**ВВЕДЕНИЕ**

В наше время всем людям так или иначе приходится сталкиваться с оформлением документов. Это, не говоря уже о представителях профессий, непосредственно связанных с работой с официальными бумагами, таких как секретари, юристы или же научные сотрудники.

Обычным людям наиболее часто приходится работать с документами во времена студенчества, когда появляется необходимость в оформлении рефератов, отчётов, научно-исследовательских работ и т. д. В то же время стиль оформления таких бумаг строго формализован и описан во множестве государственных стандартов, поэтому обывателям зачастую бывает сложно разобраться во всех тонкостях написания официальных документов.

В то же время на рынке программных продуктов практически отсутствуют решения данной проблемы. Немногочисленные же сервисы проверки документов на соответствие ГОСТу являются платными и недоступны большинству обучающихся.

Поэтому я поставил себе цель – разработать приложение, которое помогало бы обычным пользователям, не знакомым с государственными стандартами оформления документов, корректно оформить необходимые им бумаги.

Для достижения назначенной цели я выделил для себя следующие задачи:

* проанализировать структуру docx документов;
* спроектировать архитектуру приложения;
* реализовать прототип приложения;
* протестировать работоспособность приложения.

Объектом исследования является Open XML – формат описания docx документов, а также ГОСТ 7.32–2017, диктующий правила оформления научно-исследовательских работ.

В качестве теоретической и информационно аналитической базы была использована официальная документация к формату Open XML, официальная документация к Open XML SDK, государственные стандарты Российской Федерации диктующие правила оформления документов, а также методические указания Уральского федерального университета по оформлению научно-исследовательских работ.

1. **АНАЛИЗ СТРУКТУРЫ DOCX ДОКУМЕНТА**
   1. **Файлы входящие в состав простейшего docx документа**

Так как мною для работы был выбран формат docx как самый распространённый и удобный формат для редактирования документов, работу я начал с разбора внутренней структуры этого формата.

Расширение docx пришло на смену устаревающему формату doc в 2007 году. Основным отличием этих двух типов файлов было то, что docx, расшифровываемый как *Office Open XML Document*, базировался на Extensible Markup Language (XML), что в свою очередь давало ему ряд преимуществ, таких как [1]:

* уменьшение занимаемого файлами пространства на диске
* простота распространения
* общедоступность и открытость формата.

Внутри docx документ представляет из себя zip-архив, в состав которого входят XML файлы и встроенные изображения. При этом состав входящих в docx документ XML файлов варьируется и расширяется в зависимости от используемых в документе функций формата Office Open XML и стилей текста. Состав файлов, входящих в простейший docx документ можно увидеть на рисунке 1.

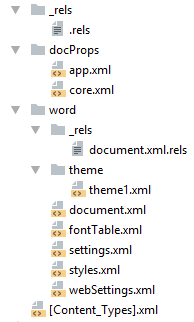


Рисунок 1 – Состав пустого docx документа.

Разберём что из себя представляет каждая из частей [2]:

* [Content\_Types].xml – XML файл, содержащийся в корне документа, хранит в себе MIME-типы содержимого документа;
* \_rels/.rels – XML файл, представляющий из себя главный список связей всего документа, в рассматриваемом простейшем docx документе приводит в соответствие идентификатор «rId1» и файл «word/document.xml», основное тело документа;
* docProps/app.xml – XML файл, содержащий общую информацию о документе: количество слов, страниц, символов а также информация о приложении в котором был создан файл;
* docProps/core.xml – XML файл, содержащий информацию об основных метаданных документа в соответствии с Open Packaging Conventions и Dublin Core;
* word/\_rels/document.xml.rels – XML файл, содержащий список связей в части word/document.xml;
* word/theme/theme1.xml – XML файл, описывающий тему документа: цветовую схему, шрифты, форматирование;
* word/document.xml – XML файл, содержащий основную разметку документа;
* word/fontTable.xml – XML файл, содержащий список используемых в документе шрифтов;
* word/settings.xml – XML файл, содержащий относящиеся к документу настройки;
* word/styles.xml – XML файл, содержащий информацию о стилях применимых к документу;
* word/webSettings.xml – XML файл, содержащий информацию о том как именно следует конвертировать документ в HTML формат.

В работе наиболее подробно будет рассматриваться файл document.xml так как он содержит исчерпывающую информацию о вёрстке документа и именно благодаря ему можно понять как будут отображаться в текстовом редакторе все структурные элементы документа.

* 1. **Структура файла document.xml**

Для анализа внутренней структуры основного файла разметки docx документа необходимо использовать текстовый редактор, имеющий возможность форматирования XML документов, так как по умолчанию Microsoft Word не проставляет никаких разделителей в XML элементах и весь текст внутри файла представлен одной строкой, что делает его нечитаемым для человека.

Я использовал программу Notepad++ – это свободно распространяемый, бесплатный текстовый редактор с открытым исходным кодом, поддерживающий подсветку синтаксиса и разметки. По умолчанию программа не поддерживает возможности автоматического форматирования XML текста, поэтому для реализации поставленной передо мной задачи был установлен плагин XML Tools, средствами которого и проводилось форматирование.

Государственные стандарты, диктующие нормы оформления документов, описывают конкретные правила для определённых элементов, содержащихся в тексте, таких как [3]:

* таблицы;
* заголовки;
* изображения;
* колонтитулы;
* абзацы.

Так же ГОСТ регламентирует вид самого шрифта, использующегося в документе, а именно:

* цвет;
* название;
* начертание;
* размер;

В изучении структуры файла document.xml я придерживался следующего алгоритма:

* создание docx документа, содержащего исследуемую часть текста;
* приведение его к виду zip архива;
* анализ изменений в структуре document.xml.

На основании полученных мною данных, я выделил следующие XML теги, на основе которых можно судить о форматировании текста и абзацев, смотри таблицу 1.

Таблица 1 – XML элементы, задающие форматирование текста и абзацев

|  |  |  |
| --- | --- | --- |
| родительский элемент | Элемент, задающий форматирование | назначение элемента. |
| <w:rPr>  …  </w:rPr> | <w:rFonts /> | содержит информацию о названии шрифта |
| <w:b /> | содержит информацию о полужирном начертании шрифта |
| <w:i /> | содержит информацию о курсивном начертании шрифта |
| <w:sz /> | содержит информацию о размере шрифта |
| <w:color /> | содержит информацию о цвете шрифта |
| <w:pPr>  …  </w:pPr> | <w:spacing /> | содержит информацию о межстрочном интервале |
| <w:ind /> | содержит информацию об абзацном отступе |
| <w:numPr>  …  </w:numPr> | содержит информацию о том является ли абзац элементом перечисления, а также хранит параметры этого перечисления |
| <w:jc /> | содержит информацию о выравнивании текста абзаца |

Как видно из таблицы 1, все элементы, задающие форматирование абзацев имеют один и тот же родительский элемент, а именно Paragraph properties, который в свою очередь является дочерним элементом тега Paragraph. Весь текст docx документа поделён на абзацы, представленные в формате Open XML элементом Paragraph.

В таблице 1 так же видно, что все элементы, задающие форматирование шрифта имеют один родительский элемент ­– Run properties. Этот элемент является дочернем элементом Run. В формате Open XML Run является дочерним элементом Paragraph, и если Paragraph представляет собой абзац, то Run – это фрагмент текста.

Форматирование таблиц, подписей к ним, а также изображений и подписей к ним можно так же определить, опираясь на описанные выше XML элементы.

Параметры полей документа, а также колонтитулов можно найти в XML элементе Section properties.

1. **ПРОЕКТИРОВАНИЕ ПРОТОТИПА ПРИЛОЖЕНИЯ**
   1. **Выбор формата взаимодействия с пользователем**

На этапе проектирования приложения необходимо было определить, как оно будет взаимодействовать с пользователем. Размышляя над форматом взаимодействия с пользователями мною были выдвинуты следующие идеи:

1. реализовать с нуля функционал для отображения docx документов с дополнительным функционалом для указания на проблемные места в документе;
2. воспользоваться интегрированным в формат Open XML инструментом рецензирования документов, представленным комментариями к конкретным частям текста.

Говоря о плюсах первой идеи, можно отметить следующее:

* наглядность, пользователь мог бы взаимодействовать с документом и видеть свои ошибки непосредственно в интерфейсе приложения;
* возможность тонкой настройки каждого комментария, реализуя функционал комментирования собственными силами можно было бы добавить возможность для каждого комментария настраивать его собственный стиль, чтобы таким образом категорировать их, дополнительно подчёркивать важность той или иной ошибки.

Однако нельзя не отметить и очевидных минусов такого подхода:

* реализация с нуля на клиентской части приложения функционала для отображения docx документов требует куда как более глубокого изучения документации для реализации всех тонкостей функционала, предоставляемого форматом Open XML;
* реализация подобного функционала является слишком затратной по времени даже для команды разработчиков, так как объём всевозможных настроек форматирования документа, предоставляемый форматом Open XML, слишком велик;
* для успешной реализации столь крупного программного продукта следовало бы прибегнуть к групповой разработке, чтобы избежать возможных ошибок проектирования и иметь возможность разделить обязанности меду разработчиками.

Рассуждая над преимуществами второй идеи, я пришёл к следующему:

* универсальность, любой текстовый редактор, поддерживающий работу с docx документами, реализует функционал по их рецензированию, так как он является частью формата Open XML;
* схожее направление разработки, реализация взаимодействия с пользователем, через встроенный механизм комментирования формата Open XML, сводится к работе с внутренней структурой docx документа [3];
* время разработки, в моём распоряжении были материалы с примерами работы со встроенным механизмом рецензирования формата Open XML, что должно было существенно ускорить разработку.

Из недостатков второго подхода можно отметить:

* малая наглядность, по результатам работы приложения пользователь получит docx документ с комментариями, но у него не будет возможности изучить их непосредственно в клиенте приложения;
* отсутствие настроек для комментариев, внешний вид рецензий, оставляемых программой будет неизменным в рамках текстового редактора, которым пользователь пользуется для открытия полученного файла.

По результатам анализа преимуществ и недостатков этих двух методов я пришёл к выводу что в рамках моей работы по созданию прототипа приложения для определения соответствия документа заданному ГОСТу следует обратиться именно ко второму способу взаимодействия с пользователем, через встроенные средства комментирования формата Open XML.

* 1. **Проектирование серверной части приложения**

Проектируя серверную часть приложения, необходимо было определится с основным функционалом приложения, архитектурой приложения и базы данных, а также выбрать язык программирования и метод разработки программного обеспечения, которой я бы следовал в процессе работы.

Размышляя над основным функционалом, я пришёл к выводу, что пользователю может потребоваться регистрация и авторизация в приложении с целью защиты его личных данных, которые могут содержаться в загруженных им документах. Так же было решено добавить возможность хранения и отображения ранее загруженных и ранее проверенных на корректность документов. Основным же функционалом приложения должна являться возможность проверки загруженного документа на соответствие стандарту, выбранному пользователем. В рамках работы по созданию прототипа приложения, мною было принято решение реализовать функционал проверки только на один стандарт, ГОСТ 7.32–2017, в связи с ограничениями по времени разработки. Use-case диаграмму, отражающую взаимодействие пользователя с приложением можно увидеть на рисунке 2.

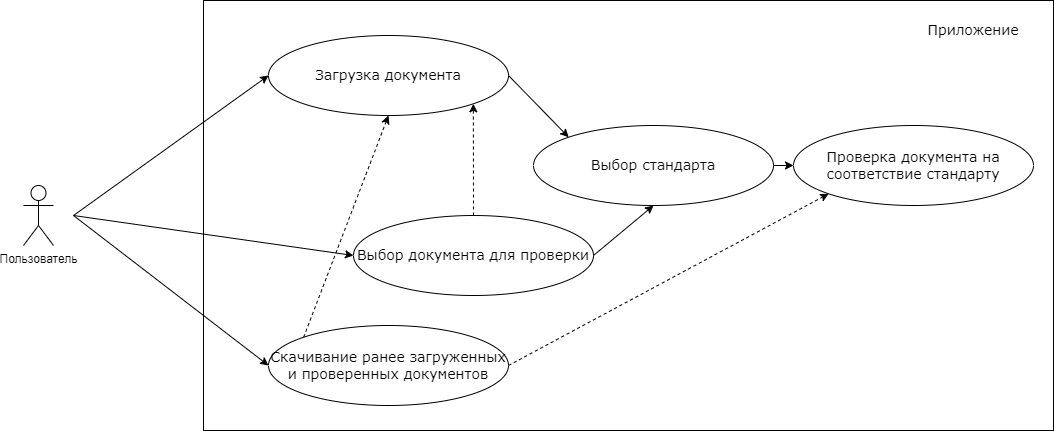


Рисунок 2 – Взаимодействие пользователя с приложением

Архитектура базы данных тесно связана с функционалом. Исходя из обозначенных ранее возможностей приложения, я выделил следующие таблицы:

* Users, таблица пользователя хранит его логин и пароль для авторизации;
* UploadedDocuments, таблица для хранения документов загруженных пользователем, ставит в соответствие путь к документу на сервере, его название и идентификатор пользователя, загрузившего документ;
* EditedDocuments, таблица для хранения документов уже проверенных программой, ставит в соответствие путь к обработанному документу на сервере, его название и идентификатор пользователя, которому принадлежит этот документ.

Описанных выше таблиц достаточно для реализации функционала, который предусматривает проектируемая серверная часть приложения.

Продумывая архитектуру приложения, я делал акцент на её расширяемости, чтобы в будущем у меня оставалась возможность добавлять новые стандарты, расширяя тем самым функционал приложения. Поэтому было важно сделать объекты в приложении слабо связными и легко заменимыми.

Одной из основных сущностей приложения должен стать стандарт, в соответствии с которым должна проводиться проверка. Для того чтобы сохранить возможность для дальнейшего расширения приложения мною было принято решения об описании интерфейса IDocumentStandard с которым и взаимодействовали компоненты приложения, вместо конкретной реализации стандарта.

IDocumentStandard должен содержать типовые методы «FindErrorIn…» которые должны проверять конкретную часть документа, отражённую в названии метода, и в случае несоответствия данной части проверяемому стандарту возвращать ошибку.

Основной функционал ошибки заключается в возможности конвертации её в комментарий формата Office Open XML. При этом для сохранения возможности внутренней настройки ошибки в зависимости от реализации, так же было принято решение выделить отдельный интерфейс IDocumentError который требовал бы от класса его реализующего наличия метода GetComment(), который возвращал бы комментарий в формате Open XML для дальнейшего добавления его в проверяемый документ. При этом мною также были выделены следующие свойства характерные для любого типа ошибок вне зависимости от их реализации, а именно:

* текст ошибки, в котором содержалась бы исчерпывающая информация о причине возникновения ошибки и давались бы рекомендации по её исправлению;
* стандарт, ГОСТ в соответствии с которым была порождена ошибка, чтобы пользователь, в случае если у него возникнут дополнительные вопросы касающиеся ошибки, мог узнать в каком именно стандарте следует искать информацию;
* подпункт стандарта, для более удобной навигации пользователя в случае необходимости изучения ГОСТа.

Так как задачей класса, реализующего IDocumentStandard, является только возможность проверки частей документа на соответствие стандарту, представляемому данным классом, то необходимо также выделить сущность, которая бы делила документ на составные части, определяла бы чем именно они являются, например заголовком, таблицей или же подписью к изображению, и отправляла на проверку в соответствующие методы IDocumentStandard, а также имела бы возможность добавлять комментарии о найденных ошибках в документ.

Сущностью, которая берёт на себя эту обязанность, стал класс Editor, реализующий два основных метода EditDocument(), проводящий покомпонентную проверку документа на соответствие стандарту и сохраняющий найденные ошибки в памяти, и AddComments(), который добавлял бы Open XML комментарии, в соответствии с найденными ошибками, в проверяемый документ.

Взаимодействие описанных выше компонентов можно проследить на рисунке 3.

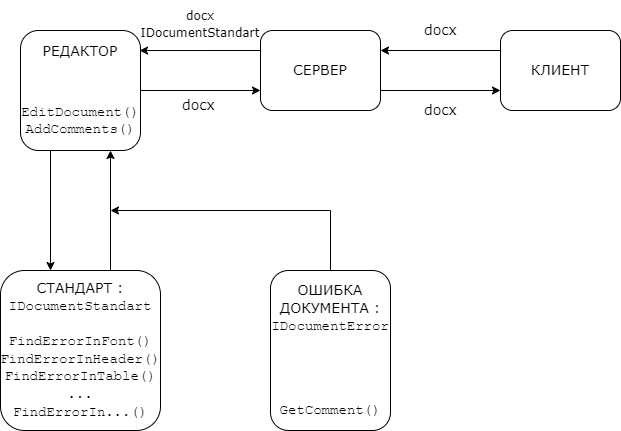


Рисунок 3 – Схема работы функции проверки

Так же для взаимодействия с базой данных следует реализовать классы, которые соответствовали сущностям, хранящимся в таблицах, они будут называться соответственно Person, UploadedDocument, EditedDocument. Набор полей этих классов должен соответствовать столбцам связанных с ними таблиц. Запросы, связанные с добавлением документов в базу загрузкой их из базы, а также регистрацией и авторизацией пользователей сервер будет обрабатывать в контроллере.

Перед началом разработки необходимо выбрать также язык программирования, на котором она будет вестись. Мне был необходим язык программирования, подходящий для написания серверных приложений, а также поддерживающий работу с Open XML SDK – набором инструментов от Microsoft, создателей формата Open XML, для работы с docx, pptx и xlsx документами. Также при выборе языка программирования я учитывал свой опыт разработки на нём.

По всем критериям подходил язык программирования С# 10. Для серверной разработки на этом языке существует официальная, поддерживаемая Microsoft библиотека ASP .NET Core 6, которая предоставляет все необходимые инструменты для комфортной и быстрой серверной разработки [4]. Также в языке C# поддерживается и работа с Open XML SDK ­– основным инструментом для программной работы с Open XML документами [5].

В качестве среды разработки приложения мною была выбрана Visual Studio Community 2022, IDE от Microsoft так как и язык программирования и основные инструменты серверной разработки и работы с документами я использовал от этой компании. Данная среда разработки поддерживает C# десятой версии, а также ASP .NET Core последней 6 версии, что и нужно мне для работы [6].

Подходя к выбору метода разработки я учитывал следующие особенности предстоящей работы:

* была понятна общая концепция задачи, однако тонкости реализации оставалось ещё выяснять;
* определение соответствия стандарту каждой отдельной категории частей документа слабо зависит от определения корректности оформления иных категорий, что позволяет их рассматривать по раздельности;
* получить первый результат нужно было как можно быстрее, для подведения выводов и прогнозирования потенциальных затрат по времени на реализацию каждой отдельной функции;
* не все элементы документа, форматирование которых предписывается ГОСТом, имеют одинаковую сложность в проверке на корректность оформления.

Проанализировав различные популярные модели разработки программного обеспечения, я пришёл к выводу что наиболее эффективной для меня станет «Incremental Model». Инкрементная модель предполагает прохождение полного цикла разработки при реализации каждой функциональной части приложения, начиная от планирования и предъявления требований к ней, заканчивая её тестированием и оценкой [7].

Я пришёл к выводу что наиболее рациональным методом разработки данного приложения станет следующий алгоритм:

* выделение части документа, оформление которой предписывается ГОСТом;
* сбор информации о требованиях, предъявляемых стандартом к выбранной части документа;
* реализация необходимого функционала проверки;
* тестирование реализованного функционала на docx файлах, содержащих ошибку в выбранных частях документа.

Повторяя данный алгоритм для всех частей документа, описываемых гостом, я приду к итоговому виду приложения.

* 1. **Проектирование клиентской части приложения**

При проектировании клиентской части приложения необходимо было определиться с его дизайном и технологиями, которые я бы применял при её разработке.

Для разработки дизайна интерфейса клиентской части приложения я воспользовался онлайн-сервисом для прототипирования и разработки интерфейсов Figma. В нём я спроектировал первоначальный вид всех окон веб-приложения таких как:

* окно регистрации;
* окно входа;
* главный экран;
* экран выбора стандарта для проверки соответствия.

Главный экран приложения вы можете видеть на рисунке 4.

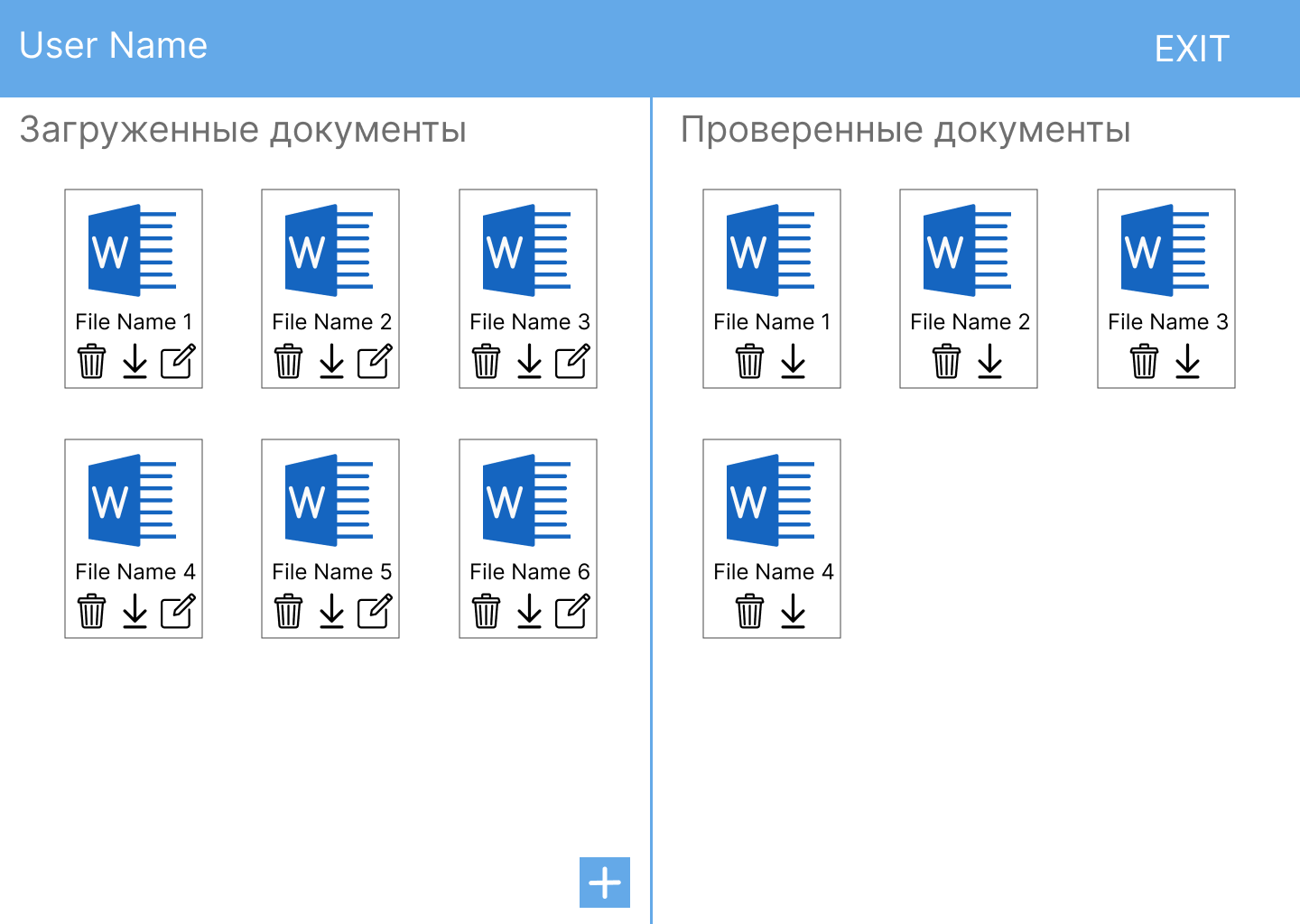


Рисунок 4 – Набросок главного экрана приложения

Так как клиентская часть приложения не реализует никакого сложного функционала мною было принято решение использовать при её разработке HTML, CSS и Java Script без каких-либо сторонних фреймворков, которые бы лишь замедлили разработку, а также негативно сказались бы на скорости отклика самого интерфейса.

1. **РАЗРАБОТКА ПРОТОТИПА ПРИЛОЖЕНИЯ**
   1. **Создание класса редактора**

После проектирования архитектуры приложения и описания всех используемых в нём интерфейсов я приступил к написанию базовых реализаций классов. Для начала работы по выявлению ошибок в различных частях документа, мне необходимо было реализовать механизм, позволяющий разделять файл на эти составные части. Так я приступил к созданию класса редактора.

Следуя построенной мною архитектуре я начал реализовывать метод EditDocument() задачей которого было деление документа на составные части, их проверка на соответствие ГОСТу и сохранение найденных ошибок. Как я отмечал ранее, в главе об анализе структуры docx документа, основным структурным элементом файла document.xml является Paragraph, именно в нём хранятся все фрагменты текста. Поэтому я принял решение разделить документ по параграфам.

Open XML SDK представляет элементы Open XML документа в качестве древовидной структуры данных [8]. Исходя из этого, чтобы получить доступ к коллекции, содержащей все элементы Paragraph, содержащиеся в документе, необходимо запросить все дочерние элементы, XML тега Body и отфильтровать из них те, которые являются абзацами. После полученную коллекцию можно поэлементно проверять в цикле.

На данном этапе, после выделения абзацев, уже можно проводить частичную проверку документа, так как каждый Paragraph хранит в себе ParagraphProperties, а значит можно получить информацию о выравнивании текста на странице междустрочном интервале отступе первой строки и т. д. Однако этого недостаточно для того, чтобы проверить все настройки форматирования документа, предписываемые стандартом.

Следующим этапом разбиения документа стало выделение из Paragrah фрагментов форматированного текста, представленного XML элементом Run. Для этого также нужно получить список всех дочерних элементов каждого абзаца и отфильтровать из них те, которые представлены XML элементом Run.

Как было описано мною ранее в главе анализа структуры docx документа, элемент Run хранить в себе RunProperties, которые являются настройками форматирования текста. Поэтому уже на этом этапе класс редактора может проверить большинство параметров задаваемых ГОСТом.

Для того чтобы сохранять ошибки найденные в процессе проверки документа мною был инициализирован словарь, ключами которого стали параграфы, в которых были обнаружены ошибки, а значениями список, хранящий в себе IDocumentError.

После реализации функционала по обнаружению ошибок я перешёл к методу AddComments(), который должен встраивать полученные из IDocumentError Open XML комментарии.

Для начала работы с open XML комментариями необходимо определить присутствуют ли они в документе на момент редактирования, так как каждый Open XML комментарий имеет свой уникальный идентификатор. В формате docx комментарии хранятся отдельно от структуры документа, в файле comments.xml, в document.xml на них оставляются только ссылки. Пример файла comments.xml вы можете увидеть на рисунке 5.

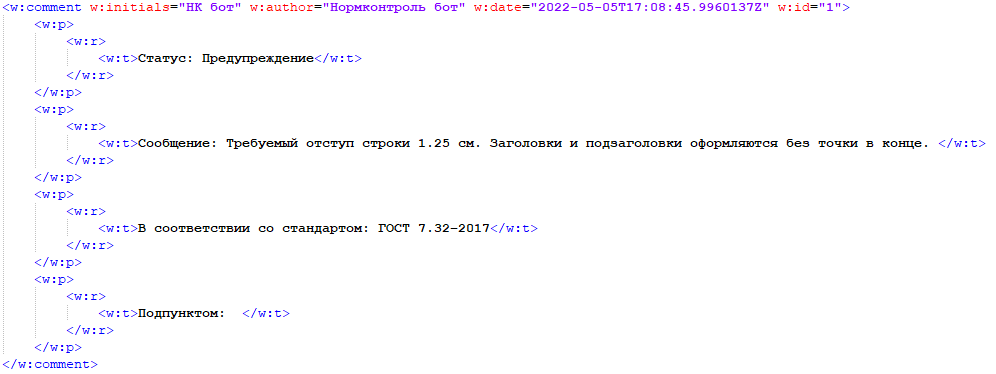


Рисунок 5 – Пример комментария из comments.xml

Метод поиска пригодного для работы идентификатора комментария выглядит следующим образом:

//Проверяется есть ли в документе комментарии

if(document.MainDocumentPart.GetPartsCountOfType<WordprocessingCommentsPart>() > 0)

{

//В случае наличия комментариев они сохраняются

comments =

document.MainDocumentPart.WordprocessingCommentsPart.Comments;

//Получается идентификатор с самым большим значением

if (comments.HasChildren)

id = comments.Descendants<Comment>()

.Select(e => e.Id.Value)

.Max();

}

else

{

//В случае отсутствия комментариев они создаются и идентификатор равен 0

WordprocessingCommentsPart commentPart = document.MainDocumentPart.AddNewPart<WordprocessingCommentsPart>();

commentPart.Comments = new Comments();

comments = commentPart.Comments;

id = 0;

}

После нахождения пригодного для использования идентификатора можно приступить к добавлению комментариев. Для этого необходимо создать новый объект класса Comment, предоставляемый Open XML SDK и являющийся представлением Open XML комментария в коде. Для этого объекта инициализировать значения идентификатора, автора комментария, инициалов автора и времени создания комментария а также добавить Paragraph содержащий текст комментария в качестве его дочернего элемента, данный функционал берёт на себя метод GetComment() интерфейса IDocumentError().

После получения объекта комментария необходимо встроить его в структуру docx документа. Для этого необходимо добавить его в качестве дочернего элемента ранее полученному объекту comments, представляющему собой comments.xml. Чтобы указать в каком именно абзаце документа была найдена комментируемая ошибка, следует создать объекты типа CommentRangeStart и CommentsRangeEnd, хранящие в себе идентификатор встраиваемого комментария, и установить их в качестве дочерних элементов параграфа, в котором были найдены ошибки. CommentsRangeStart встраивается перед первым элементом Run абзаца. CommnetsRangeEnd встраивается после последнего элемента Run абзаца.

После реализации данных методов класса Editor я приступил к разработке класса ошибки, реализующей интерфейс IDocumentError.

* 1. **Создание класса ошибки**

Главным методом класса ошибки стал GetComment(), преобразующий её в Open XML комментарий, поэтому основная работа при разработке этого класса была связана именно с реализацией этого метода.

Ошибка хранит в себе текст сообщения, стандарт в соответствии с которым проводится проверка, пункт стандарта для упрощения навигации пользователя, а также свойства приоритета ошибки и категории ошибки.

Введение свойства приоритета связано с тем, что государственный стандарт не всегда жёстко задаёт параметры оформления тех или иных частей текста, иногда даже носит скорее рекомендательный нежели запретительный характер. Исходя из этого я вывел четыре приоритета ошибок, которые можно задать при инициализации объекта данного класса, а именно:

* error, строгая ошибка, нарушение конкретных правил оформления предоставляемых стандартом, например нарушение абзацного отступа;
* warning, предупреждение, ошибка появляющаяся в случае отступления от рекомендованных ГОСТом значений, однако предусматриваемых им рамках, например использование шрифта отличного от Times New Roman;
* info, информация, используется в случае, когда определить является ли использование какого-либо форматирования нарушением правил предписываемых ГОСТом, например использование курсивного начертания текста разрешённого только для именования объектов и написания терминов;
* notAnError, приоритет показывающий, что данный объект не является по сути своей ошибкой, необходим для корректной обработки элементов документа не содержащих в себе ошибок.

Так же мною были введены категории ошибок, они необходимы так как при использовании некоторых шрифтов в текстовом редакторе, структура документа сильно искажается, в результате чего каждое отдельное слово оборачивается в элемент Run, что в свою очередь в случае возникновения ошибок в форматировании текста, приводит к созданию множества однотипных комментариев, принадлежащих одному абзацу. Категории ошибок позволяют избежать этого исключая появление схожих ошибок в рамках одного элемента Paragraph.

В реализации метода GetComment() особое внимание я уделил форматированию выводимого пользователю комментария. Несмотря на то, что возможности настройки комментариев в Open XML ограничены, можно делить выводимый текст на абзацы. Я выделил следующую структуру комментария:

* приоритет ошибки;
* текст сообщения об ошибке;
* стандарт, в соответствии с которым проходила проверка;
* подпункт этого стандарта.

Для её создания необходимо было каждое выводимое в комментарии сообщение оборачивать в XML элементы Paragraph, Run и Text.

После создания класса ошибки я перешёл к разработке класса стандарта.

* 1. **Создание класса стандарта**

В своей работе я решил использовать стандарт ГОСТ 7.32–2017 «Отчёт о научно-исследовательской работе» [9]. Этот стандарт был выбран мною по той причине, что для его тестирования можно было бы найти множество реальных документов, так как именно в соответствии с этим стандартом оформляются выпускные квалификационные работы, и у меня была бы возможность проверить корректность работы моего приложения на фрагментах работ моих сокурсников. Также для проверки некоторых элементов документа я обращался к книге «Методические рекомендации для подготовки и оформления выпускных квалификационных работ», кафедры интеллектуальных информационных технологий Уральского федерального университета, для дополнения критериев оформления, задаваемых ГОСТом [10].

Начал я с реализации метода по выявлению ошибок в начертании шрифтов. Я выделил следующие требования ГОСТ 7.32–2017 предъявляемые к начертанию текста [9]:

* полужирным шрифтом выделяются только заголовки разделов и подразделов;
* курсивный шрифт используется только в написании объектов и терминов.

Для определения используется ли в проверяемом фрагменте текста полужирное или курсивное начертание необходимо проверить содержит ли исследуемый элемент RunProperties среди своих дочерних элементов Bold или Italic. В случае если какое-либо из начертаний используется, то в текст ошибки добавляется информация об этом. После чего формируется объект класса ошибки, ей в конструкторе передаётся текст будущего комментария, приоритет Info, а также подпункт стандарта, в котором описаны данные требования.

После тестирования работы описанного функционала на примере документа, содержащего данные ошибки, я приступил к реализации следующих методов.

Для реализации метода определения корректности выбранного шрифта я рассматривал Open XML элемент RunFonts. Он содержит в себе информацию о названии шрифта используемого для вывода фрагмента текста. ГОСТ 7.32–2017 рекомендует использовать при написании отчётов шрифт Times New Roman [9]. Поэтому необходимо проверить что в свойстве «ascii» элемента RunFonts хранится именно это значение, в ином случае необходимо отправить ошибку с приоритетом Warning.

Для проверки размера шрифта следует проверить свойство value элемента FontSize. ГОСТ предписывает использовать шрифт не менее 12 пт [9]. Внутренними единицами измерения формата Open XML являются точки. На основании исследования структуры document.xml я выяснил, что точки соотносятся с пт в отношении два к одному, значит искомое значение value 24. В Open XML SDK все значения, полученные из структуры документа, хранятся в строках, поэтому для сравнения необходимо перевести их в целые числа. В случае если свойство value исследуемого фрагмента текста меньше 24, следует вернуть ошибку с приоритетом Error.

Для проверки цвета шрифта необходимо обратиться к свойству value элемента Color. ГОСТ предписывает использовать шрифт только чёрного цвета [9]. Чёрный цвет в формате Open XML задаётся как «000000», а также является цветом по умолчанию, поэтому для проверки необходимо узнать присутствует ли элемент Color в RunProperties, если нет, то значит цвет задан по умолчанию, если присутствует, то необходимо проверить чтобы значение его свойства value было рано «000000».

Для поиска ошибок в оформлении абзацев необходимо обратиться к следующим XML элементам:

* SpacingBetweenLines, отвечает за междустрочный интервал;
* Indentation, хранит информацию об абзацных отступах.

Данные XML элементы не всегда присутствуют в ParagraphProperties, при этом их значения по умолчанию отличаются от предписываемых ГОСТом. Поэтому помимо проверки их значений на соответствие ГОСТу необходимо также проверять наличие этих элементов, их отсутствие эквивалентно ошибке. В случае если элемент SpacingBetweenLines присутствует в ParagraphProperties, необходимо проверить значение его свойства Line. ГОСТ 7.32–2017 предписывает, что междустрочный интервал должен равняться полутора пунктам, что задаётся в формате Open XML как значение Line равное 360 [9]. Если элемент Indentation присутствует необходимо проверить значение его свойства FirstLine, в котором содержится информация об абзацном отступе первой строки. Стандарт устанавливает значение отступа равным 1,25 см, что в формате Open XML выражается как значение свойства FirstLine равное 709 [9]. В случае нахождения несоответствий с ГОСТом, данным методом формируется ошибка с приоритетом Error.

Для поиска более сложных ошибок форматирования, например в списках, таблицах и заголовка, необходимо было добавить функционал распознавания данных элементов в структуре документа классом редактора. Так в результате анализа структуры document.xml я пришёл к следующим выводам:

* таблица задаётся в документе элементом Table, хранящим внутри себя элементы TableRow, представляющие строки таблицы, дочерними элементами TableRow являются XML тэги TableCell, описывающими ячейки таблицы, и уже в них в свою очередь содержатся обычные элементы Paragraph;
* перечисление в документе можно определить по наличию в свойствах абзаца элемента NumberingProperties;
* изображение задаётся элементом Drawing;
* информация о полях документа храниться в XML элементе SectionProperties;
* заголовки и подзаголовки можно охарактеризовать как элементы перечисления, выделенные полужирным начертанием, поэтому для их поиска в документе нужно обратиться к элементам NumeringProperties и Bold.

На основании этих данных я добавил в класс редактора функционал по определению вышеописанных частей документа для корректной передачи их в класс стандарта.

Для поиска ошибок в форматировании полей я обратился к элементу SectionProperties. В его дочернем элементе PageMargin находятся свойства left, right, top, bottom, в которых хранятся значения левого, правого, нижнего и верхнего полей соответственно. ГОСТ 7.32–2017 предписывает использовать для полей следующие значения [9]:

* левое – 3 см;
* правое 1,5 см;
* нижнее – 2 см;
* верхнее – 2см.

В единицах измерения формата Open XML данные размеры преобразуются в 1700 для левого поля, 850 для правого, 1134 для верхнего и нижнего. Необходимо проверить равны ли значения свойств элемента PageMargin данным значениям и в если нет вернуть ошибку с приоритетом Error.

В процессе тестирования данного метода на примере реального документа выяснилось, что данные значения в формате Open XML не всегда равны для одних и тех же полей. Правое поле равное 1,5 см заданное в текстовом редакторе Microsoft Word по умолчанию преобразуется в 850 точек, однако если задать те же значения полей вручную и сохранить документ, его значение изменится на 851. Я полагаю данная погрешность связана с тем, что поле задаётся не в целых числах. Для компенсации этого факта я начал сравнивать поля документа с припуском в пять точек.

В методе поиска ошибок в перечислениях я обратился к элементу Text абзаца, распознанного как элемент перечисления. ГОСТ предписывает чтобы каждый элемент простого перечисления оканчивался на запятую, а каждый элемент сложного (многоуровневого) перечисления оканчивался на точку с запятой [9]. Поэтому я проверил чтобы свойство Text элемента Text оканчивалось на данные символы. В ином случае отправляется ошибка с приоритетом Info.

Для поиска ошибок в оформлении изображений необходимо было обратиться к абзацам предшествующему изображению и следующим за пояснением к иллюстрации, так как они не должны были содержать текста. Для проверки корректности оформления подписи к иллюстрации я обращался к элементу ParagraphProperties абзаца следующего за элементом Drawing. Стандарт предписывает что подпись к иллюстрации должна быть расположена в следующем за ней абзаце, по центру документа и не должна иметь точки в конце [9]. Для определения положения абзаца на странице следует обратиться к элементу Justification и его свойству value принимающему следующие значения:

* both, текст располагается по ширине страницы;
* left, текст выравнивается по левому краю;
* right, текст выравнивается по правому краю;
* center, выравнивание текста происходит по центру страницы.

Необходимо было проверить чтобы значение было равно center. Для проверки оканчивается ли подпись к иллюстрации на точку нужно обратиться к элементу Text содержащемуся в элементе Paragraph подписи. Требование стандарта по использовании в подписи к иллюстрации шрифта равного 14 пт можно проверить через элемент FontSize, его свойство value должно равняться 28. Если несоответствия были найдены, то метод возвращает ошибку с приоритетом Error.

При проверке оформления таблиц необходимо обратить внимание не только на содержимое таблицы, но и на оформление её подписи, поэтому мною было принято решение разделить эти две проверки на разные методы.

Проверяя на соответствие стандарту содержимое таблицы, особое внимание следовало обратить на её структуру. Стандарт требует писать название столбцов и строк с прописной буквы и без точек в конце [9]. Для проверки того, что проверяемая ячейка является заголовком столбца следует обратиться к её родительскому элементу TableRow и убедится, что ему не предшествуют другие строки таблицы. Для определения того является ли ячейка заголовком строки необходимо проверить наличие предшествующих ей элементов TableCell в рамках рассматриваемой строки. В случае если были найдены несоответствия со стандартом метод отправляет ошибку с приоритетом Error.

Название таблицы, согласно стандарту, должно отвечать следующим требованиям [9]:

* располагаться сверху слева над таблицей, в предшествующем ей абзаце;
* не должно оканчиваться на точку;
* не должно иметь абзацного отступа.

Для проверки того располагается ли название слева, необходимо проверить свойство Value элемента Justification абзаца названия. Оно должно иметь значение left. Чтобы узнать отсутствует ли абзацный отступ у названия таблицы необходимо обратиться к элементу Indentation и, если он присутствует, проверить что его свойство Value имеет значение 0. Проверка на то оканчивается ли название таблицы точкой производится путём обращение к элементу Text проверяемого абзаца и проверки его последнего символа. В случае если были найдены несоответствия метод возвращает ошибку с приоритетом Warning.

При проверке корректности оформления заголовков и подзаголовков необходимо проверить следующее [9][10]:

* что они выравнены по левому краю;
* выделены полужирным начертанием;
* не оканчиваются точкой;
* что им предшествует пустая строка
* что за ними следует пустая строка;
* что междустрочный интервал равен 1;
* что они написаны с абзацного отступа.

Для проверки выравнивания необходимо убедиться, что значение свойства Value элемента Justification равно left. Для проверки абзацного отступа необходимо убедиться, что значение свойства FirstLine элемента Indentation равно 709. Для проверки полужирного начертания необходимо выяснить является ли элемент Bold дочерним элементом проверяемого абзаца. Междустрочный интервал проверяется через элемент SpacingBetweenLines и его свойство Line значение которого должно равняться 240. Проверка наличия пустых строк, предшествующих заголовку и следующих за ним производится путём проверки наличия текста в соответствующих абзацах. Проверка на то, что заголовок не оканчивается точкой производится путём проверки, что свойство Text элемента Text не оканчивается точкой. В случае нахождения несоответствий метод возвращает ошибку с приоритетом Error.

* 1. **Разработка серверных контроллеров**

Обязанность по обработке запросов с клиента на сервере лежит на классах контроллерах. Так как в разработке серверной части приложения я использую ASP .NET Core 6, то контроллер представляет из себя C# класс, которому при поступлении запроса на сервер система маршрутизации передаёт данные запроса, после чего контроллер проводит над ними необходимые операции и отсылает результат своей работы обратно [11].

В своей серверной архитектуре я выделил два основных класса контроллера:

* AccountController, занимающийся обработкой запросов связанных с учётной записью пользователя;
* DocumentController, занимающийся обработкой запросов связанных с работой с документами.

В AccountController я реализовал методы для регистрации нового пользователя, авторизации пользователя, а также метод для удаления учётной записи пользователя.

Так как приложению нет необходимость хранить о пользователе какие-либо дополнительные данные при регистрации используется только логин и пароль. Они сохраняются в базу данных, чтобы в дальнейшем можно было авторизовать уже зарегистрированного пользователя.

При разработке контроллеров я использовал библиотеку MediatR, позволяющую отделить логику обработки запроса клиента, от логики приёма запроса и отправки ответа. В моей архитектуре контроллер отвечает только за приём запроса, переправку его параметров соответствующему классу обработчику и возврат полученного от обработчика ответа.

Для корректной работы класса обработчика, реализующего интерфейс IRequestHandler<TRequest, TResponse> необходимо создать класс, представляющий запрос, который должен реализовать интерфейс IRequest<TResult>. В классе запроса для регистрации нового пользователя достаточно определить два свойства Login и Password, оба из которых являются строками.

Классу обработчику в конструкторе передаётся контекст данных, унаследованный от стандартного класса библиотеки EntityFramework – DbContext, для получения возможности работы с данными в базе. После чего обработчик добавляет в базу данные нового пользователя, полученные из класса запроса, и сохраняет изменения. В случае если данные были добавлены успешно сервер вернёт ответ с кодом 200, если же выполнить добавление нового пользователя не удастся будет возвращён ответ с кодом 400.

Авторизация пользователей на сервере производится при помощи JSON Web Tokens. При отправке запроса по авторизации на сервер, будет проведена проверка существует ли такой пользователь в базе данных, и в том случае если он существует, будет сгенерирован специальный ключ, который будет передан клиенту [12]. В последствии клиент должен будет сохранить этот ключ и отправлять его в HTTP заголовке Authorization. Так сервер будет распознавать авторизированных пользователей.

В классе запроса для метода авторизации также необходимо реализовать два свойства Login и Password по которым будет проходить поиск пользователя в базе данных.

Также в этом случае появилась необходимость в создании класса ответа, в котором необходимо реализовать следующие свойства:

* Id, для передачи идентификатора пользователя на клиент, необходимо для работы с документами;
* Login, для отображения логина пользователя на клиенте в соответствии с набросками сайта;
* Token, непосредственно сам ключ для авторизации пользователя.

Обработчик запроса принимает данные о пользователе, проверяет их наличие в базе данных, после чего генерирует ключ инициализирует объект класса ответа передаёт ему данные об идентификаторе, логине и ключе, и отправляет соответствующему методу контроллера. Контроллер в свою очередь отправляет ответ в формате JSON на клиент.

В классе DocumentController реализован весь необходимый функционал для работы с документами, а именно:

* добавление нового документа в базу данных;
* получение всех документов, загруженных пользователем;
* удаление загруженного документа из базы данных;
* проверка загруженного документа на соответствие стандарту;
* получение всех проверенных документов пользователя;
* удаление проверенного документа из базы данных.

Так как в спроектированной мною архитектуре документы хранятся непосредственно в файловой системе сервера сохранение документов, полученных с клиента, производится в директорию upload.

Для реализации класса обработчика загрузки документов необходимо реализовать класс запроса, который будет содержать свойство Form с типом данных IFormCollection, которое представляет из себя данные с формы загрузки файла.

В конструктор обработчика запроса передаётся контекст данных, для работы с базой. Метод извлекает коллекцию файлов из формы, переданной в запросе, после чего поэлементно сохраняет файлы в директорию upload и добавляет запись с именем файла, путём до него на сервере и идентификатором пользователя, загрузившего файл в базу данных. В случае успешного добавления данных в базу отправляется HTTP ответ с кодом 200, в случае если данные добавить не удалось с кодом 400.

Класс запроса на получение документов пользователя будет содержать только одно свойство Id с идентификатором пользователя. Обработчик запроса получает информацию обо всех документах, загруженных пользователем, после чего передаёт полученную коллекцию документов на клиент.

Класс запроса на удаление документа из базы данных содержит одно свойство Id, представляющее идентификатор документа для удаления. Класс обработчика запроса на удаление получает доступ к базе данных, к таблице с документами, удаляет запись с соответствующим идентификатором и сохраняет изменения. В случае если удалось удалить запись из базы сервер вернёт HTTP ответ с кодом 200, в ином случае с кодом 400.

Класс запроса на проверку документа соответствующему стандарту содержит три свойства Id, целое число, представляющее идентификатор пользователя, DocumentId, целое число, представляющее идентификатор документа для проверки, и Standard, строка, представляющая название стандарта, в соответствии с которым будет проводиться проверка.

Класс обработчика запроса на проверку стандарта инициализирует объект класса стандарта, в соответствии с которым будет проводиться проверка, инициализирует объект класса редактора, передав ему в конструкторе стандарт и копию выбранного для обработки документа, сохранённую в директорию edited. После чего вызывает методы редактора EditDocument() и AddComments(). Добавляет в базу, в таблицу с проверенными документами, запись о названии проверенного документа, путь к нему на сервере и идентификатор пользователя, которому принадлежит документ. В случае если удалось добавить запись в базу сервер вернёт HTTP ответ с кодом 200, в ином случае с кодом 400.

Класс запроса на получение проверенных документов пользователя содержит свойство Id, целое число, представляющее идентификатор пользователя. Обработчик запроса получает информацию обо всех проверенных документах, принадлежащих пользователю, после чего передаёт полученную коллекцию документов на клиент.

Класс запроса на удаление проверенного документа из базы данных содержит Id, целое число, представляющее идентификатор проверенного документа для удаления. Класс обработчика запроса на удаление получает доступ к базе данных, к таблице с проверенными документами, удаляет запись с соответствующим идентификатором и сохраняет изменения. В случае если удалось удалить запись из базы сервер вернёт HTTP ответ с кодом 200, в ином случае с кодом 400.

В результате работы были созданы два API контроллера, позволяющие обмениваться запросами с клиентом. После их реализации я приступил к разработке клиента приложения.

* 1. **Разработка клиента приложения**

Разработка клиентской части приложения заключалась в вёрстке HTML страниц, настройке стилей для них, а также в настройке отправки запросов и корректного отображения полученных результатов посредством Java Script.

Я создал три HTML страницы:

* страницу регистрации пользователя;
* страницу авторизации пользователя;
* главную страницу приложения;

При входе на сайт пользователь попадает на главную страницу приложения. Для работы ему необходимо авторизоваться на сайте. При нажатии на кнопку регистрации происходит перенаправление на страницу регистрации. На ней пользователь вводит логин и пароль, который он в дальнейшем будет использовать для входа на сайт. После регистрации пользователя перенаправляет на страницу авторизации, где он после ввода корректного логина и пароля перенаправляется обратно на главную страницу сайта и может начинать работу.

Все страницы были созданы в соответствии с разработанными на этапе проектирования макетами.

При написании Java Script кода было необходимо реализовать механизм отправки запросов на сервер. Запросы на сервер отправляются при помощи метода fetch (url, [options]) [13]. Он принимает в себя в качестве параметров url – адрес, на который будет отправлен запрос, а также [options] – параметры запроса в формате JSON, содержащие в себе HTTP метод, заголовки и тело запроса. Fetch по результатам своей работы возвращает ответ сервера, который можно преобразовать в JSON формат для дальнейшей работы с ним.

Для реализации регистрации пользователя я считывал данные из тегов input в которые пользователь вводил свои логин и пароль, формировал на их основе данные формы, и отправлял на сервер с запросом на регистрацию, в качестве метода использовал POST, в теле запроса передавал сформированные данные формы. В случае получения от сервера кода 200 производил перенаправление на страницу авторизации пользователя.

Для реализации регистрации пользователя также считываются данные из тегов input, на их основе формируются данные формы и отправляются на сервер. В качестве параметров fetch указывается метод POST, и тело запроса – данные формы. Получив ответ от сервера, я сохраняю ключ для авторизации в веб-хранилище sessionStorge, после чего провожу перенаправление на главную страницу сайта.

При формировании содержимого главной страницы через Java Script подаются запросы на сервер на получение загруженных пользователем документов и обработанных документов пользователя. Для отправки запросов используется метод fetch без параметров, идентификатор пользователя передаётся непосредственно в url. После получения ответа от сервера поэлементно для каждого документа формирует свой блок для отображения на странице.

На главной странице при нажатии на кнопку загрузки нового документа происходит открытие дополнительного окна, в котором пользователь выбирает документ, который он хочет загрузить на сервер и подтверждает загрузку. Метод загрузки файла на сервер при помощи fetch создаёт запрос на сервер, в параметрах запроса передаётся метод POST и тело запроса – загружаемый файл.

Для удаления документов Java Script получает данные об идентификаторе документа, который нужно удалить, и методом fetch отправляет DELETE запрос на сервер. Идентификатор документа передаётся в url.

После реализации клиентской части я преступил к тестированию общей работоспособности приложения.

* 1. **Тестирование приложения**

Тестирование сайта я проводил при непосредственной работе с ним. Все элементы отображались корректно, никаких наложений или смещений элементов замечено не было. Все перенаправления работали в соответствии с описанным ранее поведением. Загрузка и скачивание документов также работали корректно без каких-либо проблем.

По результатам тестирования функционала редактирования документов мною так же не было замечено никаких отклонений или ошибок в работе обозначенного ранее функционала редактора. Пример отредактированного документа вы можете видеть на рисунке 6.

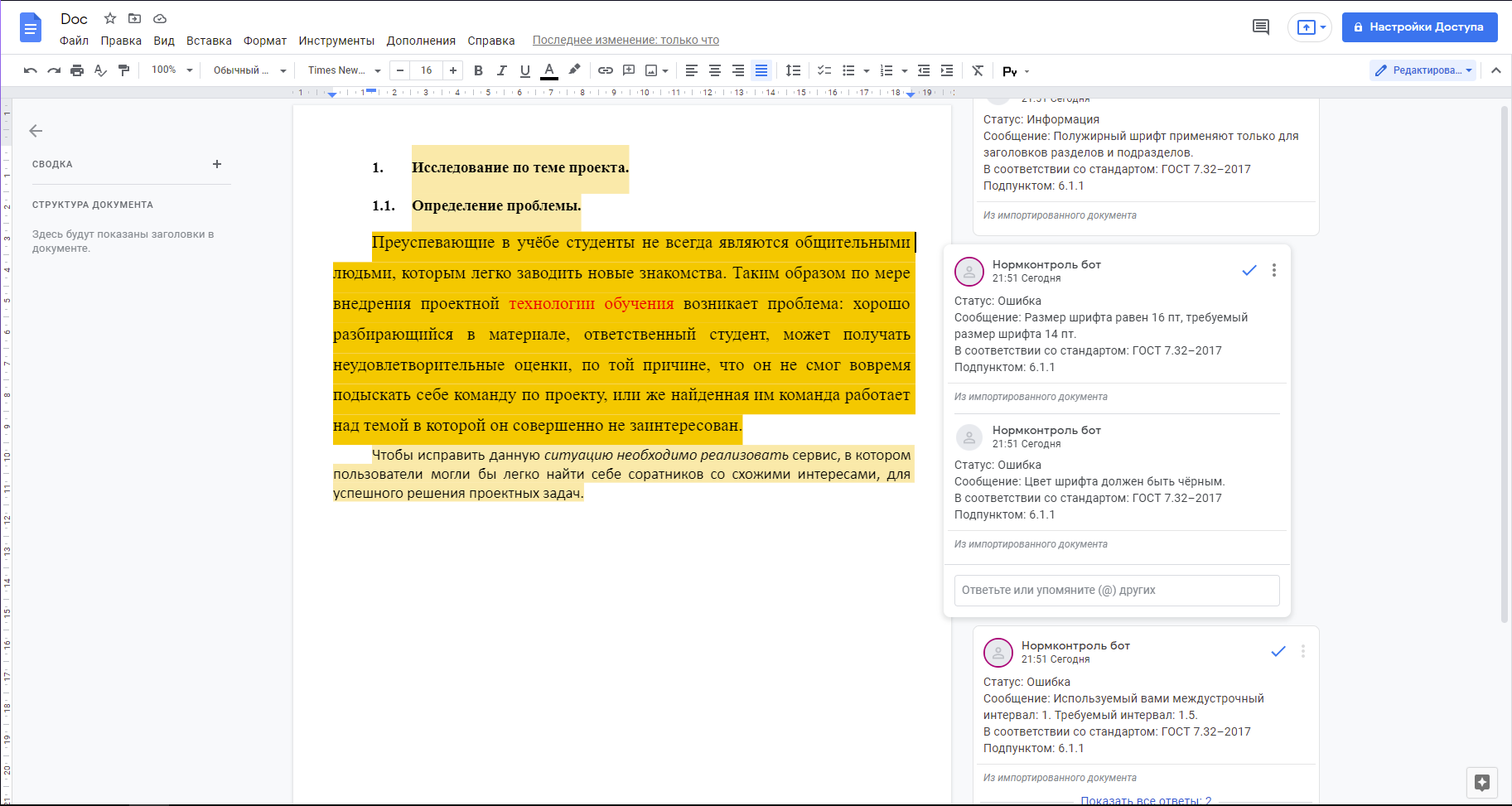


Рисунок 6 – Результат работы программы

Для тестирования сайта мною использовался браузер Opera GX, для просмотра отредактированных документов я использовал веб-программу Google Документы.

**ЗАКЛЮЧЕНИЕ**

Залью данной работы была разработка приложения для определения соответствия документа заданному ГОСТу. Для этого были осуществлены следующие этапы работы:

1. Произведён анализ структуры docx документов, а также структуры основного файла разметки docx документов document.xml. На основании полученных данных были выделены основные структурные элементы docx документов, а также числовые значения свойств соответствующие предписываемым стандартом правилам форматирования документов.
2. Была спроектирована архитектура приложения и базы данных, а также выбран метод разработки программного обеспечения и созданы макеты клиентской части приложения. Были описаны сущности базы данных, основные сущности программы, необходимые для реализации интерфейсы и классы.
3. В соответствии с проектом был создан прототип приложения. Был реализован функционал редактирования документов в соответствии со стандартом ГОСТ 7.32–2017, функционал авторизации пользователей, хранения и работы с загруженными и обработанными документами.
4. Проведено тестирование работоспособности приложения. Протестированы все заявленный функции сайта и редактора документов.

По результатам работы я получил углублённые знания внутренней структуры docx документов, формата Office Open XML Documents. Закрепил знания в проектировании и программировании серверной части приложения. Закрепил знания в вёрстке HTML страниц и отправки запросов на сервер с помощью Java Script. Получил новый опыт в создании макетов страниц сайта.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Difference Between DOC and DOCX: Which Should You Use? [Электронный ресурс]: https://www.folderit.com/blog/difference-between-doc-and-docx-which-should-you-use/ – (дата обращения: 13.05.2022).
2. ECMA-376. Fundamentals And Markup Language Reference [Текст]: дата введения декабрь 2006 г. – М.: Rue du Rhône 114: Ecma International, 2016. – 5029 с.
3. ГОСТ Р 7.0.97–2016. Национальный стандарт Российской Федерации. Система стандартов по информации, библиотечному и издательскому делу. Организационно-распорядительная документация. Требования к оформлению документов [Текст]: дата введения 01.07.2018. – М.: Госстандарт России: Стандартинформ, 2018. – 32 с.
4. Новые возможности в ASP.NET Core 6.0 [Электронный ресурс]: https://docs.microsoft.com/ru-ru/aspnet/core/release-notes/aspnetcore-6.0 – (дата обращения 13.05.2022).
5. Welcome to the Open XML SDK 2.5 for Office [Электронный ресурс]: https://docs.microsoft.com/en-gb/office/open-xml/open-xml-sdk – (дата обращения 13.05.2022).
6. What's new in Visual Studio 2022 [Электронный ресурс]: https://docs.microsoft.com/en-us/visualstudio/ide/whats-new-visual-studio-2022 – (дата обращения 13.05.2022).
7. Ещё раз про семь основных методологий разработки [Электронный ресурс]: https://habr.com/ru/company/edison/blog/269789/ – (дата обращения 13.05.2022).
8. OFFICE OPEN XML OVERVIEW [Электронный ресурс]: https://www.ecma-international.org/wp-content/uploads/Office-Open-XML-White-Paper.pdf – (дата обращения 13.05.2022).
9. ГОСТ 7.32–2017. Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления [Текст]: дата введения 01.07.2018 – М.: Госстандарт России: Стандартинформ, 2018. – 33 с.
10. Алферьева Т. И., Васина В. Н., Шадрин Д. Б. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ДЛЯ ПОДГОТОВКИ И ОФОРМЛЕНИЯ ВЫПУСКНЫХ КВАЛИФИКАЦИОННЫХ РАБОТ [Электронный ресурс]: Подготовлено кафедрой интеллектуальных информационных технологий / Т. И. Алферьева, В. Н. Васина, Д. Б. Шадрин. – Электронный текстовый ресурс. – Екатеринбург, 2019. – 1 электрон. текст. док. – Загл. с экрана.
11. Контроллеры [Электронный ресурс]: https://metanit.com/sharp/aspnet5/5.1.php – (дата обращения 13.05.2022).
12. Авторизация с помощью JWT-токенов [Электронный ресурс]: https://metanit.com/sharp/aspnet5/23.7.php – (дата обращения 13.05.2022).
13. Fetch [Электронный ресурс]: https://learn.javascript.ru/fetch – (дата обращения 13.05.2022).