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', () => {  
})

ТИПИЗАЦИЯ (TYPESCRIPT) ОБЪЕКТОВ

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'}))

08. 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()  
 }  
)

09. Reference type, mutability, мутации

Парадигма функционального программирования **не допускает изменения исходных объектов.**

Изменение **объектов** и их передача происходит по ссылке объекта. Нельзя просто присвоить значение объекта1 = оъект2 (так мы передаем ссылку **reference type**, соответственно все свойства будут меняться).

**Изменение массивов** происходит так же, передача по ссылке. Чтобы не изменять исходный массив нужно его скопировать!

**Примитивы НЕ передаются** по ссылке!

Нужно избегать МУТИРУЮЩИХ операций!

10. IMMUTABILITY

Работа React и Redux построена на **парадигме функционального программирования.**

Если не планируется изменение вложенных свойств объекта, то делать полную копию при выполнении функции не обязательно.

Очень много примеров функционального программирования без мутирования объектов в уроке №15. Например:

test('09\_replace comany name', () *=>* {

*const* replaceCompanyUser = replaceCompany(user, 2, 'Peri')

  expect(user).not.toBe(replaceCompanyUser)

  expect(user.laptop).toBe(replaceCompanyUser.laptop)

  expect(user.address).toBe(replaceCompanyUser.address)

  expect(user.books.length).toBe(replaceCompanyUser.books.length)

  expect(user.companies).not.toBe(replaceCompanyUser.companies)

  expect(user.companies.length).toBe(3)

  expect(replaceCompanyUser.companies.length).toBe(3)

  expect(user.companies[0].title).toBe('Epam')

  expect(replaceCompanyUser.companies[0].title).toBe('Epam')

  expect(replaceCompanyUser.companies[1].title).toBe('Peri')

})

export *let* user: UserType = {

  name: 'Dimych',

  hair: 32,

  address: {

    city: 'Minsk',

    house: 12

  },

  laptop: {

    title: 'Zenbook',

    screen: 17

  },

  books: ['css', 'html', 'js', 'react'],

  companies: [

    { id: 1, title: 'Epam' },

    { id: 2, title: 'Itransition' },

    { id: 3, title: 'ITincubator' }

  ]

}

//09\_replace comany name

export *function* replaceCompany (

*user*: UserType,

*toReplaceId*: *number*,

*replaceComp*: *string*

) {

  return {

    ...*user*,

    companies: [

      ...*user*.companies.map(*i* *=>*

*i*.id === *toReplaceId* ? (*i* = { ...*i*, title: *replaceComp* }) : *i*

      )

    ]

  }

}

Изменение ассоциативного массива:

export *const* companies: companiesPropsType = {

  Dimych: [

    { id: 1, title: 'Epam' },

    { id: 2, title: 'Peri' },

    { id: 3, title: 'Itransition' }

  ],

  Igor: [

    { id: 1, title: 'Maz' },

    { id: 2, title: 'Mzkt' },

    { id: 3, title: 'Doka' }

  ],

  Migel: [

    { id: 1, title: 'ISoft' },

    { id: 2, title: 'Science' },

    { id: 3, title: 'Incubator' }

  ]

}

//10\_associative array modifying

export *function* upadteCompany (

*companyList*: companiesPropsType,

*guy*: *string*,

*companyId*: *number*,

*company*: *string*

) {

  // Важно создать копию, иначе перезатрется исходный объект

*let* companyCopy = { ...*companyList* }

  return {

    ...companyCopy,

    [*guy*]: companyCopy[*guy*] = companyCopy[*guy*].map(*i* *=>*

*i*.id === *companyId* ? (*i* = { ...*i*, title: *company* }) : *i*

    )

  }

}

test('10\_associative array modifying', () *=>* {

*const* updatedCompanyList = upadteCompany(companies, 'Igor', 2, 'Kamaz')

  expect(updatedCompanyList['Igor'][1].title).toBe('Kamaz')

  expect(companies['Igor'][1].title).toBe('Mzkt')

})

14. ПРОМИСЫ

**Промис** - это ОБЪЕКТ, обещание, которое возвращает функция вместо реального ответа.

и у этого **ОБЪЕКТА ЕСТЬ ТОЛЬКО МЕТОДЫ (НЕТ СВОЙСТВ). Он может находиться в одном из трех состояний:**

**- pending**

**- resolved (fullfilled)**

**- rejected**

Я тебе обещаю что-то вернуть, ушел искать инфо!)

**МЕТОДЫ Промисов:**

**.THEN - когда обещание выполнится (зарезолвится), выполни функцию-колл-бек функцию. В нее приходят данные data, которые приходит после запроса на сервер**

axios.get('https://api.weatherstack.com/current').then(*data* *=>* {

  console.log(*data*)

})

**.CATCH - отловить ошибку если ‘rejected’, т.е в случае отклонения промиса. Как аргумент возвращает в колл-беке причину отказа.**

axios.get('https://stats.nba.com/stats/playbyplay').catch(*error* *=>* {

  console.log(*error*)

})

**Обычно пишут в цепочку два метода then и catch:**

axios

  .get(

    'http://api.rottentomatoes.com/api/public/v1.0/lists/movies/box\_office.json'

  )

  .then(*data* *=>* { // если все хорошо, пользуемся данными

    console.log(*data*)

  })

  .catch(*error* *=>* { // если есть ошибка, считываем ее

    console.log('error is...' + *error*)

  })

**.FINALLY - если нам не важно, зарезолвится ли промис или реджектнется, получаем обратно промис. Т.е.** Когда промис был выполнен, вне зависимости успешно или с ошибкой, указанная функция колл-бек будет выполнена. Это даёт возможность запустить один раз определённый участок кода, который должен выполниться вне зависимости от того, с каким результатом выполнился Promise. В КОЛЛ-БЕК НЕ ПРИХОДИТ НИКАКИХ ДАННЫХ!

promise1

  .then(*data* *=>* {

    console.log(*data*)

  })

  .catch(*error* *=>* {

    console.log('error is...' + *error*)

  })

  .finally(() *=>* {

    console.log('finally')

  })

**СТАТИЧЕСКИЕ МЕТОДЫ Promise**

Все промисы являются экземпляром класса Promise. У Класса Promise также есть методы.

**.ALL()**

**Прим anotherPromise зарезолвится тогда, когда будут выполнены все промисы, переданные в виде перечисляемого аргумента**

**Если какой-то из промисов незарезолвится, то и общий промис anotherPromise тоже не зарезолвится**

*let* anotherPromise = *Promise*.all([promise1, promise2])

**Можно получить массив результатов, например двух промисов, если пользоваться методом .all. При этом отправлять отдельные запросы не надою Данные мы получим, когда зарезолвятся оба промиса.**

anotherPromise

  .then(*result* *=>* console.log(*result*)) // data из двух промисов

  .catch(() *=>* {

    console.log('initialization failed')

  })

**.allSettled()**

возвращает промис, который исполнится когда все полученные промисы завершены (исполнены или отклонены), содержащий массив результатов исполнения полученных промисов.

Т.е. получаем data не зависимо от того, есть ли незарезолвленные промисы.

*let* anotherPromise2 = *Promise*.allSettled([promise1, promise2, promise3])

anotherPromise2.then(*results* *=>* {

  console.log(*results*)

})

У возвращаемых объектов обязательно будет поле: status, значениями которых могут быть fulfilled и rejected.

Другие статические методы:

Promise.**resolve**(value). Сразу резолвит промис со значением аргумента. Можно использовать как заглушку.

*const* resolvedPromise = *Promise*.resolve([1, 2, 3])

Promise.**reject**(value). Сразу реджектит промис

*const* rejectedPromise = *Promise*.reject({ message: 'some error' })

rejectedPromise

  .then(*e* *=>* {

    console.log(*e*)

  })

  .catch(*e* *=>* {

    console.log(*e*)

  })

**ЦЕПОЧКА THEN , ЧЕЙНИНГ**

Каждый вызов then возвращает новый промис (создается новый Promise)

*let* promise = axios

  .get('https://api.weatherstack.com/current')

  .then(*data* *=>* *data*.config) // зарезолвится конфигом

  .then(*data* *=>* *data*.url) // зарезолвится url

  .then(*data* *=>* console.log(*data*)) // заберем url

Цепочка нужна для работы в сложных сценариях.

**ASYNC AWAIT**

У слова **async** один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически.

Ключевое слово **await** заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

**await это по сути альтернатива then**

**Может работать только в АСИНХРОННОЙ ФУНКЦИИ!!!**

async *function* testAsync () {

*let* data = await axios.get('https://api.weatherstack.com/current')

  console.log(data)

}

testAsync()

**ЗАЧЕМ СОЗДАВАТЬ СВОИ ПРОМИСЫ. ПРОМИСИФИКАЦИЯ**

Например, есть сложные вычисления, которые занимают много времени, то лучше эту операцию сделать асинхронной, т.е. ее **промисифицировать**

**Было:**

*function* getNumber () {

  return  Math.random()

}

*const* n1 = getNumber()

*const* n2 = getNumber()

console.log(n1)

console.log(n2)

Стало после ПРОМИСИФИКАЦИИ:

*function* getNumber () {

  return *Promise*.resolve(Math.random()) // используем примитивный Promise.resolve

}

getNumber().then(*num* *=>* console.log(*num*))

Создадим Промис сами с помощью **new Promise. В нем мы можем прописывать любую логику.**

*function* getNumber () {

*const* promise = new *Promise*((resolve, reject) *=>* {

    resolve(Math.random())

    reject('error')

  })

  return promise

}

getNumber().then(*num* *=>* console.log(*num*))

getNumber().then(*num* *=>* console.log(*num*))

**Использование с localStorage.**

Проблема localStorage в том, что оно синхронное.

Создадим два варианта работы синхронное и асинхронное

*const* repo = {

  saveSync (*data*) { // синхронный метод

    try {

      localStorage.setItem('some-key', JSON.stringify(*data*))

    } catch (error) {

      return false

    }

    return true

  },

  saveAsync (*data*) { // асинхронный метод

*const* promise = new *Promise*((resolve, reject) *=>* {

      try {

        localStorage.setItem('some-key', JSON.stringify(*data*))

        resolve()

      } catch (error) {

        return reject(error)

      }

    })

    return promise

  }

}

*const* promise = repo.saveAsync({ name: 'IT-KAMASUTRA' })

promise

  .then(() *=>* {

    console.log('async saved')

  })

  .catch(*error* *=>* {

    console.log('not saved' + *error*)

  })

Вариант вызова с помощью async/await

*const* run = async () *=>* {

    await repo.saveAsync({ name: 'IT-KAMASUTRA' })

    console.log('saved');

  }

или

*const* run = async *data* *=>* {

  await repo.saveAsync({ name: *data* })

  console.log('saved async await')

}

run('IT-AWAIT!!!')

Напишем функцию, которая будет через секунду выводить числа

*function* waits (*timer*) {

*const* promise = new *Promise*((res, rej) *=>* {

    setTimeout(() *=>* {

      res()

    }, *timer*)

  })

  return promise

}

async *function* count () {

  await waits(1000).then(() *=>* {

    console.log('resolved')

  })

  console.log('1')

  await waits(1000).then(() *=>* {

    console.log('resolved')

  })

  console.log('2')

  await waits(1000).then(() *=>* {

    console.log('resolved')

  })

  console.log('3')

}

count()

**ОБРАБОТКА ОШИБОК TRY CATCH**

Метод **catch()** возвращает промис (Promise() и работает только в случае отклонения промиса.

**в СИНТАКСИСЕ THEN**



Если ошибка не поймана, цепочка продолжается

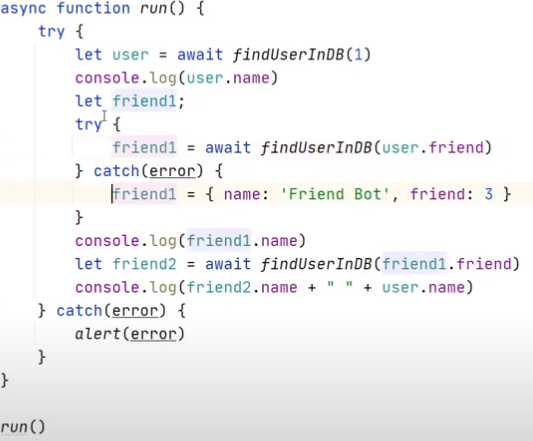
**в СИНТАКСИСЕ ASYNC AWAIT**

Обработка ошибок происходит путем оборачивания в конструкцию try catch.

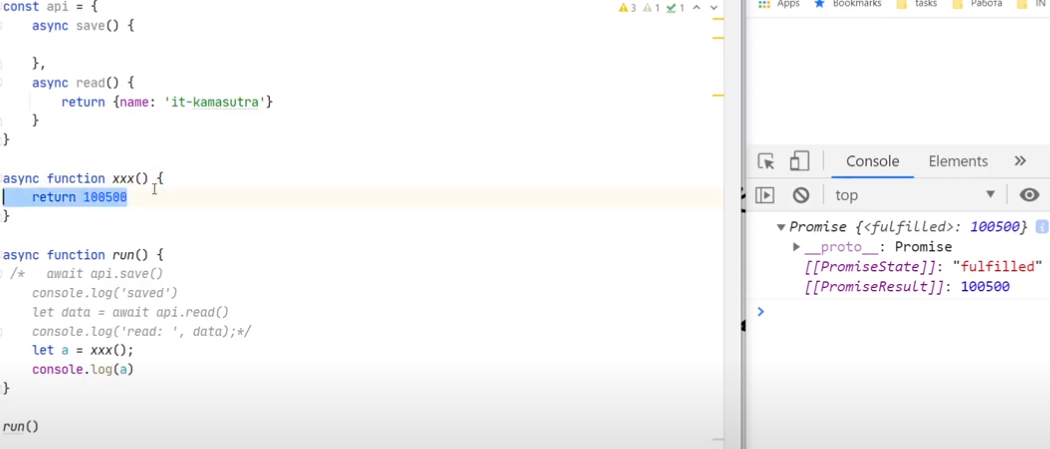
Будет перехвачена любая ошибка в цепочке



Если нужно обыграть отдельный случай, можно его отдельно обернуть в try catch



**Любая функция с async возвращает ПРОМИС!!!! Так можно сделать промисификацию любой функции**

****

Объекты, классы, ООП

**Для чего служат классы:**

- Создание однотипных объектов

- Для реализации ООП принципов:

а) инкапсуляция (сокрытие свойств объекта)

б) наследование

в) полиморфизм

**Какие есть способы создания объектов?**

1. С помощью литерала объекта
2. Конструктор new Object()
3. С помощью классов

*Создание объекта с помощью* ***литерала объекта (СОВРЕМЕННЫЙ ПОДХОД)****:*

*const* deleteUser = *userId* *=>* {

*const* action = {

    type: 'DELETE\_USER',

    payload: {

      userId: *userId*

    }

  }

  return action

}

*const* action1 = deleteUser('123')

*const* action2 = deleteUser('456')

*Создание объекта с помощью* ***функции-конструктора*** *(ЛУЧШЕ НЕ ПОЛЬЗОВАТЬСЯ):*

*function* DeleteUserAction (*userId*) {

  // создается(определяется) пустой this = {}

  this.type = 'DELETE\_USER'

  this.payload = {

    userId: *userId*

  }

  // return писать не надо!!!

}

*const* action3 = new DeleteUserAction('789')

*Создание объекта с помощью* ***Класса (СОВРЕМЕННЫЙ ПОДХОД)****:*

*class* DeleteUserActionClass {

*constructor*(*userId*) {

    this.type = 'DELETE\_USER'

    this.payload = {

      userId: *userId*

    }

  }

}

*const* action4 = new DeleteUserActionClass('654')

*const* action5 = new DeleteUserActionClass('325')

про добавление методов см. файлы в проекте.

**ИНКАПСУЛЯЦИЯ, НАСЛЕДОВАНИЕ В КЛАССАХ**

Класс - это «инструкция», с помомщью которой мы можем создавать однотипные объекты

1. **Инкапсуляция**. Нельзя пользоваться напрямую, только через методы.

* Ранее использовалось **нижнее подчеркивание \_,** чтобы сообщить другим о приватности поля.

*class* DeleteUserActionClass {

*constructor*(*userId*) {

    this.\_type = 'DELETE\_USER'

    this.payload = {

      userId: *userId*

    }

  }

}

* Можно использовать **геттеры и сеттеры и #**

*class* UserWithCounter {

  #name = ''

*constructor* (*name*) {

    this.#name = *name*

  }

  get name () {

    return this.#name

  }

  set name (*newName*) {

    this.#name = *newName*

    return this.#name

  }

}

const u1 = new UserWithCounter('John')

u1.name // только так можно получить доступ к name

u1.name = 'Mike' // только так можно поменять name

1. **Наследование**. Например, уже создан класс, но потом нам понадобилось добавить какое-то поле. Часто лучше не исправлять исходный класс, а расширить существующий.

*class* CoderUser extends UserWithCounter {

  code() {

    console.log(`I am ${this.name}, here is my code: code... `)

  }

}

*const* coder1 = new CoderUser('Mi', 'cnn1.com', '11.11.1990')

В примере выше coder1 будет иметь все поля UserWithCounter плюс CoderUser.

Иногда нужно добавить свойства в наследуемый класс с использованием конструктора. Для этого, чтобы наследовать все свойства родительского класса, нужно пользоваться **методом super()**

*class* CoderUser extends UserWithCounter {

*constructor* (*name*, *site*, *dob*, *tech*) {

    super(*name*, *site*, *dob*) // берем все поля родительского класса, передаем аргументы

    this.tech = *tech // создаем свое свойство*

  }

  code () {

    console.log(`I am ${this.name}, here is my code: code... `)

  }

hello() {

    super.hello() // вызываем метод родительского класса и потом переопределяем

    console.log('Go away!');

  }

}

*const* coder2 = new CoderUser('Mi', 'cnn1.com', '11.11.1990', ‘JS’)

1. **Полиморфизм**. Это практика проектирования объектов для совместного использования поведения и возможности переопределения общего поведения конкретными объектами. **Полиморфизм использует преимущества наследования для того, чтобы это произошло.**

Например, создаем массив кнопок в UI, которые являются экземплярами класса.

**Классовые компоненты в React?**

*class* ProfilePage extends React.Component {

  render() {

    return 'IT kamasutra subscribe!'

  }

}

// <ProfilePage/> react создает объект const obj = new ProfilePage

// <ProfilePage/> react создает объект const obj = new ProfilePage

// obj.render() реакт вызываем метод render()

Можно добавить функцию-конструктор:

*class* ProfilePage extends React.Component {

*constructor*(*props*) {

    super(*props*) // наследуем свойства родительского класса

  }

  render() {

    return 'IT kamasutra subscribe!'

  }

}

**SOLID приниципы**

**SOLID — принципы в JS**

* S — SRP -(single responsobility) — **Принцип** разделения ответственности
* O — OCP — (open-closed) — **Принцип** открытости-закрытости
* L — LSP — (Liskov) — **Принцип** подстановки Барбары Лисков
* I — ISP — (interface segregation) — **Принцип** разделения интерфейсов
* D - DIP -- (dependency Inversion Principle) -- Принцип инверсии зависимостей

**Single responsibility. Принцип разделения ответственности** *.Класс должен быть ответственен лишь за что-то одно.* Если класс отвечает за решение нескольких задач, его подсистемы, реализующие решение этих задач, оказываются связанными друг с другом. Изменения в одной такой подсистеме ведут к изменениям в другой.  
  
Обратите внимание на то, что этот принцип применим не только к классам, но и к компонентам программного обеспечения в более широком смысле.

**Open-closed. Принцип открытости-закрытости.** *Программные сущности (классы, модули, функции) должны быть открыты для расширения, но не для модификации.*

**Liskov. Принцип подстановки Барбары Лисков***. Необходимо, чтобы подклассы могли бы служить заменой для своих суперклассов.* Цель этого принципа заключаются в том, чтобы классы-наследники могли бы использоваться вместо родительских классов, от которых они образованы, не нарушая работу программы. Если оказывается, что в коде проверяется тип класса, значит принцип подстановки нарушается.

**Interface segregation. Принцип разделения интерфейсов.** *Создавайте узкоспециализированные интерфейсы(types), предназначенные для конкретного клиента (объекта, функции, массива и пр). Клиенты не должны зависеть от интерфейсов, которые они не используют.*

**Dependency Inversion. Принцип инверсии зависимостей.** *Объектом зависимости должна быть абстракция, а не что-то конкретное.*

1. Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
2. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.