**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное учреждение высшего образования**

**Национальный исследовательский ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ**

**«МИСИС»**

**Курсовая работа**

**«Технология разработки ПО»**

**Выполнил студент группы БИСТ-21-1:**

**Мангушев Ильдар Надирович**

**Преподаватель:**

**Карпишук Александр Васильевич**

Москва

2024

Оглавление

[1. Предметная область 3](#_Toc165135998)

[2. Постановка задачи 3](#_Toc165135999)

[3. Определение трудозатрат на разработку 4](#_Toc165136000)

[4. Описание архитектуры 7](#_Toc165136001)

[5. Описание структуры БД 10](#_Toc165136002)

[6. Описание серверной части 11](#_Toc165136003)

[8. Заключение 21](#_Toc165136004)

[9. Список литературы 23](#_Toc165136005)

1. **Предметная область**

В качестве предметной области, была выбрана тема «Коммерческая клиника». Коммерческая клиника предоставляет широкий спектр медицинских услуг своим пациентам. Однако, в настоящее время клиника сталкивается с рядом проблем, связанных с управлением пациентами, медицинскими записями и финансовыми операциями. Для решения этих проблем требуется разработка клиент-серверного приложения, которое автоматизирует и оптимизирует основные процессы клиники.

1. **Постановка задачи**

Коммерческая клиника, специализирующаяся на медицинских услугах, требует разработки клиент-серверного приложения, которое поможет автоматизировать и оптимизировать процессы, связанные с управлением пациентами, расписанием врачей.

**Функциональные требования:**

1. Регистрация пациентов: Приложение должно предоставлять возможность регистрации новых пациентов, сбора и хранения их персональных данных (ФИО, контактная информация, страховая информация и прочее).
2. Управление расписанием врачей: Приложение должно позволять администраторам клиники управлять расписанием работы врачей, включая добавление, изменение и удаление сеансов приема пациентов.
3. Запись на прием: Пациентам должна быть предоставлена возможность записаться на прием к выбранному врачу в удобное для них время. Приложение должно автоматически проверять доступность выбранного времени и предотвращать конфликты.
4. Управление медицинскими записями: Врачам и медицинскому персоналу должна быть предоставлена возможность создания, просмотра и редактирования электронных медицинских записей пациентов, включая историю болезни, результаты обследований и назначенные лечебные процедуры.

**Нефункциональные требования:**

1. Безопасность: Приложение должно обеспечивать высокий уровень безопасности данных пациентов и медицинской информации, включая защиту от несанкционированного доступа и шифрование данных.
2. Масштабируемость: Приложение должно быть способно масштабироваться для обработки растущего числа пациентов, врачей и медицинских записей.
3. Интерфейс пользователя: Приложение должно иметь интуитивно понятный интерфейс, который обеспечивает легкость использования для администраторов клиники, врачей и пациентов.

**Технологические требования:**

1. Язык программирования: Разработка должна быть выполнена с использованием TypeScript.
2. Фреймворк: рекомендуется использовать NestJS для разработки серверной части приложения.
3. База данных: Приложение должно использовать реляционную базу данных, такую как PostgreSQL, для хранения данных пациентов, расписания врачей и медицинских записей.
4. Аутентификация и авторизация: Приложение должно обеспечивать механизм аутентификации и авторизации пользователей, включая разграничение прав доступа между администраторами, врачами и пациентами.
5. **Определение трудозатрат на разработку**

| **Этап** | **Время, ч** | **Выявлено ошибок этапа** |  |  |
| --- | --- | --- | --- | --- |
|  |  | Анализа | Проектирования | Кодирования |
| Анализ | 24 | 1 | X | X |
| Проектирование | 60 | 1 | 0 | X |
| Кодирование | 60 | 0 | 0 | 0 |
| Тестирование | 3 | 0 | 0 | 0 |

Tk=147ч

Определим согласно правилам подсчета значение SLOC, занося промежуточные значения в таблицу:

| **Операторы** | **Время, ч** |
| --- | --- |
| if | 197 |
| for | 140 |
| ; | 783 |
| {} | 2942 |
| return | 65 |

### ****Таким образом, SLOC = 4 127. К размерноориентированным метрикам относится метрика строки кода (source lines of code, SLOC) – совокупное количество строк кода в модулях программы. Различают физические (physical) и логические (logical) строки кода****

Для расчета метрик Холстеда выделим уникальные операторы и операнды и приведем их общее количество.

| **№** | **Операторы** | **Количество** |
| --- | --- | --- |
| 1 | int | 1235 |
| 2 | ; | 783 |
| 3 | void | 2 |
| 5 | = | 3011 |
| 6 | () | 602 |
| 7 | . | 12729 |
| 8 | funcs | 19 |
| 12 | for | 140 |
| 13 | {} | 2942 |
| 14 | [] | 119 |
| 15 | < | 662 |
| 16 | > | 1294 |
| 17 | “ | 20000 |
| 18 | ++ | 23 |
| 19 | = | 3011 |
| 20 | return | 65 |
| 22 | \* | 140 |
| 23 | - | 7002 |
| 24 | / | 9762 |
| 25 | % | 14 |
| 26 | ≥ | 16 |
| 27 | import | 215 |
| 28 | string | 152 |
| 29 | Module | 1194 |
| 30 | from | 297 |
| 31 | private | 32 |
| 32 | const | 94 |
| 33 | export | 52 |
| 34 | async | 49 |
| 35 | <> | 662 |

| **№** | **Операнды** | **Количество** |
| --- | --- | --- |
| 1 | i | 14 |
| 2 | j | 7 |
| 3 | n | 3 |
| 4 | result | 14 |
| 6 | columns | 50 |
| 7 | rows | 23 |

Таким образом, ƞ1 =17 663, N1 =3 442, ƞ2 =17 542, N2 =3 142.

Рассчитаем метрики.

Словарь программы (program vocabulary, ƞ) – суммарное количество уникальных операторов и операндов в программе:

ƞ = 35 205

Длина программы (program length, N) – суммарное количество операторов и операндов в программе: N = 6 584

Оценочная длина программы (estimated program length, N̂ ) – ожидаемая сумма операторов и операндов в программе:

\overline N =17 663\* log2(17 663) + 17 542\* log2(17 542) = 496 513,76

Объем программы (program volume, V) – размер реализации алгоритма вне зависимости от конкретного языка программирования: V=6584\*log2(35205) = 99 441,39

Сложность программы (program difficulty, D) – мера сложности написания (и понимания) программного кода:

D= (17663/2) \*(3142/17542)= 1 581,836

Трудоемкость написания программы (program effort, E) – суммарное количество усилий, затрачиваемое на написание программы:

E=2949,13\*115,63=1 151 713 870,59

Прогноз времени реализации (time required to program, T) – оценка времени (в часах), необходимого для написания программы: T=1 151 713 870,59/64800=17 773,36

Прогноз числа ошибок (тumber of delivered bugs, B) – оценка потенциального количества ошибок, допущенных при реализации кода:

B=33,14713

Производительность работы программиста (programmer productivity, PP) – отношение количества строк в коде (SLOC) ко времени, затраченному на его разработку (TК):

PP=**4 127**/147=28,075

Эффективность обнаружения ошибок (defect detection efficiency, DDE) — отношение количества ошибок, внесенных и обнаруженных в течение одного этапа разработки, к общему числу ошибок, внесенных на данном этапе:

DDE1=1/1\*100 = 100

DDE1=1/0\*100=0

DDE1=0/0\*100=0

Для расчета удельной стоимости кода определим величину финансовых затрат исходя из стоимости одного часа работы программиста (условно, 1000 руб.):

Ck=1000\*147=147000

Удельная стоимость кода (СУД) – отношение финансовых затрат на разработку кода (CК) к количеству строк в нем:

Cуд=147000/4127=35,62 руб/строку

1. **Описание архитектуры**

Клиентская часть:

**Изображение выглядит как текст, Веб-сайт, веб-страница, программное обеспечение

Автоматически созданное описание**

**1** главная страница сайта

**Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание**

2 каталог услуг

**Изображение выглядит как снимок экрана, текст, Человеческое лицо, человек

Автоматически созданное описание**

3 список врачей

Несмотря на то, что у меня имеется верстка клиентской части, она не подключена к архитектуре сервера, в связи с тем, что я провел двухнедельную стажировку в школе 21. Но, если бы мне представилась возможность в подключении frontend’a и backend’а, я бы воспользовался фреймворком React.

Приступим к рассмотрению архитектуры серверной части (рис. **4**).Сервер является ответственным за обработку запросов, выполнение бизнес-логики, взаимодействии с базой данных и обеспечение механизма безопасности, и аутентификации, что обеспечивает доступ к конфиденциальной информации только авторизованным пользователям. Список возможностей серверного слоя:

* Обработка запросов. Серверный слой принимает запросы от клиентского интерфейса, включая запросы на создание, чтение, обновление и удаление информации о пациентах, медицинских записях, расписании врачей. Он обеспечивает маршрутизацию запросов и передачу данных между клиентом и сервером.
* Взаимодействие с базой данных. Серверный слой обеспечивает взаимодействие с базой данных для сохранения и извлечения данных о пациентах, медицинских записях, расписании врачей и других клиник это же сети. Он использует ORM (Object-Relational Mapping) или другие подходы для работы с базой данных, включая выполнение запросов, создание, обновление и удаление записей.
* Обеспечение безопасности. Серверный слой обеспечивает механизмы

безопасности и аутентификации, чтобы гарантировать, что только авторизованные пользователи могут получить доступ к конфиденциальной информации и выполнить определенные действия. Он может использовать механизмы аутентификации на основе токенов (например, JWT)

* Логирование и мониторинг: Серверный слой может включать легирование и мониторинг для отслеживания действий и состояния приложения. Это позволяет выявлять проблемы, отслеживать производительность и обеспечивать надлежащее функционирование приложения.

Для более удобного взаимодействия и тестирования серверного слоя я использовал набор инструментов Swagger или также известный как OpenAI. Swagger является набором инструментов и спецификаций для разработки, документирования и взаимодействия с веб-сервисами API. Он позволяет описывать структуру и функциональность API, определять доступные методы, параметры, форматы данных и другую информацию, которая позволяет разработчикам и клиентам легко понять и использовать API.

**Изображение выглядит как текст, линия, Шрифт, число

Автоматически созданное описаниеИзображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание**

**Изображение выглядит как текст, число, Шрифт, линия

Автоматически созданное описание**

**4** серверная часть

1. **Описание структуры БД**

С целью эффективного хранения и управления информацией о записях, клиентов, докторов и других клиник данной сети, я выбрал реляционную базы данных PostgreSQL. Это мощная реляционная база данных с открытым исходным кодом. Он предлагает множество функций и возможностей, делая его популярным выбором для разработчиков и организаций. Ее интуитивно понятный интерфейс и гибкость в настройке оказались идеальными, что позволило без проблем внедрить данное решение в рамках данного клиент-серверного приложения.

Изображение выглядит как текст, снимок экрана, диаграмма

Автоматически созданное описание

5 Диаграмма классов

1. **Описание серверной части**

В серверной части моего проекта находится папка с конфигурациями, в которой происходит соединение с базой данных. Дополнительно в папке src содержаться миграции (в контексте базы данных представляют собой автоматизированный процесс изменения схемы базы данных. Они используются для управления эволюцией структуры базы данных, включая создание, изменение и удаление таблиц, столбцов, индексов и других элементов схемы), а также 5 сущностей (клиент, доктор, клиника, запись и пользователь) и соответственной папка, в которой мы обрабатывает получение токена и обработку авторизации. Рассмотрим подробнее каждую часть:

Рассмотрим сущность Doctor рассмотрим в ней классы, аналогичные классы с таким же функционалом присутствуют в классах Client, Clinic, Record, User.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

6 creatDcotorDto

*Класс createDcotorDto* используется для создания объекта-представления данных при создании нового врача в системе. Он использует декораторы ApiProperty из модуля @nestjs/swagger, чтобы задать метаданные для каждого поля класса.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Автоматически созданное описание

7 Doctor

Этот код представляет сущность (Entity) Doctor для работы с врачами в рамках приложения коммерческой клиники. Каждое поле сущности имеет соответствующие аннотации для использования с TypeORM и декораторы ApiProperty из модуля @nestjs/swagger

Сущность Doctor связана с другими сущностями через аннотации TypeORM, такие как ManyToMany и OneToMany, чтобы определить отношения между врачами, клиниками и записями пациентов.Эта сущность используется для описания модели данных врача в базе данных и предоставляет информацию о врачах в рамках API коммерческой клиники.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

8 DoctorModel

Класс DoctorModel связывает все классы обьекта Doctor и предоставляет доступ к их функционалу извне.

* controllers (контроллеры): DoctorsController и DoctorsControllerDTO являются контроллерами, которые обрабатывают HTTP-запросы, связанные с врачами. DoctorsController предоставляет основные CRUD-операции для врачей, а DoctorsControllerDTO предоставляет операции, связанные с DTO (Data Transfer Object) для врачей.
* providers (провайдеры): DoctorsService является провайдером, который содержит бизнес-логику и методы для работы с данными врачей.
* imports (импорты): DatasourceModule является импортированным модулем и предоставляет доступ к необходимым источникам данных. TypeOrmModule.forFeature импортирует сущности, связанные с врачами (Doctor), клиниками (Clinic), пациентами (Client) и записями (Record), чтобы они были доступны для использования в модуле.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

9 IncompleteDoctorDto

При условии того, что пользователь не авторизовался ему будет выходить не полная информация о враче. Данный класс существует как раз для незарегистрированных пользователей

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

10 DoctorController

Этот код представляет контроллер DoctorsController, который отправляет HTTP-запросы, связанные с врачами, в рамках Nest.js-приложения.

* findAll(): Обрабатывает GET-запрос на /doctors и вызывает метод findAll() из сервиса DoctorsService, чтобы получить список всех врачей.
* findOne(): Обрабатывает GET-запрос на /doctors/:office, где :office - параметр, представляющий номер кабинета врача. Вызывает метод findOne() из сервиса DoctorsService, чтобы найти врача с указанным номером кабинета.
* update(): Обрабатывает PUT-запрос на /doctors/:id, где :id - параметр, представляющий идентификатор врача. Принимает обновленные данные врача в теле запроса и вызывает метод update() из сервиса DoctorsService, чтобы обновить информацию о враче.
* create(): Обрабатывает POST-запрос на /doctors и принимает данные нового врача в теле запроса. Вызывает метод create() из сервиса DoctorsService, чтобы создать нового врача.
* remove(): Обрабатывает DELETE-запрос на /doctors/:id, где :id - параметр, представляющий идентификатор врача. Вызывает метод remove() из сервиса DoctorsService, чтобы удалить врача с указанным идентификатором.

Контроллер также использует декораторы Swagger) для описания операций и безопасности в документации Swagger. Также в контроллере применяется Guard (AuthGuard) для защиты этих операций аутентификацией с помощью JWT-токенов.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

11 DoctorControllerDto

Имеет такой же функционал как и класс DoctorController, только он отправляет запросы от неавторизованных пользователей.

Теперь рассмотрим процесс авторизации и аутентификации, который я реализовал в папке auth.

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

12 AuthController

Данный класс принимает запросы на авторизацию проверяет сужествует ли пользователь

Изображение выглядит как текст, снимок экрана

Автоматически созданное описание

13 AuthModule

Данный класс который выполняет аналогичные функции, как и остальные классы module.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

14 AuthService

Класс, который обрабатывает задачи, связанные с аутентификацией.

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

15 AuthenticatedGuard

Код, который вы предоставили, является классом AuthenticatedGuard на языке TypeScript, который реализует интерфейс CanActivate. Данный класс представляет собой гвард Nest.js, используемый для проверки аутентификации пользователя. Результатом метода canActivate является логическое значение true или false, которое указывает, должен ли быть разрешен доступ к защищенному маршруту или нет.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

16 JwtAuthGuard

**Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание**

17 JwtStrategy

Конструктор класса JwtStrategy принимает зависимость authService типа AuthService и вызывает конструктор родительского класса PassportStrategy. В конструкторе родительского класса PassportStrategy передаются опции для настройки стратегии.

Метод validate асинхронно вызывается при проверке и валидации JWT. Он принимает объект payload типа JwtPayload, который содержит информацию, извлеченную из JWT. Метод вызывает validateUser из authService, чтобы проверить и валидировать пользователя на основе payload. Если пользователь не найден.

Изображение выглядит как текст, Шрифт, снимок экрана, линия

Автоматически созданное описание

18JwtPayload

1. **Описание клиентской части**

Клиентская часть приложения включает в себя главную страницу(рис **1**), которая предоставляет пользователям неполную информацию о клинике, и предоставляет пользователю возможность переключаться на другие страницы, в которых описана другая информация.

**Изображение выглядит как текст, Веб-сайт, веб-страница, программное обеспечение

Автоматически созданное описание**

**19** главная страница сайта

Страница услуги (рис. 2) является непосредственной частью сайта, которая позволяет пользователю записаться к нужному врачу или на нужный осмотр.

**Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание**

20 каталог услуг

Страница “Наши врачи” (рис. 3) предоставляет пользователю нашего сайта ознакомиться со всеми врачами, которые работают в данной клинике. Узнать о их стаже, узнать о их заслугах и просто их историю, однако для получения полной информации, пользователю нужно будет авторизоваться на приложении.

**Изображение выглядит как снимок экрана, текст, Человеческое лицо, человек

Автоматически созданное описание**

21 список врачей

1. **Заключение**

В данной работе была рассмотрена предметная область коммерческой клиники и выявлены проблемы, связанные с управлением пациентами, медицинскими записями, расписанием врачей и финансовым учетом. Для решения этих проблем было разработано клиент-серверное приложение, которое автоматизирует и оптимизирует основные процессы клиники.  
**В ходе работы были использованы следующие инструменты и технологии:**

**Язык программирования**: Для разработки приложения был выбран TypeScript, который обеспечивает типизацию и улучшенную разработку при помощи статического анализа кода.  
 **Фреймворк**: Для реализации серверной части приложения был использован фреймворк NestJS, который обладает мощными возможностями для построения масштабируемых и модульных приложений.  
**База данных**: Для хранения данных пациентов, медицинских записей и расписания врачей была выбрана реляционная база данных PostgreSQL, которая обеспечивает надежность и эффективность при работе с большим объемом данных.  
  
**Аутентификация и авторизация**: Для обеспечения безопасности и контроля доступа в приложении был использован механизм аутентификации и авторизации с помощью токенов JWT (JSON Web Tokens).  
  
**Облачное развертывание**: Приложение было развернуто в облачной среде AWS (Amazon Web Services) для обеспечения масштабируемости, отказоустойчивости и доступности.  
  
**В результате разработки были достигнуты следующие результаты:**  
**Централизованная система учета пациентов:** Приложение позволяет регистрировать новых пациентов, хранить и управлять их персональными данными, контактной информацией и страховой информацией. Это обеспечивает быстрый доступ и поиск необходимой информации.  
  
**Оптимизация управления расписанием врачей**: Администраторы клиники могут управлять расписанием работы врачей, добавлять, изменять и удалять сеансы приема пациентов. Это позволяет избежать перекрытий и ошибок в расписании, облегчая процесс планирования.  
  
**Улучшение доступа к медицинской информации**: Врачи и медицинский персонал получают быстрый доступ к полной медицинской истории пациентов, что повышает качество предоставляемых медицинских услуг.  
  
**Улучшение финансового учета:** Приложение позволяет эффективно выставлять счета, отслеживать оплату медицинских услуг и взаимодействовать со страховыми компаниями. Это облегчает учет финансовых операций клиники.  
Разработанное клиент-серверное приложение в предметной области коммерческой клиники успешно решает проблемы управления пациентами, медицинскими записями, расписанием врачей и просмотр адресов клиник данной сети. Оно обеспечивает удобство, эффективность и надежность ведения клинической деятельности, способствуя улучшению качества предоставляемых медицинских услуг и оптимизации работы клиники.

1. **Список литературы**

1**.**https://github.com/mbatimel/education/tree/ea9fb81109be3f72e3242197c7ae0930f112f325/src

2**.** https://docs.nestjs.com/security/authorization

3. <https://typeorm.io/many-to-one-one-to-many-relations>

4. <https://www.passportjs.org/packages/passport-jwt/>

5. https://www.youtube.com/watch?v=wdsp7BNmJRc&t=962s