МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра веб-технологий и компьютерного моделирования

ПРУДНИКОВ
Артём Сергеевич

РАЗРАБОТКА КОРПОРАТИВНОГО МЕДИА-ПОРТАЛА

Дипломная работа

Научный руководитель:
Кандидат физико-математических
наук,
доцент кафедры веб-технологий и
компьютерного моделирования
СУЗДАЛЬ Станислав Валерьевич

Допущена к защите
«___» ________ 2014 г.
Зав. кафедрой ВТ и КМ,
канд. физ.-мат. наук,
доцент Романчик В.С.

Минск, 2014

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ ... 6
ГЛАВА 1 СРАВНЕНИЕ С АНАЛОГАМИ И ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ ... 7
1.1 Сравнение с аналогами. Обзор существующих разработок. ... 7
1.2 Краткий обзор существующих технологий разработки приложения. ... 10
1.2.1. REST-сервис ... 12
1.2.2 Шаблон MVC. ASP.NET MVC Pipeline ... 13
1.2.3 Model binding в ASP.NET MVC Framework ... 18
ГЛАВА 2 ИНФОРМАЦИОННОЕ МОДЕЛИРОВАНИЕ. ПРЕДМЕТНАЯ ОБЛАСТЬ 22
2.1 Проектирование приложения. Модель ... 22
2.2 Детализация проекта. ... 26
ГЛАВА 3 ПОСТАНОВКА ЗАДАЧИ И ЦЕЛИ ПРОЕКТА ... 27
3.1 Постановка задачи. ... 27
3.2 Цель проекта ... 27
ГЛАВА 4 ФУНКЦИОНАЛЬНОСТЬ И АРХИТЕКТУРА ... 28
4.1 Описание требуемого функционала ... 28
4.2 Информационная модель и разработка приложения ... 29
ГЛАВА 5 ПЛАНЫ И РАЗВИТИЕ ... 36
5.1 Планы по развитию проекта ... 36
ЗАКЛЮЧЕНИЕ ... 37
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ ... 38
ПРИЛОЖЕНИЕ А ... 39
Примеры кода Unit-тестов ... 39
ПРИЛОЖЕНИЕ Б ... 42
Пример кода контроллера ... 42

РЕФЕРАТ

Полный объем работы – 43 страницы. Содержит 18 рисунков, 2 приложения и 7 использованных библиографических источников. Ключевые слова: ВЕБ-САЙТ, REST-СЕРВИСЫ, ASP.NET MVC, МОДУЛЬНОЕ ТЕСТИРОВАНИЕ, РАЗРАБОТКА, ПОДДЕРЖКА.

Целью работы является разработка корпоративного портала, использующего имеющийся REST-сервис в качестве источника данных.

Объектом исследования является разработка стабильной, постоянно доступной и легко адаптируемой системы.

В процессе исследования выполнен обзор и анализ имеющихся средств для разработки крупных корпоративных Веб-приложений, способы поддержки и развития таких приложений. Рассмотрен подход разработки через тестирование, позволяющий писать приложения, менее подверженные ошибкам и потому требующие сокращённой фазы тестирования.

Теоретическая значимость полученных результатов заключается в получении способа и подхода к быстрой разработке крупных корпоративных медиа-порталов, использующих общий слой данных и управляющиеся из единой точки.

Практической и экономической значимостью работы является возможность использования ее результатов для создания подобных систем с меньшими временными и финансовыми затратами. Т.к. задача создания таких систем довольно распространённая, то это позволит сэкономить большое количество различных ресурсов.

РЭФЕРАТ

Поўны абъём работы – 43 старонкі. Змяшчае 18 малюнкаў, 2 прыклдання і 7 выкарыстанных крыніц. Ключавыя словы: ВЭБ-САЙТ, REST-СЕРВІСЫ, ASP.NET MVC, МОДУЛЬНАЕ ТЭСТАВАННЕ, РАСПРАЦОЎКА, ПАДТРЫМКА.

Мэтай работы з’яўляецца распрацоўка карпаратыўнага партала, які выкарыстоўвае гатовы REST-сервіс у якасці крыніцы дадзеных.

Аб’ектам даследавання з’яўляецца распрацоўка стабільнай, заўжды даступнай і лёгка адаптуемай сістэмы.

Ў працэссе даследавання быў зроблены абзор і аналіз наяўных сродкаў для распрацоўкі буйных карпаратыўных Вэб-прыкладанняў, спосабы падтрымкі і развіцця такіх прыкладанняў. Разгледзены падыход распрацоўкі праз тэставанне, які дазваляе пісаць прыкладанні, якія менш схільны памылкам і таму патрабуюць скарочанай фазы тэставання.

Тэарэтычная значнасць атрыманых вынікаў заключаецца ў атрымліванні спосаба і падыхода да хуткай распрацоўкі буйных карпаратыўных медыа-парталаў, якія карыстаюць агульны слой дадзеных і кіруюцца праз адзіную сістэму.

Практычнай і эканамічнай значнасцю работы з’яўляецца магчымасць карыстання яе вынікаў для стварэння падобных сістэм з меншымі часовымі і фінансавымі выдаткамі. Паколькі задача стварэння такіх сістэм вельмі распаўсюджана, гэта дазволяе эканоміць выдатную кольцасць розных рэсурсаў.

ABSTRACT

Whole project contains 43 pages, 18 images, 2 appendixes and 7 sources. Keywords: WEB-SITE, REST-SERVICES, ASP.NET MVC, UNIT-TESTING, DEVELOPMENT, SUPPORT.

The purpose of the graduation project is the development of a stable, all-time available and easily adaptable system.

During study have been performed overview and analysis of existing tools for development of a huge enterprise Web-applications, ways of supporting and expanding of such applications. Overview of Test Driven Development (TDD) have been also performed. TDD approach allows to develop applications that are less fallible and require shorter testing phase.

The theoretical meaning of obtained results is in development of methods of rapid development of huge enterprise media-portals, which use shared data layer and managed from single system.

The practical and economical meaning of graduation project are in ability of using its results for creating similar systems with shorter development time and less cost. Since this is pretty common issue this project allows save a huge amount of different resources.

ВВЕДЕНИЕ

В настоящее время одним из главных источников получения медиа-информации является глобальная сеть Интернет. И эта роль Интернет с каждым годом только увеличивается, заменяя классические телевидение, радио и печатные издания.

Таким образом, компании, которые стремятся повысить эффективность своего маркетинга, увеличить продажи, или качество оказываемых услуг просто обязаны использовать средства Интернет. В частности, такая задача как распространение медиа-информации среди потенциальных потребителей, должна происходить в том числе посредством Интернет. Качественные корпоративные порталы предоставляют возможность доставки конечным потребителям этой информации. На этих порталах пользователь может ознакомиться с последними новостями компании, узнать о ее продукции и разработках, и т.д.

Разработка качественного медиа-портала - распространённая и важная задача. Ещё более важно сделать удобным управление такими порталами (в том числе, например, из одного приложения), поскольку экономит трудозатраты и снижает требования к сотрудникам, занимающимися актуализацией информации на портале.

ГЛАВА 1 СРАВНЕНИЕ С АНАЛОГАМИ И ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

1.1 Сравнение с аналогами. Обзор существующих разработок.

Brand.puma.com/news

Хорошим примером качественного корпоративного портала является новостная часть сайта brand.puma.com (Рисунок 1.1). Сайт имеет приятный и удобный дизайн и на нём есть всё необходимое для подобного рода сервисов.

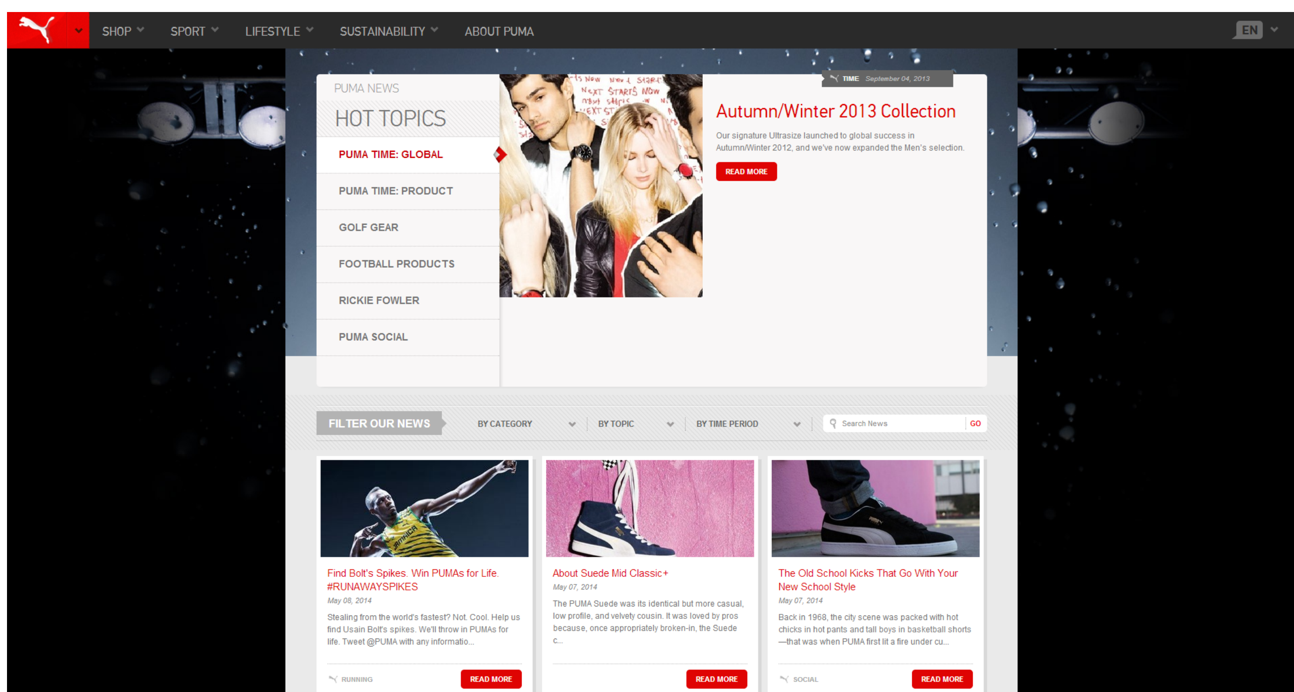

Рисунок 1.1

На сайте присутствуют различные модули, выполняющие свои информационные обязанности. Рассмотрим некоторые из этих модулей поподробнее.

На сайте присутствует модуль “HOT TOPICS” (Рисунок 1.2). Этот модуль отображает наиболее интересные на данный момент людям темы и позволяет ознакомиться с каждой из них подробнее.

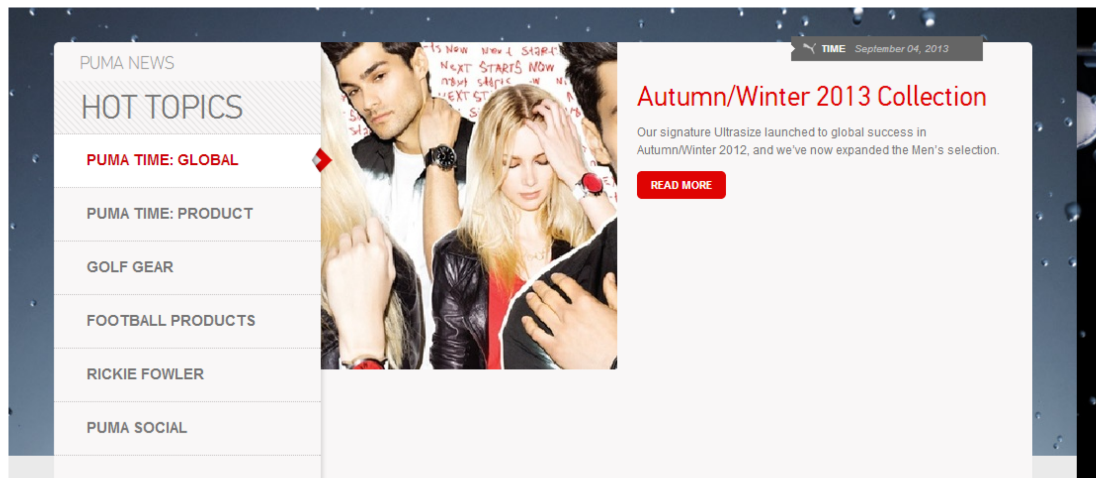

Рисунок 1.2

Основным, интересующим нас модулем, является модуль отображения новостей (Рисунок 1.3). В данном модуле отображаются новости в кратком виде с возможностью чтения полной версии. Так же присутствует кнопка загрузки следующей страницы (Рисунок 1.4), фильтрация новостей (Рисунок 1.5, Рисунок 1.6), сортировка и поиск по новостям.

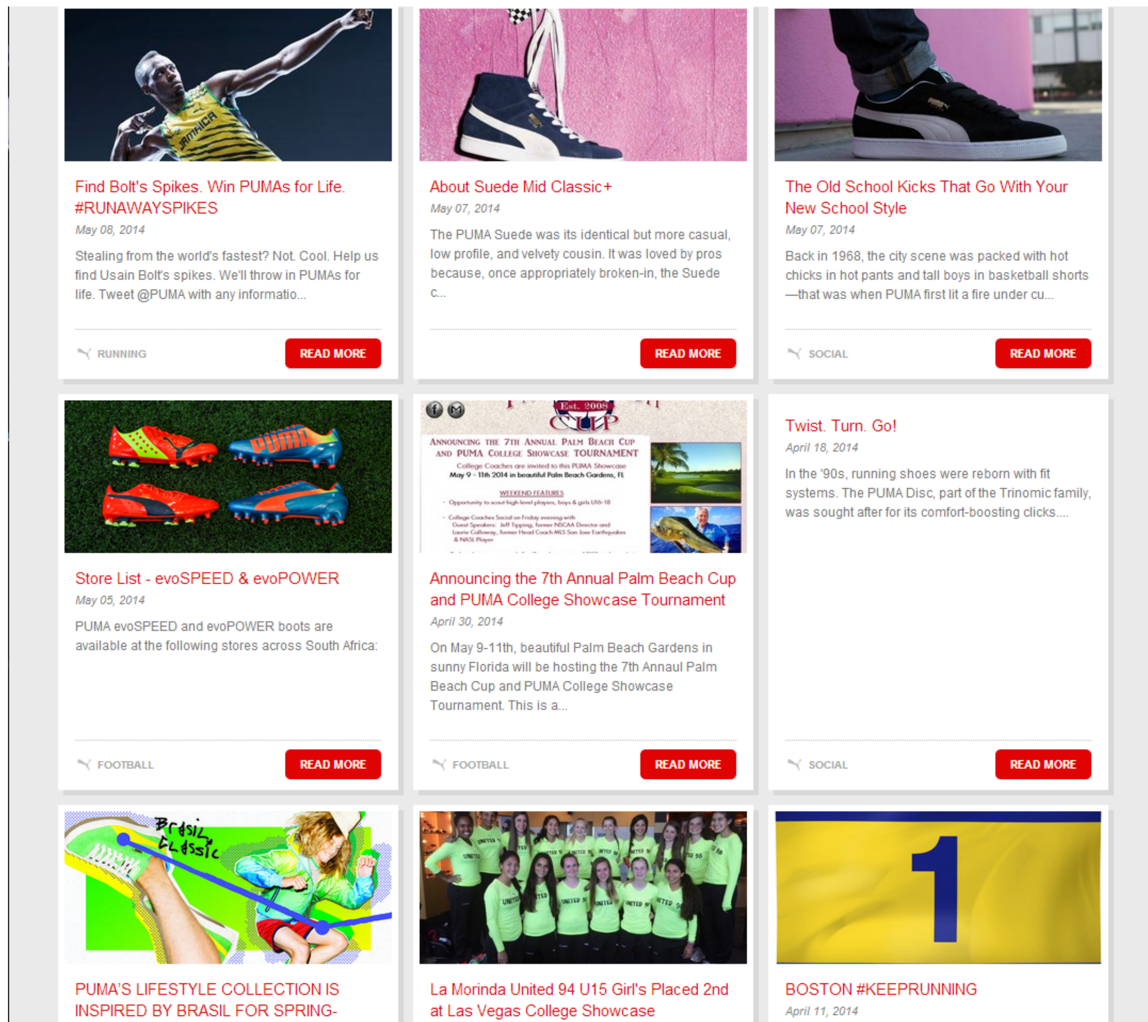

Рисунок 1.3

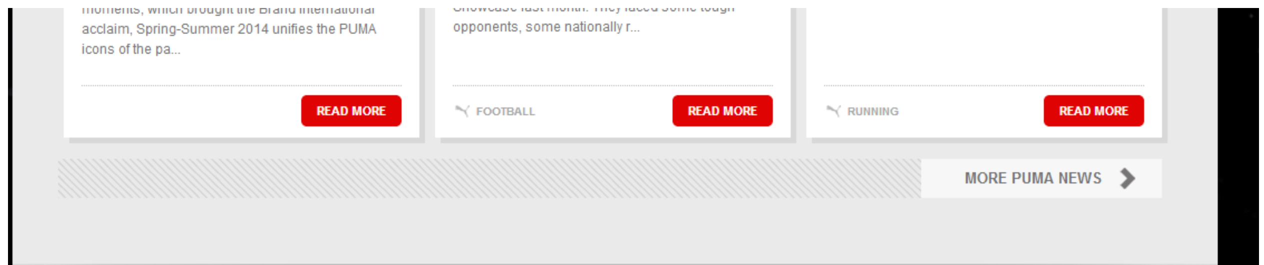

Рисунок 1.4

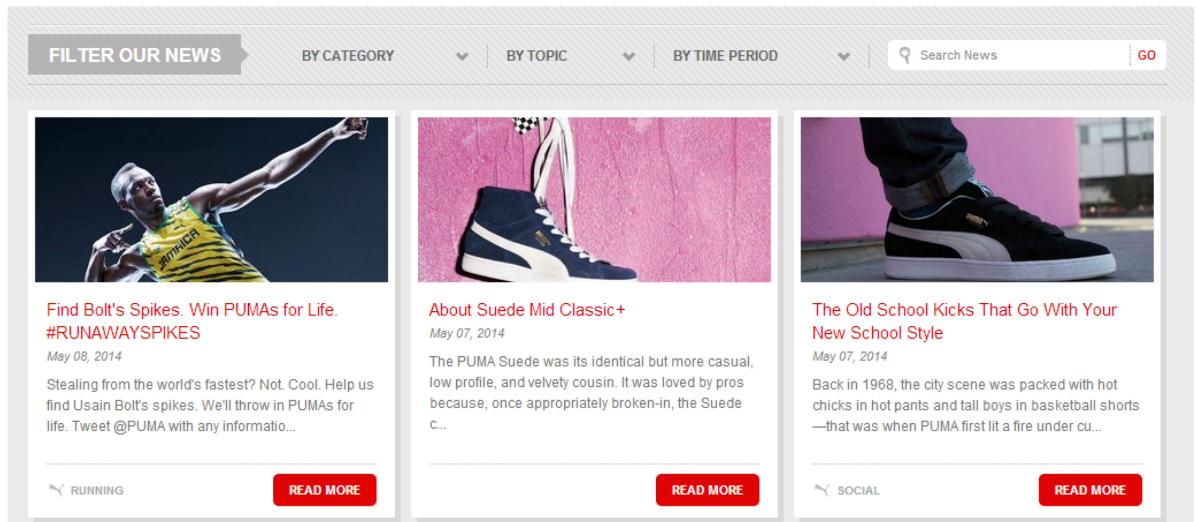

Рисунок 1.5

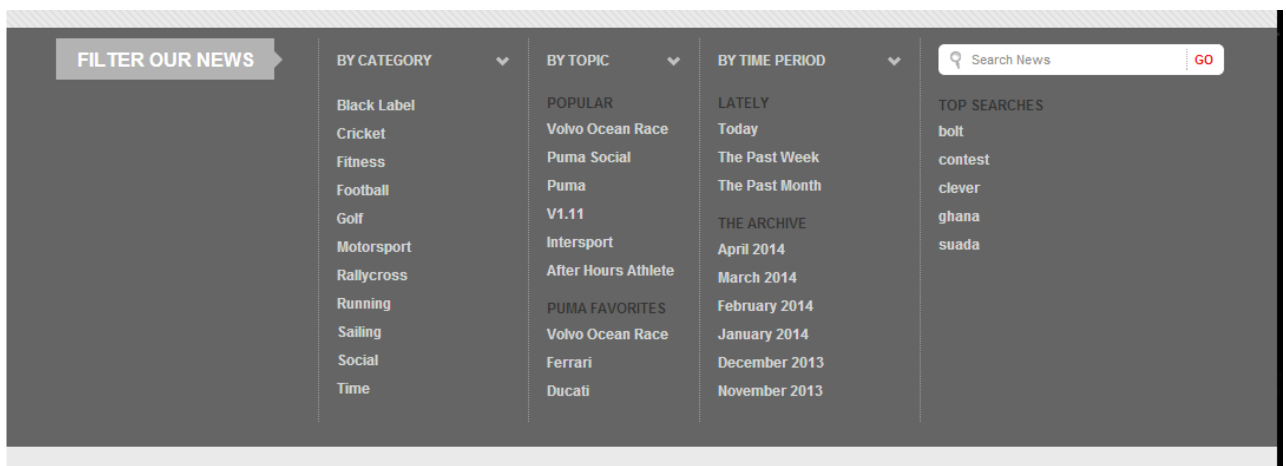

Рисунок 1.6

Так же есть возможность выбрать категорию, язык и перейти на страницы в социальных сетях (Рисунок 1.7).

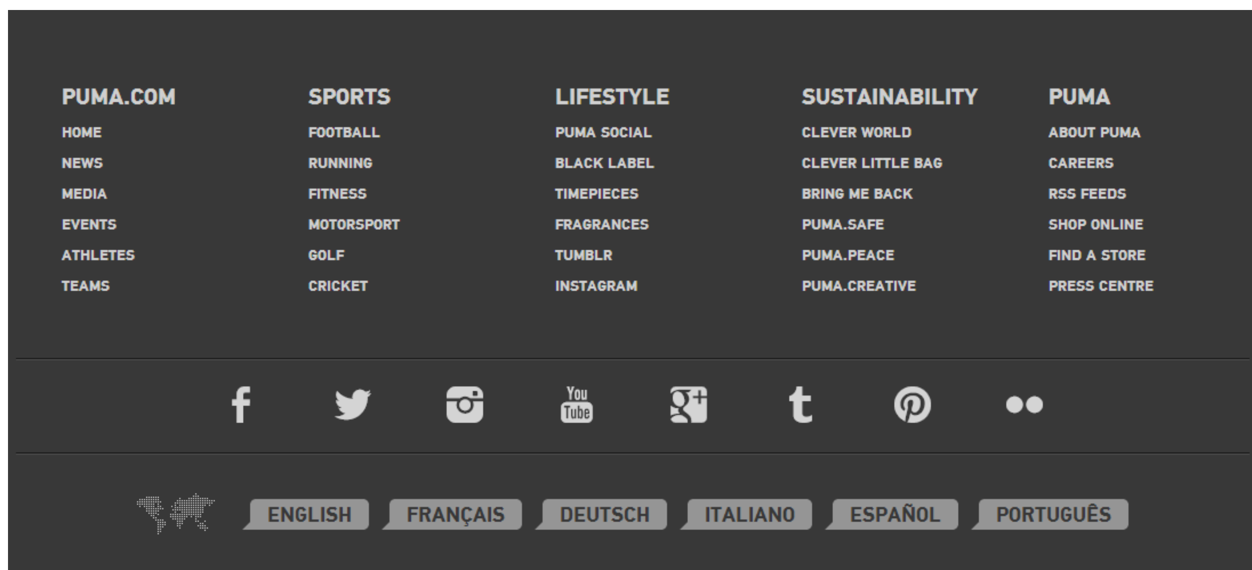

Рисунок 1.7

1.2 Краткий обзор существующих технологий разработки приложения.

Система разрабатывается на платформе ASP.NET MVC 4 с использованием языка программирования C#. ASP.NET MVC представляет собой фреймворк для создания Веб-приложений на платформе .NET, который реализует паттерн MVC.

ASP.NET MVC универсальная технология, которая позволяет использовать себя как для небольших проектов, так и для крупных высоконагруженных систем (например Stackoverflow). В отличие от другой технологии разработки Веб-приложений на .NET – ASP.NET WebForms – данный фреймворк не ставит своей целью перенести подход разработки оконных Windows-приложений на разработку Web-приложений. ASP.NET MVC практикует исключительно органичный подход к Веб-разработке, базирующийся на принципах работы в Веба и паттерне MVC [1].

Исходный код ASP.NET MVC находится в открытом доступе, хотя и нет возможности изменять его таким образом, чтобы изменения попали в новую версию. Разрабатывается полностью компанией Microsoft.

Для данного приложения в качестве Веб-сервера используется Microsoft IIS 7. А так же .NET Framework 4.

IIS и .NET Framework входят в поставку Windows Server 2008 и не требует дополнительных действий для установку и большей части конфигурирования.

Приложение не имеет собственного хранилища данных наподобие реляционных либо NoSQL баз данных. Вместо этого используется уже готовый слой данных, реализованный в виде удалённого Веб-сервиса. Этот слой данных позволяет управлять данным сразу нескольких подобных систем

Обмен данными с сервисами происходит по протоколу HTTP. Практически все данные запроса содержаться в URL. Все данные приходят от сервисов в формате XML.

ASP.NET MVC, реализуя шаблон MVC [2], облегчает управление сложными структурами путем разделения приложения на модель, представление и контроллер. Даёт разработчикам полный контроль над поведением приложения. Обеспечивает расширенную поддержку разработки на основе тестирования. Хорошо подходит для веб-приложений, поддерживаемых крупными коллективами разработчиков, а также веб-разработчикам, которым необходим высокий уровень контроля над поведением приложения.

Платформа ASP.NET MVC предоставляет следующие возможности:

- Разделение задач приложения (логика ввода, бизнес-логика и логика пользовательского интерфейса), широкое возможности тестирования и разработки на основе тестирования. Все основные контракты платформы MVC основаны на интерфейсе и подлежат тестированию с помощью макетов объекта, которые имитируют поведение реальных объектов приложения. Приложение можно подвергать модульному тестированию без запуска контроллеров в процессе ASP.NET, что ускоряет тестирование и делает его более гибким. Для тестирования возможно использование любой платформы модульного тестирования, совместимой с .NET Framework.
- Расширяемая и дополняемая платформа. Компоненты платформы ASP.NET MVC можно легко заменить или настроить. Разработчик может подключать собственный механизм представлений, политику маршрутизации URL-адресов, сериализацию параметров методов действий и другие компоненты. Платформа ASP.NET MVC также поддерживает использование моделей контейнера внедрения зависимости (DI) и инверсии элемента управления (IOC). Модель внедрения зависимости позволяет внедрять объекты в класс, а не ожидать создания объекта самим классом. Модель инверсии элемента управления указывает на то, что если один объект требует другой объект, то первые объекты должны получить второй объект из внешнего источника (например, из файла конфигурации). Это облегчает тестирование.
- Расширенная поддержка маршрутизации ASP.NET. Этот мощный компонент сопоставления URL-адресов позволяет создавать приложения с понятными URL-адресами, которые можно использовать в поиске. URL-адреса не должны содержать расширения имен файлов и предназначены для поддержки шаблонов именования URL-адресов, обеспечивающих адресацию,

оптимизированную для поисковых систем (SEO) и для передачи репрезентативного состояния (REST).

- Поддержка использования разметки в существующих файлах страниц ASP.NET (ASPX), элементов управления (ASCX) и главных страниц (MASTER) как шаблонов представлений. Вместе с платформой ASP.NET MVC можно использовать существующие функции ASP.NET, например вложенные главные страницы, встроенные выражения (<%= %>), декларативные серверные элементы управления, шаблоны, привязку данных, локализацию и т. д.
- Поддержка существующих функций ASP.NET. ASP.NET MVC позволяет использовать такие функции, как проверка подлинности с помощью форм и Windows, проверка подлинности по URL-адресу, членство и роли, кэширование вывода и данных, управление состоянием сеанса и профиля, наблюдение за работоспособностью, система конфигурации и архитектура поставщика.

ASP.NET MVC обеспечивает решения многих задач, с которыми сталкиваются Веб-разработчики и организации, которые хотят создать информационную систему, основанную на платформе .NET. ASP.NET MVC позволяет использовать различные библиотеки, которые дополняют функциональность платформы либо реализуют некоторые функции лучше. Большинство этих фреймворков и библиотек может работать независимо друг от друга, однако, они обеспечивают большую функциональность при совместном их использовании. Решено использовать следующие его элементы:

- ***Inversion of Control контейнер Unity*****:** конфигурирование компонент приложений и управление жизненным циклом объектов.
- ***Фреймворк для Unit-тестирования Nunit:*** конфигурируемый фреймворк, облегчающий создание модульных тестов.
- ***Фреймворк для создания Mock-***объектов: фреймворк для увеличения гибкости тестирования.

1.2.1. REST-сервис

Важной частью всей системы является хранилище, которое в нашем случае представляет собой REST-сервис.

REST – это набор архитектурных принципов и стиль проектирования приложений, ориентированный на создание сетевых систем, в основе которых лежат механизмы для описания и обращения к ресурсам. Примером такой системы может служить World Wide Web.

В REST определяется строгое разделение ответственности между компонентами клиент-серверной системы, облегчающее реализацию необходимых актеров (actors). Другой целью REST является упрощение семантики взаимодействия компонентов сетевых систем, что позволяет улучшить масштабируемость и повысить производительность. В основу REST заложен принцип автономности запросов, означающий, что запросы, обрабатываемые клиентом или сервером, должны включать всю контекстную информацию, необходимую для их понимания. При работе REST-систем для обмена данными стандартных медиа-типов используется минимальное количество запросов.

REST-системы используют URI (универсальные идентификаторы ресурсов) для поиска и получения доступа к представлениям необходимых ресурсов.

В течение последних нескольких лет разработчики создавали REST-сервисы для своих .NET-приложений, используя самые разнообразные технологии. Архитектура REST отличается своей простотой, требуя от приложений обеспечить только возможность приема сообщений с HTTP-заголовками. Эта функция легко реализуется простыми контроллерами в ASP.NET MVC.

1.2.2 Шаблон MVC. ASP.NET MVC Pipeline

MVC — это не шаблон проекта, это конструкционный шаблон, который описывает способ построения структуры нашего приложения, сферы ответственности и взаимодействие каждой из частей в данной структуре [2].

Впервые она была описана в 1979 году, конечно же, для другого окружения. Тогда не существовало концепции веб приложения. Tim Berners Lee (Тим Бернерс Ли) посеял семена World Wide Web (WWW) в начале девяностых и навсегда изменил мир. Шаблон, который мы используем сегодня, является адаптацией оригинального шаблона к веб разработке.

Бешеная популярность данной структуры в веб приложениях сложилась благодаря её включению в две среды разработки, которые стали очень популярными: Struts и Ruby on Rails. Эти две среды разработки наметили пути развития для сотен рабочих сред, созданных позже.

Идея, которая лежит в основе конструкционного шаблона MVC, очень проста: нужно чётко разделять ответственность за различное функционирование в наших приложениях (Рисунок 1.8):

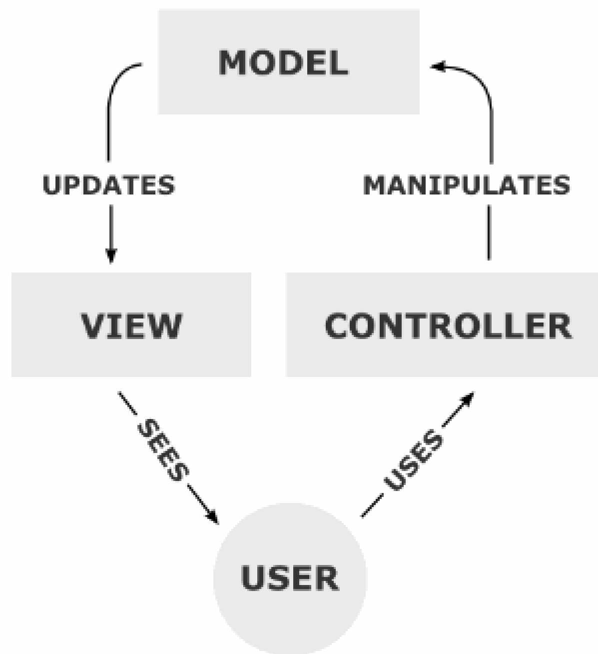

Рисунок 1.8

Приложение разделяется на три основных компонента, каждый из которых отвечает за различные задачи (принцип единой ответственности).

Контроллер управляет запросами пользователя (получаемые в виде запросов HTTP GET или POST, когда пользователь нажимает на элементы интерфейса для выполнения различных действий). Его основная функция — вызывать и координировать действие необходимых ресурсов и объектов, нужных для выполнения действий, задаваемых пользователем. Обычно контроллер вызывает соответствующую модель для задачи и выбирает подходящий вид.

Модель - это данные и правила, которые используются для работы с данными, которые представляют концепцию управления приложением. В любом приложении вся структура моделируется как данные, которые обрабатываются определённым образом. Что такое пользователь для приложения — сообщение или книга? Только данные, которые должны быть обработаны в соответствии с правилами (дата не может указывать в будущее, e-mail должен быть в определённом формате, имя не может быть длиннее X символов, и так далее).

Модель даёт контроллеру представление данных, которые запросил пользователь (сообщение, страницу книги, фотоальбом, и тому подобное). Модель данных будет одинаковой, вне зависимости от того, как мы хотим представлять их пользователю. Поэтому мы выбираем любой доступный вид для отображения данных.

Модель содержит наиболее важную часть логики нашего приложения, логики, которая решает задачу, с которой мы имеем дело (форум, магазин, банк, и тому подобное). Контроллер содержит в основном организационную логику для самого приложения (очень похоже на ведение домашнего хозяйства).

Стоит отметить, что в данном случае описан подход с «толстой» моделью и «тонким» контроллером. Очень часто практикуется подход наоборот – «тонкая» модель и «толстый» контроллер – когда бизнес-логика заключена в контроллере, а модель является лишь данными.

Вид обеспечивает различные способы представления данных, которые получены из модели. Он может быть шаблоном, который заполняется данными. Может быть несколько различных видов, и контроллер выбирает, какой подходит наилучшим образом для текущей ситуации.

Веб приложение обычно состоит из набора контроллеров, моделей и видов. Контроллер может быть устроен как основной, который получает все запросы и вызывает другие контроллеры для выполнения действий в зависимости от ситуации.

Самое очевидное преимущество, которое мы получаем от использования концепции MVC — это чёткое разделение логики представления (интерфейса пользователя) и логики приложения.

Поддержка различных типов пользователей, которые используют различные типы устройств является общей проблемой наших дней. Предоставляемый интерфейс должен различаться, если запрос приходит с персонального компьютера или с мобильного телефона. Модель возвращает одинаковые данные, единственное различие заключается в том, что контроллер выбирает различные виды для вывода данных.

Помимо изолирования представления от логики приложения, концепция MVC существенно уменьшает сложность больших приложений. Код получается гораздо более структурированным, и, тем самым, облегчается поддержка, тестирование и повторное использование решений.

ASP.NET MVC Pipeline – это процесс обработки запроса приложением и фреймворком.

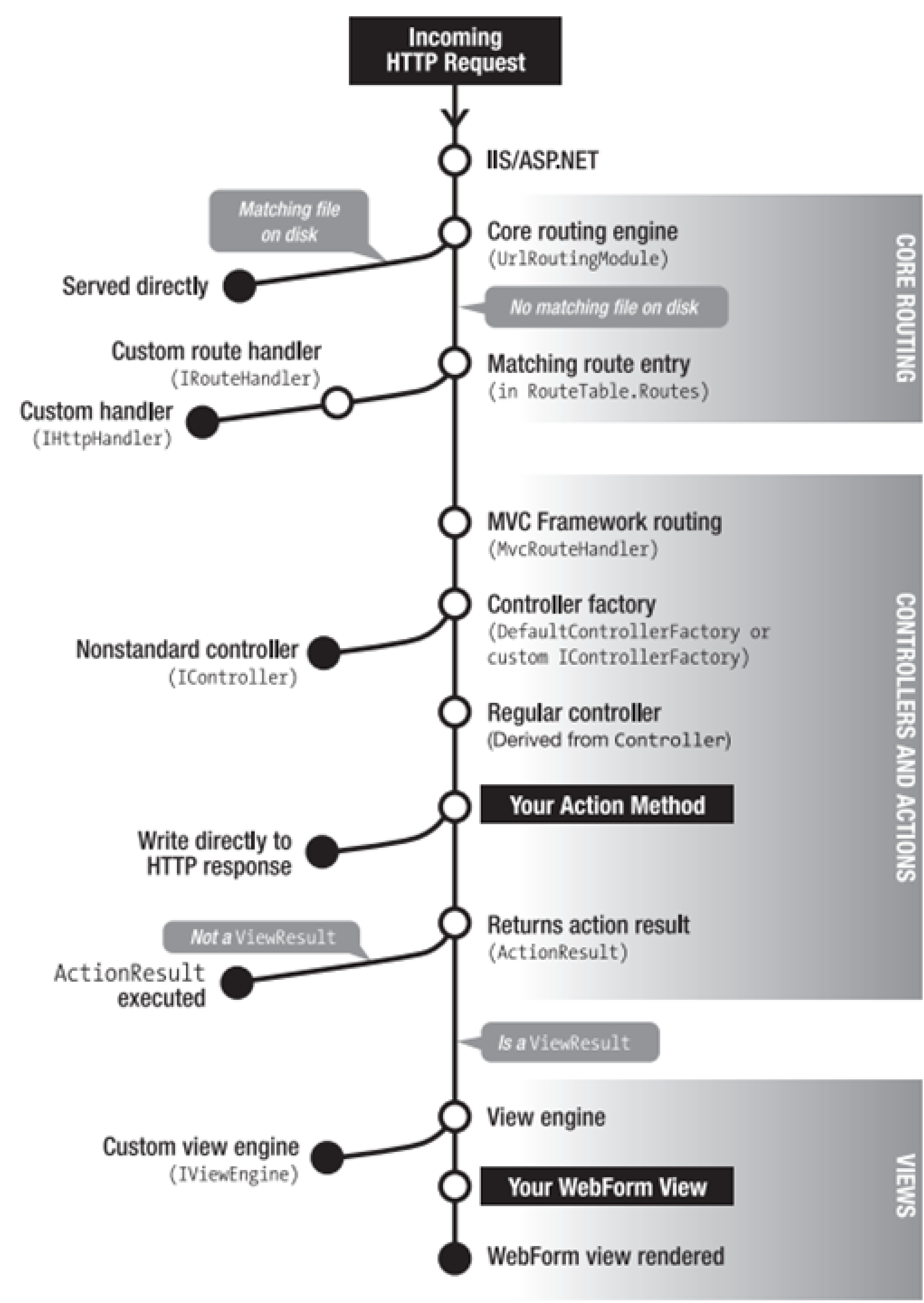

Рисунок 1.9

В кратком виде обработка запроса выглядит следующим образом (Рисунок 1.9):

1) На сервер приходит запрос от пользователя
2) IIS запускает ASP.NET модуль, который будет запускать приложение
3) Система маршрутизации определяет каким контроллером должен обрабатываться запрос
4) Выполняется нужный Action

5) Рендерится View и отдаётся пользователю

Таким образом происходит обработка запроса в ASP.NET MVC [1].

В фреймворке ASP.NET MVC, как следует из названия, реализован паттерн MVC. Основой в данной реализации являются контроллеры. Каждый контроллер представлен классом, который должен быть унаследован от класса Controller в общем случае. Данный класс имеет модули, именуемые Action'ами, которые и являются механизмом реакции на действия пользователя. Выбор выполняемого метода, как правило, осуществляется с помощью системы маршрутизации. Так же есть и другие механизмы, такие как, например, HttpHandler'ы. Вот пример обычного контроллера, унаследованного от класса Controller:

```
 public class HomeController : Controller
 {
    private readonly IUseCaseFactory _useCaseFactory;
    private readonly IViewModelMapper _mapper;

   public HomeController(IUseCaseFactory useCaseFactory, IViewModelMapper ma
pper)
    {
       _useCaseFactory = useCaseFactory;
       _mapper = mapper;
    }
    public ViewResult IpRedirection()
    {
       return View();
    }
    public ViewResult Index()
    {
       return View();
    }
    public PartialViewResult Rotator()
    {
       var useCase = _useCaseFactory.Create<GetRotatorUseCase>();
       var rotator = useCase.Act();

       return PartialView(rotator);
    }
}
```

1.2.3 Model binding в ASP.NET MVC Framework

Когда механизм MVC Framework определил класс контроллера и метод действия в нем, которые должны участвовать в формировании ответа на клиентский запрос встает задача передать параметры запроса действию. В простейшем случае, передача данных осуществляет путем сопоставления параметров запроса с параметрами метода. Так, например, если пользователь отправил форму с данными через заполнение определенных html-элементов, то значения содержимого этих элементов будет передано действию через одноименный параметр метода. Данные textarea с атрибутом name равным address будут сопоставлены с параметром метода действия определенным как string address.

Таким сопоставлением занимается механизм Model Binding, но его возможности на таком простом примере не заканчиваются. Что делать, когда параметров на форме несколько десятков? Не станем же мы создавать методы с двумя десятками параметров, только для того, чтобы получить данные с формы. Действительно, делать этого не нужно. Model Binding предоставляет замечательную возможность переопределить порядок сопоставления параметров и сделать инициализацию не в виде “параметр формы = параметр метода”, а сложнее. Например, мы можем определить класс или структуру, которые будут содержать данные формы и в методе определить только один параметр в виде экземпляра этой структуры. Затем определив механизм Model Binding для нашего комплексного типа мы можем инициализировать его свойства параметрами запроса и вызвать метод действия, передав ему инициализированный объект. В итоге, многочисленные параметры на форме передадутся в метод действия в виде экземпляра класса с заполненными свойствами. Очень удобно.

Но разработчику MVC Framework даже этого делать не нужно, поскольку в MVC Framework реализован класс DefaultModelBinder, который делает всю работу по сопоставлению параметров запроса комплексным типам. Однако, это не мешает вам определить свой собственный механизм Model Binding для определенных типов параметров. Это может быть полезно, когда кроме простой передачи данных из запроса, вы захотите произвести еще какие-нибудь действия с экземпляром комплексного типа.

У приложения MVC Framework есть стандартное свойство ModelBinders, которое содержит коллекцию определенных пользователем механизмов сопоставления параметров. Эти механизмы представляют собой реализацию интерфейса IModelBinder, который содержит только один метод BindModel со следующим определением:

```
public object BindModel(ControllerContext controllerContext,
ModelBindingContext bindingContext);
```

Используя передаваемый механизмом MVC Framework параметр controllerContext, разработчик может получит доступ ко всем данным контекста запроса, в том числе к параметрам. Механизм MVC Framework ожидает от реализации интерфейса IModelBinder объекта, который бы определял результат сопоставления параметров запроса для конкретного комплексного типа, к которому применяется атрибут. Регистрация своего механизма Model Binding производится в global.asax в методе Application_Start следующим образом:

```
ModelBinders.Binders.Add(typeof(UserData), new UserDataBinder());
```

где, typeof(UserData) – указание для какого комплексного типа создается механизм сопоставления, new UserDataBinder() – экземпляр класс реализующего IModelBinder.

Примечание. Для того, чтобы использовать эти механизмы не глобально, а только локально, одноразово, разработчик, вместо регистрации глобального механизма Model Binding, может пометить требуемый параметр действия атрибутом ModelBinder, который содержит параметр определяющий тип механизма. Как это делается, показано ниже:

```
public ActionResult Update([ModelBinder(typeof(UserDataBinder))]
UserData userData);
```

Когда MVC Framework вызывает метод действия, то для каждого параметра комплексного типа, он ищет в коллекции ModelBinders механизм Model Binding и вызывает его, если он есть. После этого, пользовательская реализация сопоставления параметров формирует объект который приводится к типу параметра и передается при вызове действия. В случае, когда пользовательской реализации для комплексного типа нет, в дело вступает механизм сопоставления параметров по умолчанию, который реализуется через DefaultModelBinder.

Разработчик может еще более гибко настроить выполнение сопоставления параметров используя атрибут BindAttribute, который позволяет определить несколько правил сопоставления:

1) префикс используемый в разметке для сопоставления по умолчанию;
2) белые и черные списки имен параметров

Пример реализации пользовательского ModelBinder'а при наследовании от DefaultModelBinder:

```
public class HomeCustomDataBinder : DefaultModelBinder
{


public override object BindModel(ControllerContext controllerContext, ModelBindin
gContext bindingContext)
{
   if (bindingContext.ModelType == typeof(HomePageModels))
   {
      HttpRequestBase request = controllerContext.HttpContext.Request;

      string title = request.Form.Get("Title");
      string day = request.Form.Get("Day");
      string month = request.Form.Get("Month");
      string year = request.Form.Get("Year");

      return new HomePageModels
      {
         Title = title,
         Date = day + "/" + month + "/" + year
      };

      }
      else
      {
         return base.BindModel(controllerContext, bindingContext);
      }
   }
}
```

Пример реализации пользовательского ModelBinder'а при реализации интерфейса IModelBinder:

```
public class HomeCustomBinder : IModelBinder
{
   public object BindModel(ControllerContext controllerContext,
                     ModelBindingContext bindingContext)
   {
      HttpRequestBase request = controllerContext.HttpContext.Request;
```

```
        string title = request.Form.Get("Title");
        string day = request.Form.Get("Day");
        string month = request.Form.Get("Month");
        string year = request.Form.Get("Year");

        return new HomePageModels
        {
            Title = title,
            Date = day + "/" + month + "/" + year
        };
    }
}
```

ГЛАВА 2 ИНФОРМАЦИОННОЕ МОДЕЛИРОВАНИЕ. ПРЕДМЕТНАЯ ОБЛАСТЬ

2.1 Проектирование приложения. Модель

В основе разрабатываемой системы лежит архитектура «клиент-сервер», в которой задания или сетевая нагрузка распределены между поставщиками услуг (сервисов), называемых серверами, и заказчиками услуг, называемых клиентами. В качестве среды взаимодействия клиента с сервером используется интернет (Рисунок 2.1).

Рисунок 2.1

Основными достоинствами архитектуры «клиент-сервер» являются:

- Возможность, в большинстве случаев, распределить функции вычислительной системы между несколькими независимыми компьютерами в сети. Это позволяет упростить обслуживание вычислительной системы. В частности, замена, ремонт, модернизация или перемещение сервера, не затрагивают клиентов.
- Все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов. На сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа.

- Позволяет объединить различные клиенты. Использовать ресурсы одного сервера часто могут клиенты с разными аппаратными платформами, операционными системами и т.п.

Основные недостатки:

- В случае использования централизованной системы, неработоспособность основного сервера может сделать неработоспособным всё приложение.
- Администрирование данной системы требует квалифицированного профессионала;
- Высокая стоимость оборудования.

В ходе выбора аппаратной платформы будут предложены и реализованы решения, позволяющие минимизировать вероятность выхода из строя серверной части приложения, а также позволяющие снизить стоимость оборудования до оплаты минимально необходимого уровня производительности.

Клиентская часть приложения должна поддерживать следующие технологии:

- Доступ к сети интернет.
- Возможность работы по протоколу HTTP.
- Поддержка устройств взаимодействия с человеком для ввода данных.

В основе самого приложения лежит шаблон MVC, который был рассмотрен выше, поэтому рассматривать здесь мы его не будем. Рассмотрим общую архитектуру системы (Рисунок 2.2), архитектуру проекта Visual Studio (Рисунок 2.3) и архитектуру классов (Рисунок 2.4, Рисунок 2.5).

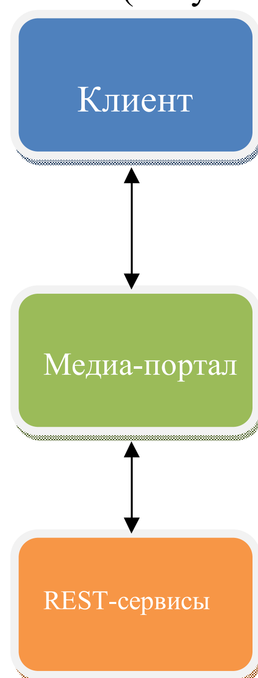

Рисунок 2.2

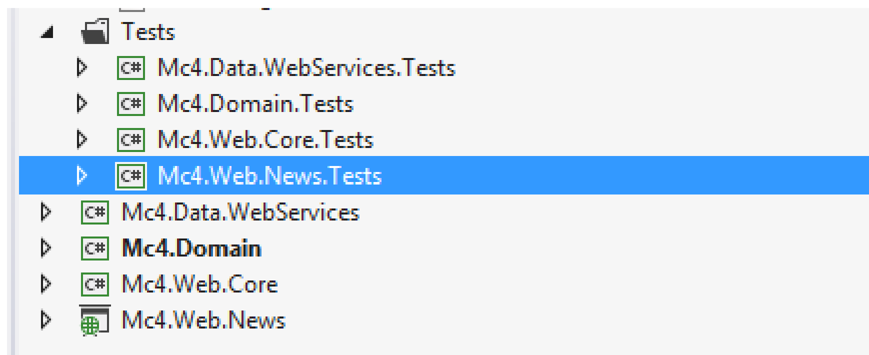

Рисунок 2.3

Mc4.Domain – смысловое ядро приложения. Содержит основные сущности и интерфейсы

Mc4.Data.WebServices – работа с удалённым Rest-сервисом. Выполняет функции слоя данных

Mc4.Web.Core – ядро веб-части

Mc4.Web.News – основная веб-часть. Отвечает за работу пользователя и отображение данных

Папка *Tests* – содержит Unit-тесты проекта. Структура проектов внутри папки повторяет структуру проектов в остальной части проекта.

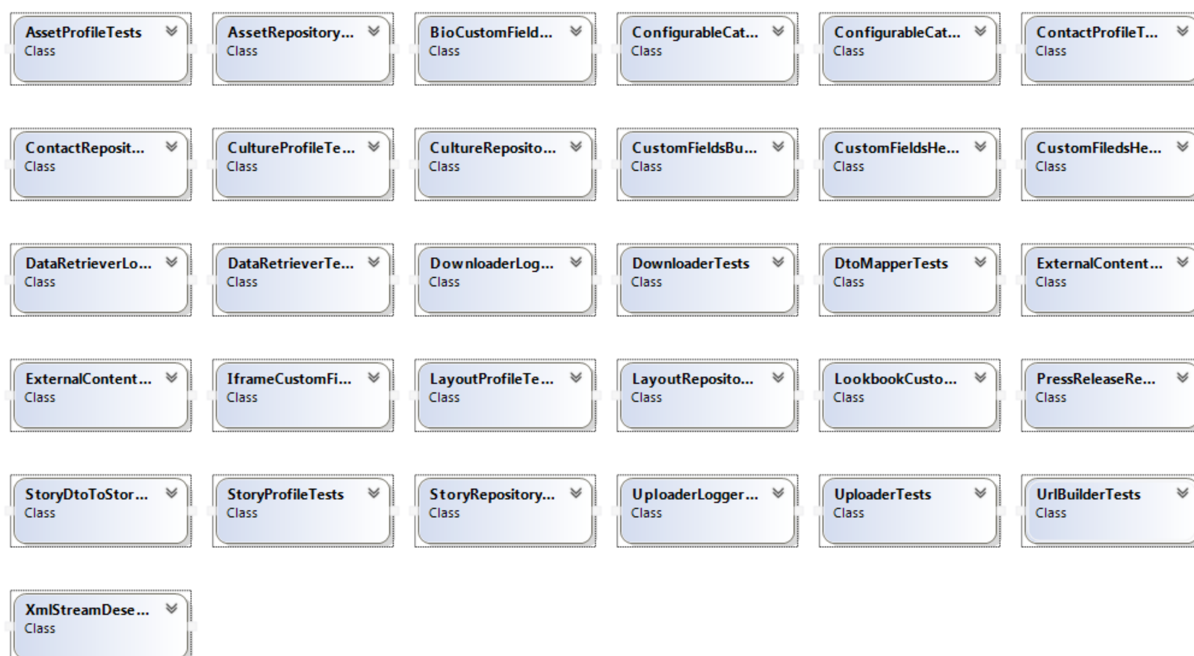

Рисунок 2.4

Рисунок 2.5

2.2 Детализация проекта.

В качестве пользователей системы выступают лица, интересующиеся новостями данной компании.

Все пользователи могут просматривать новости, делиться прочитанным в социальных сетях, добавлять интересные им темы в закладки.

Контент для сайта добавляется отдельной группой пользователей в специализированной администраторской панели, не являющейся частью рассматриваемой системы.

ГЛАВА 3 ПОСТАНОВКА ЗАДАЧИ И ЦЕЛИ ПРОЕКТА

3.1 Постановка задачи.

1. Проектирование архитектуры серверной и клиентской частей приложения.
2. Создание информационной модели
3. Разработка Web-сайта

Целью разработки является создание системы, включающей в себя серверную часть, обрабатывающую поступающие запросы пользователей системы; клиентскую часть, к которой относятся интерфейсы пользователей; и слой работы с данными, использующий удалённые веб-сервисы для доступа ко всей необходимой информации.

3.2 Цель проекта

Разработать корпоративный медиа портал, отвечающий современным стандартам производительности, безопасности и удобства. Все личные данные пользователей, если таковые имеются, должны быть недоступны никому кроме пользователя и администратора системы. Пароли и прочая чувствительная к взлому информация должны храниться в зашифрованном виде. Реакция системы на действия пользователя не регламентирована строго, но не должна быть слишком большой. Можно сказать, что, при использовании среднестатистического ADSL Интернет-соединения, ответ не должен превышать 2 сек.

ГЛАВА 4 ФУНКЦИОНАЛЬНОСТЬ И АРХИТЕКТУРА

4.1 Описание требуемого функционала

Основной функцией портала является отображение новостей. Новости должны быть разбиты на категории с возможностью просматривать выбранную категорию. На домашней странице должен присутствовать модуль отображения наиболее часто просматриваемых новостей. Было решено реализовать с помощью rotator'а или карусели (Рисунок 4.1).

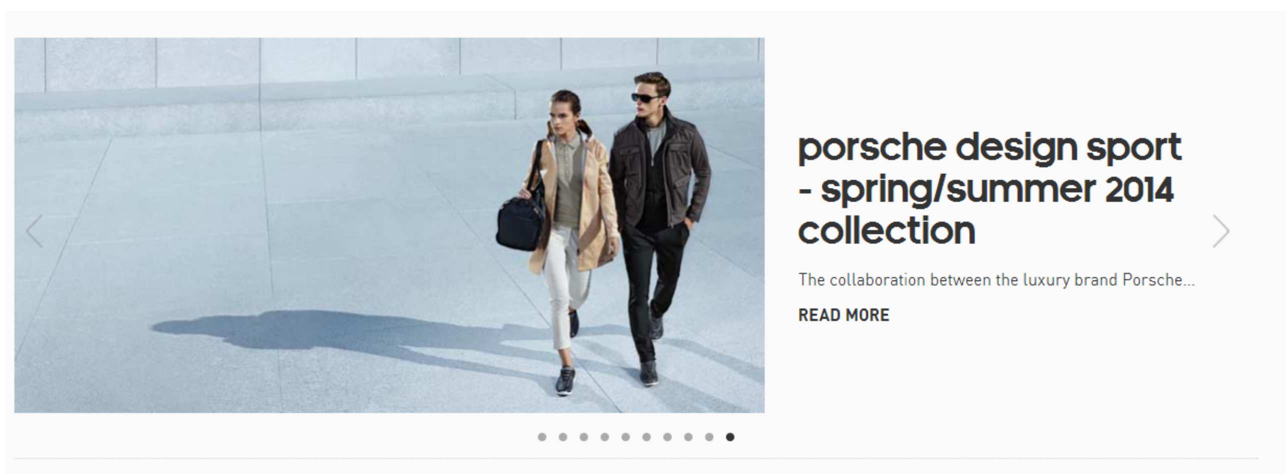

Рисунок 4.1

Данный rotator позволяет выбирать новость кликая по стрелочкам по бокам либо по кружкам снизу. Чёрным цветом кружка выделена текущая новость. Кликнув по READ MORE можно попасть на страницу новости, где тема или новость будет раскрыта в гораздо более полном объёме.

Также новости должны быть разбиты по различным секциям:

1) Latest news – последние добавленные новости
2) Images & videos – изображения и видео
3) Lookbooks – альбомы с изображениями, видео или аудио
4) Products – информация о продуктах
5) athletes & ambassadors – сотрудничающие атлеты или представители компании в случае спортивной индустрии

На сайте должен присутствовать поиск по новостям, поддержка многоязычности (Рисунок 4.2), ссылки на страницы в социальных сетях и форма подписки на рассылки портала.

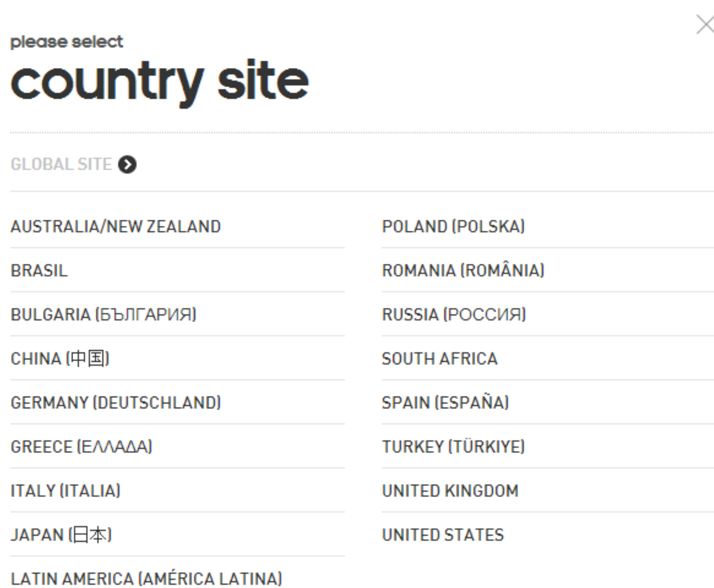

Рисунок 4.2

4.2 Информационная модель и разработка приложения

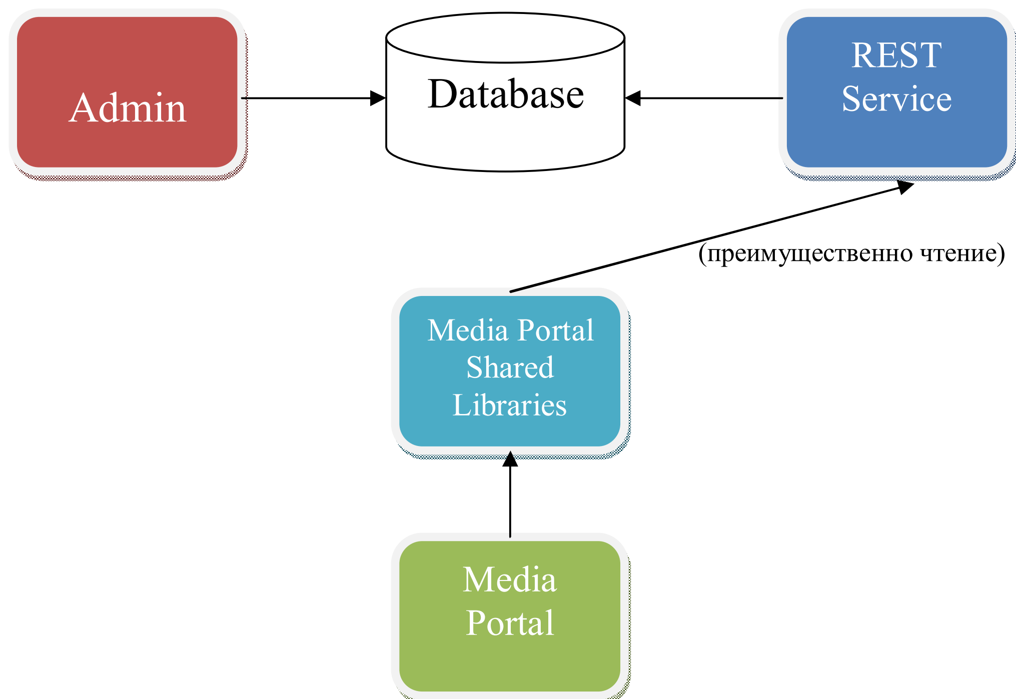

Рисунок 4.3

Рассмотрим глобальную архитектуру всей системы, которая представлена на Рисунке 4.3.

Admin – администраторская панель. Используется для управления содержимым многих медиа-порталов. В данной работе разработка и поддержка администраторской панели не рассматривались, и предполагалось, что уже имеется рабочий экземпляр такой панели.

Database – база данных. Хранит данные всех медиа-порталов, управляемых из административной панели. Поддержка и изменение схемы данной базы данных в данной работе так же не рассматривались.

REST Service – REST сервисы. Используются для получения различными медиа-порталами необходимой информации, в том числе и контента наподобие изображений, видео и т.д. Изменение и разработка методов для этих сервисов в работе не рассматривались.

Media Portal Shared Libraries – общие библиотеки для всех медиа-порталов данного типа и версии. Частью данный работы являлась разработка таких библиотек, которые можно было бы использовать в других медиа-порталах, помимо того, который разрабатывался мной. Данные библиотеки включают в себя общие утилитные классы, модули для работы с REST-сервисами, ядро приложения. Также данная часть проекта включает в себя все модульные тесты, покрывающие имеющиеся в ней классы. В качестве фреймворка для модульных тестов используется NUnit, в качестве mock-библиотеки – FakeItEasy. При разработке приложения и тестов использовался подход Test Driven Development (TDD) и Red/Green/Refactor. TDD – это техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам (см Рисунок 4.4). Использование TDD в данной работе было неким гарантом качества кода, т.к. программный код, который легко тестировать, является слабо связанным кодом и соответствует современным подходам и стандартам кодирования.

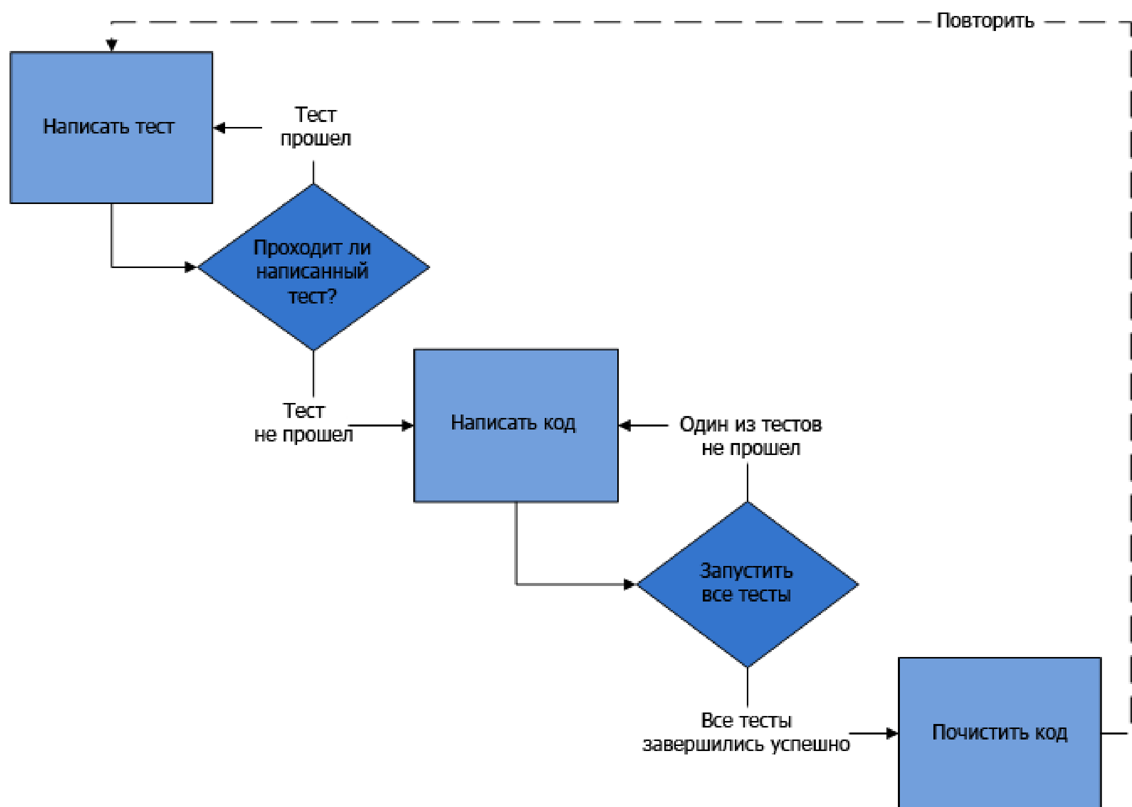

Рисунок 4.4

Для работы с REST-сервисами не использовались никакие библиотеки, за исключением стандартных из Base Class Library платформы .NET. это обусловлено тем, что эти классы имеют необходимую функциональность для работы приложения. Благодаря тому, что осуществлялось программирование на основе интерфейсов и слабая связность кода, замена этих классов на другие, в случае нехватки функциональности или скорости работы, не требует больших временных затрат.

Media Portal - непосредственно сам медиа-портал. Данная часть приложения включает в себя логику, специфическую для конкретного портала, возможно системы авторизации и аутентификации, разметку страниц, статическое содержимое, контроллеры и т.д.

В данной части задействованы классы из Shared libraries, через которые происходит взаимодействие с REST-сервисами. Логические части приложения (например, получение списка элементов содержания) заключены в функциональные акторы, именующиеся в системе как UseCase. Это применение функционального программирования в рамках парадигмы Объектно-Ориентированного Программирования.

Вот пример кода метода контроллера, в котором используется актор и отображается PartialView.

```
[ChildActionOnly]
public PartialViewResult Listing(AssetType assetType, int page, int pageSize)
{
   var useCase = _useCaseFactory.Create<GetAssetsUseCase>();
   var output = useCase.Act(assetType, page, pageSize);
   var viewModel = _mapper.Map<GetAssetsUseCase.Output, ListingViewModel>(o
utput);

   return PartialView(viewModel);
}
```

Благодаря такому подходу мы можем легко подменять актора, а методы контроллеров имеют похожую структуру.

Широкое распространение в проекте получил механизм ActionFilter'ов. Этот механизм позволяет видоизменять возможность выполнения методов контроллеров и строится на основе механизма атрибутов .NET. В коде выше мы видим применение фильтра ChildActionOnly. Так же есть и другие фильтры, например:

```
public class DetermineCultureAttribute : ActionFilterAttribute
   {
      public IUseCaseFactory UseCaseFactory { get; set; }

      public override void OnActionExecuting(ActionExecutingContext filterContext)
      {
         if (filterContext.IsChildAction) return;

         var cultureSeoName = (string)filterContext.RouteData.Values["cultureSeoNa
me"];

         var setCulture = UseCaseFactory.Create<SetCultureUseCase>();
         var output = setCulture.Act(cultureSeoName);

         if (output.Culture == null)
         {
            RedirectToCulture(filterContext, output.DefaultCulture);
         }
         else
         {
            SetThreadCultureInfos(output.Culture);
```

```
        }
    }

    private static void RedirectToCulture(ActionExecutingContext filterContext, Cu
lture culture)
    {
        var routeValues = new RouteValueDictionary(filterContext.RouteData.Values
);
        routeValues["cultureSeoName"] = culture.SeoName;

        filterContext.Result = new RedirectToRouteResult(routeValues);
    }

    private static void SetThreadCultureInfos(Culture culture)
    {
        Thread.CurrentThread.CurrentCulture = culture.CultureInfo;
        Thread.CurrentThread.CurrentUICulture = culture.CultureInfo;
    }
}
```

В качестве механизма отображения используется стандартный на данный момент в MVC Razor. Пример view с использованием Razor:

```
@using Mc4.Domain.Data
@using Mc4.Web.Resources
@model Mc4.Web.Core.Models.Assets.ListingViewModel

@if (Model.AllCount > 0)
{
  <div class="filters">
    <ul>
        @RenderTab(Model.DoShowAllTab, AssetType.ImageVideoAudio, Model.
AllCount, Phrase.AllUpper)
        @RenderTab(Model.DoShowVideoTab, AssetType.Video, Model.VideoCou
nt, Phrase.VideosUpper)
        @RenderTab(Model.DoShowImageTab, AssetType.Image, Model.ImageCou
nt, Phrase.ImagesUpper)
        @RenderTab(Model.DoShowAudioTab, AssetType.Audio, Model.AudioCou
nt, Phrase.AudioUpper)
    </ul>
```

```
    </div>

    <ul class="images-videos-items block-grid four-up tablet-two-up mobile-one-up">
        @foreach (var asset in Model.Items)
        {
            <li>@Html.Partial("_ImagesAndVideosItem", asset)</li>
        }
    </ul>

    @Html.Action("Paginator", "Snippet", new {page = Model.Page, pageSize = Mod
el.PageSize, totalCount = Model.TotalCount})
}
else
{
    @Html.Partial("_NoResults")
}

@helper RenderTab(bool doShowTab, AssetType assetType, int count, string caption
)
{
    if (doShowTab)
    {
        var active = Model.AssetType == assetType ? "active" : null;
        <li class="@active">
            <a href="@Url.Action("Index", new {t = assetType})">@caption<p>(@coun
t)</p></a>
            @if (Model.AssetType == assetType)
            {
                <i class="icon icons-active-arrow-filters"></i>
            }
        </li>
    }
}
```

Примеры кода тестов можно найти в Приложении А. Примеры кода контроллеров можно найти в Приложении Б.

Т.к. сайт написан на ASP.NET MVC, то он обладает всеми преимуществами по удобству deployment'а, в том числе прямо из Visual Studio. Важно перед размещением портала на хостинге сконфигурировать его.

Конфигурация происходит с помощью редактирования файла Web.config. Одним из наиболее важных параметров является параметр «WebServiceBaseUrl» который указывает на адрес REST-сервиса. Без указания правильного адреса портал работать не будет, т.к. неоткуда брать информацию для размещения.

ГЛАВА 5 ПЛАНЫ И РАЗВИТИЕ

5.1 Планы по развитию проекта

Основной идеей развития проекта является создание системы конструирования подобных медиа-порталов. В результате должна появиться панель администратора, где администратор мог бы указывать принадлежность портала компании, модули, расцветку и т.д.

ЗАКЛЮЧЕНИЕ

В ходе данной работы были изучены технологии ASP.NET MVC 4, работа с REST-сервисами, подход к разработке через тестирование, основы функционального программирования.

Можно констатировать тот факт, что основная цель данной работы была выполнена – был разработан корпоративный медиа-портал, обладающий богатым функционалом.

Благодаря получившемуся порталу, люди смогут получать информацию об интересующей их компании в кратчайшие сроки и в удобной для них форме.

Очень важным моментом является многоязычность портала, т.к. это в полной мере позволяет реализовать возможности Интернет, как глобально, а следовательно и многоязычной, среды.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Фримен, А. ASP.NET MVC 4 с примерами на C# 5.0 для профессионалов. 4-е изд. – Россия : Вильямс, 2013. – 688 с.

2. Фаулер, М. Шаблоны корпоративных приложений. – Россия: Вильямс, 2011. – 544 с.

3. Рихтер, Д. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. – СпБ. : Питер, 2013. – 896 с

4. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. /Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. – СПб: Питер, 2001. – 368 с.

5. Ошероув, Р. Искусство автономного тестирования с примерами на C#. – Россия : ДМК Пресс, 2013. – 360 с.

6. Симан, М. Внедрение зависимостей в .NET. – СпБ. : Питер, 2013. – 464 с.

7. Голдштейн, С. Оптимизация приложений на платформе .Net. /Голдштейн С., Зурбалев Д., Флатов И. – Россия : ДМК Пресс, 2014. – 524с.

ПРИЛОЖЕНИЕ А

Примеры кода Unit-тестов

```
[TestFixture]
public class SetCultureUseCaseTests
  {
    private readonly Culture _enUsCulture = new Culture
    {
      Locale = "en-US",
      Name = "Global",
      SeoName = "GLOBAL",
      CultureInfo = CultureInfo.GetCultureInfo("en-US"),
    };

    private readonly Culture _ruRuCulture = new Culture
    {
      Locale = "ru-RU",
      Name = "Russia (Россия)",
      SeoName = "RU",
      CultureInfo = CultureInfo.GetCultureInfo("ru-RU"),
    };

    private List<Culture> _cultures;

    private IDomainConfig _domainConfig;
    private ICultureRepository _cultureRepository;
    private IUserSession _userSession;

    private SetCultureUseCase _useCase;

    [SetUp]
    public void SetUp()
    {
      _cultures = new List<Culture> { _enUsCulture, _ruRuCulture };
      _domainConfig = A.Fake<IDomainConfig>();
      A.CallTo(() => _domainConfig.DefaultCultureLocale).Returns("en-US");
```

```
            _cultureRepository = A.Fake<ICultureRepository>();
            A.CallTo(() =>
_cultureRepository.GetCulturesForLocale(A<string>.Ignored)).Returns(_cultures.As
Queryable());

            _userSession = new UserSessionUnderTest();
            _useCase = new SetCultureUseCase(_cultureRepository, _userSession,
_domainConfig);
        }

        [Test]
        public void
OnActionExecuting_GoodCultureSeoName_SetsUserSessionCulture()
        {
            var actual = _useCase.Act("RU");

            Assert.That(actual.Culture, Is.SameAs(_cultures.First(x => x.SeoName ==
"RU")));
            Assert.That(_userSession.Culture, Is.SameAs(_cultures.First(x =>
x.SeoName == "RU")));
        }

        [Test]
        public void OnActionExecuting_GoodCultureSeoName_IgnoresCase()
        {
            var actual = _useCase.Act("rU");

            Assert.That(actual.Culture, Is.SameAs(_cultures.First(x => x.SeoName ==
"RU")));
            Assert.That(_userSession.Culture, Is.SameAs(_cultures.First(x =>
x.SeoName == "RU")));
        }

        [Test]
        public void OnActionExecuting_BadCultureSeoName_ReturnsGlobal()
        {
            var actual = _useCase.Act("BAD");

            Assert.That(actual.Culture, Is.Null);
            Assert.That(actual.DefaultCulture.SeoName, Is.EqualTo("GLOBAL"));
```

```
        }

        [Test]
        public void
OnActionExecuting_WrongDefaultCultureLocale_ThrowsException()
        {
            A.CallTo(() => _domainConfig.DefaultCultureLocale).Returns("xx-XX");

            Assert.That(() => _useCase.Act("BAD"),
Throws.InvalidOperationException.With.Message.EqualTo("Default culture locale
[xx-XX] does not exist"));
        }

        private class UserSessionUnderTest: IUserSession
        {
            public Culture Culture { get; set; }
            public ConfigurableCategory Section { get; set; }
            public ConfigurableCategory Category { get; set; }
        }
    }
```

ПРИЛОЖЕНИЕ Б

Пример кода контроллера

```
public class AssetsController : BaseController
{
    private readonly IUseCaseFactory _useCaseFactory;
    private readonly IViewModelMapper _mapper;

    public AssetsController(IUseCaseFactory useCaseFactory, IViewModelMapper
mapper)
    {
        _useCaseFactory = useCaseFactory;
        _mapper = mapper;
    }

    public ViewResult Index(AssetType t = AssetType.ImageVideoAudio, int page
= 1)
    {
        var viewModel = new IndexViewModel
                {
                    AssetType = t,
                    Page = page,
                };

        return View(viewModel);
    }

    [ChildActionOnly]
    public PartialViewResult Listing(AssetType assetType, int page, int pageSize)
    {
        var useCase = _useCaseFactory.Create<GetAssetsUseCase>();
        var output = useCase.Act(assetType, page, pageSize);
        var viewModel = _mapper.Map<GetAssetsUseCase.Output,
ListingViewModel>(output);

        return PartialView(viewModel);
    }
```

```
        [ChildActionOnly]
        public PartialViewResult MediaForStory(string storyGuid, string storyLocale)
        {
            var useCase = _useCaseFactory.Create<GetAssetsForStoryUseCase>();
            var output = useCase.Act(storyGuid, storyLocale);
            var viewModel = _mapper.Map<GetAssetsForStoryUseCase.Output,
MediaForStoryViewModel>(output);

            return PartialView(viewModel);
        }

        [ChildActionOnly]
        public PartialViewResult HeroAsset(string storyGuid, string storyLocale)
        {
            var useCase = _useCaseFactory.Create<GetHeroAssetUseCase>();
            var asset = useCase.Act(storyGuid, storyLocale);

            return PartialView(asset);
        }

        public ViewResult Details(string guid)
        {
            var useCase = _useCaseFactory.Create<GetAssetUseCase>();
            var output = useCase.Act(guid);

            if (output.Asset == null)
            {
                throw new HttpException((int)HttpStatusCode.NotFound, "");
            }

            var viewModel = _mapper.Map<GetAssetUseCase.Output,
AssetDetailsViewModel>(output);

            return View(viewModel);
        }
    }
```