Para crear pruebas unitarias y de integración para las funcionalidades críticas de tu sistema de gestión de salas y reservas, vamos a seguir un proceso paso a paso utilizando herramientas comunes en el ecosistema de Node.js. La librería que más se utiliza es Jest para realizar pruebas unitarias e integration testing. También puedes utilizar Supertest para pruebas de integración de tus endpoints REST.

Paso a Paso para Implementar Pruebas Unitarias e Integración:

1. Instalar las Dependencias Necesarias

Primero, asegúrate de tener las librerías necesarias para realizar las pruebas.

Instala Jest para pruebas unitarias.

Instala Supertest para pruebas de integración.

```
npm install --save-dev jest supertest
```

Asegúrate de que en tu archivo package.json esté configurado el script para ejecutar las pruebas:

```
"scripts": {
  "test": "jest"
}
```

2. Configuración de Jest

Crea un archivo jest.config.js en la raíz del proyecto para configurar Jest. Aquí puedes especificar configuraciones adicionales como el entorno de pruebas y directorios de salida.

```
module.exports = {
  testEnvironment: 'node',
  verbose: true,
};
```

3. Pruebas Unitarias para el Módulo de Salas

Las pruebas unitarias se centran en verificar que funciones específicas y lógicas independientes funcionan como se espera. Aquí probaremos la lógica de creación y validación de una sala.

Prueba Unitaria para Crear una Sala

Vamos a suponer que tienes una función crearSala que se encarga de crear una sala en tu sistema. Esta es la prueba para esa función:

tests/sala.test.js

```
const { crearSala } = require('../controllers/salaController');
const Sala = require('../models/Sala');

jest.mock('../models/Sala');  // Mockeamos el modelo Sala para evitar usar la base de datos real

describe('Crear Sala', () => {
  test('Debería crear una sala cuando se proporcionan datos válidos', async () => {
    const datosSala = {
      nombre: 'Sala A',
      capacidad: 20,
      recursos: ['Proyector', 'Pizarra'],
      ubicacion: 'Edificio 1'
    };

    Sala.create.mockResolvedValue(datosSala); // Simulamos el comportamiento de crear una sala

    const result = await crearSala(datosSala);

    expect(result).toEqual(datosSala);  // Verificamos que el resultado es el esperado
  });

  test('Debería fallar si no se proporciona un nombre', async () => {
    const datosSala = {
      capacidad: 20,
      recursos: ['Proyector', 'Pizarra'],
      ubicacion: 'Edificio 1'
    };

    await expect(crearSala(datosSala)).rejects.toThrow('El nombre es requerido');
  });
});
```

4. Pruebas de Integración para los Endpoints REST

Las pruebas de integración verifican que diferentes partes de tu aplicación (como controladores, middleware y bases de datos) funcionen correctamente juntas. Aquí vamos a probar un endpoint que crea una nueva sala usando Supertest.

Prueba de Integración para Crear una Sala

Crea un archivo tests/sala.integration.test.js donde probaremos los endpoints de la API. Usaremos Supertest para hacer solicitudes HTTP a la API.

tests/sala.integration.test.js

```
const request = require('supertest');
const app = require('../app');  // Importa tu aplicación Express
const Sala = require('../models/Sala');

describe('API de Salas - Crear Sala', () => {
  test('Debería crear una sala nueva con datos válidos', async () => {
    const datosSala = {
      nombre: 'Sala B',
      capacidad: 15,
      recursos: ['Pizarra', 'Proyector'],
      ubicacion: 'Edificio 2'
    };

    const response = await request(app)
      .post('/salas')
      .send(datosSala)
      .expect(201);  // Esperamos que el código de respuesta sea 201 (Created)

    expect(response.body.nombre).toBe(datosSala.nombre);
    expect(response.body.capacidad).toBe(datosSala.capacidad);
  });

  test('Debería devolver error si falta el nombre de la sala', async () => {
    const datosSala = {
      capacidad: 15,
      recursos: ['Pizarra', 'Proyector'],
      ubicacion: 'Edificio 2'
    };

    const response = await request(app)
      .post('/salas')
      .send(datosSala)
      .expect(400);  // Esperamos un código de error 400 (Bad Request)

    expect(response.body.errors).toBeDefined();
    expect(response.body.errors[0].msg).toBe('El nombre de la sala es requerido');
  });
});
```

5. Pruebas para Sistema de Reservas

De la misma manera, se pueden hacer pruebas unitarias e integraciones para las reservas.

Prueba Unitaria para Crear una Reserva

Aquí verificamos la lógica detrás de la creación de una reserva, asegurándonos de que no se puedan reservar salas con conflictos de horarios.

tests/reserva.test.js

```
const { crearReserva } = require('../controllers/reservaController');
const Reserva = require('../models/Reserva');

jest.mock('../models/Reserva');

describe('Crear Reserva', () => {
  test('Debería crear una reserva si no hay conflicto de horarios', async () => {
    const reserva = {
      salaId: '12345',
      usuario: 'usuario1',
      fechaInicio: new Date('2024-09-01T10:00:00'),
      fechaFin: new Date('2024-09-01T11:00:00'),
    };

    Reserva.find.mockResolvedValue([]);  // No hay reservas que coincidan

    const result = await crearReserva(reserva);

    expect(result.usuario).toBe('usuario1');
    expect(result.salaId).toBe('12345');
  });

  test('Debería fallar si hay conflicto de horarios', async () => {
    const reservaExistente = {
      salaId: '12345',
      fechaInicio: new Date('2024-09-01T10:00:00'),
      fechaFin: new Date('2024-09-01T11:00:00'),
    };

    Reserva.find.mockResolvedValue([reservaExistente]);  // Simulamos un conflicto

    const reserva = {
      salaId: '12345',
      usuario: 'usuario2',
      fechaInicio: new Date('2024-09-01T10:30:00'),
      fechaFin: new Date('2024-09-01T11:30:00'),
    };

    await expect(crearReserva(reserva)).rejects.toThrow('Conflicto de horarios');
  });
});
```

Prueba de Integración para Crear una Reserva

tests/reserva.integration.test.js

```
const request = require('supertest');
const app = require('../app');
const Reserva = require('../models/Reserva');

describe('API de Reservas - Crear Reserva', () => {
  test('Debería crear una reserva nueva sin conflicto de horario', async () => {
    const reserva = {
      salaId: '12345',
      usuario: 'usuario1',
      fechaInicio: new Date('2024-09-01T10:00:00'),
      fechaFin: new Date('2024-09-01T11:00:00')
    };

    const response = await request(app)
      .post('/reservas')
      .send(reserva)
      .expect(201);

    expect(response.body.usuario).toBe('usuario1');
  });

  test('Debería devolver error si hay un conflicto de horario', async () => {
    const reserva = {
      salaId: '12345',
      usuario: 'usuario1',
      fechaInicio: new Date('2024-09-01T10:30:00'),
      fechaFin: new Date('2024-09-01T11:30:00')
    };

    const response = await request(app)
      .post('/reservas')
      .send(reserva)
      .expect(400);

    expect(response.body.message).toBe('Conflicto de horarios');
  });
});
```

6. Ejecución de Pruebas

Para ejecutar todas las pruebas unitarias e integración:

```
npm test
```

Consideraciones Adicionales:

1. Cobertura de Pruebas: Para verificar la cobertura de tus pruebas, puedes utilizar la bandera --coverage en Jest. Esto te mostrará qué porcentaje de tu código está siendo cubierto por las pruebas.

npm test -- --coverage

2. Bases de Datos Mockeadas: Al realizar pruebas de integración, puedes utilizar bases de datos en memoria como MongoDB In-Memory Server para evitar afectar tu base de datos real.

Conclusión:

Este proceso detalla cómo puedes escribir pruebas unitarias y de integración para las funcionalidades más críticas de tu s