Ejercicios con Controladores, Servicios y Repositorios

🌼 Ejercicio 5: Perfil de usuario (Relación 1 a 1)

🎯 Objetivo:
Construir una pequeña aplicación que permita registrar usuarios con su perfil detallado (datos personales extendidos). Debe persistirse en base de datos. **Resolver utilizando TDD.**

📝 Requisitos:

1. Clases modelo:

```
@Entity
public class Usuario {
    @Id
    private Long id;
    private String email;
    private String password;

    @OneToOne(mappedBy = "usuario", cascade = CascadeType.ALL)
    private Perfil perfil;
}

@Entity
public class Perfil {
    @Id
    private Long id;
    private String nombre;
    private String apellido;
    private String telefono;

    @OneToOne
    @JoinColumn(name = "usuario_id")
    private Usuario usuario;
}
```

2. Repositorio `UsuarioRepository`:

- `Usuario buscarPorEmail(String email)`
- `void guardar(Usuario usuario)`

3. Servicio `UsuarioService`:

- Guardar un nuevo usuario con perfil asociado
- Buscar un usuario por email
- Pruebas unitarias con mocks para validar lógica sin DB

4. Controlador `UsuarioController`:

- GET `/registro` → muestra formulario de registro
- POST `/registro` → recibe y guarda datos del usuario + perfil

5. Vista `registro.html`:

- Formulario con campos de email, password, nombre, apellido y teléfono

---

🌼 Ejercicio 6: Sistema de órdenes (Relación 1 a N)

🎯 Objetivo:
Gestionar órdenes de compra asociadas a un cliente. Persistir en base de datos con relación 1 cliente → muchas órdenes. **Resolver utilizando TDD.**

📝 Requisitos:

1. Clases modelo:

```
@Entity
public class Cliente {
    @Id
    private Long id;
    private String nombre;

    @OneToMany(mappedBy = "cliente", cascade = CascadeType.ALL)
    private List<Orden> ordenes;
}

@Entity
public class Orden {
    @Id
    private Long id;
    private String descripcion;
    private LocalDate fecha;

    @ManyToOne
    @JoinColumn(name = "cliente_id")
    private Cliente cliente;
}
```

2. Repositorio `OrdenRepository`:

- `void guardar(Orden orden)`
- `List<Orden> buscarPorClienteId(Long clienteId)`

3. Servicio `OrdenService`:

- Crear orden nueva asociada a un cliente existente
- Obtener todas las órdenes de un cliente por ID
- Pruebas unitarias con mocks para validar lógica sin DB

4. Controlador `OrdenController`:

- GET `/clientes/{id}/ordenes` → ver órdenes

- POST `/clientes/{id}/ordenes` → crear nueva orden

5. Vista `ordenes.html`:

- Mostrar todas las órdenes del cliente en una tabla con:
  - Descripción de la orden
  - Fecha de creación
- Formulario para agregar una nueva orden:
  - Campo de texto para la descripción
  - Selector de fecha (tipo `date`)
  - Botón para enviar

🌼 Ejercicio 7: Plataforma de cursos (Relaciones 1 a N y N a N)

🎯 Objetivo:
Desarrollar una aplicación que permita gestionar cursos, cada curso tiene muchos módulos, y los cursos pueden ser realizados por múltiples estudiantes (y cada estudiante puede inscribirse a muchos cursos). **Resolver utilizando TDD.**

📝 Requisitos:

1. Clases modelo:

```
@Entity
public class Curso {
    @Id
    private Long id;
    private String titulo;

    @OneToMany(mappedBy = "curso", cascade = CascadeType.ALL)
    private List<Modulo> modulos;

    @ManyToMany
    @JoinTable(
        name = "curso_estudiante",
        joinColumns = @JoinColumn(name = "curso_id"),
        inverseJoinColumns = @JoinColumn(name = "estudiante_id")
    )
    private Set<Estudiante> estudiantes;
}

@Entity
public class Modulo {
    @Id
    private Long id;
    private String nombre;
    private int duracionHoras;

    @ManyToOne
    @JoinColumn(name = "curso_id")
    private Curso curso;
}

@Entity
public class Estudiante {
    @Id
    private Long id;
    private String nombre;
```

```
    private String email;

    @ManyToMany(mappedBy = "estudiantes")
    private Set<Curso> cursos;
}
```

2. Repositorio:

- `CursoRepository`:
    - `void guardar(Curso curso)`
    - `Curso buscarPorId(Long id)`
- `EstudianteRepository`:
    - `Estudiante buscarPorId(Long id)`
- `ModuloRepository`:
    - `void guardar(Modulo modulo)`

3. Servicio `CursoService`:

- Crear un nuevo curso con módulos
- Inscribir estudiantes a cursos existentes
- Obtener la lista de estudiantes de un curso
- Pruebas unitarias con mocks para validar lógica sin DB

4. Controlador `CursoController`:

- GET `/cursos` → mostrar cursos disponibles
- POST `/cursos` → crear curso con módulos
- POST `/cursos/{id}/inscribir` → inscribir estudiante

5. Vistas:

- `cursos.html` → lista de cursos, opción para inscribir estudiante
- `crearCurso.html` → formulario para agregar curso y módulos
