



Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

Отчет по учебной практике
Практикум по разработке РСОИ
Тема: «Новостной портал для размещения статей»

Студент ИУ7-23М
(Группа)

(Подпись, дата)

П.И. Кандрина
(И.О.Фамилия)

Руководитель

(Подпись, дата)

А.П. Ковтушенко
(И.О.Фамилия)

Оглавление

Глоссарий	3
Введение.....	3
Краткое описание предметной области	3
Основания для разработки	4
Назначение разработки.....	4
Существующие аналоги	4
Описание системы	4
Общие требования к системе.....	4
Требования к функциональным характеристикам	4
Функциональные требования к portalу с точки зрения пользователя	5
Требования к программной реализации	6
Топология системы.....	6
Общие требования к подсистемам.....	7
Функциональные требования к сервисам	8
Требования к надежности	9
Требования к документации.....	9
КОНСТРУКТОРСКИЙ РАЗДЕЛ.....	10
Концептуальный дизайн	10
Сценарии функционирования системы	11
Диаграммы прецедентов	12
Спецификации сценариев	14
Логический дизайн	16
Структура сервиса	20
Диаграммы последовательности действий.....	21
Диаграмма потоков данных.....	22
ТЕХНОЛОГИЧЕСКИЙ РАЗДЕЛ.....	24
Выбор языка и фреймворка для разработки системы	24
Выбор СУБД.....	25
Обеспечение масштабируемости.....	25
Реализация сервисной архитектуры	26
Пользовательский интерфейс	29
Сбор статистики.....	30
Реализация масштабируемости.....	32
Реализация отказоустойчивости.....	33
Реализация авторизации в системе через социальные сети	38
Заключение	49
РЕЗУЛЬТАТЫ ПРАКТИКИ.....	50
СПИСОК ЛИТЕРАТУРЫ.....	51

Глоссарий

1. Узел системы – региональный сервер, содержащий данные авторов и читателей указанного региона.
2. “Горячее” переконфигурирование системы – способность системы применять изменения без перезапуска и перекompиляции.
3. Медиана времени отклика – среднее время предоставления данных Пользователю.
4. Латентность географического положения – увеличение времени отклика приложения, обуславливаемое географическим положением элементов системы или пользователя.
5. Веб-интерфейс – интерфейс пользователя, предоставляемой системой через Web-браузер. В разрабатываемой системе только один веб-интерфейс.
6. Медиана – возможное значение признака, которое делит ранжированную совокупность (вариационный ряд выборки) на две равные части: 50% «нижних» единиц ряда данных будут иметь значение признака не больше, чем медиана, а «верхние» 50% — значения признака не меньше, чем медиана.
7. COA (SOA) – сервис-ориентированная архитектура (Service Oriented Architecture), модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных заменяемых компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам.
8. Сессия – серия запросов к portalу, сделанных одним пользователем в заданный промежуток времени, в данном документе - в течение 30 минут.
9. Аккаунт пользователя – информация о пользователе portalа, хранящаяся в portalе, в частности, имя, фамилия, электронный адрес и др.
10. Проект, портал, система – в данной работе термины «проект», «portal» и «система» взаимозаменяемы.
11. Автор – пользователь portalа, размещающий новостные статьи.
12. Читатель – пользователь portalа, имеющий возможность просмотра и оценки размещённых статей.
13. Статья - заголовок и соответствующий текст определённого автора.
14. Валидация - проверка данных на соответствие заданным условиям и ограничениям.
15. REST – архитектурный стиль взаимодействия компонентов распределённого приложения в сети.
16. Категория – общая тематика, позволяющая объединить статьи в одну группу.
17. Новость – статья, содержащая имя автора, заголовок и тело статьи.
18. Рейтинг автора – оценка, отражающая степень популярности статей автора.

Введение

Данное техническое задание составлено для разработки проекта «Новостной портал для размещения статей». Техническое задание выполнено на основе ГОСТ 19.201—78 «ЕСПД. Техническое задание. Требования к содержанию и оформлению» [5].

Краткое описание предметной области

На сегодняшний день всё больше и больше людей используют электронные площадки для поиска актуальной информации о том или ином событии. Такие приложения позволяют авторам набрать аудиторию читателей, а читателям, в свою очередь, получать наиболее «свежий» и популярный новостной контент, а также делиться им в других источниках с другими людьми.

Сервисы, позволяющие размещать новости, представляют собой агрегаторы актуальной информации и выполняются в виде portalа со списком авторов и читателей, позволяющего разбивать статьи на категории и давать им оценку.

Данное техническое задание определяет требования к разработке веб-портала для размещения новостных статей.

Основания для разработки

Разработка ведется в рамках проведения учебной практики по разработке РСОИ и лабораторных работ по курсу «Методология программной инженерии» на кафедре «Программное обеспечение ЭВМ и информационные технологии» факультета «Информатика и системы управления» МГТУ им. Н. Э. Баумана.

Назначение разработки

Назначение разрабатываемого портала – предоставление всем пользователям возможности просмотра и оценки статей публикуемых на портале авторов, а также подборок популярных и новых статей. Приложение также должно обеспечивать автору размещение его статьи. Автор использует статьи для повышения своего рейтинга на основе оценок читателей, которые, в свою очередь, используют портал для получения актуальной новостной информации.

Существующие аналоги

Среди аналогов можно отметить порталы «Livejournal», «Life.ru» и «Rusbase». Данный проект должен иметь следующие преимущества перед существующими аналогами:

- создание подборки популярных статей по заданной категории;
- предоставление возможности поделиться новостью в других источниках;
- создание подборки авторов с наивысшим рейтингом.

Описание системы

Проект должен представлять собой Web-сайт для просмотра и ранжирования новостных статей по категориям. Автор имеет возможность размещения новостных публикаций. Каждая публикация имеет показатель рейтинга, на основе которого формируются подборки самых популярных статей в данной категории и самых популярных авторов. Автор и читатель имеют возможность поделиться понравившейся статьей в других источниках.

Общие требования к системе

1. Разрабатываемое программное обеспечение должно обеспечивать функционирование системы в режиме 24/7/365 со среднегодовым временем доступности не менее 99.0%. Допустимое время, в течение которого система не доступна, за год должна составлять $24 \cdot 365 \cdot 0.01 = 87.6$ часа.
2. Время восстановления системы после сбоя не должно превышать 15 минут.
3. Система должна поддерживать возможность «горячего» переконфигурирования системы. Необходимо поддержать возможность добавления нового узла во время работы системы без рестарта.

Требования к функциональным характеристикам

1. По результатам работы модуля сбора статистики медиана времени отклика системы на запросы пользователя на получение информации не должна превышать 3 секунд без учета латентности географического расположения узла;
2. По результатам работы модуля сбора статистики медиана времени отклика системы на

запросы, добавляющие или изменяющие информацию на портале не должна превышать 7 секунд без учета латентности географического расположения узла;

3. Портал должен обеспечивать возможность запуска в современных браузерах: не менее 85% пользователей Интернета должны иметь возможность пользоваться порталом без какой-либо деградации функционала.

Функциональные требования к portalу с точки зрения пользователя

Портал должен обеспечивать реализацию следующих функций:

1. Система должна обеспечивать регистрацию и авторизацию авторов и читателей с валидацией вводимых данных через интерфейс приложения.
2. Система должна обеспечивать аутентификацию пользователей.
3. Система должна обеспечивать разделение пользователей на две роли:
 - Автор;
 - Читатель;
 - Администратор.
4. Система должна предоставлять **Автору** следующие функции:
 - добавление статьи;
 - редактирование статьи;
 - удаление статьи;
 - получение списка своих статей;
 - получение списка всех статей в заданной категории;
 - получение списка всех статей заданного автора;
 - изменение информации аккаунта;
 - просмотр статьи;
 - поделиться статьей.
5. Система должна предоставлять **Читателю** следующие функции:
 - получение списка всех статей в заданной категории;
 - получение списка всех статей заданного автора;
 - изменение информации аккаунта;
 - просмотр статьи;
 - оценка статьи;
 - поделиться статьей.
6. Система должна предоставлять **Администратору** следующие функции:
 - управление доступными категориями;
 - функционал для настройки, удаления, добавления узлов.

Входные данные Читателя:

- ФИО, не более 256 символов;
- Электронный почтовый адрес;

Входные данные Автора:

- ФИО, не более 256 символов;
- Электронный почтовый адрес;
- Статьи:
 - Заголовок;
 - Текст;
 - Категория.

Входные данные Администратора:

- ФИО, не более 256 символов;
- Электронный почтовый адрес.

Выходные данные

Выходными данными системы являются веб-страницы. В зависимости от запроса

Читателя или Автора они содержат:

- Список доступных для чтения статей;
- Список наиболее популярных статей и авторов в заданной категории;
- Список наиболее популярных авторов;
- Текст и заголовок запрашиваемой статьи;
- Дата размещения и оценка запрашиваемой статьи;
- Информацию об аккаунте пользователя.

Требования к программной реализации

1. Требуется использовать СОА (сервис-ориентированную архитектуру) для реализации системы;
2. Система состоит из микросервисов. Каждый микросервис отвечает за свою область логики работы приложения;
3. Взаимодействие между сервисами осуществляется посредством HTTP-запросов и/или очереди сообщений;
4. Данные сервисов должны храниться в базе данных. Каждый сервис взаимодействует только со своей схемой данных. Взаимодействие сервисов происходит по технологии REST;
5. При недоступности систем портала должна осуществляться деградация функционала или выдача пользователю сообщения об ошибке;
6. Необходимо предусмотреть авторизацию пользователей, как через интерфейс приложения, так и через популярные социальные сети;
7. Для запросов, выполняющих обновление данных на нескольких узлах распределенной системы, в случае недоступности одной из систем, необходимо выполнять полный откат транзакции;
8. Реализовать хранение аккаунта пользователя в базе данных в хэшированном виде.
9. Приложение должно поддерживать возможность горизонтального и вертикального масштабирования за счет увеличения количества функционирующих узлов и совершенствования технологий реализации компонентов системы.

Топология системы

Топология разрабатываемой системы, изображена на рисунке 1.

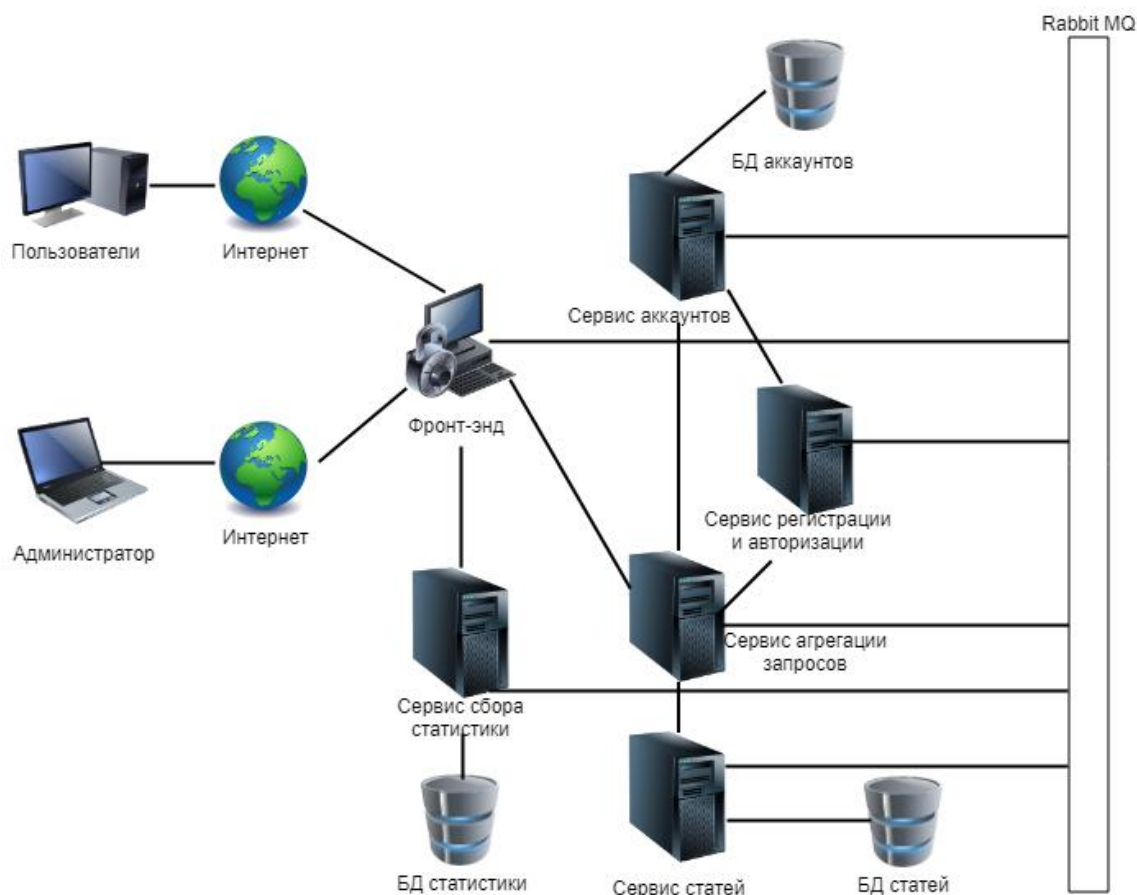


Рисунок 1. Топология системы.

Разрабатываемая система состоит из фронт-энда и 5 подсистем:

- Сервис статей;
- Сервис аккаунтов;
- Сервис сбора статистики.
- Сервис регистрации и авторизации;
- Сервис агрегации запросов;

Топология системы позволяет визуализировать связи между ее компонентами.

Общие требования к подсистемам

1. Фронт-энд — серверное приложение, при разработке которого следует учесть следующие нюансы:

- Фронт-энд должен принимать запросы по протоколу HTTP и формировать ответ пользователю в формате HTML-страниц;
- Фронт-энд является посредником между пользователями, передавая их запросы последовательно на сервис агрегации запросов;
- В соответствии с выбранными технологиями реализации целесообразно использовать библиотеку Bootstrap, язык HTML и Razor для создания верстки HTML-страниц проекта.

2. К реализации бэк-эндов должны быть предъявлены следующие требования:

- Прием и возврат данных должен происходить в формате JSON по протоколу HTTP;
- Если результаты работы сервиса необходимо сохранять в базе данных, то доступ к ней должен осуществляться по протоколу HTTP. Доступ к базе данных может осуществляться только из подсистем, работающих напрямую с данными ее таблиц.

Функциональные требования к сервисам

1. **Сервис агрегации запросов** – предоставляет пользовательский интерфейс и внешний API системы.

Сервис должен реализовывать следующий функционал:

- Проверка существования пользователя;
- Регистрация читателя/автора;
- Удаление читателя/автора;
- Изменение имени читателя/автора;
- Изменение пароля читателя/автора;
- Добавление статьи автора;
- Удаление статьи автора;
- Оценка статьи читателями;
- Редактирование статьи автора;
- Просмотр популярных статей в заданной категории;
- Просмотр популярных статей заданного автора;
- Просмотр рейтинг-листа авторов;
- Осуществление аутентификации пользователя;
- Получение списка доступных узлов системы (только для администратора);
- Добавление и удаление узлов системы (только для администратора);
- Добавление, редактирование и удаление категории (только для администратора).

2. **Сервис статей** – отвечает за добавление статей и хранение информации о них.

Хранимая в базе данных сущность, ассоциированная с сервисом, имеет следующие обязательные поля:

- Заголовок статьи;
- Текст статьи;
- Автор;
- Дата размещения;
- Категория;
- Оценка.

Сервис должен реализовывать следующий функционал:

- Добавление автором статьи (при этом необходимо обеспечить проверку существования у данного автора статьи с таким названием);
- Группировка в соответствии с выбранным количеством статей для отображения на странице;
- Удаление всех статей автора;
- Изменение идентификатора автора во всех его статьях;
- Получение всех статей автора в категории;
- Получение всех статей в категории;
- Увеличение оценки статьи читателями;
- Формирование списка наиболее популярных статей в заданной категории;
- Формирование списка наиболее популярных авторов;
- Получение всех авторов, имеющих статьи в указанной категории.

3. **Сервис аккаунтов**

Хранимая в базе данных сущность, ассоциированная с сервисом, имеет следующие

обязательные поля:

- Идентификатор читателя/автора;
- Имя читателя/автора;
- Пароль читателя/автора.

Сервис должен реализовывать следующий функционал:

- Проверка существования читателя/автора;
- Аутентификация читателя/автора;
- Регистрация нового читателя/автора;
- Удаление читателя/автора;
- Изменение имени читателя/автора;
- Изменение пароля читателя/автора.

4. Сервис сбора статистики

Сервис должен реализовывать следующий функционал:

- Получение статистики добавления статей за последний месяц;
- Получение статистики операций за выбранный промежуток времени.

5. Сервис регистрации и авторизации пользователей

Сервис должен реализовывать следующий функционал:

- Возможность регистрации нового аккаунта в системе;
- Возможность авторизации читателя/автора как через аккаунт в системе, так и через аккаунт предлагаемых социальных сетей.

Требования к надежности

Система должна работать в соответствии с данным техническим заданием без перезапуска. Необходимо использовать «зеркалируемые серверы» для всех подсистем, которые будут держать нагрузку в случае сбоя до тех пор, пока основной сервер не восстановится.

Требования к документации

Исполнитель должен подготовить и передать Заказчику следующие документы:

- руководство администратора Системы;
- руководство для читателя по использованию Системы;
- руководство для автора по использованию Системы;

КОНСТРУКТОРСКИЙ РАЗДЕЛ

Конструкторский раздел должен содержать более детальное описание функционала приложения по сравнению с тем, что было отражено в техническом задании. Использование общепринятых схем и диаграмм, а также пояснений к ним позволяет наглядно отобразить не только внешнее взаимодействие пользователя с приложением, но и внутреннее устройство системы, раскрывая детали реализации на более низком уровне.

Концептуальный дизайн

Концептуальный дизайн системы обычно содержит наиболее общие схемы описания функционала приложения с точки зрения пользователей [2]. Одной из таких схем является IDEF0-модель и графические модели, входящие в нее [3]. На рисунке 2 отображена контекстная диаграмма верхнего уровня, которая обеспечивает наиболее общее или абстрактное описание работы системы. Данный вид диаграммы позволяет формализовать описание запросов пользователя и ответов системы на данные запросы, отобразив систему в виде “черного” ящика.



Рисунок 2. Концептуальная модель системы в нотации IDEF0.

Для уточнения деталей работы системы применяется декомпозиция функций, отображенных на диаграмме верхнего уровня, при помощи создания дочерних диаграмм. В качестве примера на рисунке 3 изображена дочерняя диаграмма, которая определяет последовательность выполнения операций в системе при обработке запроса пользователя на получение информационного контента.

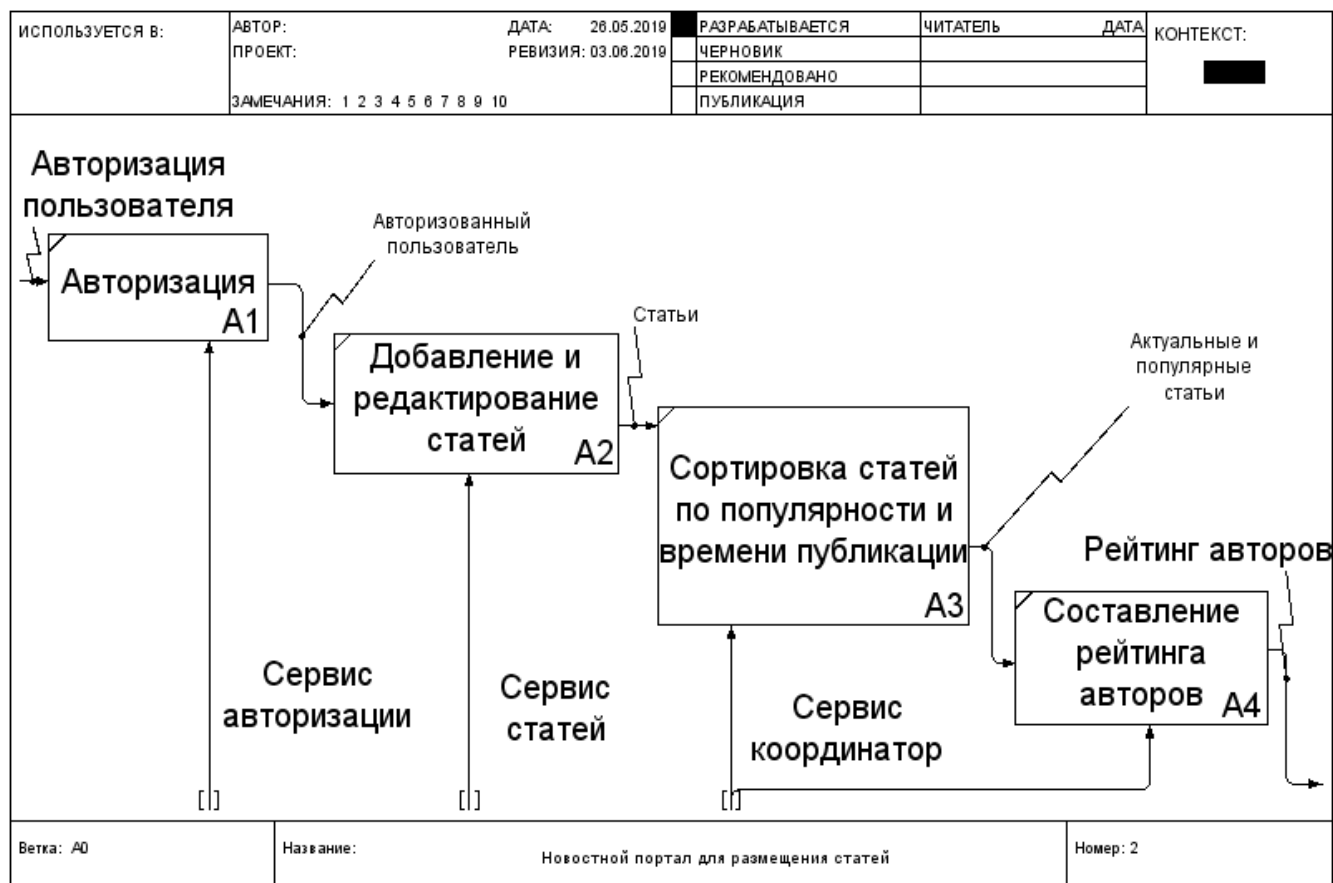


Рисунок 3. Детализированная концептуальная модель системы в нотации IDEF0.

Детализированная диаграмма, создаваемая при декомпозиции, охватывает ту же область, что и родительский блок, но описывает ее более подробно. Поэтому такая диаграмма может быть создана для любого из запросов, отображенных на диаграмме верхнего уровня. Каждый из блоков детализированной диаграммы может быть в свою очередь также описан при помощи дочерней диаграммы.

Сценарии функционирования системы

Сценарии функционирования или использования системы описывают конкретную последовательность действий, иллюстрирующую поведение пользователя при работе с приложением. Далее приведены подробные сценарии основных возможных действий пользователя.

Регистрация пользователя:

1. Пользователь нажимает на кнопку «Регистрация» в интерфейсе приложения;
2. Пользователь перенаправляется на страницу регистрации, которая содержит поля для заполнения данных (логин, пароль, подтверждение пароля);
3. Пользователь вводит данные в форму и для завершения регистрации нажимает на кнопку «Зарегистрироваться как автор» или «Зарегистрироваться как читатель», тем самым подтверждая верность своих данных, а также согласие на их обработку и хранение.
4. Если пользователь с введенным для регистрации именем уже существует, то клиент перенаправляется на страницу ошибки. При успешной регистрации пользователь перенаправляется на страницу своего профиля в системе.

Авторизация пользователя:

1. Пользователь нажимает на кнопку «Вход» в интерфейсе приложения;
2. Пользователь перенаправляется на страницу авторизации, которая содержит поля для ввода логина и пароля;
3. Пользователь завершает работу с формой авторизации нажатием кнопки «Войти»;
4. При обнаружении ошибки в данных, пользователь перенаправляется на страницу ошибки; при совпадении данных с записью в базе данных аккаунтов пользователь получает доступ к системе.

Получение ранжированного списка статей по авторам и по категориям:

1. Авторизованный пользователь нажимает на кнопку «Статьи»;
2. Пользователь перенаправляется на страницу, которая содержит список статей заданного автора и в заданной категории, ранжированных по популярности или по дате размещения.

Получение списка понравившихся статей:

1. Авторизованный пользователь нажимает на кнопку «Избранное»;
2. Пользователь перенаправляется на страницу, содержащую список оцененных пользователем статей.

Получение ранжированного списка авторов:

1. Авторизованный пользователь нажимает на кнопку «Рейтинг авторов»;
2. Пользователь перенаправляется на страницу, которая содержит список авторов, ранжированных по популярности.

Получение статистики:

1. Пользователь с ролью администратор нажимает на кнопку «Панель администратора»;
2. Пользователь перенаправляется на страницу просмотра статистики среднего времени обработки запроса сервисами.

Диаграммы прецедентов

Графически сценарии функционирования системы можно представить при помощи диаграмм прецедентов. Они позволяют схематично отобразить типичные сценарии взаимодействия между клиентами и приложением. В системе выделены 2 основных роли: пользователь и администратор, диаграммы прецедентов для этих ролей изображены на рисунках 4 и 5.

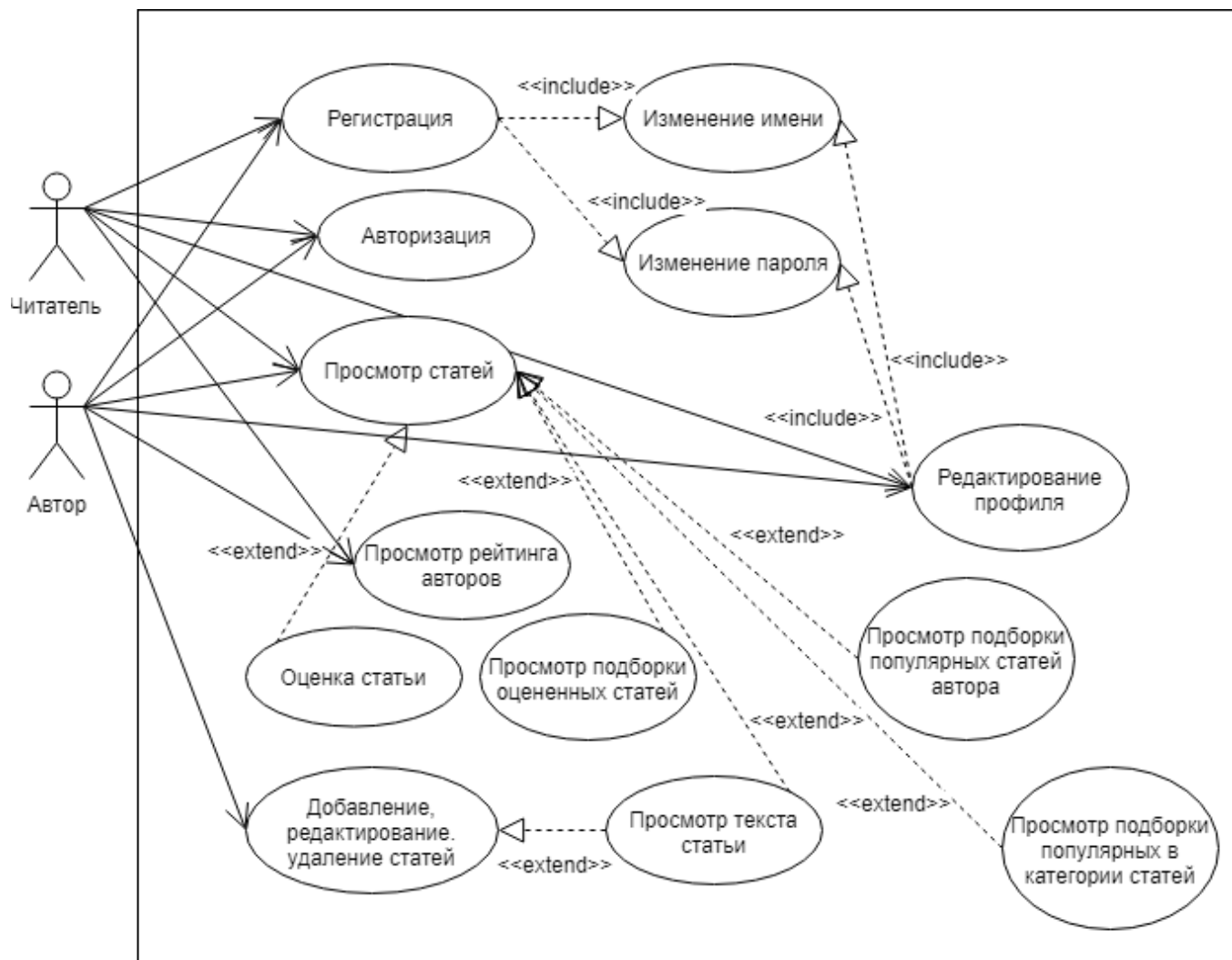


Рисунок 4. Диаграмма прецедентов с точки зрения пользователя.

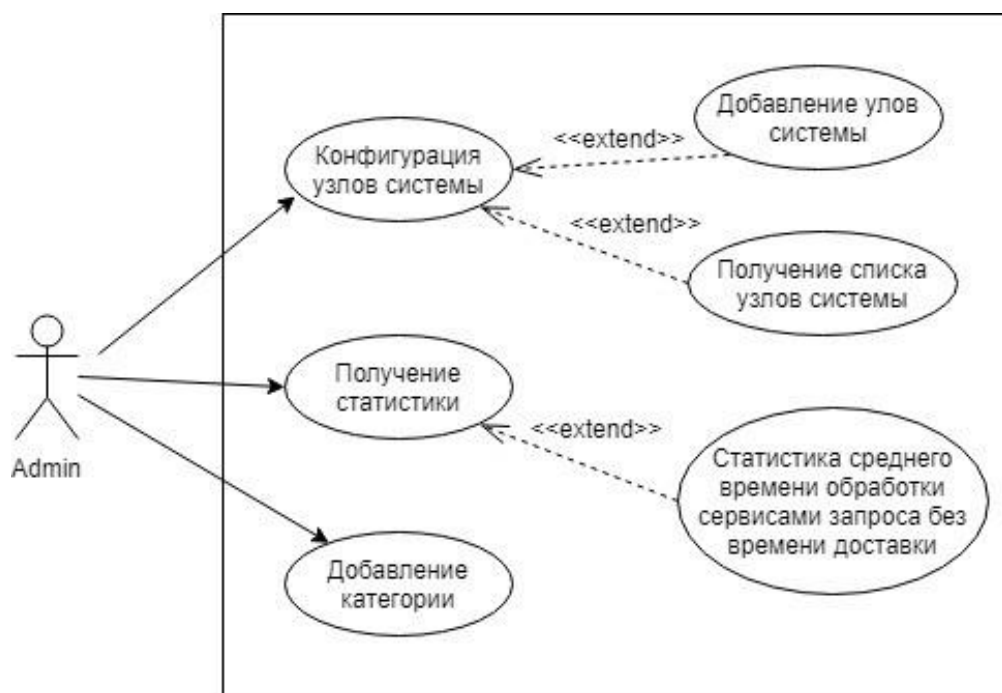


Рисунок 5. Диаграмма прецедентов с точки зрения администратора.

Спецификации сценариев

Приведенные сценарии могут иметь как основной поток выполнения, который выполняется чаще всего, так и альтернативные потоки, описывающие выполнение запроса при отклонении от основного хода сценария. Все возможные ходы выполнения сценария описываются при помощи спецификаций. Примеры спецификаций для описанных выше сценариев приведены в данном разделе.

Спецификация сценария «Регистрация»

Нормальный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Регистрация”	Открывается страница для ввода данных
Пользователь вводит данные в поля и нажимает кнопку “Зарегистрироваться как автор” или “Зарегистрироваться как читатель”	Открывается страница с профилем созданного пользователя

Альтернативный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Регистрация”	Открывается страница для ввода данных
Пользователь вводит данные существующего пользователя в поля и нажимает кнопку “Зарегистрироваться как автор” или “Зарегистрироваться как читатель”	Открывается страница с сообщением об ошибке, что пользователь с такими данными существует

Спецификация сценария «Авторизация»

Нормальный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Вход”	Открывается страница для ввода данных
Пользователь вводит данные в поля и нажимает кнопку “Войти”	Открывается страница профиля пользователя

Альтернативный ход сценария

Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Вход”	Открывается страница для ввода авторизационных данных
Пользователь вводит неверные данные в поля и нажимает кнопку “Войти”	Открывается страница ошибки с сообщением о неверно введенных данных

Спецификация сценария «Статьи по авторам»

Нормальный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Статьи” и выбирает интересующего автора или/и интересующую категорию	Открывается страница со списком статей заданного автора или/и в заданной категории, ранжированных по популярности и по дате размещения

Альтернативный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Статьи” при недоступности сервиса статей	Открывается страница ошибки с сообщением, что сервис статей недоступен

Спецификация сценария «Рейтинг авторов»

Нормальный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Рейтинг авторов”	Открывается страница со списком авторов, ранжированных по популярности.

Альтернативный ход сценария	
Действие пользователя	Отклик системы
Пользователь нажимает кнопку “Рейтинг авторов” при пустом списке авторов	Открывается страница с сообщением, что список авторов пуст

Спецификация сценария «Избранное»

Нормальный ход сценария

Действие пользователя	Отклик системы
Читатель нажимает кнопку “Избранное”	Открывается страница со списком оцененных статей

Альтернативный ход сценария	
Действие пользователя	Отклик системы
Читатель нажимает кнопку “Избранное” при отсутствии оцененных статей	Открывается страница с сообщением, что список оцененных статей пуст

Спецификация сценария «Мои статьи»

Нормальный ход сценария	
Действие пользователя	Отклик системы
Автор нажимает кнопку “Мои статьи”	Открывается страница со списком написанных статей

Альтернативный ход сценария	
Действие пользователя	Отклик системы
Автор нажимает кнопку “Мои статьи” при отсутствии написанных статей	Открывается страница с сообщением, что список написанных статей пуст

Спецификация сценария «Получение статистики»

Нормальный ход сценария	
Действие пользователя	Отклик системы
Пользователь авторизуется как администратор	Открывается страница с профилем пользователя
Пользователь нажимает кнопку “Панель администратора”	Открывается страница, содержащая статистику среднего времени обработки запроса сервисами

Логический дизайн

В процессе создания концептуального дизайна системы были отражены основные сценарии взаимодействия пользователя и системы. В разделе логического дизайна необходимо представить организацию элементов системы и их взаимодействие между собой.

На основе функциональных требований к выделенным подсистемам, а также объектов,

о которых необходимо хранить данные в системе, была разработана схема данных приложения. Результат ее проектирования отображен на условной ER-диаграмме, где овалами обозначены ключевые сущности, а ромбами - связи между ними. Участие сущности в отношении с другой сущностью отмечается линией, соединяющей их. Число, располагающееся около линии, означает тип связи между соединенными сущностями. ER-диаграмма структуры данных представлена на рисунке 6.

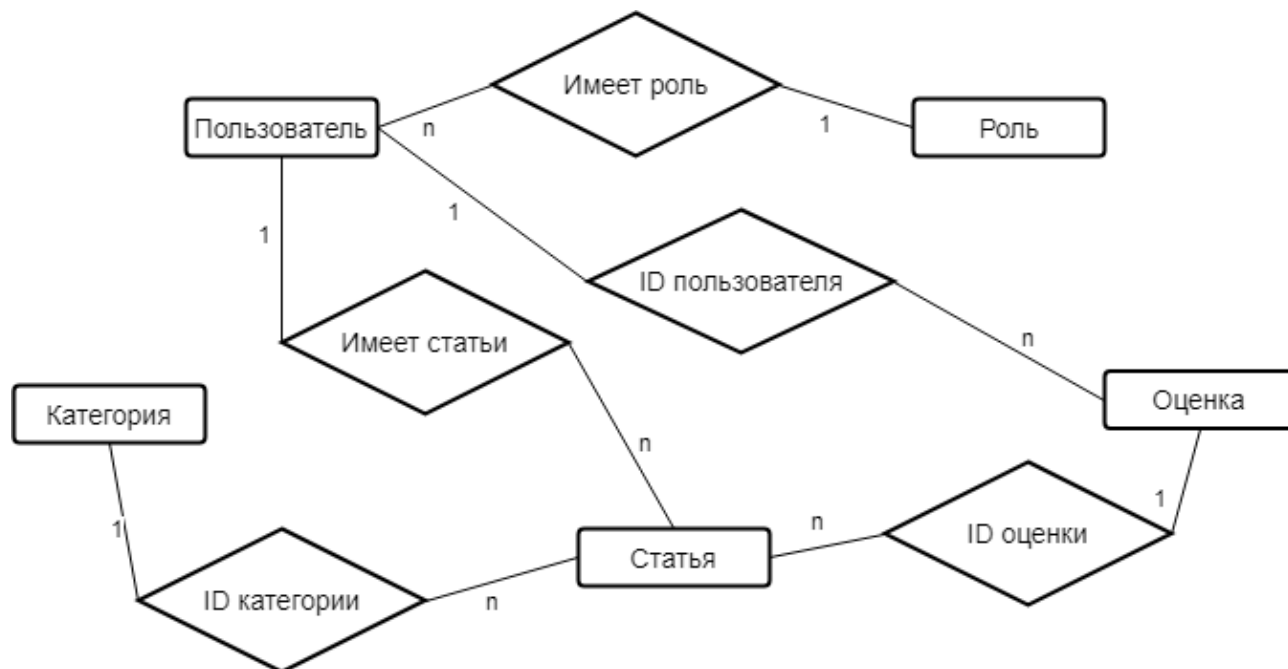


Рисунок 6. ER-диаграмма данных приложения.

Ниже приведена таблица соответствия сущностей ER-диаграммы объектам базы данных.

Название в ER-диаграмме	Название в БД
Пользователь	User
Роль	Role
Статья	Article
Категория	Category
Оценка	Mark

На следующей стадии проектирования, добавив в схему данных атрибуты сущностей, получаем схему базы данных, которая изображена на рисунке 7.

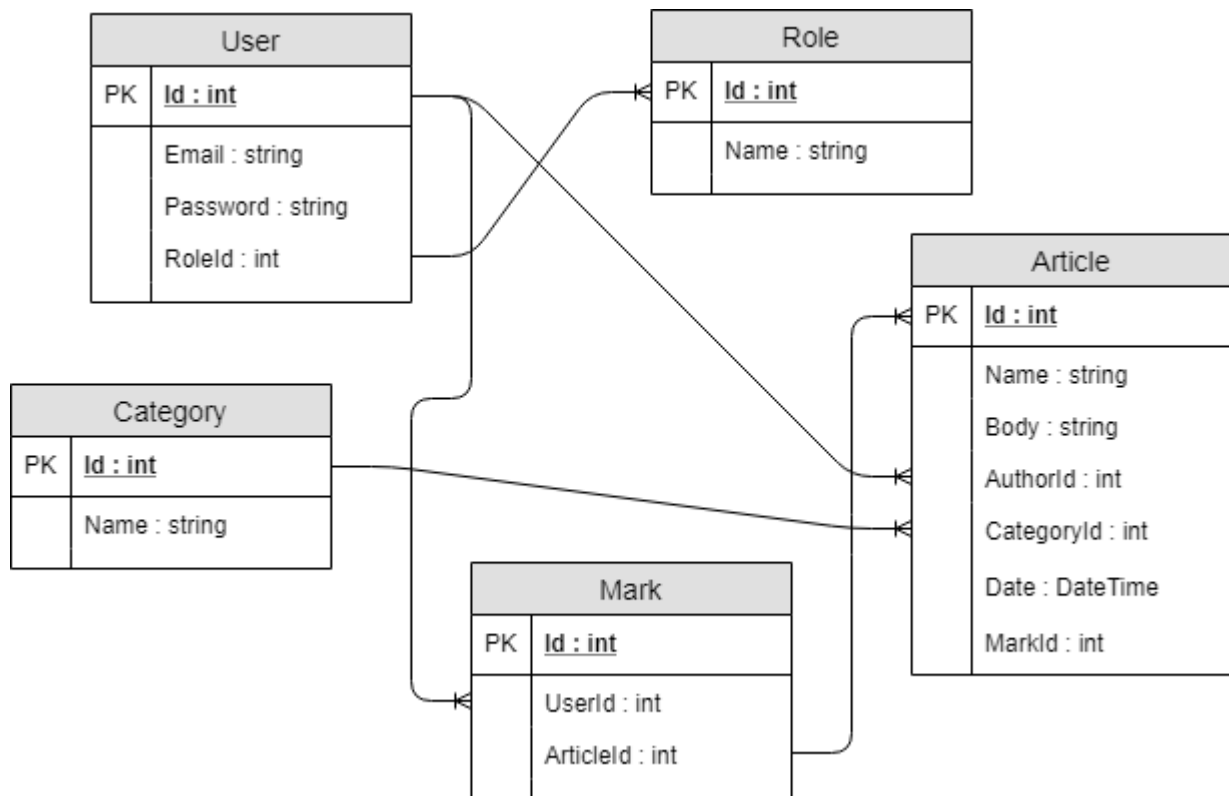


Рисунок 7. Схема базы данных системы.

Далее приводятся спецификации таблиц базы данных, приведенных на рисунке 7.

Спецификация таблицы User

Таблица User содержит данные о профиле пользователя.

Имя атрибута	Тип атрибута	Описание атрибута
Id	Public : int	Идентификатор пользователя
Email	Public : string	Логин пользователя
Password	Public : string	Пароль пользователя
RoleId	Public : int	Роль (автор, читатель или администратор)

Спецификация таблицы Category

Таблица Category предназначена для работы с категориями статей.

Имя атрибута	Тип атрибута	Описание атрибута
Id	Public : int	Идентификатор категории

Name	Public : string	Название категории
------	-----------------	--------------------

Спецификация таблицы **Articles**

Таблица Articles является основной для сервиса статей и содержит в себе всю необходимую информацию о статьях.

Имя атрибута	Тип атрибута	Описание атрибута
Id	Public : int	Идентификатор статьи
Name	Public : string	Название статьи
Body	Public : string	Текст статьи
AuthorId	Public : int	Идентификатор автора статьи
Date	Public : DateTime	Дата публикации
CategoryId	Public : int	Идентификатор категории статьи
MarkId	Public : int	Идентификатор оценки статьи

Спецификация таблицы **Mark**

Таблица Mark предназначена для работы с историей просмотров.

Имя атрибута	Тип атрибута	Описание атрибута
Id	Public : int	Идентификатор оценки статьи
ArticleId	Public : int	Идентификатор статьи
UserId	Public : string	Идентификатор пользователя, давшего оценку

Спецификация таблицы **Role**

Таблица Category выделена для работы с ролями пользователей.

Имя атрибута	Тип атрибута	Описание атрибута
Id	Public : int	Идентификатор роли
Name	Public : string	Название роли

Структура сервиса

На основе разработанной схемы данных можно установить соответствие сущностей и сервисов, описанных в предыдущем разделе. Например, для работы с сущностью Article был выделен сервис статей. Рассмотрим его устройство более подробно. Так как приложение построено на основе модели MVC, каждый сервис имеет хотя бы один контроллер и, хотя бы одну модель данных. Для связи модели и базы данных создается класс ArticleContext, который наследует функционал от системного класса DbContext от .NET Core. Данный класс позволяет получать из хранилища набор данных для реализации дальнейших действий над ним. В качестве примера на рисунке 8 представлена диаграмма классов для разработки сервиса статей.

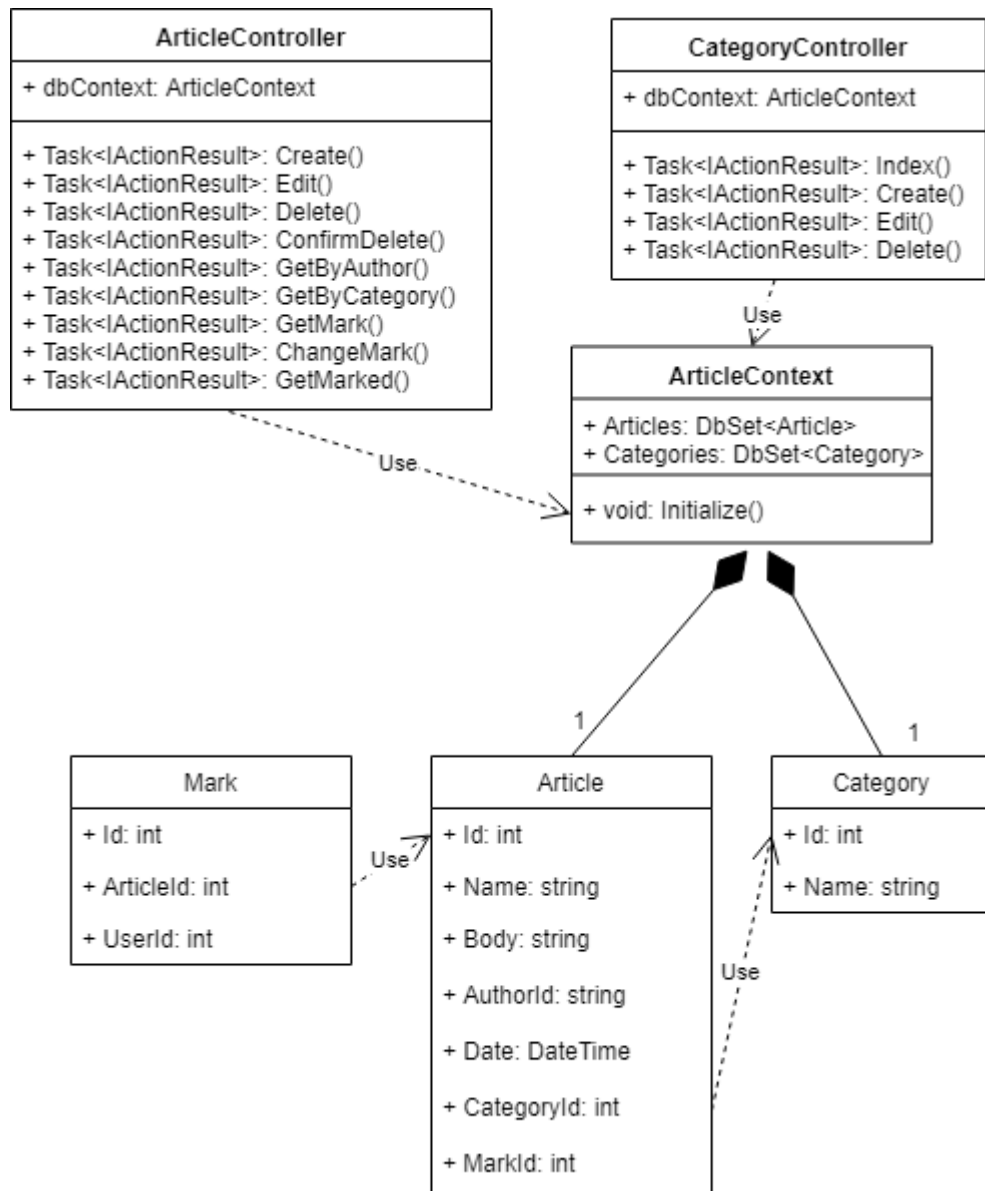


Рисунок 8. Диаграмма классов сервиса статей.

Функциональные требования, предъявляемые к сервису статей, реализуются при помощи методов контроллера ArticleController. Далее приведено описание каждого метода данного контроллера.

Спецификация класса ArticleController

Метод	Описание
Create	Добавление статьи в базу данных
Edit	Редактирование статьи
Delete	Удаление статьи
ConfirmDelete	Подтверждение удаления статьи
GetByAuthor	Получение статей указанного автора
GetByCategory	Получение статей указанной категории
GetMarked	Получение статей, оцененных пользователем
GetMark	Получить оценку статьи
ChangeMark	Изменить оценку статьи

Диаграммы последовательности действий

Для описания поведения компонентов системы на единой оси времени используются диаграммы последовательности действий, при помощи которых можно описать последовательность действий для каждого прецедента, необходимую для достижения цели. Например, на рисунке 9 изображен процесс получения рейтинга авторов.

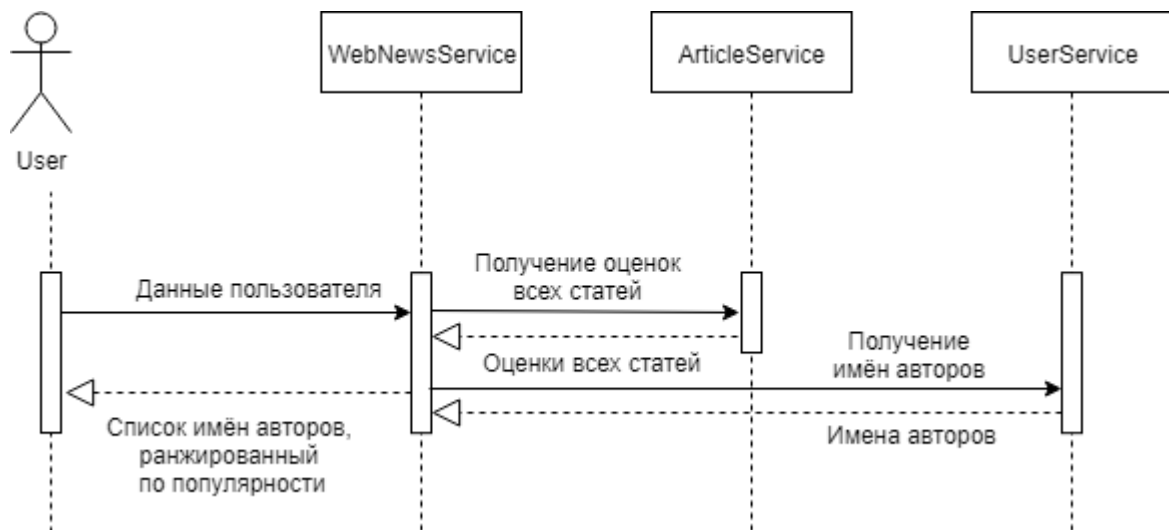


Рисунок 9. Диаграмма последовательности действий при запросе пользователем рейтинга авторов.

Главный сервис приложения отправляет запрос на получение оценок статей всех пользователей. После получения данных главный сервис обращается к сервису пользователей, чтобы получить имена авторов, которые разместили свои статьи. После окончания данной процедуры главный сервис ранжирует авторов по средней оценке их статей, формирует веб-страницу и возвращает ее пользователю.

Диаграмма потоков данных

Рассматриваемая система предполагает распределенное хранение данных. Все данные системы предполагают хранение в единой базе данных, хранилищами данных являются таблицы. Диаграмма потоков данных, представленная на рисунке 10, отображает модель информационной системы с точки зрения хранения, передачи и обработки данных во время обработки запроса пользователя на получение рейтинга авторов.

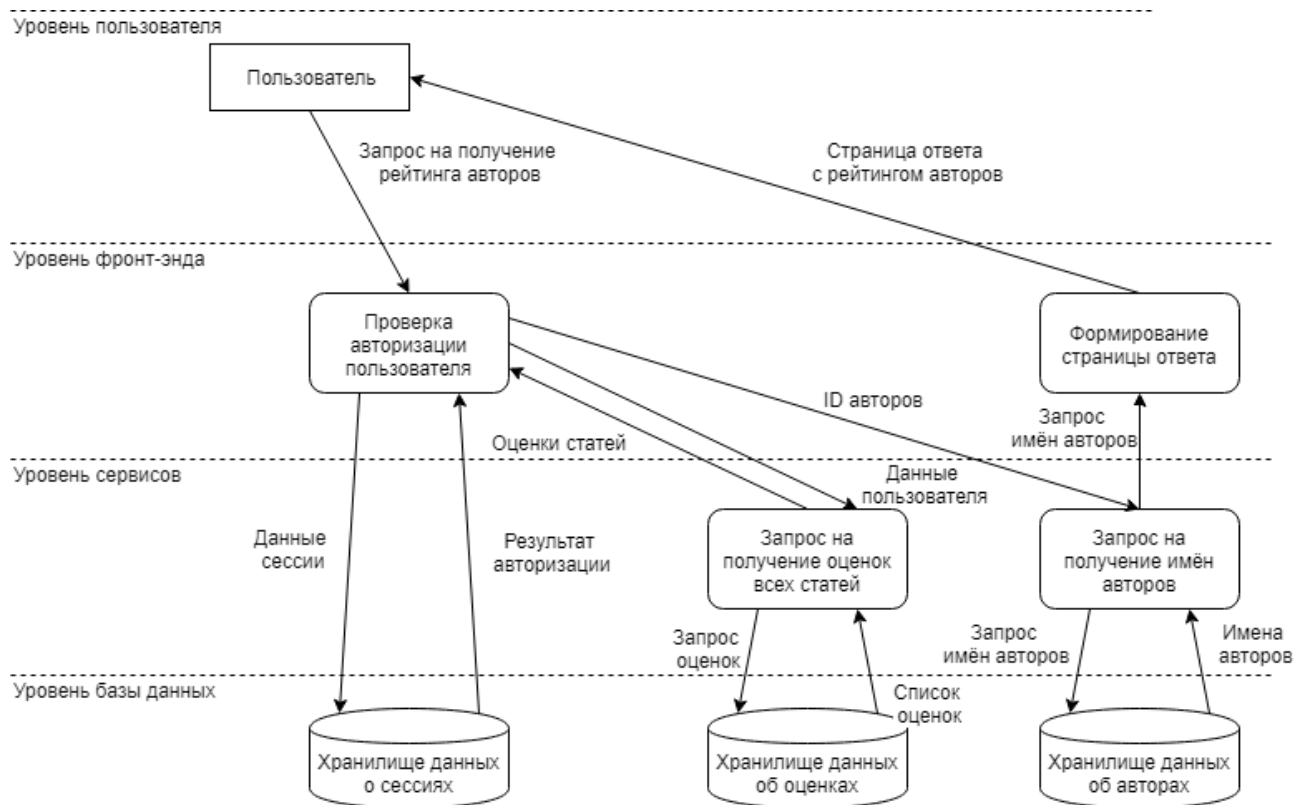


Рисунок 10. Диаграмма потоков данных при запросе пользователем рейтинга авторов.

ТЕХНОЛОГИЧЕСКИЙ РАЗДЕЛ

Технологический раздел должен содержать детальное описание выбранного архитектурного стиля взаимодействия сервисов, используемых фреймворка и СУБД. А также описание выполнения требований к программной реализации РСОИ (масштабируемость, отказоустойчивость и т.д.)

СОА не опирается на какую-либо конкретную технологию и вместо этого предоставляет набор принципов построения и взаимодействия компонентов системы. Главным принципом является разделение ответственности между сервисами, а также их независимость друг от друга. Каждый из них должен обладать открытым интерфейсом, через который его могут использовать другие сервисы. Использование интерфейсов позволяет менять реализацию сервиса, не вызывая необходимости изменять обращения к нему во всей системе. Также это позволяет с легкостью горизонтально масштабировать систему путем добавления сервиса-балансировщика между вызывающим и вызываемым сервисами [5]. Такой балансировщик отвечает интерфейсу того сервиса, нагрузку на который он балансирует, что делает его незаметным для вызывающей стороны, в то время как балансировщик распределяет получаемые запросы на ряд доступных ему сервисов. Проблемы могут возникнуть только при хранении состояния на сервисах, так как в этом случае необходимо дополнительно отслеживать какой из вызывающих сервисом как какому вызываемому обращался.

Архитектурным стилем взаимодействия между сервисами выбран REST (Representational State Transfer — «передача состояния представления») [6]. Данный стиль подразумевает отсутствие хранения состояния сервисами и передачу всей необходимой информации в теле запроса. REST опирается на протокол HTTP и использует существующие типы сообщений и их содержимое, например, HTTP GET для получения информации об объектах или HTTP POST для добавления информации. Также REST содержит рекомендации по именованию объектов и запросов, которые позволяют в дальнейшем облегчить сопровождение системы.

Выбор языка и фреймворка для разработки системы

В качестве языка реализации серверной части приложения выбран C#, так как он является одним из ведущих языков в разработке информационных систем на сегодняшний день, а также обладает всеми необходимыми особенностями для реализации СОА и REST. Для системы выбран фреймворк .Net Core 2.0 [7]. Он пришел на замену .Net Framework, и, в отличие от последнего, обладает возможностью запуска на разных плат-

формах, использует только необходимые библиотеки и быстро развивается.

В рамках фреймворка .Net Core 2.0 доступна технология ASP (Active Server Pages — «активные серверные страницы»). Она позволяет динамически создавать страницы на стороне сервера и отправлять их пользователю. Такой способ создания веб-страниц для работы с пользователем называется серверной шаблонизацией или «server-rendering». Серверная шаблонизация выполняется на одном из сервисов системы, который работает с пользователем, обращаясь при необходимости к другим сервисам.

Сама технология ASP содержит в себе фреймворк ASP.NET MVC (Model-View- Controller, «Модель-Представление-Контроллер») [8]. Он, как понятно из названия, основан на архитектуре приложения MVC, и содержит три основных компонента:

- Модель – данные и работа с ними;
- Представление – отображение данных модели;
- Контроллер – реакция на действия пользователя и оповещение модели о необходимости изменений.

Данная архитектура позволяет четко разделять ответственность между компонентами,

отделять бизнес-логику от её визуализации. Например, для добавления поддержки мобильных устройств достаточно добавить только соответствующее представление, не меняя модель и контроллер.

Выбор СУБД

Фреймворк .Net Core включает в себя фреймворк работы с реляционными базами данных Entity Framework Core [9]. Данный фреймворк позволяет обращаться к широкому спектру реляционных баз данных при помощи унифицированного интерфейса. Благодаря этому возможно менять СУБД в любой момент при минимальных изменениях в системе. Так как, согласно ТЗ, в базе данных не требуется хранить сложные объекты, например, файлы, для проекта подходит реляционная база данных. Поэтому для хранения данных была выбрана СУБД Microsoft SQL, как оптимальный вариант в связи с широким знакомством с ней в рамках курса «Базы данных».

Обеспечение масштабируемости

Как было отмечено выше, СОА позволяет масштабировать систему горизонтально с использованием сервисов-балансировщиков. Совместное использование СОА и REST- стиля, запрещающего сервисам хранить состояние, предоставляет возможность с легкостью добавлять и удалять сервисы, динамически распределяя нагрузку между существующими. Однако можно воспользоваться глобальным сервисом-координатором. Технология ASP позволяет выстраивать конвейер по обработке каждого запроса к сервису [8]. Добавив в этот конвейер этап для сбора статистики запросов, можно отслеживать нагрузку, не прибегая к добавлению сервиса-балансировщика. На основании полученной информации, сервис-координатор будет определять нагрузку на каждый из сервисов и, основываясь на этой информации, динамически определять связи между сервисами с целью минимизации задержек и снижения нагрузки. Пример работы координатора показан на Рис.12.

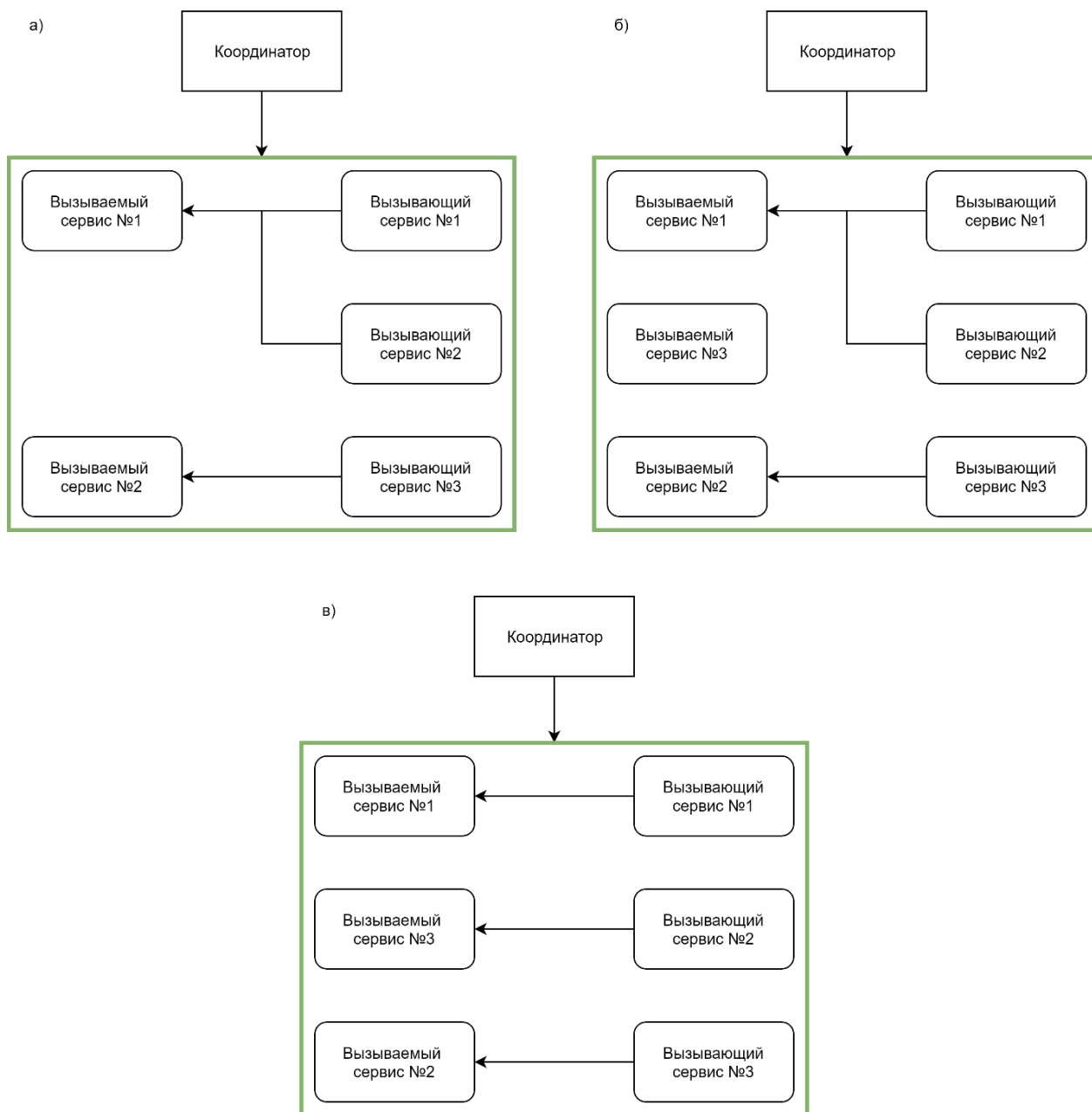


Рис. 12. Пример работы координатора. а) начальное состояние системы, состоящей из 3 однотипных вызывающих сервисов и 2 вызываемых; б) добавление нового вызываемого сервиса; в) балансировка нагрузки координатором путем переноса запросов вызывающего сервиса №2 на вызываемый сервис №3.

Реализация сервисной архитектуры

Общая архитектура сервиса

Для соответствия стилям COA и REST необходимо разбить систему на отдельные сервисы и организовать взаимодействие между ними через интерфейсы, без прямого обращения. Каждый сервис создается как веб-приложение ASP.Net Core. Данный шаблон проекта имеет несколько дочерних шаблонов, включающих в себя веб-API и веб-приложение. Основным отличием между этими двумя типами являются создаваемые по умолчанию файлы: в веб-API присутствует минимальный набор файлов, ориентированный на прием и отправку HTTP-запросов, тогда как веб-приложение сразу обладает возможностями для создания веб-страниц путем серверной шаблонизации (которые можно добавить и в веб-API). Для всех сервисов, не взаимодействующих

напрямую с пользователем, достаточно веб-API, а для взаимодействующих – веб-приложения.

Основным элементом сервисов являются контроллеры – классы, методы которых вызываются в ответ на запросы извне. Важно понимать, что объекты контроллеров не хранятся в приложении, а создаются при поступлении запроса и удаляются после его обработки. Таким образом, каждый контроллер имеет изолированные от других переменные, что, с одной стороны, делает систему менее гибкой, но с другой позволяет избежать большого числа ошибок. Контроллеры не создаются напрямую, за этим следит основная система веб-приложения совместно с системой внедрения зависимостей. Пример простого контроллера приведен в листинге 1.

```
[Route("home")]
public class HomeController : Controller
{
    private DbContext dbContext;
    public HomeController(DbContext dbContext)
    {
        this.dbContext = dbContext;
    }

    [HttpGet("about")]
    public IActionResult About()
    {
        return StatusCode(200, dbContext.TestCollection().First());
    }
}
```

Листинг 1 – Простой контроллер.

В листинге 1 показан простейший контроллер под названием Home (часть имени класса «Controller» принято опускать), получающий в конструкторе объект контекста базы данных (об этом далее). Единственным методом данного контроллера является метод About(), который будет вызван при обращении по адресу “/home/about”, который складывается из адреса контроллера, указанного в атрибуте Route и адреса метода, указанного в атрибуте HttpGet. Второй атрибут также определяет метод HTTP, который должен быть использован для обращения. В ответ на данный запрос будет получен код 200 (Ok в словаре кодов HTTP) и первый элемент коллекции объектов TestCollection.

Все приложения ASP.Net Core сразу снабжаются системой DI – Dependency Injection, внедрение зависимостей. Она отвечает за передачу нужных аргументов во все конструкторы классов, которые ей известны, а также за их инициализацию. Конфигурация данной системы происходит в файле Startup.cs. В нем, в методе ConfigureServices(...) происходит подключение допустимых классов к общей системе DI. По умолчанию, к ней также подключаются все контроллеры. Поэтому для получения объектов достаточно просто указать их как аргументы конструктора контроллера, как например, DbContext. Пример простой конфигурации DI приведен в листинге 2.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddDbContext<DbContext>();
    services.AddTransient<TransientService>();
    services.AddSingleton<SingletonService>();
}
```

Листинг 2 – Простая конфигурация DI.

Каждый подключаемый к системе DI сервис может быть объявлен либо как временный (Transient), либо как постоянный (Singleton). Временные сервисы создаются каждый

раз, когда в них возникает необходимость. Примером временных сервисов служат контроллеры. Постоянные сервисы остаются в памяти между запросами на их получение. Такие сервисы используются для хранения больших объемов данных, повторное создание которых является дорогостоящей операцией.

Базы данных, поддерживаемые Entity Framework, в свою очередь, особым образом подключаются к системе DI при помощи функции `AddDbContext<>()`. Контекст базы данных является временным сервисом, имеющим ограниченную область видимости. Это важное свойство, так как оно не позволяет получать контекст базы данных в постоянные сервисы. Это обусловлено тем, что поддержание постоянного соединения с базой данных (поддерживаемой Entity Framework) является плохим тоном (из-за возникающей нагрузки на СУБД). Однако, так как контроллеры являются временными объектами DI, получения контекста в них допустимо.

Сам контекст содержит в себе список коллекций объектов, хранящихся в базе данных. Он не привязан к конкретному представлению или СУБД, поэтому один контекст может быть использован для различных баз данных. Пример простого контекста приведен в листинге 3:

```
public class ApplicationDbContext : DbContext
{
    public DbSet<SimpleClass> SimpleCollection { get; set; }

    public ApplicationDbContext(DbContextOptions options) : base(options)
    {
    }

    protected ApplicationDbContext()
    {
    }
}
```

Листинг 3 – Простой контекст базы данных.

Приведенный в листинге 3 класс контекста базы данных содержит одну коллекцию объектов класса `SimpleClass`. В каждом классе, наследующем класс `DbContext` необходимо создавать конструктор от аргумента `DbContextOptions`. Создание второго конструктора (без аргументов) не обязательно, однако может потребоваться для тестирования.

Взаимодействие между сервисами

Для взаимодействия между сервисами используется протокол HTTP. Для общения между сервисами на главном (здесь и далее главным сервисом будет называться сервис, взаимодействующий с пользователем) сервисе создаются классы – интерфейсы сервисов. Централизация расположения этих интерфейсов облегчает дальнейшую модификацию системы, например, добавление общих заголовков к отправляемым запросам [10].

Базовым классом всех сервисов является `Service`, в котором определены функции-обертки над низкоуровневыми операциями по отправке HTTP-запросов и анализе ответов. Пример простейшей обертки над отправкой GET-запроса приведен в листинге 4:

```
protected async Task<HttpResponseMessage> Get(string addr)
{
    using (var client = new HttpClient()) try
    {
        return await client.GetAsync(GetAddress(addr));
    }
    catch { return null; }
}
```

Листинг 4 – Обертка над GET-запросом.

Запрос в листинге 4 возвращает либо ответ от сервиса (как объект класса `HttpResponseMessage`), либо `null` в случае ошибки. Функция поддерживает асинхронное выполнение, о чем свидетельствует тип возвращаемого значения `Task<>` и наличие ключевого слова `async` в заголовке метода. Механика работы системы `async/await` сложна, в отличии от её использования. Для асинхронной работы достаточно пометить метод словом `async`, обернуть возвращаемый тип в `Task<>` и вызвать метод, применяя ключевое слово `await` (как вызывается `GetAsync()` у объекта `client`). Разработанные таким образом методы будут выполняться параллельно, если это возможно без нарушения логики работы.

Пользовательский интерфейс

В приложениях ASP.Net Core используется технология серверной шаблонизации, при которой страница генерируется на сервере в соответствии с текущими данными, а затем отправляется пользователю. В ответ на его действия генерируются новые страницы. Для описания шаблона в формате `.cshtml` используется HTML вместе с языком Razor, который позволяет с легкостью использовать объекты языка C# внутри шаблона страницы. Пример простой страницы приведен в листинге 6.

```
@{
    ViewData["Title"] = "Ошибка";
}
@model NewsFeed.Models.Shared.ErrorModel
<h2>Ошибка</h2>
Код ошибки: @Model.Code
<hr />
Текст ошибки: @Model.Message
```

Листинг 6 – Шаблон страницы.

В листинге 6 приведен простой шаблон страницы с сообщением об ошибке. Он начинается с указания свойства `Title` словаря `ViewData`, которое обычно используется в качестве заголовка страницы. Далее указывается используемая на странице модель, то есть набор данных, для которых данная страница служит представлением (в соответствии со схемой Модель-Представление-Контроллер, MVC [8]). Сами страницы генерируются из контроллеров в ответ на обращения к ним. После указания модели идет обычный HTML код. Части, написанные с использованием Razor, начинаются с символа `@`. Например, такой частью является `@Model.Code`, который соответствует коду ошибки из модели. При отправке данной страницы пользователю все элементы Razor будут заменены на HTML код или текст.

Шаблоны по умолчанию должны располагаться в папке `Views` в корневой директории проекта, в подпапках, соответствующих контроллерам. Например, шаблоны, относящиеся к контроллеру `HomeController`, должны располагаться в папке `Views/Home`. Исключение составляют два служебных шаблона, `_ViewImports.cshtml` и `_ViewStart.cshtml`. Шаблон

`_ViewImports.cshtml` содержит в себе импортируемые во все шаблоны пространства имен, а также вспомогательные элементы Razor. В `_ViewStart.cshtml` находятся общие для всех шаблонов директивы Razor. Например, в файле `_ViewStart.cshtml` обычно указывают используемый макет страниц (мастер-страницу), который будет применен к каждому шаблону страницы.

Макет обычно содержит в себе большую часть разметки HTML, используемую в обычных страницах, а также навигационные элементы. Располагается он как правило в папке `Views/Shared`, под названием `_Layout.cshtml`. Пример простого макета приведен в листинге 7.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>@ViewData["Title"]</title>
```

```

        <environment exclude="Development">
        <link                                                    rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/css/bootstrap.min.css" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
        </environment>
    </head>
    <body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
    <div class="navbar-header">
    <a asp-controller="Home" asp-action="Index" class="navbar-brand">Домашняя
    страница</a>
    </div>
    </div>
    </nav>
    <div class="container body-content"> @RenderBody()
    <hr />
    <footer>
    <p>&copy; 2018</p>
    </footer>
    </div>
    </body>
    </html>

```

Листинг 7 – Макет страницы.

В листинге 7 большая часть является обычным кодом HTML страницы, хотя и с некоторыми добавлениями. В частности, здесь всё также используются элементы языка Razor, например, при обозначении заголовка страницы. Также есть вспомогательные теги HTML, такие как `<environment>`. Содержимое данного тега будет включаться (include) или исключаться (exclude) в зависимости от среды выполнения приложения. Это может быть полезно для отладки.

В макете также показан пример использования дополнительных атрибутов для указания ссылок на действия контроллеров на примере домашней страницы. Для создания ссылки на неё используется не относительный путь, а атрибуты `asp-controller=имя_контроллера` и `asp-action=имя_функции`. При их указании финальная ссылка на действие будет генерироваться динамически при отправке страницы пользователю, что позволяет избежать проблем при смене адресов доступа в приложении.

Наиболее важным элементом макета является вызов функции `RenderBody()`. Именно на её место в дальнейшем будет вставлена страница, которую запросил пользователь.

Сбор статистики

Для сбора статистики используется брокер очередей RabbitMQ. Его использование обуславливается тем, что сообщения статистики необходимо отправлять быстро и не дожидаясь ответа от сервера. Брокер при этом хранит сообщения и выдает их по требованию сервису сбора статистики, который, в свою очередь, сохраняет необходимую статистику в базу данных.

Сервис статистики оформляется как веб-API приложение, так как от него не требуется серверной шаблонизации. Также на данном сервисе хранятся классы для взаимодействия с брокером RabbitMQ. RabbitMQ — платформа, реализующая систему обмена сообщениями между компонентами программной системы на основе стандарта AMQP (от англ. Advanced Message Queuing Protocol - открытый протокол для передачи сообщений между компонентами системы) [11][12]. Другими словами, в разрабатываемой системе Rabbit MQ - это программное обеспечение, при подключении к которому сервисы могут отправлять и получать сообщения. Необходимость отправки сообщения сервисом задается разработчиком. В свою очередь менеджер очередей Rabbit MQ сохраняет отправленные сервисами сообщения до тех пор, пока другие сервисы, которым было адресовано сообщение, не подключатся и не получат его. Для хранения сообщений брокер использует очереди, при подключении к которым каждый из компонентов системы определяет сообщения, на которые он будет подписан. Такой обмен позволяет упростить

взаимодействие компонентов и достигнуть экономии ресурсов, а также обеспечить независимость компонентов и гарантировать последовательную обработку данных.

Взаимодействие источников сообщений с брокером является полностью асинхронным, а взаимодействие получателя – сервиса статистики – асинхронным, основанным на событиях. Метод отправки сообщений в очередь приведен в листинге 8:

```
public void Publish(IntegrationEvent @event)
{
    var eventName = @event.GetType().Name;
    var factory = new ConnectionFactory() { HostName = _connectionString };
    using (var connection = factory.CreateConnection())
    using (var channel = connection.CreateModel())
    {
        channel.ExchangeDeclare(exchange: _brokerName,
                                type: "direct");
        string message = JsonConvert.SerializeObject(@event);
        var body = Encoding.UTF8.GetBytes(message);
        channel.BasicPublish(exchange: _brokerName,
                             routingKey: eventName,
                             basicProperties: null,
                             body: body);
    }
}
```

Листинг 8 – Метод отправки сообщений в очередь.

В методе в листинге 8 создается отдельный поток для отправки сообщения, после чего он запускается с аргументов @event. Такой способ запуска гарантирует моментальный возврат в вызвавшую отправку функцию, что позволяет не прерывать работу для отправки статистики. Внутри потока выполняется небольшая функция, которая проверяет соединение, записывает в объект события время его возникновения (которое считается как время попадания в функцию, после чего отправляет объект брокеру. При поступлении нового сообщения в очередь, брокер уведомляет подписавшихся на получение этих событий получателей. Сервис статистики, получив уведомление, вызывает метод обработки событий, приведенный в листинге 9:

```
private async Task ReceiveMessage(string queueName)
{
    var factory = new ConnectionFactory() { HostName = "localhost" };
    using (var connection = factory.CreateConnection())
    {
        using (var channel = connection.CreateModel())
        {
            channel.QueueDeclare("queueName", false, false, false, null);

            var consumer = new QueueingBasicConsumer(channel);
            channel.BasicConsume("queueName", true, consumer);

            while (true)
            {
                var ea = (BasicDeliverEventArgs)consumer.Queue.Dequeue();

                var body = ea.Body;
                var message = Encoding.UTF8.GetString(body);
            }
        }
    }
}
```

```
}
```

Листинг 9 – Метод обработки событий.

Реализация масштабируемости

Несмотря на то, что сервисная архитектура и стиль REST позволяют легко горизонтально масштабировать систему, для автоматизации процесса необходимо приложить усилия. В разрабатываемой системе для этого выделен отдельный сервис-координатор, отвечающий за балансировку нагрузки между сервисами. Принцип работы координатора прост: анализируя полученные от сервиса статистики данные о нагрузке на сервисы, а также время доступа одного сервиса к другому, координатор меняет связи между сервисами для равномерного распределения нагрузки на систему.

Для сбора статистики запросов, а также изменения связей в каждом сервисе, используются объекты промежуточного слоя (англ. Middleware). Они располагаются в конвейере обработки запросов ASP.Net Core, позволяя анализировать полученные данные до их попадания в основное приложение. Пример промежуточного слоя для сбора статистики запросов приведен в листинге 10.

```
public class RequestStatisticsMiddleware
{
    private readonly RequestDelegate next; private readonly IEventBus eventBus;

    public RequestStatisticsMiddleware(RequestDelegate next, IEventBus eventBus)
    {
        this.next = next; this.eventBus = eventBus;
    }
    public virtual async Task Invoke(HttpContext context)
    {
        var connection = context.Connection; eventBus.SendRequestEvent(
            $"{connection.LocalIpAddress}:{connection.LocalPort}",
            $"{connection.RemoteIpAddress}:{connection.RemotePort}",
            context.Request.Path.ToString(),
            RequestType.Gateway);

        await next(context);
    }
}
```

Листинг 10 – Пример промежуточного слоя сбора статистики.

Как показано в листинге 10, классы, относящиеся к промежуточному слою, не наследуются от каких-либо специализированных классов. Требованиями к ним являются наличие объекта класса RequestDelegate в данном классе и метод Invoke(HttpContext). Делегат RequestDelegate отвечает за метод Invoke промежуточного слоя, стоящего следующим в конвейере обработки. Таким образом, если какой-либо слой не собирается препятствовать запросу и пропускает его дальше, он вызывает метод делегата RequestDelegate. В методе Invoke данного промежуточного слоя отправляется сообщение в очередь RabbitMQ, откуда оно будет извлечено сервисом статистики. Важно отметить, что методы Invoke напрямую влияют на время отклика, и потому нежелательно размещать в них большие синхронные операции.

Аналогичным образом реализуется промежуточный слой координатора, встраиваемый во все сервисы. Он анализирует пакеты на предмет наличия в них заранее определенного заголовка и, при его наличии, интерпретирует значение заголовка как одну из допустимых команд. Достаточный для реализации координатора набор состоит из команды установки нового адреса и команды замера времени доступа от одного сервиса к другому. Данные о нагрузке на сервисы поступают от сервиса статистики, собирающего информацию о запросах при помощи промежуточного слоя из листинга 5.1. Анализируя эти данные, а также обладая информацией о

времени отправки запросов между сервисами, координатор динамически меняет связи сервисов для снижения нагрузки и времени отклика системы в целом.

Использование координатора позволяет отказаться от записи необходимых адресов в конфигурационный файл каждого сервиса. Вместо этого при запуске координатора происходит инициализация всех известных координатору сервисов. В соответствии с заданными зависимостями каждому сервису отправляются необходимые для его работы адреса. При добавлении администратором новых сервисов они инициализируются, а периодически проходит балансировка на основании ранее описанных данных.

Реализация отказоустойчивости

Отказоустойчивость гарантирует работоспособность системы, несмотря на внутренние неполадки в системе. Существуют разные способы сохранения функционирования, например, деградация функциональности, полный откат операции, очередь сообщений и другие.

Деградация функциональности

Деградация функциональности - принцип сохранения работоспособности всего приложения при частичной потере функциональности.

В данном приложении отказоустойчивость распространяется на недоступность сервисов. То есть в том случае, когда один из сервисов не способен дать ответ на запрос из-за недоступности, необходимо вывести пользователю информацию, которую удалось обработать без участия недоступного сервиса, и сообщение об ошибке.

Деградацию, как правило, используют в GET-запросах, так как это не влечет никакой потери пользовательских данных.

Откат

Деградация функциональности не решает проблему в том случае, когда запрос влечет изменение данных (например, PUT, DELETE). Потому что операция изменения данных является единой транзакцией, а значит, частичная работа над ними невозможна.

Откат операции означает, что метод, который агрегирует запрос пользователя в запросы к сервисам, перед внесением изменений должен сохранять снимок исходного состояния изменяемых данных. Это необходимо для того, чтобы в процессе выполнения запроса при возникновении ошибки система могла стереть только что внесенные изменения, вернуть исходное состояние и выдать пользователю ошибку о невозможности обработки запроса.

Очередь сообщений

Суть данного метода отказоустойчивости заключается в том, что в случае, если сервис недоступен и невозможно выполнить запрос, изменяющий данные, искомый запрос становится в очередь на выполнение.

Когда задача выделяется из очереди, происходит попытка ее выполнения. Если попытка оказалась успешной, задача считается выполненной, и их очередь обрабатывает следующую задачу. При неудаче задача снова ставится в очередь.

Очередь организована классом, имеющим методы постановки задачи в очередь и обработки элемента в очереди. Реализация такого процесса приведена в листинге 11:

```
public void ProcessQueues()
{
    T item = default(T);

    while (queue.TryPeek(out item))
    {
        var result = Task.Run(() => item);
```

```

        if (result.IsCompletedSuccessfully && result.Status ==
TaskStatus.Canceled)
        {
            queue.TryDequeue(out item);
        }
        else
        {
            queue.Enqueue(item);
        }
    }
}

```

Листинг 11. Обработка элемента очереди.

Способ постановки запроса в очередь приведен в листинге 12.

```

//add to queue
var task = new Task(() => Delete(id));
taskQueue.Enqueue(task);

```

Листинг 12. Пример постановки запроса в очередь

Реализация авторизации и регистрации в системе

Для реализации регистрации и авторизации в приложении прежде всего необходимо создать сервис работы с хранилищем данных пользователя. В разрабатываемой системе за данный функционал отвечает сервис аккаунтов. В основе его реализации лежит модель под названием User, которая содержит следующие поля:

- Email
- Имя
- Пароль
- Роль

В контроллере сервиса необходимо реализовать методы работы с данными пользователя, например, сохранения информации в базу данных при регистрации или проверки данных пользователя при попытке авторизации и другие сопутствующие методы.

После реализации сервиса аккаунтов необходимо внедрить функционал регистрации и авторизации в главный сервис, во-первых, чтобы в системе мог работать только авторизованный пользователь, во-вторых, для проверки актуальности сессии клиента, работающего с приложением. В связи с тем, что проверка авторизации пользователя должна производиться перед выполнением любого запроса, требуется встроить данную проверку в процесс обработки запросов при помощи добавления компонентов промежуточного слоя.

Компонент промежуточного слоя - программное обеспечение, выстраиваемое в виде конвейера приложения ASP.NET для обработки запросов и откликов. Конвейер запросов ASP.NET Core состоит из последовательности делегатов запроса, вызываемых один за другим. Каждый из делегатов может выполнять операции до и после следующего делегата. Делегат также может принять решение не передавать запрос следующему делегату, что называется замыканием конвейера запросов. Добавление компонентов промежуточного слоя происходит в методе Configure файла Startup.cs проекта, при этом порядок, в котором компоненты добавляются в метод, определяет порядок их вызова при запросах и обратный порядок для отклика.

Содержимое класса Startup.cs приведено в листинге 13.

```

public class Startup
{
    public Startup(IConfiguration configuration)
    {

```

```

        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the
    container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<CookiePolicyOptions>(options =>
        {
            // This lambda determines whether user consent for non-essential cookies is
            needed for a given request.
            options.CheckConsentNeeded = context => false;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });
        string connection = Configuration.GetConnectionString("DefaultConnection");
        // добавляем контекст MobileContext в качестве сервиса в приложение
        services.AddDbContext<ApplicationContext>(options =>
            options.UseSqlServer(connection));
        services.AddMvc();
        services.AddDefaultIdentity<IdentityUser>()
            .AddDefaultUI(UIFramework.Bootstrap4)
            .AddEntityFrameworkStores<ApplicationContext>();
        services.AddAuthentication(options =>
        {
            //options.DefaultScheme = "Cookies";
            options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
            options.DefaultChallengeScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
            options.DefaultSignInScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
            options.DefaultSignOutScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
            options.DefaultAuthenticateScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;
            options.DefaultForbidScheme =
            CookieAuthenticationDefaults.AuthenticationScheme;

        })
        .AddCookie(options => //CookieAuthenticationOptions
        {
            options.Cookie.HttpOnly = false;
            options.LoginPath = new PathString("/Account/Login");
            options.LogoutPath = new PathString("/Account/Logout");
            options.AccessDeniedPath = new PathString("/Account/AccessDenied");
            options.ExpireTimeSpan = TimeSpan.FromMinutes(30);

            options.Cookie.Name = "MyCookies";

        })
        .AddFacebook(options =>
        {
            options.AppId = "460488657831185";
            options.AppSecret = "ee51c6282e53fb6adef88877e0735286";
            options.CallbackPath = new PathString("/Account/LoginFB");
            options.Fields.Add("Name");
            options.Fields.Add("Email");
        });

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
    }

```

```

// This method gets called by the runtime. Use this method to configure the HTTP
request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Article/Error");
        // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseAuthentication();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Article}/{action=Filtered}/{id?}");
    });
}
}

```

Листинг 13. Класс Startup.cs.

Представленный в листинге 14 метод осуществляет авторизацию пользователя, работающего с приложением. Если переданные данные не содержат верной информации аутентификации, то необходимо вернуть ответ, содержащий текст ошибки.

Фильтр `ValidateAntiforgeryToken` предназначен для противодействия подделке межсайтовых запросов, производя верификацию токенов при обращении к методу действия. Наиболее частым случаем является применение данного фильтра к методам, отвечающим за авторизацию.

```

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginModel model)
{
    if (ModelState.IsValid)
    {
        User user = await db.Users
            .Include(u => u.Role)
            .FirstOrDefaultAsync(u => u.Email == model.Email && u.Password ==
model.Password);

        if (user != null)
        {
            await Authenticate(user); // аутентификация

            return RedirectToAction("Index", "Home");
        }
        ModelState.AddModelError("", "Некорректные логин и(или) пароль");
    }
    return View(model);
}

```

Листинг 14 – Код метода Login.

Для регистрации пользователя был написан метод `Register`, который учитывает роль

пользователя (читатель или автор). В случае ввода некорректных данных выводится сообщение об ошибке. Код данного метода представлен в листинге 16.

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Register(RegisterModel model, string author, string
reader)
{
    string auth = "reader";
    if (!string.IsNullOrEmpty(author))
    {
        auth = "author";
    }
    if (!string.IsNullOrEmpty(reader))
    {
        auth = "reader";
    }
    if (ModelState.IsValid)
    {
        User user = await db.Users.FirstOrDefaultAsync(u => u.Email == model.Email);
        if (user == null)
        {
            user = new User
            {
                Email = model.Email,
                Name = model.Name,
                Password = model.Password
            };
            Role userRole = await db.Roles.FirstOrDefaultAsync(r => r.Name == auth);
            if (userRole != null)
                user.Role = userRole;
            // добавляем пользователя в бд
            db.Users.Add(user);
            await db.SaveChangesAsync();

            await Authenticate(user); // аутентификация

            return RedirectToAction("Index", "Home");
        }
        else
            ModelState.AddModelError("", "Некорректные логин и(или) пароль");
    }
    return View(model);
}
```

Листинг 15 – Код метода Register.

Метод Authenticate (листинг 17) отвечает за аутентификацию пользователя. Класс ClaimsIdentity — это конкретная реализация удостоверения на основе утверждений (удостоверение описывающее коллекцию утверждений). Утверждения — это заявления, которые описывает свойство или некоторые другие качества этого объекта сущности. Такая сущность называется субъектом утверждения. Утверждение представлено классом Claim. Метод SignInAsync осуществляет авторизацию пользователя и записывает данные о нем в соответствующих Cookie.

```
private async Task Authenticate(User user)
{
    // создаем один claim
    var claims = new List<Claim>
    {
        new Claim(ClaimsIdentity.DefaultNameClaimType, user.Name),
        new Claim(ClaimsIdentity.DefaultRoleClaimType, user.Role?.Name)
    }
}
```

```

    };
    // создаем объект ClaimsIdentity
    ClaimsIdentity id = new ClaimsIdentity(claims,
CookieAuthenticationDefaults.AuthenticationScheme);
    var authProperties = new AuthenticationProperties();
    // установка аутентификационных куки

    await HttpContext.SignInAsync(
        CookieAuthenticationDefaults.AuthenticationScheme,
        new ClaimsPrincipal(id),
        authProperties);

}

```

Листинг 16 – Код метода Authenticate.

Очень важно вставить данные команды до добавления статических файлов, чтобы проверка авторизации проводилась до начала основной обработки запросов пользователя.

Реализация авторизации в системе через социальные сети

На сегодняшний день многие пользователи имеют множество аккаунтов в различных приложениях, в связи с этим иногда гораздо удобнее авторизоваться в новом приложении при помощи уже существующего аккаунта другого сервиса, нежели заново регистрироваться. Возможность такой авторизации предоставляет фреймворк OAuth 2, который позволяет приложениям осуществлять ограниченный доступ к пользовательским аккаунтам на HTTP-сервисах, например, аккаунтам в популярных социальных сетях. Он работает по принципу делегирования аутентификации пользователя сервису, на котором находится аккаунт пользователя, позволяя стороннему приложению получать доступ к аккаунту пользователя. В данном разделе подробно описана реализация авторизация пользователя через Facebook в разрабатываемом приложении.

Для аутентификации через любую социальную сеть, необходимо зарегистрировать свое приложение в консоли разработчика выбранного провайдера.

Для рассматриваемого приложения была выбрана платформа Facebook. Как зарегистрировать свое приложение подробно описано в источнике [11]. В консоли разработчика настроен адрес обратного вызова локального IdentityServer путем добавления пути /signin-facebook к базовому адресу. После регистрации выдается AppId и AppSecret для разрабатываемого приложения.

Использование токенов безопасности

Чтобы предоставлять доступ только авторизованным пользователям на некоторый период времени, используются токены. Для реализации авторизации через OAuth рассматриваются 2 вида токенов: access-токен (для доступа пользователя к функционалу приложения) и refresh-токен (передается серверу по истечению сессии для выдачи свежего access-токена).

При авторизации клиентское приложение Client обращается к серверу аутентификации AuthServer и получает access-токен, который затем использует в качестве Bearer- токена для вызова главного сервиса приложения.

Bearer-токен обладает свойством, что любой, кто им владеет, может использовать токен любым способом. Доказательства того, что предъявитель токена является владельцем - не требуется. Создаваемый access-токен имеет три поля: идентификатор, владелец и время истечения. Хранилище токенов организовано так же, как и другие таблицы базы данных приложения. Работа с токенами происходит в контроллере сервиса авторизации.

Работа с Facebook.

Работа с авторизацией через социальную сеть происходит полностью на стороне главного сервиса.

После регистрации приложения в консоли разработчика, сначала в классе Startup в методе ConfigureServices нужно прописать обработчик аутентификации Facebook, представленный на листинге 18.

```
services.AddAuthentication().AddFacebook(facebookOptions =>
{
    facebookOptions.AppId = Configuration["Authentication:Facebook:AppId"];
    facebookOptions.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
});
```

Листинг 17. Обработчик аутентификации Facebook

Здесь AppId и AppSecret - это те Id и Secret, который выдал Facebook при регистрации приложения.

В контроллере необходим метод-обработчик кнопки авторизации через Facebook. Его ключевым атрибутом является указание в “шапке” схемы Google-аутентификации, которая задана в Startup.cs. Пример приведен на листинге 19.

```
public IActionResult LoginFB()
{
    var authProperties = new AuthenticationProperties
    {
        RedirectUri = Url.Action("LoginCallback", "Account"),
    };
    return Challenge(authProperties, FacebookDefaults.AuthenticationScheme);
}
```

Листинг 19. Метод контроллера по обработке Facebook-авторизации

Реализация основных функциональных требований к системе

В этом разделе приведена подробная реализация ответа на запрос о получении каталога статей, как пример одной из основных функциональных требований. В ответ на внешний запрос, происходящий, например, по нажатию на кнопку в интерфейсе, должна приходить страница, показывающая статьи, ранжированные по катгории. Метод получения каталога новстей из сервиса статей приведен в листинге 20.

```
[Authorize]
public async Task<ActionResult> Filtered(int? category, int page, int? author,
SortState sortOrder = SortState.MarkDesc)
{
    int pageSize = 3;
    if (page == 0)
        page = 1;
    //фильтрация
    IQueryable<Article> articles = db.Articles.Include(x => x.Category);

    if (category != null && category != 0)
    {
```

```

        if (author != null && author != 0)
        {
            articles = articles.Where(p => p.CategoryId == category && p.AuthorId ==
author);
        }
        else
            articles = articles.Where(p => p.CategoryId == category);
    }
    else if (author != null && author != 0)
        articles = articles.Where( p => p.AuthorId == author);

    // сортировка
    switch (sortOrder)
    {
        case SortState.NameDesc:
            articles = articles.OrderByDescending(s => s.Name);
            break;
        case SortState.MarkAsc:
            articles = articles.OrderBy(s => s.Mark);
            break;
        case SortState.MarkDesc:
            articles = articles.OrderByDescending(s => s.Mark);
            break;
        case SortState.DateAsc:
            articles = articles.OrderBy(s => s.Date);
            break;
        case SortState.DateDesc:
            articles = articles.OrderByDescending(s => s.Date);
            break;
        case SortState.AuthorAsc:
            articles = articles.OrderBy(s => s.AuthorName);
            break;
        case SortState.AuthorDesc:
            articles = articles.OrderByDescending(s => s.AuthorName);
            break;
        case SortState.CategoryAsc:
            articles = articles.OrderBy(s => s.Category.Name);
            break;
        case SortState.CategoryDesc:
            articles = articles.OrderByDescending(s => s.Category.Name);
            break;
        default:
            articles = articles.OrderBy(s => s.Name);
            break;
    }

    // пагинация
    var count = await articles.CountAsync();
    var items = await articles.Skip((page - 1) *
pageSize).Take(pageSize).ToListAsync();

    // формируем модель представления
    IndexViewModel viewModel = new IndexViewModel
    {
        PageViewModel = new PageViewModel(count, page, pageSize),
        SortViewModel = new SortViewModel(sortOrder),
        FilterViewModel = new FilterViewModel(db.Categories.ToList(), category,
db.Users.Where(u => u.RoleId == 3).ToList(), author),
        Articles = items
    };
    return View(viewModel);
}

```

Листинг 20 – Метод получения статей по категориям.

Для пагинации страницы используется класс PageViewModel. Он представлен на листинге


```

public class PageViewModel
{
    public int PageNumber { get; private set; }
    public int TotalPages { get; private set; }

    public PageViewModel(int count, int pageNumber, int pageSize)
    {
        PageNumber = pageNumber;
        TotalPages = (int)Math.Ceiling(count / (double)pageSize);
    }

    public bool HasPreviousPage
    {
        get
        {
            return (PageNumber > 1);
        }
    }

    public bool HasNextPage
    {
        get
        {
            return (PageNumber < TotalPages);
        }
    }
}

```

Листинг 21 – Класс PageViewModel.

```
@model WebNews.ViewModels.IndexViewModel
```

```
@{
    ViewBag.Title = "Каталог статей";
}
```

```
<h2>Каталог статей</h2>
```

```
<form method="get">
    <div class="form-inline form-group">
```

```
        <label class="control-label px-2">Категория:</label>
```

```
        @Html.DropDownList("category", Model.FilterViewModel.Categories as SelectList,
            htmlAttributes: new { @class = "form-control" })
```

```
        <label class="control-label px-2">Автор:</label>
```

```
        @Html.DropDownList("author", Model.FilterViewModel.Authors as SelectList,
            htmlAttributes: new { @class = "form-control" })
```

```
        <input type="submit" value="Показать" class="btn btn-default" />
```

```
    </div>
```

```
<table class="table">
```

```
    <tr>
```

```
        <th>
```

```
            <a asp-action="Filtered"
```

```
                asp-route-sortOrder="@Model.SortViewModel.NameSort)"
```

```
                asp-route-category="@Model.FilterViewModel.SelectedCategory)"
```

```
                asp-route-author="@Model.FilterViewModel.SelectedAuthor)">Название</a>
```

```
        </th>
```

```
    <th>
```

```
        <a asp-action="Filtered"
```

```
                asp-route-sortOrder="@Model.SortViewModel.AuthorSort)"
```

```
                asp-route-category="@Model.FilterViewModel.SelectedCategory)"
```

```

        asp-route-author="@Model.FilterViewModel.SelectedAuthor">Автор</a>
    </th>
    <th>
        <a asp-action="Filtered"
            asp-route-sortOrder="@Model.SortViewModel.DateSort"
            asp-route-category="@Model.FilterViewModel.SelectedCategory"
            asp-route-author="@Model.FilterViewModel.SelectedAuthor">Дата</a>
    </th>
    <th>
        <a asp-action="Filtered" asp-route-sortOrder="@Model.SortViewModel.MarkSort"
            asp-route-category="@Model.FilterViewModel.SelectedCategory"
            asp-route-author="@Model.FilterViewModel.SelectedAuthor">Оценка</a>
    </th>
    <th>
        <a asp-action="Filtered" asp-route-
sortOrder="@Model.SortViewModel.CategorySort"
            asp-route-category="@Model.FilterViewModel.SelectedCategory"
            asp-route-author="@Model.FilterViewModel.SelectedAuthor">Категория</a>
    </th>
</tr>
@foreach (Article a in Model.Articles)
{
    <tr>
        <td>
            <a asp-action="Details"
                asp-controller="Article" asp-route-id="@a.Id"
                class="btn btn-default btn">
                @a.Name
            </a>
        </td>
        <td>@a.AuthorName</td>
        <td>@a.Date.ToShortDateString()</td>
        <td>@a.Mark</td>
        <td>@a.Category.Name</td>
    </tr>
}
</table>
@if (Model.PageViewModel.HasPreviousPage)
{
    <a asp-action="Filtered"
        asp-route-page="@Model.PageViewModel.PageNumber - 1"
        asp-route-category="@Model.FilterViewModel.SelectedCategory"
        asp-route-author="@Model.FilterViewModel.SelectedAuthor"
        asp-route-sortOrder="@Model.SortViewModel.Current"
        class="btn btn-default btn">
        <i class="glyphicon glyphicon-chevron-left"></i>
        Назад
    </a>
}
@if (Model.PageViewModel.HasNextPage)
{
    <a asp-action="Filtered"
        asp-route-page="@Model.PageViewModel.PageNumber + 1"
        asp-route-category="@Model.FilterViewModel.SelectedCategory"
        asp-route-author="@Model.FilterViewModel.SelectedAuthor"
        asp-route-sortOrder="@Model.SortViewModel.Current"
        class="btn btn-default btn">
        Вперед
        <i class="glyphicon glyphicon-chevron-right"></i>
    </a>
}
</form>

```

Листинг 22 – Шаблон страницы каталога статей.

Как видно из листинга 22, шаблон страницы просмотров использует модель

IndexViewModel и включает в себя инструкции на языке Razor, создающие для каждого элемента списка строку в таблице. Таблица организуется при помощи элементов `<div>` с соответствующими классами Bootstrap. Это позволяет гибко настраивать внешний вид таблицы при помощи CSS.

Руководство пользователя

Данное руководство описывает варианты работы пользователя в разработанной системе.

1. Авторизация и регистрация

В связи с тем, что в приложении может работать только авторизованный пользователь, единственная страница, которая доступна в приложении до авторизации и является стартовой - страница авторизации.

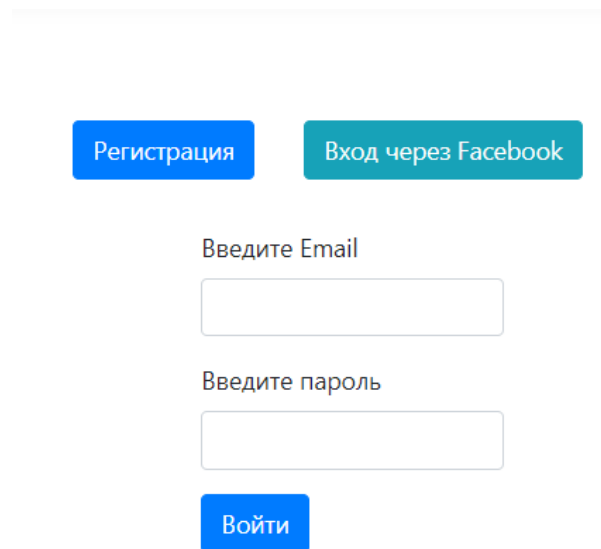


Рисунок 13. Страница авторизации приложения.

Для начала работы в системе в открывшейся форме необходимо заполнить данные пользователя: логин и пароль. Предлагаемая форма может использоваться как для регистрации нового пользователя, так и для входа уже существующего в зависимости от выбранной опции после заполнения данных.

При успешном завершении операции будет осуществлен переход на страницу профиля авторизованного пользователя.

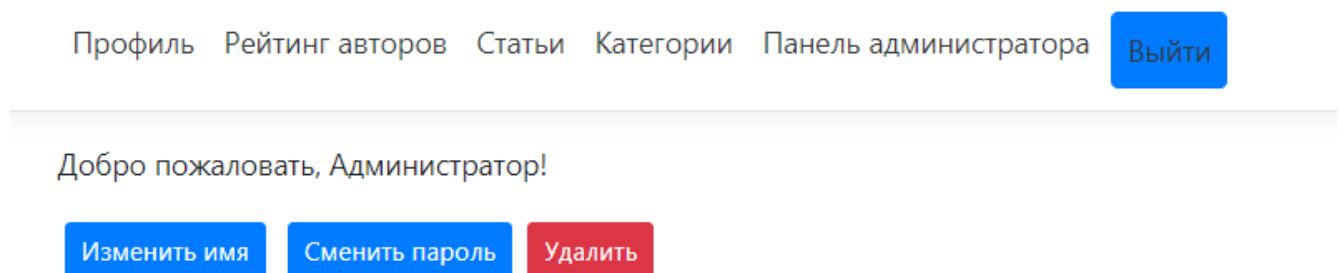


Рисунок 14. Переход на страницу профиля после успешной авторизации.

Вход в систему также может быть выполнен через социальную сеть. Для этого необходимо нажать на кнопку «Вход через Facebook» и ввести во всплывающее окно данные своего аккаунта. Затем нужно нажать на кнопку «Продолжить».



Приложение «**WebNews**» получит:
имя и фото профиля и электронный адрес.

 [Редактировать](#)

Продолжить как Polina

Отмена

 Приложение не сможет делать публикации на Facebook

[Политика конфиденциальности](#)

Рисунок 15. Всплывающее окно Facebook.

При успешном завершении операции будет также осуществлен переход на страницу профиля авторизованного пользователя.

2. Вкладка «Профиль»

На данной странице отображается общая информация о работе в системе, а также есть возможность изменить данные профиля пользователя либо удалить профиль из системы. При выборе опции изменения данных откроется страница с формой для ввода новых данных пользователя, например, имени.

Редактирование профиля для пользователя mail@mail.ru

Email

Имя

Сохранить

Рисунок 16. Страница изменения данных пользователя.

После окончания редактирования данных необходимо завершить операцию нажатием кнопки «Изменить». После чего пользователь будет перенаправлен на страницу своего профиля.

При выборе опции изменения пароля пользователь будет переведен на страницу смены пароля.

Изменение пароля для пользователя mail@mail.ru

Старый пароль

Новый пароль

Сохранить

Рисунок 17. Страница смены пароля.

3. Вкладка «Избранное»

На данной странице осуществляется просмотр оцененных статей пользователя. Также на ней отображаются оценки статей всех читателей.

Профиль Рейтинги авторов Статьи Избранное Выйти

Избранное

Название статьи	Автор статьи	Дата размещения	Категория	Оценка
Американская журналистика становится более субъективной	Автор1	04.06.2019	Политика	1
Из-за глобального потепления цивилизация придет к упадку к 2050 году	Автор1	05.06.2019	Наука	1
Искусственный интеллект DeepMind победил людей в кооперативной игре Quake III Arena	Автор2	05.06.2019	Наука	3

Рисунок 18. Страница оцененных статей пользователя.

4. Вкладка «Статьи»

Данные разделы сайта могут быть использованы для ознакомления с существующими категориями статей. На странице «Статьи» отображаются все доступные статьи, ранжированные по автору и по категории.

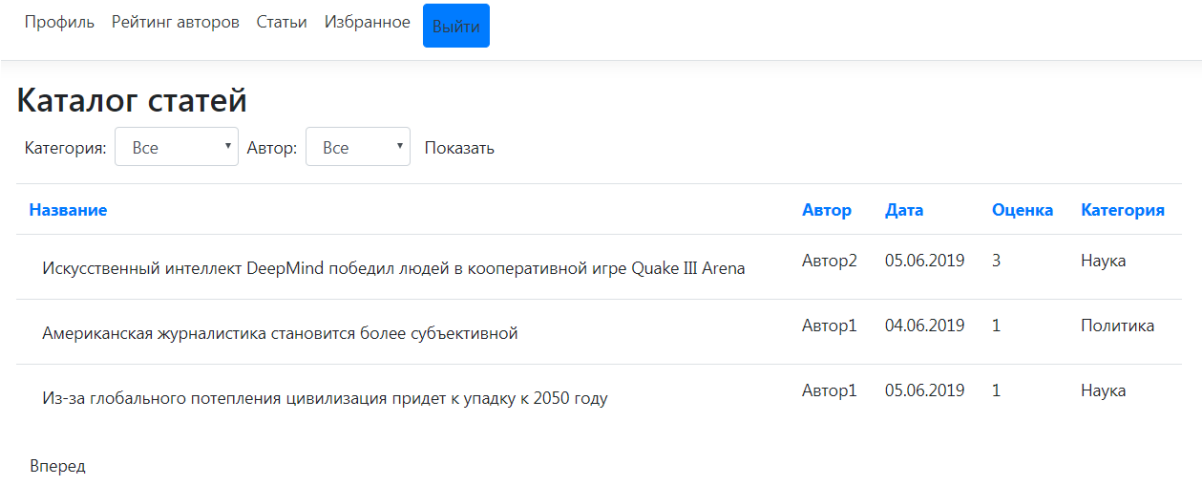


Рисунок 19. Страница каталога статей.

5. Вкладка “Рейтинг авторов”

На данной странице осуществляется просмотр рейтинга авторов, сформированного на основе оценок статьёй пользователями, и переход к подробному просмотру статей выбранного автора.

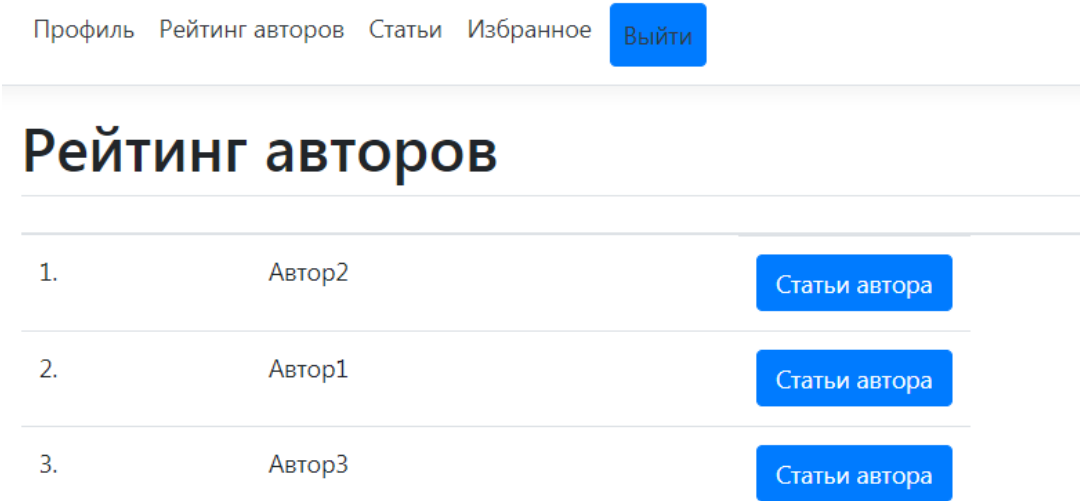


Рисунок 21. Страница с рейтингом авторов.

6. Вкладка “Мои статьи”

Данная страница содержит список статей авторизованного автора. Также в разделе доступен функционал удаления и изменения публикаций.

Профиль	Рейтинг авторов	Статьи	Мои статьи	Добавить статью	Выйти
Мои статьи					
Название статьи	Дата размещения	Категория	Оценка		
Искусственный интеллект DeepMind победил людей в кооперативной игре Quake III Arena	05.06.2019	Наука	3	Изменить	Удалить

Рисунок 22. Список статей автора.

7. Вкладка «Добавить статью»

Данная страница содержит форму для добавления статьи автором.

Добавление статьи

Статья

Название

Текст статьи

Категория

Политика

Добавить

Обратно к списку статей

Рисунок 23. Страница для добавления статьи.

8. Выход из системы

Для выхода из системы необходимо нажать кнопку “Выйти” в правом углу интерфейса приложения.

Руководство администратора

В руководстве пользователя описаны основные варианты работы в системе при авторизации через аккаунт, которому присвоена роль «Пользователь». В данном руководстве приводится описание функционала для аккаунта с ролью «Администратор». Администратор портала может использовать как стандартный функционал приложения, доступный всем зарегистрированным

пользователям, так и функционал администрирования приложения. Основные функции администратора – распределение нагрузки между сервисами системы и управление категориями публикаций. Для расширения функционала администратора или уточнения работы существующего, можно обратиться к репозиторию [17], в котором хранится программный код разрабатываемой системы.

1. Управление узлами системы

В разделе «Панель администратора» отображаются статистические данные о медиане времени отклика каждого из сервисов системы.

Управление сервисами

Добавить сервис

Название	Адрес	Медиана времени отклика	
AuthServer	https://localhost:5001/	00:00:00.1500000	Удалить
Articles	https://localhost:5005/	00:00:00.1200000	Удалить
Statistics	https://localhost:5003/	00:00:00.0200000	Удалить
Accounts	https://localhost:5007/	00:00:00.0110000	Удалить
Gateway	https://localhost:5010/	00:00:00.2000000	Удалить

Рисунок 24. Информация о нагрузке сервисов и управление ими.

Согласно требованиям к функциональным характеристикам системы данный параметр для каждого из сервисов не должен превышать 700мс. Таким образом администратор, заметив значительные задержки во времени отклика сервиса, должен воспользоваться опцией добавления сервиса к системе. Для этого необходимо нажать кнопку «Добавить сервис», после чего будет осуществлен переход на страницу добавления сервиса.

2. Добавление категорий статей

Помимо распределения нагрузки между сервисами администратор также занимается управлением категориями публикаций. Данный функционал также доступен в разделе «Панель администратора».

Категории

[Добавить](#)

Name			
Политика	Изменить	Подробнее	Удалить
Наука	Изменить	Подробнее	Удалить

Рисунок 25. Раздел управления категориями статей.

Заключение

В ходе выполнения практикума по РСОИ был разработан портал просмотра ленты новостей, формирующейся на основе подписок пользователя. Перед началом разработки сформулировано техническое задание с общими и функциональными требованиями, а также требованиями к надежности и документации. Помимо требований к системе в техническом задании приведена топология системы, на основе которой разработаны требования к подсистемам. В конструкторском разделе описано проектирование сервисов по отдельности и системы в целом, проработаны сценарии функционирования и представлена архитектура сервисов. В технологическом разделе описаны основные принципы реализации ключевого функционала сервисов. Данная работа также включает в себя реализацию описанного проекта согласно разработанному техническому заданию. Реализованный функционал отлажен и протестирован.

РЕЗУЛЬТАТЫ ПРАКТИКИ

По результатам прохождения практики планируется формирование компетенций, предусмотренных основной профессиональной образовательной программой на СУОС МГТУ им. Н.Э. Баумана по направлению подготовки магистра 09.04.04. «Программная инженерия».

Формулировка компетенции
Собственные общекультурные компетенции
Способностью заниматься научными исследованиями
Способностью использовать на практике умения и навыки в организации исследовательских и проектных работ, в управлении коллективом
Способностью к профессиональной эксплуатации современного оборудования и приборов (в соответствии с целями магистерской программы)
Способностью оформлять отчеты о проведенной научно-исследовательской работе и подготавливать публикации по результатам исследования
Собственные общепрофессиональные компетенции
Способностью применять методы фундаментальных и общетехнических наук для анализа и моделирования ключевых объектов различного функционального назначения
Способностью воспринимать математические, естественнонаучные, социально-экономические и профессиональные знания, умением самостоятельно приобретать, развивать и применять их для решения нестандартных задач, в том числе в новой или незнакомой среде и в междисциплинарном контексте
Владением методами и средствами получения, хранения, переработки и трансляции информации посредством современных компьютерных технологий, в том числе, в глобальных компьютерных сетях
Способностью анализировать профессиональную информацию, выделять в ней главное, структурировать, оформлять и представлять в виде аналитических обзоров с обоснованными выводами и рекомендациями
Собственные профессиональные компетенции
Способностью выполнить постановку новых задач анализа и синтеза сложных проектных решений
Знанием методов научных исследований и владением навыками их проведения
Способностью проектировать распределенные информационные системы, их компоненты и протоколы их взаимодействия
Владением навыками программной реализации распределенных информационных систем

СПИСОК ЛИТЕРАТУРЫ

1. Исаев Г.Н. Проектирование информационных систем. – М.: Издательство “Омега-Л”, 2013. – 424 с.
2. Вишневская Т.И., Романова Т.Н. Технология программирования: Мет. указания к лабораторному практикуму. - Ч. 1. – М: Изд-во МГТУ им. Н.Э. Баумана, 2007, 59с.
3. Вишневская Т.И., Романова Т.Н. Технология программирования: Мет. указания к лабораторному практикуму. - Ч. 2. – М: Изд-во МГТУ им. Н.Э. Баумана, 2010, 46с.
4. Вишневская Т.И., Романова Т.Н. Технология программирования: Мет. указания к лабораторному практикуму. - Ч. 3. – ФГБОУ ВПО МГТУ им. Н.Э. Баумана, 2012 ,№0321203886. [Электронный ресурс] Режим доступа: URL:<http://catalog.inforeg.ru/Inet/GetEzineByID/293408>
5. Техническое задание. Требования к содержанию и оформлению (ГОСТ 19.201-78). [Электронный ресурс] Режим доступа: URL: <http://fmi.asf.ru/library/book/Gost/19201-78.html>.
6. Техническое задание. Требования к содержанию и оформлению (ГОСТ 19.201-78). [Электронный ресурс] Режим доступа: URL: <http://fmi.asf.ru/library/book/Gost/19201-78.html>.
7. Вишневская Т.И., Романова Т.Н. Методология программной инженерии: Мет. указания к выполнению лабораторных работ М.: Изд-во МГТУ им. Н.Э. Баумана, 2017.-58с.
8. Научно-исследовательский Центр CALS - Методология функционального моделирования IDEF0: ИПК Издательство стандартов, 2000 – 75 с.
9. Богданов А., Корхов В., Мареев В., Архитектуры и топологии многопроцессорных вычислительных систем. Издательство Интуит.Ру, 2013, 176 с.
10. Ньюмэн С. Создание микросервисов - Издательство Питер, 2016 г. 392 с.
11. Amindsen M., Rubi S. RESTful Web APIs: Services for a Changing World, O'Reilly, 2013, 300 p.
12. Announcing .NET Core 2.0 [Электронный ресурс] Режим доступа: URL: <https://blogs.msdn.microsoft.com/dotnet/2017/08/14/announcing-net-core-2-0>.
13. Создание веб-приложения с ASP.NET Core MVC на Windows с помощью Visual Studio [Электронный ресурс] Режим доступа: URL: <https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/first-mvc-app/?view=aspnetcore-2.0>
14. Entity Framework Core Quick Overview [Электронный ресурс] Режим доступа: URL: <https://docs.microsoft.com/en-us/ef/core/>
15. Усачев В., Рагулин П. Концептуальная модель масштабируемого сервиса социальной сети, Молодой учёный №12 (116) июнь-2 2016 г.
16. <https://github.com/polinaanri/webnews>