Module 3: Backend API

Node.js, Express & Database

Adrián Catalán

adriancatalan@galileo.edu

Agenda

1. **Project: EstateHub API**
2. **Node.js & Express Basics**
3. **Controllers & Middlewares**
4. **Database Connection**
5. **Deep Dive**
6. **Challenge Lab**

EstateHub API

We are building a REST API for real estate property management.

Core Requirements:

1. **CRUD Operations**: Create, Read, Update, Delete properties.
2. **Filtering**: Query properties by type, price, location.
3. **Validation**: Ensure data integrity with Zod.
4. **Persistence**: SQLite database with Prisma ORM.

API Endpoints

```
┌────────────────────────────────────────────────────────────────┐
│                     RealEstate Hub API                         │
├────────────────────────────────────────────────────────────────┤
│                                                                │
│  GET    /api/properties           List all properties (filtered) │
│  GET    /api/properties/:id       Get property by ID           │
│  POST   /api/properties           Create new property          │
│  PUT    /api/properties/:id       Update property              │
│  DELETE /api/properties/:id       Delete property              │
│                                                                │
│  GET    /api/properties/stats     Get statistics               │
│                                                                │
└────────────────────────────────────────────────────────────────┘
```

2. Express.js Fundamentals

What is Express?

Express is a minimal web framework for Node.js.

```javascript
import express from 'express';

const app = express();

// Middleware
app.use(express.json());

// Route
app.get('/', (req, res) => {
    res.json({ message: 'Hello, World!' });
});

// Start server
app.listen(3000, () => {
    console.log('Server running on http://localhost:3000');
});
```

Request-Response Cycle

```
┌─────────────────────────────────────────────────────────────────┐
│                    Express Request Flow                         │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  Client                                                         │
│    │                                                            │
│    │ HTTP Request (GET /api/properties)                         │
│    ▼                                                            │
│  Express App                                                    │
│    │                                                            │
│    ├── Middleware 1 (cors)                                      │
│    ├── Middleware 2 (json parser)                               │
│    ├── Middleware 3 (logging)                                   │
│    │                                                            │
│    ▼                                                            │
│  Route Handler                                                  │
│    │                                                            │
│    ▼                                                            │
│  Response (JSON)                                                │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

Middleware

Functions that execute during the request-response cycle.

```
// Built-in middleware
app.use(express.json());        // Parse JSON body
app.use(express.static('public')); // Serve static files

// Third-party middleware
import cors from 'cors';
app.use(cors());                // Enable CORS

// Custom middleware
app.use((req, res, next) => {
    console.log(`${req.method} ${req.path}`);
    next(); // Pass to next middleware
});
```

The Request Object

Access incoming request data.

```
app.get('/api/properties', (req, res) => {
    // Query parameters: /api/properties?type=house&maxPrice=500000
    const { type, maxPrice } = req.query;

    // URL parameters: /api/properties/:id
    const { id } = req.params;

    // Request body (POST/PUT)
    const { title, price } = req.body;

    // Headers
    const token = req.headers.authorization;

    res.json({ type, maxPrice, id });
});
```

The Response Object

Send responses back to the client.

```
app.get('/api/properties/:id', (req, res) => {
    const property = findProperty(req.params.id);

    if (!property) {
        // Set status code and send JSON
        return res.status(404).json({
            error: 'Property not found'
        });
    }

    // Default status is 200
    res.json(property);
});

app.post('/api/properties', (req, res) => {
    const newProperty = createProperty(req.body);
    // 201 Created
    res.status(201).json(newProperty);
```

Router: Organizing Routes

Group related routes together.

```typescript
// routes/propertyRoutes.ts
import { Router } from 'express';
import * as controller from '../controllers/propertyController';

const router = Router();

router.get('/', controller.getAllProperties);
router.get('/:id', controller.getPropertyById);
router.post('/', controller.createProperty);
router.put('/:id', controller.updateProperty);
router.delete('/:id', controller.deleteProperty);

export default router;

// server.ts
import propertyRoutes from './routes/propertyRoutes';
app.use('/api/properties', propertyRoutes);
```

Error Handling Middleware

Centralized error handling.

```
// middlewares/errorHandler.ts
import { Request, Response, NextFunction } from 'express';

export function errorHandler(
    err: Error,
    req: Request,
    res: Response,
    next: NextFunction
): void {
    console.error('Error:', err.message);

    if (err.name === 'ValidationError') {
        res.status(400).json({ error: err.message });
        return;
    }

    res.status(500).json({ error: 'Internal server error' });
}

// server.ts (must be last middleware)
```

3. Prisma ORM

What is Prisma?

Prisma is a next-generation ORM for Node.js and TypeScript.

Components:

1. **Prisma Client**: Auto-generated, type-safe database client.
2. **Prisma Migrate**: Declarative schema migrations.
3. **Prisma Studio**: GUI to view and edit data.

```
# Installation
npm install prisma @prisma/client
npx prisma init
```

Prisma Schema

Define your data model in `prisma/schema.prisma`.

```
// prisma/schema.prisma
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = "file:./dev.db"
}

model Property {
  id          String   @id @default(uuid())
  title       String
  description String
  type        String
  price       Float
  bedrooms    Int
  bathrooms   Int
  area        Float
  address     String
  city        String
  imageUrl    String?
  createdAt   DateTime @default(now())
```

Prisma Commands

```
# Generate Prisma Client (after schema changes)
npx prisma generate

# Create and apply migrations
npx prisma migrate dev --name init

# Push schema changes (development)
npx prisma db push

# Open Prisma Studio (GUI)
npx prisma studio

# Seed the database
npx prisma db seed
```

CRUD with Prisma Client

```
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

// CREATE
const property = await prisma.property.create({
    data: {
        title: 'Beach House',
        price: 500000,
        type: 'house',
        // ... other fields
    }
});

// READ (all)
const properties = await prisma.property.findMany();

// READ (by ID)
const property = await prisma.property.findUnique({
    where: { id: 'abc123' }
});
```

Update & Delete

```
// UPDATE
const updated = await prisma.property.update({
    where: { id: 'abc123' },
    data: {
        price: 450000,
        title: 'Updated Beach House'
    }
});

// DELETE
const deleted = await prisma.property.delete({
    where: { id: 'abc123' }
});

// DELETE MANY
const count = await prisma.property.deleteMany({
    where: { type: 'apartment' }
});
```

Filtering & Sorting

```
// Complex query
const properties = await prisma.property.findMany({
    where: {
        type: 'house',
        price: {
            lte: 500000  // less than or equal
        },
        city: {
            contains: 'Beach'  // LIKE '%Beach%'
        }
    },
    orderBy: {
        price: 'asc'
    },
    take: 10,   // LIMIT
    skip: 0     // OFFSET
});
```

Prisma Filter Operators

Operator	Description	Example
`equals`	Exact match	`price: { equals: 500000 }`
`not`	Not equal	`type: { not: 'land' }`
`in`	In array	`type: { in: ['house', 'condo'] }`
`lt`, `lte`	Less than	`price: { lt: 1000000 }`
`gt`, `gte`	Greater than	`bedrooms: { gte: 3 }`
`contains`	Substring	`title: { contains: 'Beach' }`
`startsWith`	Prefix	`city: { startsWith: 'New' }`

Seeding Data

```
// prisma/seed.ts
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

async function main() {
    await prisma.property.deleteMany(); // Clear existing

    await prisma.property.createMany({
        data: [
            {
                title: 'Modern Apartment',
                type: 'apartment',
                price: 250000,
                bedrooms: 2,
                // ...
            },
            // More properties...
        ]
    });

    console.log('Database seeded!');
}

main().finally(() => prisma.$disconnect());
```

4. REST API Design

REST Principles

REpresentational **S**tate **T**ransfer

1. **Resources**: URLs represent entities (`/properties`, `/users`)
2. **HTTP Verbs**: Actions on resources (GET, POST, PUT, DELETE)
3. **Stateless**: Each request is independent
4. **JSON**: Standard data format

HTTP Status Codes

Code	Meaning	Usage
200	OK	Successful GET, PUT
201	Created	Successful POST
204	No Content	Successful DELETE
400	Bad Request	Validation error
404	Not Found	Resource doesn't exist
409	Conflict	Duplicate resource
500	Internal Error	Server error

Request Validation with Zod

Type-safe schema validation.

```
import { z } from 'zod';

// Define schema
const CreatePropertySchema = z.object({
    title: z.string().min(3).max(100),
    description: z.string().min(10),
    type: z.enum(['house', 'apartment', 'condo', 'land']),
    price: z.number().positive(),
    bedrooms: z.number().int().min(0),
    bathrooms: z.number().int().min(0),
    area: z.number().positive(),
    address: z.string().min(5),
    city: z.string().min(2),
    imageUrl: z.string().url().optional()
});

// Infer TypeScript type from schema
type CreatePropertyInput = z.infer<typeof CreatePropertySchema>;
```

Using Zod in Controllers

```typescript
// controllers/propertyController.ts
export async function createProperty(req: Request, res: Response): Promise<void> {
    // Validate request body
    const result = CreatePropertySchema.safeParse(req.body);

    if (!result.success) {
        res.status(400).json({
            error: 'Validation failed',
            details: result.error.issues
        });
        return;
    }

    // result.data is typed as CreatePropertyInput
    const property = await prisma.property.create({
        data: result.data
    });

    res.status(201).json(property);
}
```

Controller Pattern

Separate business logic from routing using Repository pattern.

```
// controllers/propertyController.ts
import type { Request, Response } from 'express';
import { propertyRepository } from '../repositories/propertyRepository';

export async function getAllProperties(req: Request, res: Response): Promise<void> {
    try {
        const filters: PropertyFilters = {
            search: req.query.search as string | undefined,
            propertyType: req.query.propertyType as PropertyFilters['propertyType'],
            // ... more filters
        };

        const properties = await propertyRepository.findAll(filters);

        res.json({ success: true, data: properties });
    } catch (error) {
        res.status(500).json({ success: false, error: { message: 'Error interno' } });
    }
}
```

Query Parameters Best Practices

```
// GET /api/properties?propertyType=casa&minPrice=100000&maxPrice=500000&city=Madrid

export async function getAllProperties(req: Request, res: Response): Promise<void> {
    const filters: PropertyFilters = {
        search: req.query.search as string | undefined,
        propertyType: req.query.propertyType as PropertyFilters['propertyType'],
        operationType: req.query.operationType as PropertyFilters['operationType'],
        minPrice: req.query.minPrice ? Number(req.query.minPrice) : undefined,
        maxPrice: req.query.maxPrice ? Number(req.query.maxPrice) : undefined,
        minBedrooms: req.query.minBedrooms ? Number(req.query.minBedrooms) : undefined,
        city: req.query.city as string | undefined,
    };

    const properties = await propertyRepository.findAll(filters);

    res.json({ success: true, data: properties });
}
```

API Response Format

Consistent response structure.

```
// Success response
{
    "success": true,
    "data": [...]
}

// Error response
{
    "success": false,
    "error": {
        "message": "Datos de entrada inválidos",
        "code": "VALIDATION_ERROR",
        "details": [
            { "path": ["price"], "message": "Price must be positive" }
        ]
    }
}
```

Full-Stack Architecture

Frontend + Backend Separation

Why Separate Frontend & Backend?

Separation of Concerns:

Layer	Responsibility	Technology
Frontend	User interface, UX	React, Vue, Angular
Backend	Business logic, data	Express, Prisma
Database	Data persistence	SQLite, PostgreSQL

Benefits:

- Independent development and deployment
- Different teams can work in parallel
- Technology flexibility (swap frontend without touching backend)
- Better scalability and caching

Communication Between Layers

```
+-----------------------+           +-----------------------+
|       FRONTEND        |           |        BACKEND        |
|    (React + Vite)     |           |   (Express + Prisma)  |
|    localhost:3001     |           |    localhost:3002     |
+-----------+-----------+           +-----------+-----------+
            |                                   |
            |  1. HTTP Request                  |
            |  GET /api/properties              |
            |---------------------------------->|
            |                                   |
            |                            2. Query DB
            |                                   |
            |  3. HTTP Response                 |
            |  { data: [...] }                  |
            |<----------------------------------|
            |                                   |
+-----------+-----------+           +-----------+-----------+
|   Update React State  |           |    Process Next Req   |
+-----------------------+           +-----------------------+
```

Port Configuration

Each layer runs on a different port:

```
# Backend (Express)
PORT=3002
npm run dev  # http://localhost:3002

# Frontend (Vite)
PORT=3001
npm run dev  # http://localhost:3001
```

Why different ports?

- Avoid conflicts
- Simulate production environment
- Enable CORS configuration
- Clear separation during development

CORS: Cross-Origin Resource Sharing

Frontend (port 3001) needs permission to call backend (port 3002).

```
// backend/src/server.ts
import cors from 'cors';

app.use(cors({
  origin: 'http://localhost:3001',  // Allow frontend
  credentials: true
}));
```

Without CORS: Browser blocks requests between different origins.

Frontend API Layer

The frontend creates an abstraction for API calls:

```
// frontend/src/lib/api.ts
const API_BASE_URL = 'http://localhost:3002/api';

export async function getAllProperties(): Promise<Property[]> {
  const response = await fetch(`${API_BASE_URL}/properties`);
  const result = await response.json();
  return result.data;
}

export async function createProperty(data: CreatePropertyInput): Promise<Property | null> {
  const response = await fetch(`${API_BASE_URL}/properties`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(data)
  });
  const result = await response.json();
  return result.success ? result.data : null;
}
```

From localStorage to API

Module 2 (localStorage - sync):

```
export function getAllProperties(): Property[] {
  const data = localStorage.getItem('properties');
  return data ? JSON.parse(data) : [];
}
```

Module 3 (API - async):

```
export async function getAllProperties(): Promise<Property[]> {
  const response = await fetch(`${API_BASE_URL}/properties`);
  const result = await response.json();
  return result.data;
}
```

Key difference: All operations become async (Promises).

Handling Async in React Components

```
// BEFORE (sync)
const loadProperties = useCallback(() => {
  const properties = filterProperties(filters);
  setProperties(properties);
}, [filters]);

// AFTER (async)
const loadProperties = useCallback(async () => {
  setIsLoading(true);
  try {
    const properties = await filterProperties(filters);
    setProperties(properties);
  } catch (error) {
    console.error('Error:', error);
  } finally {
    setIsLoading(false);
  }
}, [filters]);
```

Loading States for UX

```
function HomePage() {
  const [properties, setProperties] = useState<Property[]>([]);
  const [isLoading, setIsLoading] = useState(true);

  // ... load data

  return (
    <div>
      {isLoading ? (
        <p>Loading properties...</p>
      ) : properties.length > 0 ? (
        <PropertyList properties={properties} />
      ) : (
        <p>No properties found</p>
      )}
    </div>
  );
}
```

Error Handling

```
// API layer handles errors gracefully
export async function createProperty(data) {
  try {
    const response = await fetch(`${API_BASE_URL}/properties`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(data)
    });

    const result = await response.json();
    return result.success ? result.data : null;
  } catch (error) {
    console.error('Network error:', error);
    return null;  // Return null, not throw
  }
}
```

Summary: Layer Separation

Aspect	Frontend	Backend
Port	3001	3002
Framework	React + Vite	Express
Data Access	fetch() API	Prisma ORM
State	useState/useEffect	Database
Validation	Zod (UI)	Zod (API)
Async	Yes (fetch)	Yes (Prisma)

The API is the contract between frontend and backend.

5. Deep Dive

1. Prisma Client Generation

How Prisma generates the type-safe client.

```
┌─────────────────────────────────────────────────────────────┐
│                     Prisma Generation                        │
├─────────────────────────────────────────────────────────────┤
│                                                              │
│  schema.prisma                                               │
│       │                                                      │
│       ▼                                                      │
│  npx prisma generate                                         │
│       │                                                      │
│       ├── Reads schema                                       │
│       ├── Generates TypeScript types                         │
│       └── Creates query builder                              │
│       │                                                      │
│       ▼                                                      │
│  node_modules/.prisma/client/                                │
│       ├── index.d.ts (Types)                                 │
│       └── index.js (Runtime)                                 │
│                                                              │
```

2. SQLite: File-Based Database

SQLite stores the entire database in a single file.

Advantages:

- Zero configuration
- No separate server process
- Perfect for development and small apps
- Easy to backup (copy the file)

Limitations:

- Single writer at a time
- Not suitable for high-concurrency apps
- No network access (local only)

File Location: [illegible]

3. Express Async Errors

Handle async errors properly.

```
// BAD: Unhandled promise rejection
app.get('/api/properties', async (req, res) => {
    const properties = await prisma.property.findMany(); // May throw!
    res.json(properties);
});

// GOOD: Try-catch wrapper
app.get('/api/properties', async (req, res, next) => {
    try {
        const properties = await prisma.property.findMany();
        res.json(properties);
    } catch (error) {
        next(error); // Pass to error middleware
    }
});

// BETTER: Use express-async-errors package
import 'express-async-errors';
```

4. Environment Variables

Configure app without code changes.

```
# .env
DATABASE_URL="file:./dev.db"
PORT=3000
NODE_ENV=development
```

```
// Load with dotenv
import 'dotenv/config';

const port = process.env.PORT || 3000;
const isDev = process.env.NODE_ENV === 'development';

app.listen(port, () => {
    console.log(`Server running on port ${port}`);
});
```

Security: Never commit `.env` to git!

5. Prisma Transactions

Ensure data consistency across multiple operations.

```
// Interactive transaction
const result = await prisma.$transaction(async (tx) => {
    // Create user
    const user = await tx.user.create({
        data: { email: 'john@example.com' }
    });

    // Create their first property
    const property = await tx.property.create({
        data: {
            title: 'My House',
            ownerId: user.id
        }
    });

    return { user, property };
});
```

6. Challenge Lab

Practice & Application

Part 1: Pagination & Metadata

Context:
The API currently returns all properties at once. For large datasets, this is inefficient and slow.

Your Task:
Implement proper pagination that:

- Accepts `page` and `limit` query parameters
- Returns metadata (total count, pages, current page)
- Supports cursor-based pagination (optional bonus)
- Returns empty array for out-of-range pages

Files to Modify:

- `src/controllers/propertyController.ts`

Part 1: Definition of Done

Criteria	Description
Query params	`?page=1&limit=10` works correctly
Metadata returned	Response includes `{ data, meta: { total, page, limit, pages } }`
Total count	`meta.total` shows actual count of matching records
Page calculation	`meta.pages` = ceil(total / limit)
Empty pages	Returns empty data array, not error
Default values	page=1, limit=10 if not specified
Validation	Rejects negative or non-numeric values

Part 2: Property Statistics

Context:

Real estate managers need analytics: average prices, property counts by type, etc.

Your Task:

Create a statistics endpoint that:

- Returns count of properties by type
- Calculates average price per type
- Shows price range (min/max)
- Returns total properties count

Files to Modify:

- `src/controllers/propertyController.ts`
- `src/routes/propertyRoutes.ts`

Part 2: Definition of Done

Criteria	Description
Endpoint exists	`GET /api/properties/stats` returns data
Count by type	`{ house: 10, apartment: 15, ... }`
Average price	Average price per property type
Price range	`{ min: 50000, max: 2000000 }`
Total count	Total number of properties
Prisma aggregation	Uses `groupBy` and `aggregate`
Empty database	Returns zeros, not errors

Resources & Wrap-up

Resources

Express.js

- Express Documentation
- Express API Reference
- Express Middleware
- Error Handling in Express

Prisma

- Prisma Documentation
- Prisma Schema Reference
- Prisma Client API
- Prisma with Express Tutorial

Zod

Recommended Articles

API Design

- REST API Best Practices - Stack Overflow
- HTTP Status Codes Decision Diagram - Code Tinkerer
- API Versioning - freeCodeCamp

Prisma

- Prisma: The Complete ORM Guide - Prisma Blog
- Prisma vs Sequelize vs TypeORM - Prisma Docs
- Advanced Prisma Patterns - Prisma Blog