**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет прикладной математики и информатики**

**Кафедра информационных систем управления**

ЖДАНОВИЧ

Павел Александрович

**РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ НА ОСНОВЕ ФРЕЙМВОРКА BLAZOR**

Дипломная работа

**Руководитель:**

Конах Валентина Владимировна,

старший преподаватель каф. ИСУ

Минск

2019

**РЕФЕРАТ**

**РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ НА ОСНОВЕ ФРЕЙМВОРКА BLAZOR**

Дипломная работа, 37 с., 12 рис., 4 источника

**Ключевые слова:** Web-приложение, .NET, Blazor, производительность.

**Цель работы:** изучение и анализ характеристик фреймворка Blazor, сравнение с другими технологиями, создания прототипа Web-приложений для демонстрации возможностей.

**Методы исследования:** анализ документации, тестирование производительности, статистический анализ

**Результаты работы:** проанализирована производительность фреймворка Blazor, разработано демонстрационное приложения, разработана библиотека Blazor-компонентов.

**Область применения:** разработка Web-приложений

**ОГЛАВЛЕНИЕ**

**ВВЕДЕНИЕ 4**

**ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ И ОБЗОР ТЕХНОЛОГИЙ 5**

1.1 Постановка задачи 5

1.2 Существующие технологии, применяемые в Web-клиентах 5

1.3 Платформа .NET 9

1.4 Blazor 10

**ГЛАВА 2. ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ И РЕАЛИЗАЦИЯ ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ 13**

2.1Демонстрационное приложение 13

2.2Сравнительный анализ скорости вычислений 14

2.3Сравнительный анализ накладных расходов при вызовах между средами выполнения 17

2.4Сравнительный анализ скорости преобразования данных при передаче между средами выполнения 18

2.5Сравнительный анализ скорости работы с DOM-деревом в WEB-браузере 20

2.6Результаты тестов производительности 21

**ГЛАВА 3. СОЗДАНИЕ БИБЛИОТЕКИ ДОПОЛНИТЕЛЬНЫХ КОМПОНЕНТОВ 21**

3.1Компоненты созданные в процессе разработки демонстрационного приложения 21

3.2Реализация глобального конфигурирования приложения 22

3.3Прототип приложения для библиотечного учета как демонстрация возможностей Blazor 23

**ГЛАВА 4. РЕШЕНИЕ ТИПОВЫХ ЗАДАЧ ПРИ ПОМОЩИ ФРЕЙМВОРКА BLAZOR 28**

4.1Создание приложения по шаблону 28

4.2Работа с компонентами 29

4.3Привязка данных 31

4.4Расширенная работа с пользовательским интерфейсом 33

4.5Маршрутизация и инъекция зависимостей 34

**ЗАКЛЮЧЕНИЕ 36**

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК 37**

# ВВЕДЕНИЕ

На сегодняшний день компьютерные приложения решают большой спектр производственных задач: позволяют повысить производительность труда, облегчить документооборот на предприятиях, а также в некоторых случаях избежать человеческого фактора и повысить стабильность работы предприятия.

Огромную нишу среди компьютерных приложений занимают Web-приложения, они решают колоссальный круг задач. Web-приложения состоят из двух основных частей: клиентской и серверной. В современном мире наблюдается бурный рост количества технологий и подходов для клиентской части, однако остается не решенными еще множество проблем в разработке клиентской части в том числе проблема дублирования и кода, низкой возможности для переиспользования кода, а также проблемы с отладкой и юнит-тестированием.

Попыткой решить множество проблем разработки клиентских приложения является технология WebAssembly позволяющая расширить возможности разработчиков. На базе WebAssembly появляются многие решения. Одним из них является фреймворк Blazor, который рассматривается в данной работе. Blazor является SPA-фреймворком на технологии .NET. Для его работы в браузере он предоставляет скомпилированный в WebAssembly .NET runtime, однако подобное решение чревато проблемами с производительностью.

Данная работа содержит результаты тестирования производительности фреймворка Blazor, содержит сведения о создании библиотеки компонентом для решения тривиальных задач, и делаются выводы о пригодности фреймворка для создания новых проектов.

**ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ И ОБЗОР ТЕХНОЛОГИЙ**

**1.1** **Постановка задачи**

**Цель работы: изучение и анализ характеристик фреймворка Blazor, сравнение с другими технологиями,** создания прототипа Web-приложений для демонстрации возможностей.

**Задачами данной работы являются:**

* **Рассмотрение передовых технологий создания Web-приложений**
* **Анализ документации технологии Blazor, выявление основных характеристик**
* **Разработка тестов производительности фреймворка Blazor и альтернативных технологий**
* **Разработка средств межсредового взаимодействия и анализ их эффективности.**
* **Анализ компонентной модели Blazor, создание библиотеки дополнительных компонентов**
* Создание демонстрационного приложения для запуска тестов с использованием основных возможностей фремворка Blazor

**1.2 Существующее технологии, применяемые в Web-клиентах**

Одним из направлений развития Web-приложений стало размещение некоторой части логики приложения (такой как проверка корректности вводимых данных) в самом Web-клиенте, например, в Web-браузере. В частности, современные Web-браузеры способны интерпретировать код на скриптовых языках, выполнять Java-апплеты и элементы управления ActiveX, использовать другие дополнения, такие как Macromedia Flash Player. Рассмотрим все эти возможности браузеров подробнее.

**Скриптовые языки.** Большинство современных Web-браузеров способно интерпретировать код на скриптовых языках, таких как VBScript и JavaScript. Код на этих языках внедряется в Web-страницу и интерпретируется браузером. Типичный пример применения скриптовых языков — проверка корректности данных, вводимых пользователем в соответствующие поля HTML-формы, непосредственно в процессе ввода или после него, без обращения к Web-серверу. Подобные примеры применения скриптовых языков можно обнаружить при заполнении некоторых анкет и получении сообщений о том, что не заполнены обязательные поля (справедливости ради отметим, что далеко не все анкеты реализованы подобным образом).

Однако есть и другие примеры применения скриптовых языков, реализующие как чисто дизайнерские идеи, например, кнопки, меняющие свой вид при наведении на них курсора, «бегущие строки», так и иную функциональность, например, внедренные в Web-страницы средства обращения к поисковым системам, отображение диалоговых панелей, управление другими объектами, встроенными в Web-страницу (например, Java-апплетами или элементами управления ActiveX, о которых будет рассказано ниже).

Стоит отметить, что код, созданный с помощью скриптовых языков, не может работать самостоятельно — он выполняется в адресном пространстве браузера. Кроме того, скриптовые языки содержат ограниченный набор средств (например, они не обладают средствами доступа к файловой системе).

**Java-апплеты.** Практически все современные браузеры способны отображать и выполнять Java-апплеты — специальные Java-приложения, которые пользователь получает в составе Web-страницы. Эти приложения нередко включаются в состав Web-страниц с целью добавления функциональности, которую сложно или невозможно реализовать с помощью скриптовых языков. Апплеты могут выполняться на всех платформах, для которых доступна виртуальная Java-машина.

Апплеты обычно создаются в соответствии с правилами, оговаривающими период их жизни и способы взаимодействия со своим окружением. Чаще всего эти способы весьма ограниченны (например, такие операции, как считывание и запись файлов, по умолчанию для апплетов запрещены; если же подобные операции необходимы, разрешения на их выполнение для конкретных апплетов и конкретных файлов описываются на клиентском компьютере; сетевой доступ из апплета возможен только к тому компьютеру, с которого он был загружен; запуск других приложений на компьютере пользователя из апплетов невозможен). Однако апплет способен считывать значения параметров (например, цвета, шрифтов, файлов с графическими изображениями, используемыми при выполнении апплета) с содержащей его Web-страницы и в соответствии с этими параметрами изменять свое поведение. Кроме того, параметры апплета можно менять динамически из кода на скриптовых языках, содержащихся в составе той же страницы.

Стоит отметить, что, поскольку апплеты реализуют выполнение кода на компьютере клиента, они в определенной степени являются потенциально опасным содержимым. Именно поэтому все современные браузеры обладают доступными пользователю средствами ограничения возможностей выполнения апплетов.

**Элементы управления ActiveX.** Некоторые из современных браузеров (в частности, Microsoft Internet Explorer) могут служить контейнерами для элементов управления ActiveX — специальных COM-серверов, выполняющихся в адресном пространстве браузера и также получаемых в составе Web-страницы.

С помощью элементов управления ActiveX, как и посредством Java-апплетов, можно реализовать любую функциональность, в том числе и неблагоприятную для компьютера пользователя, при этом, в отличие от Java-апплетов, при выполнении элементов управления ActiveX в общем случае нет никаких ограничений на доступ к файлам и иным ресурсам операционной системы и сети, а код, содержащийся в них, выполняется от имени загрузившего их пользователя. Как и Java-апплеты, элементы управления ActiveX могут считывать свои свойства с содержащей их страницы; кроме того, свойства элемента управления ActiveX можно менять динамически из кода на скриптовых языках, содержащихся в составе той же страницы; в том же коде можно обрабатывать события, возникающие в таких элементах управления.

Естественно, Microsoft Internet Explorer обладает средствами ограничения возможностей выполнения элементов управления ActiveX, в том числе управления ими из кода на скриптовых языках. Однако для контроля безопасности их выполнения имеется еще одно средство, называемое электронной цифровой подписью. Цифровая подпись помещается внутрь элемента управления ActiveX, для чего требуется наличие соответствующего электронного сертификата. Электронная подпись, помимо сведений о фирме-производителе, содержит и другую полезную информацию. Так, например, если файл с элементом управления ActiveX после добавления электронной подписи был изменен, то об этом будет немедленно сообщено перед запуском такого элемента управления — при добавлении подписи к элементу управления ActiveX происходит вычисление контрольной суммы соответствующего файла. Отметим, однако, что в России в настоящее время нет авторизованных компаний, которые могли бы выдать электронный сертификат международного образца. Естественно, наличие электронного сертификата не гарантирует отсутствия потенциально опасного содержимого, но, по крайней мере, позволяет клиенту установить его источник.

Далее нам следует напомнить банальную истину, которая, как показывает практика, очевидна не для всех наших читателей. При работе с элементами управления ActiveX и Java-апплетами абсолютно бесполезно полагаться на антивирусное программное обеспечение (неважно, клиентское оно или серверное): признаков, характерных для вирусов (таких как способность внедряться внутрь исполняемых файлов и документов), подобные приложения, как правило, не содержат. Можно лишь запретить загрузку или выполнение соответствующего кода либо на уровне настроек браузера, либо на уровне корпоративных или персональных брандмауэров.

**Приложения Macromedia Flash.** Приложения Macromedia Flash являются сегодня наиболее популярным расширением функциональности Web-браузеров — с их помощью многие Web-дизайнеры придают своим сайтам интерактивность и оригинальность.

Модель безопасности приложений Flash основана на том, что Macromedia Flash Player, как и виртуальная Java-машина, выполняет приложения в ограниченном адресном пространстве, при этом выполняемые приложения не имеют доступа к файловой системе (кроме одного конкретного каталога, используемого Macromedia Flash Player для служебных целей) и другим ресурсам компьютера пользователя; исключение делается для микрофонов и видеокамер, однако пользователь должен дать разрешение на передачу данных, полученных с этих устройств. Доступ к сетевым ресурсам ограничивается доменом, с которого было получено приложение. Отметим, что приложения Flash также могут управляться с помощью кода JavaScript, присутствующего на той же странице. Сам Macromedia Flash Player для Microsoft Internet Explorer является элементом управления ActiveX и использует возможности элементов управления ActiveX для доступа к свойствам приложений Flash из скриптовых языков.

Стоит отметить, что помимо вышеперечисленных наиболее популярных средств расширения функциональности браузеров имеется и ряд других средств, реализованных обычно в виде так называемых модулей расширения (plug-in). Поскольку модули расширения также представляют собой исполняемый код, современные браузеры обладают средствами ограничения возможностей, связанных с их загрузкой и выполнением.

В заключение стоит отметить, что перечисленные средства расширения функциональности HTML-страниц могут быть использованы и в динамических страницах, генерируемых серверными Web-приложениями. Так, в последнее время широкое распространение приобрели средства создания Web-приложений, выполняющихся под управлением Web-серверов и генерирующих динамические HTML-страницы с внедренным в них кодом на скриптовых языках, предназначенным для интерпретации браузером.

**WebAssembly.** WebAssembly оказывает огромное влияние на веб-платформу — он предоставляет способ исполнения кода, написанного на различных языках, в сети, со скоростью близкой к нативной, чего ранее невозможно было достичь.

WebAssembly разработан для дополнения JavaScript – используя WebAssembly JavaScript API вы можете загружать модули WebAssembly в приложения JavaScript и обеспечивать взаимодействие между ними, используя общие функции. Такой подход позволяет вам получить производительность и мощность WebAssembly, а также выразительность и гибкость JavaScript в ваших приложениях, даже если вы не знаете, как писать код WebAssembly, а используете готовые модули.

Наиболее премечательно здесь то, что WebAssembly разрабатывается как веб-стандарт W3C WebAssembly Working Group и Community Group при активном участии основных производителей браузеров.

**1.3 Платформа .NET**

.NET - это платформа, которая была первоначально создана для создания сетевых приложений. Язык C# был создан специально для работы с фреймворком .NET, однако само понятие .NET несколько шире. Можно выделить следующие ее основные черты:

**Поддержка нескольких языков.** Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), благодаря чему .NET поддерживает несколько языков: наряду с C# это также VB.NET, C++, F#, а также различные диалекты других языков, привязанные к .NET, например, Delphi.NET. При компиляции код на любом из этих языков компилируется в сборку на общем языке CIL (Common Intermediate Language) - своего рода ассемблер платформы .NET. Поэтому мы можем сделать отдельные модули одного приложения на отдельных языках.

**Кроссплатформенность.** .NET является переносимой платформой (с некоторыми ограничениями). Например, последняя версия платформы на данный момент .NET Framework поддерживается на большинстве современных ОС Windows (Windows 10/8.1/8/7/Vista). А благодаря проектам Mono и .NET Core можно создавать приложения, которые будут работать и на других ОС семейства Linux, в том числе на мобильных платформах Android и iOS.

**Мощная библиотека классов.** .NET представляет единую для всех поддерживаемых языков библиотеку классов. И какое бы приложение мы не собирались писать на C# - текстовый редактор, чат или сложный веб-сайт - так или иначе мы задействуем библиотеку классов .NET.

**Разнообразие технологий.** Общеязыковая среда исполнения CLR и базовая библиотека классов являются основой для целого стека технологий, которые разработчики могут задействовать при построении тех или иных приложений. Например, для работы с базами данных в этом стеке технологий предназначена технология ADO.NET. Для построения графических приложений с богатым насыщенным интерфейсом - технология WPF. Для создания веб-сайтов - ASP.NET и т. д.

**Автоматическая сборка мусора.** Также еще следует отметить такую особенность языка C# и фреймворка .NET, как автоматическая сборка мусора. А это значит, что нам в большинстве случаев не придется, в отличие от С++, заботиться об освобождении памяти. Вышеупомянутая общеязыковая среда CLR сама вызовет сборщик мусора и очистит память.

**Совмещение управляемого и неуправляемого кода.** Нередко приложение, созданное на C#, называют управляемым кодом (managed code). Что это значит? А это значит, что данное приложение создано на основе платформы .NET и поэтому управляется общеязыковой средой CLR, которая загружает приложение и при необходимости очищает память. Но есть также приложения, например, созданные на языке С++, которые компилируются не в общий язык CIL, как C# или VB.NET, а в обычный машинный код. В этом случае .NET не управляет приложением. В то же время платформа .NET предоставляет возможности для взаимодействия с неуправляемым кодом. Мы наряду со стандартными классами библиотеки .NET можем также использовать сборки COM.

**JIT-компиляция.** Код на C# компилируется в приложения или сборки с расширениями exe или dll на языке CIL. Далее при запуске на выполнение подобного приложения происходит JIT-компиляция (Just-In-Time) в машинный код, который затем выполняется. При этом, поскольку наше приложение может быть большим и содержать кучу инструкций, в текущий момент времени будет компилироваться лишь та часть приложения, к которой непосредственно идет обращение. Если мы обратимся к другой части кода, то она будет скомпилирована из CIL в машинный код. При том уже скомпилированная часть приложения сохраняется до завершения работы программы. В итоге это повышает производительность.

**1.4 Blazor**

Blazor — это экспериментальный .NET фреймворк использующий C#/Razor и HTML, который работает в браузере с помощью WebAssembly. Blazor предоставляет все преимущества создания .NET приложений на клиенте и, при необходимости, на сервере.

За последние годы веб-разработка во многих отношениях улучшилась, но создание современных веб-приложений по-прежнему не простая задача. Использование .NET в браузере дает много преимуществ, которые могут помочь сделать веб-разработку проще и продуктивнее:

Стабильность и согласованность: .NET обеспечивает стандартизированные рамки программирования на всех платформах, они являются стабильными, многофункциональными и простыми в использовании.

Современные инновационные языки: Языки .NET постоянно совершенствуются, добавляя новые инновационные функции.

Лучшие инструменты в отрасли: Семейство продуктов Visual Studio обеспечивает фантастическую разработку используя .NET на всех платформах в Windows, Linux и MacOS.

Скорость и масштабируемость: .NET имеет сильную историю производительности, надежности и безопасности для разработки приложений. Использование .NET в качестве решения для full-stack разработки упрощает создание быстрых, надежных и безопасных приложений.

Full-stack разработка, которая использует существующие навыки и наработки: Разработчики C#/Razor используют их C#/Razor навыки и наработки для написания клиентских приложений и логики обмена данных с сервером.

Широкая поддержка браузеров: Blazor работает на .NET с использованием открытых веб-стандартов в браузере, без плагинов и без перекодирования кода. Он работает во всех современных веб-браузерах, включая мобильные.

Выполнение .NET-кода внутри веб-браузеров стало возможным благодаря относительно новой технологии, WebAssembly (сокращенно wasm). WebAssembly является открытым веб-стандартом и поддерживается в веб-браузерах без плагинов. WebAssembly - это компактный формат байт-кода, оптимизированный для быстрой загрузки и максимальной скорости выполнения.

Код WebAssembly позволяет получить доступ ко всем функциям браузера через интерфейс JavaScript. В то же время код WebAssembly работает в той же доверенной изолированной среде, что и JavaScript, для предотвращения вредоносных действий на клиентской машине.

Когда приложение Blazor создается и запускается в браузере:

1. Файлы C# кода и файлы Razor скомпилированы в .NET сборки.
2. Сборки и среда выполнения .NET загружаются в браузер.
3. Blazor использует JavaScript для загрузки среды выполнения .NET и настраивает среду выполнения для загрузки необходимых ссылок на сборку. Обработка объектной модели документа (DOM) и вызовы API браузера обрабатываются исполняемой средой Blazor с помощью JavaScript функционала.

Для поддержки старых браузеров, которые не поддерживают WebAssembly, Blazor использует asm.js на основе .NET.

Приложения Blazor создаются с помощью компонентов*.* Компонент — это часть пользовательского интерфейса (UI), например, страница, диалог или форма ввода данных. Компоненты могут быть вложенными, повторно использованными и разделяемыми между несколькими проектами.

В Blazor, компонент это .NET класс. Класс может быть написан непосредственно, как C# класс (\*.cs), или чаще в виде страницы разметки Razor (\*.cshtml)

.

Razor является синтаксисом для комбинирования разметки HTML с кодом C#. Razor разработан для повышения продуктивности разработчиков, позволяя разработчику переключаться между разметкой и C# кодом в том же файле с помощью поддержки IntelliSence.

Компоненты поддерживают следующие возможности:

* Вложенность
* Наследование
* Создание с помощью Razor (\*.cshtml) или кода C# (\*.cs)
* Двусторонняя привязка данных
* Создание библиотек компонентов
* Юнит-тестирование без необходимости использования DOM-браузера.

**ГЛАВА 2. ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ И РЕАЛИЗАЦИЯ ДЕМОНСТРАЦИОННОГО ПРИЛОЖЕНИЯ**

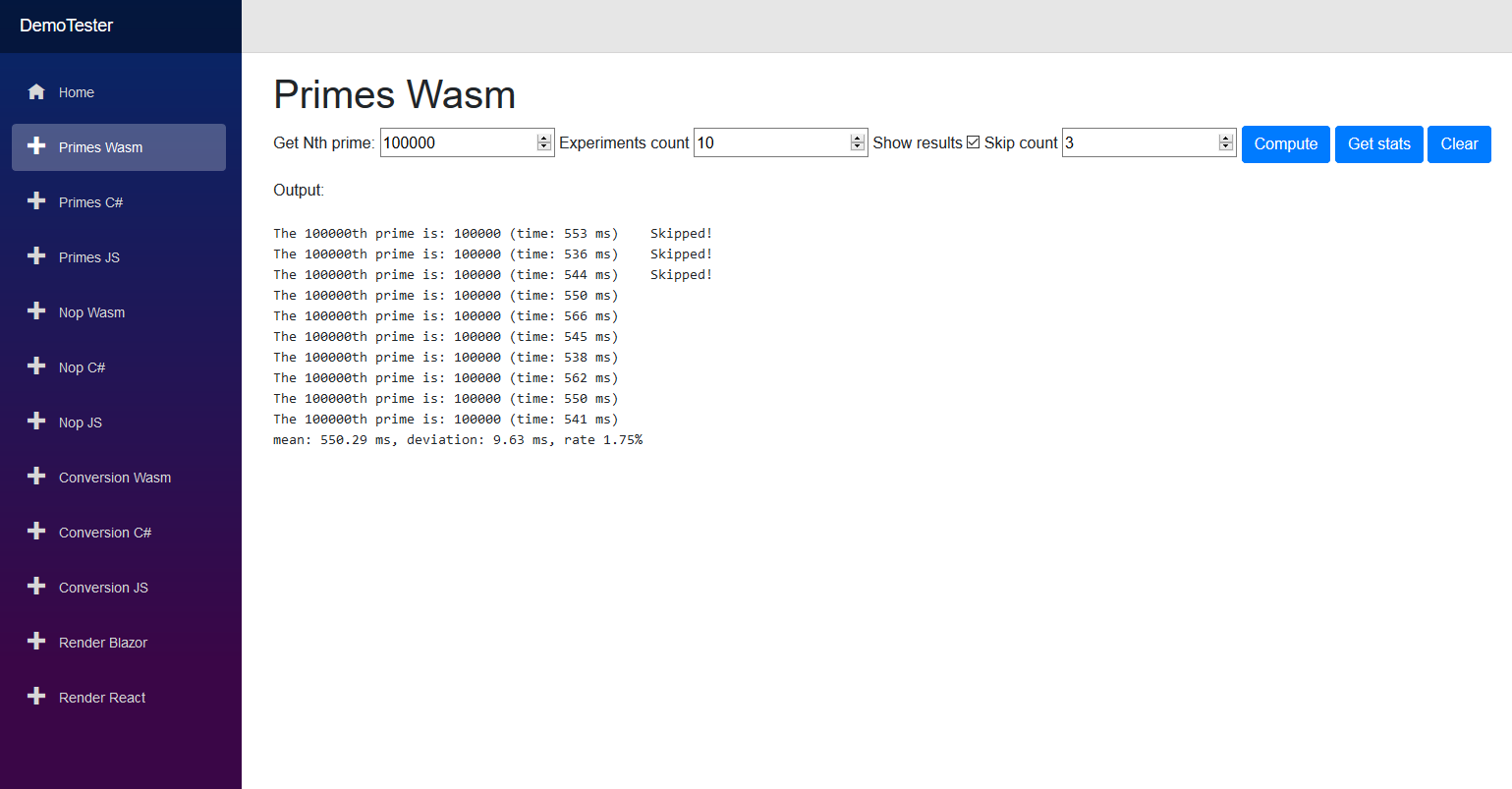
Для анализа характеристик приложения были проведены несколько тестов производительности:

* Тест скорости вычислений.
* Тест накладных расходов при вызовах между средами выполнения.
* Тест скорости преобразования данных при передаче между средами исполнения.
* Тест скорости работы с DOM-деревом в WEB-браузере.

**2.1 Демонстрационное приложение**

Было построено демонстрационное приложение, позволяющее быстро запускать тесты и получать наглядные данные. В процессе его создания было создано несколько Blazor-компонентов которые могут быть переиспользованны в других проектах. Подробнее о созданных компонентах см. в главе 3.

Демонстрационное приложение содержит несколько однообразных вкладок с тестами позволяющих запускать их с разными параметрами и варьировать результаты его работы.



**Рисунок 1. Интерфейс демонстрационного приложения**

**2.2 Сравнительный анализ скорости вычислений**

Потребление ресурсов центрального процессора является одной из основных характеристик производительности наряду с потреблением ОЗУ. Исследование скорости вычислений позволит определить набор задач, которые будут выполняться в клиентской части приложения за приемлемое время.

Для тестирования производительности возьмём простую вычислительную задачу и проведем серию запусков в различных средах исполнения с произведением замера времени выполнения. Далее вычислим среднее значение, которое и будем считать результатам измерений, а также среднеквадратичное отклонение для определения устойчивости результата.

Исследуемыми средами исполнения будут:

* Скриптовый движок браузера для JavaScript, для него будет написан скрип на этом языке.
* Среда выполнения Mono для браузера исполняющий .NET Standard сборки, скомпилированные из исходных кодов на языке C#.
* WebAssembly-движок исполняющий предварительно скомпилированные .NET Standard сборки в поддерживаемый им байт-код.

Вычислительной задачей для проведения тестов будет "определения n-ого простого числа". Возьмем самый простой алгоритм решения этой задачи:

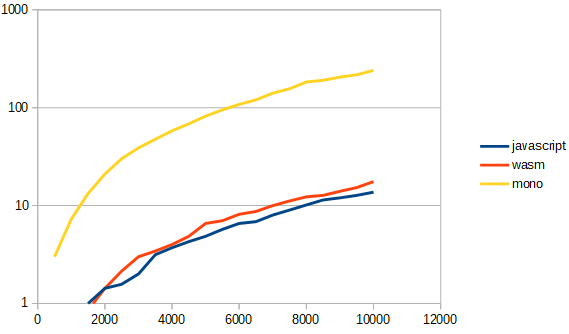
1. Установим значение счетчика равным 0
2. Будем проверять числа начиная с двойки на простоту
3. Увеличим счетчик если текущее число является простым
4. Если счетчик был увеличен и его значение равного номеру искомого числа, то текущее число является результатом, иначе проверяется следующее простое число.

Простые числа в свою очередь будут проверяться перебором делителей от двойки до квадратного корня от текущего числа. Этот предельно простой алгоритм позволит продемонстрировать разницу в скорости выполнения исследуемый средах.

В результате выполнения серии опытов были получены следующие результаты они отображены на рисунке 2.

Для порядкового номера 10000 были получены следующие результаты среднего значения времени выполнения:

* WebAssembly: 17.6 миллисекунды
* JavaScript: 13.7 миллисекунды
* Mono в режиме интерпретации: 240.6 миллисекунды



**Рисунок 2. График зависимости времени выполнения от порядкового номера**

Как видно режим интерпретации сборок работает медленнее порядка 20 раз (в 17.5 раз в данном тесте), однако предварительно скомпилированные сборки работают медленнее всего в 1.28 раза несмотря на то, что технически код для платформы WebAssembly должен решать аналогичные задачи быстрее кода на JavaScript.

Исследования производительности WebAssembly показывают, что аналогичный код исполняемый в движке JavaScript в среднем на 5-10% медленнее, чем аналогичный собранный для WebAssembly. Данные расхождения связанны с неэффективностью конвертации сборки в WebAssembly байт-код. Если же изначально писать код на языке С++ и скомпилировать его в WebAssembly то его скорость выполнения будет выше, чем у JavaScript. Был написан отдельный тест, который показал среднее значение времени выполнения в 12.2 миллисекунды для аналогичного кода на С++ скомпилированного в WebAssembly байт-код.

Таким образом, выполнение ресурсоемких задач режим интерпретации Mono неприемлемо. Требуется предварительное парообразование сборки в WebAssembly байт-код и вызовы wasm-кода из кода работающего в режиме интерпретации. Скорость исполнения WebAssembly-сборок соизмерима со скоростью исполнение кода на языке JavaScript, что позволяет использовать данные фреймворк и язык C# как альтернативу, при условии, что весь ресурсоемкий код будет скомпилирован в предварительно в формат WebAssembly байт-код.

Также было замечена важная особенность: первый запуск опыта всегда занимал значительно больше времени чем последующие. Это связано с тем, что первый запуск производит инициализацию исполняемого кода и выполняет JIT-компиляцию в машинный код, а уже последующие запуске используют скомпилированные версии кода. Поэтому для подсчета статистики среднего значения времени выполнения первые запуски опытов были скрещены чтобы исключить погрешность, вызванная задержкой при первом запуске, а также для выполнения прогрева кэша центрального процессора.

Также была замерена средняя задержка при первом запуске на разных платформах. Для кода в режиме интерпретации задержка была меньше 1 миллисекунды, что связано с только с "прогревом" кэша, так как подобный режим исполнения не требует сложной инициализации и компиляции.

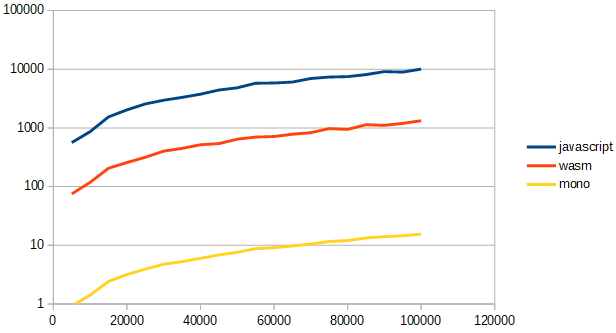
Для кода на JavaScript время инициализации составило в среднем 9 миллисекунд, а для WebAssembly в среднем 4 миллисекунды. Инициализация WebAssembly байт-кода занимает меньше времени, так как этот формат фактические является синтаксическим деревом кода программы и требует только компиляцию в машинный код. В свою очередь код на JavaScript является человекочитаемым и поэтому предварительно требует разбора (парсинга) и построения синтаксического дерева и далее компиляции в машинный код. Эти данные могут быть использованы при анализе времени отклика и "холодного" старта приложения.

В дальнейших опытах замеры времени инициализации приведены не будут, так как они аналогичны данным в этом опыте. В дальнейших тестах первые несколько запусков будут так же игнорироваться для повышения точности.

**2.3 Сравнительный анализ накладных расходов при вызовах между средами выполнения**

В данном опыте замеряется время вызова метода между средами выполнения. Между Mono и Mono, между Mono и WebAssembly и между Mono и JavaScript. Данный тест аналогичен предыдущему за исключением того, что тела вычисляющего метода пусто.

В результате выполнения серии опытов были получены следующие результаты они отображены на рисунке 3.

****

**Рисунок 3. График зависимости времени выполнения от количества вызовов**

Количества вызовов в 100000 раз были получены следующие результаты среднего значения времени выполнения:

* WebAssembly: 1319.6 миллисекунды
* JavaScript: 9679.2 миллисекунды
* Mono в режиме интерпретации: 15.5 миллисекунды

Таким образом вызов внутри Mono происходит очень быстро, вызов в WebAssembly в 80 раз дольше чем внутри Mono, а вызов в JavaScript занимает в 630 раз больше чем внутри Mono.

**2.4 Сравнительный анализ скорости преобразования данных при передаче между средами выполнения**

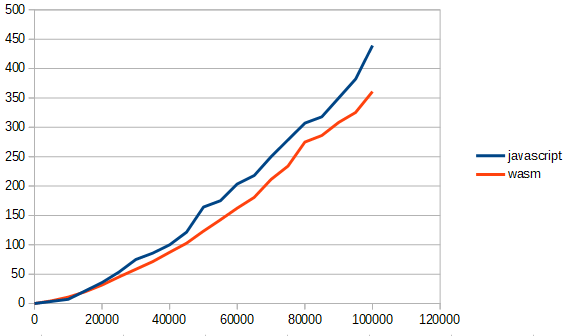
Результаты данного теста важны если часть ресурсоемкой логики приложения переноситься на другую среду исполнения. В данном тесте будет генерироваться древовидный объект на основе случайно сгенерированного дерева.

Случайное дерево генерируется следующим образом:

* Создается множество элементов
* Нулевой элемент считается корнем дерева
* Происходит последовательный обход всех элементов (с индекса 1)
* Выбирается случайный элемент с индексом меньше текущего
* Текущий элемент становиться дочерним для выбранного и происходит переход к следующему элементу

Далее сгенерированное дерево заполняется случаными данными в нашем случае одним полем-строкой фиксированной длинны. После этого происходит вызов метода в другой среде выполнения и который ничего не делает и возвращает управление и происходит замер времени выполнения этого вызова.

В результате выполнения серии опытов были получены следующие результаты они отображены на рисунке 4.



**Рисунок 4. График зависимости времени выполнения от количества элементов**

Для структуры размером в 100000 элементов были получены следующие результаты среднего значения времени выполнения:

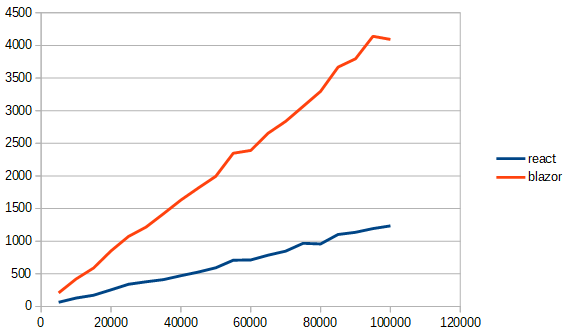
* Blazor → WebAssembly: 439.3 миллисекунды
* Blazor → JavaScript: 360.9 миллисекунды

Как правило клиентская часть Web-приложение не работает с большими объемами данных, в таких случаях этими значениями можно пренебречь.

**2.5 Сравнительный анализ скорости работы с DOM-деревом в WEB-браузере**

Для сравнения скорости работы с DOM-деревом были выбраны технологии Blazor и React. Для отображения React-элементов в Blazor был разработан компонент ReactRenderer который позволяет генерировать и отображать React-элементы. Будет отображаться вложенный маркированный список (теги ul и li) по древовидному элементу, сгенерированному по аналогии с разделом 2.4. Далее замеряется время, за которое каждая библиотека сгенерирует разметку по данным и отобразит её при помощи DOM-api.

В результате выполнения серии опытов были получены следующие результаты они отображены на рисунке 5.



**Рисунок 5. График зависимости времени отрисовки от количества элементов**

Для структуры размером в 100000 элементов были получены следующие результаты среднего значения времени отрисовки:

* Blazor: 4092.2 миллисекунды
* React: 1234.8 миллисекунды

Таким образом отрисовка в React работает в среднем в 3.3 раза быстрее.

**2.6 Результаты тестов производительности**

Blazor работает медленно и потребляет больше ресурсов процессора, чем аналогичные технологии. Скорость вычислений в среднем в 20 раз больше, а скорость отрисовки элементов верстки в 3.3 больше. Однако просадку в производительности можно считать приемлемой, так как логика обычных приложений потребляет не значительные ресурсы процессора, большую роль играет отображение элементов. За время работы с демонстрационным приложением неудобств замечено не было. В случае если производительность критична можно реализовать критичные участки приложения на более производительной технологии и воспользоваться вызовом между средами исполнения. Стоит учитывать что ведутся работы по повышению производительности Blazor.

**ГЛАВА 3. СОЗДАНИЕ БИБЛИОТЕКИ ДОПОЛНИТЕЛЬНЫХ КОМПОНЕНТОВ**

**3.1 Компоненты созданные в процессе разработки демонстрационного приложения**

В процессе реализации демонстрационного приложения было создано несколько дополнительных компонентов пользовательского интерфейса, которые могут быть переиспользованны в других проектах. Компоненты легко мигрируются из проекта в проект, достаточно перенести файл с кодом компонента (\*.cshtml), который содержит разметку и логику компонента. Если компонент использует другие компоненты, то необходимо так же перенести их файлы. Далее его можно использовать в любом желаемом компоненте или странице. Так же можно использовать nuget-пакеты для упрощения подключения компонентов к другим проектам.

**Компонент PerformanceTester**

Для обобщения реализации страниц с тестами производительности был создан компонент PerformanceTester. Он упрощает реализацию теста, имеет параметр "Action", который получает исполняемый делегат (указатель на функцию).  
  
Данный компонент отображает различные опции для запуска теста:

* количество запусков
* передаваемый аргумент
* количество пропущенных запусков для повышения точности (из-за "прогрева" кэша)
* включение/отключение подробностей по каждому запуску (время выполнения каждого конкретного запуска)

Так же содержит два режима запуска тестов:

* Одиночный запуск - запуск серии тестов с заданным аргументом
* Многократный запуск — множество запусков с постепенным увеличением аргумента

Многократный запуск выдает данные удобные для построения графиков. Дополнительно присутствует кнопка для отчистки отчета о запусках. На данном компоненте базируются все страницы запускающие тесты.

**Компонент ReactRenderer**

Компонент ReactRenderer был создан для упрощения включения в Blazor-приложения разметки сгенерированной библиотекой ReactJS он принимает в себя JavaSctipt функцию, генерирующую ReactElement и её аргументы, и создает DOM-element с уникальным идентификатором, а далее вызывается процедура React.render из js-библиотеки ReactJS принимающая сгенерированный ReactElement и идентификатор контейнера, которая отображает разметку на странице.

**Компонент BlazorPanel**

Компонент BlazorPanel является панелью с заголовком по нажатию на который её содержимое "складывается" в заголовок. Этот компонент так же поддерживает локализацию и стилизацию.

**Компоненты BlazorButton, BlazorInput, BlazorCheckbox**

Компоненты BlazorButton, BlazorInput, BlazorCheckbox являются обертками над HTML-элементами button и input, дополненные средствами локализации и стилизации.

**3.2 Реализация глобального конфигурирования приложения**

Параметры стилей и локализации могут быть переданы в поддерживаемые компоненты либо через DI-сервис, либо через каскадные параметры Blazor - это позволяет задавать глобальные стили в рамках всего приложения и переопределять их на части приложения, например, в определенном окне через каскадные параметры.

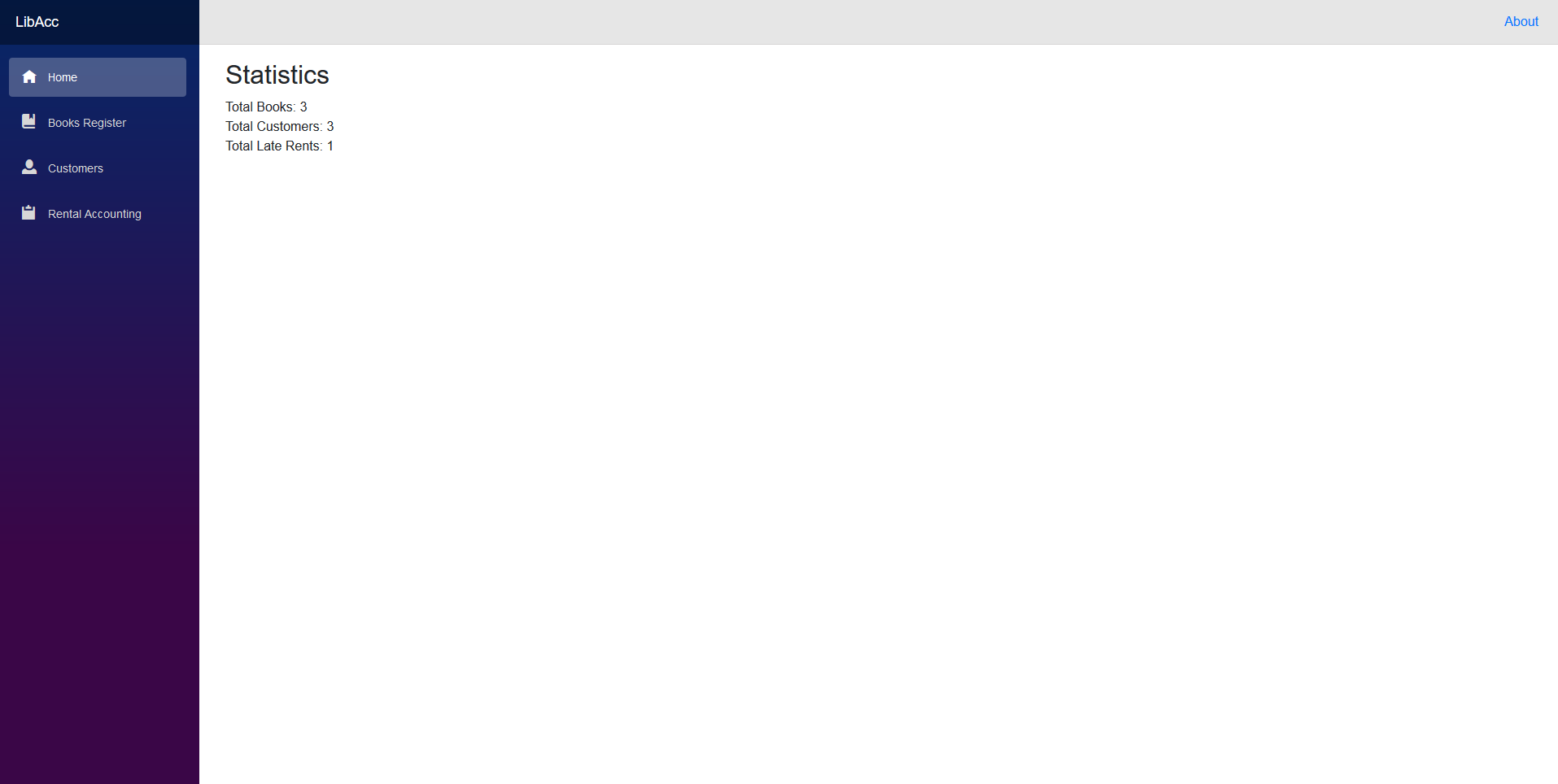
Такая реализация позволит избавить код от загрязняющего пробрасывания параметров через все дерево элементов документа. Каскадные параметры реактивные, это позволяет мгновенно перестраивать приложение в случае их изменения и реализовывать такие возможности как смена темы оформления и языка без необходимости перезагрузки страницы.

**3.3 Прототип приложения для библиотечного учета как демонстрация возможностей Blazor**

В рамках данной работы был разработан прототип приложения для библиотечного учета. Приложение разделено на заглавную страницу и три вкладки. Слева имеется навигационное меню, позволяющее переключаться между страницами.

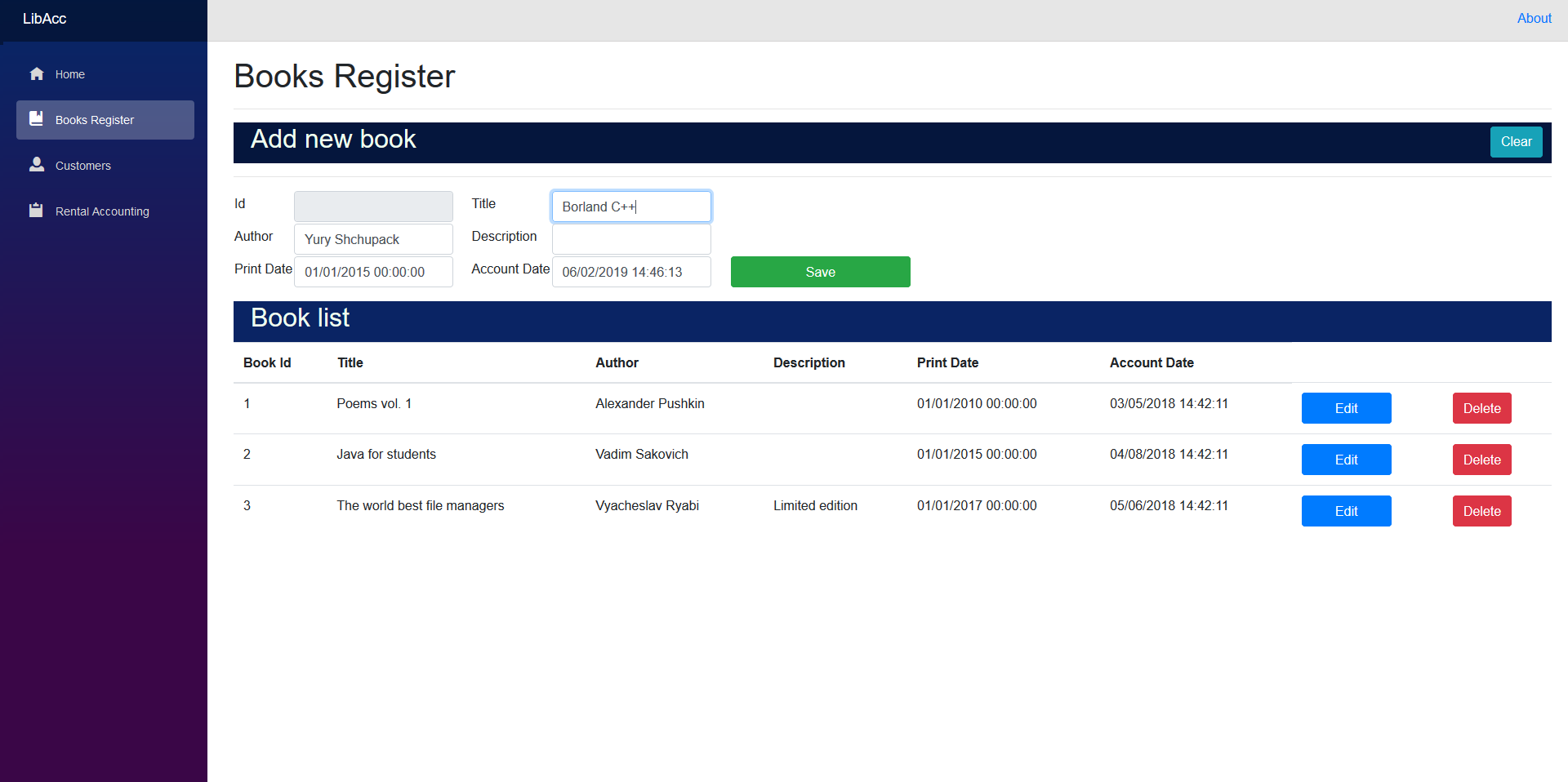
На заглавной странице отображается общая статистика:

* Количество зарегистрированных книг;
* Количество зарегистрированных пользователей;
* Количество просроченных книжных аренд.



**Рисунок 6. Главная страница приложения библиотечного учета**

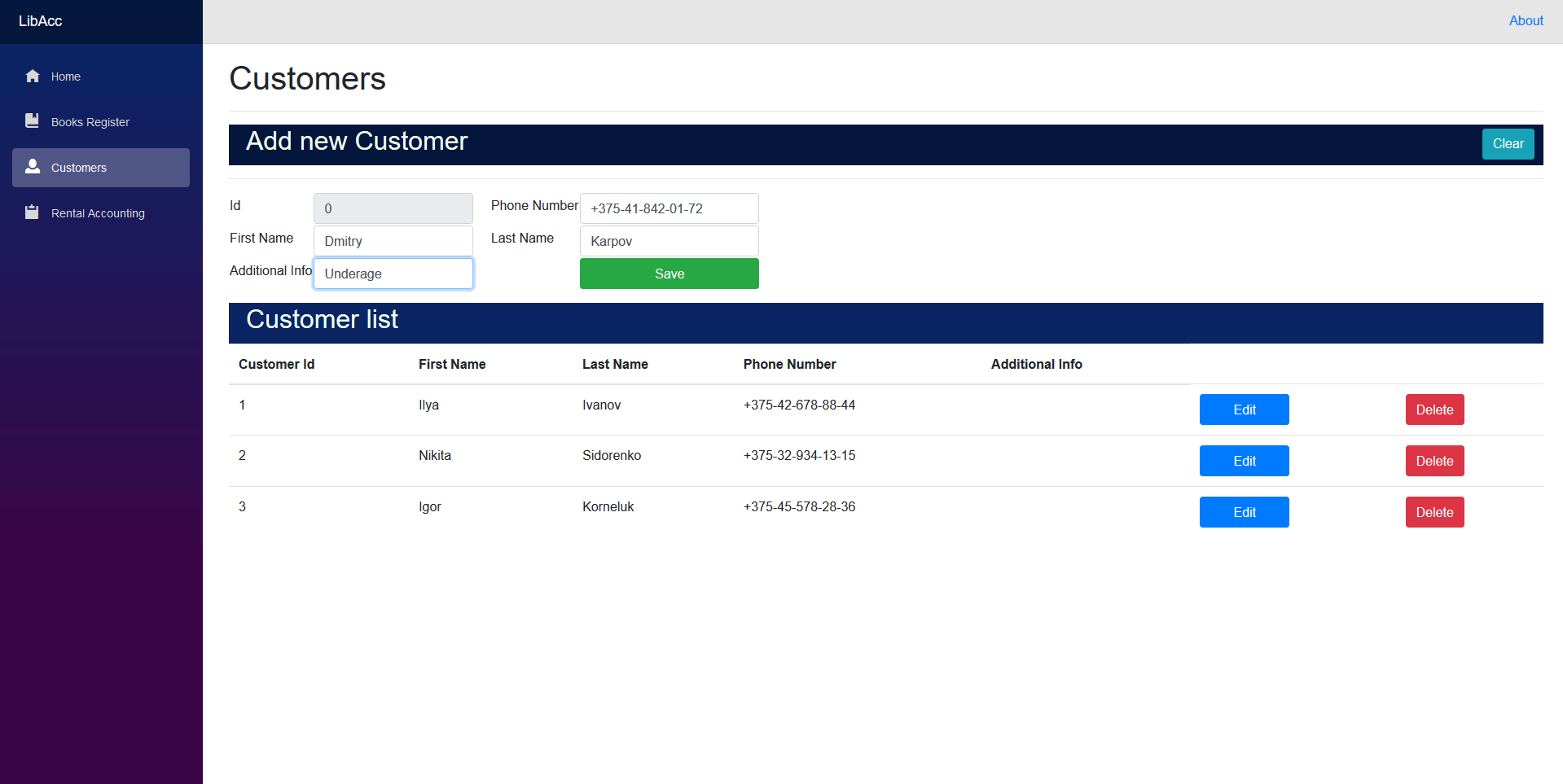
Страница редактирования реестра книг позволяет обозревать и редактировать реестр книг. Она состоит из таблицы, в которой отображается список книг и формы редактирования книги. Можно ввести данные книги в форму редактирования, далее нажать на кнопку сохранения и в результате произойдет обращение к серверу. Сервер произведет запись в базу данных, а список книг в пользовательском интерфейсе дополнится новой записью. Напротив, каждой записи находятся кнопки редактирования и удаление. Нажатие на кнопку удаления произведет запрос к серверу и запись о книге будет удалена из базы данных. Нажатие на кнопку редактирования переносит данные книги в форму редактирования, что позволяет изменить их и сохранить в базе данных произведя запрос к серверу нажатием на кнопку сохранения. Также присутствует кнопка очистки форму редактирования, которая сбросит все поля на форме без обращения к серверу.



**Рисунок 7. Страница редактирования реестра книг приложения библиотечного учета**

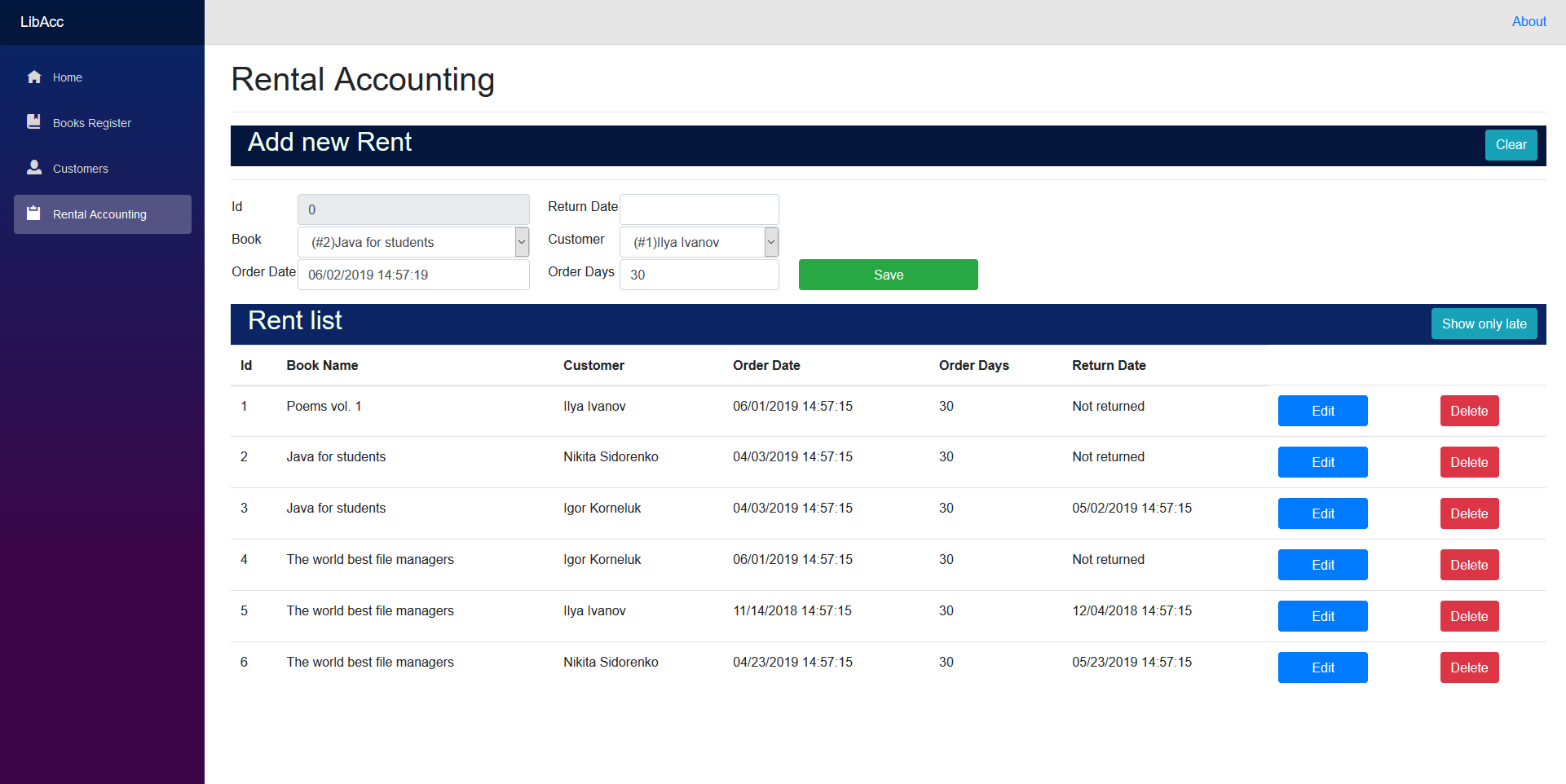
Страница редактирования базы посетителей позволяет просматривать, создавать, редактировать и удалять посетителей аналогичным образом как на вкладке со списком книг.

Можно ввести данные пользователя в форму редактирования, далее нажать на кнопку сохранения и в результате произойдет обращение к серверу. Сервер произведет запись в базу данных, а список посетителей в пользовательском интерфейсе дополнится новой записью. Напротив, каждой записи находятся кнопки редактирования и удаление. Нажатие на кнопку удаления произведет запрос к серверу и запись о пользователе будет удалена из базы данных. Нажатие на кнопку редактирования переносит данные пользователя в форму редактирования, что позволяет изменить их и сохранить в базе данных произведя запрос к серверу нажатием на кнопку сохранения. Также присутствует кнопка очистки форму редактирования, которая сбросит все поля на форме без обращения к серверу.

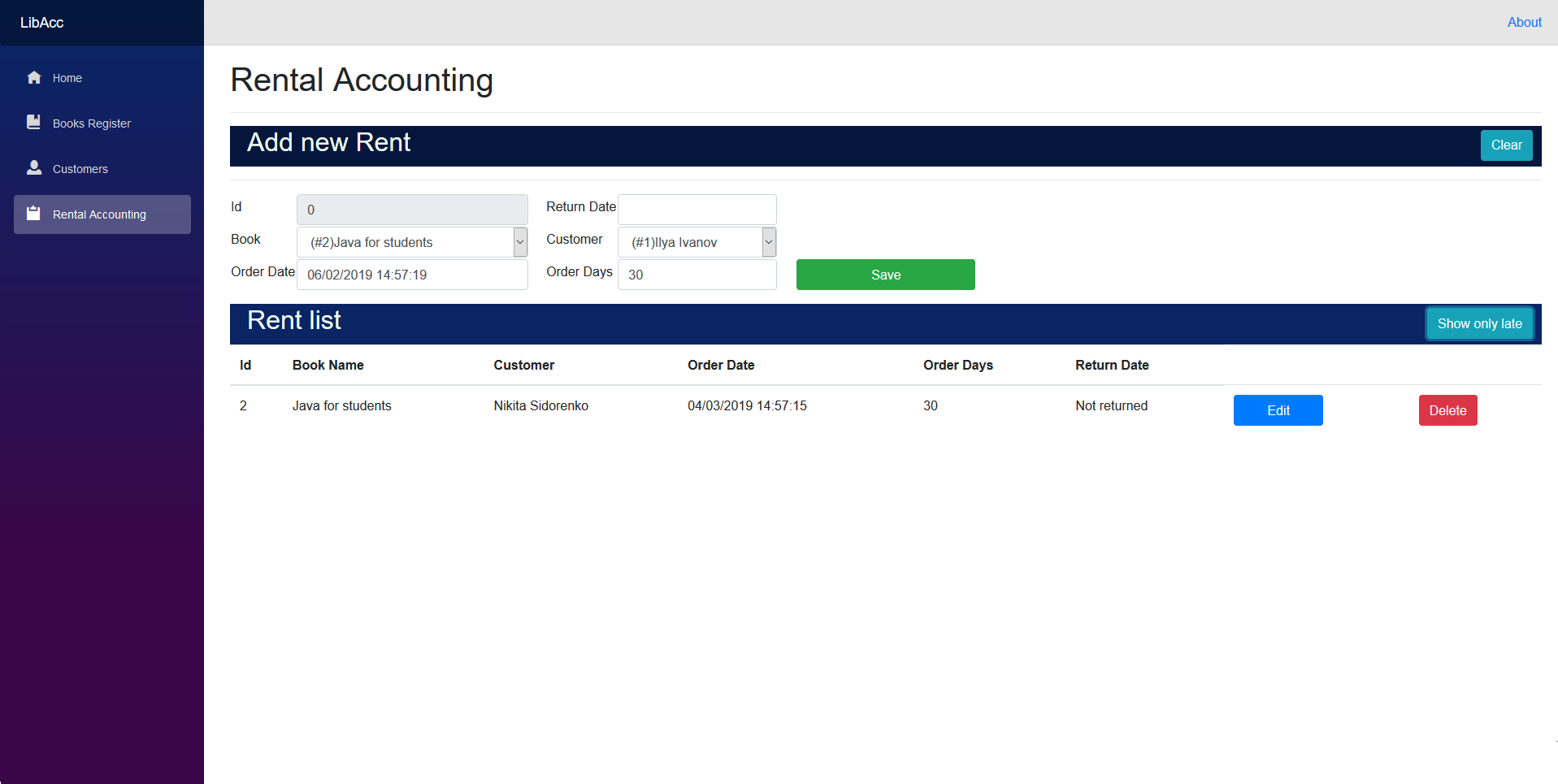


**Рисунок 8. Страница редактирования базы посетителей приложения библиотечного учета**

Страница учета выдачи книг позволяет просматривать, создавать, редактировать и удалять запили об аренде аналогичным образом как на вкладках со списком книг или пользователей, при этом связанные пользователи и книги выбираются из списка. В базу данных записывается только регистрационные номера связанного пользователя и книги, однако пользовательский интерфейс отображает имена книги и посетителя вместо номеров. Дополнительно присутствует кнопка, которая позволяет отображать только просроченные аренды.



**Рисунок 9. Страница учета выдачи книг (отображение всех записей) приложения библиотечного учета**



**Рисунок 10. Страница учета выдачи книг (отображение просроченных записей) приложения библиотечного учета**

Каждая из 4 страниц, то есть главная и три вкладки, являются Blazor-компонентом. Используя ключевое слово @page указана ссылка на этот ресурс в таблице маршрутизации. Так же отдельно был создан компонент заголовка, с использованием привязок данных в атрибуты и дочерний контент. Так же созданы дополнительные компоненты для навигационного меню и отображения статистики на главной странице.

Классы сущностей использовались совместно между клиентским и серверным приложением. Получилась удобная структура проектов, что упростит работу нескольких разработчиков над одним проектом.

Хорошо работают IDE Visual Studio 2017, а также расширение Resharper Ultimate. Данное расширение полностью поддерживает возможности Blazor.

Компонентная модель Blazor, проявила себе даже в столь небольшом приложении. Она позволила удобно разделить пользовательский интерфейс на компоненты. Было проверено большое количество современных функций SPA-фреймворка, а именно:

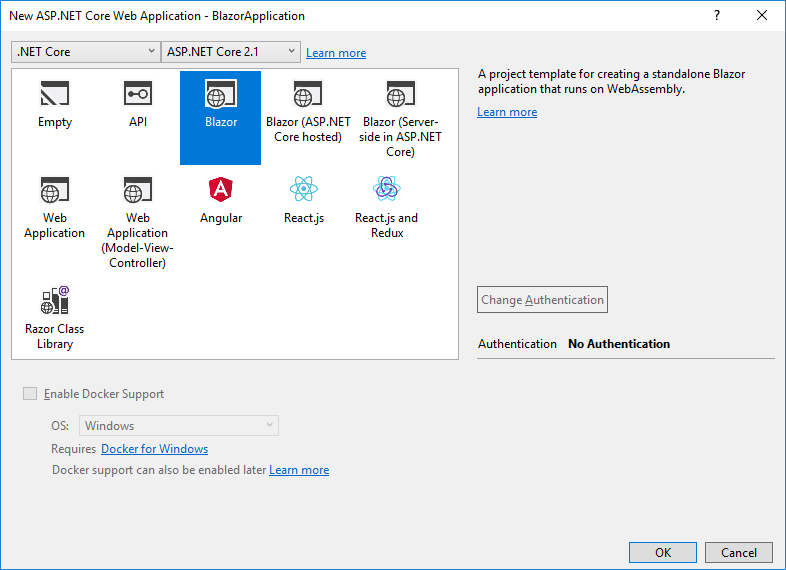
* Компонентная модель для создания пользовательского интерфейса
* Маршрутизация
* Макеты
* Формы и их валидация
* Внедрение зависимости
* Взаимодействие с JavaScript
* Перезагрузка в браузере во время разработки
* Обработка на стороне сервера
* Полная отладка .NET как в браузерах, так и в среде IDE
* IntelliSense и различные инструменты
* Возможность запуска на старых (не WebAssembly) браузерах через asm.js
* Публикация и настройка размера приложения

**ГЛАВА 4. РЕШЕНИЕ ТИПОВЫХ ЗАДАЧ SPA-ПРИЛОЖЕНИЯ ПРИ ПОМОЩИ ФРЕЙМВОРКА BLAZOR**

**4.1 Создание приложения по шаблону**

Существует официальный шаблон Blazor-приложения, используемый для быстрого старта разработки. На его основе были созданы демонстрационное приложение для запуска тестов производительности, а так же прототип приложения для библиотечного учета.

Для создание приложения по шаблону в среде разработки Visual Studio 2017 необходимо выбрать тип приложения ASP.NET Core Web Application, а далее выбрать шаблон Blazor.

****

**Рисунок 11. Мастер создания проекта в среде разработки Visual Studio 2017**

После нажатия кнопки OK будет создано приложении с реализованным базовым функционалом (компонентами-страницами, маршрутизацией и т. д.). Приложение готово к сборке и запуску.

После сборки и запуска приложение Blazor открывается в браузере и содержит три страницы: «Home», «Counter» и «Fetch data». Эти страницы реализованы тремя файлами Razor в папке Pages. Каждый из этих файлов реализует компонент Blazor, который компилируется и выполняется на стороне клиента в браузере.

**4.2 Работа с компонентами**

Компонент - это автономный фрагмент пользовательского интерфейса (например, страница, диалоговое окно или форма). Приложения Blazor создаются с использованием гибких, легких компонентов, которые можно вкладывать, использовать повторно и совместно использовать между проектами.

Компонент - это базовый элемент приложения Blazor, то есть каждая страница рассматривается как компонент в Blazor. Он использует комбинацию кода Razor, HTML и C # в качестве компонента.

Компонент в Blazor содержит как класс, включающий как HTML-разметку для рендеринга, так и логику обработки, необходимую для ввода данных или обработку событий пользовательского интерфейса.

**@page** "/counter"

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" onclick="@IncrementCount">Click me</button>

@functions {

int currentCount = 0;

void IncrementCount()

{

currentCount++;

}

}

**Листинг 1. Код страницы-компонента Counter из шаблона Blazor-приложения**

Компоненты Blazor реализованы в \*.cshtml файлах с использованием комбинации C# и HTML-разметки. Пользовательский интерфейс для компонента определяется с использованием HTML, а динамическая логика рендеринга, такая как циклы, условные выражения и операторы, добавляется с использованием встроенного синтаксиса C#, называемого Razor. В блоке @functions мы можем определить все свойства, которые используются в разметке представления, а методы связаны с событиями. Когда Blazor-приложение компилируется, разметка HTML и логика рендеринга C# преобразуются в класс компонента, а имя сгенерированного класса совпадает с именем файла.

Члены класса компонента определены в блоке @functions, вы можете использовать более одного блока @functions в компоненте. В блоке @functions указывается состояние компонента, например свойства и поля, а также методы обработки событий или определения логики других компонентов. Члены компонента могут затем использоваться как часть логики рендеринга компонента с использованием выражений C#, которые начинаются с @.

При создании компонента Blazor вы можете создавать разметку вида и логику кода C# в отдельных файлах. Использование директивы @inherits для указания компилятору Blazor выводить класс, сгенерированный из представления Razor, из класса, указанного в этой директиве.

Класс, указанный в директиве @inherits, должен быть унаследован от класса BlazorComponent, он обеспечивает всю базовую функциональность для компонента.

Компилятор Blazor генерирует класс для всех страниц представления с тем же именем класса, что и имя страницы, поэтому имя базового класса и имя файла с Razor View должны различаться, иначе это вызовет ошибку во время компиляции.

В Blazor вы можете добавлять параметры к любому компоненту, которые определены с помощью непубличных свойств класса-компонента, декорируя это свойство атрибутом [Parameter].

В разметке вы можете указать аргументы (параметры) для компонента, используя атрибуты. Атрибуты компонентов устанавливаются через синтаксис аналогичный синтаксису HTML-атрибутов.

В Blazor вы можете легко добавлять и повторно использовать любой компонент в других компонентах, объявив их с использованием синтаксиса HTML-элемента. Разметка для использования компонента выглядит как HTML-тег, где имя тега является типом компонента.

**4.3 Привязка данных**

В Blazor вы можете связывать данные как с компонентами, так и с элементами DOM, используя атрибут bind. Привязка данных является одной из самых мощных функций технологий разработки программного обеспечения. Это мост связи между View и бизнес-логикой приложения.

В Blazor вы можете связывать данные следующими способами:

* Одностороннее связывание данных
* Двусторонняя привязка данных
* Привязка событий

В других средах, таких как Angular, односторонняя привязка данных также называется интерполяцией.

При односторонней привязке нам нужно передать имя свойства или переменной вместе с @, т.е. @Title (здесь Title - это либо свойство, либо переменная).

@page "/one-way-data-binding"

<!-- Используйте эту кнопку, чтобы изменить исходные значения -->

<button onclick="@ChangeValues">Change values</button>

<p>Counter: @Count</p>

@if (ShowWarning)

{

<p style="background-color: red; padding: 5px">Warning!</p>

}

<p style="background-color: @Background; color:white; padding: 5px">Notification

</p>

<ul>

@foreach (var number in Numbers)

{

<li>@number</li>

}

</ul>

@functions {

private int Count { get; set; } = 0;

private bool ShowWarning { get; set; } = true;

private string Background { get; set; } = "red";

private List<int> Numbers { get; set; } = new List<int> { 1, 2, 3 };

private void ChangeValues()

{

Count++;

ShowWarning = !ShowWarning;

Background = Background == "red" ? "green" : "red";

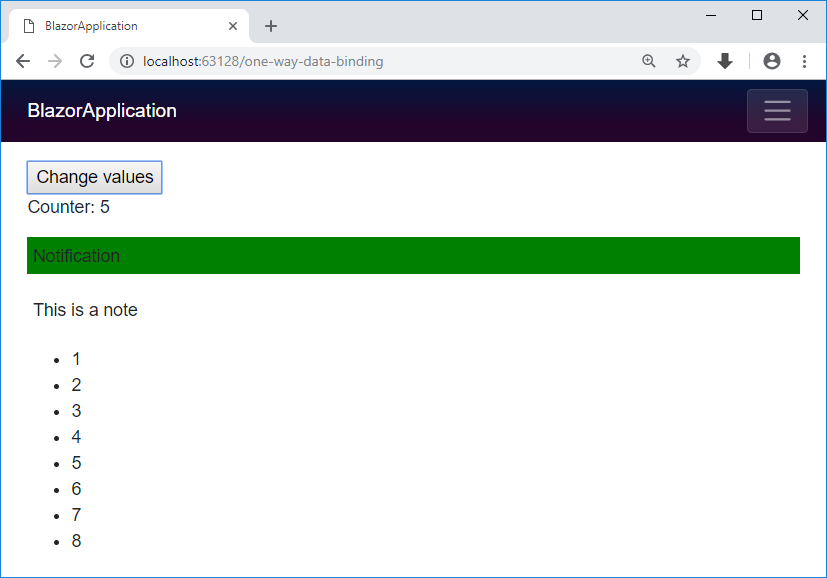
Numbers.Add(Numbers.Max() + 1);

}

}

**Листинг 2. Код демонстрирующий одностороннее связывание данных**

В приведенном выше примере мы сделали одностороннее связывание с разными переменными. Результат работы данного кода можно увидеть на рисунке 12.



**Рисунок 12. Результат работы компонента-страницы демонстрирующий одностороннюю привязку данных**

Blazor также поддерживает двустороннее связывание данных с помощью атрибута bind. В настоящее время Blazor поддерживает только следующие типы данных для двусторонней привязки данных:

* string
* int
* DateTime
* Enum
* bool

Если вам нужны другие типы (например, decimal), вам необходимо предоставить методы get и set для decimal свойства из/в поддерживаемый тип.

Так же Blazor имеет поддержку привязки C#-методов на DOM-события в том числе onclick и onchange. Список поддерживаемых событий постоянно расширяется.

**4.4 Расширенная работа с пользовательским интерфейсом**

По умолчанию Blazor автоматически обнаруживает изменения данных и совершает обновление частей пользовательского интерфейса во многих сценариях, таких как нажатие кнопки и т. д. Однако существуют ситуации, в которых вы хотите запустить обновление пользовательского интерфейса вручную. Это возможно с помощью метода StateHasChanged определенного в классе BlazorComponent. Эту возможность следует использовать если требуется добиться обновление данных и соответственно пользовательского интерфейса по таймеру.

Как правило, современные веб-приложения содержат более одной страницы с определенными элементами макета, такими как меню, логотипы, сообщения об авторских правах и т. д., которые присутствуют на всех страницах. Дублирование кода этих элементов макета на всех страницах является плохим решением, так как такое решение становится трудно поддерживать, а так же оно может вызвать рассогласованность интервейса со временем. Макеты (Layout) Blazor решают эту проблему. Каждая часть приложения в Blazor является компонентом, поэтому Layout также рассматривается как компонент в приложениях Blazor.

Макет - это компонент, который также определен при помощи шаблона Razor или кода C#, он может содержать все, что обычный компонент Blazor, в том числе привязку данных и внедрение зависимостей. Но макет Blazor включает в себя два дополнительных аспекта: наследование от BlazorLayoutComponent и ключевое слово @Body.

Компонент макета наследуется от BlazorLayoutComponent. Этот базовый класс добавляет свойство Body к компоненту, который содержит контент, отображаемый внутри макета. Разметка макета содержит ключевое слово @Body, которое во время рендеринга заменяется содержимым макета. По умолчанию Blazor использует шаблон MainLayout.cshtml в папке shared. Он работает как макет страницы.

Чтобы загрузить любой компонент Blazor в макет, вам нужно использовать директиву @layout. Когда страница-компонент запрашивается, её содержимое загружается в компонент макета в том месте, где определен тег @Body. Компилятор преобразует эту директиву в атрибут LayoutAttribute, который применяется к классу компонента. Если вы пишете компонент на чистом C# и хотите использовать макет вы можете украсить его с помощью [LayoutAttribute]. В конечном итоге это то, во что компилятор преобразует директиву @layout.

В ASP.NET MVC был добавлен файл \_ViewImports.cshtml для обеспечения механизма, позволяющего сделать директивы доступными для страниц Razor глобально, без добавления их на каждую страницу. Вы также можете использовать этот файл для определения макета в Blazor. Макет автоматически применяется ко всем страницам Blazor в иерархии папок, если вы разместите директиву подключения макета в этом файле. Файл \_ViewImports.cshtml по умолчанию содержит только одну директиву при создании нового приложения Blazor: @layout MainLayout.

В Blazor вы также можете использовать вложенные макеты, т.е. компонент Blazor может ссылаться на макет, который ссылается на другой макет. Вложенные макеты могут использоваться для отражения многоуровневой структуры меню.

Макеты уменьшают количество дублирующегося кода в вашем приложении и обеспечивают единообразие внешнего вида во всем приложении.

**4.5 Маршрутизация и инъекция зависимостей**

Маршрут - это шаблон для URL, а маршрутизация - это процесс сопоставления с шаблоном, который отслеживает запросы и определяет, что делать с каждым запросом.

Blazor предоставляет клиентский маршрутизатор, класс Microsoft.AspNetCore.Blazor.Routing.Router обеспечивает маршрутизацию в Blazor, а компонент <Router> включает маршрутизацию, и шаблон маршрута предоставляется каждому доступному компоненту. Компонент <Router> по умолчанию создан в файле App.cshtml.

В Blazor вы определяете маршруты, используя шаблоны маршрутов. Вы можете определить шаблон маршрута, добавив директиву @page в начало компонента.

В Blazor вы определяете маршруты, используя шаблоны маршрутов. Вы можете определить шаблон маршрута, добавив директиву @page в начало компонента. Также допустимо указывать несколько шаблонов маршрутов для компонента, определив несколько директив @page. Если вы определяете свой компонент как C# класс, то вы можете указать его шаблон маршрута добавив атрибут RouteAttribute. Это в конечном счете то, к чему скомпилирована директива @page. Когда файл \* .cshtml с директивой @page компилируется, сгенерированный класс получает атрибут RouteAttribute, определяющий шаблон маршрута. Во время выполнения маршрутизатор ищет классы компонентов с помощью RouteAttribute и отображает любой компонент, имеющий шаблон маршрута, который соответствует запрошенному URL-адресу.

Клиентский маршрутизатор Blazor поддерживает параметры маршрута, чтобы связать соответствующие параметры компонента и маршрута их нужно назвать одним и тем же именем (без учета регистра). Необязательные параметры пока не поддерживаются, однако подобного поведения можно достичь применив две директивы @page. Первая директива @page разрешает переход к компоненту без параметра. Вторая директива @page принимает параметр маршрута и присваивает значение свойству.

Blazor имеет поддержку инъекции зависимостей (Dependency Injection), реализованную посредством механизма ServiceProvider. Реализация аналогична таковом в ASP.NET Core, таким образом можно использовать любые доступные DI-контейнеры реализующие интерфейс IServiceProvider. Разъяснение механизмов работы ServiceProvider находиться за рамками данной работы.

Чтобы внедрить службу в компонент, используйте ключевое слово @inject. Технически, он генерирует свойство и декорирует его атрибутом InjectAttribute. Если компонентом используется базовый класс, а для базового класса также необходимы внедренные свойства, можно добавить к классу InjectAttribute вручную.

**ЗАКЛЮЧЕНИЕ**

В данной работе:

1. Рассмотрены существующие Web-технологии и произведено обозрение исследуемых (глава 1).
2. Разработано демонстрационное приложение (раздел 2.1).
3. Произведено тестирование производительности (разделы 2.2-2.5).
4. Сделан вывод о производительности Blazor (раздел 2.6).
5. Создана библиотека компонентов, готовых к использованию в новых проектах (глава 3)
6. Создано руководство по решению типовых задач SPA-приложения с использованием фреймворка Blazor (глава 4)
7. Найдены темы для дальнейших исследований (заключение).

В целом фреймворк Blazor продемонстрировал себя как целостное решение применимое для Front-end разработки. Он уступает по производительности, что может быть не приемлемо при решении некоторых задач, однако если приложение не нагружено вычислениями, то Blazor может быть использован для упрощения разработки без каких-либо потерь отзывчивости для приложения.

**Направления для дальнейших исследований**

Проведение дополнительных тестов производительности не является целесообразным на данный момент, так как все еще ведутся работы по улучшению производительности Blazor. После выхода стабильной версии можно будет провести детальные тесты производительности.

За время рассмотрения существующих библиотек компонентов была обнаружено отсутствие Blazor-библиотек для решения следующих задач:

* JWT-аутентификация на клиенте
* GRID-компонент с сортировкой, фильтрацией и пагинацией табличных данных
* Автоматические генераторы форм

Создание библиотеки компонентов, решающей данные задачи, а также типовых шаблоном для старта проекта могут быть направлениями для дальнейших исследований.

**БИБЛИОГРАФИЧЕСКИЙ СПИСОК**

# .NET Framework: [Электрон. Ресурс]. - https://ru.wikipedia.org/wiki/.NET\_Framework

# Краткий обзор технологий для Интернет-приложений: [Электрон. Ресурс]. - https://compress.ru/article.aspx?id=9825.

1. Описание API Blazor: [Электрон. Ресурс]. - <https://blazor.net/api/index.html>.
2. Примеры приложений на Blazor: [Электрон. ресурс]. - https://blazor.net/community.html.