**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

Флаг -y в команде npm init -y автоматически принимает все предлагаемые значения по умолчанию, которые npm init обычно предлагает в интерактивном режиме. 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"

}

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.

**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.

webpack является модульным сборщиком для современных JavaScript-приложений. Он преобразует модули с зависимостями в статические ассеты, представляющие эти модули.

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.

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.

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.

**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 нужно 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 понимает 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.

Последний абзац спорный.. возможно я не до конца разобрался

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)**.

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.

Теперь, при изменении темы в одном месте, об этом будет знать все приложение

const ThemeProvider: FC<PropsChildren> = ({children}) => {

здесь и далее FC (functional component) препод указывает для того, чтобы задать тип элементу, попадающему в функцию {children}). В основном для того, чтобы у него было свойство children: React.ReactNod. Но, т.к. используем разные версии React, в моей (5, новее) это устарело, и FC не содержит свойства children. Поэтому, я везде добавляю доп интерфейс для расширения FC.

*Context API это альтернатива Redux (с его store)*

Context API идеально подходит для управления глобальными состояниями, которые не изменяются часто и не имеют сложной логики. Он предоставляет прямой способ передачи данных через дерево компонентов без необходимости вручную передавать пропсы на каждом уровне. Это особенно полезно, когда у вас есть данные, которые должны быть доступны глобально в дереве компонентов React, такие как предпочтения темы или языка.

С другой стороны, Redux предлагает более мощный инструмент для управления состоянием приложения. Он предоставляет централизованный магазин, где состояние приложения хранится в одном месте и может быть доступно отовсюду. Redux также предоставляет средства для обработки асинхронных действий и дебаггинга состояния приложения. Однако, Redux может быть избыточным для менее сложных приложений, и его использование может привести к дополнительной сложности и объему кода 1.

В вашем случае, поскольку вы управляете глобальным состоянием темы, которое не изменяется часто и не имеет сложной логики, использование Context API может быть более подходящим решением. Однако, если в будущем ваше приложение станет более сложным и требует более продвинутого управления состоянием, вы можете рассмотреть возможность перехода на Redux или другой библиотеку управления состоянием 1.

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

git remote add origin <https://github.com/Ant-on-git/react-learning-project.git>

закомитили и отправили.

1.9 (теория **feature-sliced**)

В ходе курса будем использовать методологию **feature-sliced**(Архитектурная методология для фронтенд проектов).

<https://feature-sliced.design/ru/docs/get-started/overview>

Суть: идем по пирамиде снизу вверх. Кадлый слой может использовать только слои под ним.

*Shared* – переиспользуемые участки кода (кнопки, карточки, UIKIT и тд, то, что ни как не привязвно к бизнесу.. хелперы, конфиг приложения, )

*Entities* – конкретные бизнес-сущности, кот. относятся к приложению (сущность пользователя, статьи, комментария т тд. В каждой сущности есть сегменты, кот. относятся только к ней, например, запросы к api по редактированию профиля пользователя, участок state, который отвечает за профиль пользователя, компонент карточки профиля пользователя, т.е. все участки кода, кот. связаны конкретно с этой бизнес сущностью.

*Features* – фичи. Авторизация, регистрация, оплата товара, добавление товара в корзину, создание договора… Любые бизнес фичи, кот несут ценность приложению. Юизнес фичи как правило взаимодействуют с сущностями (entities). Каждую фичу можно назвать отдельным модулем. Нужно сделать их максимально не связвнными с другими фичами, изолированными и независимыми.

*Widgets* – header, nav bar, side bar, menu, footer… Внутри сегменты, кот. хранят вспомогательные участки кода, напр, сами компоненты, хелперы..

*Pages* – страницы – объединяет в сеье все, что перечислено выше и формирует страницу

*Processes* – (опционально) Как правило процессы протекают через несколько страниц, напр, создание слоэной сущности, которое проходит в несколько этапов и тд.

*App* – включает в себя корневой компонент, глоюальные стили, глобальные провайдеры, все, что влияет глобально на приложение.

Основа методологии это public api. Рассмотрим на примере фичи(feature) авторизации. Она включает в себя:

- rememberMe

- loginForm

- loginButton

- authRequest (запрос на авторизацию)

- loginValidator

- passwordValidator

- loginReducer

- loginActions

Все это называется модулем (модулем авторизации). Наружу(наверх) отдаем только определенные участки:

– LoginForm, которая включает в себя все остальное

- и, например, typescript типы, чтобы с ними выше работать

Таким образом вся система со всеми связями остается замкнутой, а наверх отдаем ровно столько, сколько необходимо.

1.10. (внедряем **feature-sliced**)

Для каждого слоя создаем отдельную папку в src(shared, entities, features, widgets, app).

app.tsx и папку styles(с глобальными стилями) переносим в папку app. Также в app создали папку types и перенесли туда global.d.ts.

удалили папку components со счетчиком – больше не нужен.

Переносим папку themes (контекст, провайдер и хук по изменению темы):

* в *app/providers/ThemeProvider/ui* переносим сам провайдер (ThemeProvider.ts) т.к. он оборачивает все приложение
* в *app/providers/ThemeProvider/lib* переносим ThemeContext.tsx (контекст) и хук UseTheme.ts,

По словам препода, темы будет использовать все приложение(видимо имеется ввиду слои между shared и app в feature-sliced) и получается, что themes можно было закинуть в нижний слой *shared/config*. Но препод вот решил закинуть наверх. Может и не будет все приложение использовать, а только верхний слой, пока я хз.

lib – папка функций типа className, которую сделали раньше, хуки типа UseTheme…

в *app/providers/ThemeProvider* создали idex.ts – он будет экспортировать наружу все, что нужно.

Теперь, во-первых, при изменении структуры файлов и папок приложения не избежны импорты типа ‘../../../../../….’, во вторых, в приложении будет public api, и хотелось бы с верхнего импорта **абсолютными импортами** **из** папки **src** доставать то, что нужно. Для этого нужно настроить tsconfig.json чтобы указывать импорты относительно.

- раскомментируем строку "baseUrl": ".",

- добавляем на том же уровне "paths": { "\*" : ["./src/\*"] }

Далее нужно настроить вебпак на абсолютные импорты. За это отвечает resolvers. Поиск webpack absolute imports. Препод говорит перходить по первой ссылке гугла(Module Resolution), но по факту берет инфу из (Resolve)

**public api** это когда компоненты не просто разбросаны в виде вайлов .jsx \ .tsx, а каждый компонент (или какой то объект, который используется в коде) распологается в специальной для него папке, и есть файл (.js \ .ts), который передает «наружу» только то, что нужно передать. Например, виджет навбар (представим его в качестае public api позднее, здесь только в качестве наглядного примера) распологается в папке navbar.

Navbar

- ui

- Navar.tsx

- index.ts

Сам компонент располагается в Navar.tsx. Его импортирует и отдает наружу index.ts. Везде, где в коде нужно получить навбар обращаемся к index.ts.

Наводим порядок в папке pages: Каждую страницу (MainPage, AboutPage) превращаем в модуль: Сами страницы (AboutPage.tsx) переносим в *pages/aboutPage/ui*, туда же AboutPage.async.ts. В *pages/aboutPage* создаем index.ts и из него импортируем асинхронную страницу. Аналогично для MainPage

Переносим папку *helpers/classNames в shared/lib/* т.к. функция classNames будет использоваться (для добавления классов) абсолютно везде

2.11 (move routing from app to shared)

Наводим порядлк в app.tsx. Это корень проекта и он должен быть максимально чистым.

Выносим роутинг в отдельную папку (*app/router*) т.к. это «такая глобальная штука, которая использует внутри себя страницы». Сейчас роуты описаны прям в компоненте

<Suspense fallback={<div>Loading...</div>}>

<Routes>

<Route path={'/'} element={<MainPage/>} />

<Route path={'/about'} element={<AboutPage />} />

</Routes>

</Suspense>

Но хотелось бы иметь какой то конфиг (*shared/config/routeConfig*), внутри которого объявим список роутов, а в *app/router* будем проходить по этому списку.

2.12 (12. move navbar (App - widgets). Links - shared/ui)

Создаем widgets/navbar.

Компоненты, кот. не требуют асинхронного чанка(lazy loading \ code splitteng) будем экспортировать не по дефолту, а именовано. Навбар как раз такой компонент, тк.к он будет везде и нет смысла для него делать lazy

поднастроили сниппет по созданию компонентов(rfc) для именно этого приложения.

Создали компонент для ссылок (*shared/ui/AppLink*) и добавили для них немного css.

2.13 (13: create ThemeSwitcher, create Button component)

.prettierrc – файл настроек расширения prettier для vs code – авто формат файлов на табуляцию

Переносим из App кнопку по переключению тем в виджеты, т.к. по мнению препода, переключение темы это не бизнес сущность, и этой кнопке место в виджетах. Но потом препод решил перекинуть папку в shared/ui. Потому что.

**FIGMA**: *https://www.figma.com/file/UfS42XWKWPNagbgWkv7aer/Production-ULBI-TV?type=design&node-id=2-11&mode=design&t=NepMBMbiw17hFKMl-0*

Вставляем иконки для смены темы (.svg) import LightIcon from 'shared/assets/icons/theme-light.svg';

Typescript ругается на такой импорт, потому что по умолчанию вебпак не понимает что с этими файлами (.svg) делать. Чтоб исправить добавляем лоадер. Гугл - **svgr webpack**. Устанавливаем (npm install @svgr/webpack --save-dev). По факту еще надо typescript настроить, но препод пока пропустил.

Добавили иконку юзера формат .png. Для него нужен другой лоадер: **file-loader**. Он устарел для вебпак 5+ (теперь нужно использовать asset modules) но пока идем по шагам препода и ставим file-loader. Делаем все по доке, плюс дополнительно прописываем возможность работы со шрифтами (woff|woff2 в test: /\.(png|jpe?g|gif|woff|woff2)$/i,).

Теперь настраиваем typescript: гугл – import svg typescript.

Создаем отдельный компонент button (shared/ui/button)

2.14 (14: add sidebar)

Сделали sidebar (widget)

2.15 (15: translation, i18n)

Добавляем **языки** \ **интернационализация** \ **i18n**

i18next - это международная библиотека JavaScript, которая помогает вам локализовать ваше приложение

Гугл – react i18n next

npm install react-i18next i18next –save

npm install i18next-http-backend i18next-browser-languagedetector –save

настраиваем i18next в *shared/config/i18n*

В настройке есть пункт debug: true / false. Нужно сделать так, чтобы значение устанавливалось в зависимости от используемого режима: isDev ? true : false.

Т.е. нужно получить из вебпака значение isDev. Для этого в вебпаке есть плагин, позволяющий прокидывать переменные через все приложение. Гугл – **webpack define Plugin**. Настраиваем в плагинах. Добавили переменную \_\_IS\_DEV\_\_, импользуем ее в *shared/config/i18n/ i18n.ts*. Теперь ругается typescript, т.к. он не знает что это за переменные такие. Добавляем декларацию типов в global.d.ts (declare const \_\_IS\_DEV\_\_: boolean;)

Импортируем конфигурацию i18n в главный файл index.tsx

Создаем файлы с переводами в *public/locales/<language\_code>/translation.json* (по документации) Папку можно изменить (<https://github.com/i18next/i18next-http-backend>)

Сейчас пользователю прогружается весь файл с переводом. Если он зашел на одну страницу, то ему пришел перевод с этой и других страниц. Можно это разбить. Создаем [page\_name].json в папках с переводами и оставляем там перевод для конкретной страницы. Далее нужно указать аргумент для useTranslate – название namespace - имя файла, хранящего перевод для страницы. Напр, для русского перевода страницы about.tsx, нужно создать *public/locales/ru/about.json* и в useTranlate(about). Если этого не указывать, по умолчанию = useTranlate(translation) и берется файл *public/locales/ru/translation.json.*

*Все сделанное работает, но у препода загружается только выбранный перевод, а у меня и ru и en. Например. При переходе на страницу about у меня загружается и en/about.json и ru/about.json. Быстро исправить это не удалось. Возможно из за разницы версий. По документации (*[*https://github.com/i18next/i18next-http-backend?tab=readme-ov-file#readme*](https://github.com/i18next/i18next-http-backend?tab=readme-ov-file#readme)*) написано добавить load: "languageOnly", Добавил, теперь работает через раз. Иногда грузится только выбранный, иногда сразу оба. Разобрался вроде – грузится и тот, что указан по дефолту и установленный. Т.е. если по дефолту установлен ru и выбран ru то грузится только ru. Но, теперь, при нажатии на перевод, загружается перевод и для текущей страницы и для других. Разобрался – помогло backend: { loadPath: "/locales/{{lng}}/{{ns}}.json", },*

Выносим функцию по переключению языка в отдельный компонент *shared/ui/LangSwitcher (можно в widgets)*.

2.16

Webpack hot module replacement

Препод говорит, что полохо работает с компонентами, что потом на курсе поставим какую то другую рабочую версию. Да и я не понял нахрен он нужен, если dev server и без того сразу отображает изменения в css. Крч я ничего не делал здесь.

2.17 (17: babel, i18next – babel-plugin-i18next-extract)

Смотрим плагин для i18next – babel-plugin-i18next-extract – позволяет автоматически при сборке вытаскивать ключи (видимо то, что будет переводиться) в отдельный файл

Я хз как у него по запросу вылезло одно, а у меня другое, но в итоге конфиг этого плагина лежал в *https://i18next-extract.netlify.app/#/*

Для него нужен babel (npm install --save-dev babel-loader @babel/core)

По документации babel:

* добавили плагин для babel
* установили (npm install @babel/preset-env --save-dev), кот. Трансформирует код из ES2015+ в более старые
* создали файл настроек для babel: babel.config.json

Устанавливаем babel-plugin-i18next-extract: (npm i --save-dev babel-plugin-i18next-extract)

* добавляем настройки для плагина в babel.config.json
* добавляем плагин в webpack – plugins – babelLoader
* донастраиваем плагин в webpack – plugins – babelLoader использую конфиг из [*https://i18next-extract.netlify.app/#/*](https://i18next-extract.netlify.app/#/)

все.

2.18 (18: add ESLint)

**ESLint**

Гугл – eslint – (npm install eslint --save-dev) – (npm init @eslint/config) - … - standard style guide (препод выбрал Airbnb но у меня его не было) - … - в корне создается файл конфиг .eslintrc.js

Запустить ESLint: в консоли npx eslint “\*\*/\*.{ts,tsx}”

(“\*\*/\*.{ts,tsx}” – регулдярка на те файлы, кот. хотим проверить)

Установил плагин eslint для vscode.

Запустить автоисправление проблем, которые можно автоисправить: в верхней строке vscode >ESLINT: fix all auto-fixable Problems  
крч ошибки получились слишком разные и чтоб хоть что то понимать, я заменил файл конфига еслинта на тот что у препода. Стал ругаться на отсутствие airbnb, я установил его (npm install --save-dev eslint-config-airbnb)

Далее куча правок и ввода правил в eslint

Пофиксить все что можно: npx eslint “\*\*/\*.{ts,tsx}” --fix

Добавили скрипты:

* поиск ошибок npm run lint:ts
* пофиксить npm run lint:ts:fix

все.

2.19 (19: stylelint, i18next for eslint)

**Stylelint** – правила для css файлов

Гугл – stylelint – (npm install --save-dev stylelint stylelint-config-standard-scss) – (создали .stylelintrc.json) – (установил расширение stylelint для vscode и посыпались ошибки).

Настройки в .stylelintrc.json

Кучу времени потратил на правило indentation, а оказывается оно устарело в старых версиях блять, а интернет херню писал в поисках.

Подключаем плагин для i18next для eslint – будет подсвечивать текст. Для которого нет перевода и он нужен

Гугл – i18next eslint - npm install eslint-plugin-i18next --save-dev – далее по документации

Добавил отсутствующие переводы для ссылок

2.20 (20: add jest and first tests)

Настраиваем **тестирование**. Подключаем **Jest**.

Гугл – jest - (npm install --save-dev jest) – (npm init jest@latest)

– yes – yes – jsdom – no – babel – yes

Появился конфиг для jest – jest.config.ts, его перенесли в config/jest т.к. такие глобальные конфиги лучше из корня выкидывать

--------- далее смотрим на ютубе про тестирование javascript (https://www.youtube.com/watch?v=y2emL1fMRyY) ---------

Настраиваем jest.config.ts..

Т.к. jest.config.ts перенесли в др место, для скрипта нужно указать адрес к этому файлу (package.json)

Пишем тест для shared/lib/classNames

Для тестов нужны типы. Устанавливаем (npm i --save-dev @types/jest)

Eslint ругается на describe -> добавляем jest: true в env в .eslintrc.js

Запускаем тесты (m run unit) – ошибка: в новых версиях нужно в ручную устанавливать jest-environment-jsdom. Установили (npm install -D jest-environment-jsdom) и тесты пошли.

Сделали тест для classNames() – ошибка – jest не знает что такое typescrtipt и его команды – документация jest – использование с typescript – (npm install --save-dev @babel/preset-typescript)

2.21 (21: add loader, NotFoundPage)

Создали NotFoundPage, добавили роуты для него

Делаем loader для индикации загрузки асинхронных чанков (shared/ui/PageLoader) но препод не уверен что он должен быть там.

Сам лоадер(css) находим по поиску в гугле css loaders

Меняем в AppRouter лоадер на новый

3.22 (22: add ErrorBoundary)

Обработка ошибок, компонент ErrorBoundary (для обработки не зафиксированных ошибок)

ErrorBoundary работает только с классовыми компонентами и не будет перехватывать ошибки, выброшенные внутри функциональных компонентов или хуков. Если ваш DummyComponent является функциональным компонентом, это может быть причиной, почему ErrorBoundary не работает как ожидалось.

Гугл \ ErrorBoundary React

ErrorBoundary обрабатывает все ошибки, кроме тех, что происходят в асинхронном коде, событиях, при server side рендеринге и ошибки, кот. возникают в самом ErrorBoundary.

ErrorBoundary будет провайдером, который оборачивает все приложение.

Providers/ErrorBoundary

i18next ругается на отсутствие перевода текста ошибки. В функуиональных компонентах для перевода использовали хук useTranslation(), но в классовых компонентах (а ErrorBoundary классовый) использовать хуки нельзя.

Для классовых компонентов в i18next есть такой хок(обертка) withTranslation

Пример использования интернационализации в классовом компоненте:

import React, { ErrorInfo, ReactNode } from "react";

import { withTranslation } from "react-i18next";

interface ErrorBounderyProps { children: ReactNode }

interface ErrorBounderyState { hasError: boolean }

class ErrorBoundary extends React.Component<ErrorBounderyProps, ErrorBounderyState> {

    constructor(props: ErrorBounderyProps)

        super(props);

        this.state = { hasError: false };

    }

    static getDerivedStateFromError(error: Error) {return { hasError: true };}

    componentDidCatch(error: Error, errorInfo: ErrorInfo) {console.log(error, errorInfo}

    render() {

        const { hasError } = this.state;

        const { children} = this.props;

        if (hasError) {

            return <h1>Something went wrong.</h1>;

        }

        return children;

    }

}

export default withTranslation()(ErrorBoundary); // <------

Но мы так делать не будем, а будем вместо текста ошибки (<h1>Something went wrong.</h1>) возвращать функциональный компонент с ошибкой.

Оборачиваем приложение в <ErrorBoundary>

Тестируем: добавляем в App.tsx ошибку:

    useEffect(() => {

        throw new Error();

    }, []);

Добавляем компонент для ошибки

Widgets/PageError

3.23 (23: add Bundle Analyzer)

Анализ размера банда

Webpack bundle analyzer

npm install --save-dev webpack-bundle-analyzer

настраиваем в config/build/buildPlugins.ts

по документации добавлять через require, но у нас обычные импорты, потому меняем на обычный импорт.

Устанавливаем типы для плагина: npm i --save-dev @types/webpack-bundle-analyzer

npm start - одни размеры файлов

npm run build:dev - другие размеры

npm run build:prod - третьи размеры

если замечаем что сборка весит дофига, устанавливаем плагин и смотрим где от чего можно избавиться

3.24 (24: add react testing library, set settings and fixed bugs)

React testing library

Важно! Когда мы добавим include в tsconfig у нас сломается сборка проекта. В данном ролике это не проявится, потому что мы будем работать с тестами. В следующем уроке в самом начале мы исправим эти проблемы со сборкой!

npm install --save-dev @testing-library/react

пишем тест src/shared/ui/button/button.test.tsx

хотим проверить наличие текста в отрендеренном компоненте, но отсутствует метод .toBeInTheDocument()

чтоб его добавить нужно установить testing-library/dom:

npm install --save-dev @testing-library/jest-dom

по документации (https://github.com/testing-library/jest-dom#usage ) нужно донастройить:

* создать файл конфиг (config/setupTests.ts)
* указать на этот файл в основном файле конфиге (здесь config/jest.config.ts))
* подружить с typescript (прописать в tsconfig.json строку include с указанием на файл)

донастроили, метод не появился. Убрали строку из tsconfig и добавили обратно и теперь появился. Магия, блять. Но если перезапустить vs code, опять методов нет, пока не удалить и добавить строку. Кроме того, после добавления строки в tsconfig.json приложение не запускается!

Пытаемся запустить тест кнопки… и… он падает.. Нужно еще babel донастроить за каким то хером. Переходим по ссылке из ошибки, говорит надо установить @babel/preset-typescript, но он уже установлен. Препод: «*Получается, что не хватает пресета для реакта.*» КАК ТАК ПОЛУЧАЕТСЯ Я ХУЙ ЕГО ЗНАЕТ. Этот придурок опять НИХЕРА НЕ ОБЪЯСНИЛ.

Вбиваем в поиске jest react babel «*и по первой же ссылочка переходим*». А там нет прямиого объяснения. Находим строку для установки (npm install --save-dev jest babel-jest @babel/preset-env @babel/preset-react react-test-renderer)

И каким то хуем понимаем, что нам не хватает именно @babel/preset-react.

Устанавливаем его: npm i -D @babel/preset-react

И добавляем в babel.config.json в спсок "presets" '@babel/preset-react'

Запускаем тест.. и.. он падает. Теперь ругается на импорт import cls from './Button.module.scss'; в Button.tsx. т.к. jest ничего не знает ни про scss ни про css modules.

Поиск – jest css modules - <https://jestjs.io/docs/webpack#mocking-css-modules>

Находим там Mocking CSS Modules

- npm install --save-dev identity-obj-proxy

- добавляем (по документации строки в jest.config.ts)

Запускаем тест кнопки.. и.. он падает. С новой ошибкой ReferenceError: React is not defined

Гугл – jest ReferenceError: React is not defined

<https://stackoverflow.com/questions/58980934/referenceerror-react-is-not-defined-in-jest-tests>

(в babel.config.json меняем

'@babel/preset-react'

на

['@babel/preset-react', {"runtime": "automatic"}]

)

Препод: *это чтоб автоматически подставлялся импорт react в файлах.*

запускаем.. и ОНО РАБОТАЕТ!!!!!!

**Такие ошибки это нормально, т.к. используется много сторонних модулей, которые сами по себе друг о друге не знают и их надо подружить.**

Делаем тест на sidebar (widgets/sidebar/ui/Sidebar.test.tsx)

Тест падает т.к. в компоненте Sidebar используются svg

Нужно в jest.config.ts настроить импорт для svg файлов

*Поскольку благодаря плагину* (видимо какой то вебпак плагин) *svg импортируются как готовые компоненты, можно сделать подобный мапер:*

‘\\.svg’: path.resolve(\_\_dirname, ‘jestEmptyComponent.tsx’)

*Грубо говоря, это* (jestEmptyComponent.tsx) *своего рода мок, который будет использоваться для всех импортов, в которых присутствует svg*

Переведя на человеческий, как я понял, там, где есть импорт svg, вместо той svg будет вставлен jestEmptyComponent.tsx

Ошибку вроде исправили, но теперь у меня одна ошибка, а у препода другая. Оказывается я не добавил export в jestEmptyComponent.tsx

Исправил и.. у меня ТЕСТ РАБОТАЕТ!!

А у препода ругается на использование асинхронной функции в i18next. На всякий решение (https://stackoverflow.com/questions/42535270/regeneratorruntime-is-not-defined-when-running-jest-test)

Тест запускается, но в консоли warning на useTranslation

Гугл – 18next testing - <https://react.i18next.com/misc/testing>

Надо завернуть компонент в хок withTranslation()()

Завернули, но ошибка осталась та же (react-i18next:: You will need to pass in an i18next instance by using initReactI18next)

По словам препода, это другая ошибка «*о том, что мы не прокинули инстанс i18next: для приложения мы импортировали его в index.tsx, а здесь мы рендерим отдельный компонент*».

Находим в докментации (<https://react.i18next.com/misc/testing>) раздел «Testing without stubbing» он касается именно тестирования компонентов с i18next «без заглушки». В нем нужно создать тестовый конфиг.

Создаем shared/config/i18n/i18nForTest.ts

Чтобы не делать импорт этого файла в каждый тест, делаем хелпер

shared/lib/tests/renderWithTranslations/renderWithTranslations.tsx

По документации, нужно обернуть нужный компонент с i18next в

            <I18nextProvider i18n={i18n}>

                <Sidebar/>

            </I18nextProvider>,

Но у меня это приводит к тому, что компонент не рендерится и тесты не проходят. Эти ошибки заебли, еду дальше. Оказывается импортировался не тот i18n, точнее не из того файла (брался главный вместо того что специально для тестов). Починили крч.

**Сначала нужно делать простые тесты. Чтобы настроить тестовую среду и убрать возможные ошибки**

После добавления и настройки react testing library в прошлом уроке появились проблемы с импортами

import cls from './Sidebar.module.scss';

«*Cannot find module './Sidebar.module.scss' or its corresponding type declarations.ts(2307*)»

Чтоб исправить надо в tsconfig.json в поле include добавить регулярку на все файлы компонентов

(./src/\*\*/\*.tsx , ./src/\*\*/\*.ts)

24 (24: add React testing library, storybook)

**STORYBOOK**

«*Здесь создаются разные состояния для компонентов чтобы другие разработчики могли посмотреть как они выглядят, какие состояния могут принимать*.»

Гугл – storybook webpack 5 - npx sb init --builder webpack5

После установки появляются папки:

1. в корне .storybook - настройки
2. src/stories - примеры использования «из коробки»

storybook из корня перенесли в config/

после переноса запустить не удается. Нужно:

1. добавить флаг в скрипт с указанием нового адреса (storybook dev -p 6006 -c config/.storybook)
2. изменить адреса историй в файле .storybook/main.ts

большинство строрис это копипаста и изменение пропсов, которые нужно передать в компоненты

пишем первый сторис src/shared/ui/button/Buton.stories.tsx

сопировали все из примера. Запускаем, ошибка: ругается на абсолютный импорт в компоненте Button

(import { classNames } from 'shared/lib/classNames/classNames';)

Т.к. дефолтная конфигурация storybook не знает про абсолютные импорты.

Гугл – storybook absolute path – на stackoverflow есть совет, но препод пошел по другому пути.

«*У storybook есть своя webpack конфигурация, там все настроено по дефолту, и если нужно, можно переопределять*»

Гугл - Storybook extends webpack- <https://storybook.js.org/docs/builders/webpack#override-the-default-configuration>

Сайт видимо обновился, у препода решение походе, но чуть другое.

Создаем config/.storybook/webpack.config.ts

И берем там конфиг вебпака и меняем его чтоб добавить адрес для абсолютных путей

Ошибку исправили, появиласяь новая: ERROR in ./src/shared/ui/Button/Button.module.scss 1:0

Ругается на css modules

Гугл – storybook css modules - <https://stackoverflow.com/questions/59761361/storybook-ui-with-css-modules-and-less>

(далеко не первая ссылка)

Крч берем конфиг css loader из основного вебпака и добавляем его в вебпак сторибука. (config/.storybook/webpack.config.ts)

Запускаем и.. работает!

Но.. теперь проблема с глобальными переменными scss. Чтоб они работали препод просто сделал импорт главного файла с темами в файл с тестом, а у меня эта хрень не работает. Похер, едем дальше.

«*Каждый оаз импортировать стили в файл с тестами это не удобно. Для этого есть декораторы*»

Создаем декоратор для импорта стилей shared/config/storybook/StyleDecorator/StyleDecorator.ts

Полключаем декоратор в config/storybook/preview.ts (addDecorator(\_\_\_))

И… переменные один хрен не применяются.

Далее препод создаем декоратор для переключения темы themeDecorator, но, т.к. у меня переменные не работают, пропускаем.

Хотя всеравно добавил (shared/config/storybook/ЕhemeDecorator /ЕhemeDecorator.ts), но как и предполагалось, не работает.

Пишем сторис для sidebar

Запускаем.. ругается на отсутствиие импорта react в компонентах.

Решение <https://stackoverflow.com/questions/74995855/storybook-canvas-referenceerror-react-is-not-defined>

Исправили.. запускаем.. ругается на испорт svg..

Отключаем встроенное в сторибук правилодля обработки svg, и вставляем лоадер свгшек из основного вебпака.

Работает

Крч глобальные переменные в стилях применяются. Стало понятно после добавления сайдбара – там размер прописан в глоб переменных и он работал. Цвета же не применялись по тому что накомпоненты не был повешен класс. После добавления (в компоненте ThemeDecorator) цвета тоже стали применяться.

Делаем сторис для навбара

Ругается по сути что используем links, не обернутые в BrowserRouter. Сделали RouterDecorator и заработало

Делаем сторис для ErrorPage, AppLink, лоадера, ThemeSwitcher

Все работает сразу

Делаем сторис для AboutPage, MainPage, NotFoundPage

Все работает сразу

25 (25: add loki skreen tests)

Скриншотные тесты

Гугл – loki testing – (npm i -D loki) – (npx loki init --config ./config/.storybook)

Появилась настройка loki в package.json:

  "loki": {

    "configurations": {

      "chrome.laptop": {

        "target": "chrome.app",

        "width": 1366,

        "height": 768,

        "deviceScaleFactor": 1,

        "mobile": false

      },

      "chrome.iphone7": {

        "target": "chrome.app",

        "preset": "iPhone 7"

      }

    }

Простыми словами – скриншоты снимаем для хрома на ноутбуке и для хрома на айфоне.

Запуск тестов: сначала запускаем сторибук (npm run storybook) затем в другом терминале тесты локи (npx loki test)

У меня все норм, но для инфы:

У препода по умолчанию было chrome.docker вместо chrome.app, из за чего тесты падали. Он исправил на chrome.app

и теперь падают с другой ошибкой:

*Connect ECONNREFUSED ::1:51863*

*Some visual test failed to run*

В итоге причина в [node\_modules/@loki/target-chrome-app/src/create-chrome-app-target.js](mailto:node_modules/@loki/target-chrome-app/src/create-chrome-app-target.js)

В этом файде импортируется сторонний const chromeLauncher = require('chrome-launcher');

И че то с ним не так. Позднее ошибку пофиксили, но преподу пришлось менять chrome.app обратно на chrome.docker, установить и запустить docker на компе и только потом тесты loki пошли.

После того как тесты выполнились, в корне появилась папка loki, в которой кучи скриншотов – это скрины всех компонентов с ноутбучного браузера и с мобильного. Если после этого поменять что то в каком то компоненте, например цвет, скрин тест loki этого компонента упадет и появятся файлы в папках loki/difference. *Это называется регрессионное тестирование, нужно чтоб проверять что после правок компонента в нем ничего не сломалось*.