[Требования задачи](https://github.com/UptraderTestTask/Junior-frontend#%D1%82%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F-%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%B8)

* Проект должен содержать 2 страницы
* Страница с выбором проекта
* Страница с задачами
* Все остальные действия должны происходить в модальных окнах
* Страница с задачами должна содержать в себе три колонки c возможностью изменения статуса с помощью drag-n-drop:
* Queue
* Development
* Done
* Каждая задача имеет:
* Номер задачи
* Заголовок
* Описание
* Дата создания
* Время в работе
* Дата окончания
* Приоритет
* Вложенные файлы
* Текущий статус
* Возможность добавлять подзадачи;
* Система каскадных комментариев(т.е. Можно оставить комментарий под комментарием... Пример можно увидеть на reddit или picabu)
* Возможность редактирования задачи
* Даже если все подзадачи выполнены, это не значит, что и основная становится выполненной
* Поиск по номеру задачи и заголовку
* Возможность создавать задачу
* Адаптация под мобильные устройства
* Обилие плавных, красивых анимаций\отзывчивость сайта на действия юзера \ Дополнительный функционал только приветствуются
* Приложение должно сохранять все данные после перезагрузки страницы есть 2 варианта
* Можете написать свой backend или использовать облачные решения, в таком случае допускается использование только Redux-saga
* Или можете использовать cookie и localStorage для сохранения всех данных

Russo One,sans-serif прикольный шрифт

План :

1. Создаем 3 страницы главная, доски, и NotFoundPage
2. В главной странице делаем :

* Окно приложения типа контейнер.
* Кнопка создать, при клике на кнопку появляется модальное окно через портал и там будет имя доски и кнопка сохранить и закрыть модальное окно.
* Когда создали доску появляется окно с названием доски (подумать максимальное кол-во букв) и три точки которые будут вызывать дропдаун с кнопкой удалить.

1. Создаем страницу доски в ней нужно будет сделать:

* header с кнопкой которая **возвращает** **назад**, **заголовок** , и **input** поиска
* Создаем три колонки статуса QUEUE, DEVELOPMENT, DONE

1. В колонках создаем будет

* id:
* name :string
* deskId: что бы показать к какой доске по id отностиься колонка
* + QUEUE делаем кнопку добавление карточки задачи

1. В создании карточки Cards задачи делаем модальное окно и указываем

* id,
* columnId, который будет смотреть в id колонки
* Заголовок,
* Описание (подумать о кол-ве букв ),
* Приоритет с тремя кнопками,
* Подзадачи,
* Добавление файла Blob с отображением.
* Кнопка создать

1. В отображаемой карточке выведем инфу о Заголовке, приоритет, номер задачи и она находиться в колонке QUEUE.
2. При клике на карточку открывается модальное окно с редактированием в нем мы можем поменять:

* Заголовок
* Приоритет с тремя кнопками,
* Подзадачи,
* Комментарии каскад +,
* Файл

Видим:

* в колонке QUEUE дату создания
* в колонке DEVELOPMENT сколько времени в работе
* в колонке DONE когда завершилась задача

+ Можем удалить карточку

1. Организовать поиск по имени карточки
2. Организация бека состоит из:

* Desks:
* id :
* name : string

Начал проект с инициализации firebase, создал проект YkBoard и подключил описывать не буду есть в К/С 4.1 подключение. Единственное применю структуру папок как у Redux RTK

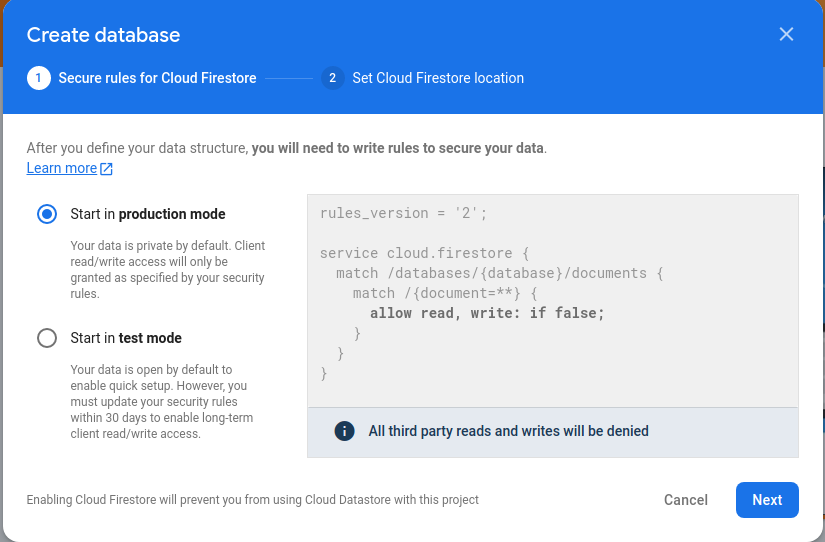
Дальше нужно будет настроить проект на firebase

Не забыть скопировать уникальные данные для подключения к проекту!

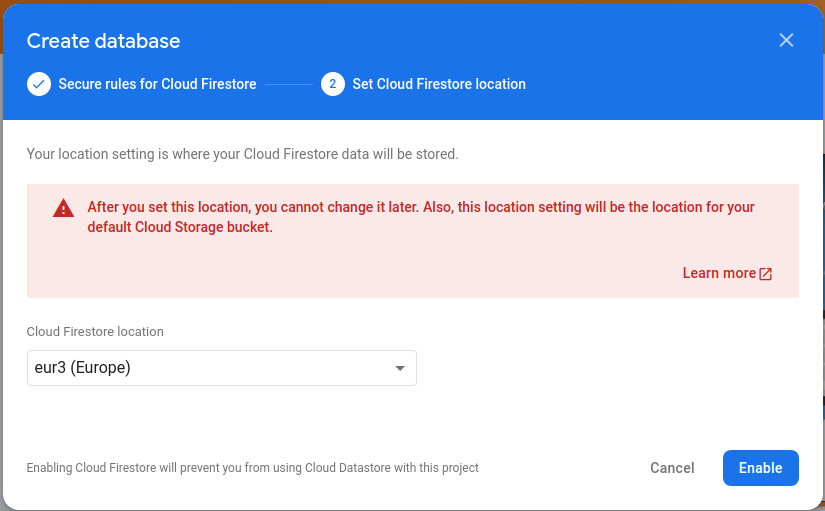
Authentication пока пропускаем

Переходим к **Firestore** **Database**, а не к Realtime Database (устарел!)

далее

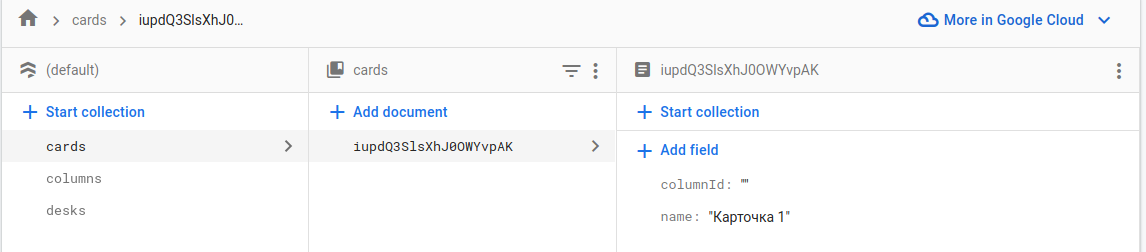


Выбираем продакшен мод поскольку отличие только в настройках правил в продакшен не читать не записывать не можем, а в тестовом можем записывать и читать но только 30 дней.



Выбираем европу

Далее создаем некое MVP структуры базы данных



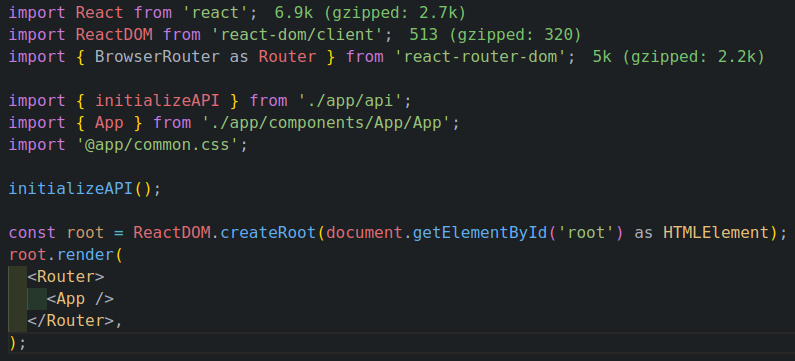
теперь нужно настроить в приложении и проверить

Важный момент инициализировать firestore нужно после создания firebaseApp!

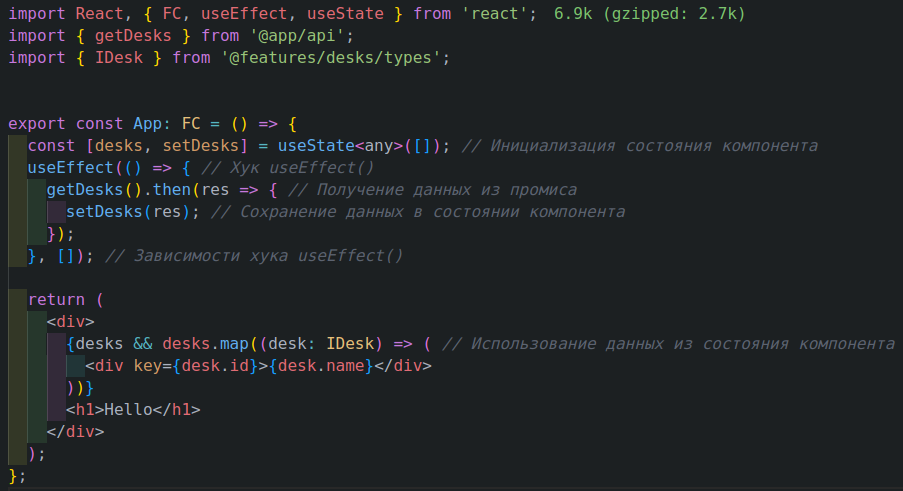
Создали папку api.ts и в нем будут все наши подключения к firebase



Для того что бы подключить его нужно в script.tsx добавить



Для проверки пока в App передадим функцию getDesks и интерфейс IDesk

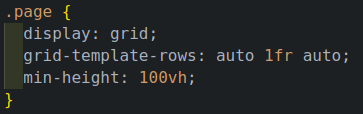
Работает вывели Данные о досках

Сделали подключение теперь будем верстать страницу Page. Cтраница это будет компонент который будет оборачивать дочерний компонент. Так вот у нее будет header main footer,

и когда делал header застрял на функционале изменения темы, он в себя включает:

* Блок аватар-логин
* Лого
* компонент смены темы

К стати когда делал Page то столкнулся с проблемой прижатия футера к низу, решил это через grid



Дальше делал Лого, сделал svg засунул в компонент там обернул в NavLink вывел на главную.

Перенес функционал из проекта Newsfeed по смене темы, и сейчас нужно дописать а именно:

появилась проблема при перезагрузке в темной теме происходит белая вспышка

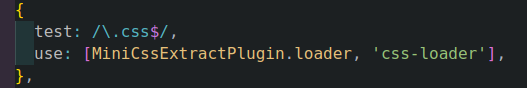
Кстати для смены темы через дев тулз используем **prefers color scheme: dark**

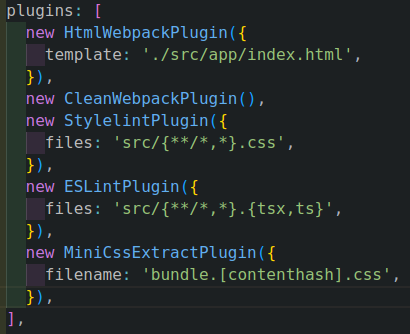
Вспышка появляется тогда когда грузиться основной бандл css по умолчанию он грузиться как defer и когда основной HTML загрузился, то он отображается со стилями по умолчанию, а там bg color белый и когда загружается основной бандл то меняется на черный и происходит мигание, для решения этой задачи можно поставить npm i mini-css-extract-plugin

по завершению установки импортнем его

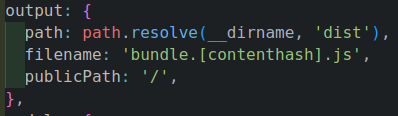
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

потом сносим 'style-loader' - это загрузчик (loader) для Webpack, который позволяет добавлять стили в виде тега <**style**> непосредственно в **HTML**-файл при использовании CSS-файлов. И заменяем на **MiniCssExtractPlugin**.loader - это загрузчик (loader) для Webpack, который извлекает CSS-код из JS-файлов и сохраняет его в отдельный CSS-файл. Это позволяет отделить стили от сценариев и уменьшить размер файлов, что положительно сказывается на производительности сайта.



потом не забыть добавить в   


и еще так как это будет отдельный ресурс нужно задать шаблон имени файла и он будет такой же как и output



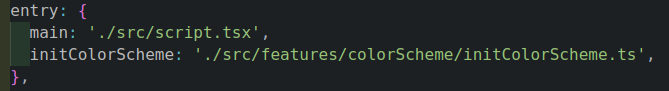
Теперь создастся bundle.css в теге link и дойдя до него bundle css загрузиться заблокировав поток и по завершению будет загружать HTML файл.

Однако проблема осталась логика осталась размазанной между Css и JS, в JS сетиться аттрибут на HTML, а в Css применяются стили, соответственно пока JS не приехал на HTML нет никакого аттрибута Scheme и браузер снова по умолчанию рисует светлую тему, в общем опять уперлись в то, что браузер загружает JS при этом продолжает строить HTML настолько быстро как только может.

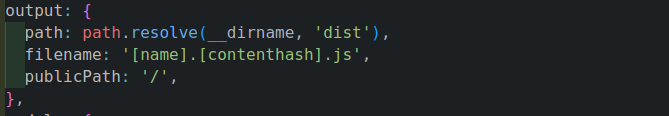
Выход:

нужно initColorScheme.ts который лежит в папке ./src/features/colorScheme/initColorScheme.ts

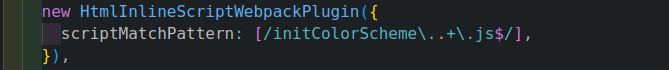
загрузить как можно раньше, для этого нужно его заинлайнить и его так вставить что бы он был между тегами script заключен без всяких запросов на сервер, для этого установим пакет **html-inline-script-webpack-plugin** он делает выходные файлы в инлайн причем делает это выборочно, добавим его в Webpack и ставим плагин **html-inline-script-webpack-plugin** теперь делаем initColorScheme как отдельный entry point



далее вводим переменную name в которой будет или main или initColorScheme



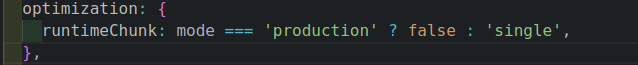
тут записываем какие скрипты инлайнить а какие пропускать

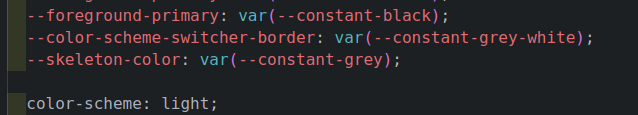


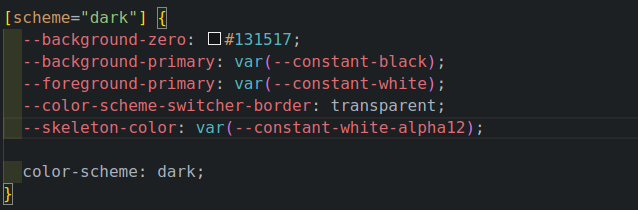
\* Обратите внимание, что в новой версии плагина опция должна быть объектом, а не массивом. Параметр **scriptMatchPattern** должен содержать регулярное выражение для сопоставления имен файлов сценария, которые вы хотите встроить в HTML.

С несколькими entry points возникает задвоение рантайм вебпака и ошибки вылазят по два раза, для того что бы этого избежать есть настройка в вебпаке, тем более ее можно сделать в режиме dev на проде нет смысла ее использовать





Еще с темой крайняя история это перекраска системных контролов для решения этого добавим во светлую тему color-scheme   


и в темную тему   


**Только вот не работает функционал dropdown,** разобрался в проблеме, оказывается почему-то хреново отрабатывал портал и когда я кликал на Dropdown менялось состояние dropdownShown на true и оно передавалось shown={dropdownShown} и волшебным образом попадал на **click** document.addEventListener, все решилось когда я за место документа сделал root и тогда разбились event на сработку overlay (открытие/закрытие окна при клике Dropdown) и root как поле закрытия



**Итог** создания Хедера первое в Page прописали хедер в нем создали контейнер в этом контейнере сделали Grid три колонки grid-template-columns: auto 1fr auto; что позволило прибить по бокам к краям ячейки и по середине растянуть

В левую поместили на будущее функционал для авторизации аватар и логин, в середину поместили Лейбл лого, а с права добавили иконку для смены темы, там есть 3 компонента 1 это **ColorSchemeSwitcher** – этот компонент отвечает за смену темы там есть вся логика по мене темы

* В начале инициализируются состояния **userScheme**, **dropdownShown** и **targetRef**.
* Затем вызывается **useEffect**, он изменяет применяемую цветовую схему в зависимости от выбора пользователя. Если user выбрал автоматический выбор цветовой схемы (userScheme === 'auto'), то используется системная цветовая схема, иначе применяется выбранная пользователем схема.
* Также **useEffect** используется для добавления и удаления слушателя событий на объект **matchMedia**. Этот слушатель отслеживает изменение предпочитаемой цветовой схемы в системе, и если у пользователя выбрана автоматическая цветовая схема (userScheme === 'auto'), то применяется системная цветовая схема.
* Возвращаемое значение компонента состоит из элемента **div** с классом **color-scheme-switcher**, содержащего элемент **button** с классом **color-scheme-switcher\_\_value** и кнопку, которая отображает текущую выбранную пользователем схему. При клике на эту кнопку выпадает список опций, которые пользователь может выбрать. Для отображения списка опций используется компонент **Dropdown**. Каждая опция это кнопка с классом color-scheme-switcher\_\_option, содержащей иконку и название цветовой схемы. Кнопка также содержит img элемент с классом color-scheme-switcher\_\_check, который отображает иконку выбранной цветовой схемы, если пользователь ее выбрал. При выборе опции пользователем, вызывается функция-обработчик, которая устанавливает новую цветовую схему.

**Dropdown –** это компонент, который представляет собой выпадающий список.

* Принимает входные параметры targetRef, shown, onShownChange, children, style и className.
* Вычисляет координаты (top и right) для позиционирования выпадающего списка по отношению к целевому элементу. Метод **getBoundingClientRect**() является частью объекта **Element** в браузере и позволяет получить информацию о размерах и координатах прямоугольной области элемента относительно видимой области окна браузера.
* Использует хук **useEffect** для установки начальных координат при первом рендере компонента.
* Также использует хук **useEffect** для добавления и удаления слушателей событий при изменении состояния **shown**. При отображении выпадающего списка, добавляется слушатель событий **click** на корневой элемент документа и слушатель событий **resize** на объект **window**. При скрытии выпадающего списка, удаляются эти слушатели.
* Функция **calcCoords** вычисляет координаты позиционирования выпадающего списка относительно целевого элемента.
* Компонент Dropdown возвращает результат функции **createPortal**, который создает DOM-элемент выпадающего списка и добавляет его в указанный контейнер с идентификатором "overlay".
* Если состояние shown равно false, то компонент Dropdown возвращает null, то есть выпадающий список не отображается.

**colorSchemeUtils**.ts содержит четыре функции, которые предназначены для управления темой оформления (цветовыми схемами) веб-приложения:

* Функция **applyScheme** принимает параметры scheme и persist, и устанавливает выбранную цветовую схему на элемент html веб-страницы путем добавления или обновления атрибута scheme у корневого элемента документа. Если параметр persist имеет значение true, то выбранная цветовая схема сохраняется в локальном хранилище браузера. Если persist имеет значение false, выбранная цветовая схема не сохраняется.
* Функция **getSystemScheme** использует метод matchMedia для определения цветовой схемы системы пользователя (dark или light). Если пользовательский интерфейс операционной системы настроен на темную тему оформления, функция вернет строку 'dark'. Если интерфейс настроен на светлую тему оформления, функция вернет строку 'light'.
* Функция **getSavedScheme** получает текущее значение выбранной цветовой схемы из локального хранилища браузера и возвращает его в виде строки 'dark' или 'light'. Если в локальном хранилище нет сохраненного значения цветовой схемы, функция возвращает значение null.
* Функция **removeSavedScheme** удаляет сохраненное значение цветовой схемы из локального хранилища браузера.

**Как я подключал Redux ToolKit**

Подключаем Redux ToolKit устанавливаем "react-redux" "@reduxjs/toolkit"

*npm i react-redux*

*npm install @reduxjs/toolkit*

*npm i react-redux-firebase*

Установили пакеты потом я сделал **store.ts** в папке **APP** как главное хранилище в нем,

по умолчанию сделал **configureStore** и добавил **reducer**

Потом отключил сеарилизацию поскольку у меня основные данные из firebase шли и они были не сереализуемые такие как : дата создания и дата обновления,

и в конце применяем трюк с devtools если он в продакшене то отключаем его на хрен.

и что бы закрыть тему **createSlice** то нужно добавить TypeScript-определения

export type RootState = ReturnType<typeof store.getState>;

Определение типа RootState в сущности говорит TypeScript, как выглядит ваше глобальное состояние в Redux. Это полезно потому, что:

* Предотвращает ошибки: Определение этого типа позволяет TypeScript убедиться, что вы используете правильные данные и свойства в своем Redux-состоянии.

export type AppAction<R> = ThunkAction<R, RootState, unknown, PayloadAction>;

Тип AppAction<R> может быть полезен для обработки данных, возвращаемых из асинхронных запросов.

Когда вы выполняете асинхронные операции в Redux, такие как загрузка данных с сервера, вы обычно используете Redux Thunk для выполнения этой операции и изменения состояния Redux на основе полученных данных. В этом контексте тип AppAction<R> представляет собой обобщенный тип, который позволяет определить тип возвращаемого значения после завершения асинхронной операции.

Например, если у вас есть функция действия для загрузки списка пользователей с сервера, то тип AppAction<User[]> позволит указать, что эта функция действия вернет массив пользователей после успешной загрузки.

Используя этот тип, вы можете обеспечить правильную типизацию и обработку данных, возвращаемых из асинхронных запросов, что поможет сделать ваш код более надежным и предсказуемым. то есть он может помочь когда что то приходит из асинхронного запроса

export type Dispatch = typeof store.dispatch;

* Когда мы определяем тип **Dispatch** как **typeof store.dispatch**, мы фактически говорим компилятору **TypeScript**, что тип **Dispatch** должен соответствовать типу функции **dispatch**, которая используется в нашем хранилище store. Это позволяет использовать тип **Dispatch** в других частях нашего кода для обозначения функции **dispatch**.
* Когда мы создаем функции действий (action creators) в Redux, мы можем указать, что они принимают параметр **dispatch** определенного типа **Dispatch**. Это помогает нам убедиться, что мы используем функцию **dispatch** правильно внутри этих функций действий, и компилятор **TypeScript** будет предупреждать нас, если возникнут ошибки.
* Таким образом, использование типа **Dispatch** помогает нам обеспечить более строгую проверку типов в нашем коде и облегчает поддержку приложения, делая его более надежным и понятным при разработке.

После этого создаем **createSlice** в него передаем нужные значения и поскольку делаем функционал для досок Получить Добавить Удалить поскольку все идет через API логичнее сделать функционал через **extraReducers**.

Я сделал в досках 2 экшена:

* получить все доски
* удалить конкретную доску

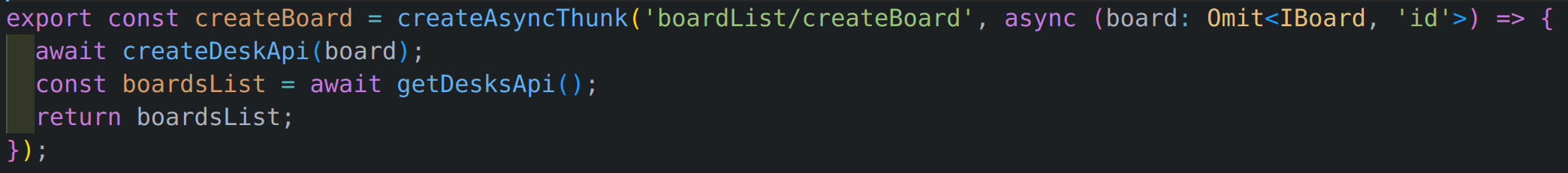
Они есть внутри приложения и так же выходят в API Firebase поэтому их записал в обычный reducer

В экстра reducer я добавил то что идет исключительно на API Firebase там конфигурируется и приходит в приложение :

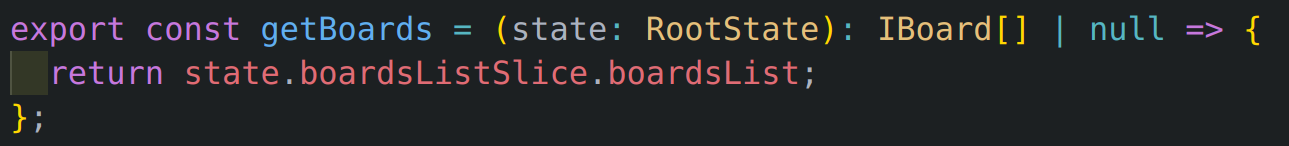
* создание доски
* редактирования доски

в самих экшенах делал обращения к функциям API Firebase и диспатчил результаты в стор





Селектор сделал что бы можно было брать данные из досок



Как я делал роутинг

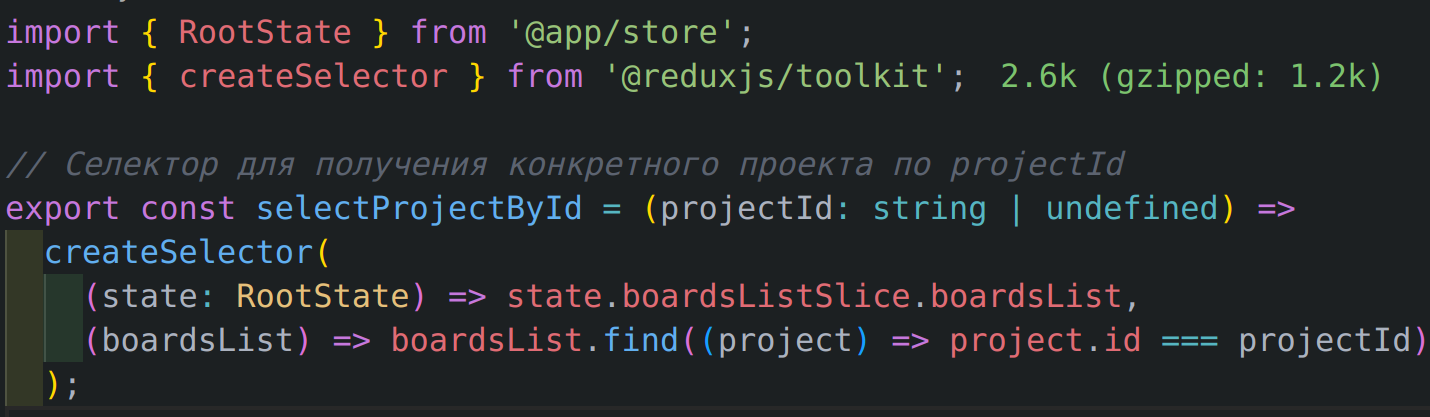
react-router-dom уже стоял я в App сделал роуты домашней страницы и карточки типа проекта.

Еще создал компонент page это обертка над теми страницами в ней есть заголовок и футтер, в заголовке аватар имя лейбл и иконка менты темы

Дальше реализовывал функционал, что при клике на доску создается отдельная карточка под эту доску, для этого повесил onClick на каждую доску в нем колбек функция сделал navigate с путем board и id этой доски.

Как я реализовывал передачу данных из одной фичи в другую передавал имя доски

для этого



**SelectProjectById** фабричная функция, которая создает мемоизированный селектор

она принимает один аргумент (id доски) и в теле вызывает метод **createSelector** в нем первым аргументом идет базовый селектор который находит нужные данные из стейта, а второй аргумент совершает операцию сравнения id проекта доски с текущим id.

И в компоненте использую результат селектора **SelectProjectById** который достает все данные из фичи BoardPage а именно данные о названии доски и дате ее создания и обновления

