**npm init -y**

Команда npm init -y используется для создания файла package.json в вашем проекте Node.js. Этот файл содержит метаданные о вашем проекте, такие как имя проекта, версия, описание, автор и другие важные данные [docs.npmjs.com](https://docs.npmjs.com/cli/v8/commands/npm-init/).

Флаг -y в команде npm init -y автоматически принимает все предлагаемые значения по умолчанию, которые npm init обычно предлагает в интерактивном режиме [stackoverflow.com](https://stackoverflow.com/questions/62725481/what-is-the-meaning-of-npm-init-y). Это означает, что вы не будете получать запросы на ввод данных и процесс инициализации будет выполнен автоматически.

Вот пример того, как будет выглядеть файл package.json, созданный с помощью npm init -y:

{

"name": "your-package-name",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC"

}

Вы можете затем редактировать этот файл вручную, чтобы добавить или изменить метаданные по мере необходимости.

Обратите внимание, что команда npm init -y необходима только при первоначальной настройке вашего проекта. После того как файл package.json был создан, вы можете устанавливать пакеты и добавлять их в ваш проект, используя команду npm install, и они будут автоматически добавлены в ваш файл package.json в секции dependencies или devDependencies [dzone.com](https://dzone.com/articles/a-beginners-guide-to-npm-the-node-package-manager).

**npm install webpack webpack-cli --save-dev**

Команда npm install webpack webpack-cli --save-dev используется для установки двух пакетов: webpack и webpack-cli, в качестве devDependencies вашего проекта [dev.to](https://dev.to/carriepascale/a-very-beginner-s-guide-to-webpack-2jal).

webpack является модульным сборщиком для современных JavaScript-приложений. Он преобразует модули с зависимостями в статические ассеты, представляющие эти модули [webpack.js.org](https://webpack.js.org/guides/installation/).

webpack-cli (CLI означает "Command Line Interface") это инструмент, который обеспечивает интерфейс для работы с webpack из командной строки. Он позволяет запускать webpack в командной строке и предоставляет некоторые дополнительные командные утилиты [npmjs.com](https://www.npmjs.com/package/webpack-cli/v/3.3.10).

Флаг --save-dev указывает npm установить эти пакеты как devDependencies, что означает, что они нужны только для разработки и тестирования, и не требуется для работы самого приложения в production [dev.to](https://dev.to/anitaparmar26/webpack-5-guide-for-beginners-314c).

После установки этих пакетов, вы можете использовать webpack для управления и оптимизации ваших JavaScript-файлов и других активов, таких как CSS и изображения. Вы также можете использовать webpack-cli для запуска webpack из командной строки или для написания пользовательских скриптов сборки.

Вот пример, как это может выглядеть в вашем файле package.json:

{

"name": "your-package-name",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^5.1.0",

"webpack-cli": "^4.0.0"

}

}

В этом примере добавлен скрипт build, который вызывает webpack, позволяя вам собирать ваш проект, запустив npm run build [dev.to](https://dev.to/carriepascale/a-very-beginner-s-guide-to-webpack-2jal).

**webpack.config.js**

**const path = require('path');**

Это подключает встроенный модуль Node.js под названием path, который предоставляет утилиты для работы с путями файлов и директорий.

**const HTMLWebpackPlugin = require('html-webpack-plugin')**

подключает плагин HTMLWebpackPlugin, который создает HTML-файл, который включает все ваши webpack-сборки в скрипт теги автоматически.

**module.exports**

Это аналог экспрота в js, только для node.js. Webpack будет использовать этот объект при запуске.

**mode: '….'**

development или production. В продакшене webpack сжимает код, убирает комменты и тд

**entry: './src/index.js'**

Это определяет точку входа в приложение. Webpack начнет обработку кода с этого файла.

**output: { ... }**

Это объект, определяющий, куда Webpack должен поместить собранные файлы.

*filename: '[name].[contenthash].js'*

Это определяет имя файла, который будет создан после сборки.

*[name]* – имя файла возьмется из entry.

*Если entry: {someFileName: './src/index.js' }*

То [name] = someFileName (ключ)

Если этого не указывать, по умолчанию entry возвращает имя файла main, т.е.

*entry: {main: './src/index.js' }*

*[contenthash]* – добавляет уникальный хэш номер. Если изменений небыло, то файл остается тем же, если были, то меняет номер хэша

*path: …………*

Это определяет путь, куда будет помещен собранный файл.

*path.resolve(\_\_dirname, 'dist'),*

*resolve* - это функция в Node.js, которая преобразует последовательность путей или сегментов пути в абсолютный путь.

*\_\_dirname* - это глобальная переменная в Node.js, которая содержит путь к директории, содержащей текущий исполняемый скрипт.

*'dist'* - это просто имя директории.

Таким образом, *path.resolve(\_\_dirname, 'dist')* преобразует эти два пути в один абсолютный путь. Если *\_\_dirname = /home/user/my\_project*,

то *path.resolve(\_\_dirname, 'dist')* вернет */home/user/my\_project/dist*.

Это означает, что Webpack будет помещать собранные файлы в директорию dist в той же директории, где находится webpack.config.js.

*clean: true*

очищает папку в *path*. При новой сборке удаляет старые файлы

**new HTMLWebpackPlugin({...})**

плагин для Webpack, который автоматически создает HTML-файл, который включает все ваши webpack-сборки в скрипт теги. Это удобно, потому что все JavaScript-файлы могут быть автоматически добавлены в HTML-документ при сборке.

*template: path.resolve(\_\_dirname, 'public', 'index.html')*

путь к шаблону HTML-файла, который будет использоваться для создания итогового HTML-файла. (из шаблона возьмется все из body и не только, к этому добавится импорт скриптов и тд и получится index.html, лежащий в path из output)

*new webpack.ProgressPlugin()*

это плагин, который выводит прогресс сборки в консоль. Это может быть полезно для отслеживания прогресса сложных или долгих процессов сборки

**module.rules**

Это массив правил, которые Webpack будет использовать для обработки различных типов файлов.

*{test: /\.tsx?$/, use: 'ts-loader',: /node\_modules/}*

Это правило говорит Webpack обрабатывать все файлы с расширениями .ts или .tsx (TypeScript файлы) с использованием ts-loader. ts-loader - это загрузчик Webpack, который компилирует TypeScript в JavaScript. exclude: /node\_modules/ говорит Webpack игнорировать все файлы в папке node\_modules.

**resolve.extensions**

массив расширений файлов, которые Webpack будет автоматически понимать при импорте файлов. Это значит, что вы можете импортировать файл index.ts просто как index, и Webpack будет понимать, что вы имеете в виду index.ts.

1.2

затем сначала установили **typescript**, перевели .js файлы в .ts и перевели webpack.config.js в …ts. Затем **декомпозировали** webpack.config.js: создали папку config, и в нее добавляем ф-ии, которые будут возвращать нужные списки или объекты для конфига. Делается для читаемости конфига. Сами настройки за каким то хером тоже засунули в отдельную функцию (buildWebpackConfig). В нее передаем объект с указанием режима и путей. Функция возвращает готовый конфиг. Почему это нельзя было сделать в самом webpack.config.ts я хз

1.3

установили **dev-server** для отслеживания изменения в файлах (чтоб не надо было каждый раз пересобирать сборку)

и т.к. используем typescript, то устанавливаем типы для dev server (npm install --save-dev @types/webpack-dev-server).

Затем добавили переменные окружения (webpack env variables). Прописали их в package.json, добавляя к скриптам «--env port=3000».   
прописали для них тип в config.ts  
Для работы со значениями из окружения (env) переписали webpack.config.ts, заменив

export default config;

на

export default (env) => {retutn config}

1.4

Подключаем **React** (npm i react react-dom) – основные зависимости

Подключаем типы для react (npm i -D @types/react @types/react-dom) – зависимости для разработки

Т.к. в реакте используем typescript, то нужен ts loader, но его уже подключали.

Если бы использовали нативный js, нужно было бы подключать babel-loader для перевода кода из последних версий ECMAscript чтобы поддерживалось всеми браузерами.

Далее подключаем css. Но берем сразу scss т.к. будем использолвать препроцессоры. Вбиваем в гугл webpack scss, переходим в webpack **sass-loader**. Здесь инструкция по установке именно препроцессоров, но нам нуджно еще css-loader и style-loader. Потому добавляем их в команду.

npm install sass-loader sass webpack style-loader css-loader --save-dev

и добавляем лодеры в вебпак конфиг лоадерс

1.5

Сейчас при сборке css вставляется в js файл. По хорошему для этого нужен отдельный файл, потому подключаем спец плагин **MiniCssExtractPlugin**. (npm install --save-dev mini-css-extract-plugin). Добавили плагин (new MiniCssExtractPlugin({…})). Затем подключаем лоадер. Т.к. этот плагин не должен использоваться с style-loader(подключали ранее вместе с sass) то в cssLoader меняем "style-loader" на “ MiniCssExtractPlugin.loader”.

Допустим есть два модальных окна и при появлении к ним обоим добавляется класс “open” появляется конфликт имен классов. Можно использовать методологию БЭМ и именовать классы [название файла]\_[название окна]\_[что с окном происходит], но тогда имена классов получаются слишком громоздскими.

Есть др. вариант – **CSS MODULES**. Настраиваем вебпак на работу как с scss фалами так и с css modules (у них название = [имя файла].module.scss). Меняем импорт css на import classes from './Counter.module.scss' и дальше получаем классы css путем classes.button. Названия классов теперь будут генерироваться из случайного набора символов, что предотвращает совпадения. Но теперь typescript не понимет что здесь надо импортировать. Решение по ссылке. (<https://stackoverflow.com/questions/41336858/how-to-import-css-modules-with-typescript-react-and-webpack>).

Допустим, мы хотим, чтобы часть css файлов импортировалась как css modules с заменой имен (это будут файлы, в имени которых есть .module.), а остальные импортировались как обычные, без замен имен классов. Для этого по документации добавляем auto: auto: (resPath: string) => resPath.includes('.module.') ---- resPath = полный путь до файла вместе с его полным названием ==== возвращает тру, если там есть '.module.'.

Итого: При запуске в dev режиме, доп файлы css не создаются, весь код остается в js файле (благодаря options.isDev ? 'style-loader' : MiniCssExtractPlugin.loader)

При запуске в прод создается отдельный css, в котором файлы, содержащие '.module.' – в них классы переименовываются в [набор букв 8 символов] а ссылающиеся на эти классы html объекты в коде получают имя класса [path][name]\_\_[local]--[hash:base64:5]. Файлы без '.module.' остаются как есть.

1.6

Устанавливаем **React-router-dom** (npm i react-router-dom) и типы для него в дев (npm i -D @types/react-router-dom)

Добавили MainPage.tsx и AboutPage.tsx, настроили роуты и линки на них в App.

Сейчас, если перейти на сайте на страницу отличную от главной и обновить, выходит ошибка *Cannot GET /about*. Т.к. разрабатываем сингл пэйдж апликэйшон(у нас один html файл), Заходим на гит webpack dev server, тнаходим там --history-api-fallback, добавляем historyApiFallback в buildDevServer со значением тру и все ок.

Теперь есть несколько страниц (tsx файлов) – main и about, но при сборке все они попадают в один js файл. Если пользователю нужна будет только одна страница, ему всеравно придется загрузить все. Это плохо. Есть механизмы, меняющие это. Принцип в том, что сборка весит не много и содержит в себе информацию о роутинге на другие страницы, тогда пользователь при переходе на нужную страницу будет получать только ее без загрузки других. Называется асинхронные чанки \ **code splitting** \ **lazy loading** .. названии разные, суть одна. В React делается с помощью функций спец. компонентов lazy и suspense. ([https://react.dev/reference/react/lazy](https://react.dev/reference/react/lazy#load)) (<https://legacy.reactjs.org/docs/code-splitting.html>). Поддерживается импорт только экспортируемых по дефолту компонентов (export default)

1.7

Создали структуру файлов стилей, определили глобальные переменные.

**переключение тем** (light \ dark): добавили переменные scss, добавили кнопку, написали обработчик, все работает, но, чтобы можно было использовать и в других компонентах, по правильному, нужно создать **контекст(ThemeContext.tsx)**.

*Контекст в React - это механизм, который позволяет передавать данные через дерево компонентов без необходимости явно передавать пропсы на каждом уровне. Это особенно полезно, когда один и те же данные должны быть доступны во многих компонентах на разных уровнях дерева и вложенности. Контекст позволяет распространить эти данные и их изменения на все компоненты ниже по дереву.*

*Провайдер (Provider) - это специальный компонент, который позволяет передавать данные через контекст. Любой компонент, который находится внутри провайдера, может получить доступ к данным контекста.*

Чтобы работать с контекстом нужно создать **провайдер(ThemeProvider.tsx)**. Если обернуть приложение в этот провайдер, то в любом месте приложения будет доступ к контексту для изменения темы.

Выносим логику по плучению темы из контекста и по переключению темы в хук UseTheme.tsx и далее, если хотим где то добавить возможность изменения темы, обращаемся к этому хуку.

1.8

**ClassNames** – библиотека для условного объединения имен классов (ClassNames). Упрощает указание классов в нодах. Сделаем что то свое на подобии (**сlassNames.ts**).

**Git**

Создали **.gitignore**, создали репозиторий на github,

git init