Оглавление

[Оглавление 1](#_Toc134651873)

[Проектирование программного обеспечения 2](#_Toc134651874)

[Проектирование базы данных 3](#_Toc134651875)

[Проектирование клиентской части 4](#_Toc134651876)

[*React* 4](#_Toc134651877)

[Redux 5](#_Toc134651878)

[Webpack 6](#_Toc134651879)

[Архитектура клиентской части 6](#_Toc134651880)

[Проектирование серверной части 7](#_Toc134651881)

[ASP.NET Core 7](#_Toc134651882)

[OWIN 7](#_Toc134651883)

[Выводы по разделу 7](#_Toc134651884)

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.

Архитектура приложения

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



Рисунок 2.1 – Структурная диаграмма основных таблиц

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.

Проектирование клиентской части (обоснование используемых технологий)

*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.

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 мы поговорим в разделе архитектура клиентской части.

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.

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 данных, 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.

Проектирование серверной части

Серверная часть приложения 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, а не на микросервисной.   
 Благодаря архитектуре OWIN, в ASP.NET приложениях появился конвеер обработки запроса. Это удобное решение для проектирования архитектуры, поскольку позволяет расположить несколько обработчиков, которые будут последовательно обрабатывать запрос и собирать ответ для клиента.

Но помимо обычного

Архитектура серверной части приложения изображена на рисунке 2.3.

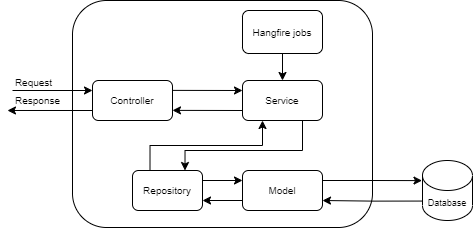


Рисунок 2.3 – Архитектура серверной части приложения

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.

Модели базы данных для программного средства представлены на рисунке 2.1.



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 А.

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

Таблица 2.1 ­­­– Свойства таблицы Account

|  |  |
| --- | --- |
| Название поля | Описание |
| р |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

Выводы по разделу

