
BUILDING MICROSERVICES FULL-STACK

From Zero to Production

A PRACTICAL ENGINEERING GUIDE
Manazir Ali

Full-Stack Software Engineer

First Edition · v1.0.0 · February 2026

Building Microservices Full-Stack: From Zero to Production

First Edition

Copyright © 2026 Manazir Ali. 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 author.

Built with: TypeScript, Express.js, Next.js 15, React 19, PostgreSQL, Prisma, Docker, Kubernetes

Source code: github.com/mnzralee/fullstack-grocery

First Edition, February 2026

Contents

Preface

- What We're Building
- Who This Is For
- How to Read This Guide

Chapter 1: The Big Picture

- What You'll See When We're Done
- Layer 1: The Frontend (Next.js)
- Layer 2: The BFF (Next.js API Routes)
- Layer 3: The Backend Microservice (Express.js)
- Layer 4: The Database (PostgreSQL + Prisma)
- The Life of a Request
- Why Not Just Put Everything in One File?

Chapter 2: Setting Up the Workshop

- What You'll Have When We're Done
- Step 1: Initialize the Project
- Step 2: Create the Directory Structure
- Step 3: Install Dependencies
- Step 4: Set Up the Shared Database Package
- Step 5: Verify Your Setup

Chapter 3: The Database Foundation

- What You'll Have When We're Done
- Designing the Data Model
- Writing the Prisma Schema
- Applying the Schema

Seeding the Database

Chapter 4: The Domain Layer - Your Business Brain

What You'll Have When We're Done

The Base Error Class

Value Objects: The Money Type

Valid State Transitions

Testing the Domain Layer

Chapter 5: The Application Layer - Contracts and Rules

What You'll Have When We're Done

Defining the Repository Ports

Defining Request DTOs with Zod

Defining Response DTOs

Building Use Cases

Testing Use Cases

Chapter 6: The Infrastructure Layer - Making It Real

What You'll Have When We're Done

The Item Repository Implementation

The List Repository Implementation

Test It

Chapter 7: The Interface Layer - Talking to the World

What You'll Have When We're Done

The Validation Middleware

The Error Handler Middleware

The Controller

The Routes

The Life of an Error

Three Categories of Errors

The No-Fallback Philosophy

The Error Contract: What Each Layer Is Allowed To Do

Testing Error Paths

Common Anti-Patterns

Chapter 8: Wiring It All Together - The DI Container

What You'll See When We're Done

The Configuration

The DI Container

The App Factory

The Entry Point

Running It

Chapter 9: The Frontend - Pages and Components

What You'll See When We're Done

Initializing the Next.js App

Project Structure

Shared Types

The Root Layout

The Providers

Reusable UI Components

The Budget Bar

The Grocery Item Card

The Item List

The Add Item Form

The Home Page

Chapter 10: The BFF - Your Frontend's Private API

What You'll See When We're Done

The Backend Client

BFF API Routes

Chapter 11: Connecting It All - TanStack Query and Data Flow

The Client-Side API Client

Query Key Factory

Custom Hooks

The Main Page

The Moment of Truth

Chapter 12: Authentication - Locking the Doors

What You'll See When We're Done

Backend: The Auth Middleware

Backend: Role Guard

Backend: Update Routes with Auth

Backend: Login Endpoint

Frontend: Session Management

Frontend: The Login Page

Test It

Chapter 13: Dockerizing Your Microservice

What You'll See When We're Done

The Dockerfile

.dockerignore

Docker Compose for Local Development

Build and Run

Chapter 14: Deploying to Kubernetes

What You'll See When We're Done

Push the Image

Kubernetes Manifests

Deploy

Per-Developer Namespaces

Chapter 15: Putting It All Together - The Complete Picture

The Updated Life of a Request

How to Add a New Feature

When to Create a New Microservice

Chapter 16: What's Next - Growing as an Engineer

The Three Principles - Revisited

Advanced Patterns to Explore

The Reading List

Closing

Appendix A: Microservice Scaffold Template

Directory Structure

Boilerplate Files

New Service Checklist

Appendix B: Glossary

Appendix C: Document History

Preface

I believe in three principles. They thread through every chapter of this guide, and they thread through every line of code I write in production.

"Every line has a reason." If you can't explain why a line of code exists, delete it. No copy-paste without understanding. No magic. Every variable name, every function, every import: it's there for a reason. If it's not, it's dead weight.

"Read the error, trust the error." The error message is telling you exactly what's wrong. Read it. Read it fully. Don't guess. Don't open a browser and search before you've read every word the error gave you. Nine times out of ten, the answer is right there.

"Ship it, then perfect it." A running ugly prototype teaches you more than a perfect plan sitting in a document. Get something working. See it work. Then clean it up layer by layer. The code you're afraid to run is the code you don't understand yet.

These aren't rules from a textbook. They're lessons I learned the hard way: debugging production systems at 2 AM, untangling spaghetti code that "worked fine on my machine," and shipping features under pressure. I engineer a financial blockchain protocol. A system where a single bug in a transfer function can mean real money disappearing. That kind of pressure teaches you to be disciplined, methodical, and deliberate.

This guide is what I wish someone had handed me when I started. Not a textbook. Not documentation. A conversation. I'm going to sit next to you and we're going to build something together. I'll tell you what I do, why I do it, and what happens when you don't.

What We're Building

We're building a **Neighborhood Grocery List**. A full-stack application where families in a household can:

- Add grocery items they need (with name, quantity, estimated price, category)
- A household manager approves or rejects items before they're purchased
- When someone buys an approved item, the cost is deducted from a monthly budget
- Everyone can see the running budget, what's been spent, what's remaining

It's simple enough to understand in five minutes, but complex enough to teach you everything: state machines, role-based access, financial calculations, form validation, API design, database modeling, and deployment.

By the end of this guide, you'll have built a complete application that runs in a Kubernetes cluster. More importantly, you'll understand *why* every piece exists and *how* to build the next one on your own.

Who This Is For

You. Whether you're a CS graduate writing your first production code, or a mid-level developer joining a team that does things differently than what you're used to. I assume you know:

- JavaScript/TypeScript basics (variables, functions, async/await)
- What HTTP is (GET, POST, status codes)
- What a database is (tables, rows, queries)

If you know those three things, you're ready. Everything else (Clean Architecture, dependency injection, BFF pattern, Docker, Kubernetes) I'll teach you from scratch.

How to Read This Guide

Build as you go. Don't just read. Every chapter ends with something you can run and see. Open your terminal, open your editor, and build it alongside me. The code in each chapter builds directly on the previous one. By the end, you'll have a working application you built with your own hands.

Every line of code is a chance to create, learn, and grow. Little by little, we shape the future one line of code at a time.

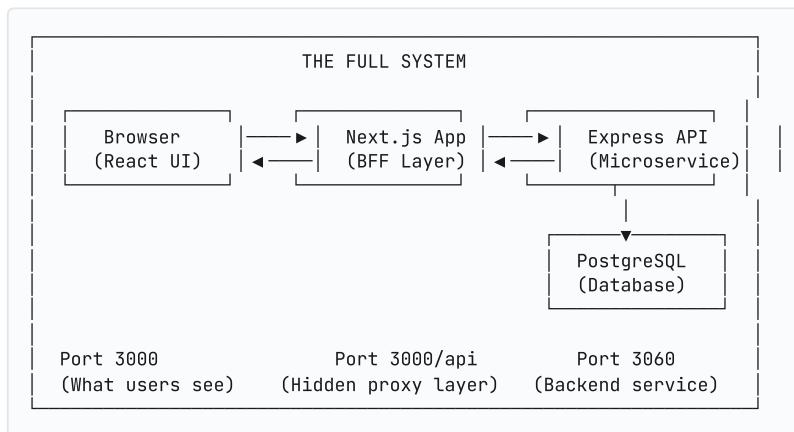
Let's begin.

CHAPTER 1

The Big Picture

"You just joined a team. The codebase has ten microservices, three frontend apps, and a blockchain network. Here's how to not be overwhelmed."

What You'll See When We're Done



That's it. Three layers. Let me tell you what each one does and *why* it exists.

Layer 1: The Frontend (Next.js)

This is what the user sees. Buttons, forms, lists. It runs in the browser. But here's the thing: the frontend doesn't talk directly to your backend microservice.

Why not? Because your backend might be running on port 3060 today, port 3070 tomorrow, and behind a load balancer next week. If the frontend knows your backend's address, you've created a fragile coupling. Change the backend, break the frontend.

Instead, the frontend talks to itself. Specifically, to API routes that live inside the same Next.js application. This is the **BFF pattern**: Backend For Frontend.

Layer 2: The BFF (Next.js API Routes)

These are server-side routes that live inside your Next.js app. They receive requests from the frontend, add authentication headers, validate the session, and forward the request to the correct backend microservice.

Think of the BFF as a receptionist. The visitor (browser) tells the receptionist (BFF) what they need. The receptionist checks their ID (session), picks up the phone, and calls the right department (microservice). The visitor never needs to know which department to call or what their extension number is.

Layer 3: The Backend Microservice (Express.js)

This is where the business logic lives. It receives HTTP requests, validates them, runs business rules, talks to the database, and returns responses. In a production system, you might have ten of these: one for authentication, one for user profiles, one for financial transactions, one for notifications.

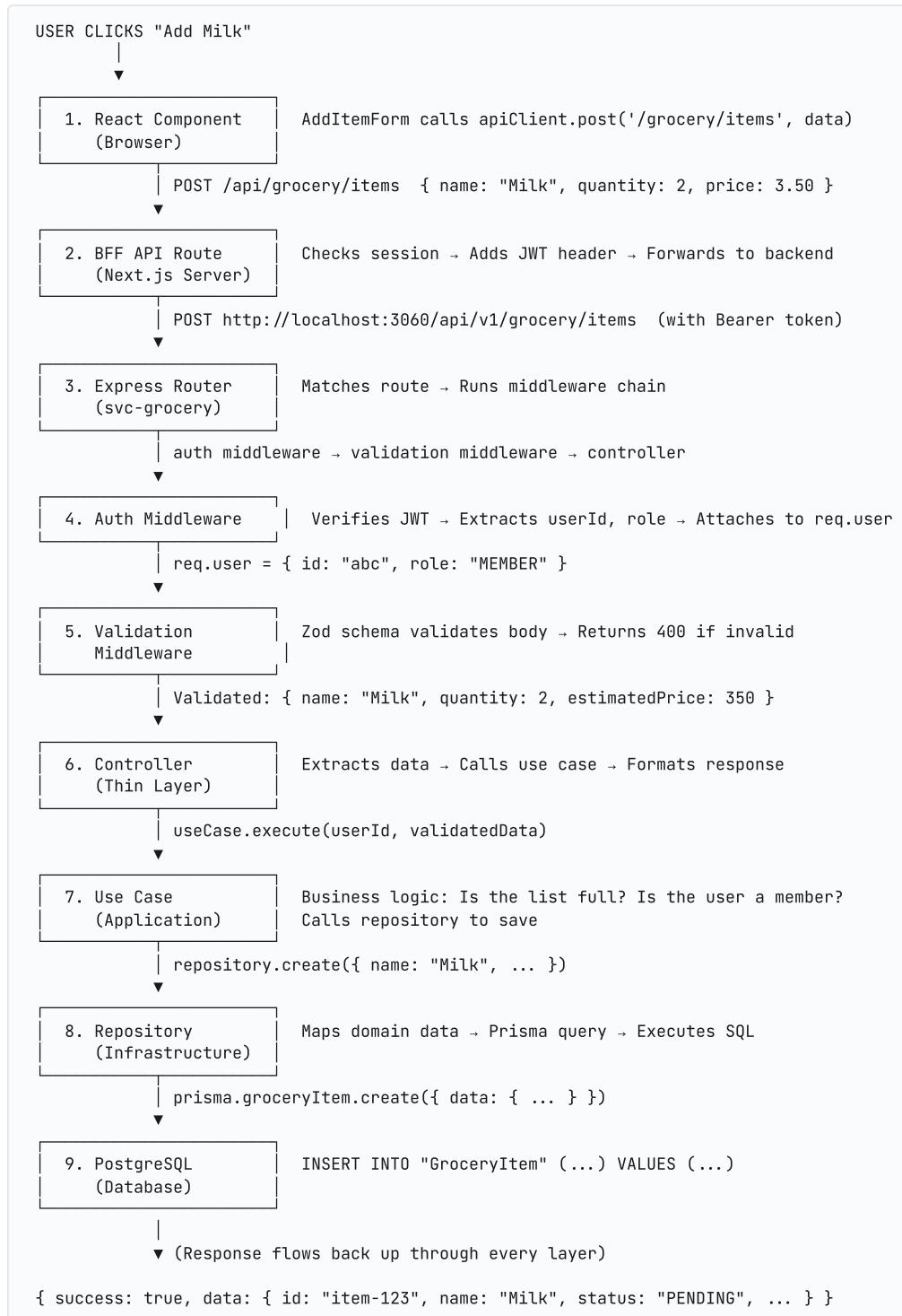
We're building one: `svc-grocery`. One microservice. But the patterns you'll learn apply to all of them.

Layer 4: The Database (PostgreSQL + Prisma)

PostgreSQL stores the data. Prisma is our bridge to it. It lets us write TypeScript instead of raw SQL, and it generates type-safe database queries. You'll never write `SELECT * FROM` in this project. Prisma handles that, and it catches your mistakes at compile time, not at runtime.

The Life of a Request

Let's trace what happens when a user clicks "Add Item" on the grocery list. This is the single most important diagram in this entire guide. If you understand this flow, you understand the whole system.



Nine steps. Each step has one job. Each step knows nothing about the steps above or below it. That's the whole point.

Why Not Just Put Everything in One File?

I want you to see what the alternative looks like. Here's the "naive approach": everything in a single Express route handler:

```
// The naive approach - DON'T do this
app.post('/api/items', async (req, res) => {
    // Auth check
    const token = req.headers.authorization?.split(' ')[1];
    const decoded = jwt.verify(token, 'secret');

    // Validation
    if (!req.body.name || !req.body.quantity) {
        return res.status(400).json({ error: 'Missing fields' });
    }

    // Business logic
    const list = await prisma.groceryList.findUnique({ where: { id: req.body.listId } });
    if (!list) return res.status(404).json({ error: 'List not found' });
    if (list.items.length ≥ 50) return res.status(400).json({ error: 'List full' });

    // Database
    const item = await prisma.groceryItem.create({
        data: {
            name: req.body.name,
            quantity: req.body.quantity,
            estimatedPrice: req.body.price * 100,
            status: 'PENDING',
            listId: list.id,
        },
    });

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

TYPESCRIPT

That's 25 lines. It works. Ship it, right?

Now imagine you have 30 endpoints like this. Each one has auth checks, validation, business logic, and database queries all tangled together. What happens when:

- You need to change how authentication works? You edit 30 files.
- You need to test the business logic without a database? You can't. The database query is on the same line as the business rule.
- A new developer joins and asks "where does the item limit check happen?" They search through 30 route handlers.

- You switch from PostgreSQL to MongoDB? You rewrite every endpoint.

This is what Robert C. Martin calls the **dependency problem** in *Clean Architecture*. When everything depends on everything, changing one thing changes everything. The solution is to separate concerns into layers, where each layer has a single responsibility and dependencies only point inward.

That's what we're building. The same 25 lines of logic, but organized so that each piece can be understood, tested, and modified independently.

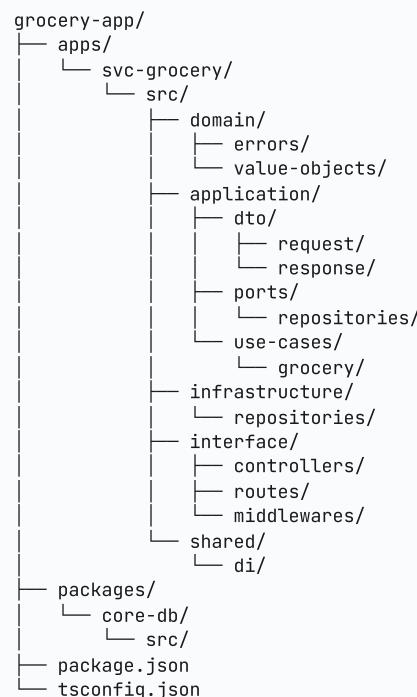
Let's start building.

CHAPTER 2

Setting Up the Workshop

"A craftsman's first job is organizing their workspace. You can't build a house with tools scattered across the floor."

What You'll Have When We're Done



Four layers. Domain. Application. Infrastructure. Interface. That's the skeleton of every microservice we'll ever build. Let me walk you through creating it.

Step 1: Initialize the Project

Open your terminal. Create a directory and initialize it:

```
mkdir grocery-app
cd grocery-app
npm init -y
```

BASH

That gives us a `package.json`. Now let's set up the **monorepo structure**. A monorepo means our backend microservice and our shared packages live in the same repository. Why? Because when you change a shared database package, you want to know immediately if it breaks your service. Not three days later when someone tries to deploy.

Open `package.json` and replace its contents:

```
{
  "name": "grocery-app",
  "private": true,
  "workspaces": [
    "apps/*",
    "packages/*"
  ]
}
```

JSON

The `workspaces` field tells npm: "There are multiple projects inside this repo. The ones in `apps/` are applications. The ones in `packages/` are shared libraries." This is how you structure a monorepo.

Step 2: Create the Directory Structure

```
mkdir -p apps/svc-grocery/src/{domain/{errors,value-objects},application/{dto/{reque
mkdir -p packages/core-db/src
```

BASH

That's one command, but let me explain what each directory is for:

| Directory | Purpose | Rule |
|---------------------------------|------------------------------------|---------------------------------|
| domain/errors/ | Business error definitions | NO external dependencies |
| domain/value-objects/ | Value types like Money, ItemStatus | NO external dependencies |
| application/dto/request/ | Input validation schemas (Zod) | Defines what comes IN |
| application/dto/response/ | Output data shapes | Defines what goes OUT |
| application/ports/repositories/ | Repository interfaces | Contracts, not implementations |
| application/use-cases/grocery/ | Business logic handlers | Orchestrates domain rules |
| infrastructure/repositories/ | Database implementations | Talks to Prisma/PostgreSQL |
| interface/controllers/ | HTTP request handlers | Thin, delegates to use cases |
| interface/routes/ | Express route definitions | Wires middleware + controllers |
| interface/middlewares/ | Auth, validation, error handling | Cross-cutting concerns |
| shared/di/ | Dependency injection container | Wires everything together |
| packages/core-db/ | Shared Prisma client | One database connection, shared |

Step 3: Install Dependencies

First, let's set up TypeScript for the whole monorepo. Create `tsconfig.json` in the root:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "commonjs",
    "lib": ["ES2022"],
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true,
    "outDir": "./dist",
    "baseUrl": ".",
    "paths": {
      "@grocery/core-db": ["packages/core-db/src"]
    }
  },
  "exclude": ["node_modules", "dist"]
}
```

JSON

I want to point out one setting: `"strict": true`. This enables TypeScript's strict mode. It means TypeScript will yell at you for sloppy code: implicit `any` types, unchecked nulls, missing return types. It's annoying at first. It will save your life in production. Every bug strict mode catches at compile time is a bug you don't debug at 2 AM.

Now install the backend dependencies:

```
cd apps/svc-grocery
npm init -y
npm install express zod jsonwebtoken bcryptjs pino
npm install -D typescript @types/express @types/jsonwebtoken @types/bcryptjs tsx
```

BASH

Let me explain each one:

| Package | Why |
|---------------------------|---|
| <code>express</code> | HTTP server framework |
| <code>zod</code> | Runtime input validation (schemas that generate types) |
| <code>jsonwebtoken</code> | JWT token verification (for auth, later) |
| <code>bcryptjs</code> | Password hashing (for auth, later) |
| <code>pino</code> | Structured logging (JSON logs, not <code>console.log</code>) |
| <code>typescript</code> | TypeScript compiler |
| <code>tsx</code> | Run TypeScript directly without compiling (for development) |

Create the service's `tsconfig.json` at `apps/svc-grocery/tsconfig.json`:

```
{
  "extends": "../../tsconfig.json",
  "compilerOptions": {
    "outDir": "./dist",
    "rootDir": "./src",
    "paths": {
      "@grocery/core-db": ["../../packages/core-db/src"]
    }
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist", "src/**/_tests_/*"]
}
```

JSON

We re-declare the `paths` alias at the service level so TypeScript resolves `@grocery/core-db` correctly from within this project. The `exclude` keeps test files out of the production `dist/` build; test code should never ship.

And add a dev script to `apps/svc-grocery/package.json`:

```
{
  "scripts": {
    "dev": "tsx watch src/index.ts",
    "build": "tsc",
    "start": "node dist/index.js"
  }
}
```

JSON

`tsx watch` means: "Run my TypeScript file, and restart automatically when I save changes." During development, this is your feedback loop. Save a file, see the change immediately.

Step 4: Set Up the Shared Database Package

```
cd ../../packages/core-db
npm init -y
npm install prisma @prisma/client
```

BASH

Now open `packages/core-db/package.json` and set the package name and entry point. This is critical: without the correct name, the workspace alias `@grocery/core-db` won't resolve, and your backend build will fail with "Cannot find module":

```
{
  "name": "@grocery/core-db",
  "main": "src/index.ts",
  "dependencies": {
    "prisma": "...",
    "@prisma/client": "..."
  }
}
```

JSON

The `"name"` must match exactly what we put in the root `tsconfig.json` paths and what the service will import. The `"main"` tells Node.js where to find the exports.

Create `packages/core-db/src/index.ts`. This file is small but critical:

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

TYPESCRIPT

That's one line. We're importing the Prisma client. Now let's add the singleton pattern:

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

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined;
};
```

TYPESCRIPT

Why this weird `globalThis` cast? Because in development, when your server restarts on every file save, Node.js creates a new `PrismaClient` each time. Each client opens a new database connection pool. After 20 saves, you have 20 connection pools and PostgreSQL starts rejecting connections. The `globalThis` trick stores the client globally so it survives restarts.

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

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined;
};

export const db = globalForPrisma.prisma ?? new PrismaClient();

if (process.env.NODE_ENV !== 'production') {
  globalForPrisma.prisma = db;
}
```

TYPESCRIPT

Read that line: `globalForPrisma.prisma ?? new PrismaClient()`. It says: "If a Prisma client already exists globally, use it. Otherwise, create a new one." In production, we always create a fresh one. In development, we reuse the existing one.

Finally, re-export everything from Prisma so other packages don't need to import from `@prisma/client` directly:

```
export * from '@prisma/client';
```

TYPESCRIPT

The complete file:

```
// packages/core-db/src/index.ts

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

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined;
};

export const db = globalForPrisma.prisma ?? new PrismaClient();

if (process.env.NODE_ENV !== 'production') {
  globalForPrisma.prisma = db;
}

export * from '@prisma/client';
```

TYPESCRIPT

Seven lines of actual logic. That's your database connection for the entire application.

Step 5: Verify Your Setup

At this point, your project should look exactly like the tree diagram at the top of this chapter. Run a quick check:

```
cd ../../ # Back to grocery-app root  
find . -type d -not -path '*/node_modules/*' -not -path '*/.git/*' | sort
```

BASH

You should see every directory we created. If something's missing, create it now. We're done setting up. Every chapter from here forward is about writing code inside this structure.

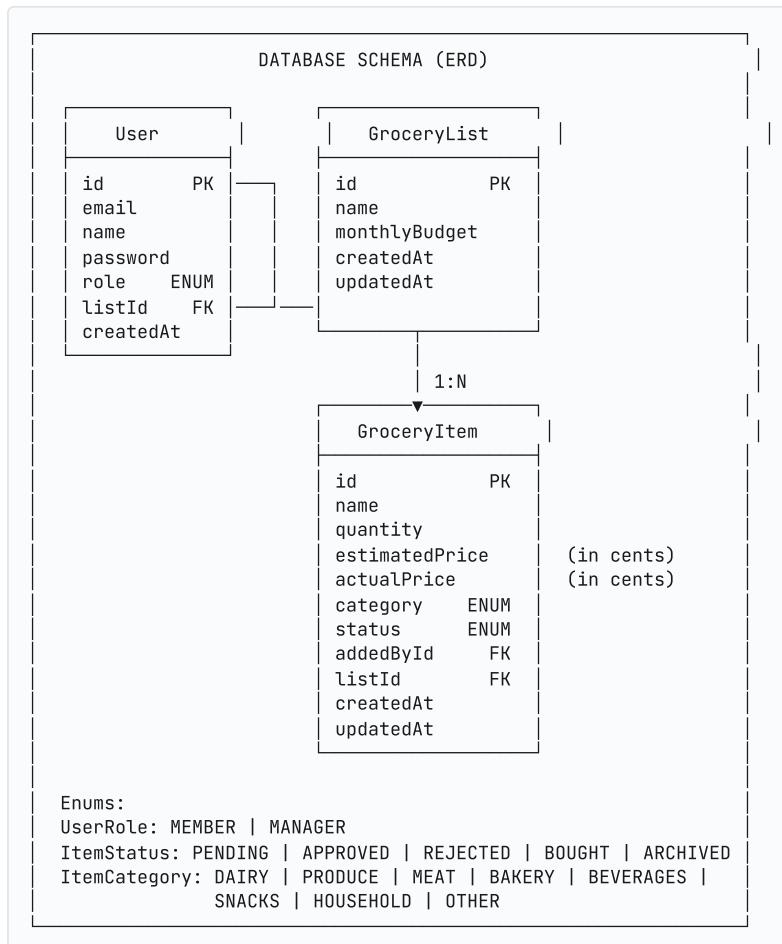
Principle check: *Every line has a reason.* The monorepo structure exists because shared packages prevent code duplication. TypeScript strict mode exists because it catches bugs at compile time. The Prisma singleton exists because connection pool exhaustion is a real production bug. Nothing here is decoration.

CHAPTER 3

The Database Foundation

"Before you build a house, you pour the foundation. Before you write business logic, you design your data."

What You'll Have When We're Done



Designing the Data Model

Before writing any schema code, let me talk about the design decisions. These matter more than the syntax.

Money is stored in cents, not dollars. This is non-negotiable. If milk costs \$3.50, we store 350. Why? Because floating-point arithmetic is broken for money. Try this in any JavaScript console:

```
0.1 + 0.2 // 0.3000000000000004
```

JAVASCRIPT

That's not a bug. It's how IEEE 754 floating-point works. In a grocery list, a one-cent rounding error is annoying. In a financial system, it's a lawsuit. Store money as integers (cents), do all math in integers, and only convert to dollars for display. Martin Kleppmann

covers this beautifully in *Designing Data-Intensive Applications*. Data representation matters more than most engineers think.

IDs are UUIDs, not auto-incrementing integers. Why? Because `item/47` tells an attacker there are at least 46 other items, and they can try `/item/1` through `/item/46`. A UUID like `a1b2c3d4-e5f6-...` reveals nothing. It's also easier to generate IDs without a database round-trip, which matters in distributed systems.

Every table has `createdAt` and `updatedAt`. Always. You will need them for debugging. "When did this item's status change?" is a question you'll ask at 2 AM. Have the answer ready.

Writing the Prisma Schema

Initialize Prisma in the core-db package:

```
cd packages/core-db
npx prisma init
```

BASH

This creates a `prisma/` directory with a `schema.prisma` file. But we're going to do something more organized. We'll use **multi-file schemas**. Instead of one giant file, we split the schema by domain.

Create the directory structure:

```
mkdir -p prisma/schema
```

BASH

First, the configuration file. Create `prisma/schema/config.prisma`:

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

PRISMA

Two things here. The `generator` tells Prisma to generate a TypeScript client. The `datasource` tells it we're using PostgreSQL, and the connection URL comes from an environment variable. Never hardcoded. (Remember: **every line has a reason.**)

We're using multi-file schemas. Prisma reads every `.prisma` file in the `schema/` directory automatically. This became stable in Prisma 5.15 and is the default in Prisma 6.x, so if you're on `prisma@^5.15.0` or later, it just works. If you're on an earlier 5.x version, add `previewFeatures = ["prismaSchemaFolder"]` to the generator block. Check your version and move on.

Now let's define the enums. Create `prisma/schema/enums.prisma`:

```
enum UserRole {
    MEMBER
    MANAGER
}
```

PRISMA

Two roles. `MEMBER` can add items to the list. `MANAGER` can approve or reject them. Simple role-based access control. We'll put both enums in one file. Let me add the next one:

```
enum UserRole {
    MEMBER
    MANAGER
}

enum ItemStatus {
    PENDING
    APPROVED
    REJECTED
    BOUGHT
    ARCHIVED
}
```

PRISMA

Five statuses. This is a **state machine**. An item moves through these statuses in a specific order:



A `PENDING` item can be `APPROVED` or `REJECTED`. An `APPROVED` item can be `BOUGHT`. A `BOUGHT` item can be `ARCHIVED` (for history). No other transitions are valid. You can't go from `REJECTED` to `BOUGHT`. You can't go from `BOUGHT` back to `PENDING`. We'll enforce this in the domain layer later.

Now add the category enum:

```

enum UserRole {
    MEMBER
    MANAGER
}

enum ItemStatus {
    PENDING
    APPROVED
    REJECTED
    BOUGHT
    ARCHIVED
}

enum ItemCategory {
    DAIRY
    PRODUCE
    MEAT
    BAKERY
    BEVERAGES
    SNACKS
    HOUSEHOLD
    OTHER
}

```

PRISMA

Now the models. Create `prisma/schema/models.prisma`:

```

model User {
    id      String  @id @default(uuid())
    email   String  @unique
    name    String
    password String
    role    UserRole @default(MEMBER)
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt
}

```

PRISMA

Let me break down every annotation:

- `@id` : This field is the primary key
- `@default(uuid())` : Prisma generates a UUID automatically when creating a row
- `@unique` : No two users can have the same email
- `@default(MEMBER)` : New users are members by default
- `@default(now())` : Set to the current timestamp on creation
- `@updatedAt` : Prisma automatically updates this on every modification

Now add the relationship. A user belongs to a grocery list:

```
model User {
    id      String  @id @default(uuid())
    email   String  @unique
    name    String
    password String
    role    UserRole @default(MEMBER)
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt

    listId String?
    list   GroceryList? @relation("ListMembers", fields: [listId], references: [id])

    addedItems GroceryItem[] @relation("AddedBy")

    @@index([email])
    @@index([listId])
}
```

`listId` is nullable (`String?`) because a user might not belong to any list yet. The `@relation` annotation tells Prisma how to join the tables. `addedItems` is the reverse relation: all items this user added. The `@@index` lines create database indexes for fast lookups on `email` and `listId`.

Now the grocery list model:

```
model GroceryList {
    id      String  @id @default(uuid())
    name   String
    monthlyBudget Int      @default(0)
    createdAt DateTime @default(now())
    updatedAt DateTime @updatedAt

    items   GroceryItem[] @relation("ListItems")
    users   User[]       @relation("ListMembers")
}
```

`monthlyBudget` is an `Int`, stored in cents. A \$500 budget is stored as `50000`. No floating point.

And the grocery item model:

```

model GroceryItem {
    id          String      @id @default(uuid())
    name        String
    quantity    Int         @default(1)
    estimatedPrice Int
    actualPrice Int?
    category    ItemCategory @default(OTHER)
    status      ItemStatus   @default(PENDING)
    createdAt   DateTime    @default(now())
    updatedAt   DateTime    @updatedAt

    listId     String
    list       GroceryList @relation("ListItems", fields: [listId], references: [id], onDelete: Cascade)

    addedById  String
    addedBy    User @relation("AddedBy", fields: [addedById], references: [id])

    @@index([listId, status])
    @@index([addedById])
}

```

Notice `estimatedPrice Int` (required) versus `actualPrice Int?` (nullable). When you add an item, you estimate the price. When you buy it, you enter the actual price. The `@@index([listId, status])` is a **composite index**, PostgreSQL can quickly find "all PENDING items in list X" using a single index scan. This is the kind of optimization that matters when you have thousands of items.

`onDelete: Cascade` means: if a grocery list is deleted, all its items are deleted too. No orphaned rows.

Applying the Schema

Create a `.env` file in `packages/core-db/`:

```
BASH
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/grocery_db?schema=public"
```

Make sure PostgreSQL is running (Docker is the easiest way):

```
docker run -d \
  --name grocery-postgres \
  -e POSTGRES_USER=postgres \
  -e POSTGRES_PASSWORD=postgres \
  -e POSTGRES_DB=grocery_db \
  -p 5432:5432 \
  postgres:16
```

BASH

Now, let's talk about the two ways to apply your schema.

For development: `prisma db push`

```
npx prisma db push
```

BASH

This is fast. It reads your schema, compares it to the database, and applies the changes directly. No migration files, no history. It's perfect for rapid development when you're still figuring out your schema. But it's dangerous in production. It can drop columns and lose data without warning.

For production: `prisma migrate dev`

```
npx prisma migrate dev --name init
```

BASH

This creates a migration file: a SQL script that describes exactly what changed. The file is committed to git, reviewed in PRs, and applied in order on every environment. It's slower, more deliberate, and safe. You can see exactly what SQL will run before it runs.

For this guide, we'll use `db push` while developing and I'll show you migrations when we deploy.

After running `db push`, verify it worked:

```
npx prisma studio
```

BASH

This opens a web UI at `http://localhost:5555` where you can see your tables. You should see three tables: `User`, `GroceryList`, and `GroceryItem`. All empty. That's our foundation.

Seeding the Database

An empty database is useless for development. Let's create a seed script that fills it with test data. Create `prisma/seed.ts`:

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

const prisma = new PrismaClient();
```

TYPESCRIPT

We import Prisma and create a client. Now let's write the seed function:

```
async function seed() {
  console.log('Seeding database...');

  // Clean slate
  await prisma.groceryItem.deleteMany();
  await prisma.user.deleteMany();
  await prisma.groceryList.deleteMany();
```

TYPESCRIPT

We delete everything first. Order matters: items reference lists and users, so delete items first.

```
// Create a household grocery list
const list = await prisma.groceryList.create({
  data: {
    name: 'Ali Family Groceries',
    monthlyBudget: 50000, // $500.00 in cents
  },
});

console.log(`Created list: ${list.name} (budget: ${list.monthlyBudget / 100})`);
console.log(` List ID: ${list.id} ← You'll need this for the frontend URL`);
```

TYPESCRIPT

One list. \$500 monthly budget. Notice how I log the budget divided by 100. That's cents to dollars for human readability. The database stores `50000`, humans see `$500.00`.

TYPESCRIPT

```
// Create users with bcrypt-hashed passwords
const bcryptModule = await import('bcryptjs');
const bcrypt = bcryptModule.default ?? bcryptModule;
const hashedPassword = await bcrypt.hash('grocery123', 12);

const manager = await prisma.user.create({
  data: {
    email: 'manager@grocery.test',
    name: 'Sarah (Manager)',
    password: hashedPassword,
    role: 'MANAGER',
    listId: list.id,
  },
});

const member = await prisma.user.create({
  data: {
    email: 'member@grocery.test',
    name: 'Alex (Member)',
    password: hashedPassword,
    role: 'MEMBER',
    listId: list.id,
  },
});

console.log(`Created users: ${manager.name}, ${member.name}`);
```

Two users. One manager, one member. Both have the password `grocery123`, hashed with `bcrypt` (12 rounds). The seed script hashes at runtime so the hash is always valid.

Notice the `bcryptModule.default ?? bcryptModule` pattern. When you dynamically `import()` a CommonJS package like `bcryptjs` from an ESM context, the actual functions end up on `.default` instead of at the top level. If you just write `bcryptModule.hash(...)`, it's `undefined`. The `??` fallback handles both cases (ESM wrapping and plain CJS) so the seed works regardless of your module resolution.

TYPESCRIPT

```
// Create some grocery items
const items = await prisma.groceryItem.createMany({
  data: [
    {
      name: 'Whole Milk',
      quantity: 2,
      estimatedPrice: 350, // $3.50
      category: 'DAIRY',
    },
  ],
});
```

```

        status: 'PENDING',
        listId: list.id,
        addedById: member.id,
    },
    {
        name: 'Sourdough Bread',
        quantity: 1,
        estimatedPrice: 450,    // $4.50
        category: 'BAKERY',
        status: 'APPROVED',
        listId: list.id,
        addedById: member.id,
    },
    {
        name: 'Chicken Breast',
        quantity: 2,
        estimatedPrice: 899,    // $8.99
        category: 'MEAT',
        status: 'BOUGHT',
        actualPrice: 799,       // $7.99 (was on sale!)
        listId: list.id,
        addedById: member.id,
    },
],
});

console.log(`Created ${items.count} grocery items`);
console.log('Seed complete!');
}

```

Three items in different states. Milk is PENDING (waiting for approval). Bread is APPROVED (ready to buy). Chicken has been BOUGHT (with an actual price less than estimated; it was on sale).

Now add the execution and cleanup:

```

seed()
  .catch((error) => {
    console.error('Seed failed:', error);
    process.exit(1);
  })
  .finally(async () => {
    await prisma.$disconnect();
  });
}

```

TYPESCRIPT

Always disconnect the Prisma client when you're done. Connection leaks are real.

Add the seed command to `packages/core-db/package.json`:

```
{
  "prisma": {
    "schema": "prisma/schema",
    "seed": "tsx prisma/seed.ts"
  }
}
```

JSON

The `"schema": "prisma/schema"` tells the Prisma CLI where to find your schema files. Without it, Prisma looks for a single `prisma/schema.prisma` file, won't find your multi-file setup, and `prisma db push` will fail.

Run it:

```
npx prisma db seed
```

BASH

Now open Prisma Studio again (`npx prisma studio`), and you'll see real data. Click on `GroceryItem`. You'll see milk, bread, and chicken with their statuses and prices.

Principle check: *Ship it, then perfect it.* We have a running database with seed data. We can see it, touch it, query it. The schema might change as we build features, and that's fine. `db push` applies changes instantly. We didn't spend three hours designing the perfect schema. We built one that works and we'll refine it.

CHAPTER 4

The Domain Layer - Your Business Brain

"The domain layer is the heart of your application. It knows the rules of your business and nothing else. It doesn't know what a database is. It doesn't know what HTTP is. It knows that a PENDING item can be APPROVED or REJECTED, and a budget can't go negative."

What You'll Have When We're Done

When something goes wrong, your API will return this:

```
{
  "success": false,
  "error": {
    "code": "INVALID_ITEM_STATE",
    "message": "Cannot approve item: current status is BOUGHT, cannot transition to APPROVED",
    "timestamp": "2026-02-23T10:30:00.000Z",
    "details": {
      "currentStatus": "BOUGHT",
      "targetStatus": "APPROVED",
      "operation": "approve"
    }
  }
}
```

JSON

That's not a generic "Bad Request." That's a precise, debuggable, machine-readable error that tells the frontend exactly what went wrong, tells the developer exactly where to look, and tells the user exactly what they did wrong. This is what **fail fast, fail loud** looks like.

The Base Error Class

Every domain error in our system extends a single base class. Create `apps/svc-grocery/src/domain/errors/index.ts`:

```
export abstract class DomainError extends Error {
```

TYPESCRIPT

It extends JavaScript's built-in `Error`. The `abstract` keyword means you can't create a `DomainError` directly. You have to create a specific one like `ItemNotFoundError`. This forces specificity.

```
export abstract class DomainError extends Error {
    public readonly code: string;
    public readonly statusCode: number;
    public readonly timestamp: string;
    public readonly details?: Record<string, unknown>;
```

TYPESCRIPT

Four fields:

- `code` : Machine-readable error code like `"ITEM_NOT_FOUND"` . The frontend can switch on this.
- `statusCode` : HTTP status code. Yes, I know I said the domain doesn't know about HTTP. This is a pragmatic compromise. The error handler in the interface layer reads this to set the HTTP response code. The domain defines the *severity* (404 = not found, 400 = bad input, 403 = forbidden), and the interface layer translates it to HTTP. It's a small coupling that saves a lot of mapping code.
- `timestamp` : When the error occurred. Essential for debugging.
- `details` : Extra context. Structured, not a string. Machine-readable.

Now the constructor:

```

constructor(
  code: string,
  statusCode: number,
  message: string,
  details?: Record<string, unknown>,
) {
  super(message);
  this.name = this.constructor.name;
  this.code = code;
  this.statusCode = statusCode;
  this.timestamp = new Date().toISOString();
  this.details = details;

  Error.captureStackTrace(this, this.constructor);
}
}

```

TYPESCRIPT

`Error.captureStackTrace` cleans up the stack trace so it starts from where the error was thrown, not from inside the `DomainError` constructor. Small detail, but it makes debugging significantly easier. This is a V8 API. It works in Node.js, Deno, and Chrome because they all run on V8. If you ever port this to Bun (which uses JavaScriptCore instead), guard it with `if (Error.captureStackTrace)`, or it'll throw.

Now let's build the specific errors. Still in the same file:

```

export class ItemNotFoundError extends DomainError {
  constructor(itemId: string) {
    super(
      'ITEM_NOT_FOUND',
      404,
      `Grocery item not found: ${itemId}`,
      { itemId },
    );
  }
}

```

TYPESCRIPT

One error, one class. The constructor takes only what's specific to this error (the item ID) and passes everything else to the base class. When you throw `new ItemNotFoundError('abc-123')`, the error message, code, status code, and details are all generated automatically.

```
export class ListNotFoundError extends DomainError {
  constructor(listId: string) {
    super(
      'LIST_NOT_FOUND',
      404,
      `Grocery list not found: ${listId}`,
      { listId },
    );
  }
}
```

TYPESCRIPT

Same pattern. Let me keep going:

```
export class InvalidItemStateException extends DomainError {
  constructor(currentStatus: string, targetStatus: string, operation: string) {
    super(
      'INVALID_ITEM_STATE',
      400,
      `Cannot ${operation} item: current status is ${currentStatus}, cannot transition to ${targetStatus}.`,
      { currentStatus, targetStatus, operation },
    );
  }
}
```

TYPESCRIPT

This is the state machine enforcement error. When someone tries to approve a BOUGHT item, this error fires with the exact context: what the status is now, what status was attempted, and what operation triggered it.

```
export class BudgetExceededError extends DomainError {
  constructor(requested: number, available: number) {
    super(
      'BUDGET_EXCEEDED',
      400,
      `Budget exceeded: requested ${requested} cents, only ${available} cents available.`,
      { requestedCents: requested, availableCents: available },
    );
  }
}
```

TYPESCRIPT

Financial error. Amounts in cents. The details include both the requested amount and the available amount so the frontend can show "You're \$12.50 over budget."

```
export class InsufficientPermissionsError extends DomainError {
  constructor(requiredRole: string, operation: string) {
    super(
      'INSUFFICIENT_PERMISSIONS',
      403,
      `Insufficient permissions: ${operation} requires ${requiredRole} role`,
      { requiredRole, operation },
    );
  }
}
```

TYPESCRIPT

Role-based access error. When a MEMBER tries to approve an item, this tells them exactly what role they need.

Now you might be tempted to add a generic `ValidationError` that catches everything. Don't. Every error should be specific. `InvalidItemStateException` is not the same as `BudgetExceededError`; they have different codes, different details, and the frontend handles them differently. Generic errors are lazy errors. Lazy errors make debugging painful.

Value Objects: The Money Type

Let me teach you something that Eric Evans describes in *Domain-Driven Design*: the concept of a **value object**. A value object is something that is defined by its value, not its identity. A \$5 bill is a \$5 bill, you don't care *which* \$5 bill it is.

We already decided to store money in cents. But that decision lives in our heads, not in our code. What happens when a new developer joins and writes:

```
const price = 3.50; // Whoops - this is dollars, not cents
await repository.create({ estimatedPrice: price }); // Stores 3 instead of 350
```

TYPESCRIPT

Let's make it impossible to make this mistake. Create `apps/svc-grocery/src/domain/value-objects/money.ts`:

```
export class Money {
  private constructor(private readonly cents: number) {}
```

TYPESCRIPT

Private constructor. You can't write `new Money(350)`. You have to use one of the factory methods I'm about to show you. This forces you to be explicit about the unit:

```

static fromCents(cents: number): Money {
    if (!Number.isInteger(cents)) {
        throw new Error(`Money must be an integer (cents), got: ${cents}`);
    }
    if (cents < 0) {
        throw new Error(`Money cannot be negative, got: ${cents}`);
    }
    return new Money(cents);
}

static fromDollars(dollars: number): Money {
    return Money.fromCents(Math.round(dollars * 100));
}

```

TYPESCRIPT

`Money.fromCents(350)` : explicit. `Money.fromDollars(3.50)` : also explicit, and it handles the rounding for you. You can never accidentally pass dollars where cents are expected.

```

get inCents(): number {
    return this.cents;
}

get inDollars(): number {
    return this.cents / 100;
}

get formatted(): string {
    return `$$ ${this.inDollars.toFixed(2)}`;
}

```

TYPESCRIPT

Three getters. `inCents` for the database. `inDollars` for calculations that need decimals. `formatted` for display. `$3.50`.

TYPESCRIPT

```

add(other: Money): Money {
    return Money.fromCents(this.cents + other.cents);
}

subtract(other: Money): Money {
    const result = this.cents - other.cents;
    if (result < 0) {
        throw new Error(`Cannot subtract: would result in negative money ${result} cents`);
    }
    return Money.fromCents(result);
}

isGreaterThan(other: Money): boolean {
    return this.cents > other.cents;
}

equals(other: Money): boolean {
    return this.cents === other.cents;
}
}

```

All arithmetic is in cents, integer math. No floating-point issues. The `subtract` method refuses to go negative. Money can't be negative in our domain. That's a business rule, enforced at the lowest level.

Valid State Transitions

Create `apps/svc-grocery/src/domain/value-objects/item-status.ts`:

TYPESCRIPT

```

import { InvalidItemStateException } from '../errors';

const VALID_TRANSITIONS: Record<string, string[]> = {
    PENDING: ['APPROVED', 'REJECTED'],
    APPROVED: ['BOUGHT'],
    REJECTED: [], // Terminal state
    BOUGHT: ['ARCHIVED'],
    ARCHIVED: [], // Terminal state
};

```

First thing: import the domain error we'll throw. Then define our state machine as data. PENDING can go to APPROVED or REJECTED. APPROVED can go to BOUGHT. REJECTED and ARCHIVED are terminal, no further transitions.

```
export function canTransition(from: string, to: string): boolean {
  const allowed = VALID_TRANSITIONS[from];
  if (!allowed) return false;
  return allowed.includes(to);
}

export function assertTransition(from: string, to: string, operation: string): void {
  if (!canTransition(from, to)) {
    throw new InvalidItemStateError(from, to, operation);
  }
}
```

TYPESCRIPT

`canTransition` checks if a transition is valid. `assertTransition` throws a `DomainError` if it's not. Use `assertTransition` in use cases. It fails fast and loud with a clear error message.

Testing the Domain Layer

I promised tests after each layer. The domain layer is the easiest to test because it has zero dependencies. Create `apps/svc-grocery/src/domain/_tests_/errors.test.ts`:

```
npm install -D vitest
```

BASH

```
import { describe, it, expect } from 'vitest';
import { ItemNotFoundError, InvalidItemStateError, BudgetExceededError } from '../error';

describe('Domain Errors', () => {
  it('should create ItemNotFoundError with correct fields', () => {
    const error = new ItemNotFoundError('item-123');

    expect(error.code).toBe('ITEM_NOT_FOUND');
    expect(error.statusCode).toBe(404);
    expect(error.message).toBe('Grocery item not found: item-123');
    expect(error.details).toEqual({ itemId: 'item-123' });
    expect(error.timestamp).toBeDefined();
    expect(error).toBeInstanceOf(Error);
  });

  it('should create InvalidItemStateError with transition context', () => {
    const error = new InvalidItemStateError('BOUGHT', 'APPROVED', 'approve');

    expect(error.code).toBe('INVALID_ITEM_STATE');
  });
}
```

TYPESCRIPT

```
expect(error.statusCode).toBe(400);
expect(error.details).toEqual({
  currentStatus: 'BOUGHT',
  targetStatus: 'APPROVED',
  operation: 'approve',
});
});

it('should create BudgetExceededError with amounts', () => {
  const error = new BudgetExceededError(1500, 800);

  expect(error.code).toBe('BUDGET_EXCEEDED');
  expect(error.details).toEqual({
    requestedCents: 1500,
    availableCents: 800,
  });
});
});
```

Create `apps/svc-grocery/src/domain/__tests__/money.test.ts`:

TYPESCRIPT

```

import { describe, it, expect } from 'vitest';
import { Money } from '../value-objects/money';

describe('Money', () => {
  it('should create from cents', () => {
    const price = Money.fromCents(350);
    expect(price.inCents).toBe(350);
    expect(price.inDollars).toBe(3.5);
    expect(price.formatted).toBe('$3.50');
  });

  it('should create from dollars', () => {
    const price = Money.fromDollars(3.50);
    expect(price.inCents).toBe(350);
  });

  it('should add correctly', () => {
    const a = Money.fromCents(350);
    const b = Money.fromCents(450);
    expect(a.add(b).inCents).toBe(800);
  });

  it('should reject negative money', () => {
    const a = Money.fromCents(100);
    const b = Money.fromCents(200);
    expect(() => a.subtract(b)).toThrow('negative money');
  });

  it('should reject non-integer cents', () => {
    expect(() => Money.fromCents(3.5)).toThrow('integer');
  });
});

```

Run them:

BASH

```
npx vitest run
```

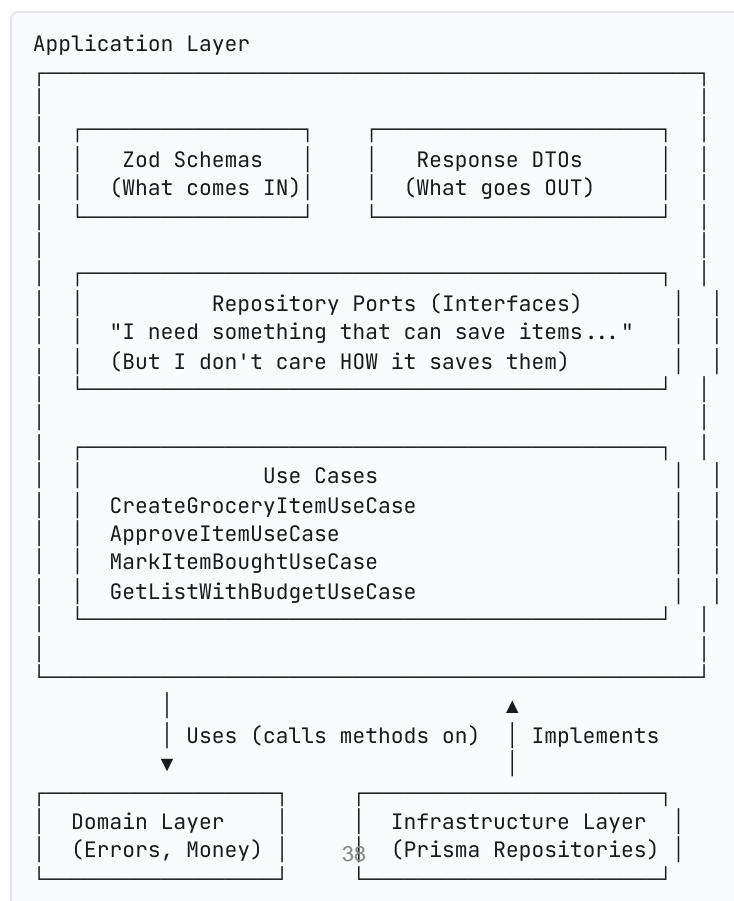
All passing. Notice what we tested: not implementation details, but *behavior*. "Can I create money from cents?" "Does addition work?" "Can money go negative?" These tests will still pass even if we completely rewrite the internal implementation. That's the mark of good tests. A widely accepted testing principle championed by Kent Beck and others: test the behavior, not the mechanism.

CHAPTER 5

The Application Layer - Contracts and Rules

"The application layer is the conductor of the orchestra. It doesn't play any instrument. It tells each musician when to play, how loud, and in what order. It knows the rules. It delegates the work."

What You'll Have When We're Done



The key insight: use cases depend on **interfaces** (ports), not on implementations. The use case says "I need something that can find an item by ID." It doesn't say "I need Prisma." This is what Robert C. Martin calls the **Dependency Inversion Principle**: high-level modules shouldn't depend on low-level modules. Both should depend on abstractions.

Why does this matter? Because you can test your use cases with a fake repository that stores data in an array. No database needed. Fast tests. And if you ever switch from PostgreSQL to MongoDB, you write a new repository implementation and the use cases don't change at all.

Defining the Repository Ports

Create `apps/svc-grocery/src/application/ports/repositories/index.ts`:

First, the **read models**. These define what data the application layer works with. They're not database rows; they're the shape of data that use cases need:

```
export interface GroceryItemReadModel {
    id: string;
    name: string;
    quantity: number;
    estimatedPrice: number; // in cents
    actualPrice: number | null; // in cents, null if not bought yet
    category: string;
    status: string;
    addedById: string;
    addedByName: string;
    listId: string;
    createdAt: Date;
    updatedAt: Date;
}
```

TYPESCRIPT

Notice `addedByName`. The database stores `addedById` (a foreign key), but the application layer wants the user's name too. The repository will join the tables and give us what we need. The use case never writes a JOIN. It just says "give me an item" and gets everything it needs.

```
export interface GroceryListReadModel {
  id: string;
  name: string;
  monthlyBudget: number; // in cents
  createdAt: Date;
  updatedAt: Date;
}
```

TYPESCRIPT

Now the **input types**. These define what data the repository needs to create or update records:

```
export interface CreateGroceryItemInput {
  name: string;
  quantity: number;
  estimatedPrice: number;
  category: string;
  listId: string;
  addedById: string;
}

export interface UpdateItemStatusInput {
  status: string;
  actualPrice?: number;
}
```

TYPESCRIPT

And now the **repository interface**, the contract:

```
export interface IGroceryItemRepository {
  findById(id: string): Promise<GroceryItemReadModel | null>;
  findByListId(listId: string): Promise<GroceryItemReadModel[]>;
  findByListIdAndStatus(listId: string, status: string): Promise<GroceryItemReadModel>;
  create(input: CreateGroceryItemInput): Promise<GroceryItemReadModel>;
  updateStatus(id: string, input: UpdateItemStatusInput): Promise<GroceryItemReadModel>;
}
```

TYPESCRIPT

Five methods. That's all the item repository needs to do. `findById`, `find.byIdListId`, `find.byIdListIdAndStatus`, `create`, and `updateStatus`. Notice the return types: always `Promise` for single lookups (it might not exist) and `Promise` for list queries.

```
export interface IGroceryListRepository {
    findById(id: string): Promise<GroceryListReadModel | null>;
    findByUserId(userId: string): Promise<GroceryListReadModel | null>;
    calculateSpent(listId: string): Promise<number>; // total spent in cents
}
```

TYPESCRIPT

The list repository has three methods. `calculateSpent` sums up the `actualPrice` of all `BOUGHT` items in the list. It returns cents.

These are the `I` prefix interfaces, `IGroceryItemRepository`, `IGroceryListRepository`. The `I` prefix is a convention that screams "this is an interface, not an implementation." When you see `I`, you know there's a contract here that someone has to fulfill.

Defining Request DTOs with Zod

Create `apps/svc-grocery/src/application/dto/request/index.ts`:

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

TYPESCRIPT

One import. Zod gives us runtime validation with compile-time types. Let me show you why that's powerful.

TYPESCRIPT

```
export const createItemSchema = z.object({
  name: z.string()
    .min(1, 'Item name is required')
    .max(100, 'Item name must be under 100 characters')
    .trim(),
  quantity: z.number()
    .int('Quantity must be a whole number')
    .min(1, 'Quantity must be at least 1')
    .max(100, 'Quantity cannot exceed 100'),
  estimatedPrice: z.number()
    .int('Price must be in cents (whole number)')
    .min(1, 'Price must be positive')
    .max(10000000, 'Price cannot exceed $100,000'),
  category: z.enum([
    'DAIRY', 'PRODUCE', 'MEAT', 'BAKERY',
    'BEVERAGES', 'SNACKS', 'HOUSEHOLD', 'OTHER',
  ]).default('OTHER'),
});
```

Every field has validation rules with human-readable error messages. `.trim()` on the name strips whitespace, " Milk " becomes "Milk". `.default('OTHER')` means the category is optional in the request. If you don't send it, it defaults to OTHER.

Now extract the TypeScript type:

TYPESCRIPT

```
export type CreateItemDTO = z.infer<typeof createItemSchema>;
```

This is the magic of Zod. You write the validation schema once, and TypeScript infers the type from it. `CreateItemDTO` is now:

TYPESCRIPT

```
{
  name: string;
  quantity: number;
  estimatedPrice: number;
  category: 'DAIRY' | 'PRODUCE' | 'MEAT' | 'BAKERY' | 'BEVERAGES' | 'SNACKS'
}
```

You didn't write that type, Zod generated it from the schema. One source of truth. Change the schema, the type changes automatically. No drift between validation and types.

Let's add the other schemas:

```
export const approveItemSchema = z.object({
  action: z.enum(['APPROVE', 'REJECT']),
});

export type ApproveItemDTO = z.infer<typeof approveItemSchema>;
```



```
export const markBoughtSchema = z.object({
  actualPrice: z.number()
    .int('Price must be in cents')
    .min(1, 'Price must be positive')
    .max(10000000, 'Price cannot exceed $100,000'),
});

export type MarkBoughtDTO = z.infer<typeof markBoughtSchema>;
```

TYPESCRIPT

Simple. The approve endpoint takes an action (APPROVE or REJECT). The mark-bought endpoint takes the actual price paid.

Defining Response DTOs

Create `apps/svc-grocery/src/application/dto/response/index.ts`:

```
export interface GroceryItemResponseDTO {
    id: string;
    name: string;
    quantity: number;
    estimatedPrice: number;
    actualPrice: number | null;
    category: string;
    status: string;
    addedBy: {
        id: string;
        name: string;
    };
    createdAt: string; // ISO string for JSON
}
```

TYPESCRIPT

Notice `createdAt: string`, not `Date`. JSON doesn't have a Date type; it gets serialized as a string. Being explicit about this prevents confusion.

```
export interface BudgetSummaryDTO {
    monthlyBudget: number;           // total budget in cents
    totalSpent: number;              // sum of bought items in cents
    totalApproved: number;           // sum of approved (not yet bought) items
    remaining: number;               // budget - spent in cents
    itemCounts: {
        pending: number;
        approved: number;
        bought: number;
        rejected: number;
    };
}
```

TYPESCRIPT

The budget summary. The frontend needs all of this to show a budget bar, item counts by status, and the remaining amount.

TYPESCRIPT

```

export interface SuccessResponse<T> {
  success: true;
  data: T;
}

export interface ErrorResponse {
  success: false;
  error: {
    code: string;
    message: string;
    timestamp: string;
    details?: Record<string, unknown>;
  };
}

```

Our **response envelope**. Every API response wraps the data in `{ success: true, data: ... }` or `{ success: false, error: ... }`. The frontend never has to guess whether the response is data or an error. It checks `success` first.

Building Use Cases

Now the business logic. Create `apps/svc-grocery/src/application/use-cases/grocery/index.ts`.

Each use case is a class with:

1. Constructor that receives repository ports (interfaces, not implementations)
2. An `execute` method that runs the business logic
3. Returns a response DTO

First, let's define a shared mapper function at the top of the file. Multiple use cases need to convert a read model to a response DTO, so we extract it once instead of duplicating it in every class:

TYPESCRIPT

```

import { IGroceryItemRepository, IGroceryListRepository, GroceryItemReadModel } from '@app/domain';
import { CreateItemDTO } from '../dto/request';
import { GroceryItemResponseDTO, BudgetSummaryDTO } from '../dto/response';
import { ListNotFoundError, ItemNotFoundError, BudgetExceededError } from '@app/domain/error';
import { assertTransition } from '../domain/value-objects/item-status';

/** Shared mapper – all use cases in this file call this. */
function mapItemToResponse(item: GroceryItemReadModel): GroceryItemResponseDTO {
    return {
        id: item.id,
        name: item.name,
        quantity: item.quantity,
        estimatedPrice: item.estimatedPrice,
        actualPrice: item.actualPrice,
        category: item.category,
        status: item.status,
        addedBy: {
            id: item.addedById,
            name: item.addedByName,
        },
        createdAt: item.createdAt.toISOString(),
    };
}

```

All the imports and the shared mapper live at the top. Every use case below calls `mapItemToResponse()`. This is the DRY principle in action, one mapping function, used everywhere.

Now let's start with `CreateGroceryItemUseCase`:

TYPESCRIPT

```

export class CreateGroceryItemUseCase {
    constructor(
        private readonly itemRepo: IGroceryItemRepository,
        private readonly listRepo: IGroceryListRepository,
    ) {}
}

```

Two dependencies, both interfaces. The use case doesn't know if these are backed by Prisma, MongoDB, or an in-memory array. It doesn't care.

```
async execute(
  userId: string,
  listId: string,
  dto: CreateItemDTO,
): Promise<GroceryItemResponseDTO> {
  // Step 1: Verify the list exists
  const list = await this.listRepo.findById(listId);
  if (!list) {
    throw new ListNotFoundError(listId);
  }
}
```

TYPESCRIPT

First thing: verify the list exists. If it doesn't, fail fast with a domain error. Don't try to create an item in a non-existent list.

```
// Step 2: Create the item
const item = await this.itemRepo.create({
  name: dto.name,
  quantity: dto.quantity,
  estimatedPrice: dto.estimatedPrice,
  category: dto.category,
  listId: listId,
  addedById: userId,
});
```

TYPESCRIPT

Create the item through the repository port. The status defaults to PENDING in the database.

```
// Step 3: Map to response DTO
return mapItemToResponse(item);
}
```

TYPESCRIPT

We call `mapItemToResponse()`, the shared mapper we defined at the top. Every use case in this file uses the same mapper. This mapping is explicit; we control exactly what the API returns.

Now **ApproveItemUseCase**:

TYPESCRIPT

```
export class ApproveItemUseCase {
    constructor(
        private readonly itemRepo: IGroceryItemRepository,
    ) {}

    async execute(
        itemId: string,
        action: 'APPROVE' | 'REJECT',
    ): Promise<GroceryItemResponseDTO> {
        // Step 1: Find the item
        const item = await this.itemRepo.findById(itemId);
        if (!item) {
            throw new ItemNotFoundError(itemId);
        }
    }
}
```

Find the item. If it doesn't exist, fail.

TYPESCRIPT

```
// Step 2: Validate state transition
const newStatus = action === 'APPROVE' ? 'APPROVED' : 'REJECTED';
assertTransition(item.status, newStatus, action.toLowerCase());
```

Here's where our domain value object pays off. `assertTransition` checks if PENDING → APPROVED (or REJECTED) is valid. If someone tries to approve a BOUGHT item, it throws `InvalidItemStateException` with the exact context.

TYPESCRIPT

```
// Step 3: Update the status
const updated = await this.itemRepo.updateStatus(itemId, {
    status: newStatus,
});

return mapItemToResponse(updated);
}
```

`assertTransition` and `ItemNotFoundError` are already imported at the top of the file in our consolidated imports block.

Now `MarkItemBoughtUseCase`. This one involves money:

TYPESCRIPT

```

export class MarkItemBoughtUseCase {
  constructor(
    private readonly itemRepo: IGroceryItemRepository,
    private readonly listRepo: IGroceryListRepository,
  ) {}

  async execute(
    itemId: string,
    actualPrice: number,
  ): Promise<GroceryItemResponseDTO> {
    // Step 1: Find the item
    const item = await this.itemRepo.findById(itemId);
    if (!item) {
      throw new ItemNotFoundError(itemId);
    }

    // Step 2: Validate state transition (must be APPROVED)
    assertTransition(item.status, 'BOUGHT', 'mark as bought');
  }
}

```

Only APPROVED items can be bought. The state machine enforces this.

TYPESCRIPT

```

// Step 3: Check budget
const list = await this.listRepo.findById(item.listId);
if (!list) {
  throw new ListNotFoundError(item.listId);
}

const totalSpent = await this.listRepo.calculateSpent(item.listId);
const remaining = list.monthlyBudget - totalSpent;

if (actualPrice > remaining) {
  throw new BudgetExceededError(actualPrice, remaining);
}

```

Budget check. We calculate what's already been spent, find the remaining budget, and verify the purchase fits. If it doesn't, `BudgetExceededError` fires with the exact amounts.

You might be tempted to move this budget check into the repository. Don't. `if (actualPrice > remaining)` is a **business rule**. It belongs in the use case. The repository just calculates the sum; that's a data operation. The *decision* about whether the sum is acceptable is business logic.

```
// Step 4: Update status and record actual price
const updated = await this.itemRepo.updateStatus(itemId, {
    status: 'BOUGHT',
    actualPrice: actualPrice,
});

return mapItemToResponse(updated);
}
```

TYPESCRIPT

`BudgetExceededError` is already imported at the top of the file.

Finally, `GetListWithBudgetUseCase`:

```
export class GetListWithBudgetUseCase {
    constructor(
        private readonly itemRepo: IGroceryItemRepository,
        private readonly listRepo: IGroceryListRepository,
    ) {}

    async execute(listId: string): Promise<{
        items: GroceryItemResponseDTO[];
        budget: BudgetSummaryDTO;
    }> {
        const list = await this.listRepo.findById(listId);
        if (!list) {
            throw new ListNotFoundError(listId);
        }

        const items = await this.itemRepo.findByListId(listId);
        const totalSpent = await this.listRepo.calculateSpent(listId);
    }
}
```

TYPESCRIPT

Get the list, get all items, calculate spending. Three data operations, no business logic yet.

TYPESCRIPT

```
// Calculate budget summary
const totalApproved = items
  .filter((i) => i.status === 'APPROVED')
  .reduce((sum, i) => sum + i.estimatedPrice * i.quantity, 0);

const itemCounts = {
  pending: items.filter((i) => i.status === 'PENDING').length,
  approved: items.filter((i) => i.status === 'APPROVED').length,
  bought: items.filter((i) => i.status === 'BOUGHT').length,
  rejected: items.filter((i) => i.status === 'REJECTED').length,
};
```

Budget calculations. `totalApproved` is the estimated cost of all approved-but-not-yet-bought items. This tells the household: "Even if nothing goes over estimate, you'll spend this much."

TYPESCRIPT

```
return {
  items: items.map((item) => mapItemToResponse(item)),
  budget: {
    monthlyBudget: list.monthlyBudget,
    totalSpent,
    totalApproved,
    remaining: list.monthlyBudget - totalSpent,
    itemCounts,
  },
};
```

Testing Use Cases

Create `apps/svc-grocery/src/application/_tests_/create-item.test.ts`:

TYPESCRIPT

```

import { describe, it, expect, vi } from 'vitest';
import { CreateGroceryItemUseCase } from '../use-cases/grocery';
import { ListNotFoundError } from '../../domain/errors';

// Mock repositories - no database needed
const mockItemRepo = {
  findById: vi.fn(),
  find.byIdListId: vi.fn(),
  find.byIdListIdAndStatus: vi.fn(),
  create: vi.fn(),
  updateStatus: vi.fn(),
};

const mockListRepo = {
  findById: vi.fn(),
  find.byIdUserId: vi.fn(),
  calculateSpent: vi.fn(),
};

```

We create mock objects that match the repository interfaces. `vi.fn()` creates a spy function that we can control and inspect.

TYPESCRIPT

```

describe('CreateGroceryItemUseCase', () => {
  const useCase = new CreateGroceryItemUseCase(mockItemRepo, mockListRepo);

  it('should create an item when list exists', async () => {
    // Arrange: list exists
    mockListRepo.findById.mockResolvedValue({
      id: 'list-1',
      name: 'Test List',
      monthlyBudget: 50000,
    });

    // Arrange: repository returns created item
    mockItemRepo.create.mockResolvedValue({
      id: 'item-1',
      name: 'Milk',
    });
  });
}

```

```

        quantity: 2,
        estimatedPrice: 350,
        actualPrice: null,
        category: 'DAIRY',
        status: 'PENDING',
        addedById: 'user-1',
        addedByName: 'Alex',
        listId: 'list-1',
        createdAt: new Date(),
        updatedAt: new Date(),
    });

// Act
const result = await useCase.execute('user-1', 'list-1', {
    name: 'Milk',
    quantity: 2,
    estimatedPrice: 350,
    category: 'DAIRY',
});

// Assert
expect(result.name).toBe('Milk');
expect(result.status).toBe('PENDING');
expect(mockItemRepo.create).toHaveBeenCalledWith(
    expect.objectContaining({ name: 'Milk', listId: 'list-1' }),
);
});

it('should throw ListNotFoundError when list does not exist', async () => {
    mockListRepo.findById.mockResolvedValue(null);

    await expect(
        useCase.execute('user-1', 'nonexistent', {
            name: 'Milk',
            quantity: 1,
            estimatedPrice: 350,
            category: 'DAIRY',
        })
    ).rejects.toThrow();
});
});
});
```

```
    }),
    ).rejects.toThrow(ListNotFoundError);
});
});
```

Two tests. Happy path and error path. No database, no HTTP server, no network. These tests run in milliseconds. That's the power of depending on interfaces instead of implementations.

CHAPTER 6

The Infrastructure Layer - Making It Real

"The infrastructure layer is where abstractions meet reality. This is where 'save this item' becomes an actual SQL INSERT statement. It's the only layer that knows about Prisma, PostgreSQL, or any other external system."

What You'll Have When We're Done

When you call `itemRepo.findById('abc-123')`, this is the SQL that actually runs:

```
SELECT gi.*, u.name as "addedByName"
FROM "GroceryItem" gi
JOIN "User" u ON gi."addedById" = u.id
WHERE gi.id = 'abc-123'
```

SQL

You'll never write that SQL. Prisma generates it. But you'll write the TypeScript that tells Prisma what to generate.

The Item Repository Implementation

Create `apps/svc-grocery/src/infrastructure/repositories/prisma-grocery-item.repository.ts`:

```
import { db } from '@grocery/core-db';
import type { PrismaClient } from '@grocery/core-db';
import type {
  IGroceryItemRepository,
  GroceryItemReadModel,
  CreateGroceryItemInput,
  UpdateItemStatusInput,
} from '../../../../../application/ports/repositories';
```

TYPESCRIPT

We import `db` (our Prisma singleton) and the interfaces we need to implement. The `type` keyword on imports tells TypeScript "I'm importing this for type checking only. Don't include it in the compiled JavaScript." It's a small optimization that keeps your bundles clean.

```
export class PrismaGroceryItemRepository implements IGroceryItemRepository {
  private readonly prisma: PrismaClient;

  constructor(prismaClient?: PrismaClient) {
    this.prisma = prismaClient ?? db;
  }
}
```

TYPESCRIPT

`implements IGroceryItemRepository`, this is the contract. TypeScript will error if we're missing any method or if the return types don't match. The constructor accepts an optional Prisma client for testing (you can pass a test client with a test database) and defaults to the singleton.

TYPESCRIPT

```

async findById(id: string): Promise<GroceryItemReadModel | null> {
  const item = await this.prisma.groceryItem.findUnique({
    where: { id },
    include: {
      addedBy: {
        select: { name: true },
      },
    },
  });

  return item ? this.toReadModel(item) : null;
}

```

`findUnique` with `include`: Prisma joins the User table to get the name. We select only `name` because that's all we need. Selecting everything (`include: { addedBy: true }`) would include the password hash, and we never want that leaking into our read model.

The `return item ? this.toReadModel(item) : null` pattern: if the item exists, map it to our read model. If not, return null. The use case handles the null.

TYPESCRIPT

```

async findListId(listId: string): Promise<GroceryItemReadModel[]> {
  const items = await this.prisma.groceryItem.findMany({
    where: { listId },
    include: {
      addedBy: {
        select: { name: true },
      },
    },
    orderBy: { createdAt: 'desc' },
  });

  return items.map((item) => this.toReadModel(item));
}

```

`findMany` with `orderBy`: returns all items in a list, newest first. The `map` call converts each database row into our read model.

TYPESCRIPT

```
async findByListIdAndStatus(
    listId: string,
    status: string,
): Promise<GroceryItemReadModel[]> {
    const items = await this.prisma.groceryItem.findMany({
        where: { listId, status: status as any },
        include: {
            addedBy: {
                select: { name: true },
            },
        },
        orderBy: { createdAt: 'desc' },
    });

    return items.map((item) => this.toReadModel(item));
}
```

Same pattern, but filtered by status. The `status as any` cast is a pragmatic shortcut. Prisma expects its generated `ItemStatus` enum type, but our repository interface uses plain strings. The strings match the enum values exactly (`"PENDING"` , `"APPROVED"` , etc.), so this is safe at runtime. In a stricter codebase, you'd create a mapping function like `toItemStatus(status: string): ItemStatus` that validates and converts. For this guide, the cast keeps the code focused on the architecture patterns.

TYPESCRIPT

```
async create(input: CreateGroceryItemInput): Promise<GroceryItemReadModel>
  const item = await this.prisma.groceryItem.create({
    data: {
      name: input.name,
      quantity: input.quantity,
      estimatedPrice: input.estimatedPrice,
      category: input.category as any,
      status: 'PENDING',
      listId: input.listId,
      addedById: input.addedById,
    },
    include: {
      addedBy: {
        select: { name: true },
      },
    },
  });
  return this.toReadModel(item);
}
```

`create` with explicit field mapping. Notice `status: 'PENDING'` -- hardcoded. New items are always PENDING. The application layer decides the business rule (all items start as PENDING), and the repository enforces it at the data level.

TYPESCRIPT

```

async updateStatus(
  id: string,
  input: UpdateItemStatusInput,
): Promise<GroceryItemReadModel> {
  const item = await this.prisma.groceryItem.update({
    where: { id },
    data: {
      status: input.status as any,
      ...(input.actualPrice !== undefined && { actualPrice: input.actualPrice }),
      include: {
        addedBy: {
          select: { name: true },
        },
      },
    },
  });

  return this.toReadModel(item);
}

```

The `...(input.actualPrice !== undefined && { actualPrice: input.actualPrice })` pattern: only include `actualPrice` in the update if it was provided. This is a TypeScript spread trick. If the condition is false, the spread is empty (no-op). If true, it adds `actualPrice` to the update. This way, approving an item doesn't accidentally set `actualPrice` to undefined.

Now the most important method, the mapper:

TYPESCRIPT

```

private toReadModel(item: any): GroceryItemReadModel {
  return {
    id: item.id,
    name: item.name,
    quantity: item.quantity,
    estimatedPrice: item.estimatedPrice,
    actualPrice: item.actualPrice,
    category: item.category,
    status: item.status,
    addedById: item.addedById,
    addedByName: item.addedBy?.name ?? 'Unknown',
    listId: item.listId,
    createdAt: item.createdAt,
    updatedAt: item.updatedAt,
  };
}
}

```

`toReadModel` maps from the Prisma database row to our application read model. `item.addedBy?.name ?? 'Unknown'` uses optional chaining; if the relation is missing for some reason, default to 'Unknown' instead of crashing.

This mapper is the **boundary** between the database world and the application world. If a column name changes in the database, only this function changes. If the application needs a different field name, only this function changes. Nothing else in the application knows about database column names.

You might be tempted to put business logic here, like checking if the budget is exceeded before saving. Don't. The repository is a data access layer. It saves and retrieves. Business rules live in use cases. Martin Fowler describes this distinction clearly in *Patterns of Enterprise Application Architecture*: the repository is about persistence, not policy.

The List Repository Implementation

Create `apps/svc-grocery/src/infrastructure/repositories/prisma-grocery-list.repository.ts`:

TYPESCRIPT

```

import { db } from '@grocery/core-db';
import type { PrismaClient } from '@grocery/core-db';
import type {
  IGroceryListRepository,
  GroceryListReadModel,
} from '../../application/ports/repositories';

export class PrismaGroceryListRepository implements IGroceryListRepository {
  private readonly prisma: PrismaClient;

  constructor(prismaClient?: PrismaClient) {
    this.prisma = prismaClient ?? db;
  }

  async findById(id: string): Promise<GroceryListReadModel | null> {
    const list = await this.prisma.groceryList.findUnique({
      where: { id },
    });

    return list ? this.toReadModel(list) : null;
  }

  async findByUserId(userId: string): Promise<GroceryListReadModel | null> {
    const user = await this.prisma.user.findUnique({
      where: { id: userId },
      include: { list: true },
    });

    return user?.list ? this.toReadModel(user.list) : null;
  }
}

```

`findByUserId` goes through the User to find their list. We could query the list directly with a join, but going through the user makes the relationship clear.

TYPESCRIPT

```

async calculateSpent(listId: string): Promise<number> {
  const result = await this.prisma.groceryItem.aggregate({
    where: {
      listId,
      status: 'BOUGHT',
      actualPrice: { not: null },
    },
    _sum: {
      actualPrice: true,
    },
  });

  return result._sum.actualPrice ?? 0;
}

```

aggregate with `_sum`: Prisma runs a `SUM(actualPrice)` query in PostgreSQL. We filter for BOUGHT items only. If nothing has been bought, the sum is null, so we default to 0.

This is a data operation, not business logic. "What's the sum of all bought item prices?" is a question, not a decision. The use case takes this number and makes the decision: "Is this within budget?"

TYPESCRIPT

```

private toReadModel(list: any): GroceryListReadModel {
  return {
    id: list.id,
    name: list.name,
    monthlyBudget: list.monthlyBudget,
    createdAt: list.createdAt,
    updatedAt: list.updatedAt,
  };
}

```

Test It

At this point, you could write integration tests that use a real database. But let's save that for later and move to the interface layer. I want you to see an API response first.

Principle check: *Ship it, then perfect it.* We have two repository implementations that satisfy their interface contracts. The use cases can now talk to a real database. Let's make them talk to the real world.

CHAPTER 7

The Interface Layer - Talking to the World

"The interface layer is a translator. It speaks HTTP on one side and use-case language on the other. It's thin. It's boring. That's the point."

What You'll Have When We're Done

```
$ curl -s http://localhost:3060/api/v1/grocery/items | jq
{
  "success": true,
  "data": {
    "items": [
      {
        "id": "a1b2c3d4-...",
        "name": "Whole Milk",
        "quantity": 2,
        "estimatedPrice": 350,
        "actualPrice": null,
        "category": "DAIRY",
        "status": "PENDING",
        "addedBy": { "id": "user-1", "name": "Alex" },
        "createdAt": "2026-02-23T10:30:00.000Z"
      }
    ],
    "budget": {
      "monthlyBudget": 50000,
      "totalSpent": 799,
      "totalApproved": 450,
      "remaining": 49201,
      "itemCounts": { "pending": 1, "approved": 1, "bought": 1, "rejected": 0 }
    }
  }
}
```

BASH

The Validation Middleware

Create `apps/svc-`

`grocery/src/interface/middlewares/validation.middleware.ts`:

```
import type { Request, Response, NextFunction } from 'express';
import type { ZodSchema } from 'zod';
import { ZodError } from 'zod';
```

TYPESCRIPT

Three imports. Request, Response, NextFunction from Express (types only). ZodSchema (the type of any Zod schema) and ZodError (the error that Zod throws on validation failure).

```
export function validateBody<T>(schema: ZodSchema<T>) {
```

TYPESCRIPT

This is a **higher-order function**: a function that returns a function. `validateBody(createItemSchema)` returns an Express middleware that validates the request body against that specific schema. Let me show you:

TYPESCRIPT

```

export function validateBody<T>(schema: ZodSchema<T>) {
  return (req: Request, res: Response, next: NextFunction) =>
    try {
      req.body = schema.parse(req.body);
      next();
    } catch (error) {
      if (error instanceof ZodError) {
        res.status(400).json({
          success: false,
          error: {
            code: 'VALIDATION_ERROR',
            message: 'Request validation failed',
            timestamp: new Date().toISOString(),
            details: error.errors.map((e) => ({
              field: e.path.join('.'),
              message: e.message,
              code: e.code,
            })),
          },
        });
      };
      return;
    }
    next(error);
  }
};

```

Let me trace through this line by line.

`req.body = schema.parse(req.body)` : Zod parses the request body. If it passes, `req.body` is replaced with the parsed, stripped, coerced version. The `.trim()` on our name field? That happens here. Extra fields not in the schema? Stripped. This is important: it means the controller and use case receive only clean, validated data.

If parsing fails, Zod throws a `ZodError`. We catch it and return a structured 400 response with field-level errors:

```
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Request validation failed",
    "timestamp": "2026-02-23T10:30:00.000Z",
    "details": [
      { "field": "name", "message": "Item name is required", "code": "too-long" },
      { "field": "estimatedPrice", "message": "Price must be positive", "code": "not-positive" }
    ]
  }
}
```

JSON

The frontend can map each detail to the corresponding form field and show the error right next to the input. That's good UX, powered by good architecture.

The Error Handler Middleware

Create `apps/svc-grocery/src/interface/middlewares/error-handler.middleware.ts`:

```
import type { Request, Response, NextFunction } from 'express';
import { DomainError } from '../domain/errors';
```

TYPESCRIPT

Two imports. Express types and our `DomainError` base class.

```
export function errorHandler(
  err: Error,
  _req: Request,
  res: Response,
  _next: NextFunction,
): void {
```

TYPESCRIPT

Express error handlers have four parameters. The underscore prefix (`_req`, `_next`) tells other developers "I receive this parameter but don't use it."

Express identifies error handlers by the four-parameter signature. If you forget one, Express treats it as a regular middleware and your errors won't be caught.

TYPESCRIPT

```

if (err instanceof DomainError) {
  res.status(err.statusCode).json({
    success: false,
    error: {
      code: err.code,
      message: err.message,
      timestamp: err.timestamp,
      details: err.details,
    },
  });
  return;
}

```

If the error is a `DomainError`, we return a structured response with the status code, code, message, and details from the error. This is why we put `statusCode` on the domain error. The error handler reads it to set the HTTP status.

TYPESCRIPT

```

// Unexpected errors – log everything, return nothing specific
console.error('Unexpected error:', err);
res.status(500).json({
  success: false,
  error: {
    code: 'INTERNAL_ERROR',
    message: 'An unexpected error occurred',
    timestamp: new Date().toISOString(),
  },
});

```

Unexpected errors get a generic message. We log the full error for debugging, but the user sees nothing specific. No stack traces, no internal paths, no database error messages. This is a security practice: error details can reveal your infrastructure to attackers.

The Controller

Create `apps/svc-grocery/src/interface/controllers/grocery.controller.ts`:

```
import type { Request, Response, NextFunction } from 'express';
import type { CreateGroceryItemUseCase } from '../application/use-cases';
import type { ApproveItemUseCase } from '../application/use-cases/grocery';
import type { MarkItemBoughtUseCase } from '../application/use-cases/grocery';
import type { GetListWithBudgetUseCase } from '../application/use-cases/budget';
```

TYPESCRIPT

The controller imports use case types. It doesn't import the implementations. It receives them through the constructor. That's dependency injection.

```
export class GroceryController {
  constructor(
    private readonly createItemUseCase: CreateGroceryItemUseCase,
    private readonly approveItemUseCase: ApproveItemUseCase,
    private readonly markBoughtUseCase: MarkItemBoughtUseCase,
    private readonly getListUseCase: GetListWithBudgetUseCase,
  ) {}
```

TYPESCRIPT

Four use cases, injected through the constructor. The controller doesn't create them. It receives them. We'll see who creates them in the next chapter (the DI container).

TYPESCRIPT

```

getListWithBudget = async (
  req: Request,
  res: Response,
  next: NextFunction,
): Promise<void> => {
  try {
    const listId = req.params.listId as string;
    const result = await this.getListUseCase.execute(listId);
    res.json({ success: true, data: result });
  } catch (error) {
    next(error);
  }
};

```

Look at how thin this is. Three lines inside the try:

1. Extract `listId` from the URL parameter
2. Call the use case
3. Send the response

That's it. No business logic. No database queries. No validation (that happened in the middleware). The controller's only job is to be a translator between HTTP and use cases.

The `= async (req, res, next) =>` syntax (arrow function assigned to a property) is important. It binds `this` automatically. If we used a regular method, we'd need to bind it manually when passing it to Express routes, or `this.createItemUseCase` would be undefined. This is an Express gotcha that catches many developers.

TYPESCRIPT

```

createItem = async (
  req: Request,
  res: Response,
  next: NextFunction,
): Promise<void> => {
  try {
    const userId = (req as any).user?.id ?? 'anonymous';
    const listId = req.params.listId as string;
    const result = await this.createItemUseCase.execute(
      userId,
      listId,
      req.body,
    );
    res.status(201).json({ success: true, data: result });
  } catch (error) {
    next(error);
  }
};

```

`res.status(201)` : 201 Created, not 200 OK. HTTP status codes matter. 201 means "I created a new resource." The frontend and API consumers use this to differentiate between "here's existing data" (200) and "I just created something new" (201).

`(req as any).user?.id` : we access the user from the request. The auth middleware (Chapter 12) will attach this and introduce an `AuthenticatedRequest` type that replaces the `(req as any)` cast with proper type safety. For now, we fallback to 'anonymous'. Ugly? Yes. But *Ship it, then perfect it*. We'll clean this up when we add auth.

TYPESCRIPT

```

approveItem = async (
  req: Request,
  res: Response,
  next: NextFunction,
): Promise<void> => {
  try {
    const itemId = req.params.itemId as string;
    const { action } = req.body;
    const result = await this.approveItemUseCase.execute(itemId, action);
    res.json({ success: true, data: result });
  } catch (error) {
    next(error);
  }
};

markBought = async (
  req: Request,
  res: Response,
  next: NextFunction,
): Promise<void> => {
  try {
    const itemId = req.params.itemId as string;
    const { actualPrice } = req.body;
    const result = await this.markBoughtUseCase.execute(itemId, actualPrice);
    res.json({ success: true, data: result });
  } catch (error) {
    next(error);
  }
};
}

```

Every controller method follows the exact same pattern: extract → execute → respond. If you're writing an `if` statement in a controller that isn't about request parsing, stop. It belongs in a use case.

The Routes

Create `apps/svc-grocery/src/interface/routes/grocery.routes.ts`:

```
import { Router } from 'express';
import { validateBody } from '../middlewares/validation.middleware';
import {
  createItemSchema,
  approveItemSchema,
  markBoughtSchema,
} from '../../application/dto/request';
```

TYPESCRIPT

We import the Router, validation middleware, and Zod schemas.

```
import { GroceryController } from '../controllers/grocery.controller';

export function createGroceryRoutes(controller: GroceryController): Router {
  const router = Router();
```

TYPESCRIPT

Another function that receives its dependencies. The controller is passed in. Not imported. We type it as `GroceryController` so TypeScript knows exactly which methods are available. This keeps the routes testable and decoupled from the DI container.

```
// GET /api/v1/grocery/lists/:listId
// Get all items and budget summary for a list
router.get(
  '/lists/:listId',
  controller.getListWithBudget,
);
```

TYPESCRIPT

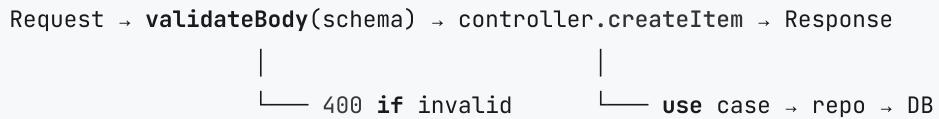
Simple GET route. No middleware needed. It's a read operation. The `:listId` parameter is accessible via `req.params.listId` in the controller.

```
// POST /api/v1/grocery/lists/:listId/items
// Create a new grocery item
router.post(
  '/lists/:listId/items',
  validateBody(createItemSchema),
  controller.createItem,
);
```

TYPESCRIPT

POST route with validation middleware. The request body is validated against `createItemSchema` before the controller ever sees it. If validation fails, the controller is never called; the validation middleware returns the 400 error directly.

This is the **middleware chain** pattern. Each request passes through a pipeline:



```
// POST /api/v1/grocery/items/:itemId/approve
// Approve or reject an item (MANAGER only)
router.post(
  '/items/:itemId/approve',
  validateBody(approveItemSchema),
  controller.approveItem,
);

// POST /api/v1/grocery/items/:itemId/buy
// Mark an item as bought
router.post(
  '/items/:itemId/buy',
  validateBody(markBoughtSchema),
  controller.markBought,
);

return router;
}
```

TYPESCRIPT

Four routes. Four endpoints. Each one validates input before processing. When we add auth in Chapter 12, we'll insert the auth middleware before `validateBody`; the chain grows but the pattern stays the same.

Principle check: *Every line has a reason.* The controller is thin. It calls use cases and formats responses. The middleware handles cross-cutting concerns. The routes wire them together. If your controller is longer than 10 lines per method, something belongs in a use case. If your middleware is doing business logic, it belongs in the domain layer.

The Exception Handling Architecture - Errors as Communication

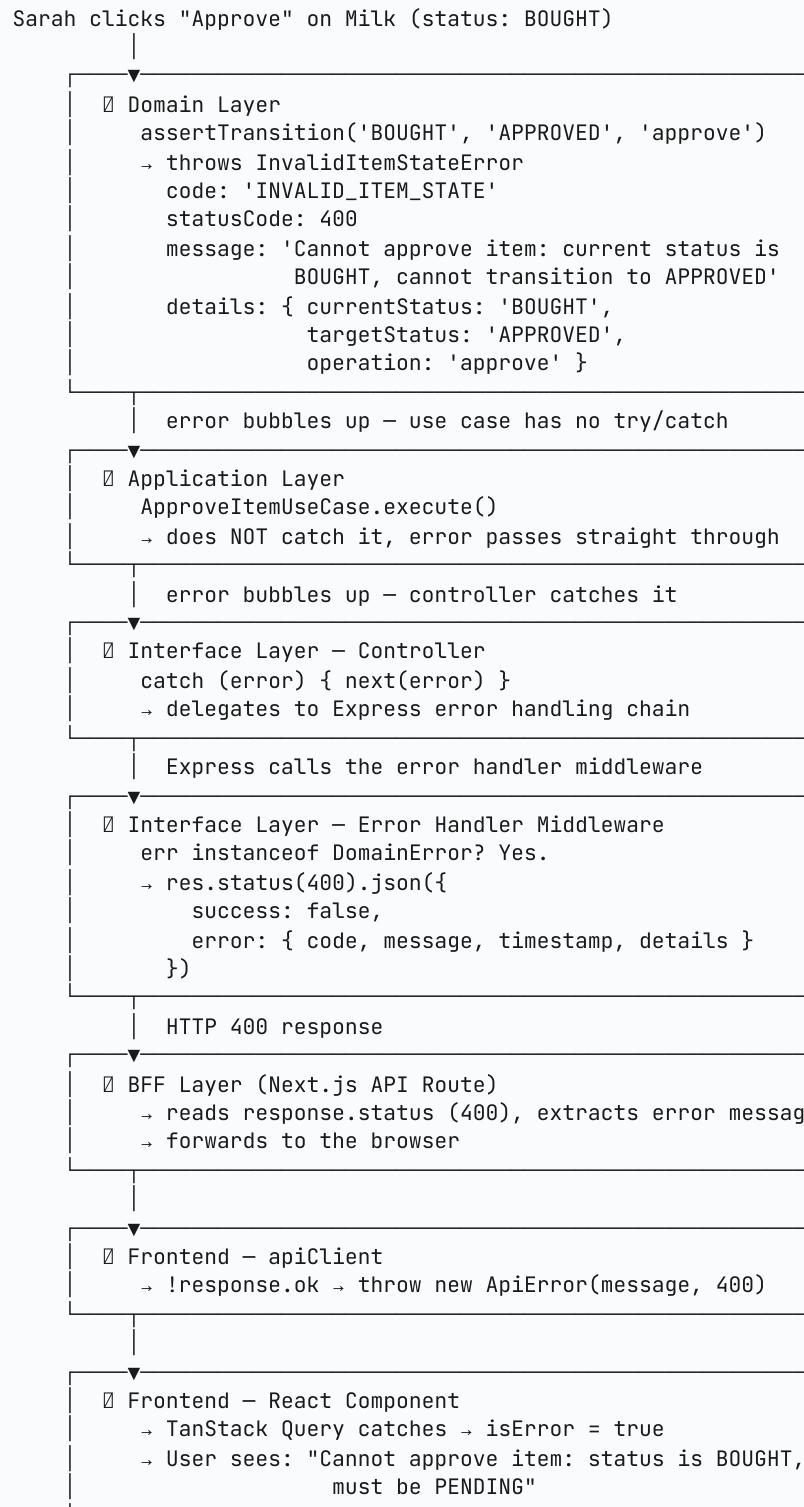
"An error is not a failure. It's a message. A well-thrown error tells the next developer exactly what went wrong, where it went wrong, and what they should have done instead. A swallowed error tells them nothing — until production breaks at 2 AM and nobody knows why."

We've built four layers. Domain, Application, Infrastructure, Interface. In each one, we wrote error handling code — the `DomainError` base class (Chapter 4), the `assertTransition` guard on state changes (Chapter 4), the validation middleware (Chapter 7), the error handler middleware (Chapter 7), the controller's `try/catch/next` pattern (Chapter 7). Each piece was introduced where it naturally belonged.

Now let me show you how all those pieces connect into a single, unified error architecture. This isn't a new chapter. It's a pause. A moment to see the complete picture before we wire everything together in Chapter 8.

The Life of an Error

Let's trace a real error through every layer. Sarah (MANAGER) tries to approve a grocery item that Alex already bought. The item's status is BOUGHT. Watch what happens:



Seven steps. Each one does exactly one thing. The domain throws. The use case lets it pass. The controller delegates. The middleware translates. The BFF forwards. The API client wraps. The component displays. Nobody swallows. Nobody guesses. Nobody adds a fallback that hides the bug.

The error message that Sarah sees in the browser is the *exact same message* that the domain layer generated. It traveled through seven layers without being altered, translated, or lost. That's the payoff.

Three Categories of Errors

Not all errors are the same. Our system handles three distinct categories, and each one follows a different path:

Domain Errors — business rule violations. These are the errors we defined in Chapter 4: `ItemNotFoundError`, `ListNotFoundError`, `InvalidItemStateException`, `BudgetExceededError`, `InsufficientPermissionsError`. Each one extends `DomainError`, has a specific code, a meaningful message, and structured details. The error handler middleware recognizes them by `instanceof DomainError` and returns the appropriate HTTP status with the full error payload.

Validation Errors — input issues caught before business logic runs. These come from Zod in the validation middleware we built in Chapter 7. When a request body fails schema validation, the middleware returns a 400 with field-level errors. The controller never sees them. The use case never runs. The error is caught at the gate.

Infrastructure Errors — unexpected failures. A database connection drops. A query times out. A disk fills up. We don't define custom classes for these because they aren't business rules — they're operational failures. They hit the `else` branch in our error handler: the full error gets logged for the developer, and the user sees a generic "An unexpected error occurred" with a 500 status. No stack traces, no internal paths, no database error messages.

| Category | Defined By | Caught By | HTTP Status | User Sees |
|----------------------|--------------------|-----------------------------|-----------------|----------------------------|
| Domain Error | Your error classes | Error handler middleware | 400 / 403 / 404 | Specific message + details |
| Validation Error | Zod | Validation middleware | 400 | Field-level errors |
| Infrastructure Error | Runtime | Error handler (else branch) | 500 | Generic message |

This table is the complete error taxonomy of our system. Every error that can occur falls into one of these three categories. If you're adding a new error and it doesn't fit, you're probably creating a fourth category that shouldn't exist.

The No-Fallback Philosophy

You might be tempted to write this:

```
// Don't do this.

const item = await this.itemRepo.findById(itemId);
const name = item?.name ?? 'Unknown Item';
const price = item?.estimatedPrice ?? 0;
```

TYPESCRIPT

It looks defensive. It looks safe. It's neither.

If the item doesn't exist, something is wrong. Either the ID is invalid, or your data is corrupted, or there's a race condition between deleting and reading. The `?? 'Unknown Item'` mask hides all three problems behind a silent default. You won't discover the bug until a user reports "Why does my grocery list say Unknown Item?" — and by then, you have no stack trace, no timestamp, no context. You have a mystery.

Here's what we do instead:

```
// Do this.

const item = await this.itemRepo.findById(itemId);

if (!item) {
  throw new ItemNotFoundError(itemId);
}

// Now 'item' is guaranteed to exist. TypeScript knows it too.
```

TYPESCRIPT

Explicit failure. The error includes the item ID, a timestamp, and a clean stack trace. When it fires in development, you see it immediately. You fix it immediately. It never reaches production.

Every code path should either succeed explicitly or throw explicitly. If you're writing `?? defaultValue` in business logic, stop and ask: *under what circumstances would this value actually be missing?* If the answer is "it shouldn't be," then throw an error. Defaults belong in configuration and UI. Not in business logic.

If you can't handle it, don't catch it. Let it bubble. Someone upstream will handle it — that's why we built the error handler middleware.

The Error Contract: What Each Layer Is Allowed To Do

Each layer has a strict contract for how it handles errors. Violate these contracts and your error architecture breaks down.

Domain Layer — throws, never catches. The domain defines errors and throws them. `assertTransition` throws `InvalidItemStateException`. The `Money` value object would throw if you tried to create a negative amount. The domain never catches errors from other layers because it doesn't depend on other layers. If a business rule is violated, throw. Period.

Application Layer (Use Cases) — orchestrates the happy path, never catches. Look at every use case we wrote in Chapter 5. Not a single `try/catch`. The use case calls the domain and the repository. If either throws, the error passes straight through. The use case's job is to orchestrate success. Errors are someone else's problem.

You might be tempted to catch a `ListNotFoundError` in a use case and return `null`. Don't. Returning `null` from a use case is a silent failure. The controller won't know if `null` means "no data found" or "something went terribly wrong." An exception is unambiguous.

Infrastructure Layer — implements, lets most errors bubble. The repository doesn't catch Prisma errors and wrap them in domain errors. If `prisma.groceryItem.findUnique` throws a connection error, it bubbles through the use case and controller to the error handler, which returns a 500. That's the correct behavior: an infrastructure failure is not a business error. It's an operational failure, and the error handler already knows how to handle it.

Interface Layer (Controller) — delegates with `next(error)`, never handles. Every controller method follows the same pattern: `try { ... } catch (error) { next(error) }`. The controller never inspects the error. It never checks `if (error instanceof ItemNotFoundError)`. It never returns a custom message. It passes every error to Express's error handling chain and moves on.

Interface Layer (Error Handler Middleware) — the single point of translation. This is the ONLY place in your entire backend that converts errors to HTTP responses. One function. One file. If you need to change how errors are formatted — add a request ID, switch to a different JSON structure, add localization — you change one file. Not twenty controllers.

BFF Layer — forwards status and message, adds no error logic. The BFF reads the HTTP status from the backend response and forwards it to the browser. It doesn't add error handling. It doesn't retry. It doesn't transform error messages. It's a passthrough.

Frontend — displays specific messages, never swallows. The `ApiError` class carries the HTTP status code so components can handle 404 differently from 500. TanStack Query catches the error and sets `isError = true`. The component renders the actual error message — the one that originated in the domain layer, seven steps ago.

Testing Error Paths

Most developers test the happy path and call it done. Your error paths need tests too — they're the paths your users will actually hit.

We already tested domain errors in Chapter 4: verifying that `assertTransition` throws `InvalidItemStateException` when the state transition is invalid, and that `Money.subtract` throws when the result would be negative. But there's one critical piece we haven't tested yet: the error handler middleware itself.

TYPESCRIPT

```

describe('errorHandler', () => {
  it('should return structured response for DomainError', () => {
    const err = new ItemNotFoundError('item-123');
    const res = { status: vi.fn().mockReturnThis(), json: vi.fn() };

    errorHandler(err, {} as any, res as any, vi.fn());

    expect(res.status).toHaveBeenCalledWith(404);
    expect(res.json).toHaveBeenCalledWith({
      success: false,
      error: expect.objectContaining({
        code: 'ITEM_NOT_FOUND',
        message: 'Grocery item not found: item-123',
      }),
    });
  });
}

it('should return generic 500 for unexpected errors', () => {
  const err = new Error('ECONNREFUSED');
  const res = { status: vi.fn().mockReturnThis(), json: vi.fn() };

  errorHandler(err, {} as any, res as any, vi.fn());

  expect(res.status).toHaveBeenCalledWith(500);
  expect(res.json).toHaveBeenCalledWith({
    success: false,
    error: expect.objectContaining({
      code: 'INTERNAL_ERROR',
      message: 'An unexpected error occurred',
    }),
  });
});
}

```

Two tests. One for domain errors, one for unexpected errors. The error handler is the funnel everything passes through. If it's wrong, every error response is wrong. Test the funnel.

Common Anti-Patterns

Before we move on, let me catalog the mistakes I see most often. If you recognize any of these in your own code, fix them.

Catching and swallowing. The caller has no idea the operation failed. Data is silently lost.

```
// Anti-pattern: the error disappears
try { await riskyOperation(); } catch (e) { console.log(e); }
```

TYPESCRIPT

Returning null instead of throwing. Was the item not found, or did the database crash? The caller can't tell. Both return `null`.

```
// Anti-pattern: silent null
async findItem(id: string): Promise<Item | null> {
  try {
    return await this.repo.findById(id);
  } catch {
    return null; // Hides infrastructure failure behind "not found"
  }
}
```

TYPESCRIPT

Generic error classes. If every error is `AppError('Something went wrong', 400)`, the frontend can't distinguish between a budget overrun and an invalid state transition. The user sees the same message for both. The developer gets the same error code for both. Nobody can debug anything.

try/catch in use cases. If you find yourself writing `try/catch` in a use case, ask why. The use case should throw errors, not handle them. The only exception is if you need to enrich an error with additional context before re-throwing — and even then, think twice.

"Something went wrong" for everything. If your frontend shows the same message for a 400 and a 500, you wasted all the effort we put into structured errors. Use the error code. Switch on it. Show "Item not found" for 404, "Budget exceeded" for `BUDGET_EXCEEDED`, and "Unexpected error, please try again" only for genuine 500s.

Principle check: *Read the error, trust the error.* We didn't build error handling to satisfy a checkbox. We built it so that when something breaks — and something will always break — the error tells you exactly what happened, exactly where, and exactly why. The domain throws specific errors with structured context. The use case doesn't interfere. The controller delegates. The error handler translates. The frontend displays. No layer swallows. No layer guesses. No layer adds a fallback that hides a bug. Errors are communication. Treat them that way.

CHAPTER 8

Wiring It All Together - The DI Container

"This is the moment everything connects. The DI container is the assembly line of your application. It creates every object, wires every dependency, and makes the whole machine run."

What You'll See When We're Done

```
$ npm run dev

  svc-grocery listening on port 3060
  Environment: development
  Database: connected

$ curl http://localhost:3060/health
{"status": "ok", "timestamp": "2026-02-23T10:30:00.000Z"}
```

The Configuration

Create `apps/svc-grocery/src/config.ts`:

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

TYPESCRIPT

We validate environment variables with Zod. If a required variable is missing, the server won't start. This is **fail fast** at the application level: better to crash at startup than to crash on the first request when a missing database URL causes a connection error.

```
const configSchema = z.object({
  port: z.coerce.number().int().min(1000).max(65535).default(3060),
  nodeEnv: z.enum(['development', 'production', 'test']).default('development'),
  databaseUrl: z.string().min(1, 'DATABASE_URL is required'),
  jwtSecret: z.string().min(16, 'JWT_SECRET must be at least 16 characters'),
});
```

TYPESCRIPT

`z.coerce.number()` converts the string environment variable to a number. Environment variables are always strings. `PORT=3060` is the string `"3060"`, not the number `3060`. Zod handles the conversion.

TYPESCRIPT

```

function loadConfig() {

  const result = configSchema.safeParse({
    port: process.env.PORT,
    nodeEnv: process.env.NODE_ENV,
    databaseUrl: process.env.DATABASE_URL,
    jwtSecret: process.env.JWT_SECRET,
  });

  if (!result.success) {
    console.error('Configuration error:');
    result.error.errors.forEach((err) => {
      console.error(`  ${err.path.join('.')}: ${err.message}`);
    });
    process.exit(1);
  }

  return result.data;
}

export const config = loadConfig();

```

`safeParse` doesn't throw. It returns a result object with `success: true/false`. If validation fails, we log each error clearly and exit. The developer sees exactly which variable is wrong:

```

Configuration error:
  databaseUrl: DATABASE_URL is required

```

No mysterious "Cannot read properties of undefined" three minutes into the startup. The error is immediate, specific, and actionable.

The DI Container

Create `apps/svc-grocery/src/shared/di/container.ts`:

TYPESCRIPT

```
// =====
// Dependency Injection Container
//
// This file wires all layers together. It's the ONLY place
// where concrete implementations are created and connected.
//
// Layer order:
//   1. Repositories (infrastructure)
//   2. Use cases (application) – receive repositories
//   3. Controllers (interface) – receive use cases
// =====
```

I start with a comment explaining what this file does and the order of operations. This file is the map of your entire application. A new developer should be able to open this one file and understand how everything connects.

TYPESCRIPT

```
// 1. Infrastructure – Repository implementations
import { PrismaGroceryItemRepository } from '../..../infrastructure/repositories/item';
import { PrismaGroceryListRepository } from '../..../infrastructure/repositories/list';

const itemRepository = new PrismaGroceryItemRepository();
const listRepository = new PrismaGroceryListRepository();
```

Create the repository instances. These are the only concrete implementations in the application. Everything else works with interfaces.

TYPESCRIPT

```
// 2. Application – Use cases (wired with repository ports)

import {
    CreateGroceryItemUseCase,
    ApproveItemUseCase,
    MarkItemBoughtUseCase,
    GetListWithBudgetUseCase,
} from '../application/use-cases/grocery';

const createItemUseCase = new CreateGroceryItemUseCase(itemRepository, listRepository);
const approveItemUseCase = new ApproveItemUseCase(itemRepository);
const markBoughtUseCase = new MarkItemBoughtUseCase(itemRepository, listRepository);
const getListUseCase = new GetListWithBudgetUseCase(itemRepository, listRepository);
```

Create the use cases. Each one receives only the repositories it needs.

`ApproveItemUseCase` only needs the item repository. It doesn't need the list repository because approving doesn't involve budget checks.

TYPESCRIPT

```
// 3. Interface – Controllers (wired with use cases)

import { GroceryController } from '../interface/controllers/grocery.controller';

const groceryController = new GroceryController(
    createItemUseCase,
    approveItemUseCase,
    markBoughtUseCase,
    getListUseCase,
);
```

Create the controller. It receives all four use cases.

TYPESCRIPT

```
// Export the container
export const container = {
    groceryController,
    // Expose repositories for testing if needed
    itemRepository,
    listRepository,
};
```

The container is a simple object. No framework. No decorators. No reflection. Just objects being passed to other objects. This is **manual dependency injection**. It's explicit, debuggable, and you can trace every dependency by reading this one file.

Some frameworks (NestJS, Spring) do this automatically with decorators and reflection. That's convenient but magical. When something breaks, you're debugging framework internals. Manual DI is more typing but zero magic. In production, when you're debugging at 2 AM, you'll appreciate being able to Ctrl+F the dependency and see exactly where it's created.

The App Factory

Create `apps/svc-grocery/src/app.ts`:

```
import express from 'express';
import { createGroceryRoutes } from './interface/routes/grocery.routes';
import { errorHandler } from './interface/middlewares/error-handler.middleware';
import { container } from './shared/di/container';
```

TYPESCRIPT

Four imports. Express, our routes, our error handler, and our DI container.

```
export function createApp(): express.Express {
  const app = express();
```

TYPESCRIPT

A factory function. It creates and returns an Express app. Why a function instead of just exporting the app? Because in tests, you want a fresh app for each test. A factory lets you create as many as you need.

```
// Parse JSON request bodies
app.use(express.json({ limit: '10mb' }));
```

TYPESCRIPT

`express.json()` parses JSON request bodies. The `limit` prevents someone from sending a 1GB JSON payload and crashing your server. 10MB is generous for a grocery app.

```
// Health check - no auth, no middleware
app.get('/health', (_req, res) => {
  res.json({ status: 'ok', timestamp: new Date().toISOString() });
});
```

TYPESCRIPT

Every service needs a health check endpoint. Kubernetes uses this to know if your service is alive. No authentication, no validation, just "am I running?" answered with a JSON response.

```
// API routes
app.use('/api/v1/grocery', createGroceryRoutes(container.groceryContro
```

TYPESCRIPT

Mount the grocery routes at `/api/v1/grocery`. The `v1` prefix is API versioning. When you need breaking changes later, you create `v2` without breaking existing clients.

```
// Error handler (must be last)
app.use(errorHandler);

return app;
}
```

TYPESCRIPT

The error handler **must be the last middleware**. Express processes middleware in order, and errors bubble down. If the error handler isn't last, errors from routes below it won't be caught.

The Entry Point

Create `apps/svc-grocery/src/index.ts`:

```
import { createApp } from './app';
import { config } from './config';

const app = createApp();

app.listen(config.port, () => {
  console.log(`svc-grocery listening on port ${config.port}`);
  console.log(`Environment: ${config.nodeEnv}`);
  console.log(`Health check: http://localhost:${config.port}/health`);
});
```

TYPESCRIPT

That's your entry point. Seven lines. Create the app, start listening. The config validation already happened when we imported `config`. If the environment is wrong, the process already exited.

Running It

Create a `.env` file in `apps/svc-grocery/`:

```
PORT=3060
NODE_ENV=development
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/grocery_db?sslmode=prefer"
JWT_SECRET=dev-secret-change-in-production-please
```

BASH

Make sure your PostgreSQL container from Chapter 3 is still running, and the seed data is loaded. Then:

```
cd apps/svc-grocery
npm run dev
```

BASH

You should see:

```
svc-grocery listening on port 3060
Environment: development
Health check: http://localhost:3060/health
```

Now test it. Open another terminal:

```
# Health check
curl http://localhost:3060/health
```

BASH

```
{"status": "ok", "timestamp": "2026-02-23T10:30:00.000Z"}
```

JSON

It's alive. Now let's test the real API. You'll need the list ID from your seed data. Check Prisma Studio or run:

```
# Get the list (replace LIST_ID with your actual list ID from the seed)
curl -s http://localhost:3060/api/v1/grocery/lists/LIST_ID | jq
```

BASH

You should see the grocery items and budget summary -- real data from your database, flowing through every layer we built.

Let's test creating an item:

```
curl -s -X POST http://localhost:3060/api/v1/grocery/lists/LIST_ID/items
-H "Content-Type: application/json" \
-d '{"name": "Orange Juice", "quantity": 1, "estimatedPrice": 599, "category": "BEVERAGES"}' \
| jq
```

BASH

```
{
  "success": true,
  "data": {
    "id": "new-uuid-here",
    "name": "Orange Juice",
    "quantity": 1,
    "estimatedPrice": 599,
    "actualPrice": null,
    "category": "BEVERAGES",
    "status": "PENDING",
    "addedBy": { "id": "anonymous", "name": "Unknown" },
    "createdAt": "2026-02-23T10:35:00.000Z"
  }
}
```

JSON

PENDING. Just created. Now approve it:

```
BASH
curl -s -X POST http://localhost:3060/api/v1/grocery/items/ITEM_ID/approve \
-H "Content-Type: application/json" \
-d '{"action": "APPROVE"}' \
| jq
```

The status changes to APPROVED. Now buy it:

```
BASH
curl -s -X POST http://localhost:3060/api/v1/grocery/items/ITEM_ID/buy \
-H "Content-Type: application/json" \
-d '{"actualPrice": 549}' \
| jq
```

Bought for \$5.49, less than the \$5.99 estimate. Now test the error handling. Try to approve the same item again:

```
BASH
curl -s -X POST http://localhost:3060/api/v1/grocery/items/ITEM_ID/approve \
-H "Content-Type: application/json" \
-d '{"action": "APPROVE"}' \
| jq
```

```
JSON
{
  "success": false,
  "error": {
    "code": "INVALID_ITEM_STATE",
    "message": "Cannot approve item: current status is BOUGHT, cannot transition to APPROVED",
    "timestamp": "2026-02-23T10:36:00.000Z",
    "details": {
      "currentStatus": "BOUGHT",
      "targetStatus": "APPROVED",
      "operation": "approve"
    }
  }
}
```

That's our domain error in action. The state machine caught the invalid transition, the use case threw a specific error, the error handler formatted it, and the frontend receives a machine-readable, human-debuggable response.

Look at the fields. `currentStatus: "BOUGHT"` -- that's where the item is now. `targetStatus: "APPROVED"` -- that's the status you tried to transition to. The state machine in `item-status.ts` only allows BOUGHT to transition to ARCHIVED. You tried to go backward, and it caught you.

Try sending invalid data:

```
BASH
curl -s -X POST http://localhost:3060/api/v1/grocery/lists/LIST_ID/items
-H "Content-Type: application/json" \
-d '{"name": "", "quantity": -1}' \
| jq
```

```
JSON
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Request validation failed",
    "details": [
      { "field": "name", "message": "Item name is required", "code": "too_short" },
      { "field": "quantity", "message": "Quantity must be at least 1", "code": "min" },
      { "field": "estimatedPrice", "message": "Required", "code": "invalid" }
    ]
  }
}
```

Field-level validation errors. The Zod middleware caught it before the controller was even called.

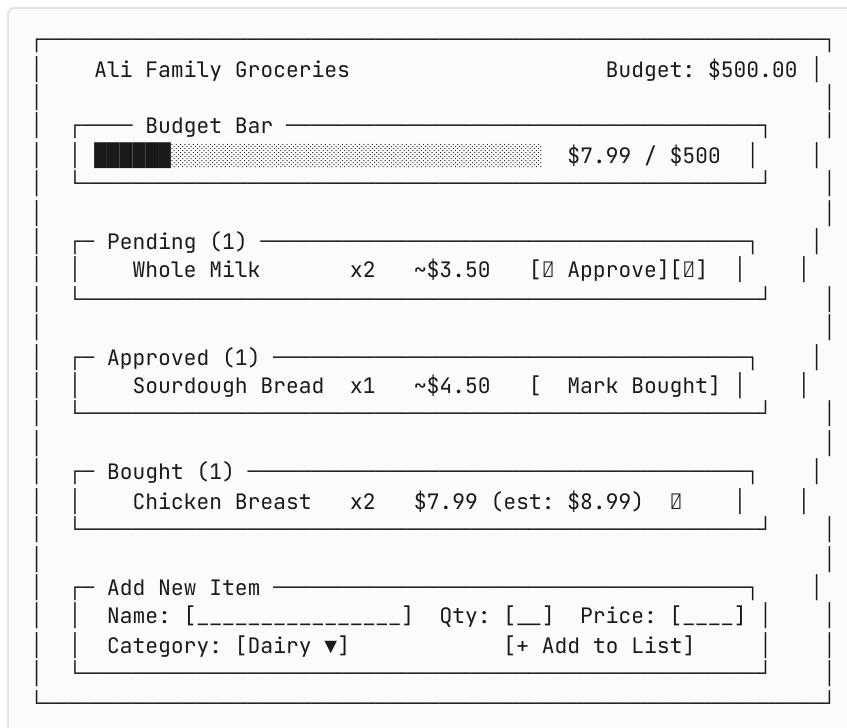
This is the backend complete. Every request flows through: Route → Middleware → Controller → Use Case → Repository → Database, and back. Let's build the frontend.

CHAPTER 9

The Frontend - Pages and Components

"The frontend is the face of your application. Nobody cares how beautiful your Clean Architecture is if the UI is confusing. But a beautiful UI backed by spaghetti code is a time bomb. Let's build both right."

What You'll See When We're Done



Initializing the Next.js App

We'll create the frontend as a **sibling directory** next to our backend monorepo, not inside `apps/`. The backend is a Node.js monorepo with its own workspace config. The frontend is a standalone Next.js app with its own `node_modules`. They communicate over HTTP, so they don't need to share a workspace.

From the root of your project (the `grocery-app/` directory):

```
BASH
npx create-next-app@latest grocery-web --typescript --tailwind --app --src-dir grocery-web
```

The flags:

- `--typescript`: TypeScript, obviously
- `--tailwind`: Tailwind CSS for styling (utility-first, no separate CSS files)
- `--app`: App Router (the modern Next.js routing, not the Pages Router)
- `--src-dir`: Puts code in `src/` to separate it from config files

Next.js will configure the `@/` import alias by default, which maps to `src/`. All our frontend components use `@/` imports (e.g., `import { Button } from '@/components/ui/Button'`), so we need this alias active.

Install our additional dependencies:

```
npm install @tanstack/react-query @tanstack/react-query-devtools
npm install react-hook-form @hookform/resolvers zod
npm install axios
```

BASH

Three groups:

1. **TanStack Query:** Server state management (fetching, caching, synchronization)
2. **React Hook Form + Zod:** Form handling with the same Zod schemas we use on the backend
3. **Axios:** HTTP client for the BFF to call the backend

Project Structure

Organize the frontend:

```
mkdir -p src/{components/{grocery,ui,forms},lib/{hooks},types}
```

BASH

```
grocery-web/src/
  app/
    api/                      # BFF routes (Chapter 10)
      grocery/
        [listId]/
          page.tsx      # Main grocery list page
    layout.tsx                # Root layout
    page.tsx                  # Home/redirect
    providers.tsx            # TanStack Query provider
  components/
    grocery/
      GroceryItemCard.tsx   # Single item display
      ItemList.tsx         # Grouped item list
      BudgetBar.tsx        # Budget visualization
    forms/
      AddItemForm.tsx       # New item form
    ui/
      Button.tsx           # Reusable button
      Badge.tsx            # Status badge
  lib/
    apiClient.ts              # Client-side API fetcher
    backendClient.ts          # Server-side backend proxy (Chapter 10)
    queryClient.ts            # TanStack Query configuration
    hooks/
      useGrocery.ts         # Custom query hooks
  types/
    grocery.ts                # Shared TypeScript types
```

Shared Types

Create `src/types/grocery.ts`:

TYPESCRIPT

```
export interface GroceryItem {
  id: string;
  name: string;
  quantity: number;
  estimatedPrice: number;
  actualPrice: number | null;
  category: string;
  status: 'PENDING' | 'APPROVED' | 'REJECTED' | 'BOUGHT' | 'ARCHIVED';
  addedBy: {
    id: string;
    name: string;
  };
  createdAt: string;
}
```

This mirrors the backend's response DTO. In a production monorepo, you could share this type between frontend and backend. For now, we define it separately. It's the contract between the two.

TYPESCRIPT

```
export interface BudgetSummary {
  monthlyBudget: number;
  totalSpent: number;
  totalApproved: number;
  remaining: number;
  itemCounts: {
    pending: number;
    approved: number;
    bought: number;
    rejected: number;
  };
}

export interface GroceryListResponse {
  items: GroceryItem[];
  budget: BudgetSummary;
}
```

The Root Layout

Edit `src/app/layout.tsx`:

```
import type { Metadata } from 'next';
import './globals.css';
import { Providers } from './providers';

export const metadata: Metadata = {
  title: 'Grocery List',
  description: 'Neighborhood grocery list with budget tracking',
};
```

TYPESCRIPT

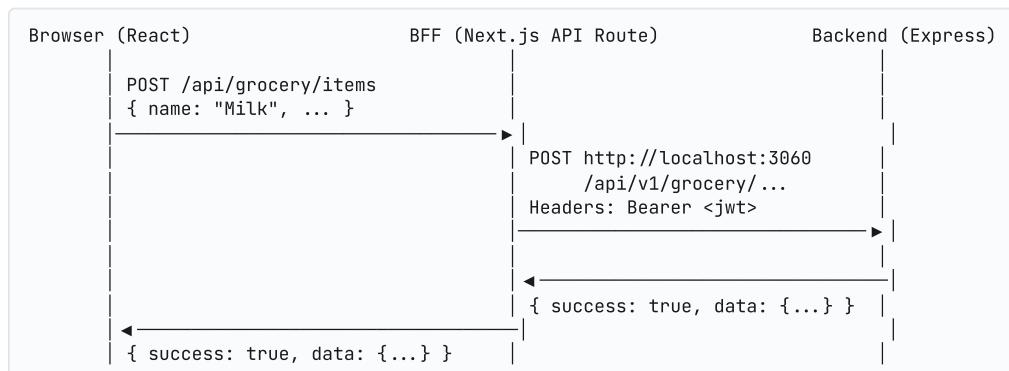
Metadata for the HTML `<head>`. Next.js generates the `<body>`

CHAPTER 10

The BFF - Your Frontend's Private API

"The BFF is your frontend's personal assistant. It handles the messy work (authentication, backend routing, error translation) so your React components stay clean and focused on the UI."

What You'll See When We're Done



The Backend Client

Create `src/lib/backendClient.ts`:

```

import axios from 'axios';
import type { AxiosInstance } from 'axios';
import { cookies } from 'next/headers';
  
```

TYPESCRIPT

Axios for making HTTP requests from the server side. The `cookies` import from Next.js gives us access to the HTTP-only cookie store. We'll use this to read the auth token. This file runs on the Next.js server, not in the browser.

```
const BACKEND_URL = process.env.BACKEND_URL || 'http://localhost:3060';

function createClient(baseURL: string): AxiosInstance {
  return axios.create({
    baseURL: `${baseURL}/api/v1/grocery`,
    timeout: 10000,
    headers: {
      'Content-Type': 'application/json',
    },
  });
}
```

TYPESCRIPT

One function, one client. The `baseURL` includes `/api/v1/grocery` so we don't repeat it in every call. `timeout: 10000` means if the backend doesn't respond within 10 seconds, the request fails. Never make requests without a timeout. A hung backend will hang your frontend forever.

```
export const groceryClient = createClient(BACKEND_URL);
```

TYPESCRIPT

One line. One client for our one backend service. In a production system with multiple microservices, you'd have one client per service:

```
// Production pattern (for reference):
// export const authClient = createClient(process.env.AUTH_URL);
// export const walletClient = createClient(process.env.WALLET_URL);
// export const kycClient = createClient(process.env.KYC_URL);
```

TYPESCRIPT

Each client points to a different microservice, each on its own port or host. The BFF routes decide which client to use based on the endpoint.

```
export function getErrorMessage(error: unknown): string {
  if (axios.isAxiosError(error)) {
    return error.response?.data?.error?.message
      ?? error.response?.data?.message
      ?? error.message;
  }
  if (error instanceof Error) return error.message;
  return 'An unexpected error occurred';
}
```

TYPESCRIPT

A helper to extract readable error messages from Axios errors. The backend might return `{ error: { message: "..." } }` or `{ message: "..." }`. This function handles both.

```
export async function getAuthHeaders(): Promise<Record<string, string>> {
  const cookieStore = await cookies();
  const token = cookieStore.get('auth-token')?.value;
  if (!token) return {};
  return { Authorization: `Bearer ${token}` };
}
```

TYPESCRIPT

This reads the `auth-token` cookie (set during login, which we'll build in Chapter 12) and returns an `Authorization` header. Every BFF route will call this to forward the user's identity to the backend. If there's no cookie (user not logged in), it returns an empty object -- and that's fine right now. Our backend routes aren't protected yet, so requests without auth headers will still work. Once we add the auth middleware in Chapter 12, unauthenticated requests will get a proper 401, and the frontend can redirect to login.

BFF API Routes

Create `src/app/api/grocery/items/route.ts`:

```
import { NextResponse } from 'next/server';
import { groceryClient, getMessage, getAuthHeaders } from '@/lib/backendClient';
```

TYPESCRIPT

`NextResponse` is Next.js's response helper for API routes. `getAuthHeaders` reads the JWT from the session cookie and returns an `Authorization: Bearer` header. We built it earlier in `backendClient.ts`. Every BFF route needs it to forward the user's identity to the backend.

TYPESCRIPT

```
// GET /api/grocery/items?listId=xxx
export async function GET(req: Request) {
  try {
    const { searchParams } = new URL(req.url);
    const listId = searchParams.get('listId');

    if (!listId) {
      return NextResponse.json(
        { success: false, message: 'listId is required' },
        { status: 400 },
      );
    }

    const headers = await getAuthHeaders();
    const response = await groceryClient.get(`/lists/${listId}`, { headers });
    return NextResponse.json(response.data);
  } catch (error) {
    const message = getErrorMessage(error);
    const status = (error as any)?.response?.status ?? 500;
    return NextResponse.json(
      { success: false, message },
      { status },
    );
  }
}
```

The GET handler. It extracts `listId` from the query string, validates it, reads the auth token from the session cookie, then calls the backend with the JWT as a Bearer token. If the backend returns an error, we forward the status code and message.

This is the BFF's job: take a frontend request, add server-side concerns (auth headers, validation), forward to the backend, and return the response. The browser never knows the backend's address, port, or API format.

TYPESCRIPT

```
// POST /api/grocery/items
export async function POST(req: Request) {
  try {
    const body = await req.json();
    const { listId, ...itemData } = body;

    if (!listId) {
      return NextResponse.json(
        { success: false, message: 'listId is required' },
        { status: 400 },
      );
    }

    const headers = await getAuthHeaders();
    const response = await groceryClient.post(`lists/${listId}/items`, itemData, { headers });
    return NextResponse.json(response.data, { status: 201 });
  } catch (error) {
    const message = getErrorMessage(error);
    const status = (error as any)?.response?.status ?? 500;
    return NextResponse.json(
      { success: false, message },
      { status },
    );
  }
}
```

The POST handler. Parse the body, extract listId, forward the item data to the backend. Return 201 Created on success.

Create `src/app/api/grocery/items/[itemId]/approve/route.ts`:

TYPESCRIPT

```

import { NextResponse } from 'next/server';
import { groceryClient, getErrorMessage, getAuthHeaders } from '@/lib/backendClient';

export async function POST(
  req: Request,
  { params }: { params: Promise<{ itemId: string }> },
) {
  try {
    const { itemId } = await params;
    const body = await req.json();
    const headers = await getAuthHeaders();
    const response = await groceryClient.post(
      `/items/${itemId}/approve`,
      body,
      { headers },
    );
    return NextResponse.json(response.data);
  } catch (error) {
    const message = getErrorMessage(error);
    const status = (error as any)?.response?.status ?? 500;
    return NextResponse.json(
      { success: false, message },
      { status },
    );
  }
}

```

Dynamic route parameter: `[itemId]` in the folder name becomes `params.itemId` in the handler. Next.js extracts it from the URL automatically. In Next.js 15+, `params` is a Promise; you must `await` it before accessing the values. And just like the `items` route, we forward the auth headers.

Create `src/app/api/grocery/items/[itemId]/buy/route.ts`:

TYPESCRIPT

```

import { NextResponse } from 'next/server';
import { groceryClient, getErrorMessage, getAuthHeaders } from '@/lib/backendClient';

export async function POST(
  req: Request,
  { params }: { params: Promise<{ itemId: string }> },
) {
  try {
    const { itemId } = await params;
    const body = await req.json();
    const headers = await getAuthHeaders();
    const response = await groceryClient.post(
      `/items/${itemId}/buy`,
      body,
      { headers },
    );
    return NextResponse.json(response.data);
  } catch (error) {
    const message = getErrorMessage(error);
    const status = (error as any)?.response?.status ?? 500;
    return NextResponse.json(
      { success: false, message },
      { status },
    );
  }
}

```

Same pattern. Extract item ID from URL, read auth headers, forward request to backend, return response.

All three BFF routes follow the same structure:

1. Parse the incoming request
2. Validate minimal requirements
3. Read the session cookie and build auth headers
4. Forward to the correct backend endpoint with the JWT
5. Return the backend's response (or a formatted error)

CHAPTER 11

Connecting It All - TanStack Query and Data Flow

"This is the chapter where everything comes alive. The components you built in Chapter 9 meet the BFF from Chapter 10, and the data starts flowing. By the end of this chapter, you'll click 'Add to List' and see the item appear. You'll click 'Approve' and watch the status change. The full stack, working."

The Client-Side API Client

Create `src/lib/apiClient.ts`:

```
export class ApiError extends Error {
  constructor(
    message: string,
    public readonly status: number,
  ) {
    super(message);
    this.name = 'ApiError';
  }
}
```

TYPESCRIPT

A custom error class for API failures. The `status` field carries the HTTP status code so components can handle 404 differently from 500.

TYPESCRIPT

```

export async function apiClient<T>(
  path: string,
  options: RequestInit = {},
): Promise<T> {
  const headers = new Headers(options.headers);
  headers.set('Content-Type', 'application/json');

  const response = await fetch(`api${path}`, {
    ...options,
    headers,
  });

  const data = await response.json();

  if (!response.ok) {
    throw new ApiError(
      data.message ?? data.error?.message ?? 'Request failed',
      response.status,
    );
  }

  return data as T;
}

```

This function calls the BFF routes: `/api/grocery/items`, `/api/grocery/items/[itemId]/approve`, etc. It prepends `/api` to every path, which maps to the `src/app/api/` directory in Next.js.

The key line: `fetch(/api${path})`. This is a **relative URL**. It goes to the same Next.js server that served the page. The browser never sees `localhost:3060`. It sees `/api/grocery/items` and Next.js handles the rest.

Query Key Factory

Create `src/lib/queryClient.ts`:

TYPESCRIPT

```

export const queryKeys = {
  grocery: {
    all: ['grocery'] as const,
    list: (listId: string) => [...queryKeys.grocery.all, 'list', listId] as const,
    budget: (listId: string) => [...queryKeys.grocery.all, 'budget', listId] as const,
  },
};

```

Query keys are how TanStack Query identifies cached data.

`queryKeys.grocery.list('abc')` produces `['grocery', 'list', 'abc']`. When you invalidate `['grocery']`, ALL grocery-related queries refetch. When you invalidate `['grocery', 'list', 'abc']`, only that specific list refetches.

This factory pattern keeps keys consistent across the entire app. No more typo bugs where one component uses `['grocery-items']` and another uses `['groceryItems']`.

Custom Hooks

Create `src/lib/hooks/useGrocery.ts`:

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { apiClient } from '@/lib/apiClient';
import { queryKeys } from '@/lib/queryClient';
import type { GroceryListResponse } from '@/types/grocery';
```

Imports. TanStack Query hooks, our API client, query keys, and types.

```
export function useGroceryList(listId: string) {
  return useQuery({
    queryKey: queryKeys.grocery.list(listId),
    queryFn: async (): Promise<GroceryListResponse> => {
      const response = await apiClient<{ success: boolean; data: GroceryListResponse }>(
        `/grocery/items?listId=${listId}`,
      );
      return response.data;
    },
    enabled: !!listId,
  });
}
```

`useQuery` fetches data and caches it. The `queryKey` identifies the cache entry. The `queryFn` makes the actual request. `enabled: !!listId` means the query won't run if `listId` is empty. No wasted requests.

The return value includes `data`, `isLoading`, `isError`, `error`, and more. The component doesn't manage any of this state. TanStack Query handles it all.

Now the mutations: actions that change data:

TYPESCRIPT

```

export function useCreateItem(listId: string) {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: async (data: {
      name: string;
      quantity: number;
      estimatedPrice: number;
      category: string;
    }) => {
      return apiClient('/grocery/items', {
        method: 'POST',
        body: JSON.stringify({ listId, ...data }),
      });
    },
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: queryKeys.grocery.list(listId) });
    },
  });
}

```

`useMutation` is for POST/PUT/DELETE: operations that change server state. The `mutationFn` makes the API call. `onSuccess` invalidates the list query, which triggers a refetch. The user sees the new item appear without a manual refresh.

This is the **cache invalidation** pattern. Instead of manually updating the cached data (which is error-prone), we tell TanStack Query: "the data for this list just changed, refetch it." Simple and reliable.

TYPESCRIPT

```

export function useApproveItem(listId: string) {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: async ({ itemId, action }: { itemId: string; action: 'APPROVE' | 'REJECT' }) => {
      return apiClient(`/grocery/items/${itemId}/approve`, {
        method: 'POST',
        body: JSON.stringify({ action }),
      });
    },
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: queryKeys.grocery.list(listId) });
    },
  });
}

export function useMarkBought(listId: string) {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: async ({ itemId, actualPrice }: { itemId: string; actualPrice: number }) => {
      return apiClient(`/grocery/items/${itemId}/buy`, {
        method: 'POST',
        body: JSON.stringify({ actualPrice }),
      });
    },
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: queryKeys.grocery.list(listId) });
    },
  });
}

```

Same pattern for approve and mark bought. Every mutation invalidates the list query so the UI stays in sync.

The Main Page

Create `src/app/grocery/[listId]/page.tsx`:

```
'use client';

import { useParams } from 'next/navigation';
import { useGroceryList, useCreateItem, useApproveItem, useMarkBought } from '@/lib/hooks';
import { BudgetBar } from '@/components/grocery/BudgetBar';
import { ItemList } from '@/components/grocery/ItemList';
import { AddItemForm } from '@/components/forms/AddItemForm';
```

TYPESCRIPT

'use client' : this page uses hooks, so it's a Client Component. `useParams()` extracts `listId` from the URL (`/grocery/abc-123`).

```
export default function GroceryListPage() {
  const params = useParams();
  const listId = params.listId as string;

  const { data, isLoading, isError, error } = useGroceryList(listId);
  const createItem = useCreateItem(listId);
  const approveItem = useApproveItem(listId);
  const markBought = useMarkBought(listId);
```

TYPESCRIPT

Six lines. The entire state management for this page. `useGroceryList` fetches the data. `useCreateItem`, `useApproveItem`, `useMarkBought` provide mutation functions. No `useState`, no `useEffect`, no manual fetching.

Anti-pattern warning: Don't call `fetch()` or `axios.get()` directly inside components. Always go through custom hooks that wrap TanStack Query. If you fetch directly, you lose caching, deduplication, background refetching, and loading state management. Similarly, avoid prop drilling server data through 3+ levels of components. If a deeply nested component needs data, give it its own `useQuery` hook with the same query key. TanStack Query will deduplicate the request automatically.

```
if (isLoading) {
  return (
    <div className="max-w-3xl mx-auto px-4 py-8">
      <div className="animate-pulse space-y-4">
        <div className="h-8 bg-gray-200 rounded w-1/3" />
        <div className="h-24 bg-gray-200 rounded" />
        <div className="h-16 bg-gray-200 rounded" />
        <div className="h-16 bg-gray-200 rounded" />
      </div>
    </div>
  );
}
```

TYPESCRIPT

Loading state with skeleton UI. `animate-pulse` creates a subtle shimmer effect. The gray rectangles match the layout of the real content, so the page doesn't "jump" when data loads.

```
if (isError) {
  return (
    <div className="max-w-3xl mx-auto px-4 py-8">
      <div className="bg-red-50 border border-red-200 rounded-lg p-4 text-red-700">
        Failed to load grocery list: {error?.message ?? 'Unknown error'}
      </div>
    </div>
  );
}

if (!data) return null;
```

TYPESCRIPT

Error state. Clear, red, with the actual error message. Not a blank page, not a generic "Something went wrong." The user sees what failed.

```
const handleApprove = (itemId: string) => {
  approveItem.mutate({ itemId, action: 'APPROVE' });
};

const handleReject = (itemId: string) => {
  approveItem.mutate({ itemId, action: 'REJECT' });
};

const handleMarkBought = (itemId: string) => {
  // For simplicity, use estimated price as actual price
  // In production, you'd show a modal asking for the actual price
  const item = data.items.find((i) => i.id === itemId);
  if (item) {
    markBought.mutate({ itemId, actualPrice: item.estimatedPrice });
  }
};

const handleAddItem = (itemData: {
  name: string;
  quantity: number;
  estimatedPrice: number;
  category: string;
}) => {
  createItem.mutate(itemData);
};
```

TYPESCRIPT

Event handlers. Each one calls a mutation. The UI updates automatically through cache invalidation.

```
return (
  <div className="max-w-3xl mx-auto px-4 py-8 space-y-6">
    <h1 className="text-2xl font-bold text-gray-900">
      Grocery List
    </h1>

    <BudgetBar budget={data.budget} />

    <ItemList
      items={data.items}
      onApprove={handleApprove}
      onReject={handleReject}
      onMarkBought={handleMarkBought}
    />

    <AddItemForm
      onSubmit={handleAddItem}
      isSubmitting={createItem.isPending}
    />
  </div>
);
}
```

TYPESCRIPT

The page is assembled from the components we built. `BudgetBar` shows the budget. `ItemList` shows the items grouped by status. `AddItemForm` lets the user add new items. The `isSubmitting` prop disables the button while the mutation is in flight.

The Moment of Truth

Start both servers:

```
# Terminal 1: Backend
cd apps/svc-grocery
npm run dev

# Terminal 2: Frontend
cd grocery-web
npm run dev
```

BASH

Open `http://localhost:3000/grocery/YOUR_LIST_ID`. You should see:

1. The budget bar showing \$7.99 spent of \$500.00

2. Whole Milk (PENDING) with Approve/Reject buttons
3. Sourdough Bread (APPROVED) with a Mark Bought button
4. Chicken Breast (BOUGHT) showing \$7.99 paid

Click "Approve" on the milk. Watch the card move from "Pending" to "Ready to Buy." Click "Mark Bought" on the bread. Watch the budget bar update. Add a new item with the form. Watch it appear under "Pending."

That's the full stack. Browser → React → BFF → Express → Use Case → Repository → PostgreSQL → and all the way back. Sixteen files across four layers, and every layer does exactly one job.

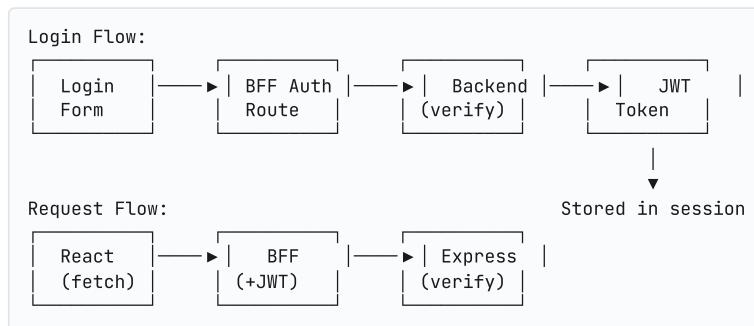
Principle check: *Ship it, then perfect it.* We built the entire application without authentication. Anyone can access any list. Anyone can approve items. That's fine for development. The architecture is sound, the data flows correctly, and the UI works. Now we add the lock on the door.

CHAPTER 12

Authentication - Locking the Doors

"Authentication is the bouncer at the club. It checks your ID, decides if you're allowed in, and stamps your hand so you don't have to show your ID every time you walk to the bar."

What You'll See When We're Done



Backend: The Auth Middleware

Create `apps/svc-grocery/src/interface/middlewares/auth.middleware.ts`:

```

import type { Request, Response, NextFunction } from 'express';
import jwt from 'jsonwebtoken';
import { config } from '../config';
  
```

TYPESCRIPT

Three imports. Express types, the JWT library, and our config (which has the JWT secret).

```
export interface AuthenticatedRequest extends Request {
  user?: {
    id: string;
    email: string;
    role: string;
    listId: string | null;
  };
}
```

TYPESCRIPT

We extend Express's Request type to include `user`. After the auth middleware runs, `req.user` is populated with the decoded JWT payload. This is how downstream handlers know who's making the request.

```
export function authMiddleware(
  req: AuthenticatedRequest,
  res: Response,
  next: NextFunction,
): void {
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    res.status(401).json({
      success: false,
      error: {
        code: 'UNAUTHORIZED',
        message: 'Missing or invalid authorization header',
        timestamp: new Date().toISOString(),
      },
    });
    return;
}
```

TYPESCRIPT

First check: does the request have an `Authorization: Bearer` header? If not, 401 immediately. No guessing, no fallback. Fail fast.

TYPESCRIPT

```

const token = authHeader.substring(7); // Remove "Bearer "

try {
  const decoded = jwt.verify(token, config.jwtSecret) as {
    id: string;
    email: string;
    role: string;
    listId: string | null;
  };

  req.user = {
    id: decoded.id,
    email: decoded.email,
    role: decoded.role,
    listId: decoded.listId,
  };

  next();
} catch (error) {
  res.status(401).json({
    success: false,
    error: {
      code: 'INVALID_TOKEN',
      message: 'Token is invalid or expired',
      timestamp: new Date().toISOString(),
    },
  });
}
}

```

`jwt.verify` checks the token's signature and expiration. If valid, we extract the user's data and attach it to the request. If invalid (expired, tampered, wrong secret), we return 401.

The `substring(7)` removes "Bearer " (7 characters) to get just the token. You'll see this pattern in every backend that uses JWTs.

Backend: Role Guard

```

export function requireRole(...roles: string[]) {
  return (req: AuthenticatedRequest, res: Response, next: NextFunction): void => {
    if (!req.user) {
      res.status(401).json({
        success: false,
        error: {
          code: 'UNAUTHORIZED',
          message: 'Authentication required',
          timestamp: new Date().toISOString(),
        },
      });
      return;
    }

    if (!roles.includes(req.user.role)) {
      res.status(403).json({
        success: false,
        error: {
          code: 'FORBIDDEN',
          message: `This action requires one of: ${roles.join(', ')}`,
          timestamp: new Date().toISOString(),
        },
      });
      return;
    }

    next();
  };
}

export const requireManager = requireRole('MANAGER');
export const requireMember = requireRole('MEMBER', 'MANAGER');

```

`requireRole` is another higher-order function. `requireManager` checks that the user has the `MANAGER` role. `requireMember` allows both `MEMBER` and `MANAGER` (a manager can also add items).

Backend: Update Routes with Auth

Update `apps/svc-grocery/src/interface/routes/grocery.routes.ts`:

```
import { authMiddleware, requireManager, requireMember } from './middlewares/auth.middleware'
```

TYPESCRIPT

Add the import, then update the routes:

```
// Anyone authenticated can view the list
router.get(
  '/lists/:listId',
  authMiddleware,
  controller.getListWithBudget,
);

// Members and managers can add items
router.post(
  '/lists/:listId/items',
  authMiddleware,
  requireMember,
  validateBody(createItemSchema),
  controller.createItem,
);

// Only managers can approve/reject
router.post(
  '/items/:itemId/approve',
  authMiddleware,
  requireManager,
  validateBody(approveItemSchema),
  controller.approveItem,
);

// Members and managers can mark items as bought
router.post(
  '/items/:itemId/buy',
  authMiddleware,
  requireMember,
  validateBody(markBoughtSchema),
  controller.markBought,
);
```

TYPESCRIPT

The middleware chain grew: `authMiddleware → requireRole → validateBody → controller`. Each middleware has one job:

1. `authMiddleware` : Is the user authenticated?
2. `requireRole` : Does the user have the right role?
3. `validateBody` : Is the request body valid?

4. controller : Process the request.

If any middleware rejects, the chain stops. The controller is only reached if all checks pass.

Backend: Login Endpoint

Add a simple login route for development. Create `apps/svc-grocery/src/interface/routes/auth.routes.ts`:

```
import { Router } from 'express';
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';
import { db } from '@grocery/core-db';
import { config } from '../../config';

const router = Router();

router.post('/login', async (req, res) => {
  try {
    const { email, password } = req.body;

    if (!email || !password) {
      res.status(400).json({
        success: false,
        error: { code: 'VALIDATION_ERROR', message: 'Email and password required' },
      });
      return;
    }

    const user = await db.user.findUnique({ where: { email } });

    if (!user) {
      res.status(401).json({
        success: false,
        error: { code: 'INVALID_CREDENTIALS', message: 'Invalid email or password' },
      });
      return;
    }

    const isValid = await bcrypt.compare(password, user.password);
    if (!isValid) {
      res.status(401).json({
        success: false,
        error: { code: 'INVALID_CREDENTIALS', message: 'Invalid email or password' },
      });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({
      success: false,
      error: { code: 'INTERNAL_ERROR', message: 'Internal server error' },
    });
  }
});
```

TYPESCRIPT

```

    });

    return;
}

const token = jwt.sign(
{
    id: user.id,
    email: user.email,
    role: user.role,
    listId: user.listId,
},
config.jwtSecret,
{ expiresIn: '24h' },
);

res.json({
    success: true,
    data: {
        token,
        user: {
            id: user.id,
            email: user.email,
            name: user.name,
            role: user.role,
        },
    },
});
} catch (error) {
    res.status(500).json({
        success: false,
        error: { code: 'INTERNAL_ERROR', message: 'Login failed' },
    });
}
);

export const authRoutes = router;

```

Mount it in `app.ts`:

```

import { authRoutes } from './interface/routes/auth.routes';

// In createApp(), add:
app.use('/api/v1/auth', authRoutes);

```

TYPESCRIPT

Now test it:

```
# Login
curl -s -X POST http://localhost:3060/api/v1/auth/login \
-H "Content-Type: application/json" \
-d '{"email": "manager@grocery.test", "password": "grocery123"}' \
| jq

# Use the returned token
curl -s http://localhost:3060/api/v1/grocery/lists/LIST_ID \
-H "Authorization: Bearer YOUR_TOKEN_HERE" \
| jq
```

BASH

Frontend: Session Management

For the frontend, we'll use a simple approach: store the JWT in a cookie via the BFF. Create `src/app/api/auth/login/route.ts`:

```
import { NextResponse } from 'next/server';
import axios from 'axios';
import { cookies } from 'next/headers';

export async function POST(req: Request) {
  try {
    const body = await req.json();

    // Forward login directly to the backend auth endpoint
    const backendUrl = process.env.BACKEND_URL || 'http://localhost:3060';
    const response = await axios.post(
      `${backendUrl}/api/v1/auth/login`,
      body,
    );

    const { token, user } = response.data.data;

    // Store token in HTTP-only cookie (secure, not accessible by JavaScript)
    const cookieStore = await cookies();
    cookieStore.set('auth-token', token, {
      httpOnly: true,
      secure: process.env.NODE_ENV === 'production',
      sameSite: 'lax',
      maxAge: 60 * 60 * 24, // 24 hours
      path: '/',
    });
  }
}
```

TYPESCRIPT

```

});
```

```

// Decode the JWT payload to extract listId (no verification needed - token came from a cookie)
const payload = JSON.parse(
  Buffer.from(token.split('.')[1], 'base64url').toString(),
);
const listId = payload.listId ?? null;
```

```

return NextResponse.json({
  success: true,
  data: { user, listId },
});
} catch (error) {
  return NextResponse.json(
    { success: false, message: 'Invalid credentials' },
    { status: 401 },
  );
}
}
```

The JWT is stored in an **HTTP-only cookie**. HTTP-only means JavaScript in the browser can't access it, protecting against XSS attacks. The BFF reads the cookie on every request and adds it as a Bearer token to the backend call.

We decode the JWT payload to extract `listId`; the login page needs it to redirect the user to their grocery list. No cryptographic verification is needed here. We just received this token from our own backend moments ago; we're reading our own data back.

Remember the `getAuthHeaders` function we wrote in `src/lib/backendClient.ts` back in Chapter 10? It reads the `auth-token` cookie and returns an Authorization header. At the time, there was no cookie to read -- the function just returned an empty object. Now the auth chain is complete: the login route sets the cookie, `getAuthHeaders` reads it on every BFF request, and the backend auth middleware verifies it.

Frontend: The Login Page

We have the BFF login route and the auth middleware. But users need a way to actually log in through the browser. Create `src/app/login/page.tsx`:

```
'use client';

import { useState } from 'react';
import { useRouter } from 'next/navigation';

export default function LoginPage() {
```

TYPESCRIPT

```

const router = useRouter();
const [email, setEmail] = useState('');
const [password, setPassword] = useState('');
const [error, setError] = useState('');
const [loading, setLoading] = useState(false);

const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault();
  setError('');
  setLoading(true);

  try {
    const res = await fetch('/api/auth/login', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email, password }),
    });

    const data = await res.json();

    if (!res.ok || !data.success) {
      setError(data.message ?? 'Invalid credentials');
      return;
    }

    const listId = data.data.listId;
    if (listId) {
      router.push(`grocery/${listId}`);
    } else {
      setError('No grocery list found for this account');
    }
  } catch {
    setError('Something went wrong. Please try again.');
  } finally {
    setLoading(false);
  };
}

return (
  <div className="min-h-screen flex items-center justify-center bg-gray-50">
    <div className="max-w-md w-full space-y-8 p-8 bg-white rounded-xl shadow-lg">
      <div>
        <h2 className="text-center text-3xl font-bold text-gray-900">

```

```

    Grocery List
  </h2>
  <p className="mt-2 text-center text-sm text-gray-600">
    Sign in to manage your household groceries
  </p>
  </div>

  <form onSubmit={handleSubmit} className="mt-8 space-y-6">
    {error && (
      <div className="bg-red-50 border border-red-200 text-red-700 px-4 py-3 rounded-lg">
        {error}
      </div>
    )}
  </form>

  <div className="space-y-4">
    <div>
      <label
        htmlFor="email"
        className="block text-sm font-medium text-gray-700"
      >
        Email
      </label>
      <input
        id="email"
        type="email"
        required
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        className="mt-1 block w-full px-3 py-2 border border-gray-300 rounded-md placeholder='manager@grocery.test'"
      />
    </div>

    <div>
      <label
        htmlFor="password"
        className="block text-sm font-medium text-gray-700"
      >
        Password
      </label>
      <input
        id="password"
        type="password"
      </input>
    </div>
  </div>

```

```

        required
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        className="mt-1 block w-full px-3 py-2 border border-gray-300 rounded-lg"
        placeholder="grocery123"
      />
    </div>
  </div>

  <button
    type="submit"
    disabled={loading}
    className="w-full flex justify-center py-2 px-4 border border-transparent text-white bg-blue-500 rounded-lg font-medium transition duration-150 ease-in-out"
  >
    {loading ? 'Signing in...' : 'Sign In'}
  </button>
</form>

<div className="text-center text-xs text-gray-500 space-y-1">
  <p>Test accounts:</p>
  <p>manager@grocery.test / grocery123</p>
  <p>member@grocery.test / grocery123</p>
</div>
</div>
</div>
);
}

```

A simple login form. Email and password fields, a submit button, error handling, and test credential hints at the bottom. The form POSTs to our BFF login route (`/api/auth/login`). On success, the response includes `listId`. We redirect to `/grocery/${listId}` to load the user's grocery list. On error, we display the message.

Remember the home page from Chapter 9? It redirects to `/login`. Now the user lands here, logs in, and gets redirected to their list. The full flow: `/ → /login → /grocery/{listId}`.

Test It

Login as the manager:

```

curl -s -X POST http://localhost:3000/api/auth/login \
-H "Content-Type: application/json" \
-d '{"email": "manager@grocery.test", "password": "grocery123"}'

```

BASH

The cookie is set. Now all BFF routes include the JWT automatically. The manager can approve items. A member can't. Role-based access is enforced on every request.

Principle check: *Read the error, trust the error.* When auth fails, our system tells you exactly what happened: `INVALID_CREDENTIALS` (wrong email or password), `INVALID_TOKEN` (expired, tampered, or missing JWT), `FORBIDDEN` (authenticated but lacking the required role). Each error has a code, message, and details. When you see a 403, the user is logged in but doesn't have the right role. When you see a 401, the token is missing, expired, or the credentials were wrong. Trust these errors. They were designed to guide you.

CHAPTER 13

Dockerizing Your Microservice

"Docker is a shipping container for your code. It packages your application, its dependencies, and its configuration into a single unit that runs identically everywhere: on your laptop, on a server, on a Kubernetes cluster."

What You'll See When We're Done

```
$ docker ps
```

| CONTAINER ID | IMAGE | PORTS | STATUS |
|--------------|-----------------|-----------------------|--------------|
| a1b2c3d4e5f6 | svc-grocery:1.0 | 0.0.0.0:3060→3060/tcp | Up 2 minutes |
| f6e5d4c3b2a1 | postgres:16 | 0.0.0.0:5432→5432/tcp | Up 5 minutes |

BASH

The Dockerfile

Create `apps/svc-grocery/Dockerfile`:

```
# =====
# Stage 1: Dependencies
# Install all node_modules (including devDependencies for build)
# =====
FROM node:20-alpine AS deps

WORKDIR /app

# Copy package files for all workspaces
COPY package.json package-lock.json ./
```

```
COPY apps/svc-grocery/package.json ./apps/svc-grocery/
COPY packages/core-db/package.json ./packages/core-db/
```

DOCKERFILE

Multi-stage build. The first stage installs dependencies. We copy `package.json` files first because Docker caches layers; if the package files haven't changed, Docker reuses the cached `npm install` layer. This saves minutes on every build.

```
RUN npm ci --workspace=apps/svc-grocery --workspace=packages/core-db
```

DOCKERFILE

`npm ci` is like `npm install` but stricter. It uses the lockfile exactly, deletes `node_modules` first, and fails if the lockfile is outdated. Always use `ci` in Docker, never `install`.

```
# =====
# Stage 2: Build
# Compile TypeScript to JavaScript
# =====
FROM node:20-alpine AS build

WORKDIR /app

COPY --from=deps /app/node_modules ./node_modules
COPY --from=deps /app/apps/svc-grocery/node_modules ./apps/svc-grocery/node_modules
COPY --from=deps /app/packages/core-db/node_modules ./packages/core-db/node_modules
```

DOCKERFILE

Copy `node_modules` from the deps stage. We don't reinstall. We reuse.

```
# Copy source code
COPY packages/core-db ./packages/core-db
COPY apps/svc-grocery ./apps/svc-grocery
COPY tsconfig.json .

# Generate Prisma client
RUN cd packages/core-db && npx prisma generate

# Build TypeScript
RUN cd apps/svc-grocery && npx tsc
```

DOCKERFILE

Copy source, generate Prisma client (it needs the schema to generate types), and compile TypeScript. The compiled JavaScript goes into `dist/`.

Here's a gotcha that catches almost everyone: **transitive dependencies**. If `svc-grocery` imports from `@grocery/core-db`, and `core-db` depends on `@prisma/client`, then `@prisma/client` must exist in the final image. Missing it means the build passes but the app crashes at runtime with `Cannot find module '@prisma/client'`. I've lost hours to this. Copy every package your service depends on, directly or indirectly.

```
# =====
# Stage 3: Production
# Only the compiled code and production dependencies
# =====
FROM node:20-alpine AS production

WORKDIR /app

ENV NODE_ENV=production

# Copy only what we need to run
COPY --from=build /app/apps/svc-grocery/dist ./apps/svc-grocery/dist
COPY --from=build /app/apps/svc-grocery/package.json ./apps/svc-grocery/
COPY --from=build /app/packages/core-db ./packages/core-db
COPY --from=build /app/node_modules ./node_modules
COPY package.json ./
```

DOCKERFILE

The production stage starts fresh. No dev dependencies, no build tools. TypeScript source is gone. Only the compiled JavaScript in `dist/`. The `core-db` package comes along because it has the generated Prisma client the app needs at runtime. The result: a smaller image with a smaller attack surface.

```
EXPOSE 3060

CMD ["node", "apps/svc-grocery/dist/index.js"]
```

DOCKERFILE

Expose the port and start the server. `node` (not `tsx`) -- we're running compiled JavaScript in production.

.dockerignore

Create `.dockerignore` in the **project root** (same level as `docker-compose.yml`). Docker reads `.dockerignore` from the build context root, which is `.` in our compose file:

```
node_modules
dist
.env
.env.*
*.md
.git
```

These files aren't copied into the Docker build context. Without this, Docker copies `node_modules` (hundreds of MB) before immediately deleting them to install fresh ones. The `.dockerignore` skips that wasted copy.

Docker Compose for Local Development

Create `docker-compose.yml` in the project root:

```

# Note: The 'version' field is deprecated in Docker Compose V2 (the default since 2022).
# If you're using `docker compose` (without hyphen), you can omit it entirely.

YAML

services:
  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: grocery_db
    ports:
      - "5432:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data

  svc-grocery:
    build:
      context: .
      dockerfile: apps/svc-grocery/Dockerfile
    ports:
      - "3060:3060"
    environment:
      PORT: 3060
      NODE_ENV: production
      DATABASE_URL: postgresql://postgres:postgres@postgres:5432/grocery_db
      JWT_SECRET: dev-secret-change-in-production-please
    depends_on:
      - postgres

    volumes:
      pgdata:

```

`depends_on` ensures PostgreSQL starts before the grocery service. The `DATABASE_URL` uses `postgres` (the service name) as the hostname. Docker Compose creates an internal network where services can reach each other by name.

Build and Run

```
# Build the image
docker compose build

# Start everything
docker compose up -d

# Check it's running
docker compose ps

# Test it
curl http://localhost:3060/health

# Check logs if something's wrong
docker compose logs svc-grocery
```

BASH

If the health check returns `{"status": "ok"}`, your service is running in Docker. The same code, the same behavior, but now in a container that can run anywhere.

Read the error, trust the error. If Docker build fails, read the full error output. It usually tells you exactly which file is missing or which command failed. Don't guess, read.

CHAPTER 14

Deploying to Kubernetes

"Kubernetes is the conductor of a container orchestra. It decides how many copies of your service to run, restarts them when they crash, and routes traffic to healthy instances. Docker packages your app. Kubernetes runs it at scale."

What You'll See When We're Done

```
$ kubectl get pods -n grocery-dev
NAME                  READY   STATUS    RESTARTS   AGE
svc-grocery-6d4f8b7c9-x2k4m   1/1     Running   0          2m
svc-grocery-6d4f8b7c9-j7n3p   1/1     Running   0          2m
```

BASH

Two copies of your service, running and healthy.

Push the Image

First, your Docker image needs to be in a **registry**: a server that stores and distributes Docker images. For local Kubernetes development, you can use a local registry:

```
# Start a local registry (one-time setup)
docker run -d -p 5000:5000 --name registry registry:2

# Tag and push your image
docker tag svc-grocery:latest localhost:5000/svc-grocery:1.0.0
docker push localhost:5000/svc-grocery:1.0.0
```

BASH

In production, you'd push to Docker Hub, AWS ECR, or your company's private registry.

Kubernetes Manifests

Create a `k8s/` directory:

```
mkdir -p k8s
```

BASH

ConfigMap: Non-secret configuration. Create `k8s/configmap.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: svc-grocery-config
  namespace: grocery-dev
data:
  PORT: "3060"
  NODE_ENV: "production"
```

YAML

Secret: Sensitive values. Create `k8s/secret.yaml`:

```
apiVersion: v1
kind: Secret
metadata:
  name: svc-grocery-secrets
  namespace: grocery-dev
type: Opaque
stringData:
  DATABASE_URL: "postgresql://postgres:postgres@postgres-service:5432/grocery_db"
  JWT_SECRET: "your-production-jwt-secret-at-least-32-chars"
```

YAML

Never commit the actual secret values to git. In production, use `kubectl create secret` or a secrets manager like Vault. This example uses `stringData` for readability. Kubernetes base64-encodes it automatically.

You might be tempted to put the `DATABASE_URL` in the ConfigMap because "it's just a URL." Don't. It contains the password. ConfigMaps are visible to anyone with read access to the namespace. Secrets have stricter access controls.

Deployment: Runs your containers. Create `k8s/deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: svc-grocery
  namespace: grocery-dev
  labels:
    app: svc-grocery
```

YAML

```

spec:
  replicas: 2
  selector:
    matchLabels:
      app: svc-grocery
  template:
    metadata:
      labels:
        app: svc-grocery
    spec:
      containers:
        - name: svc-grocery
          image: localhost:5000/svc-grocery:1.0.0
          ports:
            - containerPort: 3060
          envFrom:
            - configMapRef:
                name: svc-grocery-config
            - secretRef:
                name: svc-grocery-secrets
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 500m
          memory: 512Mi
      livenessProbe:
        httpGet:
          path: /health
          port: 3060
        initialDelaySeconds: 10
        periodSeconds: 30
      readinessProbe:
        httpGet:
          path: /health
          port: 3060
        initialDelaySeconds: 5
        periodSeconds: 10

```

Let me walk through the important parts:

`replicas: 2`: Two copies of your service. If one crashes, the other serves traffic while Kubernetes restarts the crashed one.

`envFrom` : Loads environment variables from the ConfigMap and Secret. Your code reads `process.env.DATABASE_URL` as usual. It doesn't know it came from Kubernetes.

`resources` : CPU and memory limits. `requests` is the minimum (guaranteed). `limits` is the maximum (if exceeded, Kubernetes throttles CPU or kills the pod for memory). These prevent one misbehaving service from starving others.

`livenessProbe` : Kubernetes hits `/health` every 30 seconds. If it fails three times, Kubernetes kills and restarts the pod. This is why we wrote that health endpoint in Chapter 8.

`readinessProbe` : Similar to liveness, but controls traffic routing. If readiness fails, Kubernetes stops sending traffic to that pod. It's still running, but it's "not ready". Maybe it's still connecting to the database.

Service: Routes traffic to your pods. Create `k8s/service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: svc-grocery
  namespace: grocery-dev
spec:
  selector:
    app: svc-grocery
  ports:
    - port: 80
      targetPort: 3060
  type: ClusterIP
```

YAML

The Service is a stable network address. Pods come and go (restarts, scaling, updates), but the Service always points to healthy pods. `port: 80` is what other services call. `targetPort: 3060` is where your app listens. Inside the cluster, other services reach yours via `http://svc-grocery.grocery-dev:80`.

Deploy

```
# Create namespace
kubectl create namespace grocery-dev

# Apply all manifests
kubectl apply -f k8s/

# Check status
kubectl get pods -n grocery-dev

# Wait for pods to be ready
kubectl wait --for=condition=ready pod -l app=svc-grocery -n grocery-dev --timeout=60s

# Port-forward to test
kubectl port-forward -n grocery-dev svc/grocery 3060:80

# Test
curl http://localhost:3060/health
```

BASH

Your service is now running in Kubernetes. Two replicas, with health checks, resource limits, and automatic restarts. The same code that ran on your laptop in Chapter 8 now runs in a cluster that can scale, heal, and route traffic automatically.

Per-Developer Namespaces

In a team environment, each developer gets their own namespace:

```
kubectl create namespace grocery-dev-sarah
kubectl create namespace grocery-dev-alex
```

BASH

Each developer deploys to their namespace. Isolated environments. No stepping on each other's toes. The manifests are the same, only the `namespace` field changes.

CHAPTER 15

Putting It All Together - The Complete Picture

"Step back. Look at what you've built. Not just the code. The system."

The Updated Life of a Request

Now with all layers in place, here's what happens when Sarah (MANAGER) approves Alex's milk request:



Thirteen steps. Each step does one thing. Each step can be tested independently. Each step can be modified without touching the others. That's the payoff of Clean Architecture.

How to Add a New Feature

When you need to add a new feature to this system, follow this checklist:

1. **Database:** Need a new table or column? Update the Prisma schema, run `prisma db push`.
2. **Domain:** New business rules? Add domain errors and value objects.
3. **Application:** New operation? Create the use case with its DTOs and update repository ports if needed.
4. **Infrastructure:** New repository method? Implement it in the Prisma repository.
5. **Interface:** New endpoint? Add the route with middleware, controller method, and validation schema.
6. **DI Container:** Wire the new use case into the container.
7. **BFF:** New frontend endpoint? Add a BFF API route.
8. **Frontend:** New UI? Create the component, hook, and page.

Always bottom-up. Database → Domain → Application → Infrastructure → Interface → Frontend. Each layer builds on the one below it.

When to Create a New Microservice

Not every feature needs a new service. Create a new microservice when:

- The feature has its own database tables and doesn't share tables with existing services
- The feature has a different scaling profile (e.g., image processing needs more CPU)
- A separate team will own and deploy it independently
- The feature could reasonably run on a different schedule (always on vs. batch processing)

If the feature is just a new set of CRUD operations on existing data, it probably belongs in an existing service. Don't create microservices for the sake of having microservices. Each one adds operational complexity (deployment, monitoring, networking).

CHAPTER 16

What's Next - Growing as an Engineer

"You've built a complete application from scratch. Every layer, every pattern, every deployment step. But this is just the beginning."

The Three Principles - Revisited

Throughout this guide, three principles guided every decision:

"Every line has a reason." We never wrote code without explaining why. The `DomainError` base class exists because errors need to be structured and debuggable. The `toReadModel()` mapper exists because database shapes and API shapes are different concerns. The Zod schemas exist because validation should be declarative, not imperative. Every line we wrote serves a purpose. If you ever find yourself writing code you can't explain, stop and think.

"Read the error, trust the error." When the Zod middleware returned `VALIDATION_ERROR` with field-level details, we didn't need to add logging or breakpoints -- the error told us exactly what was wrong. When `InvalidItemStateException` fired, it told us the current status, the required status, and the operation. Build your errors to be self-documenting, and debugging becomes reading instead of guessing.

"Ship it, then perfect it." We built the entire application without authentication first. We used `db push` instead of migrations. We hardcoded the list ID. None of that was "correct." But it got us to a running application fast. Then we layered on auth, Docker, and Kubernetes incrementally. The worst code is code that never runs. The best code is code that runs, then gets better.

Advanced Patterns to Explore

Now that you understand the fundamentals, here's where to go next:

CQRS (Command Query Responsibility Segregation): Separate your read and write models. Instead of one repository handling both queries and mutations, have dedicated read models optimized for display and command handlers optimized for writes. This is powerful for systems where read patterns differ significantly from write patterns.

Event Sourcing: Instead of storing the current state, store the events that led to the current state. "Item created", "Item approved", "Item bought". Replay these events to reconstruct any point in time. Powerful for audit trails and financial systems.

Saga Pattern: Coordinate complex multi-service transactions. If placing an order involves the grocery service, the payment service, and the notification service, a saga ensures either all succeed or all roll back.

Observability: Structured logging (Pino), distributed tracing (OpenTelemetry), and metrics (Prometheus). When your system runs 20 microservices, you need to trace a request across all of them.

CI/CD Pipelines: Automate testing and deployment. Every push runs tests. Every merge to main deploys to staging. Every tag deploys to production. No manual steps, no human errors.

The Reading List

These books shaped the architecture you just learned. Read them, not to memorize patterns, but to understand the thinking behind them.

1. **Clean Architecture** by Robert C. Martin: The dependency rule, layer boundaries, and why inner layers must never know about outer layers. Chapters 19-22 are the most relevant to what we built.
2. **Clean Code** by Robert C. Martin: Naming conventions, function design, error handling, and why code should read like well-written prose. This book changed how I name variables.
3. **Designing Data-Intensive Applications** by Martin Kleppmann: Data modeling, replication, consistency, and why we store money in cents. The best book on understanding what happens below the application layer.
4. **The Pragmatic Programmer** by Andrew Hunt & David Thomas: DRY, orthogonality, tracer bullets, and practical engineering wisdom. This book teaches you to think like an engineer, not just code like one.
5. **Domain-Driven Design** by Eric Evans: Entities, value objects, bounded contexts, and ubiquitous language. Our `Money` value object and `DomainError` hierarchy come directly from DDD principles.
6. **Patterns of Enterprise Application Architecture** by Martin Fowler: Repository pattern, Unit of Work, Data Mapper. Our `IGroceryItemRepository` interface and `PrismaGroceryItemRepository` implementation are textbook examples.
7. **Release It!** by Michael Nygard: Timeouts, circuit breakers, bulkheads, and what happens when production goes wrong. This book teaches you why we set `timeout: 10000` on Axios clients and why health checks exist.

But remember, engineers engineer. Books give you patterns. Production gives you wisdom. Read the books, then build things. Build things that break, fix them, build them better. That's how you grow.

Closing

You started with an empty directory. You now have:

- A PostgreSQL database with a well-designed schema
- A backend microservice with Clean Architecture (domain, application, infrastructure, interface)
- Manual dependency injection with zero framework magic
- Zod validation on both frontend and backend

- A React frontend with TanStack Query for server state
- A BFF layer that hides your backend topology
- JWT authentication with role-based access control
- A Docker image ready for any environment
- Kubernetes manifests for production deployment

More importantly, you understand *why* each piece exists. You can explain every layer, every pattern, every decision. You can modify any part without breaking the rest. You can add a new microservice by following the same patterns.

Every line of code is a chance to create, learn, and grow. Little by little, we shape the future one line of code at a time.

For a great legacy, and a greater impact.

APPENDIX A

Microservice Scaffold Template

Use this template when creating a new microservice. Copy the structure, rename the files, and fill in the blanks.

Directory Structure

```
mkdir -p apps/svc-YOUR_SERVICE/src/{domain/errors,application/{dto/{request,response},ports/r BASH
```

Boilerplate Files

1. Domain Error Base (`domain/errors/index.ts`):

TYPESCRIPT

```
export abstract class DomainError extends Error {  
    public readonly code: string;  
    public readonly statusCode: number;  
    public readonly timestamp: string;  
    public readonly details?: Record<string, unknown>;  
  
    constructor(  
        code: string,  
        statusCode: number,  
        message: string,  
        details?: Record<string, unknown>,  
    ) {  
        super(message);  
        this.name = this.constructor.name;  
        this.code = code;  
        this.statusCode = statusCode;  
        this.timestamp = new Date().toISOString();  
        this.details = details;  
        Error.captureStackTrace(this, this.constructor);  
    }  
}  
  
// Add your specific errors:  
// export class YourEntityNotFoundError extends DomainError { ... }
```

2. Config (config.ts):

TYPESCRIPT

```

import { z } from 'zod';

const configSchema = z.object({
  port: z.coerce.number().int().min(1000).max(65535),
  nodeEnv: z.enum(['development', 'production', 'test']).default('development'),
  databaseUrl: z.string().min(1),
  jwtSecret: z.string().min(16),
  // Add your service-specific config here
});

function loadConfig() {
  const result = configSchema.safeParse({
    port: process.env.PORT,
    nodeEnv: process.env.NODE_ENV,
    databaseUrl: process.env.DATABASE_URL,
    jwtSecret: process.env.JWT_SECRET,
  });

  if (!result.success) {
    console.error('Configuration error:');
    result.error.errors.forEach((err) => {
      console.error(` ${err.path.join('.')}: ${err.message}`);
    });
    process.exit(1);
  }

  return result.data;
}

export const config = loadConfig();

```

3. DI Container (shared/di/container.ts):

TYPESCRIPT

```

// 1. Infrastructure
import { PrismaYourEntityRepository } from '../../infrastructure/repositories/prisma-your-entity';
const yourEntityRepository = new PrismaYourEntityRepository();

// 2. Use cases
import { CreateYourEntityUseCase } from '../../application/use-cases/your-domain';
const createYourEntityUseCase = new CreateYourEntityUseCase(yourEntityRepository);

// 3. Controllers
import { YourController } from '../../interface/controllers/your.controller';
const yourController = new YourController(createYourEntityUseCase);

export const container = { yourController };

```

4. App Factory (app.ts):

```
import express from 'express';
import { createYourRoutes } from './interface/routes/your.routes';
import { errorHandler } from './interface/middlewares/error-handler.middleware';
import { container } from './shared/di/container';

export function createApp(): express.Express {
  const app = express();
  app.use(express.json({ limit: '10mb' }));
  app.get('/health', (_req, res) => {
    res.json({ status: 'ok', timestamp: new Date().toISOString() });
  });
  app.use('/api/v1/YOUR_DOMAIN', createYourRoutes(container.yourController));
  app.use(errorHandler);
  return app;
}
```

TYPESCRIPT

5. Entry Point (index.ts):

```
import { createApp } from './app';
import { config } from './config';

const app = createApp();
app.listen(config.port, () => {
  console.log(`svc-YOUR_SERVICE listening on port ${config.port}`);
});
```

TYPESCRIPT

New Service Checklist

- Create directory structure
- Copy boilerplate files
- Define Prisma models in `packages/core-db/prisma/schema/`
- Run `prisma db push` or `prisma migrate dev`
- Define domain errors
- Define repository ports (interfaces)
- Define DTOs (Zod schemas)
- Implement use cases

- Implement repositories
 - Create controller
 - Create routes with middleware
 - Wire DI container
 - Create config with Zod validation
 - Create app factory and entry point
 - Write unit tests for use cases
 - Test with curl
 - Create Dockerfile
 - Create K8s manifests
 - Deploy to dev namespace
-

APPENDIX B

Glossary

BFF (Backend For Frontend): A server-side layer that sits between your frontend and your microservices. Think of it as a personal assistant for your React app. It handles authentication, routes requests to the right microservice, and transforms responses into what the frontend needs.

Clean Architecture: An architectural pattern where code is organized in concentric layers. The inner layers (domain, application) contain business rules. The outer layers (infrastructure, interface) handle external concerns. Dependencies only point inward. The domain never knows about the database.

Controller: The thinnest layer in the interface. It receives an HTTP request, extracts the relevant data, calls a use case, and formats the response. If your controller has business logic, something is wrong.

Dependency Injection (DI): Instead of a class creating its own dependencies (`new DatabaseConnection()`), dependencies are passed in through the constructor. This makes classes testable (pass a mock) and swappable (pass a different implementation).

DI Container: The "assembly line" that creates all objects and wires their dependencies. In our system, it's a simple TypeScript file. No framework magic. Read it and you understand the entire application's wiring.

Domain Error: A structured error that represents a business rule violation. Unlike generic errors (`throw new Error('...')`), domain errors have a code, status code, timestamp, and structured details. They're machine-readable and human-debuggable.

DTO (Data Transfer Object): A structure that defines the shape of data moving between layers. Request DTOs define what the API accepts. Response DTOs define what the API returns. They're contracts.

Middleware: A function that sits in the request pipeline and processes the request before it reaches the controller. Auth middleware checks credentials. Validation middleware checks the request body. Error handler middleware catches errors and formats responses.

Monorepo: A single repository containing multiple related projects. Our backend service and shared packages live in the same repo, so a change to the database package is immediately visible to all services.

Port: An interface that defines what an outer layer must provide. "I need something that can save items" -- that's a port. The infrastructure layer provides the concrete implementation (a Prisma repository). The use case depends on the port (interface), never on the implementation.

Prisma: An ORM (Object-Relational Mapper) for TypeScript. It generates type-safe database queries from a schema file. Instead of writing SQL, you write TypeScript. Prisma translates it to SQL and gives you results with full type safety.

Query Key: A unique identifier for cached data in TanStack Query. `['grocery', 'list', 'abc-123']` identifies the grocery list with ID abc-123. Invalidating a query key triggers a refetch.

Read Model: The shape of data as the application layer sees it. It might combine fields from multiple database tables, rename columns, or compute derived values. The repository maps database rows to read models.

Repository: A class that abstracts data access. The application layer calls `repository.findById(id)` without knowing if the data comes from PostgreSQL, MongoDB, or an in-memory array. This abstraction makes the application testable and database-agnostic.

Use Case: A single business operation. "Create a grocery item," "Approve an item," "Calculate the budget." Each use case is a class with an `execute` method that receives input, runs business rules, and returns output. It's the core of your application logic.

Value Object: A domain concept defined by its value, not its identity. Money(\$5) is Money(\$5), regardless of which particular \$5 bill it is. Value objects validate their own invariants (Money can't be negative) and provide domain-specific operations (Money.add, Money.subtract).

Cache Invalidation: The process of marking cached data as stale so it gets refetched. In TanStack Query, invalidating a query key triggers a background refetch, ensuring the UI shows fresh data after a mutation.

Higher-Order Function: A function that takes another function as an argument or returns a function. Our `createGroceryRoutes(controller)` is one -- it takes a controller and returns a configured Router. Middleware factories like `validateBody(schema)` are another example.

HTTP-only Cookie: A cookie that JavaScript cannot read via `document.cookie`. It's only sent automatically with HTTP requests. We use it for JWT storage because it protects against XSS attacks -- even if an attacker injects JavaScript, they can't steal the token.

Liveness Probe: A Kubernetes health check that determines if a container is alive. If it fails, Kubernetes restarts the container. Our `/health` endpoint serves this purpose.

Multi-stage Build: A Docker technique where you use multiple `FROM` statements. The first stage compiles/builds the application. The final stage copies only the production artifacts, resulting in a much smaller image.

Readiness Probe: A Kubernetes health check that determines if a container is ready to receive traffic. Unlike liveness probes, a failed readiness probe removes the pod from the service load balancer without restarting it.

Response Envelope: A wrapper structure around API responses: `{ success: true, data: ... }` or `{ success: false, error: ... }`. The frontend checks `success` first, so it never guesses whether the response is data or an error.

Singleton Pattern: A design pattern that ensures only one instance of a class exists. Our Prisma client uses this: in development, we store the client in `globalThis` so hot-reloads don't create multiple database connection pools.

State Machine: A model where an entity can be in one of a finite number of states, with defined transitions between them. Our item status (PENDING → APPROVED → BOUGHT → ARCHIVED) is a state machine. Invalid transitions are rejected.

Zod: A TypeScript-first schema validation library. Define the shape and rules once, get both runtime validation and compile-time types. We use Zod for request validation (backend), form validation (frontend), and configuration validation (startup).

APPENDIX C

Document History

| Version | Date | Description |
|---------|---------------|--|
| v1.0.0 | February 2026 | Initial release. Complete guide covering backend, frontend, auth, Docker, and Kubernetes |

Written by Manazir Ali, Full-Stack Software Engineer.

Every line of code is a chance to create, learn, and grow. Little by little, we shape the future one line of code at a time. For a great legacy, and a greater impact.

About the Author

Manazir Ali is a full-stack software engineer who builds production financial blockchain systems. His work spans the entire stack: from React frontends to NestJS microservices, from PostgreSQL databases to Hyperledger Fabric smart contracts, and from Docker containers to Kubernetes clusters.

The patterns in this book are drawn directly from his experience engineering a production financial protocol, where a single bug in a transfer function can mean real money disappearing. That kind of pressure teaches discipline, methodology, and deliberate engineering.

He believes that every line of code is a chance to create, learn, and grow, and that little by little, we shape the future one line of code at a time.