* npm create next-app@latest <nombre proyecto>
* npx create-next-app@latest
* pnpm create next-app <nombre proyecto>

# 17 inferior

## /pages

Aquí se van a ubicar las rutas (es como sveltkit),

Al nombrar rutas evitar usar el ‘-’

Si se crea un archivo cuyo nombre sea distinto des ‘index.jsx’, se considera una ruta a partir de esta carpeta /miruta

## /pages/MiRuta/index.jsx

Si se crea un carpeta y dentro un index.jsx entonces esto es una nueva ruta /Miruta donde se renderiza el contenido de index.jsx

export default function index(){

    return (

        <>

        Hello

        </>

    )

}

# /app/page.tsx .jsx

Es la primera vista del proyecto

Cada nueva carpeta que se cree que tenga un archivo page.tsx o .jsx, se considera una ruta

Y debería de tener una función componente llamada Page, aunque creo que no es ncesario este nombre

export default function Page(){

    return (<>

        <h2>Mi ruta1</h2>

    </>)

}

# /app/layout.tsx .jsx

Tiene el componente layout por defecto

Envuelve a sus hijos según la carpeta en la que este

export default function Layout(

    {children}: {children:React.ReactNode}

){

    return(<>

        Mi Layout

        {children}

        End Mi Layout

    </>)

}

# /app/lib

Archivos js con utilidades

# /public/

Poner los archivos estáticos, como el /img/

# Link

Import Link from ‘next/link’

<link href=’/ruta/mivista’ > Algo </>

# Image

Import Image from ‘next/image’;

<Image src=”..jpg” width={600} height={600} alt=”descripcion” />

La carpeta public se considera la raíz asi que no hay que incluirla

# Layout

Crear un componente (en cualquier parte, pero preferiblemente en su carpeta components) que haga de esta función,

export default function Layout(

    {children}:{children:React.ReactNode}

)

{

    return (

        <>{children}</>

    )

}

Tener en cuenta que en la ruta que se declaren va a ser utilizado tambien en sus subrutas (creo?)

# defaultProps

Para poner parámetros por defecto tambien se puede

Componente.defaultProps={

    title:"el titulo"

}

# Estilos en modo clase

/styles/miestilos.css

.container{

    max-width: 36rem;

    font-size: large;

    color: blue;

}

En el componente

import styles from '../styles/Layout.module.css'

<div className={styles.container}>

En los estilos no se pueden usar etiquetas html, solo clases

Fuentes de Google

/app/ui/fonts.tsx

Cargar en un archivos las fuentes de Google que trae Next

import { Inter,Montserrat} from 'next/font/google';

export const moneserrat= Montserrat({subsets:['latin']})

Luego usarlas en otro componente

import {moneserrat} from './ui/fonts'

<h1

           className={`${moneserrat.className} antialiased`}

          >

          Hola</h1>

# Estilos globales

En un layout

import "./globals.css";

# getStaticProps

Supuestamente este metodo no se compila al frontend

export default function index({data}){

    return (

        <>

        Hello

        {

            data.map(({id,title,body}, index) => {

                return (

                    <div key={id}>

                        <h3>{id} - {title}</h3>

                        <p>{body}</p>

                    </div>)

        })

        }

        </>

    )

}

export async function getStaticProps() {

    try{

        const response = await fetch('https://jsonplaceholder.typicode.com/posts')

        const data = await response.json()

        return {

            props: { data },

            revalidate: 60 // 1 minute

            }

    }catch(error){

        console.error(error)

    }

}

# Rutas dinámicas

/pages/ruta/[parametro].jsx

Parecido a sveltekit

## params

Se puede acceder a los parámetros de la ruta mediante el atributo params que recibe el Page

export default function Page({params}:{

    params:{productId:string}

}) {

    return (<>

        <p>El id es: {params.productId}</p>

    </>)

}

# use client

para hacer se renderice en el cliente

'use client'

import { usePathname } from "next/navigation"

export default function Page(){

    const pathname=usePathname()

    return (<>

        <h2>Mi ruta2 {pathname}</h2>

    </>)

}

# Postgres

* pnpm install @vercel/postgres

# Seed

Suponiendo que hay un archivo /scripts/seed.js y tienes un .env

Agregar al package.json un script de arranque para esto

"scripts": {

    "dev": "next dev",

    "build": "next build",

    "start": "next start",

    "lint": "next lint",

    "seed": "node -r dotenv/config ./scripts/seed.js"

  },

Luego

* npm run seed

# async Componentes

Los componentes de react que se renderizan del lado del servidor pueden ser async, incluyendo los Page

export default async function Page(){

    return (<>Es asincorono</>)

}

## Loading

En lo que carga un componente se puede crear otro que se muestre mientras el usuario espera, para eso al mismo nivel del page.tsx se crea un loading.tsx

export default function Loading(){

    return (<>Cargando...</>)

}

# Suspense

Si se tiene un componente asíncrono que carga lento, se puede usar de react el Suspense para crear el efecto de cargando.., este suspense recibe un argumento fallback donde se le pasa el componente de cargando

Componente lento

export default async function ComponenteLento(){

    await new Promise((resolve)=> setTimeout(resolve,3000));

    return (<>Lento</>)

}

Se utiliza

import ComponenteLento from "@/app/lib/components/component1"

import { Suspense } from "react"

export default function Page(){

    return (

    <>

    Lento2

    <Suspense fallback={<>Espere...</>} >

        <ComponenteLento/>

    </Suspense>

    Lento2

    </>

)

}

## Key

El suspense solo carga una vez, pero si se necesita que se actualice cada ves que cambie algo específico se le agrega una key string dinámica

import { Suspense } from "react";

import Search from "../lib/components/search";

import ComponenteLento from "../lib/components/component1";

export default  function Page(

    {searchParams}

    :{

        searchParams?:{

            query?:string

            page?:string

        }

    }){

        const currentPage= Number(searchParams?.page)||1

        const query= searchParams?.query || ''

    return (

    <>Busqueda: <Search placeholder="termino"/>

        <p><span>Page:</span>{currentPage}</p>

        <p><span>Query:</span>{query}</p>

        <Suspense

        key={query+currentPage}

        fallback={<>Espere...</>} >

            <ComponenteLento/>

        </Suspense>

    </>)

}

# Skeleton

Crearlo en skeletonreact.com

# useRouter

solo se puede usar del lado del cliente

## redirigir

"use client";

import { useRouter } from "next/router"

export default function Page() {

    const router =useRouter();

    const handlerClick=()=>{

        router.push("/otraruta")

    }

    return (<>

        <button onClick={handlerClick}>Redirigir</button>

    </>)

}

# Estado en url

La idea es que si un usuario copia su url y se la lleva a otro navegador, vea lo mismo

## Search

En caso de tener un input de búsqueda, que afecta la url, podemos sincronizarlo con los Query params de la url

Componente search

"use client";

import { useSearchParams, usePathname, useRouter } from "next/navigation";

export default function Search({placeholder}:{placeholder:string}){

    const searchParams= useSearchParams()

    const pathname=usePathname()

    const {replace} =useRouter()

    const handleSearch =(term:string)=>{

        const params=new URLSearchParams(searchParams);

        if(term){

            params.set('query',term)

        }else{

            params.delete('query')

        }

        replace(`${pathname}?${params.toString()}`)

    }

    return (<>

        <label htmlFor="search">Buscar</label>

        <input

        style={{color:"red"}}

            onChange={(event)=>handleSearch(event.target.value)}

            defaultValue={searchParams.get('query')?.toString()}

        />

    </>)

}

searchParams

Los page reciben automáticamente los Query params, como un parámetro

export default  function Page(

    {searchParams}

    :{

        searchParams?:{

            query?:string

            page?:string

        }

    }){

    return (<>Busqueda: <Search placeholder="termino"/></>)

}

# Use-debounce

Para cuando tienes un input que va a hacer algun tipo de consulta cada ves que cambie algo y, la idea para no sobrecargar de consultas es hacer una consulta cada ciertos cambios

* pnpm install use-debounce

import {useDebounceCallback} from ‘use-debounce’

const handleSearch = useDebounceCallback(

función de búsqueda , 300//cantidad de milisegundos que espera a ver si termino de escribir

)

# Acciones

Se suelen poner en app/lib/actions.ts

La primera línea debe de ser 'use server', con lo que se marca el archivo para que todas las funciones que se exportan en este archivo son de servidor y por lo tanto no se ejecutan ni se envían al cliente

'use server'

export async function createInovece(formData:FormData) {

    console.log(formData)

}

Un componente con form

import { createInvoice } from "../actions";

export default function FormExapmple(){

    return (

        <div className="form" style={{color:'red'}}>

            <form action={createInvoice}>

                <label>

                    Name:

                </label>

                <input type="text" name="name" />

                <label>

                    Age:

                </label>

                <input type="number" name="age" />

                <button type="submit">Submit</button>

            </form>

        </div>

    )

}

# Zod

* pnpm install zod

'use server'

import {z} from 'zod'

const CreateInvoiceSchema= z.object({

    id:z.string(),

    invoice\_number:z.coerce.number(),

    customer\_name:z.coerce.string()

})

const CreateInvoiceFormSchema= CreateInvoiceSchema.omit({

    id: true

})

# revalidatePath

Manda a recargar una ruta

import { revalidatePath } from 'next/cache'

revalidatePath('/rutaform')

# redirect

import { redirect } from 'next/dist/server/api-utils'

redirect('/rutaform')

# not-found

Para sobrescribir la Plantilla del error 404 crear el archivo app/not-found.tsx, como esta a este nivel hereda el layout

export default async function NotFound(){

    return (<>No lo encuentro</>)

}

# Ocultar rutas

Al nombre de la carpeta se le agrega un “\_” delante del nombre “\_oculto”

# Agrupar rutas

Si el nombre de un carpeta se pone entre paréntesis, entonces esta carpeta como ruta es ignorada pero sus hijas no,

Ejemplo:

carpetas

/app/(agrupada)/ruta1

/app/(agrupada)/ruta2

/app/ruta3

urls

…./ruta1

…./ruta2

…./ruta3

# Metadata

En el layout principal app/layout.tsx o en cualquier otro se declaran los metadatas como una variable externa

export const metadata: Metadata = {

  title: "Create Next App",

  description: "Generated by create next app",

};

No se puede utilizar en componentes “use client”

## Metadatos

Para el CEO

Import Head from “next/head”;

<Head>

<title> Mi titulo</>

<link rel=“icon” ../>

<meta …/>

</Head>

## Dinámicos

Para agregárselos de forma dinámica por ruta, en su layout correspondiente hay que agregarle el metodo generateMetadata

import { Metadata } from "next"

type Props={

    params:{productId:string}

}

export const generateMetadata=({params}:Props):Metadata=>{

    return {

        title: `Producto ${params.productId}`

    }

}

export default function Page({params}:Props) {

    return (<>

        <p>El id es: {params.productId}</p>

    </>)

}

## Template

El template sirve para los títulos dinámicos, osea que si en una pagina ya hay un titulo definido entonces se utiliza el template superior y se sustituye el %s por el titulo de la pagina

Layout superior

export const metadata: Metadata = {

  title: {

    default:"Por defecto",

    absolute:"",

    template:"%s Resto del titulo"

  },

  description: "Generated by create next app",

};

Layout hijo

export const metadata: Metadata = {

  title: "Create Next App",

  description: "Generated by create next app",

};

El titulo resultante seria "Create Next App Resto del titulo ",

## Absolute

Para evitar que el tamplate haga efecto en un hijo hay que utilizar el absolute en el layout hijo

export const metadata: Metadata = {

  title: {

    absolute:"Este es todo el titulo",

  },

  description: "Generated by create next app",

};

El titulo de esta pagina seria :"Este es todo el titulo",

# Template

Si hay un estado en el layout, al cambiar entre sus paginas el estado se conserva

Para que esto no ocurra, osea que entre cada navegación el estado se reinicie, en ves de llamar al archivo layout.tsx seria template.tsx

# refresh

import { useRouter } from "next/navigation"

const router=useRouter()

router.refresh()

# @carpeta

Para paginas complejas donde se va a trabajar cada parte con sus propios archivos, pero dentro de la misma carpeta de la ruta, (osea si se usaran componentes estos podrían estar externos en otra carpeta como lib, pero en este caso se asume que almenos la estructura general se va a encontrar dentro de la misma carpeta )

/ruta/layout.tsx

export default function Layout(

    {children,chat,video}: {children:React.ReactNode,

        chat:React.ReactNode,

        video:React.ReactNode}

){

    return(<>

        Mi Layout Complejo

        {chat}

        -----------

        {children}

        ---------

        {video}

        Mi Layout Complejo

    </>)

}

[/ruta/@chat/page.tsx](mailto:/ruta/@chat/page.tsx)

export default function Page(){

    return (<><p>El chat2</p></>)

}

[/ruta/@video/page.tsx](mailto:/ruta/@video/page.tsx)

export default function Page(){

    return (<><p>333El video</p></>)

}

/ruta/page.tsx

export default function Page(){

    return (<>colmplejo</>)

}

Las @carpeta no son rutas, solo se cargan dentro del layout

# Api

Generalmente para ello se crea un carpeta /app/api/

Y en cada ruta carpeta un archivo /app/api/ruta/route.ts con una funcion con el nombre del tipo de metodo

export async function GET(){

    return new Response("hola mundo")

}

## Json

import { products } from "./data";

export async function GET() {

    return Response.json(products)

}

export function DELETE(){

    return NextResponse.json({

        message:"deleting single note"

        })

}

export function PUT(){

    return NextResponse.json({

        message:"updating single note"

        })

}

## Post

export async function POST(request:Request) {

    const product=await request.json();

    products.push(product);

    return Response.json(products,{

        status:201,

        headers:{

            "Content-Type":"application/json"

        }

    })

}

## error

export async function GET(){

    try {

        const notes =await prisma.note.findMany();

        return NextResponse.json(notes);

    } catch (error) {

        if(error instanceof Error){

            return NextResponse.json(error,{

                status:500

            });

        }

    }

}

# getServerSideProps

1. getServerSideProps() debe ser una función exportada.
2. getServerSideProps() solo puede ser usado en páginas (dentro de la carpeta 'pages'). No puede ser usado en componentes.
3. getServerSideProps() se debe exportar como una función independiente; no funcionará si se agrega getServerSideProps como una propiedad del componente de la página.

# Prisma

Es un ORM

* pnpm install prisma -D

crear configuraciones para las conexiones de forma automática

para postgres es por defecto

* npx prisma init

para sqlite

* npx prisma init --datasource-provider sqlite

Necesita

* pnpm install @prisma/client
* pnpm install tsx --save-dev

## .env

DATABASE\_URL="file:./dev.db"

## /prisma/schema.prisma

Datos sobre la conexión a la base de datos

generator client {

provider = "prisma-client-js"

}

datasource db {

provider = "sqlite"

url = env("DATABASE\_URL")

}

### Modelos

En este archivo /prisma/schema.prisma tambien se pueden declarar los modelos

Ejemplo

model Note{

id Int @id @default(autoincrement())

title String

content String?

createdAt DateTime @default(now())

updatedAt DateTime @updatedAt

}

Básicamente

Todo separado por 1 espacio y por saltos de línea

model NombreModelo{

nombreAtributo1 Tipo @modificador1 @modificador2(metodoArgumento())

nombreAtributo2Opcional TipoOpcional? @modificador

}

#### Tipos

Int

String

DateTime

##### Opcional ?

Si el campo puede ser nuleable se le agrega a continuación del tipo el “?”

model NombreModelo{

content String?

}

#### Modificadores

##### @id

para cuando es id

model NombreModelo{

id Int @id @default(autoincrement())

}

Suele ser acompañado de @default(autoincrement())

##### @default()

###### @default(now())

Suele ser usado para la creación de fechas de forma automática

model NombreModelo{

createdAt DateTime @default(now())

}

##### @updatedAt

Suele ser para la actualización de fecha cada ves que ocurra la creación y edición del modelo

model NombreModelo{

updatedAt DateTime @updatedAt

}

## Migraciones

La inicial y cada ves que haces un cambio

* npx prisma migrate dev --name init

Si clonaste el proyecto y vas a crear la bd y aplicar las migraciones existentes

* npx prisma migrate dev

si por algun motivo no se pudo instalar completo

* pnpm prisma generate

## studio

Supuestamente es como una administración de la base de datos, pero parece que necesita internet para funcionar

* pnpm prisma studio

Para ejecutarlo sin internet agrega al env

PRISMA\_ENGINES\_CHECKSUM\_IGNORE\_MISSING=1

## queris

### src/libs/prisma.ts

es necesario un archivo src/libs/prisma.ts donde se cree el objeto prisma global

import { PrismaClient } from "@prisma/client";

declare global {

    var prisma: PrismaClient | undefined;

}

export const prisma= global.prisma || new PrismaClient();

if (process.env.NODE !== "production"){global.prisma= prisma;}

donde se vaya a usar

import {prisma} from "@/libs/prisma"

los métodos de la api [GET,POST …] deben de ser async

### findMany

retorna todos

const notes =await prisma.note.findMany()

### create

export async function POST(request:Request){

    const {title,content} = await request.json();

    const newNote =await prisma.note.create({

        data:{

            title,

            content,

        },

    })

    return NextResponse.json(newNote);

### findFirst

interface Params{params:{id:string}};

export async function GET(request:Request,{params}:Params){

    try {

        const note =await prisma.note.findFirst({

            where:{id:Number(params.id)}

        });

        return NextResponse.json(note);

    } catch (error) {

        if(error instanceof Error){

            return NextResponse.json(error,{

                status:500

            });

        }

    }

}

### Delete

export async function DELETE(request:Request,{params}:Params){

    try{

        const deleteNote=await prisma.note.delete({

            where:{id:Number(params.id)}

        });

        if(!deleteNote) return NextResponse.json({

            message:"not found"

            },{

                status:404

            })

            return NextResponse.json(deleteNote);

    } catch (error) {

        if(error instanceof Prisma.PrismaClientKnownRequestError){

            if(error.code==="P2025"){

                return NextResponse.json({

                    message:"not found"

                    },{

                        status:404

                    })

            }

        }

        if(error instanceof Error){

            return NextResponse.json(error,{

                status:500

            });

        }

    }

    return NextResponse.json({

        message:"deleting single note"

        })

}

### Update

export async function PUT(request:Request,{params}:Params){

    try {

        const {title,content} = await request.json();

        const note =await prisma.note.update({

            where:{id:Number(params.id)},data:{

                title,

                content

            }

        });

        if(!note) return NextResponse.json({

            message:"not found"

            },{

                status:404

            })

        return NextResponse.json(note);

    } catch (error) {

        if(error instanceof Prisma.PrismaClientKnownRequestError){

            if(error.code==="P2025"){

                return NextResponse.json({

                    message:"not found"

                    },{

                        status:404

                    })

            }

        }

        if(error instanceof Error){

            return NextResponse.json(error,{

                status:500

            });

        }

    }

}

## Seed

Si se tiene se va a aplicar automáticamente al llamar al método de aplicar migraciones si no existe la bd

### package.json

Agregar al package.json

"prisma": {  
 "seed": "tsx prisma/seed.ts"  
 },

### prisma/seed.ts

Crearse un archivo prisma/seed.ts

Que va a tener una función asíncrona main y exportada donde se van a crear los datos

import prisma from "@/libs/prisma";

export async function main*()* {

  // Crear categorías

  const categories = await Promise.all([

    prisma.category.create({

      data: {

        name: "Electrodomésticos",

        image: "/assets/categories/img/electrodomesticos.png",

      },

    }),

    prisma.category.create({

      data: {

        name: "Farmacia",

        image: "/assets/categories/img/farmacia.png",

      },

    }),

    prisma.category.create({

      data: {

        name: "Bebidas",

        image: "/assets/categories/img/bebidas.png",

      },

    }),

  ]);

  // Crear compañías

  const companies = await Promise.all([

    prisma.company.create({

      data: {

        name: "TechCorp",

      },

    }),

    prisma.company.create({

      data: {

        name: "FashionStyle",

      },

    }),

    prisma.company.create({

      data: {

        name: "HomeDecor",

      },

    }),

  ]);

  // Crear productos

  const products = await Promise.all([

    // Productos de TechCorp

    prisma.product.create({

      data: {

        name: "Cerveza Premium",

        description: "Cerveza artesanal de alta calidad",

        price: 899.99,

        originalPrice: 999.99,

        stock: 50,

        image: "/assets/products/img/cerveza.png",

        discountPercentage: 10,

        freeShipping: true,

        categoryId: categories[2].id,

        companyId: companies[0].id,

      },

    }),

    prisma.product.create({

      data: {

        name: "Pasta Barbacue",

        description: "Pasta especial para barbacoa",

        price: 1299.99,

        originalPrice: 1499.99,

        stock: 30,

        image: "/assets/products/img/pasta\_barbicue.png",

        discountPercentage: 13.33,

        freeShipping: true,

        categoryId: categories[1].id,

        companyId: companies[0].id,

      },

    }),

    // Productos de FashionStyle

    prisma.product.create({

      data: {

        name: "Especias Refinadas",

        description: "Mezcla de especias gourmet",

        price: 29.99,

        originalPrice: 39.99,

        stock: 100,

        image: "/assets/products/img/especias\_refinidas.png",

        discountPercentage: 25,

        freeShipping: false,

        categoryId: categories[1].id,

        companyId: companies[1].id,

      },

    }),

    // Productos de HomeDecor

    prisma.product.create({

      data: {

        name: "Masa para Pizza",

        description: "Masa fresca para pizza artesanal",

        price: 49.99,

        originalPrice: 59.99,

        stock: 40,

        image: "/assets/products/img/molo\_para\_piza.png",

        discountPercentage: 16.67,

        freeShipping: true,

        categoryId: categories[1].id,

        companyId: companies[2].id,

      },

    }),

  ]);

  console.log("Seeding completado exitosamente!");

  console.log("Categorías creadas:", categories.length);

  console.log("Compañías creadas:", companies.length);

  console.log("Productos creados:", products.length);

}

main()

  .catch(*(e)* => {

    console.error("Error durante el seeding:", e);

    process.exit(1);

  })

  .finally(async *()* => {

    await prisma.$disconnect();

  });

### prisma db seed

Luego si no es que no ha sido aplicado ya la crear las migraciones, ejecutar

* npx prisma db seed

# Comandos

Para usar crear un comando personalizado, que pueda utilizar el orm del proyecto

En algún lugar del proyecto crear un archivo.ts y luego agregarlo como comando al package.json agregándole “tsx” delante de la dirección del archivo

Ejemplo: src/commands/generateAvailableLocations.ts

"scripts": {

    "dev": "next dev",

    "devt": "next dev --turbopack",

    "build": "next build",

    "start": "next start",

    "lint": "next lint",

    "generate:locations": "tsx src/commands/generateAvailableLocations.ts"

  },

# Mongosee

* pnpm install mongodb
* pnpm install mongoose

-1ro configurar el entorno, apuntar al servidor de la base de datos, no a la base de datos

.env

MONGODB\_URI="mongodb://localhost:27017 "

-2do configurar la conexón

import mongoose from "mongoose";

const MONGODB\_URI=process.env.MONGODB\_URI;

const connect = async ()=>{

    const connectionState=mongoose.connection.readyState;

    if(connectionState===1){

        console.log("already connected");

        return;

    }

    if(connectionState===2){

        console.log("Connecting...");

        return;

    }

    try{

        await mongoose.connect(MONGODB\_URI!,{

            dbName:'testusers',

            bufferCommands:true

        });

        console.log("connected")

    }catch(err){

        console.log('error')

        console.log(err)

        throw new Error("Error",err)

    }

}

export default connect;

-3ro crear un esquema de un modelo (lo crea automáticamente en la base de datos)

import { Schema,model,models } from "mongoose";

import { unique } from "next/dist/build/utils";

const UserSchema=new Schema({

    email:{type:"string",require:true,unique:true},

    username:{type:"string",require:true,unique:true},

    password:{type:"string",require:true},

},{

    timestamps:true

})

const User=models.User || model("User",UserSchema);

export default User;

# testing-library/react

* pnpm install --save-dev @testing-library/react @testing-library/dom @types/react @types/react-dom

## jest (Framework de testing js)

* pnpm install -D jest jest-environment-jsdom @testing-library/react @testing-library/dom @testing-library/jest-dom ts-node @types/jest
* pnpm install ts-jest

### eslint

* pnpm install eslint-plugin-jest-dom eslint-plugin-testing-library

#### eslint.config.mjs

agregar el plugin al archivo de configuración de eslint "plugin:testing-library/react","plugin:jest-dom/recommended" en la variable eslintConfig

import { dirname } from "path";

import { fileURLToPath } from "url";

import { FlatCompat } from "@eslint/eslintrc";

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = dirname(\_\_filename);

const compat = new FlatCompat({

  baseDirectory: \_\_dirname,

});

const eslintConfig = [

  ...compat.extends("next/core-web-vitals", "next/typescript","plugin:testing-library/react","plugin:jest-dom/recommended"),

];

export default eslintConfig;

### crear el Proyecto test?

* pnpm create jest@latest

### jest.config.ts

Archivo de configuracion general del test

Ponerle la siguiente configuración que es la recomendada (da igual el caso)

import type { Config } from "jest";

import nextJest from "next/jest.js";

const createJestConfig = nextJest({

  // Provide the path to your Next.js app to load next.config.js and .env files in your test environment

  dir: "./",

});

// Add any custom config to be passed to Jest

const config: Config = {

  coverageProvider: "v8",

  testEnvironment: "jsdom",

  // Add more setup options before each test is run

  setupFilesAfterEnv: ["<rootDir>/jest.setup.ts"],

  preset: "ts-jest",

};

// createJestConfig is exported this way to ensure that next/jest can load the Next.js config which is async

export default createJestConfig(config);

### jest.setup.ts

Crear este archivo al mismo nivel del anterior

import "@testing-library/jest-dom";

si, de momento solo tiene esta línea

### package.json

agregar los comandos

"test": "jest  --passWithNoTests",

"test:watch": "jest --watchAll"

Luego para ejecutar (y asi probar rápido si esta bien configurado)

* pnpm test

### \_\_tests\_\_

Los test los vas a poner en la carpeta src/\_\_tests\_\_

Dentro de subcarpetas, parece que es común imitar el contenido de subcarpetas presente en src y dentro de app

#### Ejemplo comprobar un texto

src/\_\_test\_\_/app/homepage.tsx

Se va a utilizar el objeto render para renderizar un componente

Y el objeto screen para buscar cosas en este componente

Luego el resultado de esto se va a analizar con expect (de test para comprobar)

import { render, screen } from "@testing-library/react";

import HomePage from "@/app/page";

it("El texto", () => {

  render(<HomePage />);

  expect(screen.getByText("El texto")).toBeInTheDocument();

});

### HelloWord.test.tsx

Otra forma de organizar los test es poniendo al lado de cada componente un archvio para sus test que se va a llamar NombreComponente.test.tsx, automáticamente jest rastrea y busca estos archivos

import { render, screen } from "@testing-library/react";

import HelloWord from "./HelloWord";

describe("HelloWord", () => {

  it("Hello Word", () => {

    render(<HelloWord />);

    expect(screen.getByText("Hello Word")).toBeInTheDocument();

  });

});

npx jest

Puedes llamar a un archivo test expecifico con

* npx jest NombreArchivo.tsx //este caso es dentro de \_\_test\_\_
* npx jest NombreArchivo.test.tsx //cuando están al lado de su componente

### getByText

Busca coincidencias exactas

screen.getByText("El texto")

para ignorar mayúsculas y minúsculas, supongo que la forma / / es por lo de expresiones regulares

screen.getByText(/otro texto/i)

### describe

Una manera para agrupar un conjunto de test que tienen que ver con algo es usando el método describe , no hay que importarlo, toma un atributo nombre y una función void donde pondremos nuestros test

import { render, screen } from "@testing-library/react";

import HomePage from "@/app/page";

describe("Home page", () => {

  it("El texto", () => {

    render(<HomePage />);

    expect(screen.getByText("El texto")).toBeInTheDocument();

  });

  it("OTRO TEXTO", () => {

    render(<HomePage />);

    expect(screen.getByText(/otro texto/i)).toBeInTheDocument();

  });

});

### beforeEach

Se puede usar para llamar a algo que tengan en común todo los test (lo e visto usado dentro de describe, no estoy segura que se pueda usar fuera de este), por ejemplo si todos renderizan el mismo componente, para que esta renderización se haga antes de cada test

Recibe una función void donde se llaman las cosas comunes

No es valido si usas slint porque es incorrecto renderizar algo dentro del before

import { render, screen } from "@testing-library/react";

import HomePage from "@/app/page";

describe("Home page", () => {

  beforeEach(() => {

    render(<HomePage />);

  });

  it("El texto", () => {

    expect(screen.getByText("El texto")).toBeInTheDocument();

  });

  it("OTRO TEXTO", () => {

    expect(screen.getByText(/otro texto/i)).toBeInTheDocument();

  });

});

### getByRole

Comprueba dentro del elementos de un tipo expecisifico de etiqueta, tener en cuenta que la manera de identificar los tipos de etiquetas es por su nombre real. Por ejemplo de los “h” seria “heading” para todos (sin importar el h#)

Esta función recibe el nombre de la etiqueta y un objeto con un campo “name” donde puedes poner el contenido del texto que estas buscando

expect(

      screen.getByRole("heading", {

        name: /Titulo/i,

      })

    ).toBeInTheDocument();

Explicación ia:

Cuando utilizas **getByRole** en React Testing Library (usado en tus tests de Next.js con TypeScript y Jest), los nombres de los roles no son directamente equivalentes a las etiquetas HTML , sino a los roles de accesibilidad (ARIA) que los elementos HTML asumen por defecto o explícitamente. Aquí tienes una lista de los equivalentes comunes:

#### Roles ARIA equivalentes a etiquetas HTML comunes

| HTML Tag | Role ARIA | Ejemplo de Uso |
| --- | --- | --- |
| **<h1>**a**<h6>** | **"heading"**(con**level**) | **screen.getByRole("heading", { level: 1, name: /Título/i })** |
| **<button>** | **"button"** | **screen.getByRole("button", { name: /Enviar/i })** |
| **<a>** | **"link"** | **screen.getByRole("link", { name: /Ir a inicio/i })** |
| **<input>** | **"textbox"** | **screen.getByRole("textbox", { name: /Nombre/i })** |
| **<input type="checkbox">** | **"checkbox"** | **screen.getByRole("checkbox", { name: /Aceptar términos/i })** |
| **<input type="radio">** | **"radio"** | **screen.getByRole("radio", { name: /Opción 1/i })** |
| **<select>** | **"combobox"** | **screen.getByRole("combobox", { name: /País/i })** |
| **<textarea>** | **"textbox"** | **screen.getByRole("textbox", { name: /Comentarios/i })** |
| **<nav>** | **"navigation"** | **screen.getByRole("navigation")** |
| **<main>** | **"main"** | **screen.getByRole("main")** |
| **<section>** | **"region"**(si tiene**aria-label**o**aria-labelledby**) | **screen.getByRole("region", { name: /Sección/i })** |
| **<table>** | **"table"** | **screen.getByRole("table", { name: /Datos/i })** |
| **<tr>** | **"row"** | **screen.getByRole("row", { name: /Fila 1/i })** |
| **<th>** | **"columnheader"**o**"rowheader"** | **screen.getByRole("columnheader", { name: /Nombre/i })** |
| **<img>** | **"img"**(si tiene**alt**o**aria-label**) | **screen.getByRole("img", { name: /Logo/i })** |
| **<ul>**,**<ol>** | **"list"** | **screen.getByRole("list")** |
| **<li>** | **"listitem"** | **screen.getByRole("listitem", { name: /Elemento 1/i })** |

#### Casos especiales

* Botones con iconos : Si un botón solo tiene un ícono (sin texto), usa **aria-label** y prueba con:

ts

1

screen.getByRole("button", { name: "Cerrar" });

* Inputs de tipo **search** : Sigue siendo **"textbox"** pero puedes usar **name** para identificarlo:

ts

1

screen.getByRole("textbox", { name: /Buscar/i });

#### Tips importantes

1. **name** es sensible a mayúsculas/minúsculas : Usa expresiones regulares (**/Texto/i**) para ignorar el caso.
2. No todos los elementos tienen un rol explícito : Algunos requieren el atributo **role** en el HTML (ej: **<div role="alert">**).
3. Niveles de encabezado (**level**) : Para **<h1>** a **<h6>**, usa la opción **level**:

ts

1

screen.getByRole("heading", { level: 2 }); // Busca <h2>

1. Documentación oficial : [ARIA Roles](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles)

#### Ejemplo completo

tsx

expect(screen.getByRole("heading", { level: 2, name: /Título/i })).toBeInTheDocument();

// Para <button type="submit">Enviar</button>

expect(screen.getByRole("button", { name: /Enviar/i })).toBeInTheDocument();

### toBe

Para probar una function expecifica

Utils.ts

export function sumar(a: number, b: number): number {

  return a + b;

}

Test

import { sumar } from "@/utils/utils";

describe("utiles", () => {

  it("sumar", () => {

    expect(sumar(2, 3)).toBe(5);

  });

});

### Not Tobe

it("sumar 9", () => {

    expect(sumar(2, 3)).not.toBe(9);

  });

### toEqual

utils

export function crearObjeto(llave: string, valor: string) {

  return {

    oldkey: "oldvalue",

    [llave]: valor,

  };

}

Test

it("crearObjeto", () => {

    const newKey = "newkey";

    const newValue = "newValue";

    expect(crearObjeto(newKey, newValue)).toEqual({

      oldkey: "oldvalue",

      [newKey]: newValue,

    });

  });

### toMatch

Expreciones regulares

it("OTR exprecion regular", () => {

    expect("oretasotrasd").toMatch(/otr/);

  });

### Asyc

It actepta que su método sea asyc para que puedas usar el aweit

export async function crearObjetoAsy(llave: string, valor: string) {

  return {

    oldkey: "oldvalue",

    [llave]: valor,

  };

}

it("crearObjeto asy", async () => {

    const newKey = "newkey";

    const newValue = "newValue";

    const response = await crearObjetoAsy(newKey, newValue);

    expect(response).toEqual({

      oldkey: "oldvalue",

      [newKey]: newValue,

    });

  });

### data-testid

Puedes maracar un lemento html directamente agregándole el atributo

data-testid=”idATestear” , luego se busca el elemento mediante getByTestId

<span data-testid="idtexto">El texto</span>

it("Titulo id test", () => {

    render(<HomePage />);

    expect(screen.getByTestId("idtexto")).toHaveTextContent(/texto/);

  });

### toBeDisabled

Para comprobar que un elemento esta en estado disabled

it("Es disabled default", () => {

    render(<HomePage />);

    expect(screen.getByRole("button")).toBeDisabled();

  });

### toBeEnabled

it("abilita el boton", () => {

    render(<HomePage />);

   const boton = screen.getByRole("button");

    expect(boton).toBeEnabled();

  });

### getByPlaceholderText

<input

          type="text"

          onChange={handlerChange}

          placeholder="escriba algo"

          className="w-full ring-2 ring-yellow-300"

        />

it("buscar por plaholder", () => {

    render(<HomePage />);

    const element = screen.getByPlaceholderText("escriba algo");

    expect(element).toBeInTheDocument();

  });

### fireEvent.change

Simular la escritura en un input text es llamando a

fireEvent.change(inputText, { target: { value: "new text" } });

Este fireEvent no hay que importarlo

ejemplo

it("escribir texto abilita el boton", () => {

    render(<HomePage />);

    const inputText = screen.getByPlaceholderText("escriba algo");

    fireEvent.change(inputText, { target: { value: "new text" } });

    const boton = screen.getByRole("button");

    expect(boton).toBeEnabled();

  });

### within

Luego de tener un elemento, si llega ha ser necesario hacerle consulta a este elemento porque es un contenedor, se debe de usar within, para que crear un elemento parecido a screen que tenga métodos de consultas

import { fireEvent, render, screen, within } from "@testing-library/react";

it("sin notas al principio", () => {

    render(<HomePage />);

    // Obtener el <ul> por su testId

    const notesContainer = screen.getByTestId("idtestnotes");

    // Usar 'within' para crear consultas dentro de este elemento

    const { queryAllByRole } = within(notesContainer);

    // Buscar todos los elementos de tipo "listitem" (li)

    const listItems = queryAllByRole("listitem");

    // Verificar que no haya elementos <li>

    expect(listItems).toHaveLength(0);

  });

### toHaveLength

Para contar la cantidad de elementos internos

it("sin notas al principio", () => {

    render(<HomePage />);

    // Obtener el <ul> por su testId

    const notesContainer = screen.getByTestId("idtestnotes");

    // Usar 'within' para crear consultas dentro de este elemento

    const { queryAllByRole } = within(notesContainer);

    // Buscar todos los elementos de tipo "listitem" (li)

    const listItems = queryAllByRole("listitem");

    // Verificar que no haya elementos <li>

    expect(listItems).toHaveLength(0);

  });

### fireEvent.click

Dispara el evento de click de un elemento

it("cantidad de notas", () => {

    render(<HomePage />);

    const inputText = screen.getByPlaceholderText("escriba algo");

    fireEvent.change(inputText, { target: { value: "new text" } });

    const boton = screen.getByRole("button");

    fireEvent.click(boton);

    // Obtener el <ul> por su testId

    const notesContainer = screen.getByTestId("idtestnotes");

    // Usar 'within' para crear consultas dentro de este elemento

    const { queryAllByRole } = within(notesContainer);

    // Buscar todos los elementos de tipo "listitem" (li)

    const listItems = queryAllByRole("listitem");

    // Verificar que haya elementos <li>

    expect(listItems).toHaveLength(1);

  });

### Mock

Jest proporciona el método **jest.fn()** para crear funciones simuladas (mock functions) , que son esenciales para aislar y probar comportamientos en tus tests. Estas funciones no solo ejecutan código, sino que también rastrean llamadas, argumentos y resultados . Además, el objeto **mock** asociado a cada función simulada ofrece métodos y propiedades clave para validar y controlar su comportamiento.

#### 📌 Principales métodos y propiedades del objeto mock

##### 1. mock.calls

* Propósito : Muestra todos los argumentos con los que se llamó la función.
* Uso :

const mock = jest.fn();

mock('hola', { id: 1 });

mock(42);

expect(mock.calls).toEqual([

['hola', { id: 1 }],

[42]

]);

##### 2. mock.results

* Propósito : Muestra los resultados de cada llamada (valores devueltos o errores).
* Uso :

const mock = jest.fn(() => 'hola');

mock();

mock.mockImplementation(() => 42);

mock();

expect(mock.results).toEqual([

{ type: 'return', value: 'hola' },

{ type: 'return', value: 42 }

]);

##### 3. mock.instances

* Propósito : Muestra las instancias creadas si la función fue usada como constructor (**new**).
* Uso :

const mock = jest.fn();

new mock();

new mock();

expect(mock.instances.length).toBe(2);

##### 4. mock.invocationCallOrder

* Propósito : Muestra el orden de ejecución de todas las llamadas (útil para validar secuencias).
* Uso :

const mock1 = jest.fn();

const mock2 = jest.fn();

mock1();

mock2();

mock1();

expect(mock1.invocationCallOrder).toEqual([1, 3]);

expect(mock2.invocationCallOrder).toEqual([2]);

#### 🧪 Métodos para configurar comportamiento

##### 5. mockImplementation(fn)

* Propósito : Define qué hace la función cuando se llama.
* Uso :

const mock = jest.fn().mockImplementation((a, b) => a + b);

expect(mock(2, 3)).toBe(5);

##### 6. mockImplementationOnce(fn)

* Propósito : Define una implementación temporal para la próxima llamada .
* Uso :

const mock = jest

.fn()

.mockImplementationOnce(() => 'primera vez')

.mockImplementationOnce(() => 'segunda vez');

expect(mock()).toBe('primera vez');

expect(mock()).toBe('segunda vez');

expect(mock()).toBe(undefined); // Sin más implementaciones

##### 7. mockReturnValue(value)

* Propósito : Hace que la función siempre devuelva un valor específico.
* Uso :

const mock = jest.fn().mockReturnValue('hola');

expect(mock()).toBe('hola');

expect(mock()).toBe('hola');

##### 8. mockReturnValueOnce(value)

* Propósito : Devuelve un valor solo para la próxima llamada .
* Uso :

const mock = jest

.fn()

.mockReturnValueOnce(10)

.mockReturnValueOnce(20);

expect(mock()).toBe(10);

expect(mock()).toBe(20);

expect(mock()).toBe(undefined);

##### 9. mockReturnThis()

* Propósito : Hace que la función devuelva la propia instancia (útil para encadenar métodos).
* Uso :

const mock = jest.fn().mockReturnThis();

expect(mock()).toBe(mock);

#### 🧹 Métodos para reiniciar o limpiar

##### 10. mockClear()

* Propósito : Limpia los datos de llamadas (pero mantiene la implementación ).
* Uso :

const mock = jest.fn(() => 'hola');

mock(1);

mock.mockClear();

expect(mock).not.toHaveBeenCalled(); // ✅

expect(mock()).toBe('hola'); // ✅ La implementación sigue

##### 11. mockReset()

* Propósito : Reinicia la función: limpia llamadas y restablece la implementación por defecto (vuelve a ser una función vacía).
* Uso :

const mock = jest.fn(() => 'hola');

mock(1);

mock.mockReset();

expect(mock).not.toHaveBeenCalled(); // ✅

expect(mock()).toBe(undefined); // ❌ La implementación se reinicia

##### 12. mockRestore()

* Propósito : Restaura una función original si fue reemplazada con **jest.spyOn()**.
* Uso :

const original = console.log;

const spy = jest.spyOn(console, 'log').mockImplementation(() => {});

console.log('hola'); // No imprime nada

spy.mockRestore();

console.log('hola'); // Vuelve a imprimir

#### 🧠 Usos comunes en tests

* Verificar llamadas :

expect(mock).toHaveBeenCalledWith('arg1', 'arg2');

expect(mock).toHaveBeenCalledTimes(2);

* Simular errores :

const mock = jest.fn().mockImplementation(() => {

throw new Error('Error simulado');

});

expect(() => mock()).toThrow('Error simulado');

* Validar secuencias :

const mock = jest.fn();

mock('primero');

mock('segundo');

expect(mock).toHaveBeenNthCalledWith(1, 'primero');

expect(mock).toHaveBeenNthCalledWith(2, 'segundo');

### (mok) Simular llamada a la api

Suponiendo que necesitamos una respuesta de api

{json:{data…},ok:boolean} // esta es la estructura de cualquier fecth

Hay que declarar un mok generar y reiniciarlo después de cada test, para crear un mok se utiliza el objeto global (que no hay que importarlo), y es global.objetoAMoquear

const originalFetch = global.fetch;

para limpiar luego de cada test

afterEach(() => {

    global.fetch = originalFetch;

  });

Y en el test para simular la llamada a la api sustituyéndola por el mok, tiene que ser antes de renderizar el componente (se desactiva el slint, porque oviamente con el mok no vamos a simular completo la respuesta de una api)

// @ts-expect-error is a mok

    global.fetch = jest.fn(() =>

      Promise.resolve({

        json: () => Promise.resolve(fakeUsers),

        ok: true,

      })

    );

Luego de renderizar el componente, generalmente los llamados a la api tardan algo, asi que se trata de buscar una condición que solo sea cierta luego de que se resuelva la llamada a la api

await waitFor(() => {

      const element = screen.getByText("Nombre");

      expect(element).toBeInTheDocument();

    });

El test se detiene y espere a que esto sea cierto (no me queda claro si esperara indefinidamente, asi que creo que lo mejor es atar esta condición a alguna que salte luego de que se resuelva la llamada, ya sea positiva o negativa)

Luego se declara la condición a esperar si es positiva

expect(global.fetch).toHaveBeenCalledWith("/api/users");

#### Ejemplo completo

Componente

"use client"; // Necesario porque usamos fetch en el cliente

import { useEffect, useState } from "react";

interface User {

  id: number;

  name: string;

  email: string;

}

export default function UserList() {

  const [users, setUsers] = useState<User[]>([]);

  const [loading, setLoading] = useState(true);

  const [error, setError] = useState<string | null>(null);

  useEffect(() => {

    const fetchUsers = async () => {

      try {

        const res = await fetch("/api/users");

        if (!res.ok) throw new Error("Error al cargar los usuarios");

        const data = await res.json();

        setUsers(data);

      } catch {

        setError("No se pudieron cargar los usuarios.");

      } finally {

        setLoading(false);

      }

    };

    fetchUsers();

  }, []);

  if (loading) return <p className="text-blue-500">Cargando usuarios...</p>;

  if (error) return <p className="text-red-500">{error}</p>;

  return (

    <div className="overflow-x-auto">

      <table className="min-w-full  shadow-md rounded-lg overflow-hidden">

        <thead className=" text-red-500">

          <tr>

            <th className="py-3 px-4 text-left">ID</th>

            <th className="py-3 px-4 text-left">Nombre</th>

            <th className="py-3 px-4 text-left">Email</th>

          </tr>

        </thead>

        <tbody className="divide-y divide-gray-200">

          {users.map((user) => (

            <tr key={user.id} className="hover:bg-amber-700 transition">

              <td className="py-3 px-4">{user.id}</td>

              <td className="py-3 px-4 font-medium">{user.name}</td>

              <td className="py-3 px-4 ">{user.email}</td>

            </tr>

          ))}

        </tbody>

      </table>

    </div>

  );

}

Test:

import { render, screen, waitFor } from "@testing-library/react";

import UserPage from "@/app/users/page";

const originalFetch = global.fetch;

describe("Home page", () => {

  afterEach(() => {

    global.fetch = originalFetch;

  });

  const setup = () => {

    render(<UserPage />);

  };

  it("probar APi", async () => {

    const fakeUsers = [

      { id: 1, name: "Alice Johnson", email: "alice@example.com" },

      { id: 2, name: "Bob Smith", email: "bob@example.com" },

      { id: 3, name: "Charlie Brown", email: "charlie@example.com" },

      { id: 4, name: "Dana White", email: "dana@example.com" },

      { id: 5, name: "Eve Adams", email: "eve@example.com" },

      { id: 6, name: "Frank Wilson", email: "frank@example.com" },

    ];

    // @ts-expect-error is a mok

    global.fetch = jest.fn(() =>

      Promise.resolve({

        json: () => Promise.resolve(fakeUsers),

        ok: true,

      })

    );

    setup();

    await waitFor(() => {

      const element = screen.getByText("Nombre");

      expect(element).toBeInTheDocument();

    });

    expect(global.fetch).toHaveBeenCalledWith("/api/users");

  });

});

### (mok) Simular llamada router

Si queremos testear que en algún punto se llama a un router.push("other"), en el test hay importar también el

import { useRouter } from "next/navigation";

Con el objetivo de mokearlo (afuera del test)

jest.mock("next/navigation", () => ({ useRouter: jest.fn() }));

luego dentro del test

const mockPush = jest.fn();

    // @ts-expect-error is a mok

    useRouter.mockReturnValue({

      push: mockPush,

    });

Y suponiendo que se llama producto de la interacción con un botón

const boton = screen.getByRole("button");

    fireEvent.click(boton);

    expect(mockPush).toHaveBeenCalledWith("other");

#### Ejemplo completo

Pagina de router

"use client";

import { useRouter } from "next/navigation";

import React from "react";

const Page = () => {

  const router = useRouter();

  return (

    <div>

      <h1>Redirect</h1>

      <button

        className="p-2 bg-amber-400 text-black border-2 border-red-500"

        onClick={() => router.push("other")}

      >

        Ir

      </button>

    </div>

  );

};

export default Page;

test

import { render, screen, fireEvent, waitFor } from "@testing-library/react";

import RedirectPage from "@/app/redirect/page";

import { useRouter } from "next/navigation";

jest.mock("next/navigation", () => ({ useRouter: jest.fn() }));

describe("Redirect page", () => {

  const setup = () => {

    render(<RedirectPage />);

  };

  it("probar redirect", async () => {

    const mockPush = jest.fn();

    // @ts-expect-error is a mok

    useRouter.mockReturnValue({

      push: mockPush,

    });

    setup();

    const boton = screen.getByRole("button");

    fireEvent.click(boton);

    expect(mockPush).toHaveBeenCalledWith("other");

  });

});

#### spyOn

Esta forma es mas pesada (hace lo mismo que la anterior)

import { render, screen, fireEvent, waitFor } from "@testing-library/react";

import RedirectPage from "@/app/redirect/page";

import NextNavigation from "next/navigation";

jest.mock("next/navigation", () => ({ useRouter: jest.fn() }));

describe("Redirect page", () => {

  const setup = () => {

    render(<RedirectPage />);

  };

  it("probar redirect 2", async () => {

    const mockPush = jest.fn();

    const spyOnRouteMock = jest.spyOn(NextNavigation, "useRouter");

    // @ts-expect-error is a mok

    const useRouteMock = spyOnRouteMock.mockReturnValue({ push: mockPush });

    setup();

    const boton = screen.getByRole("button");

    fireEvent.click(boton);

    expect(mockPush).toHaveBeenCalledWith("other");

    useRouteMock.mockRestore();

  });

});

### act

Cuando estas testeando un componente muy grande como una pagina entera puede que necesites que primero se cargen todos los efectos y el store se inicie, entonces tienes que envolver al render en el método asíncrono act y esperar por el. A lint no le gusta porque considera que debes de probar bien todo de manera individual sin llegar a esto, pero si hay que utilizarlo entonces hay que desactivar al linter en esta zona.

/\* eslint-disable testing-library/no-unnecessary-act \*/

    await act(async () => {

      render(

        <ProviderRootLayout>

          <HomePage />

        </ProviderRootLayout>

      );

    });

    /\* eslint-enable testing-library/no-unnecessary-act \*/

Explicación ia :

el uso de **act()** es necesario para garantizar que todas las actualizaciones de estado y efectos de React se hayan completado antes de realizar las aserciones. Esto asegura que el componente esté en un estado estable y predecible, lo cual es especialmente importante cuando hay actualizaciones asíncronas o efectos secundarios (como **useEffect**, **useState**, o cambios derivados del contexto)

##### ¿Por qué tuviste que usar act()?

Cuando renderizas un componente completo (como **<HomePage />** dentro de un **<ProviderRootLayout>**), es posible que:

1. El componente dependa de proveedores de contexto (como **SessionProvider** o **ReduxProvider**) que inicializan estados asíncronos.
2. Algunos componentes internos realicen llamadas a **useState**, **useEffect** o actualizaciones de estado derivadas de props o contexto.
3. El árbol de componentes tenga efectos colaterales que no se resuelven inmediatamente al renderizar.

Sin **act()**, Jest podría realizar aserciones antes de que estas actualizaciones se hayan aplicado, lo que lleva a pruebas frágiles o falsas negativas. Al envolver **render()** en **act()**, aseguras que todo el árbol de React se actualice completamente antes de probarlo

##### ¿Por qué el linter te salta?

El linter probablemente está aplicando reglas estrictas de React Testing Library o Jest , como:

* **testing-library/no-unnecessary-act** : Si el código no requiere **act()** explícitamente (por ejemplo, si usas métodos de **screen** que ya lo manejan internamente).
* **jest/require-top-level-describe** u otras reglas de estilo.

En tu caso, el uso de **act()** es válido porque estás renderizando un componente que depende de proveedores de contexto o efectos asíncronos. Sin embargo, el linter no siempre puede detectar este contexto y genera una advertencia.

##### Alternativas más limpias

Si el linter te obliga a evitar **act()** explícito, considera estas opciones:

###### 1. Usa screen.findBy\* en lugar de screen.getBy\*

Los métodos **findBy\*** (como **screen.findByText**) son asíncronos y ya envuelven la lógica en **act()** internamente , esperando a que el DOM se actualice antes de buscar elementos.

Ejemplo:

// En lugar de:

const element = screen.getByText("Loading...");

// Usa:

const element = await screen.findByText("Loading...");

###### 2. Mockea proveedores de contexto

Si el problema viene de **ProviderRootLayout** (como un contexto que requiere estado asíncrono), puedes mockearlo para evitar renderizar el árbol completo.

Por ejemplo:

jest.mock("@/components/ProviderRootLayout", () => ({

ProviderRootLayout: ({ children }: { children: React.ReactNode }) => (

<MockedContextProvider>{children}</MockedContextProvider>

),

}));

Esto elimina la necesidad de **act()** al simplificar el entorno de prueba

### Next Image src

Para testear una imagen (su src) el componente de next es un poco complicado asi que lo que funciona es:

La imagen:

<Image

              src={img}

              alt={img}

              width={160}

              height={160}

              className="absolute top-1/2 left-1/2 transform -translate-x-1/2 -translate-y-1/2 w-40"

            />

Parámetros de la imagen:

const img = "/test-image.jpg";

Obtener la imagen:

const image = screen.getByAltText(img);

expect(image).toBeInTheDocument();

testear su src

const attribute = image.getAttribute("src");

expect(attribute).toContain(encodeURIComponent(img));

Store

Tener en cuenta que el store se declara como constante en el código y que por eso suele ser el mismo para todo el test (lo que pasa con la base de datos tambien)

Si tienes un componente que depende de un store y te gustaría que el store fuera lo mas parecido posible al orignal pero se resetease en cada test, la unica forma que encontré para eso es que siempre se cree uno nuevo por test y por tanto no puedas usar el Provider original de tu código, sino uno personalizado que va a crear un nuevo store por test (porque te repito el original solo usa uno fijo, ósea el provider si se recrea pero llama a un store que ya existe, no lo crea el, ese es el comportamiento correcto de los Provider, por eso tenemos que crear uno que no tenga este comportamiento que se seria correcto en el código original, pero que para nuestro test seria necesario la recreación )

Para esto si hay que modificar probablemente el código original del store, porque normalmente para crear el store se llama a configureStore, como ves es una función,

export const store = configureStore( ….);

Entonces la idea va a ser crear un función exportable que retorne el resultado de la llamada a configureStore

export const createStore = () => {

  return configureStore({

    reducer: persistedReducer,

    middleware: (getDefaultMiddleware) =>

      getDefaultMiddleware({

        serializableCheck: {

          ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],

        },

      }),

  });

};

export const store = createStore();

Y es esta función la que luego usaremos en nuestro test, asi será un store con la estructura del original el que estaremos usando

En nuestro test creamos el Provider personalizado

const AllProviders = ({ children }: { children: React.ReactNode }) => (

  <Provider store={createStore()}>{children}</Provider>

);

Y cada ves que usemos render va a utilizar un wrapper

render(<ProductCard product={productCopy} />, {

      wrapper: AllProviders,

    });

### Test api

Para probar la api, se importa directamente el metodo que esta dentro del route.ts (GET) y se llama pasandole directamente los parámetros. Y para evitar los errores con next hay que decirle que utilice node? Poniendo al principio del archivo de test

/\*\*

 \* @jest-environment node

 \*/

El test

it("should return the product from the real database", async () => {

    const response = await GET(

      {} as any,

      { params: { id: String(productId) } }

    );

    const json = await response.json();

    expect(json.status).toBe("success");

    expect(json.data).toHaveProperty("id", productId);

    expect(json.data).toHaveProperty("name", "ProductoTest");

    expect(json.data).toHaveProperty("category");

    expect(json.data).toHaveProperty("company");

  });