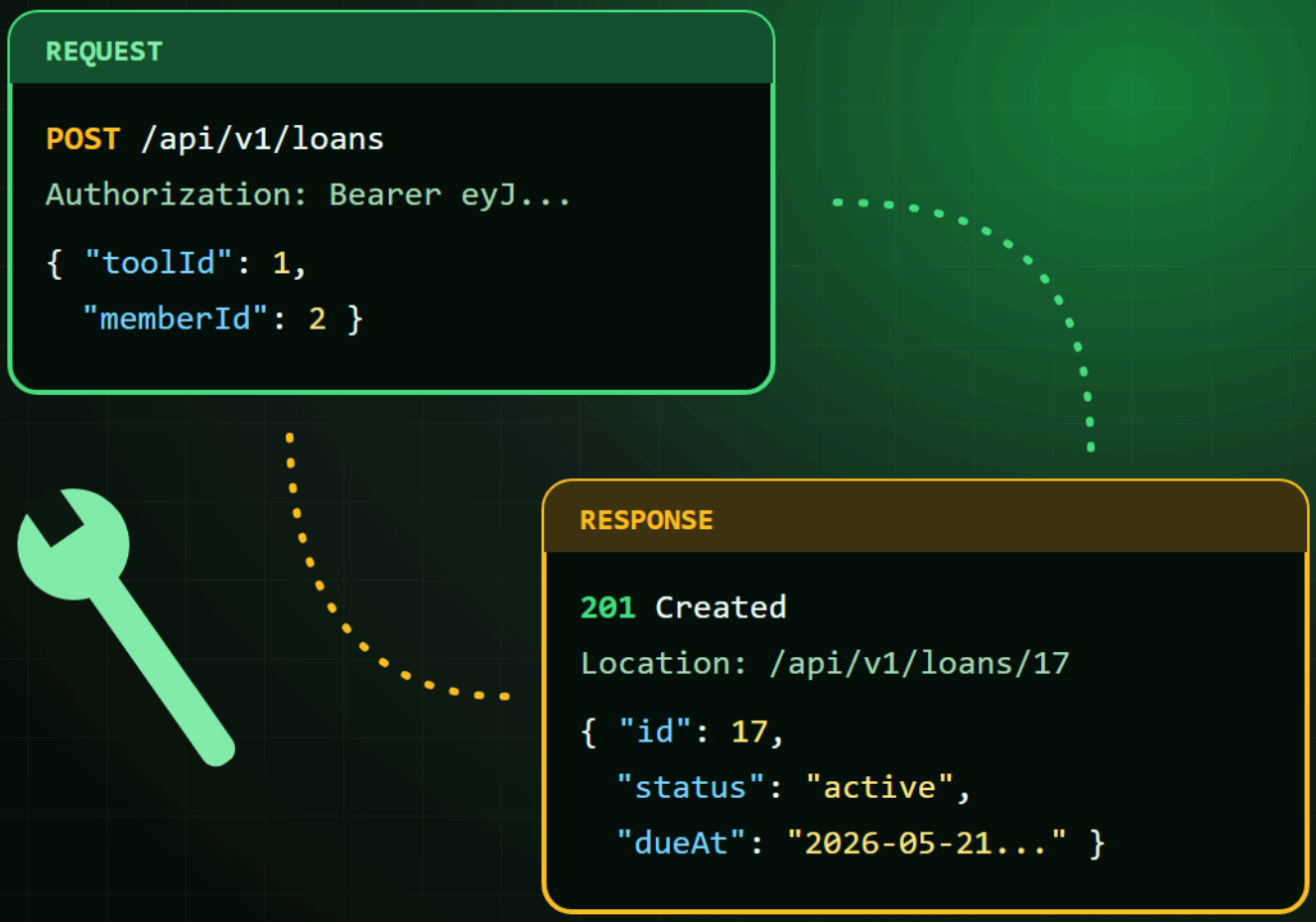


GET · POST · PATCH · DELETE · 200 · 201 · 409

A HANDS-ON GUIDE

Build REST APIs with Node.js

Design, build, secure, test, and deploy a real API from an empty folder

Express 5 · PostgreSQL · Zod · JWT · node:test
OpenAPI · Docker · Production readiness

ROUTES · DATA · SECURITY · SHIPPING

THENEXBOOKS

Build REST APIs with Node.js

A Hands-On Guide to Designing, Building, Securing, Testing, and Deploying a Real API With Express 5, PostgreSQL, Zod, and JWT

First Edition

THENEXBOOKS

Build REST APIs with Node.js
A Hands-On Guide to Designing, Building, Securing, Testing, and Deploying a Real API

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Node.js and the libraries discussed in this book evolve; always consult the official documentation for the versions you use.

Node.js is a trademark of the OpenJS Foundation. Express is a project of the OpenJS Foundation. PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. Docker is a trademark of Docker, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by the OpenJS Foundation or the maintainers of any library mentioned.

Lendshelf, the API built in this book, is fictional, as are its members and data. The domains `example.com` and `lendshelf.example` are used for illustration only.

Preface

Almost every modern application is really two applications: a client that people see, and an API that does the work behind it. Mobile apps, single-page web apps, partner integrations, internal dashboards, and increasingly AI agents all talk to servers through HTTP APIs. Learning to build a good one is one of the most useful skills a developer can have — and Node.js is one of the most popular places to do it.

The first endpoint is easy. A few lines of Express and your server answers with JSON. What comes next is where most tutorials stop and most real projects struggle: validating every input, returning consistent errors, storing data safely in a real database, handling two requests that race for the same record, logging people in without leaking passwords, protecting routes by role, accepting file uploads, testing everything, documenting the API, and running it in production where it restarts cleanly and tells you when something is wrong.

This book walks through all of it by building one complete API from an empty folder. Every chapter adds a real feature to the same project, explains *why* we do it that way, and shows the requests and responses so you can see the result. By the end you will have a working, tested, documented, deployed API — and the judgment to build your own.

Who This Book Is For

- **JavaScript developers** who have built front ends or small scripts and want to build the server side properly.
- **Beginners to back-end development** who want one clear path rather than a survey of every framework.
- **Developers from other languages** who need to become productive with Node.js and Express quickly.
- **Students and bootcamp graduates** who want a portfolio-quality project built with professional practices.

You need basic JavaScript: variables, functions, arrays, objects, arrow functions, and a first look at promises. Everything else is explained as it appears.

What You Will Learn

- How HTTP, JSON, and REST work, and how Node.js handles many requests at once.
- Building routes, middleware, and complete CRUD endpoints with Express 5.
- Validating input with Zod and returning clear, consistent error responses.
- Storing data in PostgreSQL with node-postgres, migrations, joins, and transactions.
- Pagination, filtering, sorting, and search that scale beyond a handful of rows.

- Password hashing, JSON Web Tokens, roles, ownership checks, CORS, rate limiting, and safe file uploads.
- Testing with node:test and Supertest, documenting with OpenAPI and Swagger UI, and structured logging.
- Caching and ETags, versioning, idempotency, background jobs, webhooks, Docker, deployment, and production readiness.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) explains HTTP and REST, sets up your tools, covers the Node.js you need, builds a first server by hand, and plans the project. **Part II — Building with Express** (Chapters 6–10) builds routes, middleware, CRUD endpoints, validation, and error handling. **Part III — Data and Persistence** (Chapters 11–15) connects the API to PostgreSQL. **Part IV — Security** (Chapters 16–19) adds authentication, authorization, hardening, and uploads. **Part V — Quality and Documentation** (Chapters 20–24) adds tests, documentation, logging, performance work, and safe evolution. **Part VI — Shipping** (Chapters 25–28) adds background work, deploys the API, makes it production-ready, and reviews the whole journey. The appendices provide a cheat sheet, templates, a launch checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Lendshelf

Throughout the book we build **Lendshelf**, the API behind a neighborhood tool-lending library. Neighbors sign up as members, browse a catalog of drills, ladders, pressure washers, and garden tools, and reserve what they need for the weekend. Librarians manage the catalog, check tools out, and check them back in. Late returns are flagged automatically, and members get reminder emails. It is small enough to understand completely and real enough to need everything a production API needs: validation, relationships, transactions that prevent double-lending, roles, uploads, pagination, tests, and deployment.

Conventions

Examples use a current Node.js LTS release (22 or later) with ES modules, Express 5, and PostgreSQL. Requests are shown with `curl`; where PowerShell needs different quoting, a Note shows how. JSON responses are formatted for readability.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake, a security risk, or something that can break in production.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `src/routes/tools.routes.js`, tell you where the code lives.

Contents

Preface

PART I — GETTING STARTED

1 What Is a REST API? HTTP, JSON, and the Web in Plain Words

2 Setting Up: Node.js, npm, an Editor, and Tools for Calling APIs

3 Node.js Essentials for API Developers: Modules, async/await, and the Event Loop

4 Your First Server with node:http

5 Planning the Lendshelf API: Resources, Endpoints, and Project Structure

PART II — BUILDING WITH EXPRESS

6 Express 5 Fundamentals: Routes, Requests, and Responses

7 Middleware: The Request Pipeline

8 CRUD Endpoints Done Right: Status Codes, Headers, and JSON

9 Validating Input with Zod

10 Error Handling and Consistent Error Responses

PART III — DATA AND PERSISTENCE

11 PostgreSQL Basics for API Developers

12 Connecting with node-postgres: Pools, Queries, and Parameters

13 Migrations and Seed Data

14 Relationships, Joins, and Transactions

15 Pagination, Filtering, Sorting, and Search

PART IV — SECURITY

16 Authentication: Password Hashing and JSON Web Tokens

17 Authorization: Roles, Ownership, and Protecting Routes

18 API Security Essentials: CORS, Helmet, Rate Limiting, and Safe Input

19 File Uploads: Images, Limits, and Storage

PART V — QUALITY AND DOCUMENTATION

20 Testing Your API with node:test and Supertest

21 Documenting Your API with OpenAPI and Swagger UI

22 Configuration, Environment Variables, and Structured Logging

23 Performance Basics: Caching, ETags, Compression, and Query Tuning

24 Evolving Your API: Versioning, Idempotency, and Concurrency

PART VI — SHIPPING

25 Background Work: Scheduled Jobs, Emails, and Webhooks

26 Deploying with Docker and a Cloud Platform

27 Production Readiness: Health Checks, Graceful Shutdown, and Monitoring

28 Case Study: Lendshelf from Empty Folder to Production

APPENDICES

A Cheat Sheet

B Templates and Reference

C API Launch Checklist

D Glossary

PART I

Getting Started

How HTTP, JSON, and REST fit together, a working Node.js setup, the language features every API developer relies on, a first server built by hand, and a plan for Lendshelf.

1 · What Is a REST API?
2 · Setting Up
3 · Node.js Essentials
4 · Your First Server with node:http
5 · Planning the Lendshelf API

CHAPTER 1 · PART I — GETTING STARTED

What Is a REST API? HTTP, JSON, and the Web in Plain Words

Before we write a single line of server code, we need a shared vocabulary. This chapter explains how programs talk to each other over the web, what an HTTP request and response really contain, what people mean when they say "REST," and what the finished Lendshelf API that we build in this book will look like.

Why APIs Are Everywhere

An **API** (application programming interface) is a contract that lets one piece of software ask another piece of software to do something. When you call `Math.max(3, 7)` in JavaScript, you are using an API: you know the function name, you know what arguments it accepts, and you know what it returns. You do not need to know how it is implemented inside the JavaScript engine.

A **web API** is the same idea stretched across a network. Instead of calling a function in the same process, a program sends a message over the internet to another program running on a different machine, and that program sends a message back. The weather widget on your phone, the "Sign in with..." button on a website, the checkout page of an online shop, and the dashboard your company uses to track orders all depend on web APIs. Most of the time the messages travel using **HTTP**, the same protocol your browser uses to load web pages, and most of the time the data inside those messages is written in **JSON**.

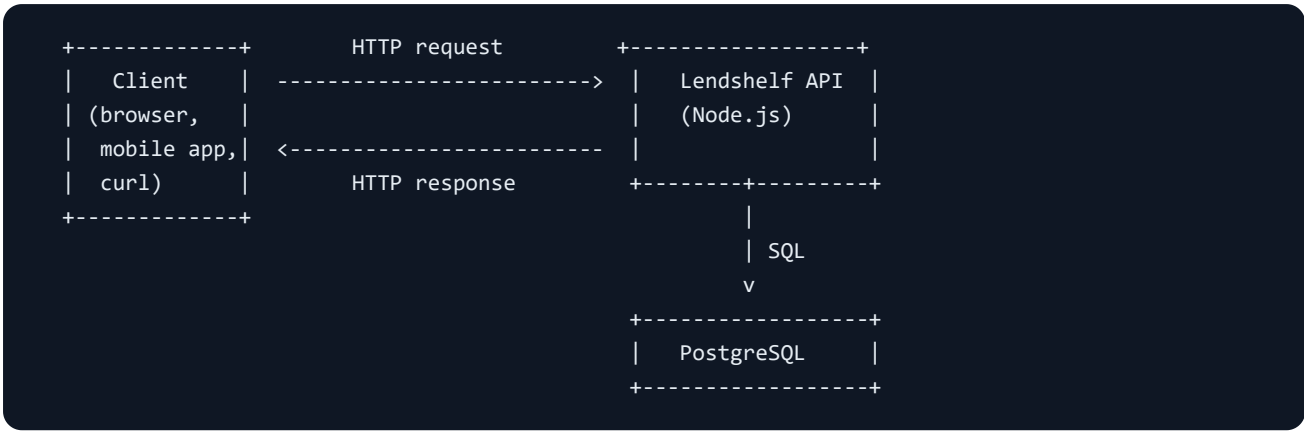
This book is about building the program on the other end of the wire: the one that receives requests, does real work, and replies. We will build it with Node.js, step by step, until it is tested, documented, secured, and deployed.

Our running example is **Lendshelf**, a fictional neighborhood tool-lending library. Neighbors register as members, browse a catalog of tools (drills, ladders, pressure washers), reserve and borrow them, and bring them back. Librarians manage the catalog and check tools out and in. Admins manage users. It is small enough to understand in an afternoon but rich enough to need everything a real API needs: authentication, roles, validation, relationships between data, pagination, file uploads, and careful error handling.

Clients and Servers

Every conversation on the web has two sides. The **client** starts the conversation by sending a **request**. The **server** listens for requests and answers each one with a **response**. The roles are defined by who asks and who answers, not by the kind of machine. A phone app is a client. A browser

is a client. A command-line tool like `curl` is a client. Even another server becomes a client when it calls someone else's API.



Notice that the client never talks to the database directly. The API sits in the middle and acts as a gatekeeper. It decides who may do what, it checks that incoming data makes sense, and it translates between the outside world (HTTP and JSON) and the inside world (SQL tables and business rules). That separation is the main reason APIs exist: you can change the database, rewrite internal logic, or add a new mobile app without breaking every other client, as long as the contract stays the same.

Clients and servers are also written independently. A Lendshelf mobile app might use Kotlin and a neighbor's script might use Python; none of them care that our server is JavaScript. They only care about the contract: which URLs exist, what they accept, and what they return.

The Anatomy of an HTTP Request

HTTP is a text-based protocol at heart. Modern versions (HTTP/2 and HTTP/3) encode messages in binary frames for efficiency, but the *meaning* of a message is the same in every version, and tools usually show it to you in the classic HTTP/1.1 text form. Here is a complete request that a client might send to Lendshelf to create a reservation:

Raw HTTP request

```
POST /api/v1/reservations HTTP/1.1
Host: localhost:3000
Content-Type: application/json
Accept: application/json
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIxIn0.abc123
Content-Length: 39

{"toolId": 1, "pickupOn": "2026-10-03"}
```

Every request has the same four parts:

1. **The request line.** It contains the **method** (`POST`), the **target** or path (`/api/v1/reservations`), and the protocol version. The method says what kind of action the client wants; the path says which thing the action applies to.

2. **Headers.** Each header is a `Name: value` pair that carries metadata about the request. `Host` names the server. `Content-Type` says what format the body is in. `Accept` says what format the client would like back. `Authorization` carries credentials. Header names are case-insensitive, so `content-type` and `Content-Type` mean the same thing.
3. **A blank line.** It marks the end of the headers.
4. **An optional body.** This is the actual payload: here, a small JSON document describing the reservation. `GET` requests normally have no body at all.

The path can also carry a **query string**, the part after the question mark. In `/api/v1/tools?category=ladders&page=2`, the path is `/api/v1/tools` and the query string contains two parameters, `category=ladders` and `page=2`. Query parameters are the conventional place for optional inputs that refine a read, such as filters, search terms, sorting, and pagination.

The Anatomy of an HTTP Response

The server's answer has the same overall shape, with a **status line** instead of a request line:

Raw HTTP response

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: /api/v1/reservations/17
Content-Length: 115

{"id":17,"toolId":1,"memberId":1,"pickupOn":"2026-10-03","status":"pending",
"createdAt":"2026-09-29T14:02:11.000Z"}
```

The status line contains a three-digit **status code** (`201`) and a short human-readable **reason phrase** (`Created`). The status code is the most important single piece of information in a response, because it is what client code checks first to decide what happened. Headers follow: `Content-Type` describes the body, and `Location` tells the client the URL of the newly created reservation. Then comes a blank line and the body.

A useful mental model: the status code answers "did it work?", the headers answer "how should you handle this message?", and the body holds the data. Good APIs keep all three consistent. Returning `200 OK` with a body of `{"error": "not found"}` forces every client to parse the body just to learn that the request failed.

HTTP Methods: The Verbs

HTTP defines a small set of methods. Each one has a well-understood meaning, and following those meanings is what lets generic tools (browsers, caches, proxies, client libraries, and other developers) make correct assumptions about your API.

Method	Typical use	Safe	Idempotent	Lendshelf example

GET	Read a resource or a list	Yes	Yes	GET /api/v1/tools/1
POST	Create, or trigger an action	No	No	POST /api/v1/loans
PUT	Replace a resource entirely	No	Yes	(not used in Lendshelf)
PATCH	Change some fields	No	Not guaranteed	PATCH /api/v1/tools/1
DELETE	Remove a resource	No	Yes	DELETE /api/v1/tools/1
HEAD	Like GET, headers only	Yes	Yes	HEAD /api/v1/tools
OPTIONS	Ask what is allowed	Yes	Yes	CORS preflight checks

Two words in that table deserve careful definitions because they come up again and again.

A method is **safe** if the client does not ask for any change on the server. `GET` is safe: reading the tool catalog should never modify it. That promise is why browsers happily prefetch links and why crawlers follow them. If you ever build a `GET` endpoint that deletes something, a search engine bot may eventually delete your data for you.

A method is **idempotent** if sending the same request once or ten times leaves the server in the same state. `DELETE /api/v1/tools/1` is idempotent: after the first call the tool is gone (in Lendshelf, marked inactive), and further calls do not make it "more gone." `POST /api/v1/reservations` is not idempotent: sending it twice could create two reservations. Idempotency matters because networks are unreliable. When a request times out, the client does not know whether the server processed it. For an idempotent request, the client can simply retry. For a non-idempotent one, it needs extra help, and in Chapter 24 we will add an `Idempotency-Key` header to reservations so that retries are safe.

> **NOTE**
>
> Safe and idempotent describe what the client *intends*, not everything the server does. Logging a `GET` request writes a line to a log file, but the request is still safe because the client did not ask for a change to the resource.

Status Codes: Did It Work?

Status codes are grouped into five families by their first digit. Once you know the families, you can make sense of codes you have never seen before.

Family	Meaning	Codes we use in Lendshelf
1xx	Informational, rarely seen directly	none
2xx	Success	200 OK, 201 Created, 204 No Content
3xx	Redirection or "use your cache"	304 Not Modified (Chapter 23)
4xx	The client made a mistake	400, 401, 403, 404, 409, 412, 429

5xx	The server failed	500 Internal Server Error, 503

The distinction between 4xx and 5xx is the one to internalize first. A 4xx code means "the request was wrong, and sending it again unchanged will fail again." A 5xx code means "something broke on our side," so retrying later might succeed.

Here are the codes you will meet most in this book, in plain words:

- **200 OK**: the request worked and the body contains the result.
- **201 Created**: a new resource was created; the `Location` header says where it lives.
- **204 No Content**: it worked and there is intentionally nothing to send back, typical for deletes.
- **400 Bad Request**: the input is malformed or invalid, such as a missing name or a negative deposit.
- **401 Unauthorized**: we do not know who you are. Despite the name, it is about *authentication*: the token is missing, expired, or invalid.
- **403 Forbidden**: we know who you are, but you are not allowed to do this, such as a member trying to add a tool to the catalog.
- **404 Not Found**: there is nothing at this URL, such as a tool ID that does not exist.
- **409 Conflict**: the request clashes with the current state, such as checking out a tool that is already on loan.
- **429 Too Many Requests**: the client is sending requests too quickly and should slow down.
- **500 Internal Server Error**: an unexpected failure on the server. Clients should never see the details, only that it happened.

> **TIP**
> When you are unsure which code to use, ask two questions. First: whose fault is it, the client's or the server's? That picks 4xx or 5xx. Second: what would a well-written client do differently for this case? If the answer is "nothing different," a more general code is fine. If the client should, for example, prompt the user to log in again, a specific code like 401 is worth using.

Headers That Matter

HTTP defines dozens of headers, but an API developer uses a small core set every day. Knowing what they do will save you from many confusing bugs.

Header	Direction	Purpose
Content-Type	Both	Format of the body, e.g. application/json
Accept	Request	Formats the client can handle
Authorization	Request	Credentials, e.g. Bearer token

Location	Response	URL of a newly created resource
Cache-Control	Both	Whether and how long to cache
ETag / If-None-Match	Both	Version tags for caching (Chapter 23)
If-Match	Request	Safe concurrent edits (Chapter 24)
Retry-After	Response	How long to wait after a 429 or 503
X-Request-Id	Both	Correlates logs for one request

The most common beginner bug in API work is a missing `Content-Type` on a request. If a client sends JSON without saying so, many servers (including Express with its standard JSON parser) will not parse the body, and your handler will see no data at all.

JSON in Plain Words

JSON (JavaScript Object Notation) is a text format for structured data. It borrows its syntax from JavaScript object literals, which makes it feel natural to JavaScript developers, but it is language-independent: every mainstream language can read and write it. Here is a Lendshelf tool as JSON:

A tool resource

```
{
  "id": 1,
  "name": "Cordless Drill 18V",
  "description": "Two batteries and a charger included.",
  "categoryId": 1,
  "condition": "good",
  "depositCents": 2000,
  "maxLoanDays": 7,
  "photoPath": null,
  "isActive": true,
  "version": 1,
  "createdAt": "2026-09-01T09:00:00.000Z",
  "updatedAt": "2026-09-01T09:00:00.000Z"
}
```

JSON has exactly six kinds of values: **strings** (always in double quotes), **numbers**, **booleans** (`true` and `false`), **null**, **arrays** (ordered lists in square brackets), and **objects** (key-value pairs in curly braces, with keys always in double quotes). That is the whole format. It is stricter than JavaScript: no single quotes, no trailing commas, no comments, no `undefined`, no functions, and no special date type.

That last point shapes a lot of API design. Because JSON has no date type, we send dates as strings in the ISO 8601 format, such as `"2026-09-01T09:00:00.000Z"` for a moment in time (the `Z` means UTC) or `"2026-10-03"` for a calendar date like a pickup day. Money has a similar problem: JSON numbers

are usually parsed as floating-point values, and floating-point arithmetic cannot represent many decimal fractions exactly. So Lendshelf stores deposits as whole numbers of cents. A deposit of twenty dollars is `"depositCents": 2000`, which is exact everywhere.

Notice also the key naming style. Lendshelf's JSON uses **camelCase** (`categoryId`, `maxLoanDays`), which is what JavaScript developers expect. The database underneath uses **snake_case** (`category_id`, `max_loan_days`), which is what SQL developers expect. The API translates between the two in one place, the repository layer, which we build in Chapter 12. Consistency matters more than the specific choice; the worst option is an API where some fields use one style and some use another.

> **WARNING**
> JSON is not JavaScript. Pasting a JavaScript object literal with single quotes or a trailing comma into a request body produces invalid JSON, and a well-behaved server will reject it with `400 Bad Request`. If a request mysteriously fails, validate the body with a JSON linter or with `JSON.parse()` in the Node.js REPL before suspecting the server.

What "REST" Means in Practice

REST stands for *Representational State Transfer*. The term comes from Roy Fielding's 2000 doctoral dissertation, which described the architectural style behind the web itself. The dissertation is precise and demanding; everyday usage is much looser. When developers today say "a REST API," they usually mean an HTTP API that follows a handful of practical conventions. This book takes that practical view, and we will point out where it matters.

Here are the ideas that carry real weight in day-to-day work:

Everything Is a Resource

A **resource** is any "thing" the API exposes that is worth naming: a tool, a member, a loan, a list of categories. Each resource has a URL that identifies it. `/api/v1/tools/1` identifies the cordless drill; `/api/v1/tools` identifies the collection of all tools. URLs are nouns, not verbs. You will not see `/api/v1/getTool?id=1` or `/api/v1/createLoan` in Lendshelf, because the method already supplies the verb.

Clients Work with Representations

The client never touches the resource directly; it receives a **representation** of it, a snapshot in some format. The JSON object you saw above is a representation of tool 1. The same resource could, in principle, also be represented as XML or CSV. The "representational state transfer" in the name refers to exactly this: clients and servers move representations of resource state back and forth.

A Uniform Interface

Every resource is manipulated through the same small set of methods with the same meanings. Once you know how to list tools, read one tool, create one, change one, and delete one, you already know how to do the same things with categories or loans. This uniformity is the real superpower of REST: it makes an API predictable, and predictable APIs need less documentation and produce fewer bugs.

Statelessness

Each request carries everything the server needs to understand it. The server does not remember "where you were" between requests. In Lendshelf, that means every protected request includes its own `Authorization` header; the server does not keep a login session in memory. Statelessness is what lets you run three copies of the API behind a load balancer and send each request to any of them.

Actions That Are Not CRUD

Some operations do not map neatly to "create, read, update, delete." Returning a tool sets a timestamp and changes a status, so Lendshelf uses `POST /api/v1/loans/7/return`: a clearly named action on a specific resource. Pragmatic REST allows such action endpoints when they make the API clearer; use them sparingly.

> **NOTE**
> Fielding's definition also includes *hypermedia*: responses containing links that tell the client what it can do next (search for "HATEOAS"). Few JSON APIs follow this strictly, and Lendshelf does not either.

Designing Resources and URLs

Good URLs are boring in the best way. They follow a few simple rules so that a developer can guess them correctly:

- Use plural nouns for collections: `/tools`, `/loans`, `/reservations`.
- Identify a single item by appending its ID: `/tools/3`.
- Use lowercase letters and hyphens for multi-word segments, and keep JSON field names separate from URL style.
- Put optional refinements in the query string: `/tools?category=garden&available=true`.
- Keep nesting shallow. `/tools/3/photo` is a clear sub-resource; four levels of nesting usually means a design problem.
- Put a version in the path, `/api/v1`, so that a future incompatible design can live at `/api/v2` without breaking old clients.

A special case is the "current user." Members frequently need their own profile, and making them look up their own numeric ID first would be clumsy. Lendshelf offers `GET /api/v1/members/me`, where `me` is resolved from the access token. This is a widely used convention and reads naturally.

When a list can be long, the API returns it one page at a time. Lendshelf wraps every list in an **envelope** that holds the items plus information about the page:

A list response

```
{
  "data": [
    { "id": 2, "name": "Extension Ladder 6m", "categoryId": 2, "maxLoanDays": 3 }
  ],
  "meta": { "page": 1, "limit": 20, "total": 1, "totalPages": 1 }
}
```

(Here and in other short examples we show only some fields to save space; the real API returns the full object for each item.) The envelope means clients never have to guess how many pages exist, and we can add more metadata later without breaking anything. A single resource, by contrast, is returned as the object itself, with no wrapper.

When Things Go Wrong: Error Responses

A status code alone rarely tells the whole story. A `400` could mean a missing name, a malformed date, or a deposit that is negative. Clients need details, and they need them in a consistent shape so that one piece of client code can handle every error. Lendshelf uses the standard format called **problem details**, defined in RFC 9457. Error responses use the content type `application/problem+json` and look like this:

A validation error

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{
  "type": "https://lendshelf.example/problems/validation-error",
  "title": "Validation failed",
  "status": 400,
  "detail": "The request body contains 2 invalid fields.",
  "errors": [
    { "path": "name", "message": "Required" },
    { "path": "depositCents", "message": "Must be greater than or equal to 0" }
  ]
}
```

The `type` is a URL that identifies the kind of problem; clients can switch on it reliably. The `title` is a short, fixed summary for that type. The `status` repeats the HTTP status for convenience. The `detail` explains this particular occurrence. Problem details allow extra fields, and Lendshelf adds an `errors`

array for validation failures so that a form can highlight each broken field. Lendshelf defines a small set of problem types: `validation-error` (400), `unauthorized` (401), `forbidden` (403), `not-found` (404), `conflict` (409), `rate-limited` (429), and `internal-error` (500). We build the machinery for all of them in Chapter 10.

One rule is absolute: error responses never include stack traces, SQL statements, or file paths. Those details belong in the server's logs, where developers can read them, and never in responses, where attackers can.

A Tour of the Finished Lendshelf API

Let's look at where we are headed. By the end of the book, Lendshelf exposes the following endpoints, all under the base path `/api/v1` except the operational ones:

Method	Path	Who	What it does
POST	/auth/register	Anyone	Create a member account
POST	/auth/login	Anyone	Exchange email and password for a token
GET	/members/me	Signed in	Your own profile
GET	/categories	Anyone	List tool categories
GET	/tools	Anyone	List, filter, search, sort tools
GET	/tools/:id	Anyone	One tool
POST	/tools	Librarian+	Add a tool
PATCH	/tools/:id	Librarian+	Change some fields
DELETE	/tools/:id	Librarian+	Retire a tool (soft delete)
POST	/tools/:id/photo	Librarian+	Upload a photo
POST	/loans	Librarian+	Check a tool out to a member
POST	/loans/:id/return	Librarian+	Check a tool back in
GET	/loans	Signed in	Your loans, or all for librarians
GET	/loans/:id	Signed in	One loan you may see
POST	/reservations	Member	Reserve a tool for pickup
DELETE	/reservations/:id	Owner	Cancel a reservation
GET	/health, /ready	Operations	Liveness and readiness checks
GET	/docs	Anyone	Interactive documentation

"Librarian+" means librarians and admins; permissions nest, so an admin can do everything a librarian can, and a librarian can do everything a member can. The seed data we load in Chapter 13 includes four fictional people: Ana Reyes and Jonah Price are members, Mei Tanaka is a librarian, and Sam Okoro is an admin. The catalog starts with five categories (Power tools, Ladders, Garden, Cleaning, Hand tools) and five tools, including a Cordless Drill 18V and an Extension Ladder 6m that can only be borrowed for three days.

Here is a short session against the finished API, as you will be able to run it yourself later. Don't worry about the `curl` flags yet; Chapter 2 covers them. First, Ana logs in:

Terminal

```
$ curl -s -X POST localhost:3000/api/v1/auth/login \
    -H "Content-Type: application/json" \
    -d '{"email": "ana@example.com", "password": "correct-horse-42"}'
{"accessToken":"eyJhbGciOiJIUzI1NiJ9...","tokenType":"Bearer","expiresIn":900}
```

The response contains an **access token**, a signed string that proves who Ana is. It expires in 900 seconds (fifteen minutes). Ana's app stores it and sends it with later requests. Next, she looks for ladders that are available right now:

Terminal

```
$ curl -s "localhost:3000/api/v1/tools?category=ladders&available=true"
{
  "data": [
    {
      "id": 2,
      "name": "Extension Ladder 6m",
      "categoryId": 2,
      "condition": "good",
      "depositCents": 3000,
      "maxLoanDays": 3,
      "isActive": true
    }
  ],
  "meta": { "page": 1, "limit": 20, "total": 1, "totalPages": 1 }
}
```

The URL is in quotes because the `&` character has a special meaning in shells. She reserves the ladder for Saturday, sending her token in the `Authorization` header. The `-i` flag asks curl to show the response status and headers too:

Terminal

```
$ TOKEN="eyJhbGciOiJIUzI1NiJ9..."
$ curl -si -X POST localhost:3000/api/v1/reservations \
    -H "Authorization: Bearer $TOKEN" \
    -H "Content-Type: application/json" \
    -d '{"toolId": 2, "pickupOn": "2026-10-03"}'
HTTP/1.1 201 Created
```