

Aprendizaje-Express — ISS-03

Feature Client: CRUD por Capas A...E (Arquitectura Limpia)

Modelos Sequelize, Hooks de bcrypt, DTOs, Repositorio, Servicio con Borrado Lógico, Controlador y Rutas REST

Propósito del ISS-03: Despliegue de la Arquitectura por Capas

Creación de la primera entidad de dominio (Client) sobre la ruta /api/clientes

Desacoplamiento Total

Se separan las responsabilidades en 5 capas estrictas: HTTP → Controller → Service → Repository → Model → BD.

Seguridad & Criptografía

Hasheo automático de contraseñas con bcrypt (12 rondas) integrado en hooks del modelo Sequelize.

Persistencia & Borrado Lógico

Soporte de borrado físico e implementación de borrado lógico preservando historial (status = 'inactive').

La Cadena de Responsabilidades en 5 Capas

Flujo atómico de una petición HTTP en la feature Clients

1. Controller: Expone handlers HTTP, valida parámetros de entrada y formatea respuestas con BaseController.

2. Service: Aplica reglas de negocio (borrado lógico, unicidad de correo) y maneja excepciones con AppError.

3. Repository: Aísla las consultas de acceso a datos de Sequelize (findAll, findById, create, update).

4. Model: Define el esquema Sequelize de la tabla 'clients', tipos TypeScript y hooks de bcrypt.

5. Database: Tabla física 'clients' en la BD relacional sincronizada automáticamente por la ORM.

Capa 1 — Modelo Sequelize e Hooks Criptográficos

Protección de credenciales mediante bcrypt a nivel de infraestructura ORM

Definición del Modelo (client.model.ts)

- Mapea la tabla 'clients' con atributos TypeScript en Sequelize.
- Campos: id, name, email, password_hash, status ('active'/'inactive'), timestamps.
- Garantiza que la propiedad 'password_hash' nunca se guarde en texto claro.

Hooks de Bcrypt (beforeCreate / beforeUpdate)

```
Client.beforeCreate(async (client) => {  
  if (client.password_hash) {  
    const salt = await bcrypt.genSalt(12);  
    client.password_hash = await  
    bcrypt.hash(client.password_hash, salt);  
  }  
});
```

Capa 2 — DTOs y Contratos de Entrada/Salida

Saneamiento de payloads y prevención de fuga de hashes sensibles

CreateClientDto

Valida que name, email y password_hash vengan presentes al crear un cliente.

UpdateClientDto

Permite actualización completa de los datos del cliente (PUT).

PatchClientDto

Permite actualización parcial opcional de campos (PATCH).

ClientResponseDto

Filtra explícitamente el campo password_hash para no retornarlo en JSON de respuesta.

Capa 3 — ClientsRepository (Acceso a Datos)

Aislamiento directo de las operaciones de lectura y escritura de Sequelize

Métodos del Repositorio (`clients.repository.ts`)

- **`findAll()`**: Retorna todos los clientes registrados en la base de datos.
- **`findById(id)`**: Busca un cliente por su clave primaria ID (retorna null si no existe).
- **`findByEmail(email)`**: Busca un cliente por dirección de correo electrónico para validar duplicidades.
- **`create(data)`**: Persiste un nuevo registro en la tabla `clients` ejecutando los hooks del modelo.
- **`update(id, data)`**: Actualiza los campos de un cliente existente por su ID.
- **`delete(id)`**: Ejecuta un borrado físico DELETE en la tabla `clients`.

Capa 4 — ClientsService (Lógica de Negocio)

Reglas de dominio, borrado lógico y manejo centralizado con AppError

Validaciones e Invariantes

- Verificación de Duplicidad: Al crear un cliente, consulta si el email ya existe lanzando `AppError('Email ya registrado', 409)`.
- Verificación de Existencia: Al buscar o actualizar por ID, si el cliente es null lanza `AppError('Cliente no encontrado', 404)`.
- Transformación DTO: Mapea la salida mediante `toUserResponse`.

Borrado Lógico (`deleteLogical`)

- En lugar de ejecutar una instrucción DELETE en la BD, el servicio cambia el campo status a 'inactive'.
- Preserva la integridad referencial histórica con ventas u otros módulos.
- Reutiliza el método update del repositorio enviando `{ status: 'inactive' }`.

Capa 5 — ClientsController y BaseController

Manejo estandarizado de peticiones HTTP e inyección de respuestas estructuradas

Anatomía de ClientsController (extends BaseController)

- **BaseController:** Clase base que provee helper methods como `sendSuccess(res, data, status)` y `sendError(res, error)`.
- **getAll():** Petición GET `/api/clientes` → Llama al servicio y responde HTTP 200 con la colección.
- **getById():** Petición GET `/api/clientes/:id` → Parsea `req.params.id` e invoca servicio (200 OK / 404 Not Found).
- **create():** Petición POST `/api/clientes` → Procesa `req.body` mediante `CreateClientDto` y responde HTTP 201 Created.
- **update() / patch():** Petición PUT/PATCH `/api/clientes/:id` → Actualiza datos y retorna objeto saneado (HTTP 200).
- **delete():** Petición DELETE `/api/clientes/:id` → Invoca `deleteLogical` y confirma desactivación (HTTP 200).

Enrutamiento y Cableado Global del Sistema

Exposición de endpoints en `clients.routes.ts`, `routes/index.ts` y `config/index.ts`

Rutas de la Feature (`clients.routes.ts`)

- `GET /` → `clientController.getAll`
- `GET /:id` → `clientController.getById`
- `POST /` → `clientController.create`
- `PUT /:id` → `clientController.update`
- `PATCH /:id` → `clientController.patch`
- `DELETE /:id` → `clientController.delete`

Agregador y Servidor Central

1. `src/routes/index.ts`:
`router.use('/clientes', clientRoutes);`
2. `src/config/index.ts` (`App.routes`):
`this.app.use('/api', apiRoutes);`

Resultado: Toda la feature se expone bajo el prefijo unificado `/api/clientes`.

Pruebas de Integración con Archivos HTTP

Verificación mediante el cliente REST integrado en src/features/business/clients/http/

Colección de Pruebas REST (client.http)

- **POST /api/clientes:** Envía JSON con cliente válido → Responde 201 Created con objeto saneado (sin hash).
- **GET /api/clientes:** Consulta lista completa → Responde 200 OK con array de clientes.
- **GET /api/clientes/:id:** Consulta ID existente → Responde 200 OK / Consulta ID inexistente → Responde 404 Not Found.
- **PUT / PATCH /api/clientes/:id:** Actualiza campos parciales o totales → Responde 200 OK con cliente actualizado.
- **DELETE /api/clientes/:id:** Ejecuta borrado lógico → Responde 200 OK y asigna status = 'inactive' en BD.

Criterios de Cierre GATE (Definition of Done)

Verificaciones obligatorias para dar por aprobado el ISS-03

Checklist de Cierre del ISS-03

- [x] **Compilación Estricta TypeScript:** `npx tsc --noEmit` finaliza con 0 errores de tipos.
- [x] **Sincronización de BD:** Sequelize crea la tabla física 'clients' en la BD al arrancar el servidor.
- [x] **Hasheo Seguro de Claves:** Verificado que la columna `password_hash` contiene un hash de bcrypt (12 rondas).
- [x] **Sanitización de Salida:** Comprobado que ninguna respuesta HTTP expone el campo `password_hash`.
- [x] **Integración de Rutas HTTP:** Pruebas REST completas sobre `/api/clientes` aprobadas (200, 201, 400, 404, 409).

¡ISS-03 Completado con Éxito!

Feature Client completamente funcional con arquitectura en 5 capas.

Siguiente Hito: ISS-04 — Seeders con Faker (Feature + Runner)