Содержание

[Содержание 1](#_Toc135068270)

[Реферат 3](#_Toc135068271)

[Abstract 4](#_Toc135068272)

[Введение 5](#_Toc135068273)

[2 Обзор литературы и аналогов 6](#_Toc135068274)

[2.1 Требования к приложению 6](#_Toc135068275)

[2.2 Аналогичные программные решения 6](#_Toc135068276)

[2.3 Патентный поиск 6](#_Toc135068277)

[2.4 Обоснование технических приемов программирования 6](#_Toc135068278)

[2.5 Выводы по разделу 6](#_Toc135068279)

[3 Проектирование программного обеспечения 7](#_Toc135068280)

[3.1 Архитектура приложения 8](#_Toc135068281)

[3.2 Диаграмма вариантов использования 8](#_Toc135068282)

[3.3 Проектирование клиентской части (обоснование используемых технологий) 8](#_Toc135068283)

[*3.3.1 React* 8](#_Toc135068284)

[3.3.2 Redux 9](#_Toc135068285)

[3.3.3 Webpack 10](#_Toc135068286)

[3.3.4 Архитектура клиентской части 10](#_Toc135068287)

[3.4 Проектирование серверной части 11](#_Toc135068288)

[3.4.1 Процесс проверки пользовательской цели 12](#_Toc135068289)

[3.5 Проектирование базы данных 13](#_Toc135068290)

[3.6 Выводы по разделу 18](#_Toc135068291)

[4 Разработка программного средства 19](#_Toc135068292)

[4.1 Серверная часть приложения 19](#_Toc135068293)

[4.1.1 Структура проекта 19](#_Toc135068294)

[4.1.2 Конфигурация и запуск проекта 21](#_Toc135068295)

[4.1.3 Уровень доступа к данным 24](#_Toc135068296)

[4.1.4 Уровень представления 26](#_Toc135068297)

[4.1.5 Уровень бизнес-логики 29](#_Toc135068298)

[4.1.6 Класс фоновых работ 30](#_Toc135068299)

[4.2 Клиентская часть приложения 31](#_Toc135068300)

[4.2.1 Структура проекта 31](#_Toc135068301)

[4.2.2 Настройка Webpack 34](#_Toc135068302)

[4.2.3 Разработка компонентов 34](#_Toc135068303)

[4.2.4 Менеджер состояний Redux 36](#_Toc135068304)

[4.3 Добавление стилей 36](#_Toc135068305)

[4.4 Выводы по разделу 37](#_Toc135068306)

[5 Тестирование приложения 38](#_Toc135068307)

[6 Руководство пользователя 38](#_Toc135068308)

[7 Экономическое обоснование проекта 38](#_Toc135068309)

[Заключение 39](#_Toc135068310)

[Список использованных источников 40](#_Toc135068311)

[Приложение А 41](#_Toc135068312)

[Приложение Б 42](#_Toc135068313)

[Приложение В 43](#_Toc135068314)

[Приложение Г 44](#_Toc135068315)

[Приложение Д 45](#_Toc135068316)

Реферат

Abstract

Введение

В наше время экономическая стабильность является одним из ключевых приоритетов для многих людей. Эффективное управление финансами и правильное распределение денежных средств могут существенно повлиять на качество жизни и достижение финансовых целей. С развитием технологий возрос интерес к различным инструментам и приложениям, которые могут помочь в сохранении денег и управлении финансами.

Целью дипломного проекта является разработка веб-приложения «Better save», которое предоставляет пользователям инструменты для контроля над своими финансами, такие как календарь долгов, конструктор финансовых целей, аналитика о совершенных операциях.

Для достижения цели были поставлены следующие задачи:

* выбор и обоснование средств разработки;
* исследовать преимущества и недостатки аналогичных приложений;
* составить набор требований к разрабатываемому программному средству;
* спроектировать архитектуру приложения и базы данных;
* разработать серверную и клиентскую части приложения;
* провести тестирование работоспособности приложения;
* написать руководство пользователя.

Таким образом разрабатываемое приложение должно предоставляться пользователю в виде набора страниц, просматриваемых в браузере, между которыми присутствует возможность перехода. Веб-приложение должно взаимодействовать с централизованной базой данных, обеспечивающей хранение пользовательской информации. Само взаимодействие между клиентской и серверной частью должно базироваться на клиент-серверной архитектуре, пользователь должен иметь доступ к приложению через протокол HTTPS.

1. Обзор литературы и аналогов
   1. Требования к приложению
   2. Аналогичные программные решения
   3. Патентный поиск
   4. Обоснование технических приемов программирования
   5. Выводы по разделу

По результатам исследования реализаций аналогов создаваемой социальной сети, анализа литературы и патентного поиска был обнаружено, что на данном этапе не существует программного средства, которое реализовало бы все возможности разрабатываемого приложения, а именно:

* наличие социальной сети кулинарных рецептов для браузеров;
* быстрый поиск по рецептам;
* общение с другими пользователями в режиме реального времени;
* удобство интерфейса;
* защита персональных данных пользователей.

Сводя всё вышесказанное воедино, можно утверждать, что проведенная аналитическая работа в обзоре литературы является основополагающим фактором для создания качественного программного обеспечения. Этот этап работы дает нам возможность перейти к рассмотрению более конкретных технических приемов программирования, которые являются ключевыми для разработки программного продукта.

1. Проектирование программного обеспечения

Перед тем, как приступать к проектированию программного средства, необходимо провести анализ всех поставленных задач и возможных вариантов его использования. Для этой цели следует разработать диаграмму вариантов использования, которая станет основой дальнейшей работы над приложением. Пример такой диаграммы можно увидеть в приложении Д научной работы.

Разрабатываемое клиент-серверное приложение обладает несколькими преимуществами, такими как доступность через браузер на любом устройстве с интернет-соединением, кроссплатформенность и совместимость. Роль пользовательского интерфейса выполняется клиентской частью, которая отображает информацию и предоставляет пользователю возможность взаимодействия с ней.

Клиентская часть разработана с использованием библиотеки React. Эта библиотека позволяет одностраничные приложения. Клиентская часть разрабатываемого приложения будет обмениваться данными с сервером незаметно для пользователя: без перезагрузки страницы или временной остановки возможности использования приложения. В качестве библиотеки, на которой будет основана вся остальная разработка клиентской части, была выбрана именно React, потому что это современная, быстро развивающаяся библиотека, имеющая открытый исходный код, поддерживаемая компанией Meta, использующая язык TypeScript, который в свою очередь является надмножеством языка JavaScript. В отличии от других библиотек и фреймворков, в React используется технология Virtual DOM, использование которой позволяет оптимизировать производительность приложения и ускорить работу с интерфейсом, особенно в случае больших и сложных приложений.

Серверная часть представляет собой ASP.NET Core Web API приложение. Web API это фреймворк для создания веб-сервисов, которые могут общаться через HTTP-протокол. Он позволяет создавать RESTful веб-сервисы, которые могут обмениваться данными с клиентскими приложениями или другими серверами программного обеспечения. Серверная часть приложения разделена на три слоя: PL(Presentation Layer), DAL (Data Access Layer), BL (Business layer). Общая идея слоистой архитектуры заключается в том, чтобы разделить приложение на набор слоев, каждый из которых решает определенную задачу и имеет свои собственные обязанности. Это позволяет легче понимать и изменять код, уменьшает связность между разными частями приложения и упрощает тестирование и развертку приложения.

Для хранения информации пользователя используется система управления базами данных MSSQL. Для взаимодействия Web API и базы данных используется фреймворк Entity Framework Core. EF Core - это средство объектно-реляционного отображения (Object-Relational Mapping, ORM), которое позволяет отображать данные из базы данных в реальные объекты программы и обратно. Использование ORM экономит время разработчиков и позволяет им сосредоточиться на бизнес-логике, а не на управлении базой данных.

* 1. Архитектура приложения

Для разрабатываемого проекта была разработана диаграмма архитектуры приложения, представленная на рисунке 2.1, а также в приложении А.



Рисунок 2.1 – Диаграмма архитектуры приложения

Приложение будет работать по принципу взаимодействия клиента и сервер: клиент через API веб-браузера будет посылать асинхронные запросы на сервер и получать от него данные. Сервер будет работать на платформе ASP.NET Core, которая гарантирует кроссплатформенность полученного серверного решения. В качестве сервера будет использоваться IIS.

* 1. Диаграмма вариантов использования
  2. Проектирование клиентской части (обоснование используемых технологий)
     1. Использование *React*

Клиентская часть приложения написана с использованием библиотеки *React*.

*React* – это JavaScript библиотека, которая используется для создания динамических пользовательских интерфейсов. Она позволяет разработчикам создавать компоненты, которые отличаются высокой переиспользуемостью.

В *React* клиентская часть разбивается на компоненты, которые можно многократно использовать. Это позволяет существенно ускорить процесс создания, тестирования и модификации приложений.

Каждый компонент в React может состоять из нескольких других компонентов, а также быть вложенным в другие компоненты. Они могут реагировать на действия пользователя, что делает сайт более интерактивным.

*React* использует виртуальный DOM, что увеличивает производительность приложения, так как обновления в реальном DOM происходят только при изменении данных. Это позволяет избежать его полной перерисовки, что делает работу с библиотекой более эффективной.

Одним из главных преимуществ React является то, что он позволяет разработчикам легко работать с данными. React позволяет получить данные из сервера при помощи AJAX-запросов, что делает приложение более динамичным.

Библиотека имеет больше количество инструментов для оптимизации работы приложения, смысл работы которых можно описать как попытка изменять только те части приложения, которые нуждаются в этом, и попытка избежать лишних отрисовок там, где в этом нет необходимости

В целом, React - это мощный инструмент, который позволяет разработчикам создавать динамические и интерактивные пользовательские интерфейсы. Использование этой библиотеки в разработке клиентской части может существенно ускорить процесс создания и модификации приложения.

* + 1. Redux

*Redux* – это библиотека для управления состоянием приложения в JavaScript и TypeScript. Она позволяет хранить все данные в едином объекте-хранилище (*store*) и изменять их с помощью действий (*actions*) и редюсеров (*reducers*). Это делает управление состоянием приложения более предсказуемым и удобным, а также улучшает производительность при работе с большим объёмом данных. Redux часто используется вместе с React, но может применяться и с другими фреймворками и библиотеками JavaScript.

Redux позволяет отслеживать неконтролируемые изменения данных, хранимых в приложении, поскольку кроме хранения данных, после каждой операции делает снимок данных лежащих в памяти. Такие мутации в *TypeScript* могут быть опасными, так как они нарушают ограничения типизации и могут привести к непредсказуемому поведению в вашей программе. В частности, мутации могут вызвать ошибки времени выполнения или привести к несогласованности данных. Но это не повод отказываться от *TypeScript* в пользу *JavaScript*, поскольку в *JavaScript* вообще нет типизации. Также *Redux* позволяет избежать дополнительных запросов на сторону сервера за одними и теми же данными, которые используются в разных местах приложения.

Для удобства разработки, повышения качества кода и удобства тестирования приложения, мною была выбрана библиотека *redux-toolkit.* Эта библиотека содержит внутри себя набор функций для работы с *redux.* Она позволяет уменьшить количества написанного кода путем инкапсуляции некоторых деталей реализации функций-редюсеров, действий и т.д. С помощью этой библиотеки, разработчик способен манипулировать абстракциями, созданными за него и при желании изменять их.

Также, *redux-tookit* содержит в себе функции, с помощью которых разработчик приложения может доставать объекты из памяти браузера, с помощью которых он может производить некоторые изменения.

Для взаимодействия с серверной частью приложения используется библиотека *redux-thunk*. Как можно понять из названия, эта библиотека тоже является частью экосистемы *Redux*. Redux-thunk используется для того, чтобы с помощью асинхронных методов осуществлять запросы к серверу для получения данных. Плюсами этой библиотеки является то, что разработчик может контролировать благодаря механизмам языка TypeScript(JavaScript) стадии выполнения запроса, и, например, показывать пользователю приложения статус выполнения его запроса. Также плюсом этой библиотеки можно назвать разделение приложения на несколько частей, о которых будет рассказано в разделе архитектура клиентской части.

* + 1. Webpack

Webpack является инструментом сборки JavaScript-приложений, которые используются веб-браузерами. Это программное обеспечение позволяет разработчикам создавать оптимизированные пакеты кода, которые могут быть загружены в браузер для выполнения.

Webpack используется для компиляции и упаковки программных модулей, таких как JavaScript, CSS, изображения и др. в один или несколько файлов, что упрощает развёртывание и распространение JavaScript-приложений. Он также позволяет использовать новейшие функции JavaScript, которые не поддерживаются внешними браузерами, и транспилировать код на более старые версии, которые могут работать на более широком диапазоне браузеров.

Webpack также позволяет конфигурировать и настраивать сборку проекта в зависимости от конкретных потребностей разработчика, а также подключать плагины для оптимизации сборки, например для минификации и оптимизации размера файлов.

Благодаря возможностям Webpack, разработчики имеют возможность создавать мощные и высокоэффективные web-приложения для современных браузеров.

Webpack позволяет бесконечно расширять возможности сборки приложения, путем добавления различных плагинов, библиотек, которые добавляются при сборке приложения, например, с помощью настроенной библиотеки Webpack, можно использовать определенные препроцессоры css, такие как sass и less.

* + 1. Архитектура клиентской части

Архитектура клиентской части содержит в себе три слоя, каждый из которых отвечает за отдельную группу функций. Взаимодействие между слоями приложения показано на рисунке 2.2.



Рисунок 2.2 – схема архитектуры клиентской части приложения

Когда пользователь нажимает на кнопку, уровень бизнес-логики обрабатывает это действие, и отправляет запрос к уровню доступа к данным, чтобы изменить или получить результат. Уровень доступа к данным, в свою очередь, инкапсулирует логику работы с сервером, и для уровня бизнес-логики создается впечатление, что сервера вовсе не существует, потому что о сервере знает только один уровень.

Когда запрос с сервера приходит на клиент, уровень доступа к данным обновляет значения, которые хранятся в памяти, и дает понять компонентам, которые подписаны на изменения состояния, что нужно перерисовать разметку страницы. После этого, функции бизнес-логики, которые подписаны на изменения, производят свои расчеты и возвращают данные в компоненты, которые видит пользователь.

* 1. Проектирование серверной части

Серверная часть приложения будет написана на языке C# и будет отвечает за выполнение функций бизнес-логики и хранения данных пользователей приложения. При выборе архитектуры приложения я основывался на вопросах целесообразности потенциально возможного масштабирования приложения, удобства разработки, тестирования и развертки приложения, поэтому мною было принято решение остановиться на монолитной архитектуре.  
 Благодаря архитектуре OWIN, в ASP.NET приложениях появился конвейер обработки запроса. Это удобное решение для проектирования архитектуры, поскольку позволяет расположить несколько обработчиков, которые будут последовательно обрабатывать запрос и собирать ответ для клиента.

Серверная часть приложения разделена на контроллеры, которые хранят логику работы со своей доменной областью. Контроллеры расположены на уровне представления монолитной архитектуры.

Благодаря механизму внедрения зависимостей, разработчик может с легкостью подключать различные объекты, которые реализуют интерфейс, определяемый при внедрении. С помощью этого механизма мы связываем все уровни приложения: на уровне контроллера мы внедряем сервис, который отвечает за бизнес-логику приложения, когда происходит инициализация сервиса, мы внедряем в него репозиторий, который содержит в себе методы для работы с сущностями, которые предоставляет выбранная для разработки *ORM*. При разработке мною был выбран подход «тонкий контроллер», который подразумевает, что вся логика будет сосредоточена в сервисе, а контроллер будет использоваться только для приема запроса, валидации и отправки ответа.

Для корректного отображения данных на стороне клиента необходима информация о пользователе, которая также используется в дальнейших запросах. Поэтому для решения этой проблемы была выбрана аутентификация на основе *JWT* токена, потому что токен может хранить дополнительную информацию, такую как идентификатор пользователя и аккаунта, которые участвуют во многих запросах. Благодаря этому, нет необходимости делать дополнительные запросы к базе данных для получения этой информации.

Тело (*payload*) в *JWT* токене содержит информацию, которая передается между двумя сторонами. Оно состоит из набора пар ключ-значение, обычно содержащих информацию о пользователе, разрешениях, сроке действия и других метаданных.

Тело JWT токена может содержать любые данные в формате JSON, но должно быть закодировано в формат Base64Url. Ключ "iss" (issuer) указывает на источник, который создал токен, "sub" (subject) указывает на субъект (например, имя пользователя), "exp" (expiration time) указывает на время, когда токен истекает, и т.д. Есть также возможность включить в токен собственные поля, если это необходимо.

Тело JWT токена не является зашифрованным, но может быть защищено цифровой подписью для проверки подлинности и целостности данных. При получении JWT токена, получатель може раскодировать тело, прочитать данные и использовать их для проверки прав доступа или других целей.

Кроме общей привычной структуры сервера был класс-обработчик фоновых работ. Фоновые работы запускаются вместе с сервером и выполняются в зависимости от CRON выражения, описанного при запуске. За функционал фоновых работ отвечает библиотека *Hangfire*.

Архитектура серверной части приложения изображена на рисунке 2.3.

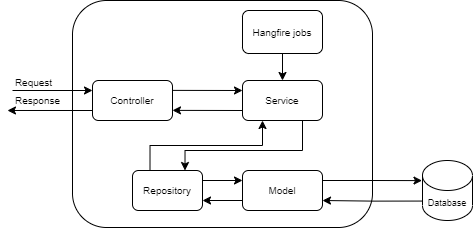


Рисунок 2.3 – Архитектура серверной части приложения

* + 1. Процесс проверки пользовательской цели

Что за цель

* 1. Проектирование базы данных

Для проектирования базы данных необходимо изначально проанализировать предметную область и сущности в ней. В итоге мы получим модель данных, показывающая нам как взаимодействовать с сущностями и как они взаимодействуют между собой.

Главными целями проектирования баз данных являются:

* гарантирование сохранения всей необходимой информации в базе данных;
* обеспечение возможности получения данных при помощи различных запросов;
* сокращение избыточности и дублирования информации в базе данных;
* обеспечение быстрого доступа к данным при поиске по записям;
* установление связей между таблицами в базе данных;
* обеспечение уникальности определенных столбцов в базе данных;
* сокращение необходимости вычисления данных в базе данных;
* гарантия доступности данных для пользователей;
* обеспечение целостности всей базы данных в целом.

Правильное проектирование базы данных обеспечивает эффективную работу с ней и гарантирует, что необходимые данные могут быть получены быстро и точно. Более того, правильно спроектированная база данных может предотвратить ошибки, связанные с неправильным использованием данных, и защитить целостность базы данных.

Модели базы данных для программного средства представлены на рисунке 2.1 и в приложении Б.

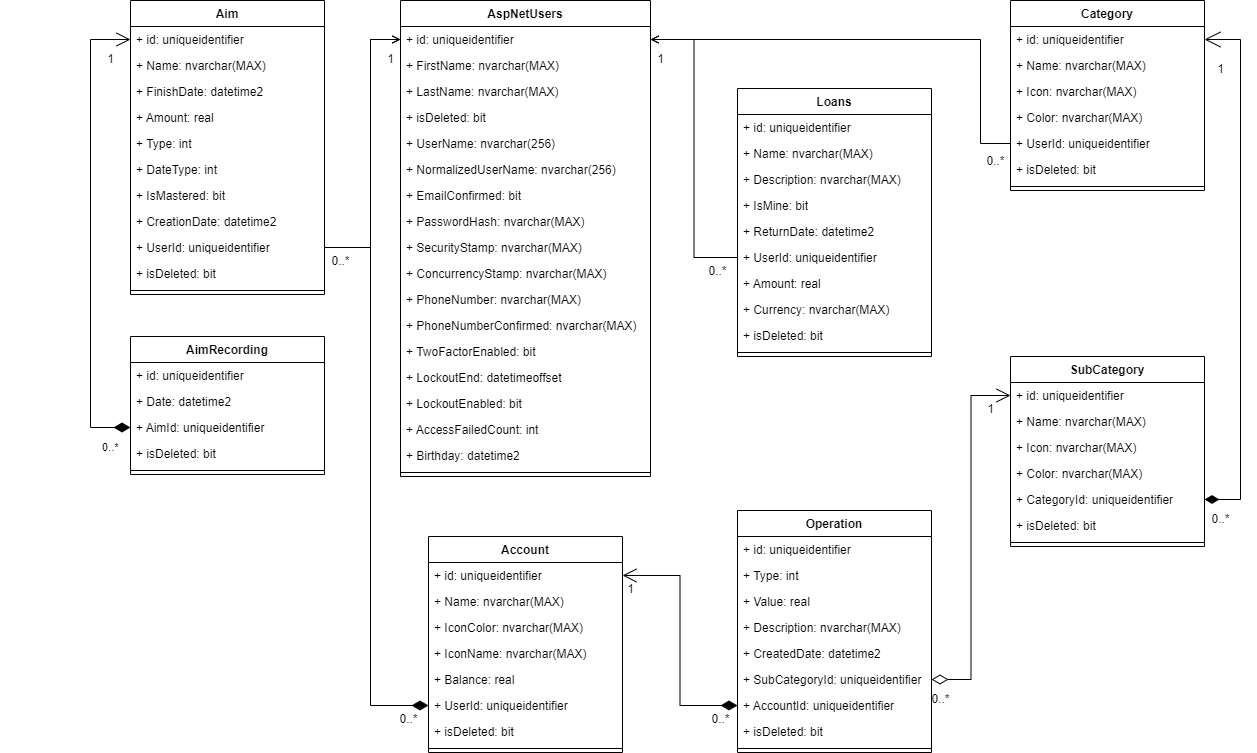


Рисунок 2.1 – Структурная диаграмма основных таблиц

База данных состоит из основных таблиц и вспомогательных, которые в основном используются для осуществления связи многие-ко-многим. Список всех основных таблиц базы данных:

* *account;*
* *aim;*
* *aimRecording;*
* *aspNetUsers;*
* *category;*
* *loans;*
* *operation;*
* *subcategory.*

В базе данных все уникальные значения первичных ключей, которые используются для идентификации записей, представлены и сохранены в виде 128-битных глобальных идентификаторов GUID, которые состоят из тридцати двух шестнадцатеричных цифр, разделенных на группы дефисами. Этот формат идентификаторов обеспечивает максимальную уникальность каждого ключа и предотвращает коллизии, что позволяет точно идентифицировать каждую запись в базе данных. В результате, надежность и точность работы системы баз данных повышаются, что является критически важным для успешной работы любой организации. Структура всех таблиц представлена в приложении Б.

Таблица Account хранит информацию о счетах пользователей. Все поля таблицы Account перечислены в таблице 2.1

Таблица 2.1 ­­­– Свойства таблицы Account

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *id* | uniqueidentifier | Уникальный идентификатор записи |
| *name* | nvarchar(max) | Название счета |
| *iconcolor* | nvarchar(max) | Название цвета иконки |
| *iconname* | nvarchar(max) | Название иконки |
| *balance* | real | Баланс счета |
| *userid* | uniqueidentifier | Уникальный идентификатор пользователя, которому принадлежит счет |
| *isdeleted* | bit | Флаг, который обозначает, удалена ли запись |

Таблица Aim хранит информацию о цели пользователя. Все поля таблицы перечислены в таблице 2.2.

Таблица 2.2 ­­­– Свойства таблицы Aim

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *id* | Uniqueidentifier | Уникальный идентификатор записи |
| *name* | Nvarchar(max) | Название цели |
| *finishdate* | Datetime2 | Дата окончания цели |
| *amount* | Real | Цель пользователя в денежном эквиваленте |
| *type* | Int | Тип цели |
| *datetype* | Int | Тип цели по дате |
| *ismastered* | Bit | Завершена ли цель |
| *creatingdate* | Datetime2 | Дата создания цели |
| *userid* | Uniqueidentifier | Уникальный идентификатор пользователя, которому принадлежит цель |
| *isdeleted* | bit | Флаг, который обозначает удалена ли запись |

Таблица AimRecording содержит информацию о прогрессе пользователя в выполнении цели. Поля таблицы содержатся в таблице 2.3.

Таблица 2.3 ­­­– Свойства таблицы AimRecording

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | Uniqueidentifier | Уникальный идентификатор записи |
| *Date* | Datetime2 | Дата, когда была выполнена цель |
| *AimId* | Uniqueidentifier | Уникальный идентификатор цели, которой принадлежит метка о выполнении |
| *isdeleted* | bit | Флаг, который обозначает удалена ли запись |

Таблица AspNetUsers содержит информацию о пользователях. Поля таблицы содержатся в таблице 2.4.

Таблица 2.4 ­­­– Свойства таблицы AspNetUsers

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | uniqueidentifier |  |
| *FirstName* | Nvarchar(max) | Имя пользователя |
| *LastName* | Nvarchar(max) | Фамилия пользователя |
| *IsDeleted* | Bit | Флаг, который обозначает удалена ли запись |
| *UserName* |  |  |
| *NormalizedUserName* |  |  |
| *EmailConfirmed* | Bit | Флаг, который обозначает подтверждена ли почта пользователя |
| *PasswordHash* |  |  |
| *SecurityStamp* |  |  |
| *ConcurrencyStamp* |  |  |
| *PhoneNumber* |  |  |
| *PhoneNumberConfirmed* |  |  |
| *TwoFactorEnabled* |  |  |
| *LockoutEnd* |  |  |
| *LockoutEnabled* |  |  |
| *AccessFailedCount* |  |  |
| *Birthday* |  |  |

Таблица Category необходима для хранения информации о главной категории, которая хранит подкатегории. Поля таблицы Category содержатся в таблице 2.5.

Таблица 2.5 ­­­– Свойства таблицы Category

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| Id | uniqueidentifier | Уникальный идентификатор записи |
| Name | Nvarchar(max) | Название категории |
| Icon | Nvarchar(max) | Название иконки |
| Color | Nvarchar(max) | Цвет иконки |
| UserId | uniqueidentifier | Уникальный идентификатор пользователя, которому принадлежит категория |
| IsDeleted | Bit | Флаг, укзаывающий, удалена ли запись |

Таблица Loans содержит информацию о долгах пользователя. Таблица 2.6 содержит все поля таблицы Loans.

Таблица 2.6 ­­­– Свойства таблицы Loans

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| Id | uniqueidentifier | Флаг, который обозначает удалена ли запись |
| Name | Nvarchar(max) |  |
| Description | Nvarchar(max) |  |
| IsMine | Bit |  |
| ReturnDate | dateTime2 |  |
| UserId | uniqueidentifier |  |
| Amount | Real |  |
| Currency |  |  |
| IsDeleted | bit | Флаг, который обозначает удалена ли запись |

Таблица Operation содержит информацию о операциях пользователя по определенному счету. Поля таблицы содержатся в таблице 2.7.

Таблица 2.7 ­­­– Свойства таблицы Operation

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| Id | Uniqueidentifier | Уникальный идентификатор записи |
| Type | Int | Тип операции |
| Value | Real | Значение |
| Description | Nvarchar(max) | Описание операции |
| CreatedDate | dateTime2 | Дата создания |
| SubCategoryId | Uniqueidentifier | Уникальный идентификатор категории |
| AccountId | UniqueIdentifier | Уникальный идентификатор счета |
| IsDeleted | bit | Флаг, который обозначает удалена ли запись |

Таблица Subcategory используется для хранения информации о подкатегориях для операций. Таблица 2.8 содержит все поля таблицы Subcategory.

Таблица 2.8 ­­­– Свойства таблицы Subcategory

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| Id | uniqueidentifier | Флаг, который обозначает удалена ли запись |
| Name | Nvarchar(max) | Название |
| Icon | Nvarchar(max) |  |
| Color | Nvarchar(max) |  |
| CategoryId | uniqueidentifier |  |
| IsDeleted | Bit | Флаг, который обозначает удалена ли запись |

* 1. Выводы по разделу

На данном этапе мы спроектировали архитектуру приложения. Была спроектирована

Спроектировали серверную часть, разработали ее архитектуру и связь между классами. Также мы определили, какие сторонние библиотеки будут использованы в работе серверной части приложения.

Спроектировали архитектуру клиентской части приложения, решили вопросы хранилища данных приложения клиентской части и выделение функций библиотеки redux-thunk в отдельный слой приложения.

Разработали диаграмму вариантов использования и определили все основные сценарии, как может быть использовано приложение.

Кроме того, было спроектировано хранилище данных и структурная диаграмма приложения, а также подробно описаны все спроектированные таблицы базы данных, их предназначение, предназначение всех полей каждой из таблиц и тип данных каждого из полей.

Был описан основной алгоритм проверки цели, который будет использоваться в приложении.

После проектирования можно приступать к разработке программного средства.

1. Разработка программного средства
   1. Серверная часть приложения
      1. Структура проекта

Для разработки серверной части был использован язык C# и платформа ASP.NET Core. Структура проекта изображена на рисунке 3.1

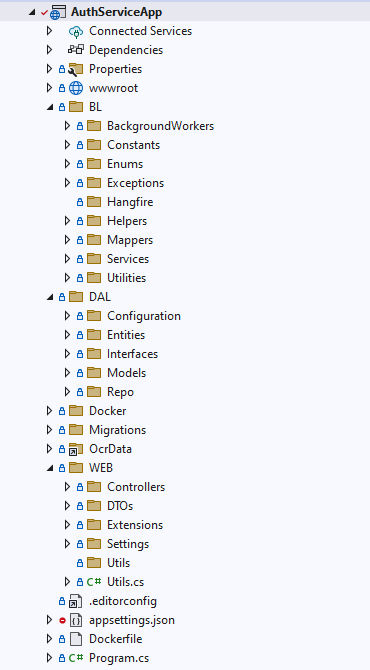


Рисунок 3.1 ­­­– Строение проекта серверной части приложения

Файлы различных слоев приложения расположены в соответствующих папках. Рассмотрим их содержимое подробнее. Внутри папки *Settings*, хранятся файлы настроек, которые импортируются из файла конфигурации проекта appsettings.json.

Файлы настроек инициализируются на старте приложения в файле *program.cs* – главном файле приложения. Он содержит в себе главную точку входа в приложения, выполняет важную функцию, так как запускает приложение и настраивает его основные компоненты, такие как веб-сервер и конфигурацию.

Рассмотрим содержимое папки WEB. Эта папка отвечает за объекты слоя представления. В папке Controllers содержатся классы контроллеры, которые принимают все запросы и адресуют выполнения сервисам.

В папке DTOs содержатся специальные классы dto (data transfer object), которые нужны для передачи данных между сервером и клиентом. Они используются для скрытия информации от клиента.

В папке Extensions хранятся классы, методы которых расширяют объекты платформы ASP.NET. Это делается для удобства разработки, поскольку улучшает читаемость файла program.cs и позволяет лучше в нем ориентироваться.

После обработки запроса приложение вызывает методы папки BL. Папка Background workers отвечает за методы для работы с Hangfire.

Папка Constants содержит в себе файлы с константами, которые используются в целях отображения сообщений об ошибках, статус-кодах.

Папка Enums хранит информацию о статус кодах, типах операций клиента и типе окружающих сред для развертки, например Docker.

Папка Exceptions хранит в себе определение уникальной ошибки, которая используются для отлова исключительных ситуаций в приложении и отображении этих ошибок для клиентской стороны.

Папка Helpers содержит классы, которые по-всякому помогают в разработке, например, помогают достать полезные данные из JWT токена.

Папка Mappers содержит в себе профили для библиотеки AutoMapper.

Папка Services содержит в себе классы, которые исполняют всю бизнес-логику приложения.

Папка Utilities содержит файлы, которые предоставляют функции для работы с отправкой почтовых сообщений и созданием токенов

Рассмотрим содержимое папки DAL. В папке Configuration хранятся данные для конфигурации таблиц с помощью Fluent API. С помощью этой технологии можно более гибко конфигурировать поведение таблицы базы данных, например, настроить внешние ключи, настроить поведение связей, при удалении данных. Пример такого файла содержится в листинге 3.1.

|  |
| --- |
| public class CardConfiguration : IEntityTypeConfiguration<CardEntity>  {  public void Configure(EntityTypeBuilder<CardEntity> builder)  {  builder.HasKey(key => key.Id);  builder.Property(prop => prop.Id).IsRequired();  builder.HasMany(key => key.Spendings).WithOne(one => one.Card).OnDelete(DeleteBehavior.ClientCascade);  builder.HasOne(key => key.User).WithMany(one => one.Cards).OnDelete(DeleteBehavior.NoAction);  }  } |

Листинг 3.1 – пример конфигурации с помощью Fluent API.

В папке Entities содержатся сущности базы данных в виде классов C#.

В папке Interfaces хранятся интерфейсы для репозиториев, которые нужны для инъекции репозиториев в сервисы.

В папке Models хранится файл, с помощью которого происходит конфигурация базы данных и идет подключение конфигурации Fluent API из папки Configuration.

В папке Repo содержатся файлы репозиториев для работы с сущностями базы данных.

* + 1. Конфигурация и запуск проекта

Этот файл содержит в себе пары ключ-значение, которые отвечают за настройку приложения, например, хранят пароли от почтового аккаунта. Далее все настройки собираются в один файл для удобства импорта всех данных.

Пример файла настроек представлен в листинге 3.1.

|  |
| --- |
| namespace AuthServiceApp.WEB.Settings;  public class DatabaseSettings  {  public string Server { get; set; }  public string Port { get; set; }  public string Database { get; set; }  public string UserId { get; set; }  public string Password { get; set; }  public int ConnectTimeout { get; set; }  public bool Encrypt { get; set; }  public bool TrustedServerCertificate { get; set; }  public string ApplicationIntent { get; set; }  public bool MultiSubnetFailover { get; set; }  public string ConnectionString =>  $"Server={Server},{Port};" +  $"Database={Database};" +  $"User ID={UserId};" +  $"Password={Password};" +  $"Connect Timeout={ConnectTimeout};";  } |

Листинг 3.1 – Файл настроек базы данных

В файле program.cs создается объект класса `WebHostBuilder`, который дает возможность настроить веб-хост для ASP.NET Core приложения. Этот объект имеет ряд методов, которые позволяют настроить различные свойства веб-хоста, такие как порт, адрес, использование SSL, логгирование и многие другие.

В методе «`*CreateDefaultBuilder*`» устанавливаются значения по умолчанию, такие как порт и адрес. Затем выполняются дополнительные настройки веб-хоста, которые передаются в метод `ConfigureWebHostDefaults`.

В методе `ConfigureWebHostDefaults` выполняются настройки веб-сервера, создается экземпляр класса `Startup`, который содержит настройки маршрутизации приложения и вызывается метод `Configure` на этом объекте.

Метод `Configure` настраивает конвейер обработки запросов, который включает middleware, обрабатывающие запросы до того, как они попадают в приложение, и после завершения работы приложения. Здесь можно также указать источник данных, используемых приложением, а также расположение статических файлов.

Таким образом, файл *program.cs* является ключевым файлом для запуска и настройки ASP.NET Core приложения, и содержит множество возможностей для настройки веб-сервера и компонентов приложения. Содержимое файла представлено в приложении В.

Внутри файла program.cs мы получаем доступ к объекту ConfigurationManager, который реализует интерфейс IConfiguration и находится внутри объекта WebApplicationBuilder. Для удобства разработки все функции работы с коллекцией сервисов приложения, были объединены в файлы, которые используют механизм расширения методов. То есть были написаны методы, расширяющие функционал классов, которые используются в файле program.cs. Процесс добавления настроек приложения последовательно изображен на листингах ниже.

|  |
| --- |
| var appSettings = builder.Configuration.RegisterSettings(); |

Листинг 3.2 – Получение объекта настроек

Содержимое метода *RegisterSettings* представлено в листинге 3.3.

|  |
| --- |
| public static AppSettings RegisterSettings(this IConfiguration configuration)  {  Console.WriteLine(Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT"));  var conf = new ConfigurationBuilder()  .AddJsonFile("appsettings.json")  .AddJsonFile(  $"appsettings.{Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT") ?? "Development"}.json",  optional: true)  .AddEnvironmentVariables()  .Build();  return new()  {  Database = conf.GetSection(nameof(AppSettings.Database))  .Get<DatabaseSettings>(),  Token = conf.GetSection(nameof(AppSettings.Token))  .Get<TokenSettings>(),  SmtpClientSettings = conf.GetSection(nameof(AppSettings.SmtpClientSettings))  .Get<SmtpClientSettings>(),  GoogleAuthSettings = conf.GetSection(nameof(AppSettings.GoogleAuthSettings))  .Get<GoogleAuthSettings>()  };  } |

Листинг 3.3 – получение настроек в зависимости от переменных среды.

Файл appsettings.json меняется в зависимости от среды разработки. Так, этот файл будет содержать другие значения для процесса разработки и для развертки в Docker. Поэтому для правильной работы приложения необходимо иметь актуальный файл.

Далее мы передаем этот файл в метод расширения, который отвечает за внедрение всех зависимостей, которые мы создали как разработчики. После того, как мы создаем объект-singleton, мы можем внедрять этот объект через конструктор при инициализации других объектов. На листинге 3.4 представлен код для внедрения написанных зависимостей.

|  |
| --- |
| public static void RegisterServices(this IServiceCollection services, AppSettings appSettings)  {  //settings  services.AddSingleton(appSettings);  //Services  services.AddScoped(typeof(IGenericService<>), typeof(GenericService<>));  services.AddTransient<IServiceManagement, ServiceManagement>();  services.AddTransient<IAccountService, AccountService>();  services.AddTransient<IOperationService, OperationService>();  services.AddTransient<ICategoryService, CategoryService>();  services.AddTransient<IAimService, AimService>();  services.AddTransient<IPictureService, PicturesService>();  services.AddTransient<ILoanService, LoanService>();  services.AddTransient<ICardService, CardService>();  services.AddTransient<IAuthService, AuthService>();  services.AddTransient<IRoleService, RoleService>();  services.AddTransient<IUserService, UserService>();  services.AddTransient<ISpendingService, SpendingService>();  services.AddTransient<IShopService, ShopService>();  //Repositories  services.AddScoped(typeof(IBaseRepository<>), typeof(BaseRepository<>));  services.AddTransient<IAccountRepository, AccountRepository>();  services.AddTransient<ICategoryRepository, CategoryRepository>();  services.AddTransient<ICardRepository, CardRepository>();  services.AddTransient<ISpendingRepository, SpendingRepository>();  services.AddTransient<IShopRepository, ShopRepository>();  services.AddTransient<IUserRepository, UserRepository>();  services.AddTransient<IEmailSender, EmailSender>();  } |

Листинг 3.4 – Внедрение зависимостей в приложении.

После того, как настроен файл program.cs и все сопутствующие компоненты, можно переходить к другим аспектам серверной части приложения.

* + 1. Уровень доступа к данным

Когда приложение запускается, срабатывает метод, который открывает соединение с базой данных, после этого происходит проверка моделей, которые описаны в приложении. При создании базы данных был использован подход *code-first*, при котором сначала проектируются сущности, с которыми будет работать инженер, а на основе этих моделей уже создается база данных. В файле *ApplicationContext.cs*, содержится метод, который применяет конфигурацию таблиц базы данных, написанную с помощью технологии *fluent api*. Содержимое метода *OnModelCreating* представленов листинге 3.5

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder builder)  {  base.OnModelCreating(builder);  builder.ApplyConfiguration(new OperationConfiguration());  builder.ApplyConfiguration(new UserConfiguration());  builder.ApplyConfiguration(new ShopPositionConfiguration());  builder.ApplyConfiguration(new SpendingConfiguration());  builder.ApplyConfiguration(new CardConfiguration());  builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());  } |

Листинг 3.5 – Метод OnModelCreating, который добавляет конфигурацию таблиц.

После произведенных манипуляций, мы можем приступить к разработке репозиториев приложения. Репозитории представляют классы, содержащие набор методов, которые реализуют определенный интерфейс. Реализация интерфейсов особенно важна в данном контексте поскольку это основной механизм внедрения зависимостей в приложении.

Поскольку работу с некоторыми сущностями приложения можно объединить и обобщить в один репозиторий, мы сначала создадим обобщенный интерфейс, который будет содержать набор основных методов для работы с сущностью, на основе которой будет создан репозиторий. Далее создадим базовый класс-репозиторий, который реализует этот интерфейс. Содержимое этого репозитория представлен в приложении Г.

После создания базового репозитория, уже можно внедрять его в сервисы и использовать для манипуляции над данными. Но могут возникнуть случаи, когда функционала, который содержится в базовом классе, недостаточно, а другим потомкам базового класса эти методы не нужны. В таком случае, разработчик создает новый интерфейс, описывает список методов, реализует их в новом классе, и наследуется от базового класса. Пример такого репозитория показан на листинге 3.6

|  |
| --- |
| using AuthServiceApp.DAL.Entities.Account;  using AuthServiceApp.DAL.Interfaces;  using AuthServiceApp.DAL.Models;  using Microsoft.EntityFrameworkCore;  namespace AuthServiceApp.DAL.Repo.Account;  public class AccountRepository : BaseRepository<AccountEntity>, IAccountRepository  {  public AccountRepository(ApplicationDbContext databaseContext) : base(databaseContext)  {  }  public Task<List<AccountEntity>> GetAllUserAccounts(Guid userId)  {  return Entity.Where(account => account.UserId == userId).Include(x => x.Operations).ToListAsync();  }  }  public interface IAccountRepository : IBaseRepository<AccountEntity>  {  Task<List<AccountEntity>> GetAllUserAccounts(Guid userId);  } |

Листинг 3.6 – репозиторий приложения серверной части

После создания репозиториев можно приступать к работе над бизнес-логикой приложения.

* + 1. Уровень представления

После того, как мы при запуске приложения получили набор абстракций для работы над данными, мы можем начинать разработку бизнес логики. Мы можем расположить ее в контроллерах, но в таком случае они получатся громоздкими, трудными для тестирования и поддержки. Плюсы использования отдельного слоя для таких задач неоспорима, поэтому оставим разработку бизнес-логики на потом и займемся уровнем представления.

Для мануального тестирования написанного *API* был добавлен сервис Swagger. Он позволяет проектировать документацию для сервера в удобном формате, а также предоставляет визуальный инструмент для тестирования написанных точек.

Когда сервер принимает запрос, объект запроса проходит через конвейер обработки, модифицируется и попадает в контроллер в виде объекта. Для того, чтобы ограничить количество информации получаемой и отправляемой клиентской частью, используются *dto*. Например, клиент не сможет увидеть идентификатор сущности, если ее скрывает *dto.*

Для быстрой конвертации между сущностями базы данных и *dto* используется библиотека *AutoMapper*. Для ее использования нужно создать профиль маппинга – замена значений сущностей одного объекта на поля другого. Затем вызвать функцию *CreateMap* и указать, класс-источник и класс-назначение на который мы накладываем новые значения.

Пример класса-профиля AutoMapper содержится на листинге 3.7

|  |
| --- |
| public class MapperProfile: Profile  {  public MapperProfile()  {  AccountMappings();  OperationMappings();  }  private void AccountMappings()  {  CreateMap<AccountEntity, AccountModel>().ReverseMap();  CreateMap<CreateAccountModel, AccountEntity>();  CreateMap<UpdateAccountModel, AccountEntity>();  }  private void OperationMappings()  {  CreateMap<OperationModel, OperationEntity>().ReverseMap();  CreateMap<CreateOperationModel, OperationEntity>();  }  } |

Листинг 3.7 – класс профиль, который настраивает библиотеку AutoMapper

Когда метод контроллера получает обработанный объект запроса, он проводит свои проверки, например, если приходит запрос на получение данных о пользователе, но не предоставлена информация о его идентификаторе, нужно выбросить ошибку. (Упростить) Затем вызывается метод, который отвечает за выполнение бизнес-логики. Для того чтобы вызвать этот метод, в контроллер при инициализации через конструктор внедряется сервис, который содержит внутри себя все методы, необходимые для работы с сущностью.

Когда метод сервиса возвращает результат выполнения, контроллер формирует ответ, определяет статус код и возвращает ответ клиенту.

Но также может возникнуть ситуация, когда внутри сервиса возникает исключительная ситуация. Например, сервис будет ожидать, что вернется из базы одна сущность, но при этом вернулось несколько. В таком случае, будет поднята уникальная ошибка, назначение которой быть отловленной конвейером обработки запросов. Код промежуточного ПО цикла обработки запроса, которое отлавливает ошибки представлен на листинге 3.8.

|  |
| --- |
| public static class ExceptionHandlerExtension  {  public static void RegisterExceptionHandler(this IApplicationBuilder app, Serilog.ILogger Logger)  {  #if DEBUG  app.UseDeveloperExceptionPage();  #endif  app.Use(async (context, next) =>  {  try  {  await next();  }  catch (ApplicationHelperException ex)  {  context.Response.StatusCode = ex.ErrorStatus switch  {  ServiceResultType.NotFound => StatusCodes.Status404NotFound,  ServiceResultType.InvalidData => StatusCodes.Status400BadRequest,  ServiceResultType.NotAuthorized => StatusCodes.Status401Unauthorized,  ServiceResultType.Forbidden=> StatusCodes.Status403Forbidden,  ServiceResultType.ServerError => StatusCodes.Status500InternalServerError,  \_ => throw ex  };  Logger.Information($"Error caught in ApplicationHelperException: {ex.Message}");  var result = new { errorMessage = ex.Message };  await context.Response.WriteAsJsonAsync(result);  }  catch (System.Exception ex)  {  Logger.Information($"{ex.Message}");  //#if DEBUG  throw ex;  //#else  context.Response.StatusCode = (int)System.Net.HttpStatusCode.InternalServerError;  var result = new { errorMessage = "Something went wrong..." };  await context.Response.WriteAsJsonAsync(result);  //#endif  }  });  }  } |

Листинг 3.8 – класс обработчика уникальных ошибок.

Этот класс расширяет функционал интерфейса IApplicationBuilder. Разработчик может удобно вызвать этот метод из класса Program.cs, чтобы добавить ПО в конвейер обработки запросов. Также здесь добавлен логгер, который записывает информацию об ошибке в отдельный файл, чтобы позднее разработчик мог отследить, что могло произойти не так. В случае ошибки, которую мы сами вызвали, ПО само формирует ответ и отсылает его, не передавая управление методу контроллера.

* + 1. Уровень бизнес-логики

После того, как контроллер отдает на обработку объект запроса, он попадает на уровень бизнес-логики. Там он с помощью объекта mapper, который реализует интерфейс IMapper и внедряется с помощью конструктора, приводится к объекту типа сущности, далее над ним проводятся некоторые действия, и результат обработки отправляется обратно в контроллер.

Для того, чтобы успешно выполнить все манипуляции необходимы зависимости, такие как репозиторий из которого будут браться и куда будут сохраняться сущности, другие сервисы. Для их внедрения используются конструктор, который ожидает, что в него придут объекты, которые реализуют интерфейсы определенных типов. Пример сервиса, который работает с репозиторием можно найти в приложении Д.

Перед тем как вернуть результат выполнения метода, этот результат приводится к другому типу, чтобы ограничить доступ уровня представления к данным. Это делается с помощью того же объекта mapper.

Все сервисы, в целом, выполняют схожие задачи, но есть некоторые, возможности которых выделяется. Примерами таких сервисов могут служить сервис пользователей. Это обусловлено тем, что он использует внутри себя функционал, который предоставляет Identity Server framework, а именно классы для работы с пользователями, авторизацией. Эти классы позволяют пользователям входить в систему, выходить, менять пароли.

Для прохождения авторизации пользователь отправляет POST запрос в контроллер пользователей, где результат обрабатывается, и клиент получает JWT токен. Этот токен содержит внутри закодированный идентификатор пользователя, который вошел в систему. Далее клиент с этим идентификатором может получать информацию от сервера, посылая этот идентификатор в запросе и получая персонализированные данные. Когда пользователь выходит из приложения, его токен стирается и пользователь не может получить доступ к API сервера.

Также нужно отметить использование библиотеки Mailkit. С ее помощью происходит уведомление пользователей о событиях системы: приближение выплаты долга, успешная регистрация, сообщение с ссылкой на подтверждение почты. Данные для работы библиотеки, такие как пароль и имя аккаунта, с которого будет идти отправка почтовых писем, берутся из класса AppSettings, который мы внедрили в приложение и который хранит значения файла appsettings.json. На рисунке 3.2 можно увидеть пример нотификации пользователя.

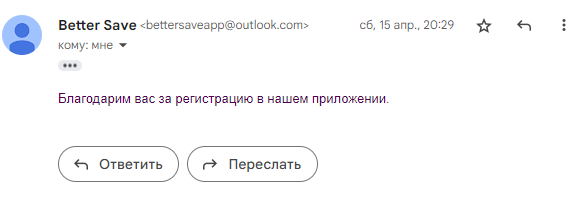


Рисунок 3.2 – Оповещение пользователя об успешной регистрации

* + 1. Класс фоновых работ

Для выполнения специфичных работ, таких как ежедневный мониторинг выполнения целей, актуальности целей пользователей и их нотификация, мною используется библиотека Hangfire. Она предоставляет по для работы с фоновыми работами. Эта библиотека также предоставляет интерфейс для мониторинга и анализа выполнения фоновых работ. Изображение интерфейса сервиса фоновых работ содержится на рисунке 3.3.

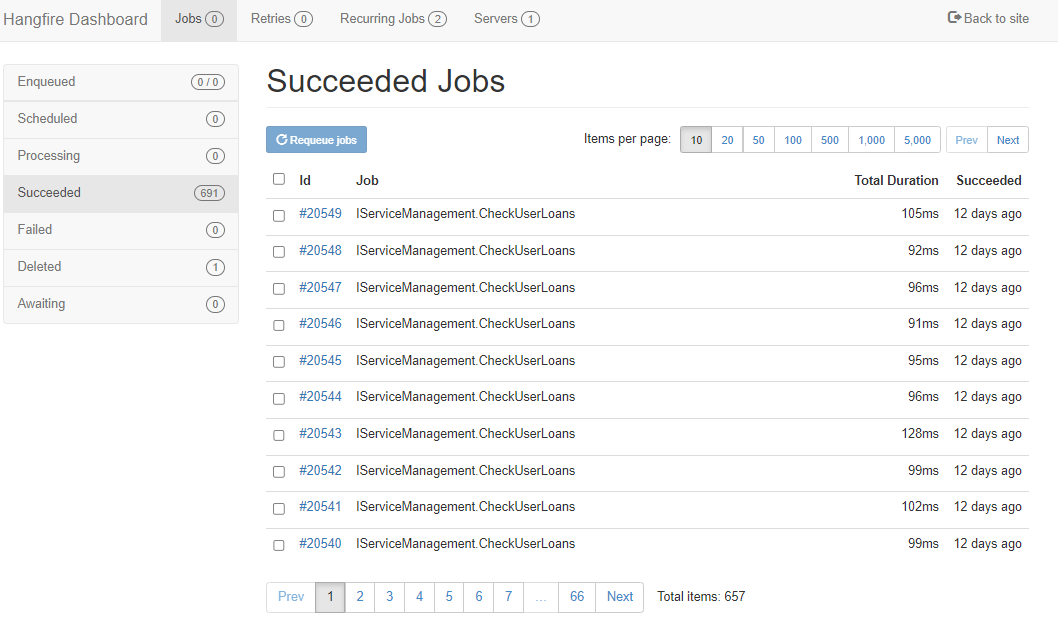


Рисунок 3.3 – интерфейс управления фоновыми работами

В приложении есть три операции, которыми занимается Hangfire. Во-первых, это оповещение пользователя после успешной регистрации, во-вторых, это оповещение пользователей, если приближается срок выплаты долга, в-третьих, анализ выполнения целей и создание объектов прогресса в дни, когда цели пользователей выполняются. Последние две операции выполняются с частотой в один день. Пример запуска функции через Hangfire содержится на листинге 3.9

|  |
| --- |
| /// <summary>  /// Confirms user email and send email  /// </summary>  /// <param name="id">User id</param>  /// <param name="token">Email confirmation token</param>  /// <returns>No content</returns>  /// <response code="204">Email confirmed successfully</response>  /// <response code="400">Email cannot be confirmed</response>  [HttpGet("email-confirmation")]  [ProducesResponseType(StatusCodes.Status204NoContent)]  [ProducesResponseType(StatusCodes.Status400BadRequest)]  public async Task<IActionResult> ConfirmEmail(string id, string token)  {  var confirmResult = await \_authService.ConfirmAsync(id, token);  var jobId = BackgroundJob.Enqueue<IServiceManagement>(x => x.SendSuccessMessage(confirmResult));  return Ok("Confirmed");  } |

Листинг 3.9 – Запуск функции через Hangfire.

* 1. Клиентская часть приложения
     1. Структура проекта

Структура проекта является ключевым элементом для успешного и эффективного развития и поддержки проекта на React. Это важно по таким причинам, как улучшение читаемости кода, легкость поддержки, облегчение масштабирования. Структура проекта клиентской части изображена на рисунке 3.4

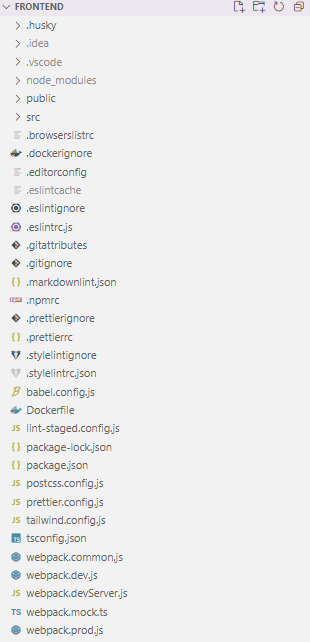


Рисунок 3.4 – Структура проекта клиентской части

По рисунку мы можем увидеть обилие файлов конфигурации, которые необходимы для правильной работы проекта. Разберем поочередно для чего нужны папки и файлы проекта.

Папка .husky содержит конфигурационную информацию для Husky, определяющую, какие скрипты должны быть запущены при определенных операциях Git. Например, при фиксации изменений (commit) или при пуше (push) изменений в удаленный репозиторий. В случае нашей программы этот файл занимается запускает программу-линтер, которая проверяет все файлы проекта и запрещает фиксировать изменения, если в программе найдены ошибки. Это заставляет ответственнее подходить к написанию кода и приучает отправлять в рабочий репозиторий только работающий код.

Папка .vscode содержит настройки для среды разработки Visual Studio Code, такие как действие на нажатие клавиши сохранения и т.д.. В данном случае, эти действия запускают библиотеку prettier, которая приводит код на странице к определенному виду, который описан правилами приложения.

Папка node\_modules содержит код библиотек зависимостей проекта, которые подключаются в приложении.

В папке public хранятся различные источники медиаконтента, такие как логотипы, картинки, фоны к приложению, шрифты.

В папке src хранится весь код к приложению. Структура папки src представлена на рисунке 3.5.

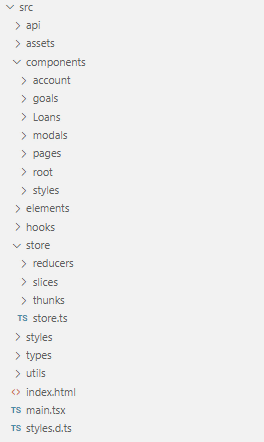


Рисунок 3.5 – структура папки src

В папке api хранится набор модулей, которые посылают запросы к серверу.

Папка assets содержит изображения, которые используются в приложении.

Папка компонент содержит в себе все структурные блоки приложения.

Папка elements хранит в себе элементы страницы, которые могут быть многократно переиспользованы в приложении в разных компонентах.

В папке hooks расположены функции для работы с механизмом react, называемым хуками. Хуки являются функциями, которые обращаются к API React и могут использовать другие функции, но при этом не могут находиться нигде, кроме функциональных компонентов. Поэтому нельзя просто написать модуль, который экспортирует функцию, но нужно написать хук.

Далее в папке store, хранится функционал для работы с уровнем доступа к данным. Здесь расположены файлы для работы с redux, redux-toolkit, redux-thunk, а именно для создания хранилища данных, редюсеров, которые манипулируют данными, и функций, которые отправляют запросы на сервер, используя модули папки api.

Папка styles хранит набор стилей приложения, различные scss переменные, такие как цветовая палитра приложения. Также здесь подключаются стили библиотеки tailwind.

В папке types находятся файлы, содержимое которых определяет все классы приложения, интерфейсы и типы. Это преимущество языка TypeScript над JavaScript, поскольку разработчик во время написания кода может видеть, какие поля содержит тот или иной объект, а также с помощью инструмента *ESLint* проверить написал ли он код правильно еще до запуска приложения.

* + 1. Настройка Webpack

Для запуска приложения используется команда описанная в файле package.json. Эта команда запускает пакет Webpack. Пакет считывает данные из файлов настроек проекта и понимает, каким образом ему нужно собрать все файлы проекта в одну сборку. Существуют файлы настроек для каждого типа среды: среда разработки и среда развертки приложения.

Для среды разработки существует пакет, который запускает приложение Webpack в режиме быстрой перезагрузки, что позволяет моментально видеть изменения на странице, что важно в разработке клиентской части.

Для среды публикации существует специальный файл настроек, с помощью которого Webpack максимально оптимизирует получающуюся сборку, что уменьшает размер конечного пакета. В ходе этого процесса убираются пакеты, которые использовались в процессе разработке, уменьшается размер файлов стилей, сжимаются файлы, которые используются в проекте.

Настройки для среды развертки приложения позволяют уменьшить время загрузки приложения, что существенно влияет на пользовательский опыт, ведь никто из пользователей не хочет, чтобы доступ к их приложению занимал много времени. Быстродействие – это ключ к победе над конкурентами.

* + 1. Разработка компонентов

Главный плюс приложения на react заключается в возможности повторного использования программных компонентов. Грамотное использование компонентов может сыграть большую роль во времени и ресурсах затраченных на разработку пользовательского интерфейса.

В приложении есть главный компонент App, который содержит в себе обертку приложения компонентом redux, который предоставляет в дочерние элементы дерева хранилище данных и методы для работы с ним, также в этом компоненте содержится другой компонент, который отвечает за маршрутизацию в приложении, путем сопоставления маршрута в браузере и отрисовываемого компонента. Также тут содержится компонент библиотеки react-toastify, который встраивает в разметку страницы элементы, которые сигнализирует о статусе выполнения api запросов, когда они выполняются, путем показа уведомлений в правой части экрана. Содержимое компонента App находится в листинге 3.9.

|  |
| --- |
| import store from "@/store/store";  import { ToastContainer } from "react-toastify";  import { Provider } from "react-redux";  import AppRouter from "./appRouter";  import "rsuite/dist/rsuite.css";  import "slick-carousel/slick/slick.css";  import "slick-carousel/slick/slick-theme.css";  import "react-toastify/dist/ReactToastify.css";  function AppRoot() {    return (      <Provider store={store}>        <AppRouter />        <ToastContainer />      </Provider>    );  }  export default AppRoot; |

Листинг 3.9 – Содержимое компонента App.

Также здесь происходит добавление глобальных стилей для используемых библиотек приложения. Это одна из отличительных функций react. Разработчик может писать стили, которые в зависимости от компонента и способа написания могут быть как глобальными, так и инкапсулированными внутри своего компонента. Это заключается в том, что модульные стили при сборке приложения сокращаются и изменяются сборщиком, после чего даже два одинаковых класса определенные в разных компонентах не будут перекрывать друг друга.

Маршрутизация приложения настроена таким образом, что, если у пользователя нет JWT токена для доступа к определенному маршруту в приложении, то его перебрасывает на страницу с авторизацией в приложении. После чего устанавливается токен, и пользователь может пользоваться приложением.

* + 1. Менеджер состояний Redux

Отличием от обычного порядка разработки хранилища на redux, является использование пакета redux-toolkit. С его помощью многие объекты можно создать, вызвав определенную функцию. Так, чтобы создать *reducer*, которые будет определять, какие действия происходят с данными при вызове той или иной функции, необходимо вызвать функцию createSlice, передав в нее состояние функции пакета *redux-thunk*, при котором будет срабатывать необходимая функция, а также функцию, которая будет производить манипуляции над данными.

На листинге 3.11 содержится процесс создания редюсера для хранения информации о пользователе и способах авторизации.

|  |
| --- |
| const authSlice = createSlice({    name: "auth",    initialState: initState,    reducers: {      logout(state) {        state.user = initState.user;        state.authStatus = "notauthenticated";      },    },    extraReducers: (builder) => {      builder.addCase(loginThunk.fulfilled, (state, action) => {        if (action.payload !== null) {          state.user = action.payload;          state.authStatus = "authenticated";        }      });      builder.addCase(getUserInfoThunk.fulfilled, (state, action) => {        if (action.payload !== null) {          state.user = action.payload;          state.authStatus = "authenticated";        }      });    },  });  export default authSlice.reducer; |

Листинг 3.11 – Функция редюсер для изменения информации о пользователе.

* 1. Добавление стилей

Стили приложения в основном написаны на библиотеке *tailwind*. Также для сложных случаев, которые не предусмотрены библиотекой, были написаны собственные модульные стили, а также переписаны оригинальные стили библиотеки. Например, для анимации на странице входа в приложении.

Механизм разработки стилей в приложении представляет собой сборку интерфейса по проектированному дизайну, путем итеративного подхода, когда разработчик постепенно добавляет классы в атрибут *className*.

Библиотека tailwind предоставляет набор классов, который покрывает практически все доступные стили css, исключением являются стили, которые могут быть недоступны в различных браузерах. Названия классов происходят от стилей, которые они реализуют. Пример класса tailwind и свойства, которые он реализует изображены на рисунке 3.6.



Рисунок 3.6 –Классы tailwind и их реализация в библиотеке

С помощью этой библиотеки были настроены такие аспекты приложения как отступы между компонентами верстки, размер шрифтов, их вес, способ расположения элементов, их порядок и т.д.

Также библиотека tailwind предоставляет набор классов для создания простой анимации в приложении, например, пульсации некоторых компонентов, что может обратить внимание пользователей к компоненту, например, показать, что у поля ввода есть описание, которое он может прочитать, чтобы лучше понимать, за что отвечает вводимое им значение.

* 1. Выводы по разделу

Руководствуясь раннее спроектированной архитектурой, были разработаны приложения серверной и клиентской частей. В качестве источника данных серверная часть использует раннее спроектированную базу данных. Было достигнуто взаимодействие между различными частями приложения.

Серверная часть отвечает за обработку запросов клиентского приложения, выполняет всю бизнес-логику приложения и отправляет данные для просмотра клиенту. В качестве протокола передачи данных между клиентом и сервером используется http, а в качестве формата передачи используется json, ввиду простоты чтения его клиентской частью.

Клиентская часть предоставляет интерфейс пользователю для чтения, изменения, создания и удаления контента. Она состоит из повторно используемых компонентов, что позволяет составить унифицированный интерфейс и ускорить разработку, а также улучшает структуру проекта и читаемость кода. Также клиентское приложение содержит в своей структуре компонент, который отвечает за маршрутизацию запросов и отрисовку соответствующих страниц.

После того как пройден этап разработки, можно заняться тестированием приложения.

1. Тестирование приложения
   1. Тестирование серверной части приложения
      1. Unit тестирование приложения
      2. Тестирование Web API через
   2. Тестирование клиентской части

Над клиентской частью было проведено мануальное тестирование.

Мануальное тестирование рассматривается как процесс проверки функциональности и качества программного продукта вручную, без использования автоматизированных инструментов или скриптов.

Гибкость и адаптивность мануального тестирования является ключевым фактором в выборе способа проверки программного средства на предмет соответствия норме. Мануальное тестирование может быть очень гибким и адаптивным к требованиям проекта, поскольку тестировщик может реагировать на изменения в продукте и изменять тест-кейсы по мере необходимости.

Недостатком мануального тестирования является то, что оно требует больших затрат времени и усилий, поскольку тестирование выполняется вручную. Также мануальное тестирование может быть не эффективным в случае больших и сложных проектов.

Для проверки поведения программного средства при вводе невалидных данных в заполняемые формы были проведены тесты на контрольных примерах. Для выявления ошибок необходимо было выполнить определенные действия, сравнить полученные результаты с ожидаемыми и сделать вывод о наличии ошибок. Для этого были разработаны тестовые случаи.

* 1. Выводы по разделу

1. Руководство пользователя
2. Экономическое обоснование проекта

Заключение

Список использованных источников

Приложение А

Приложение Б

Приложение В

Приложение Г

Приложение Д