Покрытие теста кодом

В проекте настроена среда тестирования, поэтому надо называть файлы правильно.

1. Создаем файл с названием:

<filename>.test.ts

<filename> - имя файла, который будем тестировать

1. Пишем в файл тест, например:

import {sum} from './01'  
  
***test***('sum should be correct', () => {  
 // starting data  
 const a = 1;  
 const b = 2;  
 const c = 3;  
 // action  
 const result = sum(a, b);  
 //expect result  
 ***expect***(result).toBe(3);  
})

1. Далее мы можем запустить тест в терминале

npm test

На данном этапе npm анализирует се файлы с \*.test.\* и запускает их.

1. Тесты автоматически перезапускаются после изменения и сохранения файла теста.

//Можно выносить переменные вне тестов, например

const a = 1;  
const b = 2;  
const c = 3;  
  
//перед выполнением теста можно запускать call-back функцию  
  
***beforeEach***(()=>  
{  
 a = 1;  
 b =2;  
 c =3  
})

Если дописать к тесту .skip, то он выполняться не будет.

***test***.skip('test city should contain hospital and fire station', () => {  
})

ОБЪКТЫ

TypeScript это про типизацию.

Допустим у нас есть такой объект

const ***student*** = {  
 name: "Dimych",  
 age: 32,  
 isActive: false,  
 address: {  
 streetTitle: "Surganova 2",  
 city: {  
 title: "Minsk",  
 countryTitle: "Belarus"  
 }  
 },  
 technologies: [  
 {id:1, title: "HTML"},  
 {id:2, title: "CSS"},  
 {id:3, title: "React"},  
 ]  
}

Для него обязательно надо сделать **типизацию.** Если в объекте есть вложенные объекты, для них нужно писать **отдельную типизацию**. Удобнее идти от детей к родителям.

type CityType = {  
 title: string  
 countryTitle: string  
}  
type AddressType = {  
 streetTitle: string  
 city: CityType  
}  
  
type TechType = {  
 id: number  
 title: string  
}  
  
type StudentType = {  
 id: number  
 name: string  
 age: number  
 isActive: boolean  
 address: AddressType  
 technologies: Array<TechType>  
}

Поиграемся с тестами:

ФУНКЦИИ

Подход **функционального программирования**: функции должны быть чистыми, они не могут изменять ничего, что приходит в них. Например, функция, которая добавляет свойство объекта, не укладывается в концепцию функционального программирования.

**ООП**  наоборот, говорит о том, что объект это главное, нужный результат мы должны получать путем изменений и преобразований объектов.

04. Array.prototype.FILTER()

Метод фильтр можно записать следующим образом:

const predicate = (age: number) => {  
 return age > 90;  
}  
  
const oldAges = ages.filter(predicate);

Но, можно значительно сократить запись с использованием стрелочных функций:

const cheapCourses = courses.filter((course: CourseType) => {  
 return course.price < 160  
});

Если в предикативной функции нет логики, можно **убрать return и {}**

const cheapCourses = courses.filter((course: CourseType) =>  
 course.price < 160  
);

Также можно не писать **тип данных**

const cheapCourses = courses.filter(course => course.price < 160);

05. Array.prototype.MAP()

Часто используется для преобразования массива данных, например

type Names2Type = {  
 id: number  
 name: string  
}  
  
function Names2() {  
  
 const namesArray2 = [  
 {  
 id: 1,  
 name: 'Mike'  
 },  
 {  
 id: 1,  
 name: 'John'  
 },  
 {  
 id: 1,  
 name: 'Stan'  
 }  
 ]  
  
 const liNamesArray2 = namesArray2.map((item: Names2Type, index: number) => {  
 return <li key={item.id}>{index} --- {item.name}</li>  
 })  
  
 return (  
 <div>  
 <ul>  
 {liNamesArray2}  
 </ul>  
 </div>  
 )  
  
}

ВНИМАНИЕ! Часто нужно присваивать KEY для самого наружного тега, выводимого мапом!

06. CallBack, onClick, onChange, onBlur

Callback hell

Callback функция тоже является объектом. Ее можно создать и отправить куда-то. ЕЕ там вызывали - он колл-бек.

Коллбеки можно отдавать в другие функции. Например setTimeout:

БЕЗ СКОБОК!!!

window.setTimeout(callback, 1000) - передаем функцию

~~window.setTimeout(callback(), 1000)~~ - передаем ВЫЗОВ функции. Так делать нельзя, setTimeout сам ее вызывает!!!

const callback = () => {  
 alert('Hey')  
}  
  
window.setTimeout(callback, 1000)

При этом мы сами не вызываем коллбек, его вызывает setTimeout.

// Обработка событий UI

в событие можно передавать вызов только одной функции

<button onClick={deleteUser}>delete</button>

Если нужно вызвать при клике несколько функций, создаем функцию, которая будет вызывать несколько функций.

Например,

const deleteUser = () => {  
 alert('deleteUser')  
 alert('other function2')  
 alert('other function3')  
}

При любом событии происходит создание объекта события. Например, при клике **создается объект** со свойствами координат, где произошло событие и тд.

Например, при клике мышкой мы можем передать какие-то значения в функцию-коллбек(т.к. при событии создается объект). Тип данных этого объекта будет в данном случае:

MouseEvent<HTMLButtonElement>

тип данных нужно импортировать из React. Итого мы получим:

import {MouseEvent} from 'react';  
  
export function User() {  
  
 const deleteUser = (event: MouseEvent<HTMLButtonElement>) => {  
 alert('deleteUser')  
 }  
  
 const saveUser = () => {  
 alert('saveUser')  
 }  
  
 return (  
 <div>  
 <span>Dimych</span>  
 <button onClick={deleteUser}>delete</button>  
 <button onClick={saveUser}>save</button>  
 </div>  
 )  
}

07. Деструктурирующее присваивание

Например, нам нужно внести в отдельные переменные свойства объекта, чтобы было дальше удобнее с ними работать.

let man = {  
 name: 'Dimych',  
 age: 32,  
 lessons: [  
 {title: '1'}, {title: '2'}  
 ]  
}  
  
const age = man.age  
const lessons = man.lessons

Но! это можно сделать намного проще с помощью ДП:

const {age, lessons} = man

ИМЕНА ПЕРЕМЕННЫХ должны совпадать с названиями свойств!

Также ДП можно использовать В ФУНКЦИЯХ

Создадим компонент, где будем использовать ДП:

import {ManType} from './07\_Destructuring\_assignment.test';  
import React from 'react';  
  
type PropsType = {  
 title: string  
 man: ManType  
}  
  
export const ManComponent: React.FC<PropsType> = (props) => {  
 **const {title, man: {name}} = props // присваиваем переменные через деструктуризацию**  
  
 return (  
 <div>  
 <h1>{title}</h1>  
 <hr/>  
 <div>  
 {man}  
 </div>  
 </div>  
 )  
}

ДП можно сделать прямо в PROPS!

export const ManComponent: React.FC<PropsType> = **({title, man}**) => {  
 return (  
 <div>  
 <h1>{title}</h1>  
 <hr/>  
 <div>  
 {name}  
 </div>  
 </div>  
 )  
}

Объяснение:

на входе PROPS это объект, и сразу используя этот объект используем ДП.

НО! В этом случае мы не сможем получить доступ к остальным свойствам объекта PROPS

Есть выход - использование **оператора spread (или rest?) таким образом**:

export const ManComponent: React.FC<PropsType> = ({**title, man,...props**}) => {  
 return (  
 <div>  
 <h1>{title}</h1>  
 <hr/>  
 <div>  
 {man}  
 </div>  
 {props.children}  
 </div>  
 )  
}

или можно сделать так:

export const ManComponent: React.FC<PropsType> = (props) => {  
 const {title, man, ...resProps} = props  
 return (  
 <div>  
 <h1>{title}</h1>  
 <hr/>  
 <div>  
 {man}  
 </div>  
 </div>  
 )  
}

Это может понадобиться, если нужно прокинуть PROPS дальше.

Деструктуризация в МАССИВАХ

Синтаксис:

[variable1, variable2] = array

Это не делается для больших массивов.

**Как наиболее частый пример - использование в useState()**

export const ManComponent: React.FC<PropsType> = (props) => {  
  
 const [message, setMessage] = useState<string>('hello')  
  
 return (  
 <div>  
 <h1>{message}</h1>  
 <hr/>  
 <div>  
   
 </div>  
 </div>  
 )  
}

Если нужно пропустить какой=либо элемент массива, пишем так:

**const [ , , variable3] = array**

**или так используя …rest**

**const [variable1, …restArray] = array**

**Если в тесте нужно затестить соответствие равенства объекту, пишем так:**

expect(restLessons[0].toStrictEqual({name: 'react', title: '3'}))

07. Ассоциативные массивы

Это структура данных это **совокупность пар «ключ-значение»**. Доступ к значению осуществляется по имени ключа.

доступ к свойствам объекта можно получить двумя способами

**user.address.city.title**

или

**user[“address”][“city”][“title”]**

**Создадим ассоциативный массив, в нем мы сможем иметь доступ ко всем элементам объекта как по индексам в массиве**

let usersObj = {

"1": "Name1",

"2": "Name2",

"3": "Name3",

"4": "Name4",

}

**Доступ к элементам:**

usersObj[0]

или

usersObj[“0”]

**ИТЕРАЦИЯ:**

Object.keys(usersObj) - вернет массив ключей ["1", "2", "3", "4", "hi how are you"]

Object.values(usersObj) - вернет массив значений ["Name1", "Name2", "Name3", "Name4", "fine"]

Добавить user id в ассоциативный массив: (O(1))

const users:UsersType = {  
 '101': {id: 101, name: 'Dimych'},  
 '1201': {id: 1201, name: 'Nat'},  
 '1101': {id: 1101, name: 'Val'},  
 '201': {id: 201, name: 'Kate'},  
}  
  
let user = {id: 100500, name: "Igor"}  
  
  
users[user.id] = user

ИТОГ

101: {id: 101, name: "Dimych"}

201: {id: 201, name: "Kate"}

1101: {id: 1101, name: "Val"}

1201: {id: 1201, name: "Nat"}

100500: {id: 100500, name: "Igor"}

потому что user.id = 100500 и к этому свойству мы добавляем объект.

Удалить из ассоциативного массива: (O(1))

delete users[10500]

ТЕСТЫ ВАЖНО

type UsersType = {  
 [key: string]: { id: number, name: string }  
}  
let users: UsersType = {}  
  
beforeEach(() => {  
 users = {  
 '101': {id: 101, name: 'Dimych'},  
 '1201': {id: 1201, name: 'Nat'},  
 '1101': {id: 1101, name: 'Val'},  
 '201': {id: 201, name: 'Kate'},  
 }  
})  
  
test('Should update corresponding user', () => {  
 users['201'].name = 'Ekaterina'  
 expect(users['201'].name).toBe('Ekaterina')  
})  
  
test('Should be equal structure', () => {  
 users['201'].name = 'Ekaterina'  
 expect(users['201']).toEqual({  
 id: 201,  
 name: 'Ekaterina'  
 })  
 }  
)  
  
test('Should delete corresponding user', () => {  
 delete users["201"]  
 expect(users['201']).toBeUndefined()  
 }  
)