**Prompt para la IA: Estructura de Tests con Jest en Next.js**

Hola, estoy trabajando en un proyecto de tienda online usando Next.js con TypeScript, Prisma para la conexión a una base de datos PostgreSQL y Redux Toolkit para la gestión del estado. Quiero implementar tests con Jest siguiendo la estrategia de "Testear con una Base de Datos de Prueba Real" para mis API Routes y otros componentes que interactúan con la base de datos. A continuación, detallo mi configuración y preferencias para que generes tests sólidos y bien estructurados.

**Detalles de mi Configuración y Preferencias**

* **Framework:** Next.js (App Router)
* **Lenguaje:** TypeScript
* **Base de Datos:** PostgreSQL, gestionada con Prisma
* **Gestión del Estado:** Redux Toolkit
* **Testing:** Jest con @testing-library/react para componentes
* **Estrategia de Testing para API Routes:** Usar una base de datos de prueba real (PostgreSQL)
* **Linter:** Uso un linter (probablemente ESLint con reglas de TypeScript). Los tests deben estar bien escritos para pasar las reglas de linting. Si es inevitable violar alguna regla, puedes usar comentarios como // eslint-disable-next-line o /\* eslint-disable \*/ en líneas o bloques específicos, pero evita esto en lo posible.

Quiero que me ayudes a crear tests para mis API Routes, componentes, slices de Redux y otras partes de mi aplicación, siguiendo las mejores prácticas y asegurando que la estructura sea sólida, mantenible y legible.

**Estructura de Archivos**

* Los tests deben colocarse en un directorio \_\_tests\_\_ junto al archivo que están probando. Ejemplos:
  + Para app/api/products/route.ts, el test debe estar en app/api/products/\_\_tests\_\_/route.test.ts
  + Para components/ui/Button.tsx, el test debe estar en components/ui/Button/\_\_tests\_\_/Button.test.tsx
  + Para store/features/cart/cartSlice.ts, el test debe estar en store/features/cart/\_\_tests\_\_/cartSlice.test.ts
* Usa la convención de nombrar los archivos de test como [nombreDelArchivo].test.ts o [nombreDelArchivo].test.tsx según corresponda.

**Configuración de Jest**

* Uso ts-jest como transformador para TypeScript.
* Tengo un archivo jest.config.js configurado con ts-jest y ajustes necesarios para Next.js y TypeScript.
* Uso un archivo jest.setup.ts para configuraciones globales, como importar @testing-library/jest-dom y configurar mocks globales si es necesario.

**Base de Datos de Prueba**

* Uso una base de datos PostgreSQL separada para los tests, configurada mediante una variable de entorno DATABASE\_URL en un archivo .env.test.
* Antes de correr los tests, aplico las migraciones a la base de datos de prueba con prisma migrate deploy o prisma migrate reset --force (siendo consciente de que esto es destructivo y solo debe usarse en entornos de prueba).
* Antes de cada test, limpio las tablas relevantes (por ejemplo, con prisma.model.deleteMany({})) para asegurar un estado inicial limpio y consistente.

**Buenas Prácticas a Seguir**

**1. Tests Unitarios**

* **Componentes:** Usa @testing-library/react y enfócate en el comportamiento desde la perspectiva del usuario (qué ve y cómo interactúa el usuario).
* **Funciones y Utilidades:** Prueba diferentes casos de entrada y salida esperada.
* **Slices de Redux:** Prueba reducers y action creators (incluyendo thunks), verificando los cambios en el estado.

**2. Tests de Integración**

* **API Routes:** Usa la base de datos de prueba real para verificar la interacción con Prisma, probando respuestas exitosas (200, 201), errores de cliente (400, 404) y errores de servidor (500).
* **Flujos de UI:** Prueba cómo interactúan varios componentes juntos en escenarios realistas.

**3. Mocking**

* Mockea dependencias externas (como APIs o servicios) con jest.mock o jest.fn, pero para las API Routes, interactúa directamente con la base de datos real de prueba.
* Usa mocks solo cuando sea necesario para aislar unidades específicas fuera del alcance de la base de datos.

**4. Cobertura de Código**

* No busco el 100% de cobertura, pero usa --coverage para identificar áreas no testeadas y priorizar la lógica crítica.

**5. Nomenclatura Descriptiva**

* Usa bloques describe e it con nombres claros que expliquen qué se está probando y el resultado esperado. Ejemplo: it('should create a product and return it with status 201', ...).
* Estructura los tests en bloques lógicos anidados si es necesario (por ejemplo, un describe por método HTTP).

**6. Limpieza y Legibilidad**

* Aplica principios de código limpio: evita duplicación, usa variables descriptivas y mantén los tests concisos.
* Usa comentarios para deshabilitar reglas de linting solo si es estrictamente necesario (por ejemplo, // eslint-disable-next-line no-undef).

**7. Gestión del Estado de la Base de Datos**

* Usa beforeAll, beforeEach, afterAll y afterEach para gestionar la base de datos:
  + beforeAll: Aplica migraciones o configura la base de datos de prueba.
  + beforeEach: Limpia las tablas relevantes para cada test.
  + afterAll: Desconecta Prisma y realiza cualquier limpieza final.
* Asegúrate de que los tests sean independientes y no dependan del orden de ejecución.

**Ejemplo de Test para API Route**

Por favor, genera un ejemplo de test para una API Route que crea un producto (como app/api/products/route.ts), siguiendo estas buenas prácticas:

* Usa TypeScript y la base de datos de prueba real con Prisma.
* Prueba casos de éxito (creación con datos válidos) y error (datos inválidos).
* Asegúrate de que el código pase el linter y sea limpio.

**Otros Detalles**

* **Independencia:** Cada test debe ser autónomo y no depender de datos generados por otros tests.
* **Componentes:** Usa render y queries accesibles de @testing-library/react (como getByRole, getByText).
* **Redux:** Prueba reducers y thunks despachando acciones y verificando el estado resultante.
* **Errores:** Incluye pruebas para manejar errores esperados y excepciones.

Por favor, basa los tests en esta configuración y preferencias, generando código que sea robusto, mantenible y cumpla con las expectativas de calidad de mi proyecto.