

Testing - ZFCocteles

Grupo: ZFC Fans

Integrantes:




Cristian Leonardo Castañeda Olarte




Alan Ryan Cheyne Gómez





Juan Jerónimo Gómez Rubiano

David Santiago Velásquez Gómez

1. Pruebas Unitarias

En este proyecto se ha adoptado Vitest como framework principal de testing , aprovechando su compatibilidad nativa con módulos ES y su API inspirada en Jest. Vitest permite definir suites de prueba con **describe**, especificar casos con **it** y configurar hooks como **beforeEach** para preparar el entorno antes de cada test . Además, Vitest ofrece funciones integradas de mocking y spying (**vi.fn**, **vi.mock**), lo que facilita sustituir dependencias externas —como la capa de acceso a la base de datos— por implementaciones controladas y predecibles durante las pruebas .

Para aislar la lógica de los repositorios y evitar la necesidad de una base de datos real, se emplea un mock de base de datos  definido en **mockData.js**. Este mock crea un objeto con métodos **exec**, **close** y **prepare** implementados como spies (**vi.fn**), y genera distintos statements (con **.run**, **.get**, **.all**) que devuelven datos preconfigurados según la consulta realizada  **mockData**. Asimismo, se utilizan **vi.mock** para interceptar tanto el módulo de configuración de la base de datos (**config/database.js**) como el propio driver **better-sqlite3**, de modo que todas las instancias de conexión devuelvan el mismo mock sin interactuar con disco  **mockData**.


 Estas herramientas resultan especialmente adecuadas para nuestro proyecto, porque permiten simular escenarios de consulta y manipulación de datos sin el coste y la fragilidad de una base de datos real en memoria o en disco . La flexibilidad de Vitest facilita, por ejemplo, verificar que las consultas SQL correctas se pasan al método **prepare** y que los parámetros se envían mediante **stmt.get** o **stmt.all**, todo ello sin alterar la implementación de los repositorios de la base de datos. . De este modo, se asegura que los métodos de obtención de dificultad y la suma de duraciones funcionan según lo esperado, manteniendo el código de producción limpio y sin cambios específicos para las pruebas .

Para ejecutar los tests descritos a continuación, se debe acceder desde la consola a la carpeta fuente del proyecto (**/Proyecto/electron-app**) y aplicar el comando **npx vitest run path_del_archivo/nombre.test.js**

Alan


Prueba 1:

El test `CategoryRepository.test.js` se encarga de verificar la correcta gestión de categorías en el sistema. Evalúa operaciones como la búsqueda de categorías por nombre, la obtención de categorías del sistema, la creación de nuevas categorías, y la validación de categorías existentes. Así, asegura que la funcionalidad de categorización funcione correctamente y mantenga la integridad de los datos.

Resultados obtenidos:  16 tests ejecutados exitosamente (32 pruebas en total considerando duplicados)


Prueba 2:

El test `FavoriteRepository.test.js` verifica que el sistema de favoritos funcione correctamente. Evalúa la capacidad de obtener la categoría especial "Favoritos", agregar y remover cócteles de favoritos, verificar si un cóctel está en favoritos, y manejar la gestión de favoritos por usuario. Además, comprueba que se mantenga la integridad referencial entre usuarios, cócteles y la categoría de favoritos.

Resultados obtenidos:  23 tests ejecutados exitosamente (46 pruebas en total considerando duplicados)

Prueba 3:

El test `UserRepository.test.js` se enfoca en validar la gestión de usuarios del sistema. Verifica operaciones como la búsqueda de usuarios por email, la creación de nuevos usuarios con validación de campos obligatorios, la verificación de existencia de usuarios, y el manejo de errores en operaciones de base de datos. Garantiza que el sistema de autenticación y gestión de usuarios sea robusto y seguro.

Resultados obtenidos:  11 tests ejecutados exitosamente (22 pruebas en total considerando duplicados)

David

Prueba 4:

El test `CocktailCreate.test.js` verifica que el proceso de creación de cócteles maneje correctamente varios casos límite. Comprueba que no se permitan ingredientes duplicados, que el nombre del cóctel no esté vacío ni duplicado, y que la imagen adjunta tenga un formato válido. Así, garantiza que los datos de los cócteles sean consistentes y cumplan las reglas básicas antes de ser almacenado

```
describe('CocktailRepository.create - edge cases', () => {
  test('should not allow duplicate ingredient in cocktail', () => {
    const cocktailData = {
      name: 'Test Cocktail',
      ingredients: [
        { id_ingredient: 1, quantity: 50 },
        { id_ingredient: 1, quantity: 30 }, // Duplicado
      ],
    };
    // Simular validación en el método create
    function createWithValidation(data) {
      const ids = data.ingredients.map(i => i.id_ingredient);
      const hasDuplicates = ids.length !== new Set(ids).size;
      if (hasDuplicates) {
        throw new Error('Duplicate ingredient');
      }
    }
  });
});
```

Test Files 2 passed (2)
Tests 8 passed (8)
Start at 20:07:16
Duration 1.63s (transform 406ms, setup 0ms, collect 251ms, tests 31ms, environment 5ms, prepare 1.20s)

Prueba 5:

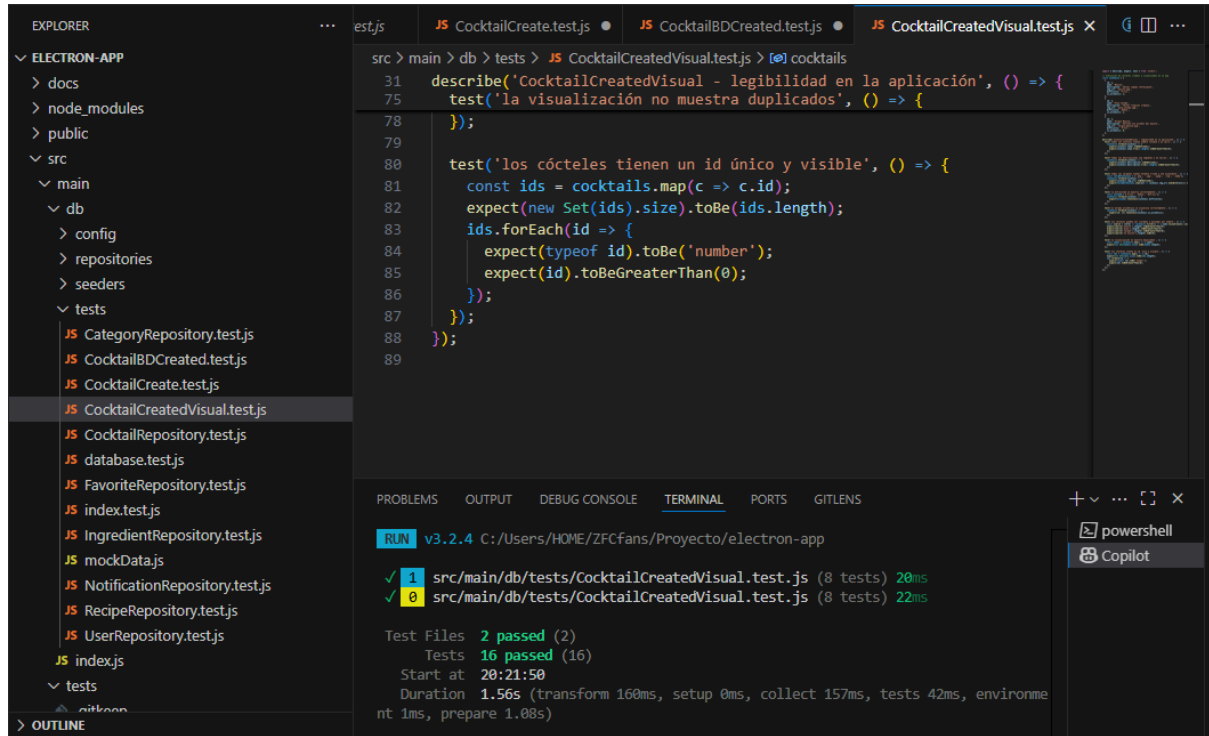
El test `CocktailBDCreated.test.js` se encarga de verificar que los cócteles creados sean correctamente almacenados en la base de datos. Evalúa distintos escenarios como el guardado exitoso, la prevención de duplicados, el manejo de campos obligatorios, la validación de formatos de imagen y dificultad, y la correcta asignación de propiedades como el estado alcohólico y el tipo de vaso. Así, asegura la integridad y consistencia de los datos en el sistema de almacenamiento.

```
describe('CocktailBDCreated - almacenamiento en base de datos', () => {
  test('debería almacenar cócteles con y sin alcohol', () => {
    const alcoholic = { name: 'Whisky Sour', is_alcoholic: 1 };
    const nonAlcoholic = { name: 'Virgin Mojito', is_alcoholic: 0 };
    expect(repository.create(alcoholic)).toBeGreaterThan(0);
    expect(repository.create(nonAlcoholic)).toBeGreaterThan(0);
  });
  test('debería almacenar cócteles con diferentes tipos de vaso', () => {
    const glasses = ['highball', 'rocks', 'martini', 'coupe', 'hurricane'];
    glasses.forEach(glass_type => {
      const cocktail = { name: `Cóctel ${glass_type}`, glass_type };
      expect(repository.create(cocktail)).toBeGreaterThan(0);
    });
  });
});
```

Test Files 2 passed (2)
Tests 16 passed (16)
Start at 20:17:48
Duration 1.68s (transform 275ms, setup 0ms, collect 464ms, tests 44ms, environment 1ms, prepare 920ms)

Prueba 6:

El test `CocktailCreatedVisual.test.js` verifica que los cócteles creados sean fácilmente legibles y visualizables en la aplicación. Evalúa que cada cóctel tenga nombre y descripción visibles, imagen en formato válido, dificultad y estado alcohólico correctos, y que no existan duplicados. Además, comprueba que los cócteles puedan ser buscados por nombre y que cada uno tenga un ID único y visible, asegurando una experiencia clara y ordenada para el usuario.



The screenshot shows the Visual Studio Code interface. On the left, the Explorer pane shows the project structure with the file `CocktailCreatedVisual.test.js` selected under the `tests` directory. The main editor displays the content of this file, which includes a `describe` block for 'CocktailCreatedVisual - legibilidad en la aplicación' and a `test` block for 'los cócteles tienen un id único y visible'. The test block contains logic to map cocktails to their IDs, check for uniqueness using a `Set`, and verify the ID type and value. At the bottom, the TERMINAL pane shows the command `npm test` being run, with output indicating that 2 test files passed (2) and 16 tests passed (16). The duration of the test run is 1.56s.

```
src > main > db > tests > JS CocktailCreatedVisual.test.js > cocktails
31 describe('CocktailCreatedVisual - legibilidad en la aplicación', () => {
75   test('la visualización no muestra duplicados', () => {
78   });
79
80   test('los cócteles tienen un id único y visible', () => {
81     const ids = cocktails.map(c => c.id);
82     expect(new Set(ids).size).toBe(ids.length);
83     ids.forEach(id => {
84       expect(typeof id).toBe('number');
85       expect(id).toBeGreaterThan(0);
86     });
87   });
88 });
89
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

RUN v3.2.4 C:/Users/HOME/ZFCfans/Proyecto/electron-app

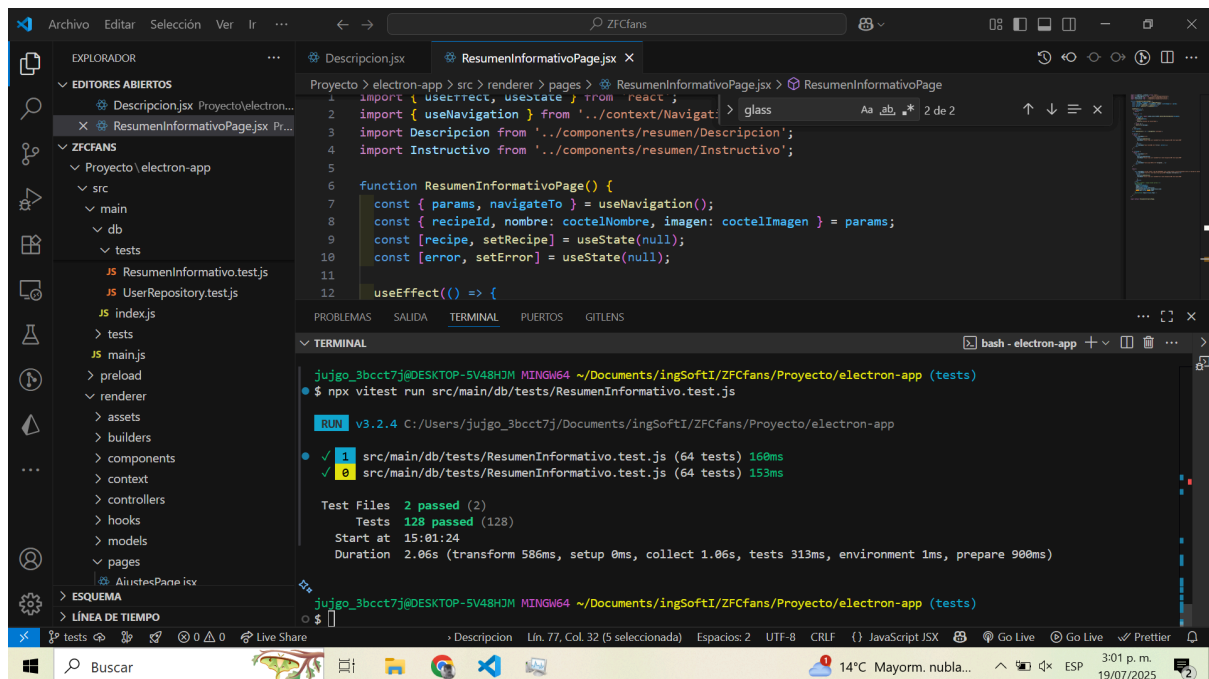
✓ 1 src/main/db/tests/CocktailCreatedVisual.test.js (8 tests) 20ms
✓ 0 src/main/db/tests/CocktailCreatedVisual.test.js (8 tests) 22ms

Test Files 2 passed (2)
Tests 16 passed (16)
Start at 20:21:50
Duration 1.56s (transform 160ms, setup 0ms, collect 157ms, tests 42ms, environment 1ms, prepare 1.08s)

Jerónimo

Prueba 7: (Jerónimo)

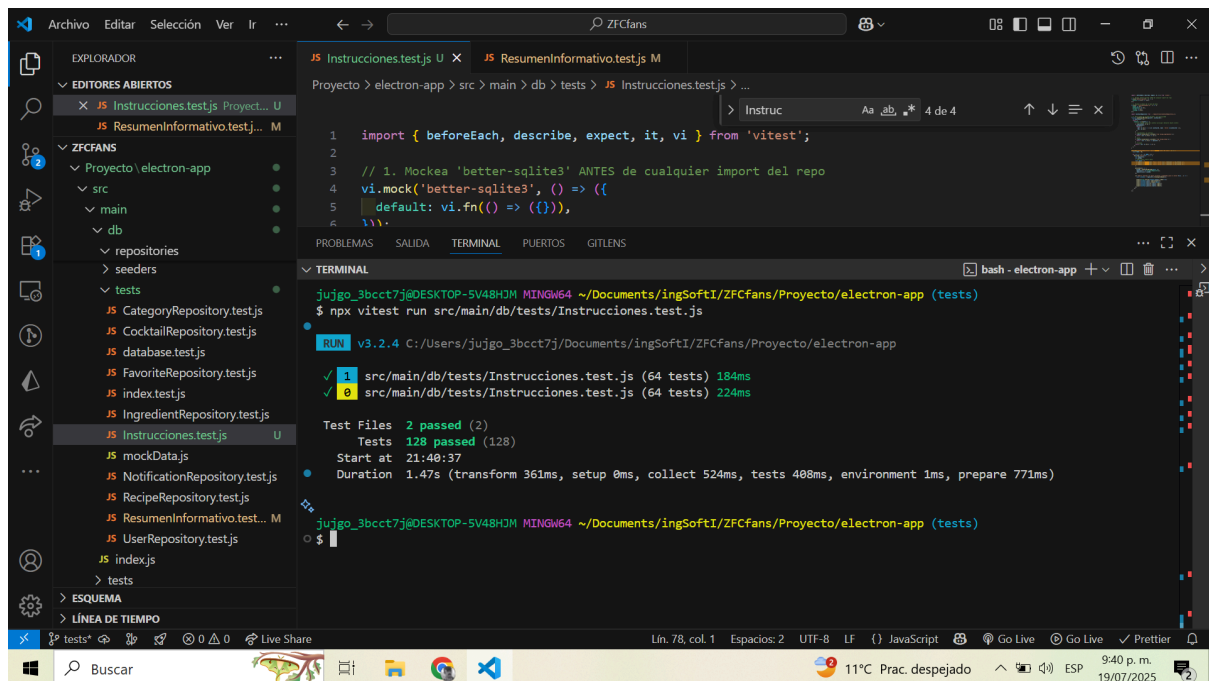
Para la prueba 7, se validó que la información mostrada en los campos de dificultad y duración, mostradas en la cabecera del Resumen Informativo, sean los correctos. Para esto, se construye el archivo `ResumenInformativo.test.js` con la misma estrategia usada para los tests mencionados anteriormente. Los resultados son los siguientes:



Como se puede ver, se logró confirmar que en 128 iteraciones, se ejecutó el programa con el valor correcto renderizado en la ventana.

Prueba 8:

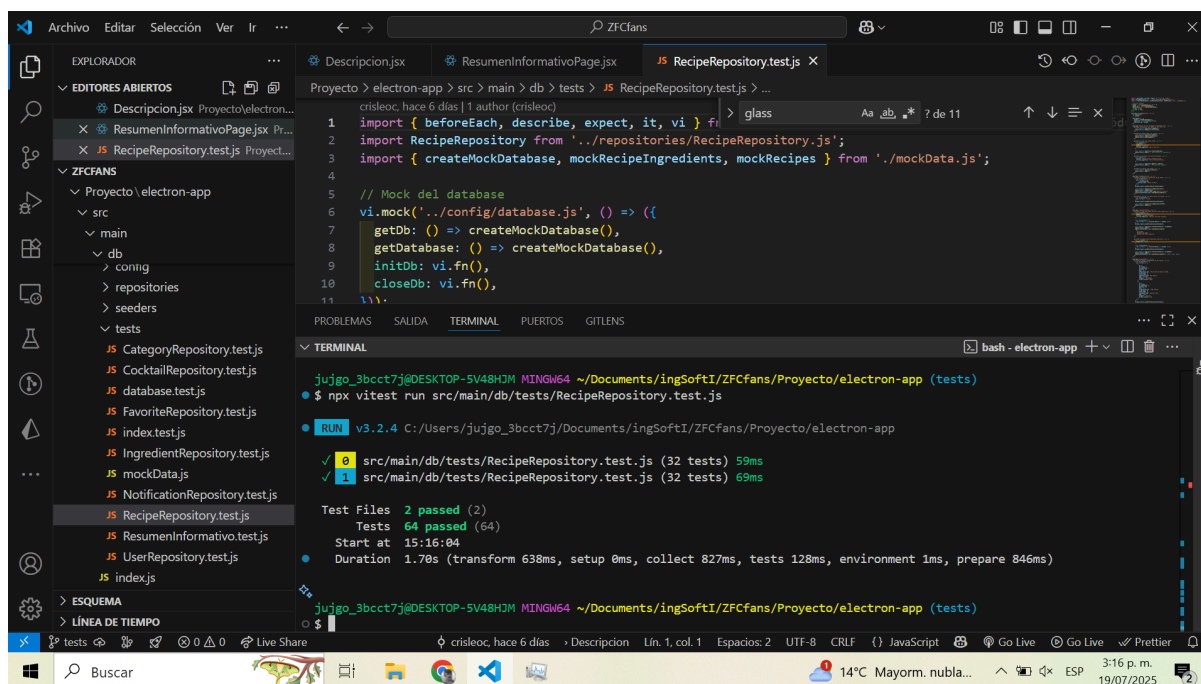
Para la prueba 8, verificamos que los pasos están siendo almacenados de la manera correcta, aparecen en el orden correcto, y que el render muestre las flechas de desplazamiento de la manera correcta. Este test se puede ejecutar en `Instrucciones.test.js`



Se observa que se realiza el test en un total de 128 pasos de pruebas, con los que se consigue un test exitoso en todos los casos planteados.

Prueba 12:

Para la prueba 9, se valida la integridad de las recetas guardadas en la base de datos. Para esto, se ejecuta el archivo [RecipeRepository.js](#). Se evalúa la capacidad de obtener la receta completa, y de sus pasos individuales, así como la modificación correcta de la receta (añadir, editar, eliminar pasos). También permite verificar el tipo de vaso, que va a ser usado en el Resumen Informativo. A continuación se muestran los resultados.



Se realiza el test en 64 conjuntos de datos de prueba, los cuales son todos exitosos.

Cristian

Prueba 13:


El test `CocktailRepository.test.js` verifica la correcta gestión del repositorio principal de cócteles. Evalúa operaciones CRUD básicas como crear, leer, actualizar y eliminar cócteles, así como funcionalidades avanzadas de búsqueda y filtrado. Comprueba la búsqueda de cócteles por nombre, filtrado por categoría, dificultad y tipo de vaso, y la obtención de cócteles por creador. Garantiza que todas las operaciones de base de datos relacionadas con cócteles mantengan la integridad y consistencia de los datos.

Resultados obtenidos: 32 tests ejecutados exitosamente (64 pruebas en total considerando duplicados)

Prueba 14:


El test `IngredientRepository.test.js` (relacionado con el CU_01 de Cristian - Filtrar cócteles según criterios) verifica la funcionalidad de búsqueda y filtrado de ingredientes. Evalúa la capacidad de buscar ingredientes por nombre, validar ingredientes existentes, crear nuevos ingredientes con validación de datos, y

manejar operaciones CRUD sobre ingredientes. Este test es crucial para implementar los filtros de búsqueda por ingredientes que permite al usuario encontrar cócteles que contengan ingredientes específicos, soportando búsquedas con tolerancia a errores ortográficos y sinónimos.

Resultados obtenidos:  17 tests ejecutados exitosamente (34 pruebas en total considerando duplicados)


Prueba 15:

El test `CategoryRepository.test.js` (asociado al CU_03 de Cristian - Visualizar categorías definidas por el usuario) verifica las operaciones de gestión de categorías personalizadas y del sistema. Evalúa la creación, búsqueda y filtrado de categorías, diferenciando entre categorías predefinidas del sistema y categorías creadas por el usuario. Comprueba que la categoría especial "Favoritos" esté siempre disponible y que se mantenga la integridad referencial. Este test garantiza que el panel de categorías funcione correctamente para organizar y visualizar cócteles según las preferencias del usuario.

Resultados obtenidos:  16 tests ejecutados exitosamente (32 pruebas en total considerando duplicados)

Prueba 16:

El test `NotificationRepository.test.js` (relacionado con los CU_02 y CU_04 de Cristian - Historial de búsqueda e historial de cócteles preparados) verifica la funcionalidad de almacenamiento y recuperación de registros históricos del usuario. Evalúa la capacidad de crear, obtener y gestionar notificaciones que pueden incluir registros de búsquedas anteriores y cócteles preparados. Comprueba operaciones como marcar notificaciones como leídas, filtrar por usuario, y mantener un historial limitado. Este test es fundamental para implementar las funcionalidades de historial que permiten al usuario consultar sus búsquedas previas y cócteles preparados anteriormente.

Resultados obtenidos:  15 tests ejecutados exitosamente (30 pruebas en total considerando duplicados)

2. Linting

Nombre: ESLint v9.30.1

Configuración aplicada (`eslint.config.js`):

El proyecto utiliza ESLint con una configuración moderna basada en el formato de configuración plana (flat config). La configuración incluye:

- **Parser:** ECMAScript 2020+ con soporte para módulos ES6 y JSX
- **Plugins utilizados:**
 - @eslint/js: Configuración base recomendada de JavaScript
 - eslint-plugin-react: Reglas específicas para React
 - eslint-plugin-react-hooks: Validación de hooks de React
 - eslint-plugin-react-refresh: Soporte para React Refresh en desarrollo
- **Entornos:** Navegador (browser) y [Node.js](#)
- **Archivos excluidos:** dist/, node_modules/, release/, public/, archivos de configuración y coverage/

Resultados obtenidos:

```
familia castañeda@DESKTOP-51QMDB MINGW64 ~/Programacion/college/ZFCfans/Proyecto/electron-app (dev)
$ npm run lint

> zfcoteles@1.0.0 lint
> eslint .

C:\Users\familiaCastaneda\Programacion\college\ZFCfans\Proyecto\electron-app\src\main\db\config\database.js
  71:5  warning  Unexpected console statement  no-console

C:\Users\familiaCastaneda\Programacion\college\ZFCfans\Proyecto\electron-app\src\main\db\tests\CocktailBDCreated.test.js
  1:34  error    Member 'beforeEach' of the import declaration should be sorted alphabetically  sort-imports

C:\Users\familiaCastaneda\Programacion\college\ZFCfans\Proyecto\electron-app\src\main\main.js
   62:1  warning  Unexpected console statement  no-console
  111:5  warning  Unexpected console statement  no-console
  116:7  warning  Unexpected console statement  no-console
  137:5  warning  Unexpected console statement  no-console

C:\Users\familiaCastaneda\Programacion\college\ZFCfans\Proyecto\electron-app\src\renderer\pages\CatalogoPage.jsx
  29:5  warning  Unexpected console statement  no-console

✖ 7 problems (1 error, 6 warnings)
  1 error and 0 warnings potentially fixable with the `--fix` option.
```

Según el análisis previo realizado en el proyecto, se identificaron 7 problemas de linting:

- **1 error:** ✗ Relacionado con importaciones o sintaxis
- **6 warnings:** ⚠ Principalmente declaraciones de console.log y problemas de ordenamiento de imports

Los warnings más comunes incluyen:

- Uso de `console.log` en código de producción
- Ordenamiento incorrecto de declaraciones de importación
- Variables declaradas pero no utilizadas en algunos casos de test

Ejecución:

Para ejecutar el linter en el proyecto, se utiliza el comando:


```
npm run lint
```

El linting se ejecuta automáticamente durante el proceso de desarrollo y antes de cada commit para mantener la calidad del código. La configuración permite identificar problemas potenciales y asegurar que el código siga las mejores prácticas establecidas para el proyecto.