1. Введение.

1.1. Обоснование выбора темы.

1.2. Цели и задачи проекта.

1.3. Актуальность исследования.

2. Техническое задание.

2.1. Описание функциональности веб-приложения.

2.1. Определение требований к пользовательскому интерфейсу.

3. Выбор технологий.

3.1. Использование фреймворка Vue.js

3.2. Использование библиотеки Vue Router.

3.3. Использование библиотеки Vuex.

4. Разработка пользовательского интерфейса.

4.1. Использование Vue компонентов для создания интерфейса.

4.2. Хуки жизненного цикла.

4.3. Интеграция Google Maps для отображения расположения зубных клиник.

5. Авторизация.

5.1. Реализация системы авторизации с использованием Firebase Authentication.

5.2. Обеспечение безопасности данных.

6. База данных и хранение данных.

6.1. Использование Firebase для хранения данных.

7. Методы создания адаптивного сайта.

7.1. Подключение стилей, использование препроцессоров.

7.2. Подключение шрифтов.

7.3. Адаптивный сайт.

7.4. Требования к дизайну.

8. Улучшение пользовательского опыта, создание небольшой игры.

9. Развертывание приложения.

9.1. GitLab.

9.2. CI/CD.

Заключение.

Сводка выполненной работы.

Результаты и достижения проекта.

Перспективы развития.

Возможные улучшения и расширения функционала приложения.

11. Список использованных источников.

12. Приложения.

# **1. Введение**

1.1. Бережное отношение к нашему здоровью это путь к полноценной жизни. Очень важно сохранить его и не сокрушаться в будущем. Необходимо закладывать привычку заботиться о себе с самого детства. Как много маленьких пациентов не хотят идти к врачу, не хотят сдавать анализы или со слезами делают уколы и прививки. А поход к зубному врачу для многих детей и родителей это испытание.

1.2. Разработка современного приложения позволит существенно улучшить качество обслуживания пациентов. Важной составляющей данного проекта является создание удобного и интуитивно понятного интерфейса, способствующего максимальному комфорту как для пациентов, так и для сотрудников клиники. Легкость в поиске информации о клинике, врачах, контактах и других аспектах, которые могут понадобиться потенциальному клиенту.

В хорошем приложении должны быть такие неотъемлемые части как:

* Понятный и простой интерфейс.
* Удобство записи к врачам и отмены посещения. Возможностью самостоятельного выбора способа напоминания о предстоящем визите (по смс, по эл. почте, на вотсап или в телеграм).
* Возможность узнать отзывы о клинике и врачах.

В современном мире информационных технологий очень важно иметь доступное, понятное и актуальное веб-приложение для бизнеса. В этом контексте, разработка современного веб-приложения для стоматологической клиники представляет собой актуальную задачу, направленную на оптимизацию процессов в медицинской сфере.

1.3 Актуальность

С каждым годом уровень цифровизации во всех областях возрастает, и важность внедрения современных информационных технологий становится все более очевидной. Веб-приложение для клиники открывает новые перспективы во взаимодействии клиента и клиники.

# **2. Техническое задание**

## 2.1. Описание функциональности веб-приложения.

* Авторизация и регистрация. Здесь буду использовать сервис от Google Firebase.
* Уведомления и система напоминаний. Автоматические уведомления о предстоящих приемах. Google Firebase предоставляет такой сервис.

## 2.2. Определение требований к пользовательскому интерфейсу.

* Интуитивно понятный интерфейс. Легкость навигации и использования.
* Адаптивный дизайн для использования на различных устройствах.
* Использование возможностей API для удобной подачи информации. В дипломном проекте будет реализован API Google Maps для удобного информирования и визуального представления о нахождении клиник.
* Важность фронтенда.

Фронтенд - это то, что любой пользователь интернета и клиент видит в первую очередь, это интерфейс, через который пользователи взаимодействуют с системой. Он играет решающую роль в формировании первого впечатления от приложения, а также в том, насколько удобно и эффективно пользователи смогут взаимодействовать с функционалом. Поэтому сложно переоценить важность этого компонента системы.

Существует несколько видов веб-приложений:

SPA (Single Page Application) — одностраничное приложение, которое не требует перезагрузки основной страницы во время своей работы. В таком приложении браузер загружают сразу весь код, но отображается только конкретный компонент, а при переходе по ссылкам, открывает заранее загруженные данные. Поэтому работа такого приложения более быстрая.

Плюсы:

* Удобный интерфейс. Пользователю легко ориентироваться странице, где весь контент размещен на одном экране и есть несколько переходов по страницам.
* Высокая скорость. Так как запрос на сервер происходит только один раз в самом начале, тогда и подгружается весь код, то во время работы, не происходит обновление страницы при каждом взаимодействии с приложением, а приложение лишь подгружает данные, которые были загружены в самом начале.
* Кэширование данных. Как было написано в предыдущем пункте, вся информация подгружается сразу поэтому после загрузки приложение может работать даже без доступа к интернету.
* SPA может обмениваться данными с сервера с помощью ajax-запросов. Под AJAX подразумевают любое общение с сервером без перезагрузки страницы, организованное при помощи JavaScript. Так, например, в социальных сетях без обновления страницы появляются уведомления о новых сообщениях.

Минусы:

* Недостаточная SEO-оптимизация. Поисковые системы при обработке запросов опираются на заголовки, описание и другие мета-теги(мета-заголовок). У SPA всего одна страница, поэтому разработчик может привязать к ней только один поисковый запрос.
* Недоступность таких сервисов, как Google Аналитика или Яндекс Метрика.

MPA (Multi Page Application) — многостраничное веб-приложение. Такое приложение содержит много страниц и имеет сложную архитектуру. Страницы, как правило имеют статический контент и множество ссылок на внутренние ресурсы. При каждом новом запросе страница загружается заново.

Плюсы:

* Хорошая SEO оптимизация. У разработчика есть возможность на каждую страницу добавить мета-теги, заголовки.
* Неограниченная масштабируемость. В таких приложения можно создавать неограниченно много страниц и создавать сложную вложенность.

Минусы:

* Низкая скорость. При каждом взаимодействии с приложением, сервер перезагружает и обновляет все ресурсы — HTML, CSS, скрипты и все данные страницы.
* Более сложная и кропотливая разработка.

# **3. Выбор технологий.**

## 3.1. Использование фреймворка Vue.js.

В рамках данного дипломного проекта планируется использование современных технологий в сфере фронтенд-разработки. Применение препроцессора sass позволит создать высококачественный и адаптивный пользовательский интерфейс, а фреймворк Vue.js обеспечит плавную и динамичную работу приложения, что существенно повысит его функциональность и привлекательность для пользователей. Объединение этих технологий позволит создать актуальную платформу для информации в стоматологической клинике, что в свою очередь приведет к улучшению взаимодействия с клиентами.

Vue.js имеет много преимуществ таких как:

* Простота использования.

Vue.js интуитивно понятен и обладает легким синтаксисом, что делает его прекрасным выбором для начинающих.

* Гибкость и масштабируемость.

Во Vue.js можно легко встроить другие библиотеки, что открывает много возможностей для написания быстрого и качественного кода. А использование Google Maps API и Firebase с Vue.js может быть реализовано с минимальными усилиями. Масштабируемость легко осуществляется за счет концепции использования компонентов.

* Хороший официальный сайт с подробным описанием.

По Vue.js легко найти понятную документацию и поддержку от других разработчиков.

* Инструменты для разработчиков.

Такой инструмент, как Vue DevTools, позволяет эффективно отлаживать приложение на этапе разработки.

* Использование компонентов.

Vue.js построен на концепции компонентов, что облегчает структурирование кода и позволяет повторно использовать компоненты в различных частях кода.

* Поддержка одностраничных приложений (SPA).
* Активная разработка.

Vue.js является современным и популярным фреймворком, его разработка продолжается.

* Эффективная работа с DOM:

Vue.js обеспечивает эффективную работу с DOM, что способствует быстрой отрисовке интерфейса и обеспечивает плавное взаимодействие с приложением.

* Реактивность.

Vue может реагировать на изменение данных в автоматическом режиме. Реактивная система отслеживает зависимости между различными частями данных и автоматически обновляет зависимые части, когда исходные данные изменяются. Реактивность позволяет мгновенно обновлять пользовательский интерфейс или другие части системы при изменении данных без явного вмешательства программиста.

Как сделать сайт более быстрым. Для создания быстрого веб-приложения, следует учитывать ряд основополагающих принципов и применять оптимизационные техники: Оптимизация изображений и мультимедийных файлов: Используйте сжатие изображений и видео без потери качества. Избегайте загрузки изображений большого разрешения, если они не требуются в таком формате. Ленивая загрузка контента: Загружайте контент только когда он необходим пользователю. Например, изображения за пределами видимой области могут загружаться только при прокрутке. Минимизация HTTP-запросов: Объединяйте файлы CSS и JavaScript, чтобы сократить количество запросов к серверу. Используйте CSS-спрайты для сокращения числа запросов к изображениям. Кеширование ресурсов: Используйте HTTP-кэширование для временного хранения ресурсов на стороне клиента. Это позволяет браузеру использовать ранее загруженные данные вместо их повторной загрузки. Минимизация и объединение CSS и JavaScript файлов: Удалите избыточные пробелы, комментарии и лишние символы из файлов. Объедините несколько файлов в один, чтобы уменьшить количество запросов. Использование CDN (Content Delivery Network): Используйте сети доставки контента для хранения и распространения ресурсов (например, библиотек JavaScript, стилей) на сервера распределенные по всему миру. Это ускорит загрузку ресурсов. Оптимизация серверной части: Поддерживайте эффективный хостинг, используйте кэширование на сервере, уменьшите время обработки запросов. Использование асинхронных запросов: Используйте асинхронные запросы для загрузки данных и контента на страницу, чтобы не блокировать остальной контент во время загрузки. Использование шрифтов локально: Если используются веб-шрифты, предпочтительно загружать их с сервера вместо подключения через сторонние сервисы, что может привести к дополнительной задержке. Регулярные тесты и анализ производительности: Регулярно тестируйте производительность своего приложения с помощью инструментов, таких как Google PageSpeed Insights, и исправляйте выявленные узкие места. Использование сжатия GZIP/Deflate: Настройте сервер так, чтобы он отправлял данные сжатыми, что уменьшит объем передаваемой информации. Оптимизация базы данных: Оптимизируйте запросы к базе данных и используйте индексы для ускорения поиска данных. Соблюдение этих принципов и техник поможет создать быстрое и отзывчивое веб-приложение для стоматологической клиники, что повысит удовлетворенность пользователей и эффективность работы персонала.

## 3.2. Vue Router представляет собой официальную библиотеку для управления маршрутизацией в приложениях Vue.js. Она позволяет управлять навигацией между разными views/pages или компонентами веб-приложения. Вот несколько ключевых аспектов и возможностей Vue Router, которые могут быть полезными при разработке веб-приложения для зубной клиники:

* Декларативная Навигация:

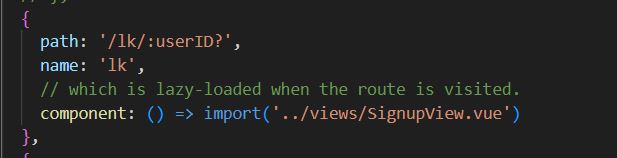
Vue Router позволяет определять маршруты в декларативном стиле, что облегчает создание и поддержание маршрутизации в вашем приложении.

* Вложенные маршруты.

Некоторые пользовательские интерфейсы приложений состоят из вложенных на нескольких уровнях компонентов.

* Динамические маршруты.

Vue Router поддерживает динамические маршруты, что позволяет передавать параметры в адресной строке и обрабатывать их в компонентах. Так при входе в личный кабинет к пути добавляется userID. Так же на скриншоте ниже показан вариант ленивой загрузки – когда изначально компонент не загружается при запуске веб приложения, загрузка происходит тогда, когда пользователь сам обращается к этому контенту.



* Режим истории и хеширования.

Vue Router поддерживает как режим истории (history mode), который убирает хэш из URL, так и режим хеширования (hash mode), что полезно при развертывании на статических хостингах. Статический хостинг позволяет размещать статические файлы (HTML, CSS, JavaScript, изображения и т. д.) без необходимости запуска сервера. Это отличное решение для проектов, которые не требуют серверной логики, баз данных или других бэкенд-сервисов.

Режим истории использует API истории браузера для манипуляции. В этом режиме URL выглядит чистым и естественным. При использовании этого режима необходимо настроить корректную обработку запросов к несуществующим маршрутам.

Режим хеширования добавляет метку (хеш) к URL #. Хеш-фрагмент не отправляется на сервер, поэтому такие приложения легче развертывать на статических хостингах. Режим хеширования обеспечивает совместимость с браузерами, не поддерживающими режим истории. В отличие от динамических веб-сайтов, где контент формируется на сервере во время запроса пользователя, статические веб-сайты имеют заранее подготовленные файлы

* Параметры запроса и запись состояния.

Vue Router позволяет передавать параметры запроса в адресной строке URL, такие как: example.com/path?param1=value1&param2=value2. Эти параметры могут использоваться для передачи данных между маршрутами или для фильтрации и сортировки контента.

* Удобство использования с Vue.js.

Поскольку Vue Router является официальной частью экосистемы Vue.js, его использование полностью интегрировано с фреймворком.

* Совместимость с Vuex.

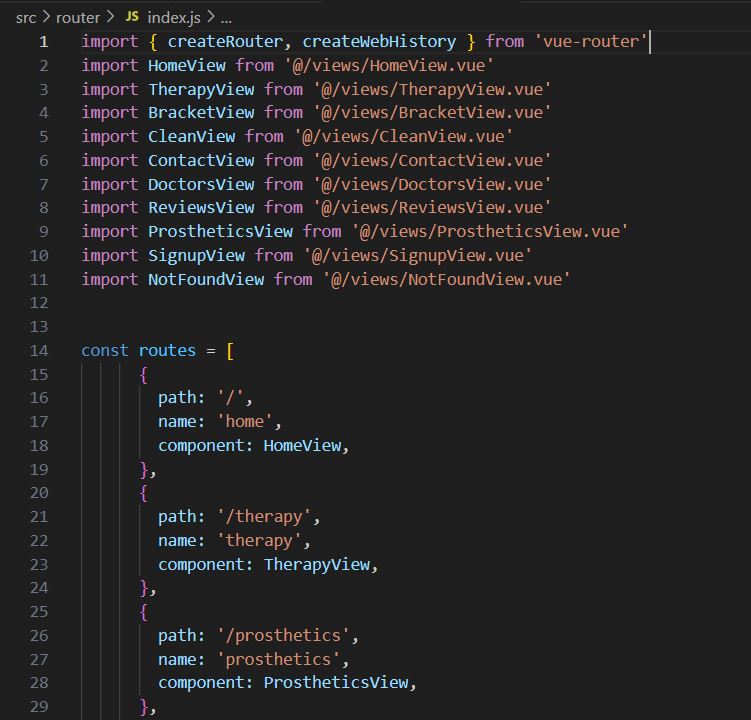
В случае использования Vuex для управления состоянием, Vue Router легко интегрируется с хранилищем, предоставляя средства для более сложных сценариев управления состоянием в приложении.

Для интеграции Vue Router в проект необходимо выполнить следующие шаги:

* Установить библиотеку одной из команд: npm install vue-router или yarn add vue-router.
* Импорт и подключение Vue Router в main.js:



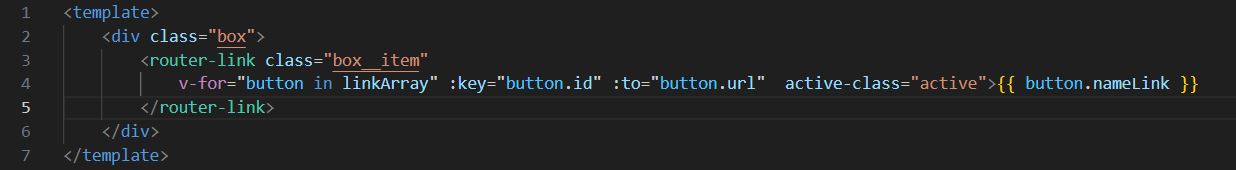
* В папке router/index.js выполняется импорт роутера. В этом же файле мы создаем переменную router, в которой будут храниться объекты – представляющие страницы приложения (с указанием адреса ссылки, названия страницы и связанного компонента).



* Для странички NotFound путь будет выглядеть так:



Для навигации в приложении создаем компонент NavComp, который будет отвечать за переход по страницам.



С помощью компонента <router-link> мы создаем ссылку, благодаря этому мы сможем изменять URL без перезагрузки страницы. Свойство to определяет URL для перехода, так же добавляем active-class, который будет отвечать за стилистику активной кнопки. Затем в главном компоненте App.vue подключаем компонент NavComp и <router-view>, который отвечает за отображение, соответствующего компонента.

3.3. Использование Vuex.

В приложениях может быть сложно эффективно управлять обменом данными между компонентами особенно если они находятся в разных частях приложения и не связаны напрямую. В таких сценариях использование стандартных методов передачи данных между компонентами, таких как $emit и props во Vue, может быть неудобным и привести к неструктурированному коду и сложной поддержке.

В таких случаях удобно использовать хранилище данных, которое будет централизованно управлять состоянием приложения. Это хранилище позволяет компонентам получать доступ к данным, необходимым именно им, и устраняет необходимость передавать данные через множество промежуточных компонентов.

Преимущества такого подхода включают:

* Централизованное хранение данных: Данные выносятся из компонентов и хранятся в едином месте (хранилище), что облегчает управление состоянием приложения.
* Управление изменениями данных: Хранилище обеспечивает мутации, которые позволяют изменять данные. Это делает возможным обновление данных в разных частях приложения.
* Реактивность и обновление компонентов: При изменении данных все компоненты, зависящие от этих данных, автоматически обновляются.
* Управление состоянием приложения: Vuex может облегчить отслеживание и изменение состояния приложения, особенно в сложных приложениях с множеством компонентов.

Основные понятия:

State (Состояние): Этот объект представляет собой единый источник данных для всего приложения. Состояние доступно из любого компонента в приложении. Обычно состояние в Vuex описывается в виде объекта, содержащего различные свойства и значения, представляющие текущее состояние приложения.

Mutations (Мутации): Мутации - это синхронные функции, предназначенные для изменения состояния. Они являются единственным способом изменения данных в хранилище. Мутации принимают текущее состояние и параметры, которые необходимо изменить, и вносят изменения в состояние.

Actions (Действия): Действия используются для выполнения асинхронных операций или сложной бизнес-логики перед изменением состояния через мутации. Действия запускаются из компонентов и могут содержать логику, такую как обработка сетевых запросов.

Getters Геттеры (): Геттеры используются для получения данных из состояния в хранилище. Они предоставляют вычисляемые значения, которые можно использовать в компонентах.

Для установки данного приложения используем команду “npm install vuex”, в некоторых случаях, возможно понадобится указать версию, например “npm install vuex@3”. Далее в папке src создаем папку store и в ней файл index.js, там и будет хранится состояние нашего приложения. 

Далее не забываем подключить данный файл в main.js.

# **4**. **Разработка пользовательского интерфейса.**

4.1. Использование Vue компонентов для создания интерфейса.

Использование компонентов в Vue.js - это ключевая концепция, которая позволяет создавать модульный и переиспользуемый код. Компоненты представляют собой изолированные блоки кода, включающие в себя HTML, CSS и JavaScript, и они могут вкладываться друг в друга для создания сложных пользовательских интерфейсов.

Родительский компонент может передавать данные дочернему компоненту через атрибуты props. Дочерний компонент может отправлять сообщения обратно родителю с использованием событий $emet.

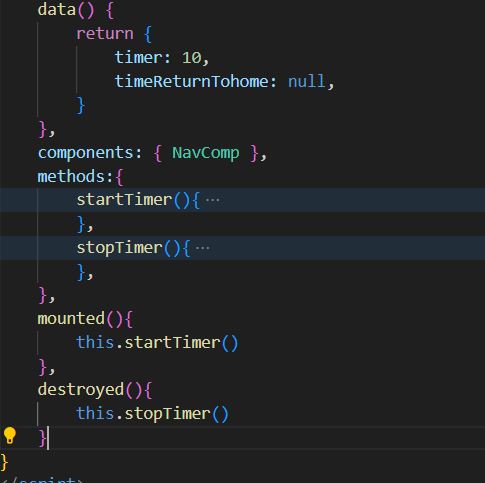
Каждый компонент может иметь свою собственную логику и стили, что позволяет создавать переиспользуемые блоки для различных частей приложения.

4.2. Жизненный цикл компонента.

* Хуки создания, это beforCreate и created. BeforeCreate - вызывается до создания экземпляра Vue. В этот момент ещё нет доступа к данным и методам компонента. Доступ к реактивным данным и событиям можно получить при помощи хука created. Шаблоны и виртуальный DOM еще не смонтированы и не проанализированы.
* Хуки монтирования. Хук beforeMount запускается сразу перед начальным рендерингом, здесь можно привязываться к глобальным объектам, например, к window. Хук mounted дает полный доступ к компоненту, шаблонам и визуальной модели DOM (через this.$el). Mounted используется для внесения изменений в DOM, особенно при интеграции библиотек, не относящихся к Vue:
* Хуки обновления. Хуки обновления вызываются при каждом изменении реактивного свойства, которое использует ваш компонент, или в том случае, если что-то еще повторно его визуализирует. BeforUpdate вызывается до изменений в DOM дереве страницы, здесь можно обращаться к реактивным переменным. Updated запускается после того, как внесены изменения.
* Хуки разрушения. BeforeDestroy - вызывается перед уничтожением компонента, здесь можно освобождать ресурсы, отвязываться от каких-то обработчиков. Здесь можно выполнять очистку ресурсов и отменять подписки, т.к. это не происходит в автоматическом режиме. Destroyed - вызывается после того, как компонент полностью уничтожен. Нет доступа к данным компонента в этот момент.
* Активация и деактивация. Используется, когда содержимое в компоненте будет определяться динамическим способом. Применяется, когда нам надо сохранять состояние компонента при его переключении.

Хук created в моем проекте реализован на странице с контактами и на странице с отзывами. И там и там мне надо чтобы приложение начало обращаться к API как можно раньше. Это момент, когда компонент уже готов к взаимодействию, но еще не виден пользователю. Примеры решения в коде будут приведены ниже в разделе подключения Google Maps и в разделе 6.1. использование Firebase для хранения данных.

Хук mounted и destroyed реализованы на странице 404, где приложение делает автоматический возврат пользователя на главную страницу, по истечению некоторого времени. Таймер возврата запускается после того как, как будет сформировано DOM дерево и пользователь сможет увидеть эту страницу. Хук destroyed здесь необходим, чтобы остановить выполнение setInterval, который был использован в запуске таймера.



## 4.3. Интеграция Google Maps для отображения расположения зубных клиник.

Что такое API.

**API, или интерфейсы программирования приложений, представляют собой набор средств и правил, которые позволяют программам взаимодействовать между собой. Они играют ключевую роль в современном программном обеспечении и веб-разработке по следующим причинам:**

* **Интеграция:**

**API позволяют различным программам и сервисам обмениваться данными и взаимодействовать друг с другом. Это особенно важно в мире, где существует множество разнообразных приложений и сервисов.**

* **Разделение Обязанностей:**

**Использование API позволяет разработчикам разделять функциональность программы на отдельные компоненты. Это способствует более четкому и легкому сопровождению кода, а также облегчает масштабирование приложения.**

* **Доступ к Сервисам и Данным:**

**API предоставляют стандартизированный способ получения доступа к сервисам и данным, предоставляемым сторонними провайдерами. Например, социальные сети предоставляют API для взаимодействия с их платформой, а картографические сервисы - для доступа к картам и геоданным.**

* **Разработка Клиент-Серверных Приложений:**

**API являются неотъемлемой частью клиент-серверной архитектуры. Они позволяют фронтенд-части (клиенту) и бэкенд-части (серверу) взаимодействовать, передавая данные и команды между ними.**

* **Микросервисная Архитектура:**

**В микросервисной архитектуре приложение строится как набор мелких сервисов, каждый из которых выполняет свою конкретную функцию. API играют ключевую роль в взаимодействии между этими сервисами.**

* **Разработка Мобильных Приложений:**

**При разработке мобильных приложений API позволяют приложениям взаимодействовать с серверами, получать данные, отправлять запросы, а также обмениваться информацией.**

* **Развитие Экосистемы Приложений:**

**Открытые API способствуют созданию экосистем приложений, когда сторонние разработчики могут создавать приложения, используя возможности и данные вашего сервиса.**

* **Инновации и Расширения:**

**API позволяют разработчикам интегрировать новые технологии и функциональность в свои приложения без необходимости изменения внутренней логики или архитектуры приложения.**

**Итак, API являются ключевым инструментом для создания гибких, расширяемых и взаимосвязанных приложений, а также обеспечивают возможность создания разнообразных и инновационных продуктов.**

Как работать с API.

Асинхронные методы в программировании используются для обработки задач, которые могут занимать время, без блокировки основного потока выполнения программы. Это позволяет эффективно управлять ресурсами и обеспечивать отзывчивость приложений в ситуациях, когда выполнение задачи может занять некоторое время, например:

* Сетевые запросы:

Отправка запросов к серверу для получения данных. Асинхронные методы позволяют приложению продолжать выполнение других задач во время ожидания ответа от сервера.

* Файловые операции:

Загрузка, сохранение или обработка файлов может быть времезатратной операцией. Асинхронность позволяет избежать блокировки интерфейса пользователя во время выполнения этих операций.

* Обработка событий:

Асинхронные методы широко используются в обработке событий в веб-разработке. Например, обработка кликов, отправка форм, и другие события выполняются асинхронно.

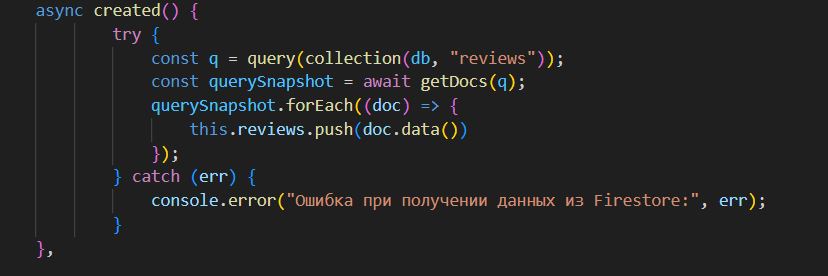
* Таймеры и интервалы:

Использование таймеров и интервалов для выполнения задач в определенные моменты времени. Асинхронные методы обеспечивают выполнение других операций во время ожидания таймеров.

* Параллельное выполнение задач:

Асинхронность позволяет параллельно выполнять несколько задач, улучшая производительность и сокращая время ожидания.

Есть разные способы работы с асинхронным кодом, например промисы и Async/Await. Пример использования асинхронных функций, где ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится приведен ниже.



Google Maps

В своей работе я буду подключать Google Maps. Google Maps — это сервис от Google, предоставляющий картографические данные, маршруты и другую связанную с геолакицией информацию. Он широко используется в веб-разработке для встраивания интерактивных карт в веб-приложения и сайты.

Возможности Google Maps:

* Отображение карт:

Встраивание интерактивных карт на веб-страницы, с возможностью управления масштабированием, перемещением и отображением различных слоев (например, улицы, места).

* Маршруты и навигация:

Построение маршрутов между двумя или более точками, расчет расстояний, времени в пути и отображение навигации.

* Места и геокодирование:

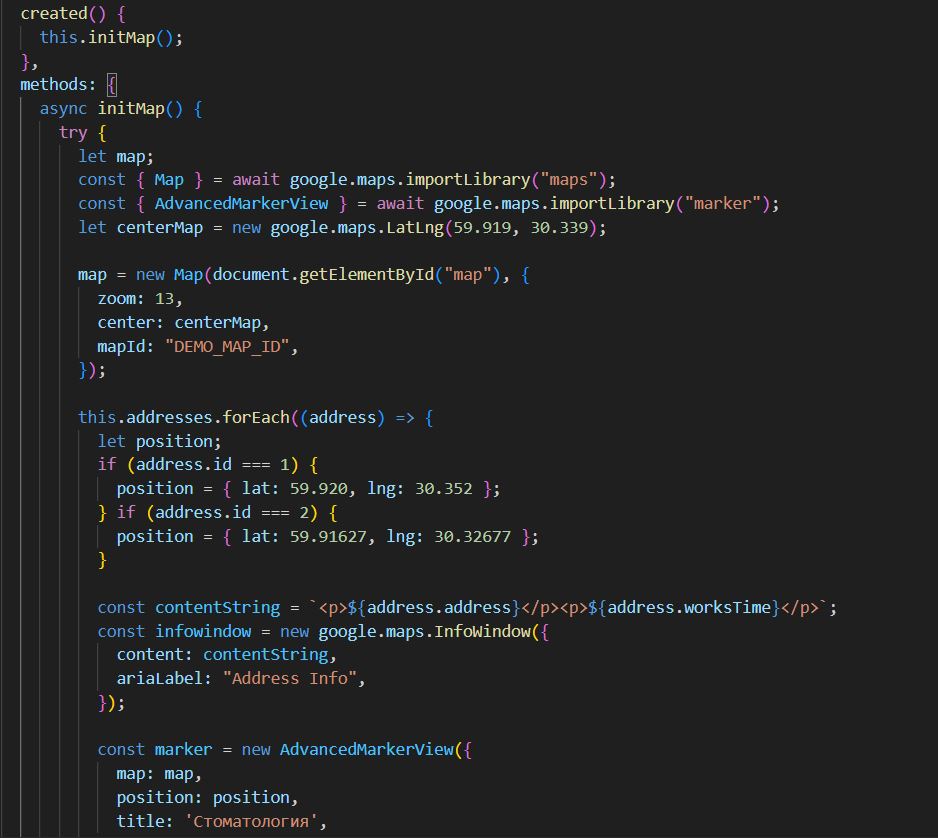
Получение информации о местах, поиск мест в определенном районе, а также преобразование адресов в координаты и наоборот.

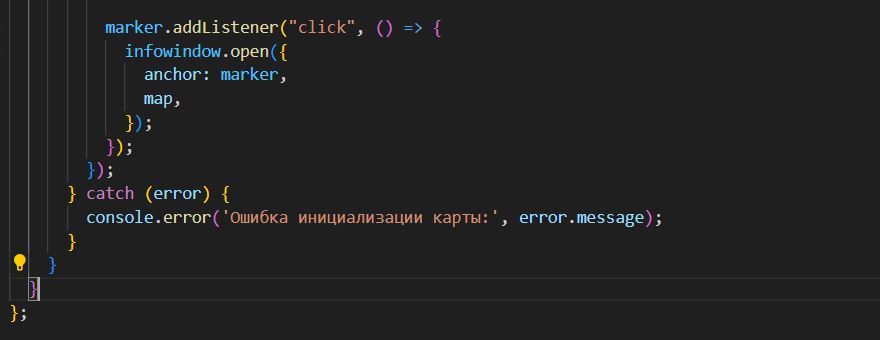
* Дополнительные компоненты:

Использование дополнительных служб, таких как Places API для поиска мест, Street View для просмотра уличных изображений, Elevation API для получения высоты над уровнем моря и других.

Так же Google Maps дает возможность изменять внешний вид карты, устанавливать маркеры, что дает возможность сделать приложение удобным для конечного пользователя.

Реализация компонента с картами представлена ниже





Хук created вызывается после того, как экземпляр компонента был создан, и все его свойства данных (определенные в data) и методы (определенные в methods) доступны. Однако, в этот момент экземпляр Vue ещё не cмонтирован в DOM. Сreated используют, чтобы установить начальные значения для данных, осуществить первоначальную загрузку данных из API или других источников, что и сделано в моем коде. Так как я использую сторонний сервис необходимо воспользоваться асинхронным методом.

# **5. Авторизация.**

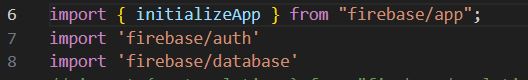
## 5.1. Авторизация.

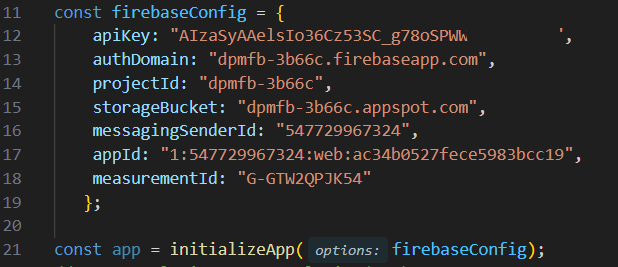
Firebase - это облачная платформа, разработанная Google, которая предоставляет разработчикам инструменты для создания приложений. Она позволяет создавать приложения для мобильных устройств и других платформ, предоставляя готовые инструменты для разработки и развертывания приложений без необходимости управления инфраструктурой.

Firebase включает в себя:

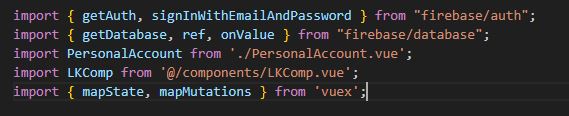
* Firebase Realtime Database: Облачная база данных реального времени.
* Firebase Authentication: Механизм аутентификации и управления пользователями.
* Firebase Cloud Messaging: Сервис для отправки уведомлений на устройства пользователей.
* Firebase Hosting: Облачный хостинг для статического и динамического контента.
* Другие компоненты (Firebase Analytics, Firebase Functions и т.д.).

Для подключения Firebase необходимо установить в проект сам Firebase командой “npm i firebase” и сделать соответствующие импорты в main.js:





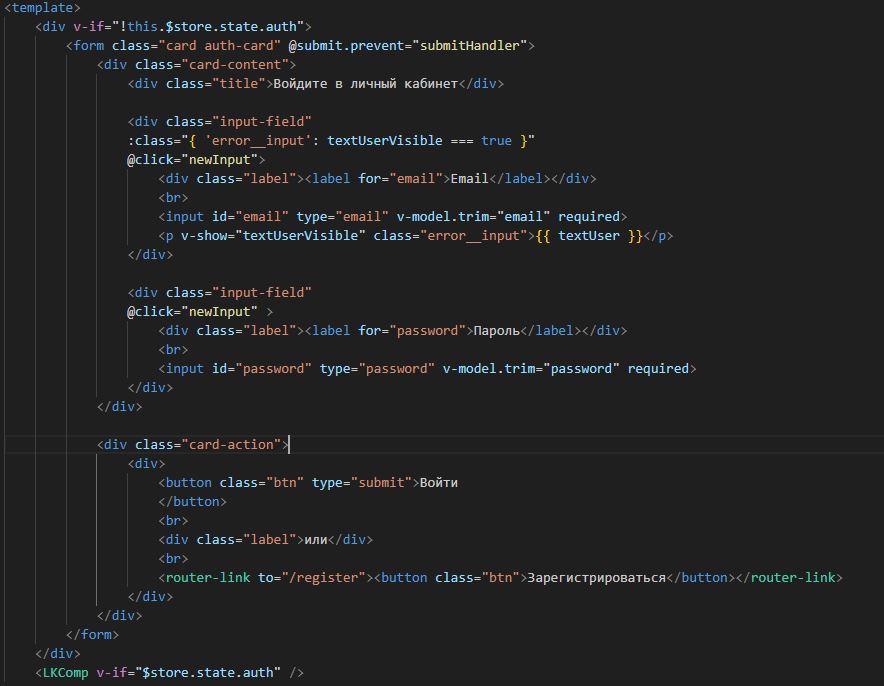
В необходимом компоненте так же нужно импортировать соответствующие модули firebase:



На данном примере это компонент аутетинфикации. Авторизация пользователя будет проходить по почте и паролю. Далее необходимо сделать соответствующий обработчик событий. По нажатию на кнопку «Войти» будет обрабатываться событие submitHandler:



В самом вызове метода надо отключить превентивную перезагрузку станицы: @submit.prevent="submitHandler". Так как мы обращаемся к постороннему сервису метод сделаем асинхронным и пропишем два сценария. Первый успешный и мы перенаправляем пользователя на страницу личного кабинета, и второй, где мы выведем сообщение для пользователя, если что-то пойдет не так. Для этого вводим переменную textUser, где пишем сообщение для пользователя, и textUserVisible, это флаг, который изначально мы устанавливаем в значение false. При возникновении ошибки флаг переводим в положение true и для пользователя появляется сообщение, для этого биндим :class=«error\_\_input», а для того чтобы очистить форму ввода при клике на поле сделаем метод newInput. Шаблон html будет выглядеть так:



На странице регистрации технология будет похожая, но там мы будем импортировать соответствующие модули farebase:

Страница регистрации:

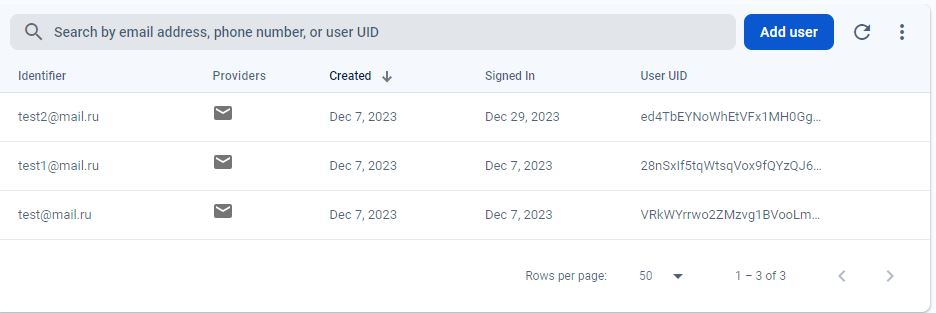
import { getAuth, createUserWithEmailAndPassword } from "firebase/auth";



Здесь так же не забываем обработать ошибки, при попытке зарегистрироваться пользователем, который уже существует в базе и при вводе пароля меньше 6 символов, будут выводиться соответствующие сообщения пользователю.



После регистрации нового пользователя на платформе Firebase этот пользователь появляется в базе данных



Обработка выхода из личного кабинета:

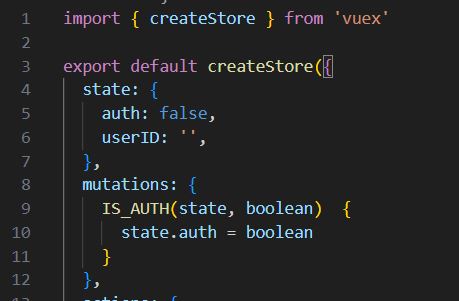
Как обычно, импортируем нужные модули:

import { getAuth, signOut } from "firebase/auth";



При входе и выходе из личного кабинета будем менять состояние приложения.

Изначально состояние нашего приложения выглядит так:



Пользователь не авторизован и userID представляет из себя пустую строку. Изменение состояния можно сделать напрямую в методе, например:

this.$store.state.auth = true;.

Но, как уже сказано выше, лучше это делать через mutation, на скриншоте выше видно, как это сделано в моем приложении.

Вызвать mutation в компоненте можно несколькими способами.

* commit:

Простой и явный способ вызвать мутацию.

Подходит для случаев, когда требуется явное указание хранилища и мутации.

Позволяет более гибко управлять параметрами и логикой вызова мутаций.

В моем приложении это выглядело бы так:

this.$store.commit('IS\_AUTH', true);

* ...mapMutations:

Удобный способ создать методы компонента, которые автоматически вызывают мутации.

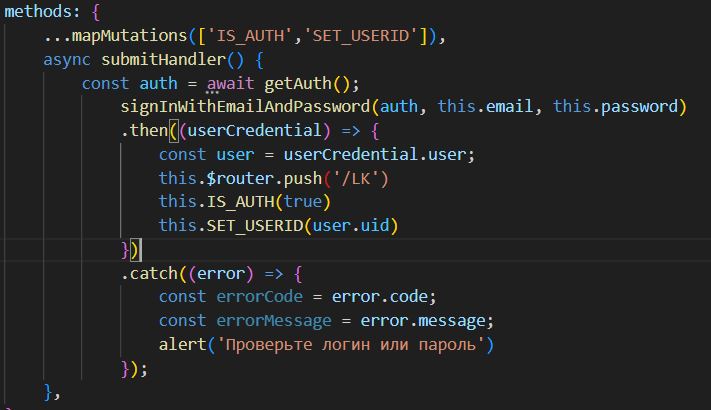
Полезен, если нужно вызывать несколько мутаций в компоненте.

Может сделать код компактнее и более читаемым, особенно в компонентах с большим количеством мутаций.

Пример использования ...mapMutations:

Необходимо сделать соответствующий импорт:

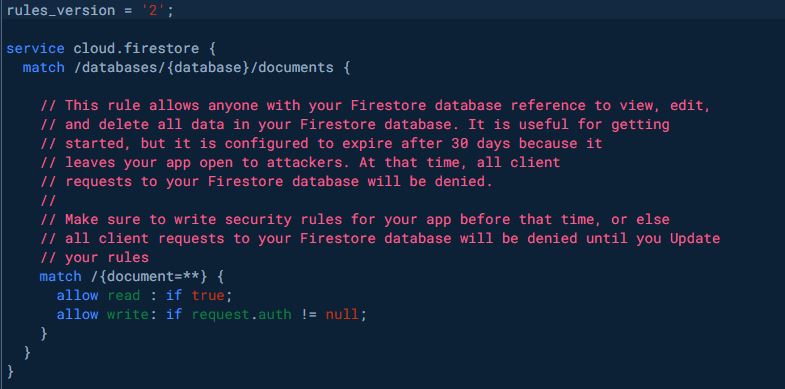
import { mapMutations } from 'vuex';



## 5.2. Обеспечение безопасности данных.

Firebase предоставляет механизм управления доступом к данным с помощью правил безопасности. Эти правила могут быть настроены для различных уровней, начиная от операций записи во всю базу данных до более детального управления операциями над конкретными документами.

Так в моем приложении настроено правило, что отзывы, хранящиеся в базе данных может прочитать каждый, но оставить только авторизованный пользователь.

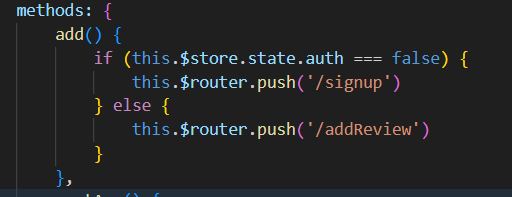


# **6. База данных и хранение данных.**

## 6.1. Использование Firebase для хранения данных.

Так же в Firebase буду хранить отзывы о клинике и врачах.

Кнопка «оставить отзыв» будет размещаться на станице с отзывами и в личном кабинете, но в любом случае только пройдя авторизацию эта функция станет доступна. Для этого надо будет добавить проверку на состояние приложения, если state.auth === false, пользователь будет направлен на страницу входа в личный кабинет, иначе на страницу, где можно заполнить форму ввода.



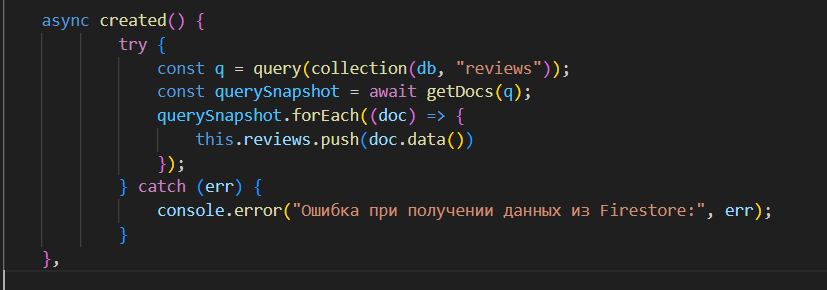
На самой же странице с отзывами будут подгружаться сами отзывы из Firebase.

Не забываем сделать необходимые импорты:

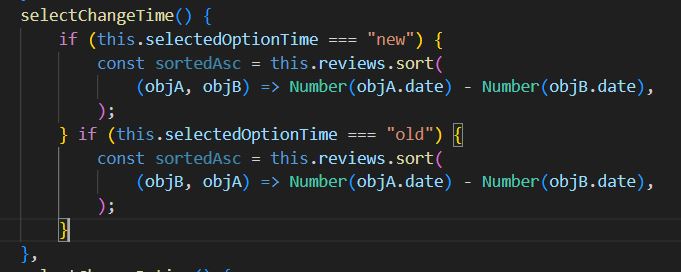
import { db } from '@/main.js';

import { collection, query, where, getDocs } from "firebase/firestore";

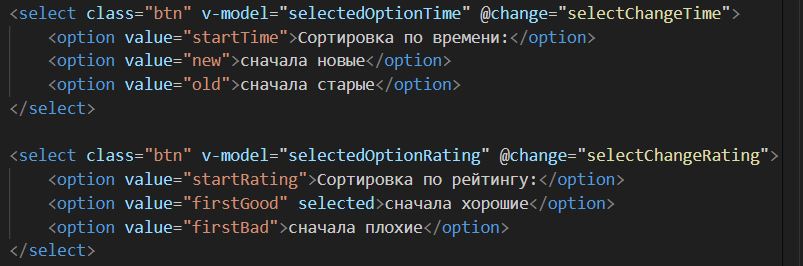
Опять же обращаемся к стороннему сервису, поэтому используем async/await, и здесь так же, как и Google maps, загрузка будет происходить в created.



На данной странице реализована сортировка отзывов по времени, от более новых к старым и наоборот для этого в момент создания отзыва приложение передает текущее время. Так же реализована сортировка по рейтингу. На скриншоте ниже приведен пример сортировки по времени, по рейтингу выглядит аналогично.



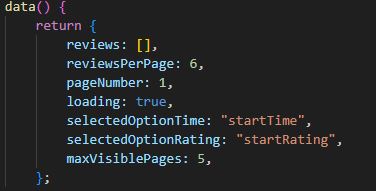
Для обработки событий по сортировке использую тэг select. Так как select не поддерживает событие «click», буду использовать событие «change». Пример кода приведен ниже:



Обращение к базе данных может занять время, чтобы пользователь понимал, что приложение работает и находится в процессе загрузки создам компонент preloadComp, который будет визуально информировать о загрузке. Для этого импортирую данный компонент в шаблон c директивой v-if и создаю переменную loading: true, пока подгружается массив с отзывами значение этой переменной остается true, по окончанию загрузки флаг переключается в значение false. Сам preloadComp сделан силами HTML и CSS.

Для того чтобы большое количество отзывов не превратило наше приложение в бесконечную ленту, в данном компоненте реализована пагинация.

В data определяем необходимые переменные



reviewsPerPage это количество отзывов на одной странице, и задаем номер страницы отображаемой по умолчанию (pageNumber).

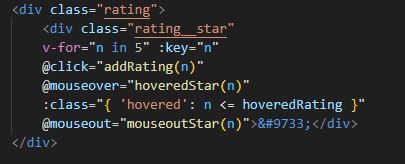
В computed свойствах рассчитываем количество страниц, в зависимости от длины массива с отзывами, полученного из базы данных. И пишем метод, который будет показывать нам нужное количество отзывов.



Шаблон компонента для пагинации будет такой:



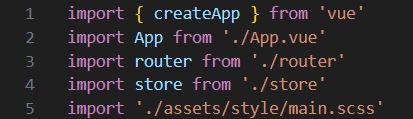
В компоненте, где можно оставить отзыв, представлены стандартное поле ввода для имени и textarea для самого текста отзыва. Для присвоения рейтинга (количества звездочек от 1 до 5) использую три события click, mouseover и mouseout. По клику будет происходить присвоение рейтинга, а mouseover и mouseout нужны для улучшения пользовательского опыта. При наведении на звездочку, предыдущее тоже подсвечиваются, для этого я использую переменную hoveredRating, которая изначально не определена, а при наведении на звездочку с помощью mouseover получает значение, далее с помощью директивы v-bind:class динамически присваиваю класс, который при выполнении условия задает элементам класс. Шаблон HTML присвоения рейтинга представлен ниже.



# **Методы создания адаптивного сайта.**

* 1. Подключение стилей, использование препроцессоров.

В своей работе я буду использовать препроцессоры, с их помощью можно структурировать стили, путем создания отдельных файлов с переменными и миксинами и импорта их в главный файл стиля main.scss, который в свою очередь импортируется в main.js:



Использование миксинов позволяет переиспользовать код, что значительно сокращает его количество, а так же, дает возможность комплексно изменять стили при необходимости.

Так как у каждого компонента своя область видимости, чтобы использовать переменные и миксины в отдельных компонентах необходимо сделать импорт файла с переменными и миксинами в сам компонент.



## 7.2. Подключение шрифтов.

Подключение шрифтов может быть выполнено несколькими способами.

* Google Fonts предоставляет простой способ подключить шрифты, хранящиеся на серверах Google, использование этих шрифтов бесплатно.
* Локальные шрифты. Если у вас есть локальный файл шрифта (например .woff, .woff2, .ttf, .otf), вы можете подключить его с помощью @font-face в вашем CSS файле.

@font-face – позволяет использовать свои собственные шрифты, не зависящие от наличия шрифтов на устройстве пользователя. Это полезно, когда нужно использовать специфические шрифты, которые не доступны по умолчанию. Рекомендуется предоставлять шрифты в нескольких форматах для лучшей поддержки различными браузерами. Форматы woff и woff2 часто используются для веба, так как обеспечивают хорошее сжатие и поддержку большинства современных браузеров.

## 7.3. Адаптивный сайт.

Адаптивность веб-сайта — это способность сайта корректно отображаться на различных устройствах. Адаптивный дизайн важен для обеспечения удобства использования и улучшения пользовательского опыта.

Средства для адаптивности:

* Медиазапросы (Media Queries). Гибкий и тонкий инструмент, позволяющий изменять стили в зависимость от разрешения экрана.
* Flexbox и Grid Layout. Flexbox используют когда нужно распределить элементы в одном направлении (строка или столбец), а Grid, когда нужна более сложная двумерная структура сетки. Так, например, grid-template-columns обладает широкими свойствами, благодаря которым элементы могут самостоятельно перестраиваться.
* Относительные единицы измерения (проценты, fr, vh, vw, rem). Такие единицы измерения, как vh, vw является актуальным, если нам необходимо применять стили к объекту, относительно высоты и ширины окна, в этих случаях, как раз и используют vh, vw, 100vh или 100vw –это 100% высоты или ширины экрана. Для адаптивности текста можно использовать rem. Rem представляет собой относительное значение размера шрифта, основанное на размере шрифта корневого элемента (<html>). Допустим, мы устанавливаем, что 1 rem = 16 px, тогда в верстке можно задавать размеры не в пикселях, а в rem, например, font-size: 2rem;. Преимущества использования rem. Масштабируемость, если изменить размер шрифта для <html>, все значения rem автоматически изменяются. Удобство в адаптивном дизайне, достаточно один раз указать в медиа запросе необходимые значения, и они автоматически применятся ко всему проекту.

## 7.4. Требования к дизайну.

Требования к дизайну для полноэкранных и мобильных версий могут отличаться.

* Размеры и расположение элементов. В некоторых случаях необходимо изменять размеры и расположение элементов, чтобы они были оптимальными для маленьких экранов.
* Навигация. Для мобильных устройств удобно использовать бургер-меню или иные методы сокрытия навигационных элементов.
* Типография. Не стоит делать мелкий шрифт, чтобы пытаться уместить весь контент. В некоторых случаях для мобильных версий нужно уменьшить количество текста.
* Изображения и мультимедиа. Необходимо оптимизировать изображения для экономии трафика и ускорения загрузки на мобильных устройствах.
* Скрытие или изменение некоторых элементов.

В моем проекте реализована смена контента в навигации в зависимости от размера экрана. На больших экранах в кнопках навигации отображаются текстовые элементы и изображение, но на экранах с разрешением 425 px и меньше текст пропадает и остается только изображение. Для этого в хуке created необходимо организовать прослушивание размера экрана. А в хуке beforDestroy или в destroy, необходимо очистить наши ресурсы, т.к. мы работаем с глобальным объектом window и на прямую к данному компоненту он не относится.



В шаблоне с помощью директивы v-if я прописываю два варианта



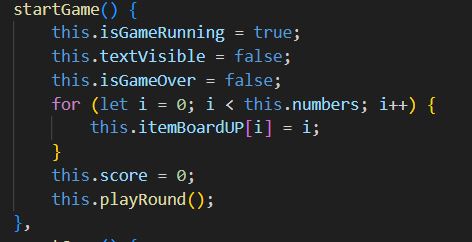
В стиле данного компонента я так же меняю расположение навигационного меню. В мобильной версии он будет располагаться внизу экрана.

# **8. Улучшение пользовательского опыта, создание небольшой игры.**

Компонент "Терапевтическая Игра" разработан для предоставления пользователям не только развлечения, но и терапевтического воздействия. Игра может помочь пользователям расслабиться во время ожидания приема, а детям может указать на важность профилактических мер таких как, ежедневная чистка зубов и периодические осмотры врачем-стоматологом.

Задача игры – вовремя вылечить зубы. В игре есть таймер, который отсчитывает время для каждого хода. Если игрок не успевает «вылечить зубы» за отведенное время, игра завершается.

В логике используется setTimeout, setInterval и Math.random(). В начале игры необходимые флаги устанавливаются в начальное положение, генерируется игровое поле, в котором случайным образом будет появляться «кариес».



Основная логика представлена ниже:



Создаем переменную playerClicked, которая будет указывать нажал ли игрок на кнопку мыши.

Далее создаем стрелочную функцию, которая будет отслеживать клики игрока по элементам:

Флаг playerClicked переводим в положение true.

СlearInterval(this.timeGameOver) очищает интервал timeGameOver( время за которое игрок должен успеть выбрать элемент). Это останавливает выполнение кода, связанного с завершением игры, если игрок успел кликнуть по элементу до завершения интервала.

Document.removeEventListener('itemClicked', itemClickedHandler) удаляет слушателя события 'itemClicked'( клик по элементу).

Генерируется случайный индекс из массива itemBoardUP, затем устанавливается активный индекс, чтобы подсветить соответствующий элемент на игровом поле, через 1 секунду с помощью setTimeout убираем эту подсветку.

this.timeGameOver = setInterval(() => ... , 1500) запускает таймер, который уменьшает значение переменной timer. Если таймер достигает 0, игра завершается. Установка этого интервала необходима для автоматического завершения игры, если пользователь отвлекся или не попал по нужному элементу.

document.addEventListener('itemClicked', itemClickedHandler);: Устанавливается слушатель события 'itemClicked'. Если пользователь кликнет на правильный элемент, событие 'itemClicked' будет вызвано, и обработчик события itemClickedHandler выполнит соответствующие действия.

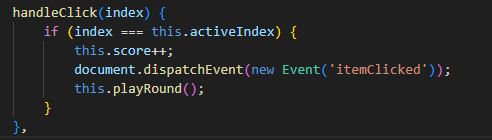
Если пользователь кликнул (переменная playerClicked стала true), игровой цикл завершается, и интервал timeGameOver останавливается.

Метод handleClick обрабатывает клик пользователя на элемент игрового поля.

Осуществляется проверка, был ли клик на правильном элементе, сравнивая индекс кликнутого элемента (index) с активным индексом (this.activeIndex). Если клик был на правильном элементе счет в игре увеличивается.

dispatchEvent - это метод JavaScript, используемый для инициирования событий на DOM-элементах. В моем случае он используется для создания и отправки пользовательского события 'itemClicked'.

Когда document.dispatchEvent(new Event('itemClicked')) вызывается, создается и инициируется новое событие 'itemClicked', которое затем может быть обработано другими частями кода, слушающими это событие. В данном случае событие используется для прерывания текущего раунда игры и перехода к следующему.



# **9. Развертывание приложения.**

## 9.1. GitLab.

GitLab - это веб-платформа для удаленного хранения проектов и управления ими, предоставляющая инструменты для совместной разработки программного обеспечения.

* Управление репозиториями.
* Интегрированный CI/CD. GitLab обладает встроенной системой Continuous Integration (CI) и Continuous Deployment (CD). Это позволяет автоматизировать процессы сборки, тестирования и развертывания приложений.
* Защита веток от случайных изменений.
* Управление доступом и ролями. GitLab позволяет назначать роли и уровни доступа для участников проекта, обеспечивая контроль над информацией.
* Интеграция с инструментами сторонних разработчиков (JIRA, Slack, Jenkins).

## 9.2. CI/CD.

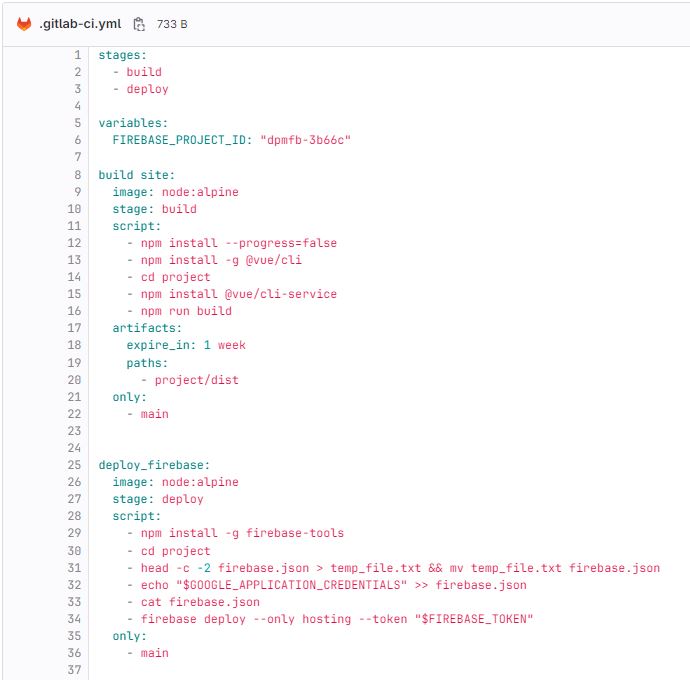
* Continuous Integration (Непрерывная Интеграция) - это методология, которая предполагает частое и автоматизированное объединение кода разработчиков в общий репозиторий, автоматическое выполнение тестов после слияния. Это позволяет регулярно проверять работоспособность кода и предотвращать накопление проблем.
* Continuous Deployment (непрерывное развертывание - это продолжение CI, предполагающее автоматизированное развертывание кода после успешного прохождения всех этапов.

Необходимые шаги:

Создание, клонирование или зазеркаливание репозитория.

Настройка пайплайна.

Пайплайн это набор автоматизированных этапов и процессов. CI/CD-пайплайн включает в себя необходимые этапы, переменные, артефакты и т.д. Для настройки пайплайнов в GitLab используются специальные файлы конфигурации, которые должны называться .gitlab-ci.yml, в данном файле необходимо прописать все шаги развёртывания, используемые переменные и прочие настройки.



На скриншоте выше представлен файл .gitlab-ci.yml, используемый в моём проекте.

В моем CI/CD пайплайне выделяются два ключевых этапа: build и deploy. Начинаем с определения переменных, где я указываю идентификатор проекта на платформе Firebase. Далее следует описание этапа build.

Для сборки приложения я использую образ node:alpine. Этот образ содержит Node.js, что дает возможность выполнять JavaScript-код на сервере, благодаря небольшому размеру образа достигается быстрое развертывание. В рамках этого этапа прописаны команды сборки приложения. Так же у меня указано, что сборка будет проходить только в главной ветке.

После успешной сборки необходимо сохранить результаты. Для этого применяется механизм артефактов, где указывается путь к результатам сборки. Таким образом, успешно завершенные задачи сохраняют свои артефакты, которые становятся доступными для использования в следующих этапах, обеспечивая последовательное выполнение CI/CD процесса.

На этапе deploy происходит развертывание приложения на платформе firebase. Здесь я использую тот же образ, что и в предыдущем этапе. Далее прописываю необходимые команды. На этом этапе было необходимо завести две переменные это GOOGLE\_APPLICATION\_CREDENTIALS и FIREBASE\_TOKEN, эти данные необходимо получить на самой платформе firebase. Развертывание также происходит в главной ветке.

Для сборки проекта удалённо с помощью GitLab CI необходим «Runner» - сервер на котором будет проводиться сборка и тестирование приложения. В моём случае используется shared-runner предоставленный платформой GitLab, при необходимости можно сконфигурировать использование своего специфического раннера. Использование того или иного раннера влечёт за собой как преимущества так и недостатки, вот некоторые из них:

1. **Локальная настройка и конфигурация:**
   * **Свой раннер:** Вы имеете полный контроль над конфигурацией своего раннера. Вы можете настроить его согласно специфическим требованиям вашего проекта и инфраструктуры.
   * **Shared-runner GitLab:** В общем раннере вы ограничены предустановленными конфигурациями, предоставляемыми GitLab. Это может означать, что некоторые требования вашего проекта не будут удовлетворены.
2. **Высокая производительность:**
   * **Свой раннер:** Поскольку свой раннер работает на вашей собственной инфраструктуре, вы можете оптимизировать его для максимальной производительности в соответствии с вашими потребностями.
   * **Shared-runner GitLab:** Общие раннеры могут быть загружены в периоды высокой загрузки, что может повлиять на производительность вашего пайплайна.
3. **Контроль над безопасностью:**
   * **Свой раннер:** Вы имеете полный контроль над безопасностью своего раннера и можете принимать меры для обеспечения безопасности в соответствии с вашими стандартами и требованиями безопасности.
   * **Shared-runner GitLab:** Общие раннеры могут предоставлять доступ к общим ресурсам, что может создать потенциальные угрозы безопасности.
4. **Выделенные ресурсы:**
   * **Свой раннер:** Свой раннер обеспечивает выделенные ресурсы для ваших задач, что гарантирует стабильность и предсказуемость процесса сборки и развертывания.
   * **Shared-runner GitLab:** Общие раннеры разделяются между несколькими проектами, что может привести к изменениям в производительности и ресурсах в зависимости от активности других проектов.
5. **Интеграция с локальными ресурсами:**
   * **Свой раннер:** Вы можете легко интегрировать свой раннер с локальными ресурсами и сервисами вашей инфраструктуры.
   * **Shared-runner GitLab:** Использование общих раннеров может потребовать дополнительных настроек и интеграции для работы с локальными ресурсами.

Выбор между своим раннером и shared-runner GitLab зависит от конкретных требований проекта, уровня необходимого контроля, и инфраструктуры, которую готовы предоставить для выполнения CI/CD-пайплайнов.

На практике, после каждого коммита, процесс CI запускает сборку приложения и разворачивает приложение в целевой среде, будь то тестовый сервер или продакшн среда.

Так же можно добавить такие этапы, как тестирование, проверка стиля написания кода, анализ безопасности и другие. А так же настроить разное поведение в зависимости от названия ветки или комментария к коммиту.

# **10. Перспективы развития.**

В дальнейшем в проекте необходимо реализовать:

* При регистрации запрашивается имя пользователя, но пока не реализовано его хранение в базе данных. В дальнейшем надо будет при оставлении отзыва автоматически подгружать имя в поле ввода. Так же надо ограничить по времени возможность оставлять отзыв, например, не чаще одного раза в месяц или после посещения.
* Реализовать возможность записи в клинику, выбора даты, времени и врача. При отсутствии собственного бэкэнда, такую возможность реализуют сторонние сервисы. Существуют как платные, так и бесплатные сервисы.
* Так же в реальном проекте необходимо будет настроить подтверждение почты при регистрации, Farebase такую возможность предоставляет.
* Организовать в личном кабинета возможность просмотра рекомендаций врача, стоимости проведенного лечения и т.п.

Сделать!:

4. Возможность онлайн записи

4.1 подключение к crm

4.2 автоматическая отправка письма или сообщения о записи

4.3 Автоматическое напоминание о записи за сутки

Изображения взяты из источника:

https://www.freepik.com/