществляется только через AppStore
Наличие альтернативных сборок	Имеется огромное количество	Нет	Нет
Видимость папок других операционных систем	Видит все папки	Нет	Нет

Встроенный сервер	Есть	Нет	Нет
Возможность конфигурирования системы	Есть	Нет	Нет
Предзагрузчик	Имеется предзагрузчик GRUB, который дает возможность пользователю выбирать операционную систему	Удаляет предзагрузчик GRUB, что не дает возможность поставить Windows поверх Linux. Linux же ставится поверх Windows.	-
Наличие вирусов и вредоносных программ	Нет	Да	Нет
Возможность использования облачных сервисов	Да	Нет	Нет

Говорить о функциональных возможностях операционной системы *Windows* 10 можно бесконечно долго, потому как они практически безграничны. Компания *Microsoft* не поскупилась и добавила в ОС сотни различных функций и возможностей, только вот работают многие из них без согласия пользователя, и еще многие нам, как пользователям, никогда не пригодятся. В данной ОС полно серьезных минусов, но главным, что делает эту операционную систему худшей среди других, является в политика обновления.

Windows 10 делает все без согласия пользователя. Причем, некоторые возможности можно отключить при использовании стороннего ПО, да и то мы никогда не можем быть уверены в том, что они действительно отключены [10].

В добавок к этому, *Microsoft* продает лицензию для Windows 10 Pro более чем за 14 000 рублей, но сейчас ее временно можно приобрести более чем в 40 раз дешевле - всего за 300 рублей. Ноутбуки под управлением Windows 10 стоят дороже на 600 рублей. Конечно, мы можем скачать пиратское программное обеспечение и установить Windows бесплатно, однако при этом остается лишь надеяться на то, что вместе с операционной системой не установятся вирусы.

По всем критериям сравнения Linux превосходит другие системы, что обосновывает наш выбор данной операционной системы. Документация по наиболее актуальной версии ядра Linux расположена в Интернете по адресу:

– *http://www.kernel.org/doc/Documentation*

Версиии Ubuntu (операционная система на основе ядра Linux) доступны по адресу:

– *http://ftp.uni-kl.de/pub/linux/ubuntu-dvd/*

Для установки Unux-подобных операционных систем без скачивания, можем воспользоваться дистрибьютивом программы unetbootin, которую предворительно нужно установить. Существуют варианты этой программы для *Linux*, *Windows* и *Mac*.

– *http://unetbootin.github.io/*

Установим и запустим программу.

После запуска получим такое окно, которое изображено на рисунке 1.

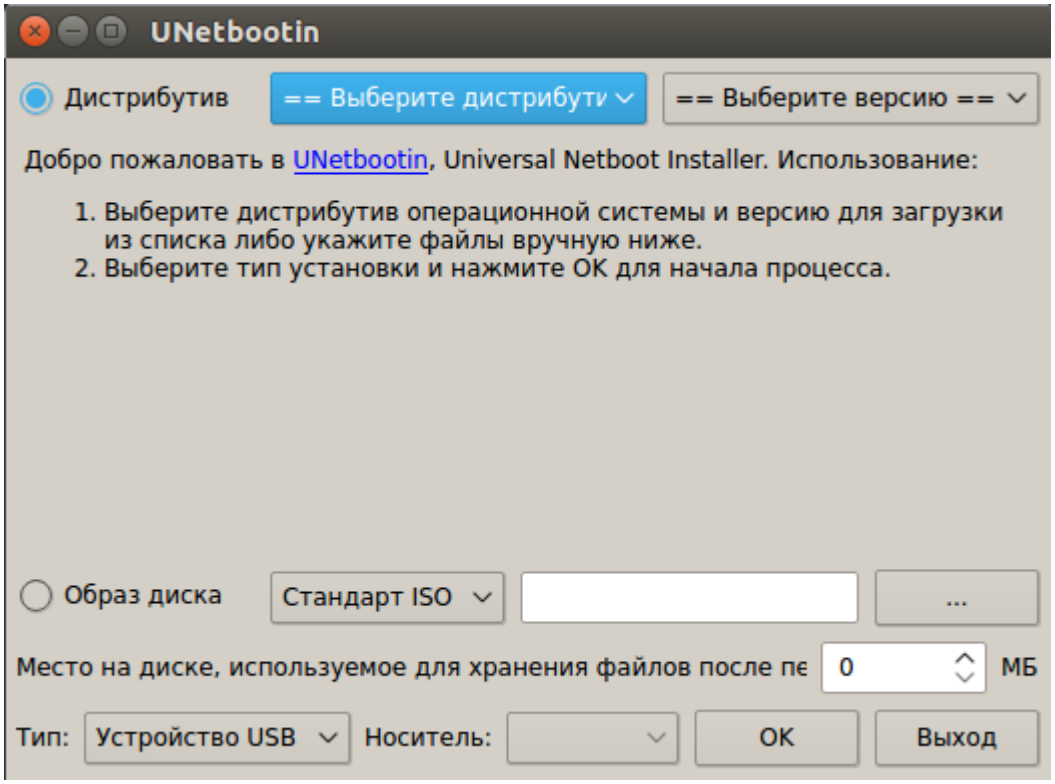


Рисунок 1. Использование Unetbootin для установки операционных систем Linux

Далее необходимо выбрать дистрибьютив операционной системы и версию для загрузки из выпадающих списков, также в типе установки выбираем носитель — свободный диск.

UNetbootin может создавать загрузочный *USB*-диск либо через скаченный *iso*-образ, либо через удаленные репозитории. Воспользуемся удаленным репозиторием и выберем последнюю версию дистрибьютива, рисунок 2.

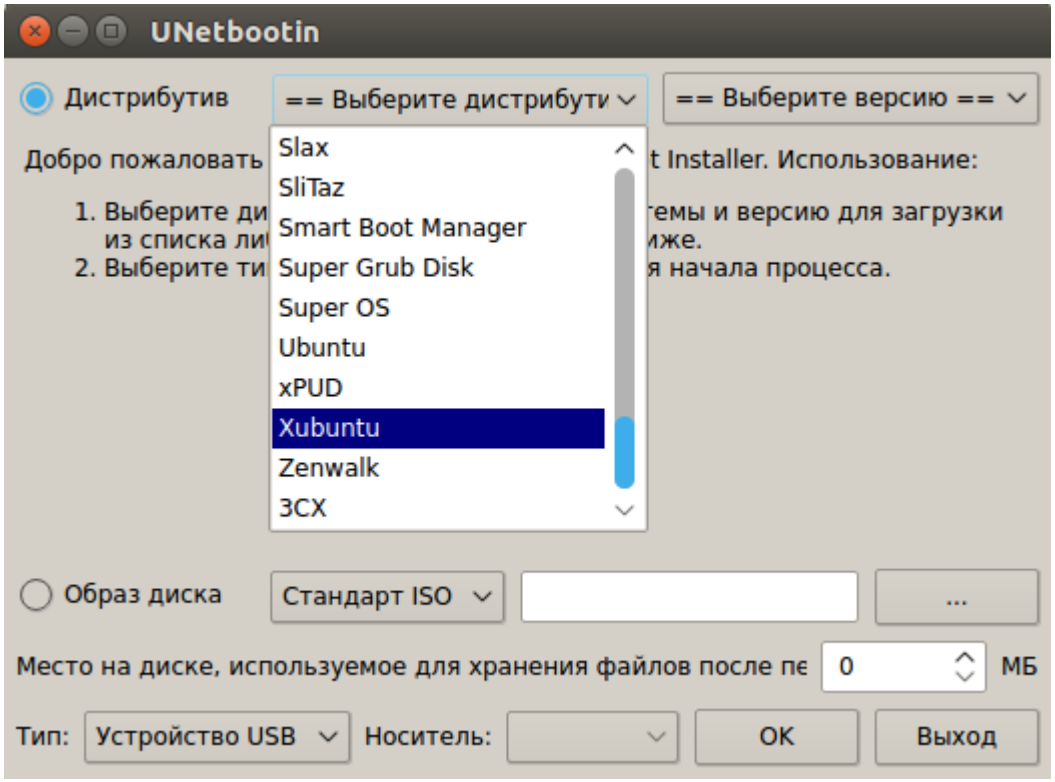


Рисунок 2. Установка операционной системы Xubutnu

Устанавливаем операционную систему с загрузочной флэшки, отвечая на вопросы установщика выбранной операционной системы.

1 .2 Сравнительный анализ одностраничных, интеграционных и мульти-интеграционных Web-приложений

По способу функционирования *web*-приложения можно разделить на 3 группы:

– Одностраничные *web*-приложения, у которых вся маршрутизация запросов выполняется в одной загруженной странице.

– Интеграционные. Работа этих *web*-приложений основана на взаимодействии серверов, например сервер *PHP* и сервер *MySQL*.

– Мульти-инеграционные. Работа этих web-приложении основана на взаимодействии серверных и клиентских *API*, например технология *RestFull API* позволяет обойти ограничения *SQL*-запросов, которые выполняются только на стороне сервера. *RestFull API* преобразоваывает данные сервера в формат *JSON*, с которым взаимодействует *JavaScript*. Формат *JSON* является единым унифицированным форматом данных как на стороне клиента, так и на стороне сервера.

Мульти-интеграционными являются такие приложения, которые могут:

– взаимодействовать с любыми, как серверными, так и клиентскими технологиями.

– самообновляться.

Мульти-интеграционные программы можно постоянно дописывать и обновлять. Объектно-ориентированные решения в написании кода при построении таких программ предоставляет еще больше гибкости, поскольку классы библиотек можно изменять, добавлять и удалять, а объекты, построенные на их основе таких классов могут иметь широкий диапазан характеристик. Мульти-интеграционные web-приложения - это современные, и с какой-то точки зрения, вечные приложения (т.к. они могут интегрироваться с любой базой данных и запускаться на любом сервере). Такие серверные технологии, менеджер зависимости для *PHP* - *Composer* и обновляемый фрэймворк *Laravel*, предоставляют мульти-интеграционность на стороне *web*-сервера. Клиентскую часть мы всегда можем переписать, добавив туда новый функционал, либо изменив существующие файлы стилей и скриптов. А вот с сервером баз данных дело обстоит иначе. Не все языки программирования настроены на удобное взаимодействие с базой *MySQL*, используемой в приложении [11]. Поэтому для добавления мульти-интеграционности на стороне сервера баз данных, было решено разработать RestFull Api.

Транзакция по такому *API* будет состоять, как минимум, из следующих запросов и ответов:

– метод запроса, например: *GET, POST, PUT, DELETE*, также имеется возможность использовать любые другие типы Request-переменных;

– маршрут запроса на добавление данных;

– маршрут запроса на удаление данных;

– маршрут запроса на вывод всех данных;

– маршрут запроса на вывод данных по идентификатору;

– маршрут запроса на обновление данных;

– могут присутствовать иные запросы, взаимодействующие с данными;

– тело ответа в формате *JSON* или в формате статуса страницы;

Многие разработчики стали использовать парадигму RestFull Api в разработке *web*-сервисов с использованием *HTTP* [12]. Особенностью *RestFull API* является его универсальность. Однако при разработке модуля были введены некоторые ограничения. А именно:

– использовались всего два метода отправки запроса *GET* и *POST*.

– тело ответа всегда формируется в формате *JSON*, даже если это метод удаления или обновления. При удалении записи возвращается ответ в виде пустых фигурных скопок, т.е. пустой *JSON* [13].

Для реализации модуля *RestFull Api* было решено использовать *Node.js*. Данный модуль на прямую не связан с самим приложением, и может быть использован другими приложениями, которым необходимо обращаться к данным.

Разработка любой node-программы начинается с определения зависимостей. *RestFull Api* не будет исключением. Рассмотрим файл зависимостей *packages.json*, в котором указаны название модуля, версия, опсание, исполняемые файлы, скрипты и необходимые зависимости.

```
{
  "name": "todolistapi",
  "version": "1.0.0",
  "description": "RESTful todoListApi",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon server.js"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/mikhalkevich/RestfullApi"
  },
  "keywords": [
    "RESTful",
    "API",
    "Tutorial"
  ],
  "author": "Mikhalkevich Alexandr",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/mikhalkevich/RestfullApi/issues"
  },
  "homepage": "https://github.com/mikhalkevich/RestfullApi#readme",
  "devDependencies": {
    "nodemon": "^1.11.0"
  },
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "mongoose": "^4.7.2"
  }
}
```

Из листинга видно, что для реализации модуля необходимо установить следующие зависимости: *body-parser*, *express* и *mongoose* [1-А]. Для установки зависимостей, воспользуемся менеджером зависимостей npm:

```
npm install
```

Реализация необходимых *GET* и *POST*-запросов находится в маршрутизаторе, роль которого выполняет файл *api/routes/todoListRoutes.js*:

```
'use strict';
module.exports = function(app) {
        var todoList = require('../controllers/todoListController');
        // todoList Routes
        app.route('/tasks')
                .get(todoList.list_all_tasks)
                .post(todoList.create_a_task);
        app.route('/tasks/:taskId')
                .get(todoList.read_a_task)
                .put(todoList.update_a_task)
                .delete(todoList.delete_a_task);
};
```

Скачать готовый модуль *RestfullApi* на *Node.js* можно с авторского репозитория по адресу:
– *https://github.com/mikhalkevich/RestfullApi.git*

1 .3 Обоснование выбора шаблона проектирования Web-приложения HMVC

Все существующие шаблоны проектирования условно можно разделить на две большие группы: архитектурные шаблоны и шаблоны практических решений. Если шаблоны практических решений – это шаблоны решений конкретных задач, с которыми программист сталкиваемся в процессе решения задач, то архитектурные шаблоны проектирования – это способы структурирования файлов проекта, и так как архитектурный шаблон предоставляет архитектуру проекта, то разбираться с ними приходится до создания проекта [13].

Все мы стремимся к хорошему коду, поэтому в разработке программ необходимо использовать архитектурные шаблоны проектирования: для того, чтобы код был понятен другим программистам, а также для повторного использования удачного ранее решения.

Шаблоны проектирования нужны для того, чтобы помочь реализовать какую-то идею. Начинаем с построение архитектуры проекта, и только потом подбираем шаблонное решение для контретной практической задачи [14].

Классификация архитектурных шаблонов проектирования [15]:

– Простой шаблон;

– Шаблонная функция и метод буферизации;

– *MVC* (модель, представление, контроллер);

– *HMVC* (иерархический MVC);

– *MV-VM* (данные передаются сразу в шаблон, минуя логику контроллеров, и пользователь из шаблона может сразу менять данные). Данный способ организации кода используется в фрэймворках *JavaScript*.

Простой шаблон

Простой шаблон используется для простых проектов, как правило, не используется в фрэймворках и *cms*-ках, но он полезен при разработке простых проектов либо в процессе изучения *PHP*, когда акцент делается на скорость разработки и функциональность в ущерб архитектуре проекта и масштабируемости (последующего расширения) [15-А].

В простом шаблоне страница визуально делится на изменяющуюся и неизменную части. Сперва выделяется изменяющаяся центральная часть дизайна, для главной страницы - это *index.php*. Неизменная часть - это *top.php*, и *bottom.php*, соответственно, то, что идет выше, и ниже изменяющейся части. Файлы *top.php* и *bottom.php* подключаются на всех страницах.

Пример использования:

Изменяемая часть шаблона, файл *index.php.*

```
require_once ("templates/top.php");
  if (!$_GET['id']){
     $file = (int)$_GET['id'];
  }else {
     $file = "index";
  }
  $query = "SELECT * FROM news WHERE id = '".$file."' AND hide='show'";
    $adr = mysql_query($query);
   if (!$adr) exit($query);
   $tbl_users = mysql_fetch_array($adr);
   echo $tbl_users['name'];
   echo $tbl_users['body'];
 require_once ("templates/bottom.php");
```

Неизменная часть шаблона находится в файлах *top.php* и *bottom.php*

Плюсы:

– простота.

Минусы:

– разрезан на 2 части;

– видимость переменных;

– проблемы со вложенной структурой;

– не предназначен для последующего расширения.

Шаблонная функция

Шаблонная функция - это такой шаблон проектирования, при котором изменяемые части шаблона определяются в функциях [15-Б].

Файл index.php.

```
function content()
{
  // тут содержимое функции
```

```
}
// Установка переменных основного шаблона.
$title = 'Главная';
// Генерация HTML всей страницы.
include 'v_main.php';
```

Плюсы:

– целостность;

– ограничение видимости переменных.

Минусы:

– большинство минусов совпадает с минусами простого шаблона;

– представление вызывает функцию контроллера.

Метод буферизации

Метод буферизации - это модификация шаблонной функции. Вместо функции используется буфер [18].

Занесение шаблона в буфер.

```
ob_start();
$content = ob_get_clean();
```

Вывод буфера

Плюсы:

– вложенность шаблонов

– независимость представления от контроллера

– целостность шаблона

– возможность кэширования

Минусы:

– видимость переменных

– громоздкость кода

– проблеммы с вложенностью файлов

Весь код, который находится между функциями *ob_start()* и *ob_get_clean()*, на экран не выводится, а заносится в переменную.

MVC

Расшифруем само понятие *MVC* [15-В].

Model - модели данных, которые многие и без того используют без фреймворков. Фактически, обычные классы для работы с разными данными.

View - представления, или вид, в котором отображаются данные.

Controller – основной вызываемый класс, содержащий базовую логику приложения.

По концепции *MVC*, когда пользователь делает запрос, запрос сперва попадает в контроллер (*Controller*). Затем в конроллере может происходить вызов модели (*Model*) и затем передача данных в шаблон представления (*View*). Основной недостаток *MVC* заключается в том, что запрс работает только с одним контроллером.

HMVC

В *HMVC* фактически путь запроса такой же, как и у *MVC*. Но каждый контроллер может

стать родителем другого контроллера. Главное отличие от *MVC* – это возможность передачи запроса по контроллерам и другим классам. Один и тот же запрос может быть передан в цепочку классов, начиная от родительского класса [15-Г].

HMVC позволяет легко собирать воедино и легко управлять даже большими кусками кода, хотя больших, с усложненной абстракцией.

Сравнение архитектурных шаблонов проектирования *MVC* и *HMVC* представлено на рисунке 3.

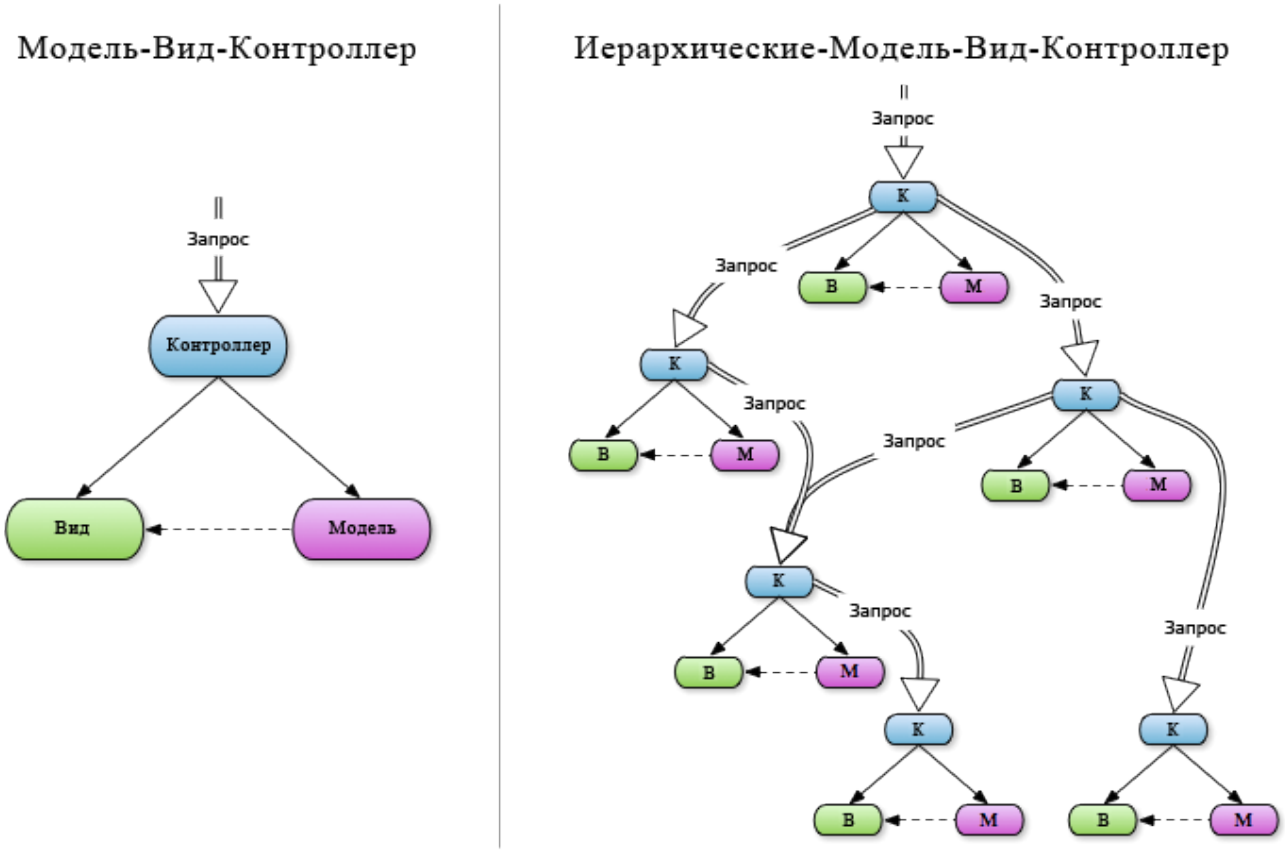


Рисунок 3. HMVC и MVC

MV-VM

Отличие от *MVC* состоит в отстутствии контроллеров. Данные из модели попадают в элементы представления, и пользователь меняя элементы представления, меняет данные. Данный шаблон проектирования подходит для используют фрэймворки на стороне клиента. Например, *React.js* и *Angular.js* позволяют писать код как на *MVC*, так и на *MV-VM* [15-Д].

MVW

Данный шаблон проектирования оперирует моделями, элементами представления и вспомогательными файлами (не контроллерами), в которых реализована вся логика их взаимодействия [15-Е].

Model – View – What you want to work

Модель и элементы представления взаимодействуют между собой через какой-то внешний интерфейс, не контроллер.

Шаблоны проектирования практических задач

Шаблоны проектирования практических задач представляют наилучшие решения часто

встречаемых задач и упрощают повторное использование удачных решений [16]. В разработке приложения использовалось множество разных шаблонов проектирования практических задач. Ведь каждый такой шаблон — это решение своей задачи, нельзя одним паттерном решить все задачи и нельзя всеми паттернами решить одну и ту же проблему…

Очень сложно описать все используемые в разработке приложения шаблоны, тем более, что смешиваясь между собой шаблоны проектирования могут пораждать новые шаблоны. Однако, необходимо отметить, что используемый в разработке фрэймворк Laravel выступает не только в роли архитектурного шаблона проектирования, предоставляя структуру кода в формате HMVC, но и содержит ряд встроенных решений, или шаблонов проектирования практических задач, упрощающих разработку web-приложений.

Основой разработки на Laravel и связующим звеном различных шаблонов проектирования является внедрение зависимостей (классов или библиотек). Поэтому, вместо описания используемых шаблонов проектирования, рассмотрим, как осуществляется внедрения зависимостей в проект.

Внедрение зависимостей в Laravel

Есть два способа, которыми IoC-контейнер разрешает зависимости: через функцию-замыкание и через автоматическое определение [17]. Для начала исследуем замыкания. Сперва рассмотрим, как объявляется контейнер:

```
App::bind('foo', function ($app) {
  return new FooBar;
});
```

Создание объекта из контейнера:

```
$value = App::make('foo');
```

При вызове метода App::make() вызывается соответствующее замыкание и возвращается результат её вызова. Если возникнет необходимость поместить в контейнер тип, который должен быть извлечён (создан) только один раз, и чтобы все последующие вызовы возвращали бы тот же объект, мы можем это сделать таким простым способом:

```
App::singleton('foo', function () {
  return new FooBar;
});
```

Мы также можем поместить уже созданный экземпляр объекта в контейнер, используя метод instance():

```
$foo = new Foo;
App::instance('foo', $foo);
```

1 .4 Выводы по главе 1

Обоснование выбора Xubuntu. По всем критериям сравнения *Linux* превосходит другие

системы, что обосновывает наш выбор *Linux*-подобной операционной системы. Из множества таких систем была выбрана *Xubuntu*, т.к. она поставляется вместе с встроенным сервером *Apache2* и необходимыми серверными технологиями.

Обоснование выбора HMVC.

Выглядящий в теории идеальным, *MVC* на практике сталкивается с рядом проблем. Для начала вспомним какую основную проблему он должен разрешить: разделив приложение на три различных аспекта (контроллер, вид и модель) добиться минимальной зависимости между ними, а также между различными частями приложения. При использовании *MVC*, разработчик рано или поздно сталкивается с рядом проблемм.

Проблема первая. На практике обычно приходиться оперировать одновременно несколькими моделями, например: статьи, пользователи, комментарии. В принципе, это не критично, паттерн *MVC* это предусматривает, но это увеличивает количество зависимостей — вид и контроллер зависят более чем от одной модели, а от одной модели зависят более одного вида и контроллера.

Проблема вторая. Для отображения данных из разных моделей хотелось бы использовать виды созданные специально для них. Например, выглядит логичным отображение комментариев одинаково для статей и для товаров. Такое классический *MVC* не предусматривает, но это частично обходится использованием шаблонов. Т.е. используется один вид который получает данные из моделей, а для отображения их отображения используется комбинация нескольких шаблонов. И снова увеличивается количество зависимостей.

Проблема третья. Иногда действия надо выполнить не над одной моделью, а над несколькими одновременно. Например, при удалении пользователя надо удалить все его статьи и комментарии. В результате приходится создавать контроллер, который описывает операции не только над моделью к которой он относится, но над моделями к которым непосредственного отношения контроллер не имеет. Таким образом появляются не очевидные зависимости.

Поскольку проблемы возникают из-за того, что вместо *MVC* получается что-то наподобие *MMMVVVCC*, где каждая модель вид и контроллер могут принадлежать разным системам, ответ очевиден — вернуться к *MVC* в котором есть лишь по одной модели, виду и контроллеру.

Важной отличительной особенностью *HMVC* является использование в приложении только жестко фиксированных триад модель-вид-контроллер, которые взаимодействуют с остальными подсистемы через контроллер. На первый взгляд, может показаться, что для возможности реализации *HMVC* достаточно возможности вызова контроллера из другого контроллера. Но в *web*-приложении поведение зависит не просто от команды переданной контроллеру, а от *http*-запроса в целом. И модель и вид могут сами анализировать запрос и некоторым образом изменять свое поведение. Поэтому требуется возможность передать запрос другому контроллеру, причем не обязательно тот же, что был получен. Сделать это можно тремя различными методами.

HMVC позволяет распределить задачи между контроллерами, при этом запрос проходит только через один контроллер. Например, один сервер управляет статьями, один профилями пользователей, а третий комментариями. В случае достаточной изолированности модулей, для

быстрой реализации этого достаточно лишь прописать в соответствующих запросах что они внешние, и указать адреса серверов для их обработки.

В серверных фрэймворках контроллеры выполняют основную логику запроса. Это обосновывает выбор шаблона проектирования *HMVC,* а следовательно и фрэймворк – *Laravel (The best php framework)*.

Фреймворк должен обеспечить определенные механизмы для минимизации рутинной работы по взаимодействию контроллеров, слои абстракции для работы с запросами, базой данных и т.д.

Ключевые особенности, лежащие в основе архитектуры Laravel:

– Логика приложения работает по шаблону проектирования *HMVC.*

– Пакеты (англ. *Packages*), которые позволяют создавать и подключать модули в формате *Composer* к приложению на *Laravel*. Многие дополнительные возможности уже доступны в виде таких модулей.

– Обратная маршрутизация связывает между собой генерируемые приложением ссылки и маршруты, и позволяет изменять последние с автоматическим обновлением связанных ссылок. При создании ссылок с помощью именованных маршрутов Laravel автоматически генерирует конечные URL.

– Автозагрузка классов — механизм автоматической загрузки классов PHP без необходимости подключать файлы их определений в include. Загрузка по требованию предотвращает загрузку ненужных компонентов; загружаются только те из них, которые действительно используются.

– Составители представлений (англ. view composers) — блоки кода, которые выполняются при генерации представления (шаблона).

– Инверсия управления (англ. Inversion of Control) — позволяет получать экземпляры объектов по принципу обратного управления. Также может использоваться для создания и получения объектов-одиночек (англ. singleton).

– Миграции — система управления версиями для баз данных. Позволяет связывать изменения в коде приложения с изменениями, которые требуется внести в структуру БД, что упрощает развёртывание и обновление приложения.

– Модульное тестирование (юнит-тесты) — играет очень большую роль в Laravel, который сам по себе содержит большое число тестов для предотвращения регрессий (ошибок вследствие обновления кода или исправления других ошибок).

– Потраничный вывод (англ. pagination) — упрощает генерацию страниц, заменяя различные способы решения этой задачи единым механизмом, встроенным в *Laravel*.

Для работы с базами данных использовался самый популярный в этой области язык – *MySQL*.

Архитектура *MySQL* сильно отличается от архитектур иных серверов баз данных и делает эту СУБД удобной для разработки web-приложений. Логический вид архитектуры *MySQL* представлен тремя уровнями.

На верхнем уровне располагаются службы, не являющиеся уникальными компонентами *MySQL*, которые необходимы для обслуживания соединений, аутентификации и обеспечения

безопасности. Второй уровень — это уровень оптимизации и кэширования запросов, а также встроенных функций — даты, времени и шифрования. Третий уровень содержит системы хранения данных.

Основной системой хранения данных, использованной в разработке является *InnoDB*. Современная версия *InnoDB* обеспечивает следующий функционал:

– построение индексов путем сортировки;

– возможность добавления и удаления индексов без перестройки всей таблицы;

– особый формат хранения данных, сжимающий данные;

– специальные методы, для хранения различных типов данных: строковых, двоичных, числовых, даты и времени, битовых и других специальных типов.

2 Операционная среда разработки

Ядром среды разработки является *Linux*-дистрибьютив *Xubuntu*. *Xubuntu* поставляется вместе с графической оболочкой *Xfce*, некоторым минимальным набором дистрибьютива. В набор программ по-умолчанию входят офисные (*LibreOffice*), медийные, графические, системные программы, менеджер приложений, браузер и терминал.

После входа в систему открывается домашний каталог текущего пользователя. Все содержащиеся в нем файлы и подкаталоги уже принадлежат нам (в отличии от *Windows* и *Mac*, в которых пользователь не имеет доступа к системным папкам проекта). Однако, пользователи *Linux*, если у них нет *root*-прав не могут ни изменять, ни удалять системные файлы, но могут их читать. Для редактирования системных файлов, пользователь linux должен получить root-права.

Домашний каталог сокращенно обозначается тильдой (~). У обычного пользователя Linux домашний каталог располагается по адресу

/home/name

У администратора аналогичный адрес таков:

/root.

Каталог сервера:

/var/www

Путь к файлу host

/etc/hosts

Сервер Apache

/etc/apache2/apache2.conf

2 .1 Системный инструментарий

Конечно же, главным инструментом любого разработчика является операционная система.

Запись образа Xubuntu на флэшку

Файловая система EXT4 является дефолтным выбором для большинства дистрибутивов Linux [18]. Она проверена, протестирована, стабильна, отлично работает и широко поддерживается. В данном случае, мы ищем стабильность, поэтому для флэшки выбираем файловую систему

EXT4.

Утилита dd позволяет побайтово переносить содержимое ISO образа на флешку. Такой метод записи менее удобен, чем использование графических утилит, но иногда работает когда не помогают другие.

Сначала узнаем имя флешки в файловой системе. Для этого используем утилиту fdisk:

```
sudo fdisk -l
```

```
Диск /dev/sdb: 29,72 GiB, 31914983424 байт, 62333952 секторов
Disk model: Storage Device
Единицы: секторов по 1 * 512 = 512 байт
Размер сектора (логический/физический): 512 байт / 512 байт
Размер I/O (минимальный/оптимальный): 512 байт / 512 байт
Тип метки диска: dos
Идентификатор диска:
```

В данном случае флешка имеет имя /dev/sdb.

Теперь запишем на неё образ:

```
sudo dd if=~/Загрузки/ubuntu20_04.iso of=/dev/sdb bs=1M
```

Опция if нужна для передачи образа, который надо записать, а of - устройство, на которое его надо записать. Опция bs помогает утилите работать быстрее. Обратите внимание, что данные надо записывать именно на флешку, а не на раздел на ней. Больше никаких операций не потребуется, так как вся структура образа будет перенесена на устройство.

Установка программ

Для установки программного обеспечения и пакетов системы воспользуемся утилами командной строки, в частности, утилами apt, apt-get и snap.

apt это утилита, которая появилась, как альтернатива apt-get. Она выполняет практически все те же функции, что и apt-get, но с ней проще и понятнее работать. Например, все команды apt имеют простой синтаксис: *apt название_команды*. А у apt-get есть дополнительные команды, например, *apt-cache*. Таким образом, при использовании apt, пользователю не нужно запоминать дополнительные наборы команд.

Помимо упрощения работы с командами, apt нагляднее выводит информацию, вроде бы мелочи, но работать удобнее. Например, apt умеет показывать прогресс бар, а при выполнении *apt update* можно увидеть сколько пакетов можно обновить.

Стоит отметить, что утилита apt-get более функциональна, чем apt. Но для рядового пользователя apt будет более чем достаточно.

sudo apt list - список доступных программ

sudo apt list --installed список всех установленных программ

sudo apt-get --purge remove programm_name - удаление программы.

Git

Лучше установить *Git* из исходных кодов, поскольку так вы получите самую свежую версию. Каждая новая версия *Git*'а обычно включает полезные улучшения пользовательского интерфейса, поэтому получение последней версии — часто лучший путь, если, конечно, вас не

затрудняет установка программ из исходников. К тому же, многие дистрибутивы *Linux* содержат очень старые пакеты. Поэтому, если только вы не на очень свежем дистрибутиве или используете пакеты из экспериментальной ветки, установка из исходников может быть самым выигрышным решением.

GitHub — крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки. Для создания удаленного репозитория, нужна регистрация на *github.com*.

По адресу http://github.com/mikhalkevich можно ознакомиться со всеми открытыми репозтиориями автора.

Для установки *Git*'а понадобятся библиотеки, от которых он зависит: *curl, zlib, openssl, expat и libiconv* [19]. Можно воспользоваться следующими командами улиты apt-get, чтобы разрешить все зависимости:

apt-get install git

PHPStorm

PHPStorm - интегрированная среда разработки для *web*, стоимостью $199. Но для учебных заведений, в том числе и для БГУИР распространяется бесплатно [20].

В сравнении с другими операционными системами, установка PHPStorm на Ubuntu наиболее удобна и проста. Для установки необходимо ввести консольную команду:

```
sudo snap install phpstorm --classic
```

Docker

Для развертывания и запуска приложения с помощью контейнеров воспользуемся Docker [21].

Устанавливаем последнюю версию, и информацию по установке берем из официальной документации, по ссылке https://docs.docker.com/install/linux/docker-ce/ubuntu

Процесс установки Docker можно разбить на несколько шагов:

1. Обновляем пакеты Ubuntu

```
sudo apt-get update
```

2. Устанавливаем необходимые зависимости

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common
```

3. Добавляем официальный ключ

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

И сравниваем отпечаток:

```
sudo apt-key fingerprint 0EBFCD88
```

4. Добавляем репозиторий где расположен Docker

```
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
```

5. Обновляем индекс пакетов apt

```
sudo apt-get update
```

6. Установка Docker

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

7. Убеждаемся, что Docker CE установлен правильно, запустив образ hello-world.

```
sudo docker run hello-world
```

2 .2 Серверный инструментарий

Операционная система *Ubuntu* (*Xubuntu*) поставляется вместе с встроенным сервером *Apache2*.

Apache

Когда возникает необходимость обновить либо переустановить сервер *Apache*, можем воспользоваться репозиториями утилы apt-get [22].

sudo apt-get update

sudo apt-get install apache2 (опционально)

sudo service apache2 restart - перезапуск сервиса apache.

В Ubuntu конечный файл настройки (*apache2.conf*) расделён на несколько файлов, расположенных в разных поддиректориях. Подробней написано в комментариях файла *apache2.conf*.

/etc/apache2/ |-- apache2.conf
| `-- ports.conf
|-- mods-enabled
| |-- *.load
| `-- *.conf
|-- conf-enabled
| `-- *.conf
`-- sites-enabled
`-- *.conf

Настройки модулей расположены в директории */etc/apache2/mods-available*. Для подключения или отключения модулей (настроек модулей) следует использовать соответствующие команды *a2enmod* или *a2dismod*. Подключения модуля:

```
sudo a2enmod
```

Свои настройки следует сохранять в файлы, расположенные в директории */etc/apache2/conf-available*. Для подключения или отключения своих настроек используем соответствующие команды *a2enconf* или *a2disconf*. Пример подключения файла со своими настройками:

```
sudo a2enconf
```

Настройки виртуальных хостов следует сохранять в файлы, расположенные в директории */etc/apache2/sites-available*. Для подключения виртуальных хостов следует использовать соответствующие команды *a2ensite* или *a2dissite*. Пример подключения виртуального хоста:

```
sudo a2ensite
```

Для указания кодировки по умолчанию используем директиву *AddDefaultCharset* в файле */etc/apache2/conf-available/charset.conf*:

```
AddDefaultCharset UTF-8
```

Виртуальные хосты

Файлы настроек виртуальных хостов хранятся в */etc/apache2/sites-available/*.conf*. По умолчанию в *Apache* уже настроен один виртуальный хост. Его настройки лежат в файле *000-default.conf*.

Настройки виртуального хоста:

```
#Имя хоста
ServerName host1.server1
#Корневая папка хоста
DocumentRoot /var/www/host1.server1
#Разрешение на перезапись всех директив при помощи .htaccess
AllowOverride All
```

Назовём файл настройки именем хоста *erud.server1.conf* и сохраним.

После создания файла настроек необходимо дописать в */etc/hosts* имя хоста: 127.0.0.1 *host1.server1*

Для включения созданного виртуального хоста воспользуемся утилитой a2ensite:

```
sudo a2ensite host1.server1
```

Отключается хост аналогично утилитой *a2dissite*:

```
sudo a2dissite host1.server1
```

Node

В стандартных репозиториях *Ubuntu* 18.04 уже есть версия *Node.js*, которую удобно использовать для обеспечения однородной среды выполнения сетевых приложений сразу на нескольких серверах [23].

Для установки последней версии воспользуемся пакетным менеджером *apt*. Сначала

обновим локальный индекс пакетов:

```
sudo apt update
```

Теперь установим *Node.js* из репозиториев утилы apt:

```
sudo apt install nodejs
```

На этом установка *Node.js* закончена. Однако для полноценной работы с Node.js требуется уще установить npm - менеджер зависимостей. Это можно сделать при помощи следующей команды:

```
sudo apt install npm
```

Это позволит легко устанавливать модули и пакеты для *Node.js*.

Для проверки установленной версии *Node.js* выполним команду:

```
nodejs -v
```

```
npm -v
```

MySQL

По умолчанию в репозиторий пакетов *APT* в *Ubuntu* 18.04 включена только последняя версия *MySQL*. На момент написания этой диссертации это *MySQL 5.7* [24].

Установка пакета *MySQL* осуществляется с помощью следующей команды:

```
sudo apt install mysql-server
```

Эта команда установит *MySQL*, но при этом не предлагается задавать пароль или вносить какие-либо правки в конфигурацию.

Начиная с 2010 года в *MySQL 5.5* движок хранения данных InnoDB является основным движком в *MySQL*, и используется по умолчанию. Отличительными особенностями этого движка являются:

– высокая производительность;

– автоматическое восстановление данных после сбоя;

– автоинкрементируемые значения;

– данные сохраняются в одном или нескольких файлах.

PHP

Язык программирования *PHP* поставляется вместе с операционной системой *Xubuntu*. Для обновления *PHP* можно воспользоваться следующей командой [25].

```
apt-get install php
```

Кроме того, в процессе разработки нам потребуются некоторые модули для PHP, которые необходимо установить отдельно. Наличие этих модулей обязательно, так как без них не установится *Laravel*.

```
apt-get install php-pear php7.0-dev php7.0-zip php7.0-curl php7.0-gd php7.0-mysql php7.0-mcrypt php7.0-xml libapache2-mod-php7.0
```

Composer

Composer - менеджер зависимостей для PHP [26].

Сперва необходимо скачать установщик composer. Для этого можно воспользоватся диррективой `wget`:

```
sudo wget -O composer-setup.php https://getcomposer.org/installer
```

После чего, запускаем команду установки:

```
sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer
```

Composer установлен. В чём можно убедиться, запустив команду

```
composer
```

PHPMyAdmin

Кроме сервера базы данных *MySQL*, нам потребуется система управления базой данных *phpMyAdmin* [27].

Для установки обновим наш локальный индекс пакетов, а затем используем систему управления пакетами *apt* для загрузки и установки необходимых файлов:

```
sudo apt-get update
```

```
sudo apt-get install phpmyadmin php-mbstring php-gettext
```

В процессе установки будет задано несколько вопросов по конфигурации.

При выборе сервера, выбераем *apache2*.

На вопрос, хотим ли мы использовать *dbconfig-common* для настройки базы данных отвечаем утвердительно.

Далее будет запрошен пароль администратора базы данных и повтор пароля.

В процессе установки в директорию */etc/apache2/conf-enabled/* будет добавлен файл конфигурации *phpMyAdmin* для *Apache*.

Единственное, что мы должны сделать вручную, так это включить расширения *PHP mcrypt* и *mbstring* следующими командами:

```
sudo phpenmod mcrypt
```

```
sudo phpenmod mbstring
```

Далее перезапустим Apache для применения изменений:

```
sudo systemctl restart apache2
```

Теперь мы можем осуществить доступ к веб-интерфейсу *phpMyAdmin*, введя имя домена или

публичного статического *IP* адреса сервера и строки */phpmyadmin*:

```
https://localhost/phpmyadmin
```

Установка Laravel, Nginx и MySQL с помощью Docker Compose

В последнее время Docker часто используется для развертывания приложений, поскольку он упрощает этот процесс с помощью виртуальных контейнеров. Например, при использовании стека LEMP (который состоит из PHP, Nginx и MySQL) и Laravel, Docker может значительно ускорить процедуру настройки приложения.

Docker Compose упрощает разработку, поскольку дает возможность определять инфраструктуру, — включая сервисы приложений, сети и тома, — в едином файле. Docker Compose легко заменяет собой сразу несколько команд, в том числе `docker container create` и `docker container run`.

1. Загрузка Laravel и установка зависимостей

Для начала загрузим последнюю версию Laravel и установим зависимости программы, в том числе и Composer, менеджер пакетов PHP уровня приложения. Используем Docker, чтобы не устанавливать Composer глобально. Перейдем в домашний каталог и клонируем последнюю версию Laravel в каталог laravel-app:

```
cd ~
git clone https://github.com/laravel/laravel.git laravel-app
```

Затем переходим в появившийся каталог:

```
cd ~/laravel-app
```

Теперь через образ composer смонтируем каталоги проекта Laravel:

```
docker run --rm -v $(pwd):/app composer install
```

Опции `-v` и `—rm` в команде `docker run` создают контейнер, который привязывается к текущему каталогу до тех пор, пока он не будет удален. Содержимое каталога `~/laravel-app` скопируется в контейнер, а содержимое папки vendor, которую Composer создает внутри контейнера, будет скопировано в текущий каталог.

Теперь отредактируем привилегии каталога, все права на него нужно передать пользователю sudo:

```
sudo chown -R $USER:$USER ~/laravel-app
```

Это нужно для того, чтобы запускать процессы в контейнере через пользователя sudo.

2. Создание конфигурационного файла Docker Compose

Сборка приложений с помощью Docker Compose упрощает настройку и контроль версий в инфраструктуре. Настройка приложения осуществляется в файле docker-compose, в котором определяются сервисы веб-сервера, базы данных и приложения.

Откроем файл в редакторе gedit

```
sudo gedit ~/laravel-app/docker-compose.yml
```

В файле docker-compose нужно определить три сервиса: app, webserver и db:

```
version: '3'
services:
#PHP Service
app:
build:
context: .
dockerfile: Dockerfile
image: digitalocean.com/php
container_name: app
restart: unless-stopped
tty: true
environment:
SERVICE_NAME: app
SERVICE_TAGS: dev
working_dir: /var/www
networks:
- app-network
#Nginx Service
webserver:
image: nginx:alpine
container_name: webserver
restart: unless-stopped
tty: true
ports:
- "80:80"
- "443:443"
networks:
- app-network
#MySQL Service
db:
image: mysql:5.7.22
container_name: db
restart: unless-stopped
tty: true
ports:
- "3306:3306"
environment:
MYSQL_DATABASE: laravel
MYSQL_ROOT_PASSWORD: your_mysql_root_password
SERVICE_TAGS: dev
SERVICE_NAME: mysql
networks:
- app-network
#Docker Networks
networks:
app-network:
driver: bridge
```

В файл входят следующие сервисы:

app: определение сервиса, которое содержит приложение Laravel и запускает кастомный образ Docker, его мы определим позже. Также оно присваивает значение /var/www для параметру working_dir в контейнере.
webserver: загружает образ nginx:alpine и открывает порты 80 и 443.
db: извлекает образ mysql:5.7.22 и определяет новые переменные среды, в том числе имя базы данных для приложения и пароль пользователя root этой БД. Это определение также связывает порт хоста 3306 и такой же порт контейнера .

Свойство container_name определяет имя контейнера, которое должно совпадать с именем сервиса. Если вы не определите это свойство, Docker будет присваивать контейнерам случайные имена (по умолчанию он выбирает имя исторической личности и случайное слово, разделяя их символом подчеркивания).

Для простоты взаимодействия между контейнерами сервисы подключаются к соединительной сети app-network. Соединительная сеть использует программный мост, который позволяет подключенным к этой сети контейнерам взаимодействовать друг с другом. Драйвер устанавливает правила хоста автоматически, чтобы контейнеры из разных соединительных сетей не могли взаимодействовать напрямую. Это повышает безопасность приложений, поскольку взаимодействовать в таких условиях смогут только связанные сервисы.

3. Постоянное хранение данных

Docker предоставляет удобные средства для постоянного хранения данных. Для этого в данном тестовом приложении мы будем использовать тома и монтируемые образы. Тома очень гибкие в резервном копировании, их легко сохранять по истечении жизненного цикла контейнера, а монтируемые образы упрощают редактирование кода во время разработки и немедленно отражают изменения файлов или каталогов хоста в контейнерах. Используем оба варианта.

Монтируемые образы позволяют менять файловую систему хоста через работающие в контейнерах процессы, в том числе создавать, изменять и удалять важные системные файлы и каталоги. Это может повлиять на процессы в системе хоста, не связанные с Docker.

Для постоянного сохранения базы данных MySQL определим том dbdata в файле `docker-compose`, в определении сервиса db:

```
...
#MySQL Service
db:
...
volumes:
- dbdata:/var/lib/mysql
networks:
- app-network
...
```

Том dbdata используется для постоянного сохранения содержимого `/var/lib/mysql` в контейнере. Это позволяет останавливать и перезапускать сервис db, не теряя данных. В конеце файла определим том dbdata:

```
...
#Volumes
volumes:
dbdata:
driver: local
```

Теперь можно использовать этот том для разных сервисов. Затем добавим к сервису db привязку монтируемого образа для конфигурационных файлов MySQL.

```
...
#MySQL Service
db:
...
volumes:
- dbdata:/var/lib/mysql
- ./mysql/my.cnf:/etc/mysql/my.cnf
...
```

Это привяжет файл `~/laravel-app/mysql/my.cnf` к каталогу `/etc/mysql/my.cnf` в контейнере.

А теперь добавим монтируемые образы в сервис webserver. Образов будет два: один для кода приложения, а второй — для определения настроек Nginx.

```
#Nginx Service
webserver:
...
volumes:
- ./:/var/www
- ./nginx/conf.d/:/etc/nginx/conf.d/
networks:
- app-network
```

Первый образ привязывает код приложения из каталога `~/laravel-app` к каталогу `/var/www` внутри контейнера. Конфигурационный файл, добавляемый в `~/laravel-app/nginx/conf.d/`, также монтируется в папку `/etc/nginx/conf.d/` в контейнере, что позволяет менять содержимое каталога по мере необходимости.

Теперь добавим следующие привязки образов в сервис app для кода приложения и конфигурационных файлов:

```
#PHP Service
app:
...
volumes:
- ./:/var/www
- ./php/local.ini:/usr/local/etc/php/conf.d/local.ini
networks:
- app-network
```

Сервис app привязывает монтируемый образ каталога ~/laravel-app, который содержит код

приложения, к /var/www. Это позволяет ускорить разработку, поскольку все изменения в локальном каталоге приложения сразу же отразятся в контейнере. Также конфигурационный файл PHP ~/laravel-app/php/local.ini привязывается к файлу /usr/local/etc/php/conf.d/local.ini в контейнере. Мы создадим локальный конфигурационный файл PHP в разделе 5.

Теперь файл docker-compose имеет такой вид:

```
version: '3'
services:
#PHP Service
app:
build:
context: .
dockerfile: Dockerfile
image: digitalocean.com/php
container_name: app
restart: unless-stopped
tty: true
environment:
SERVICE_NAME: app
SERVICE_TAGS: dev
working_dir: /var/www
volumes:
- ./:/var/www
- ./php/local.ini:/usr/local/etc/php/conf.d/local.ini
networks:
- app-network
#Nginx Service
webserver:
image: nginx:alpine
container_name: webserver
restart: unless-stopped
tty: true
ports:
- "80:80"
- "443:443"
volumes:
- ./:/var/www
- ./nginx/conf.d/:/etc/nginx/conf.d/
networks:
- app-network
#MySQL Service
db:
image: mysql:5.7.22
container_name: db
restart: unless-stopped
tty: true
ports:
- "3306:3306"
environment:
MYSQL_DATABASE: laravel
```

```
MYSQL_ROOT_PASSWORD: your_mysql_root_password
SERVICE_TAGS: dev
SERVICE_NAME: mysql
volumes:
- dbdata:/var/lib/mysql/
- ./mysql/my.cnf:/etc/mysql/my.cnf
networks:
- app-network
#Docker Networks
networks:
app-network:
driver: bridge
#Volumes
volumes:
dbdata:
driver: local
```

Теперь пора создать пользовательский образ приложения.

4. Создание Dockerfile

Docker позволяет настраивать среду внутри отдельных контейнеров с помощью файла Dockerfile, который создает пользовательские образы.

Файл Dockerfile должен находиться в каталоге `~/laravel-app`. Создадим этот файл:

```
sudo gedit ~/laravel-app/Dockerfile
```

Этот Dockerfile будет определять базовый образ и необходимые команды для сборки образа приложения Laravel. Добавим в файл следующие строки:

```
FROM php:7.2-fpm
# Copy composer.lock and composer.json
COPY composer.lock composer.json /var/www/
# Set working directory
WORKDIR /var/www
# Install dependencies
RUN apt-get update && apt-get install -y \
build-essential \
mysql-client \
libpng-dev \
libjpeg62-turbo-dev \
libfreetype6-dev \
locales \
zip \
jpegoptim optipng pngquant gifsicle \
vim \
unzip \
git \
curl
# Clear cache
RUN apt-get clean && rm -rf /var/lib/apt/lists/*
# Install extensions
```

```
RUN docker-php-ext-install pdo_mysql mbstring zip exif pcntl
RUN docker-php-ext-configure gd --with-gd --with-freetype-dir=/usr/include/ -
-with-jpeg-dir=/usr/include/ --with-png-dir=/usr/include/
RUN docker-php-ext-install gd
# Install composer
RUN curl -sS https://getcomposer.org/installer | php -- --install-
dir=/usr/local/bin --filename=composer
# Add user for laravel application
RUN groupadd -g 1000 www
RUN useradd -u 1000 -ms /bin/bash -g www www
# Copy existing application directory contents
COPY . /var/www
# Copy existing application directory permissions
COPY --chown=www:www . /var/www
# Change current user to www
USER www
# Expose port 9000 and start php-fpm server
EXPOSE 9000
CMD ["php-fpm"]
```

Сначала Dockerfile создает образ на основе образа php:7.2-fpm. Это образ Debian с предустановленным экземпляром PHP FastCGI PHP-FPM. Также этот файл устанавливает необходимые пакеты для Laravel: mcrypt, pdo_mysql, mbstring, imagick и composer.

Директива RUN определяет команды для обновления, установки и настройки параметров внутри контейнера, включая выделенного пользователя и группу www. Директива WORKDIR указывает рабочий каталог приложения (в данном случае это каталог `/var/www`).

Используя отдельного пользователя и группу с ограниченными правами доступа, вы снижаете уязвимости при запуске контейнеров Docker (по умолчанию они запускаются с правами root). Чтобы не запускать контейнер с привилегиями root, мы создали пользователя www с правами на чтение и запись для каталога /var/www.
Это делается с помощью команды COPY и флага —chown для копирования прав каталога приложения.

Команда EXPOSE открывает в контейнере порт 9000 для сервера php-fpm. CMD определяет команду, которая будет запускаться после создания контейнера. Здесь CMD содержит команду php-fpm, которая запускает сервер.

Теперь пора определить конфигурации PHP.

5. Настройка PHP

Итак, мы определили инфраструктуру в файле docker-compose. Теперь можно настроить сервис PHP в качестве процессора для входящих запросов Nginx.

Для настройки PHP нужно создать файл local.ini в каталоге php. Это файл, который в разделе 2 мы привязали к файлу /usr/local/etc/php/conf.d/local.ini в контейнере. Имея этот файл, мы можем игнорировать файл по умолчанию php.ini, который PHP считывает при запуске.

```
mkdir ~/laravel-app/php
```

Открываем для редактирования:

```
sudo gedit ~/laravel-app/php/local.ini
```

Чтобы продемонстрировать настройку PHP, добавим следующий код для установки ограничений размера выгруженных файлов:

```
upload_max_filesize=40M
post_max_size=40M
```

Директивы upload_max_filesize и post_max_size задают максимальный допустимый размер выгружаемых файлов и показывают, как задавать конфигурации php.ini из local.ini. Все параметры конфигурации PHP, которые можно игнорировать, поместим в файл local.ini.

Теперь пора настроить веб-сервер.

6. Настройка Nginx

Теперь можно настроить Nginx на поддержку PHP-FPM в качестве сервера FastCGI для обслуживания динамического контента. Для Nginx нужно создать в папке `~/laravel-app/nginx/conf.d/` файл app.conf с конфигурацией сервисов.

Сперва создадим каталог `nginx/conf.d/`:

```
mkdir -p ~/laravel-app/nginx/conf.d
```

Затем создадим файл app.conf:

```
sudo gedit ~/laravel-app/nginx/conf.d/app.conf
```

Код, для определения конфигурации Nginx:

```
server {
listen 80;
index index.php index.html;
error_log  /var/log/nginx/error.log;
access_log /var/log/nginx/access.log;
root /var/www/public;
location ~ \.php$ {
try_files $uri =404;
fastcgi_split_path_info ^(.+\.php)(/.+)$;
fastcgi_pass app:9000;
fastcgi_index index.php;
include fastcgi_params;
fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
fastcgi_param PATH_INFO $fastcgi_path_info;
}
location / {
try_files $uri $uri/ /index.php?$query_string;
gzip_static on;
}
}
```

В блоке location для php директива fastcgi_pass указывает, что сервис app прослушивает сокет TCP по порту 9000. Благодаря этому сервер PHP-FPM прослушивает запросы через сеть,

а не через сокет Unix. Сокет Unix имеет небольшое преимущество в скорости по сравнению с сокетом TCP, однако у него нет сетевого протокола и он пропускает сетевой стек. Если хосты находятся в одной системе, использование сокета Unix может иметь смысл, но если сервисы работают на разных хостах, сокет TCP гораздо лучше, поскольку позволяет подключаться к распределенным сервисам. Поскольку контейнеры app и webserver работают на разных хостах, в данной конфигурации сокет TCP будет эффективнее.

Все изменения в каталоге nginx/conf.d/ прямо отразятся в контейнере webserver.

7. Настройка MySQL

Настроив PHP и Nginx,можем включить MySQL как базу данных для приложения.

Для MySQL нужно создать файл my.cnf в каталоге mysql. Этот файл мы привязали к файлу /etc/mysql/my.cnf внутри контейнера. Привязка монтируемого образа позволяет игнорировать все параметры my.cnf, когда это потребуется.

Чтобы посмотреть, как это работает, давайте добавим в файл my.cnf параметры, которые включают лог общих запросов и задают лог-файл.

Создадим каталог mysql и пустой файл my.cnf:

```
mkdir ~/laravel-app/mysql
nano ~/laravel-app/mysql/my.cnf
```

Поместим в файл следующий код, чтобы включить лог запросов и задать расположение лога:

```
[mysqld]
general_log = 1
general_log_file = /var/lib/mysql/general.log
```

Файл my.cnf включает лог, задавая параметру general_log значение 1. Параметр general_log_file указывает, где будут храниться логи.

Все готово, пора запускать контейнеры.

8. Запуск контейнеров и изменение настроек среды

Итак, мы определили все сервисы в файле docker-compose и создали конфигурации для всех сервисов. Теперь можно запустить контейнеры. В качестве последнего шага мы создадим копию файла .env.example, который Laravel включает по умолчанию, и назовем ее .env, поскольку именно такой файл Laravel использует для определения среды:

```
cp .env.example .env
```

После запуска контейнеров мы вставим в этот файл параметры.

Теперь все сервисы определены в файле docker-compose. Запустим одну команду, которая запустит все контейнеры, создаст тома и настройки и подключит сети:

```
docker-compose up -d
```

При первом запуске команда docker-compose up загрузит все необходимые образы Docker, что может занять некоторое время. После загрузки образов на локальный компьютер Compose

создаст контейнеры. Флаг -d преобразует процесс в демон, что позволяет поддерживать работу контейнеров в фоновом режиме.

После завершения этого процесса используем следующую команду, чтобы запросить список всех запущенных контейнеров:

```
docker ps
```

Увидим следующий вывод с данными о контейнерах app, webserver и db:

```
CONTAINER ID        NAMES               IMAGE
STATUS              PORTS
c31b7b3251e0        db                  mysql:5.7.22                       Up
2 seconds        0.0.0.0:3306->3306/tcp
ed5a69704580        app                 digitalocean.com/php               Up
2 seconds        9000/tcp
5ce4ee31d7c0        webserver           nginx:alpine                       Up
2 seconds        0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp
```

В этом выводе CONTAINER ID — это уникальный идентификатор контейнера, а NAMES перечисляет имена сервисов. IMAGE определяет имя образа каждого контейнера, а STATUS предоставляет данные о состоянии.

Теперь отредактируем файл .env в контейнере app, чтобы добавить необходимые параметры.

Откроем файл с помощью docker-compose exec, что позволяет запускать определенные команды в контейнерах. В данном случае команда откроет файл для редактирования:

```
docker-compose exec app nano .env
```

В блоке DB_CONNECTION и определим особенности настройки системы.

DB_HOST – нужно указать контейнер базы данных db.
DB_DATABASE – укажите здесь БД laravel.
DB_USERNAME – укажите имя пользователя БД. В этом случае мы используем laraveluser.
DB_PASSWORD – надежный пароль этого пользователя.

```
DB_CONNECTION=mysql
DB_HOST=db
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=laraveluser
DB_PASSWORD=your_laravel_db_password
```

Затем настраиваем ключ для приложения Laravel с помощью команды php artisan key:generate. Команда сгенерирует ключ и скопирует его в файл .env, что защитит сессии пользователя и шифрованные данные:

```
docker-compose exec app php artisan key:generate
```

Теперь для запуска приложения имеются все необходимые настройки среды. Чтобы

кэшировать эти настройки для ускорения загрузки приложения, запустим команду:

```
docker-compose exec app php artisan config:cache
```

Конфигурации будут загружены в /var/www/bootstrap/cache/config.php в контейнере.

На этом этапе можно проверить. Откроем в браузере сайт http://your_server_ip. На экране появится главная страница приложения Laravel.

Теперь можно перейти к настройке данных пользователя БД laravel в контейнере db.

9. Создание пользователя MySQL

Установка MySQL по умолчанию предоставляет только учетную запись root с неограниченными привилегиями доступа к серверу СУБД. Обычно при работе с базой данных лучше не использовать администратора, root. Вместо этого лучше создать специального пользователя для базы данных нашего приложения.

Чтобы создать его, запустим интерактивную оболочку bash в контейнере db с помощью команды docker-compose exec:

```
docker-compose exec db bash
```

Внутри контейнера входим в MySQL как root:

```
mysql -u root -p
```

Для начала проверим наличие базы данных laravel, которая была определена в файле docker-compose. Запустим show databases для проверки существующих баз данных:

```
show databases;
```

В выводе увидим БД laravel:

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| laravel            |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

Затем создаем пользователя, у которого будет доступ к этой базе данных. Используем имя laraveluser. Имя пользователя и пароль должны соответствовать учетным данным, указанным ранее в файле .env.

```
GRANT ALL ON laravel.* TO 'laraveluser'@'%' IDENTIFIED BY
'your_laravel_db_password';
```

Чтобы сообщить серверу MySQL об изменениях, сбросим привелегии:

```
FLUSH PRIVILEGES;
```

Закроем MySQL:

```
EXIT;
```

И выходим из контейнера:

```
exit
```

10. Миграция данных

Теперь приложение запущено. Осталось произвести миграцию данных:

```
docker-compose exec app php artisan migrate
```

В случае успешной миграции, увидим следующее:

```
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table
```

Теперь приложение Laravel работает! Убедимся в этом перейдя в браузере по адресу:

```
http://your_server_ip
```

2 .3 Конфигурирование операционной среды разработки

С помощью консольной команды выведем в файл все установленные программы с помощью следующей команды:

```
dpkg --get-selections | grep -v deinstall > backup.txt
```

Получим примерно следующий список программ:

```
account-plugin-facebook         install
account-plugin-google           install
accountsservice                 install
acl                             install
acpi-support                    install
acpid                           install
activity-log-manager            install
adduser                         install
adium-theme-ubuntu              install
adwaita-icon-theme              install
aisleriot                       install
alsa-base                       install
alsa-utils                      install
anacron                         install
```

```
apache2                         install
apache2-bin                     install
apache2-data                    install
apache2-utils                   install
 ...
xz-utils                        install
yelp                            install
yelp-xsl                        install
zeitgeist-core                  install
zeitgeist-datahub               install
zenity                          install
zenity-common                   install
zip                             install
zlib1g:amd64                    install
```

Имея список установленных программ в файле backup.txt имеется возможность быстро установить эти программы на чистую операционную систему с помощью следующей команды

```
sudo dpkg --set-selections < backup.txt
```

Резервное архивирование образа системы

Ubuntu позволяет из всей файловой системы сделать архив. Фактически, мы можем потом развернуть этот архив на любой машине и получить полноценную операционную систему после настройки драйверов [29].

Для архивирования достаточно просто использовать утилиту *tar* и не нужны сторонние программы. Для создания архива используем такую команду:

```
sudo tar czf /backup.tar.gz --exclude=/backup.tar.gz --exclude=/home --
exclude=/media --exclude=/dev --exclude=/mnt --exclude=/proc --exclude=/sys -
-exclude=/tmp /
```

В этой команде все достаточно просто несмотря на ее запутанность. Опция «*c*» означает, что нужно создать архив (*Create*), *z* — включает сжатие *Gzip*. Затем с помощью опции - *f* мы указываем файл, в который нужно сохранить результат.

Затем с помощью серии опций --*exclude* мы исключаем из архива сам файл архива, домашний каталог и директории с виртуальными файловыми системами. В самом конце указываем папку, с которой стоит начать сбор данных — /.

Процесс займет очень много времени, но когда он завершится, мы получим полную резервную копию системы в корневом каталоге.

Для восстановления системы, нужно загрузиться с *LiveCD/USB*, и примонтировать корневой каталог в /*mnt*/. Затем подключить носитель с резервной копией и выполнить команду для распаковки. В Linux всё это можно выполнить с помощью одной консольной команды:

```
sudo tar xf /run/media/имя_носителя/backup.tar.gz -C /mnt
```

2 .4 Выводы по главе 2

Во второй главе представлена операционная среда разработки. Рассмотрен системный и серверный инструментарий, а также произведено конфигурирование операционной системы Xubuntu.

Исходя из задач, поставленных перед разрабатываемой системой, приходится оперировать большим количеством информации. Выходит, что должна быть база данных позволяющая производить действия над данными, а так же ограничивать права доступа к информации. Для решения всех перечисленных задач, было решено использовать следующие программные продукты:

– СУБД MySQL;

– web-сервер Apache;

– PHPMyAdmin - веб-интерфейс для администрирования СУБД MySQL;

– язык написания сценариев PHP5;

– язык разметки страниц гипертекста HTML;

– Notepad - текстовый редактор;

– PHPStorm — интеграционная среда разработки

В качестве СУБД была использована MySQL с системой хранения данных InnoDB, которая отвечает всем необходимым требованиям:

– реализует архитектуру клиент-сервер, что значительно упрощает клиентские приложения (все работы по обслуживанию БД будет выполнять сервер БД);

– работа с данными осуществляется по средствам языка структу-рированных запросов SQL, что приводит к снижению сетевого трафика;

– наличие необходимых средств для распределения прав доступа, что упрощает администрирование БД и повышает их защищенность.

Основные конкурентные преимущества MySQL:

– производительность (поэтому многие компании например такие, как Google и Yahoo используют именно MySQL);

– масштабируемость (к примеру, в компании Omniture в реальном масштабе времени используется 7000 серверов MySQL);

– надежность (в коде проприетарных продуктов содержится намного больше уязвимостей);

– простота использования, простота внедрения (установка вместе с скачкой займёт в среднем 15 минут);

– открытая и модульная разработка;

– низкие совокупные затраты (платить нужно только при потребности в поддержке).

Для разработки, тестирования и реализации подсистемы необходимо серверное программное обеспечение. Сервер Apache - Web-сервер с открытым исходным кодом, один из самых популярных в мире, на нем построено около двух третей хостов Интернета. Основные достоинства Apache – это надежность, безопасность и гибкость настройки. Apache позволяет подключать различные модули, добавляющие в него новые возможности.

Для разработки был выбран язык написания сценариев. PHP - это широко используемый

язык сценариев общего назначения с открытым исходным кодом. Важным преимуществом языка PHP перед такими языками, как языков Perl и C заключается в возможности создания HTML документов с внедренными командами PHP. Так же отличием PHP от какого-либо кода, выполняющегося на стороне клиента, например, JavaScript, является то, что PHP-скрипты выполняются на стороне сервера. Как средство разработки Web-приложений PHP сейчас является одним из самых популярных языков. PHP прост для освоения, и вместе с тем способен удовлетворить запросы профессиональных программистов.

Так же для разработки системы понадобится язык разметки страниц гипертекста HTML (HyperText Markup Language - язык маркировки гипертекстов). HTML содержит набор средств для описания визуальных свойств (позиция, размер, цвет и т.д.) различных элементов, в частности текста или графики. Под гипертекстовым документом подразумеваются документы с гипертекстовыми ссылками - указателями на другие гипертекстовые документы.

3 Этапы разработки клиент-серверного web-приложения

Разработка клиент-сервреного web-приложения состоит из следующих этапов:

– Разработка макета приложения;

– Программирование серверной части;

– Программирование клиентской части.

Размещение web-приложения на репозиториях и на хостинговых плащадках.

Макет приложения, в свою очередь, состоит из следующих блоков:

Предметика. В этот блок входит шапка сайта и верхний блок меню. Если пользователь заинтересовался блоком предметика, он переходит к следующему блоку.

Конкретика. В этот блок входит боковой блок мению (или блоки) и центральная часть сайта.

Для пользователя. В этот блок входит всё то, что не относится на прямую к тематике, предметике и конкретике: реклама, внешние виджеты (погоды, курсы валют, новостные ленты).

В зависимости от выбранного шаблона проектирования технологии, используемые на серверной и клиентской стороне разнятся [30].

3 .1 Макет главной страницы web-приложения

Даже идеальное браузерное отображение не делает никакой разницы между десктопными браузерами и принтерами или между мобильными устройствами и голосовым браузером. Чтобы решить эту проблему, *W3C* создала список медиатипов для классификации каждого браузера или устройства по медиакатегориям. Медиатипы могут принимать значения: all, braille, embossed, handheld, print, projection, screen, speech, tty и tv.

Все эти медиатипы созданы с одной целью: чтобы мы могли лучше проектировать дизайн для каждого типа браузера или устройства, просто загружая нужный *CSS*.

Следовательно, устройство с экраном будет игнорировать *CSS*, созданный для медиатипа print, и наоборот. А для стилевых правил, которые применимы ко всем устройствам, в

спецификации создана супергруппа all. На практике это означает правку media-атрибута ссылки.

Однако, вместо того, чтобы для каждого медийного устройства создавать различные файлы стилей, я использовал следующие более удобные возможности реализации адаптивности web-приложения:

1. Media-блоки в самих стилях:

```
@media screen {
        body {
                font-size: 100 %;
        }
}
@media print {
        body {
                font-size: 15 pt;
        }
}
```

Такой подход позволяет адаптироваться не под само медиа-устройство, а под размер экрана любого устройства. Например, для устройств, размер которых не превышает 768px прописывались отдельные стили.

```
#page {
        margin: 36px auto;
        width: 90 %;
}
@media screen and (max-width: 768px) {
        #page {
                position: relative;
                margin: 20px;
                width: auto;
        }
}
```

2. Библиотека Bootstrap (*http://getbootstrap.com*), основная задача которой как раз и заключается в реализации адаптивности [31].

Интуитивно простой и в тоже время мощный фрэймворк *Bootstrap* не только облегчил разработку макета web-приложения, но и ускорил этот процесс. Библиотека поставляется вместе с несколькими файлами стилей, среди них есть *bootstrap.min.css* и *bootstrap.css*. Для увеличения скорости загрузки библиоеки на стороне клиента, использовалась минимизированная версия (*min.css*). Это тот же код, просто компактнее. Минимизированы таблицы стилей используют меньшую ширину канала, что есть хорошо, особенно в рабочем (*production*) среде. Поэтому, так как нет особых причин использовать полную версию, останавливаемся на минимизированной, которая доступна по адресу:

```
https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css
```

Либо на странице загрузки библотеки.

Особенности вёрстки макета приложения

Для разбиения сайта на блоки использовался единичный набор *.col-md-** классовой разметки, была создана базовая система разметки, которая запускается одновременно на мобильные устройства и планшетные устройства (от экстра маленьких до малых диапазонов). Размещаются данные столбцы разметки в любом классе *.row*.

Чтобы колонки иначе складывались на небольших устройствах, используем очень маленькие xs или средние md классы разметки устройства добавляя *.col-xs-** *.col-md-** к столбцам.

```
<div class="row">
  <div class="col-xs-12 col-md-8">.col-xs-12 .col-md-8</div>
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
</div>
<div class="row">
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
  <div class="col-xs-6 col-md-4">.col-xs-6 .col-md-4</div>
</div>
<div class="row">
  <div class="col-xs-6">.col-xs-6</div>
  <div class="col-xs-6">.col-xs-6</div>
</div>
```

Для перемещения колонки направо использовались классы *.col-md-offset-**. Эти классы увеличивают отступ слева столбца * колонки. Например, *.col-md-offset-4* сдвигает *.col-md-4* пропуская один такой же столбец.

```
<div class="row">
  <div class="col-md-4">.col-md-4</div>
  <div class="col-md-4 col-md-offset-4">.col-md-4 .col-md-offset-4</div>
</div>
<div class="row">
  <div class="col-md-3 col-md-offset-3">.col-md-3 .col-md-offset-3</div>
  <div class="col-md-3 col-md-offset-3">.col-md-3 .col-md-offset-3</div>
</div>
<div class="row">
  <div class="col-md-6 col-md-offset-3">.col-md-6 .col-md-offset-3</div>
</div>
```

Зебра-чередование для любой строки таблицы осуществляется с помощью класса *table* с добавлением класса *table-striped*.

```
<table class="table table-striped">
  ...
</table>
```

Для оконтовки таблицы и яичеек необходимо использовался класс *table-bordered*.

```
<table class="table table-bordered">
  ...
</table>
```

Кроме того для ссылок и кнопок использовался класс *btn* со следующими контекстными

классами:

.active применяет цвет при наведении на конкретную строку или ячейку;

.success указывает на успешное или позитивное действие;

.info указывает на нейтральные информативные изменения;

.warning указывает на предупреждения, которые могут потребовать внимания;

.danger Указывает на опасное или потенциально негативное действие.

Для остальных элементов дизайна были использованы собственные стили и скрипты. В результате была сверстана адаптивная страница, макет которой предоставлен на изображении.

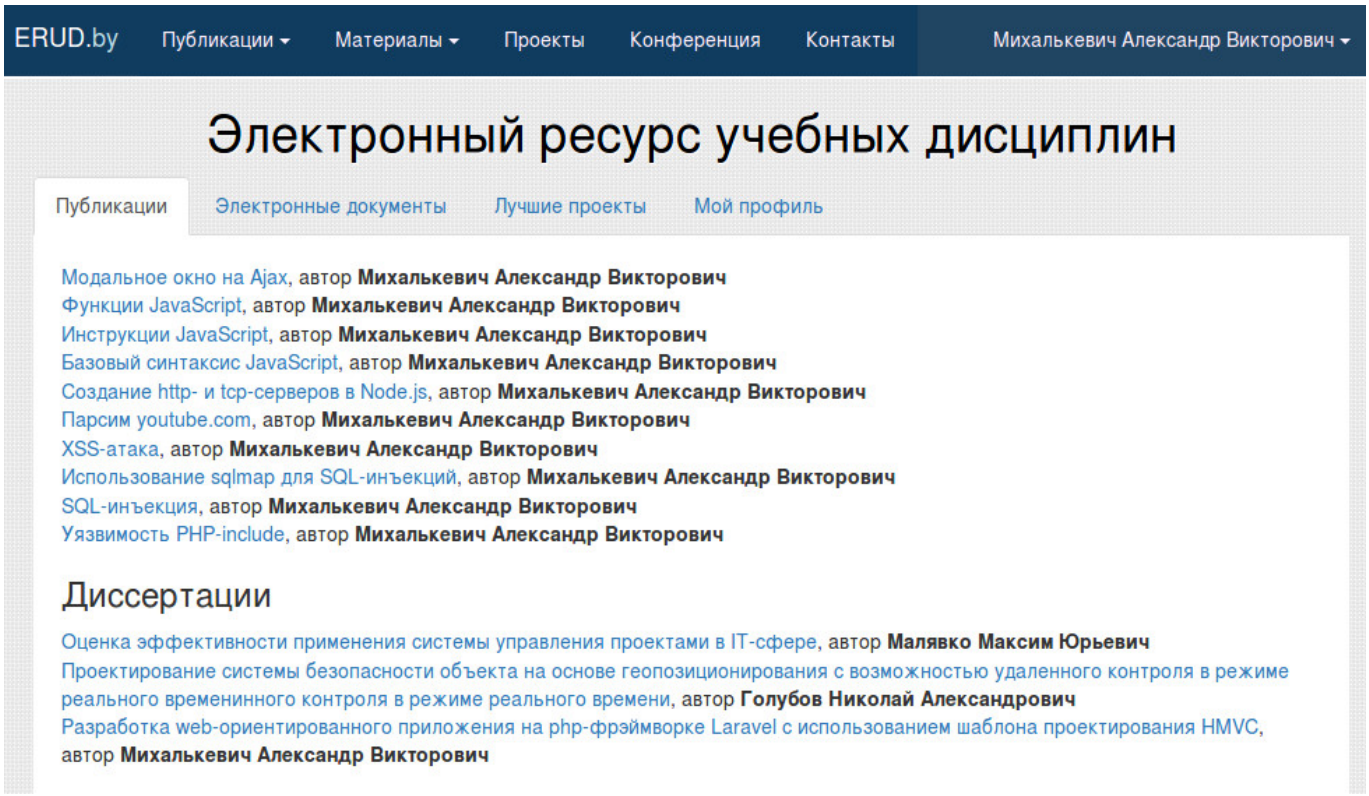


Рисунок 5. Макет главной страницы ресурса erud.by

3 .2 Программирование серверной части web-приложения

Фреймворк Laravel пользуется популярностью за счет ряда факторов:

он объектно-ориентированный;
соблюдается модель MVC;
доступна поддержка нескольких баз данных.

Кроме того, в нем доступна масса инструментов для развертывания приложений и упрощения веб-разработки [31-А].

Чтобы запустить на веб-сервере Laravel, в Ubuntu необходимо удовлетворить ряд зависимостей. Все необходимые для работы компоненты имеются в наличии на виртуальной машине Laravel Homestead. Поэтому разработчики рекомендуют использовать именно ее для запуска на локальной машине.

Тем не менее, развертывать и тестировать приложение придется на веб-серверах с Linux, а не на Homestead. Чтобы установить Laravel на таком веб-сервере, на нем должны быть

установлена версия PHP >= 7.2.0 и следующие расширения PHP:

BCMath PHP Extension;
Ctype PHP Extension;
JSON PHP Extension;
Mbstring PHP Extension;
OpenSSL PHP Extension;
PDO PHP Extension;
Tokenizer PHP Extension;
XML PHP Extension.

В Ubuntu эти зависимости можно установить при помощи команды в терминале:

```
sudo apt install php php-mysql php-mbstring php-tokenizer php-xml php-json php-common php-dompdf
```

Кроме того, у *Laravel* еще есть несколько требований к серверу:

PHP >= 7.2

OpenSSL PHP Extension

Если с момента последнего запуска composer прошло более 30 дней, то необходимо обновить *composer*.

```
composer slef-update
```

После обновления *composer* запустим команду установки *Laravel* [32].

```
Composer create-project laravel/laravel -—prefer-dist
```

Laravel установлен.

Далее необходимо изменить права на запись. Папки внутри *storage* должны быть доступны web-серверу для записи. Воспользуемся командой *chmod*.

```
sudo chmod -R 777 storage
```

Изабвляеся от *public* в запросах. Для того, чтобы слово *public* не присутстовало в запросах приложения, создадим файл *.htaccess*, который будет перенаправлять запрос.

```
 RewriteEngine On
 RewriteRule ^(.*)$ public/$1 [L]
```

Теперь все запросы автоматически перенаправляются в папку *public*, а все остальные папки проекта стали закрытыми.

Маршрутизация проекта

Все маршруты прописаны в файле *routes/web.php*. Так как каждый маршрут является неотъемлимой частью всего web-приложения, рассмотрим их все:

```
//--------------------------------------------------------------------
Auth::routes();
Route::get('/home', 'HomeController@index');
```

```
//ajax
Route::post('ajax', 'AjaxController@postIndex');
Route::post('/ajax/public/{id}/', 'Ajax\MenuController@postPublics');
Route::post('ajax/showCatalog', 'AjaxController@postCatalog');
Route::post('ajax/showArticle', 'AjaxController@postArticle');
Route::post('/ajax/addAnswer', 'AjaxController@postAnswer');
Route::post('/ajax/register', 'Ajax\MenuController@postRegister');
Route::post('/ajax/common/{id}/{into}', 'Ajax\MenuController@postCommon');
Route::group(['middleware' => ['auth']], function () {
Route::get('/delete/{id}', 'PublicController@getPublicDelete');
//for student
Route::get('work/delete/{id}','Student\CourseController@getDelete');
Route::get('work/edit/{id}','Student\CourseController@getEdit');
Route::get('lab/delete/{id}','Student\LabController@getDelete');
Route::post('lab/edit/{id}','Student\LabController@postEdit')->middleware('te
acher');
Route::post('student/work/','Ajax\StudentController@postWork');
Route::post('student/work/edit/{id}','Student\CourseController@postEdit');
Route::post('/course/into/{id?}', 'Student\CourseController@postInto');
Route::post('addCourse', 'Student\CourseController@addCourse');
// for user
Route::get('home/edit', 'HomeController@edit');
Route::get('home/edit/{id}','HomeController@getOne');
Route::get('home/projects', 'Auth\ProjectController@getIndex');
Route::get('home/students', 'Auth\GroupController@getGroups');
Route::get('/profile', 'Auth\ProfileController@getIndex');
Route::get('course/add/{cat_id}/{course_id}','Auth\CourseController@getAdd');
// ajax
Route::get('/home/ajax/group','Ajax\GrController@getIndex');
Route::post('/ajax/into/', 'Ajax\MenuController@postInto');
Route::post('/ajax/work/', 'Ajax\LabsController@postForm');
Route::post('/ajax/editArticle', 'AjaxController@postEditArticle');
Route::post('/ajax/editMaterial','AjaxController@postMaterial');
Route::post('/ajax/editCatalog', 'AjaxController@postEditCatalog');
Route::post('/ajax/addPublic', 'AjaxController@postPublic');
Route::post('/ajax/editCatalogStudent',
'Ajax\StudentController@postEditCatalog');
// post
Route::post('home/edit/{id}','HomeController@postOne');
Route::post('home/group/add', 'Auth\GroupController@postGroup');
Route::post('/profile', 'Auth\ProfileController@postIndex');
Route::post('home/project','Auth\ProjectController@postIndex');
Route::post('addCatalog/{catalog?}', 'FormController@addCatalog');
Route::post('addArticle', 'FormController@addArticle');
Route::post('editArticle', 'FormController@editArticle');
Route::post('materials/link/{url}', 'MaterialController@postLink');
Route::post('materials/lists/{id}', 'MaterialController@postList');
Route::post('/form/into/{id?}', 'FormController@postInto');
Route::get('/{chapter}/{cat}/{art}/edit', 'Auth\EditController@getArticle');
Route::post('/{chapter}/{cat}/{art}', 'Auth\EditController@postArticle');
Route::post('lab/{id}','Ajax\LabsController@postAdd');
```

```
Route::post('/ajax/course', 'Ajax\CourseController@postAll');
//files
Route::get('delete/{id}', 'UploaderController@getDelete');
Route::post('laravel-filemanager/upload','UploaderController@postFirst');
Route::post('/uploader', 'LibsController@getUpload');
});
Route::post('/ajax/labs/', 'Ajax\LabsController@postCourse');
//othes
Route::get('/', 'IndexController@getChapt'); //controller for main page
Route::get('projects','ProjectController@getAll');
Route::get('project/{id}','ProjectController@getOne');
Route::get('/conference', 'ConferenceController@getIndex');
Route::get('/eruds', 'ThesesController@getEruds'); //controller for theses
Route::get('/theses', 'ThesesController@getIndex'); //controller for theses
Route::get('/links', 'LinkController@getIndex');
Route::get('/lists', 'ListController@getIndex');
Route::get('/courses', 'ThesesController@getCourses');
Route::get('/course/{id}', 'WorkController@getCourse');
Route::get('/lab/{id}', 'WorkController@getCourse');

Route::get('/media/{name}','MediaController@getOne');
Route::get('/publics/add/{catalolg_id}/{public_id}',
'PublicController@getIndex');
Route::get('/document/doc/{url}','DocsController@getDoc'); //in doc
Route::get('/document/pdf/{url}','DocsController@getPdf'); //in pdf
Route::get('/document/html/{url}','DocsController@getHtml'); //in html
Route::get('/course/doc/{id}','DocsController@getCourseDoc');
Route::get('/course/html/{id}','DocsController@getCourseHtml');
Route::get('/articles', 'ThesesController@getArticles');
Route::get('/keywords','KeyController@getAll');
Route::get('/authors','AuthorController@getAll');
Route::get('/author/{name}','AuthorController@getOne');
Route::get('/user/{id}', 'UserController@getOne');
Route::get('key/{id}','KeyController@getIndex');
Route::get('/page/{url}','PageController@getIndex');
Route::get('/{chapter}', 'IndexController@getChapt'); //chapters of erud
Route::get('/{chapter}/{cat}', 'IndexController@getCat'); //chapter and
catalog
Route::get('/{chapter}/{cat}/{art}', 'IndexController@getArt');
Route::post('backend/upload','UploaderController@files');
```

Подключение базы данных

Сперва создадим базу данных. Для этого перейдем в *PHPMyAdmin*.

```
http://127.0.0.1/phpmyadmin/
```

Создаем базу с именем *web*.

Поключение к базе осуществляется в файле *.env* в корне проекта. Для этого переопределим переменные окружающей среды.

```
DB_CONNECTION=mysql
```

```
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=web
DB_USERNAME=root
DB_PASSWORD=
```

Artisan

Laravel поставляется совместно с встроенным программным обеспечением — *Artisan*. *Artisan* — название интерфейса командной строки, входящей в состав *Laravel*. Он предоставляет полезные команды для использования во время разработки вашего приложения. Работает на основе мощного компонента *SymfonyConsole* [31-Б].

Чтобы вывести все доступные команды *Artisan*, воспользуемся командой *list*:

```
php artisan list
```

Рассмотрим используемые в проекте команды подробнее:
make:command — создаёт новый класс команды
make:console — создаёт новую команду *Artisan*
make:controller — создаёт новый класс контроллера
make:event — создаёт новый класс события
make:middleware — создаёт новый класс промежуточного ПО
make:migration — создаёт новый файл миграции
migrate - выполнение существующих миграций
make:model — создаёт новый класс модели, опционально и миграцию
make:provider — создаёт новый класс поставщика услуг
make:request — создаёт новый класс запроса формы
event:generate — генерирует пропущенные события и обработчики

Каждая команда также включает и инструкцию, которая отображает и описывает доступные аргументы и опции для команды. Чтобы её вывести, необходимо добавить слово *help* перед командой:

```
php artisan  help migrate
```

Для определения текущей версии *Laravel*, можно воспользоваться опцией — *version*.

```
php artisan –version
```

Усовершенствование концепции *HMVC* в фрэймворке *Laravel* заключается во введении вспомгательных и промежуточных решений применяемых во взаимодействии контроллеров с моделями. Такими решениями являются: миграции, *middleware*, или подготовительное программное обеспечение, *serviceProvider*, или классы библиотек, другие решения [47].

Миграции

Миграции базы данных являются весьма полезны для любого проекта, особенно для проектов с несколькими разработчиками, позволяя иметь последнюю версию базы данных у всех разработчиков. В *Laravel* для этого достатчно выполнить одну команду в командной

строке [33].

Для создания и выполнения миграций воспользуемся интерфейсом командной строки *Artisan*.

Откроем консоль командной строки из папки, где расположен файл *artisan*. В консоли введем следующие команды:

```
php artisan make:model -m Catalog //команда для создания файлов модели и миграции Catalog
php artisan make:model -m Article //команда для создания файлов модели и миграции Article
```

Данные команды создадут файлы миграций вместе с моделями. Для реализации миграций можно воспользоваться примером готовой миграции из файла 2014_10_12_000000_create_users_table.php.

```
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Файл миграции представлен двумя методами, up() и down(). Метод up() содержит схемы,

используемые при создании таблиц, а метод down() вызвается, когда необходимо удалить таблицу.

Метод down() будет одинаков для всех миграций, а вот методы up(), реализуюище сами таблицы, необходимо рассмотреть.

Выполняемый метод up() файла миграции таблицы categories содержит следующий код.

```
public function up()
{
    Model::unguard();
    Schema::create('categories',function(Blueprint $table){
        $table->increments("id");
        $table->string("name");
        $table->string("parent_id")->nullable();
        $table->enum("showhide", ["show", "hide"])->nullable();
        $table->string("type")->nullable();
        $table->timestamps();
        $table->softDeletes();
    });
}
```

Выполняемый метод up() файла миграции таблицы articles содержит следующий код.

```
public function up()
 {
     Schema::create('articles', function (Blueprint $table) {
       $table->increments('id');
       $table->integer('user_id')->default(0);
       $table->integer('category_id')->default(0);
       $table->string('name');
       $table->text('description');
       $table->string('url');
       $table->enum('showhide', array('show','hide'))->default('show');
       $table->timestamps();
     });
 }
```

Если база данных подключена, можно приступить к выполнению миграций:

```
php artisan migrate
```

Данная команда создает несколько таблиц, в том числе таблицы users, catalogs и articles.

Controller

Контроллеры хранятся в папке *app/controllers/*. Этот путь, в свою очередь определен в файле *composer.json* в настройке *classmap*.

Все контроллеры должны наследовать класс *BaseController*. Этот класс также может хранится в папке *app/controllers*, и в него можно поместить общую логику для других контроллеров. *BaseController* расширяет базовый класс *Controller*.

В приложении имеется более десяти контроллеров. Основными являются *IndexController,*

ThesesController и DocsController.

Создаем контроллеры:

```
php artisan make:controller IndexController
php artisan make:controller ThesesController
php artisan make:controller DocsController
```

Конечной точкой маршрутизатора является контроллер. Контроллер принимает переменные запроса (если таковые имеются) и содержит логику приложения. Логика приложения может быть прописана либо в самих контрллерах, либо во вспомогательных файлах, к которым контроллер обращается.

Каждый из контроллеров выполняется независимо от других, и содержит следующую логику:

IndexController - главный контроллер приложения, который содержит экшны (методы) реализующие главную страницу, список открытых категорий и открытую статью выбранной категории

ThesesController - контроллер содержащий экшны диссертаций

DocsController - контроллер, реализующий ответы страницы в виде различных форматов: *.doc, .pdf, .html* и другие.

Рассмотрим контроллер *IndexController.php*

```
namespace App\Http\Controllers;
use App\Article;
use Illuminate\Http\Request;
use App\Catalog;
use App\MaterialPicture;

class IndexController extends Controller
{
    public function getChapt() {
        $getChapter = Catalog::where('parent_id', 0)->get();
        return view('index', compact('getChapter'));
    }

    public function getCat(){
        return view('article_list');
    }

    public function getArt($prog=null, $cat=null, $id=null){
        $pictures = MaterialPicture::where('url',$cat)
                        ->orderBy('id','DESC')->get();
        $catalog = Catalog::where('url', $cat)->first();
        $previous = Article::where('id', '<', $id)
                       ->where('cat_id', $catalog→id)→
                       ->max('id');
        $next = Article::where('id', '>', $id)→
                       ->where('cat_id', $catalog→id)→
                       ->min('id');
```

```
    return view('article_show', compact('previous','next','pictures'));
    }
}
```

Model

Модель - это класс, через который программист взаимодействует с сущностями базы данных. По-умолчанию *Laravel* хранит все модели в папке *app/* [31-В].

При создании моделей мы можем обращаться либо к существующим таблицам текущей базы данных, либо одной командой созавать файлы и модели и миграции, которая содержит структуру таблицы. Для одновременного создания файлов моделей и миграций, можно воспользоваться командой:

```
php artisan make:model Category -m
```

Модель Article связана с моделью Category по ключу category_id.

Сама связь таблиц реализована в файлах моделей.

Рассмотрим класс модели Category:

```
class Category extends Model
{
    protected $fillable = [
          'name',
          'parent_id',
          'type',
    ];

    public function articles()
    {
        return $this->hasMany('App\Article','cat_id');
    }
    public function user(){
        return $this->belongsTo('App\User');
    }
}
```

Модель *Category* содержит две связи:

articles() - связь с моделей *Article* по полю *cat_id*. Каждая запись модели Category содержит множество записей модели Article.

users() - связь с моделью *User* по полю *user_id* (по-умолчанию). Каждая запись модели *Category* принадлежит какой-то записи модели *User*.

Расмотрим класс модели *Article*.

```
class Article extends Model
{
    protected $fillable = [
        'name',
        'description',
        'user_id',
```

```
        'cat_id'
    ];
    public function catalogs()
    {
        return $this->belongsTo('App\Category', 'cat_id');
    }
    public function users()
    {
        return $this->belongsTo('App\User', 'user_id');
    }
}
```

Каждая запись модели *Article* принадлежит записи моделям *Category* и *User*, для реализации связи были созданы методы *catalogs* и *users*.

На данном этапе у нас уже прописана маршрутизация проекта, созданы модели, через которые мы можем делать запросы в базу данных и сами контроллеры.

View

По умолчанию, *Laravel* работает с шаблонизатором *blade*. Шаблоны создаются в папке *app/views* и имеют расширение *blade.php*. Шаблоны подключаются в экшне через хелпер *view*(), входящим параметром в который передается имя шаблона без расширения blade.php [31-Г].

Blade — простой, но мощный шаблонизатор, поставляемый с *Laravel*. В отличие от других популярных шаблонизаторов для *PHP Blade* не ограничивает вас в использовании чистого *PHP*-кода в ваших шаблонах. На самом деле все шаблоны *Blade* скомпилированы в чистый *PHP*-код и кешированы, пока в них нет изменений, а значит, *Blade* практически не нагружает ваше приложение. Файлы шаблонов *Blade* используют расширение .blade.php и обычно хранятся в директории resources/views.

Два основных преимущества использования Blade — наследование шаблонов и секции. Для начала давайте рассмотрим простой пример. Во-первых, рассмотрим макет "главной" страницы. Поскольку многие веб-приложения используют один общий макет для разных страниц, удобно определить этот макет как один шаблон Blade:

```
// Хранится в resources/views/layouts/app.blade.php
```

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

Файл имеет типичную *HTML*-разметку, со специальными директивами @section и @yield. Директива @section, как следует из её названия, определяет секцию содержимого, а директива @yield используется для отображения содержимого заданной секции.

Определив макет для нашего приложения, давайте определим дочернюю страницу, которая унаследует макет.

При определении дочернего шаблона используйте Blade-директиву @extends для указания макета, который должен быть "унаследован" дочерним шаблоном. Шаблоны, которые наследуют макет Blade, могут внедрять содержимое в секции макета с помощью директив @section. Запомните, как видно из приведённого выше примера, содержимое этих секций будет отображено в макете при помощи @yield:

```
@extends('layouts.app')
@section('content')
    Это html-содержимое меняющейся части страницы.
@endsection
```

Директива @section('content') будет заменена содержимым макета при отрисовке шаблона.

Специальной *artisan*-команды для создания элементов представления не существует. Поэтому *blade*-файлы создаются в ручную в папке *resources/views/*

Middleware

HTTP Middleware (посредники) - это фильтры обработки *HTTP*-запроса. Так, например, в *Laravel* включены *middlewares* для проверки аутентификации пользователя. Если пользователь не залогинен, middleware перенаправляет его на страницу логина. Если же залогинен - *middleware* не вмешивается в прохождение запроса, передавая его дальше по цепочке middleware-посредников к собственно приложению [31-Д].

Middlewares можно использовать в качестве фильтров для роутов.

Конечно, проверка авторизации - не единственная задача, которую способны выполнять *middlewares*. Это также добавление особых заголовков (например, *CORShttp*-ответ вашего приложения) или логирование всех http-запросов.

В *Laravel* есть несколько дефолтных *middleware*, которые находятся в папке *app/Http/Middleware*. Это *middlewares* для реализации режима обслуживания сайта ("сайт

временно не работает, зайдите позже"), проверки авторизации, *CSRF*-защиты и т.п.

Для создания middleware можно воспользоваться artisan-командой make:middleware:

```
php artisan make:middleware Admin
```

В папке app/Http/Middleware будет создан файл с классом Admin и методом handle(), к которому добавим следующую логику:

```
if(Auth::user()->isAdmin != 1){
  return redirect()->guest('auth/login');
}
```

Чтобы пропустить запрос дальше, нужно вызвать функцию-замыкание *$next* с параметром *$request*.

Лучше всего представлять *middlewares* как набор уровней, которые *HTTP*-запрос должен пройти, прежде чем дойдёт до вашего приложения. На каждом уровне запрос может быть проверен по различным критериям и, если нужно, полностью отклонён.

Далее необходимо добавить *middleware* в свойство *routeMiddleware* класса *app/Http/Kernel.php*, назначив ему некоторое имя, например, *admin*, которое будет ключем массива:

```
protected $routeMiddleware = [
 'auth' => 'App\Http\Middleware\Authenticate',
 'admin' => 'App\Http\Middleware\Admin',
 'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',
];
```

Теперь в приложении имеется возможность скрывать контроллеры и маршруты как от не авторизованных пользователей, так и от не админов, путем подключения данного middleware.

```
public function __construct(){
 parent::__construct();
 $this->middleware('admin');
}
```

ServiceProviders

Основная задача поставщика услуг, или *ServiceProviders* – это загрузка вспомогательных, в том числе, и собственных классов [32].

Если открыть конфигурационный файла app.php, то мы увидим там множество провайдеров, определенных в массиве providers. Это все классы, которые будут загружены для приложения. Многие из них являются отложенными, что означает, что они не будут загружаться при каждом запросе, а только тогда, когда они действительно необходимы, т.е. когда они будут вызываться.

Все провайдеры должны наследоваться от класса *Illuminate\Support\ServiceProvider*. Данный абстрактный класс требует, чтобы был определен хотя бы один метод класса-провайдера: *register*. В этом методе регистрируется вспомогательный класс.

Сам провайдер можно создать с помощью artisan:

```
php artisan make:provider PhpqueryServiceProvider
```

В папке providers появился файл PhpqueryServiceProvider.

Провайдеры регистрируются в файле config/app.php.

```
'providers' => [
   // другиесервис-провайдеры
   'App\Providers\AppServiceProvider',
],
```

Теперь вся логика поставщика услуг будет доступна всему приложению. Удобство использования поставщиков услуг заключается в том, что мы можем передавать переменные в элементы представлений минуя контроллеры.

Request

Классы Request предназначены для валидации данных форм, создаются с помощью слдеюущей artisan-команды.

```
php artisan make:request CategoryRequest
```

Команда создаст файл с именем CategoryRequest и с соответствующим классом, который содержит два метода authorize() и rules() в папке requests/.

```
public function authorize()
{
  return true; // имеет ли доступ текущий пользователь
}


public function rules()
{
  return []; // возвращаем массив правил по обработке
}
```

Правила валидации, которые используются в методе rules(), прописаны в файле resources/lang/validation.php. Все элементы массива, которые есть в этом файле, можно использовать в качестве правил в request-файлах.

```
public function rules(){
     return [
            'name'=>'required|max:100',
            'body' => 'required|min:2',
            'theme_id'=>'required|numeric'
     ];
}
```

В модели, к которой мы хотим привязать эти request-данные, должно быть определено свойство $fillable, в котором мы указываем, какие имена полей таблицы будут задействованы при множественной вставки данных.

```
class Category extends Model
{
    protected $fillable = [
        'name',
        'body',
        'user_id',
        'theme_id',
    ];
}
```

Контроллер, который будет перехватывать данные из request-запроса, и через модель, с помощью множественной вставки (метод create()) помещать их в таблицу news:

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Auth;
use App\Http\Requests;
use App\Http\Controllers\Controller;
class CabinetController extends Controller
{
    public function __consturct(){
        parent::__construct();
    }
    public function getIndex(){
        return view('cabinet');
    }
    public function postIndex(Requests\CategoryRequest $r){
        $r['user_id'] = Auth::user()->id;
        \App\Category::create($r->all());
         return redirect()->back();
    }
}
```

При множественной вставке имена полей в таблице должны совпадать с именами элементов форм.

В шаблоне мы можем перехватывать ошибки валидации (волшебство *laravel* заключается в том, что специально эти ошибки в шаблон передавать не надо). Просто проверяем существует ли массив *$errors*, и если он существует, значит форма не прошла валидацию, и с помощью *foreach* проходимся по всем ошибкам, выводя их на экран.

```
@if (count($errors) > 0)
Whoops!Найдены следующие ошибки.
    @foreach ($errors->all() as $error)
     {{ $error }}
   @endforeach
```

```
@endif
```

Массив *$errors* является ассоциативным, где ассоциацией (индексом) массива является имя элемента формы, в которой выявлена ошибка. В тоже время Laravel позволяет работать с ним, как с объектом.

Проверим каждый элемент с помощью специальных методов:

```
 @if($errors->has('name'))
      {{ $errors->first('name') }}
 @endif
```

Для перевода компонентов сайта на русский язык, в конфигурационном файле app.php, в настройках locale укажем ru.

```
'locale' => 'ru',
```

Если локаль выставлена в значение «ru», то файлы переводов будут искаться в папке ru. В папке resources/lang создадим еще одну папку – ru. В ее можно вставить скопированный файл en/validation.php. И перевести на русский язык необходимые ошибки. При этомом слова, начинающиеся с : (двоеточия) не переводим, т.к. это атрибуты.

Авторизация

Модуль авторизации поставляется совместно с фрэймворком *Laravel*. Файл конфигурации авторизации находится в файле *config/auth.php* [33].

По умолчанию, для сохранения пользовательских данных, *Laravel* использует модель *App\User*.

Также модуль поставляется с двумя контроллерами аутентификации из коробки, которые находятся в *App\HTTP\Controllers\Auth*. *AuthController* содержит методы регистрации нового пользователя и аутентификации, в то время как *PasswordController* содержит логику помощи существующим пользователям сбросить свои забытые пароли. Каждый из этих контроллеров использует трэйты (*traits*), чтобы включить их необходимые методы. Для многих приложений вам не нужно будет изменить эти контроллеры вообще.

Для создания шаблонов и ротов авторизации, необходимо выполнить *artisan*-команду *make:auth.*

```
php artisan make:auth
```

Авторизация готова. Теперь в приложении доступны следующие url:

```
http://localhost//register
http://localhost/login
http://localhost/home
http://localhost/logout
```

Маршруты */register* и */login* определены для неавторизованных пользователей, а */home* и */logout* соответственно для авторизованных.

Для разделения логики авторизованных пользователей от неавторизованных в

шаблонизаторе blade используются диррективы @auth или @guest:

```
@guest()
 // реализация html кода для неавторизованных пользователей
@else
// реализация html кода для авторизированных  пользователей
@endguest
```

В результате проделанных действий была разработана серверная часть web-приложения.

3 .3 Программирование клиентской части web-приложения

Для клиентской части веб-сайта использовались языки разметки *HTML* и *CSS*, а также же скриптовый язык *JavaScript* – обязательные технологии для разработки frontend части любого веб-приложения. Также использовались *bootstrap, jQuery* и *jQuery-dm-uploader* [34].

JavaScript подключается к *HTML*-странице и выполняет код на стороне клиента (браузера). Возможны два варианта подключения *JavaScript* к странице *HTML*:

```
<script>//JavaScript код</script>
<script src="/my/script.js"></script>
```

JavaScript-среды поддерживают две версии языка: строгий режим и не строгий режим.

Включим строгий режим путем добавления в начале кода строковой константы:

```
“use strict”
```

Разработка на клиентской части велась с использованием строгого режима.

Основная часть клиентского кода велась с использованием библиотеки jQuery.

Библиотека *jQuery* помогает легко получать доступ к любому элементу *DOM*, обращаться к атрибутам и содержимому элементов *DOM*, манипулировать ими. Также библиотека *jQuery* предоставляет удобный *API* для работы с *AJAX*. Сейчас разработка *jQuery* ведётся командой *jQuery* во главе с Джоном Резигом.

JQuery-dm-uploader – Query Ajax File Uploader Widget. Виджет устанавливаем с открытого репозитория по ссылке *https://github.com/danielm/uploader*.

Подключение файлов стилей и скриптов осуществляется в файле *layouts/app.blade.php*. Пути к файлам прописываем с помощью функции *asset()*. Тэги помещаем в диррективу *@section*, закрывая ее диррективой *@show*. Что позволит в последствии подключиться к этой переменной в файле подшаблона и дописать переменную со стилями или скриптами дополнительными элементами.

Рассмотрим реализацию множественной загрузки изображений с помощью перетаскивания. В файле *demo-config.js* - прослушка событий перетаскивания и вызов серверного скрипта *backend/upload/*, в котором происходит загрузка файлов на сервер.

```
$(function(){
  $('#drag-and-drop-zone').dmUploader({
```

```
    url: '/backend/upload',
    maxFileSize: 3000000, // 3 Megs
    onDragEnter: function(){
      // Happens when dragging something over the DnD area
      this.addClass('active');
    },
    onDragLeave: function(){
      // Happens when dragging something OUT of the DnD area
      this.removeClass('active');
    },
    onInit: function(){
      // Plugin is ready to use
      ui_add_log('Penguin initialized :)', 'info');
    },
    onComplete: function(){
      // All files in the queue are processed (success or error)
      ui_add_log('All pending tranfers finished');
    },
    onNewFile: function(id, file){
      ui_add_log('New file added #' + id);
      ui_multi_add_file(id, file);
    },
    onBeforeUpload: function(id){
      ui_add_log('Starting the upload of #' + id);
      ui_multi_update_file_status(id, 'uploading', 'Uploading...');
      ui_multi_update_file_progress(id, 0, '', true);
    },
    onUploadCanceled: function(id) {
      // Happens when a file is directly canceled by the user.
      ui_multi_update_file_status(id, 'warning', 'Canceled by User');
      ui_multi_update_file_progress(id, 0, 'warning', false);
    },
    onUploadProgress: function(id, percent){
      // Updating file progress
      ui_multi_update_file_progress(id, percent);
    },
    onUploadSuccess: function(id, data){
       console.log(data);
      var obj = JSON.parse(data);
    });
});
```

Благодаря библиотеке jQuery, модулю JQuery-dm-uploader и вспомогательным классам была реализована множественная загрузка файлов на сервер.

3 .4 Размещение web-приложения в сети интернет

Возможны следующие варианты размещения *web*-приложения: хостинг, выделенный сервер, виртуальный сервер и собственный сервер.

Стандартный хостинг, предоставляемый большинством хостинговых компаний, подходит для

большинства web-приложений. В услуги хостинга входят *Apache + PHP + MySQL (PHPMyAdmin)*, а также системы управления сайтами и самим хостингом [35].

Выделенный сервер – это тот же компьютер, доступный по *ssh, http, https, ftp* и другим протоколам.

Виртуальный сервер – почти тоже, что и выделенный, с одной, но очень существенной разницей: на одном компьютере может быть установлено множество виртуальных серверов. Ресурсы компьютера распределяются между виртуальными серверами, а это значит они работают медленее, чем выделенные сервера.

Собственный сервер**:** любой домашний компьютер, у которого есть статический *ip*-адрес сети может выступать в роли сервера. [36]

Для размещения *web*-приложения в сети интернет воспользуемся встроенным сервером.

Благодаря операционной системе *Ubuntu* каждый может сделать из своего компьютера полноценный *web*-сервер. Для этого комьютер должен соответствовать следующим требованиям:

- объем жесткого диска должен быть достаточным для хранения, как сайта, базы данных, вспомогательных файлов, так и необходимого инструментария;

- статический *IP*, его можно получить у провайдера (только по технологии *adsl*) у нас в Беларуси, это 3$ в месяц.

- высокоскоростной интернет;

Все файлы приложения находятся в специальной дирректории var/www/html. Исполняемые файлы сервера *apache2* находится по адресу /etc/apache2/sites-available/. Для запуска приложения в этой дирректории был создан файл default.conf

```
<VirtualHost *:80>
        ServerAdmin admin@example.com
        ServerName obmenka
        ServerAlias obmenka
        DocumentRoot /var/www/obmenka.by
        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Для запуска виртуального хоста, обслуживающего web-ресурс, выполним следующую команду:

```
sudo a2ensite example.com.conf
```

Для применения изменений в настройках необходимо перезапустим демон Apache:

```
sudo service apache2 restart
```

Для привязки статического ip-адреса к домену http://erud.by, был заключен договор с компанией ООО «Надежные программы hoster.by», которая оказала услугу по регистрации домена на период с 19.06.2018 года.

В последствии предполагается продление данной услуги.

3 .5 Выводы по главе 3

Разработанный *web*-сайт является совокупностью определенных информационных блоков и инструментов, предназначенных для взаимодействия с одним или несколькими сегментами целевой аудитории потребителей.

В целом структура представленной информации, набор задействованных инструментов и их взаимодействие зависит от выбранной владельцем сайта бизнес-модели, поставленных долгосрочных и краткосрочных задач, типа и величины сегмента целевой аудитории, а также от возможности взаимодействовать с ней тем или иным способом.

Разработанный ресурс должен быть доступен в результатах поиска поисковых систем *Google* и *Yandex*. Для этого были реализованы следующие задачи:

1. Выбора места размещения сервера, выбора провайдера, разработки дизайна сервера и его структуры, первичное информационное наполнение сервера. Следующим шагом этапа является решение вопросов сообщения сервера с существующей бизнес-системой фирмы, его предварительное тестирование и размещение в Internet.

2. Определение коммерческих целей и путей их достижения, для чего проводятся маркетинговые исследования и разрабатывается детальный план необходимых мероприятий.

3. Проведение комплекса мероприятий для привлечения клиентов на сервер с применением различных видов рекламы.

4. Подведение итогов проводимого путем сравнения полученных результатов с ранее прогнозируемыми показателями.

Существует такое понятие, как требование поисковых систем к сайту. При разработке http://erud.by были выполнены все необходимые требования. В результате, в соответствии с анализатором Google PageSpeed Insights, разработанный сайт имеет очень высокий рэйтинг у Google — 96%. Этот показатель намного выше показателей таких раскрученных ресурсов, как tut.by, onliner.by, bsuir.by и многих других белорусских ресурсов.

Google PageSpeed Insights предложил следующие рекомендации по ускорению загрузки контента:

– устранить ресурсы, блокирующие отображение;

– включить сжатие текста;

– отложить загрузку неиспользуемого контента CSS.

Устранение данных замечаний приведет к закреплению высоких позиций ресурса.

Заключение

Список использованных источников

1. [Печатное издание] **Слива М.В. Кроссплатформенный подход как средство унификации обучения программированию в различных операционных системах** М.В.Слива – Нижневартовский университет: 2012 – 45 с.
2. [Электронный ресурс] **Интеграционные среды разработок Java** режим доступа: http://habr.com/post/140745/. Дата доступа 19.05.2019.

3. [Электронный ресурс] **PHP** Режим доступа: https://ru.wikipedia.org/wiki/PHP. Дата доступа 19.01.2019.
4. [Печатное издание] **ГОСТ 19.105-78 Общие требования к программным документам** Режим доступа: http://rugost.com/index.php?option=com_content&view=article&id=52:19105. Дата доступа 19.01.2019.

Приложения

1. [slide] 5aed65380bacf_Linux3.png
2. [picture] 5aed670b4f14b_linux0.png
3. [picture] 5aed670b5de9d_Linux1.png
4. [slide] 5aed670b72a78_Linux3.png
5. [picture] 5aedd26b19a61_wx1080.jpg
6. [picture] 5af1770835b50_vydel.jpg
7. [picture] 5b1d3f788760d_comands
8. [picture] 5b1d5ab352988_Linux1.png
9. [picture] 5b1d5abd3ee07_Снимок экрана от 2018-04-08 02-28-53.png
10. [picture] 5b1d5acd283bd_comands
11. [more] 5b1d5b0eaf3ec_Снимок экрана от 2018-04-08 02-28-53.png
12. [slide] 5b1d5b9d1bb4b_Снимок экрана от 2018-04-08 02-28-53.png
13. [] 5b1d5ba75fa5d_linux0.png
14. [auto] 5b1f0db8f0c7d_Автореферат.docx
15. [title] 5b1f0e3a37c57_Титульник.docx
16. [review] 5b1f117c8f498_Отзыв.docx
17. [picture] 5b9bc6c9b8c57_72. Гран Торино - Gran Torino (2008) BDRip 1080p 7.83 ГБ.torrent
18. [title] 5c365d7f48f17_title.docx
19. [auto] 5c3673e3cbd5f_Autoreferat.docx
20. [auto] 5c3675dc2d516_Autoreferat.docx
21. [auto] 5c368df6b8e2d_Autoreferat.docx
22. [auto] 5c369e8a867ca_Autoreferat.docx
23. [auto] 5c378d91a08ea_Autoreferat.docx
24. [act] 5c37981b43db0_act.docx
25. [picture] 5c3c87fd3b199_Autoreferat.docx
26. [picture] 5c3c8f6239a33_Autoreferat.docx
27. [title] 5c3c99f3359e2_title.docx
28. [act] 5c3e44aea9abf_act.docx
29. [title] 5c3e46d506295_title.docx
30. [auto] 5c3e499b2fdf4_Autoreferat.docx
31. [auto] **Автореферат** 5c3e4a14871fe_Autoreferat.docx
32. [protocol] **Протокол** 5c57c58f8f675_protocol.docx
33. [review] **Отзыв** 5c57ceb374b9c_otzyv.docx