PART I

PROJECT FOUNDATION

"The beginning is the most important part of the work."

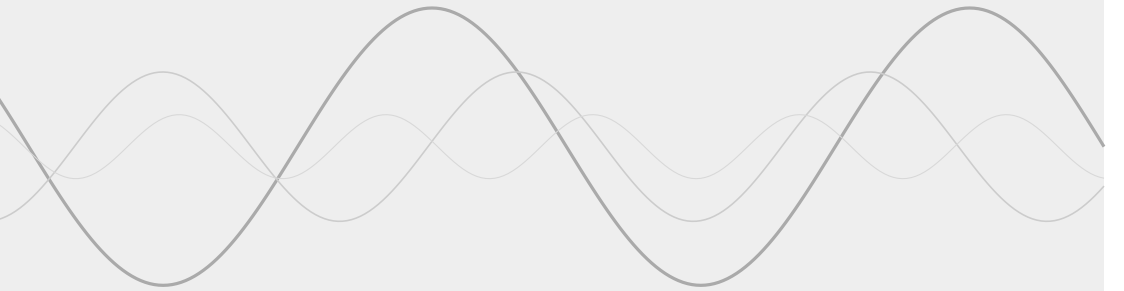
— Plato, The Republic

Chapter 1

Welcome to the NetBanking Project

In this chapter

- Who this book is for and what prior knowledge is assumed
- The NetBanking portal — a complete feature-by-feature overview
- The technology stack and the reason each tool was chosen
- How every feature chapter is structured as a vertical slice
- The eight-part roadmap of this volume
- Habits and practices that make working through this book effective



1.1 Who This Book Is For

You already write Java. You know what a class is, what an interface does, and how to run something in a terminal. You have probably connected to a database before. But you have never built a complete application from start to finish, one with a login system, a database, a REST API, a frontend, and tests that actually verify the code does what it claims. That gap, between knowing the language and building something real, is exactly what this book closes.

This book is also for developers who have used Spring Boot on a project but inherited an existing codebase and never set one up from scratch. If you can add a method to a service class but are not sure how security, database migrations, and integration tests connect to each other, this book shows you how all those pieces work together in a single codebase you build yourself.

If you have completed online tutorials and finished them able to copy examples but still unsure how to start a project from a blank folder, this book is for you.

1.1.1 What This Book Expects From You

You need to know the items on the left. Everything on the right is taught in this book, from first principles.

You already know	This book teaches
Basic Java, variables, methods, classes	Spring Boot from blank project to production
What HTTP requests and responses are	React and the JavaScript frontend
How to run commands in a terminal	Docker and container-based deployment
What a relational database table is	Banking domain, accounts, transfers, KYC, lending

You already know	This book teaches
Basic Git, add, commit, push	Agile, CI/CD pipelines, and professional team practices

> **A note on language**
>
> This book avoids idioms and informal expressions so the meaning stays clear for readers whose first language is not English. Technical terms are explained the first time they appear. Every code listing is explained line by line in the surrounding text. If a sentence is confusing, that is a writing problem, not a reading one.

Each chapter fits into one or two focused sessions of two to three hours. You can stop at any numbered section boundary and pick up again later. The code is always in a working state at those points.

Some chapters are longer than others. The fund transfer chapter is the longest because fund transfer is the most complex feature in the application. If a chapter is hard, that reflects the real difficulty of the problem, not a gap in the explanation.

1.2 What You Will Build

The project you work on throughout all four volumes is the **NetBanking Portal**, an internet banking application where customers manage accounts, move money, pay bills, handle cards and cheques, and apply for loans from a web browser. A bank administrator can approve customer registrations, verify identity documents, and control accounts.

This is not a demonstration project. It follows the same rules a real bank applies: money is stored as a decimal, never a floating-point value. Every transfer writes two matching ledger entries that must always balance. Sending the same transfer request twice produces one transfer, not two. Every action that touches money is written to an audit log that cannot be changed after the fact. These rules are not extras, they are why every design decision in the code is made the way it is.

Most tutorial applications let you skip these concerns. A to-do list app does not care if you insert the same task twice. A blog does not need to reconcile two ledger entries after every post. A banking application has no shortcuts. If a fund transfer subtracts from one account, it must add to another. If those two operations do not both succeed or both fail together, money is either created from nothing or destroyed. That constraint forces you to learn database transactions properly in a way that optional exercises never quite manage.

The application you build in this series handles real business rules: a beneficiary must wait 24 hours after registration before the first transfer goes through. A customer account locks after five consecutive failed login attempts. A standing instruction that runs on the first of every month must still run correctly if the first falls on a weekend. These details are not invented for the book. They are the actual requirements in production banking systems.

These rules also make excellent learning material because they are verifiable. After you build the login lockout feature in Chapter 12, you can test it yourself: send five wrong passwords, then try a sixth. The account locks. You built something real, and you can prove it works. That feedback loop, building a rule and then testing it directly, is what turns reading into skill.

By the end of this series you will have built software that handles money correctly, scales under load, deploys without downtime, and passes a test suite that covers every feature end to end. That is a portfolio piece you can point to in any technical interview and walk through line by line. No other format, not a course, not a bootcamp, not a tutorial series, leaves you with something this complete and this yours.

The project runs on your laptop from Chapter 8 onwards. You will register a customer, log in, open a bank account, transfer money between accounts, add a beneficiary, pay a bill, request a cheque book, apply for a loan, and view a statement, all through a React interface you built yourself talking to a Spring Boot backend you wrote from scratch. Every step is verified by automated tests you wrote alongside the feature.

The codebase also grows in a way that mirrors real team projects. Early chapters establish the foundation: the database schema, security configuration, and shared utilities. Later chapters add features on top of that foundation without breaking anything already built. When you reach Chapter 25, the application you started in Chapter 8 is still running, still tested, and now deployable as a Docker container through a GitHub Actions pipeline.

Figure 1.1, The eight feature areas of the NetBanking portal. All eight are built across the four volumes of this series.

By the end of Volume 1 you will have a working implementation of all eight feature areas shown above, with a React frontend, a fully tested Spring Boot backend, and a running CI/CD pipeline. Each area is fully tested, documented, and deployable. Volume 2 builds additional features and scalability on top of this foundation.

1.2.1 Why a Banking Application

Banking is one of the most demanding areas to write software for. The rules are strict: money must not go missing, duplicate requests must be caught before they do damage, every action must be recorded, and access must be tightly controlled. Working within these rules teaches you the techniques that separate code that works in a tutorial from code that works in a production system.

Once you build a fund transfer with double-entry accounting and idempotency, you understand database transactions and concurrency in a way that a to-do list application cannot teach. Once you build a KYC state machine, you understand state management in a way that applies to any domain, online orders, insurance claims, hospital admissions. Banking is genuinely hard, and that difficulty is what makes it worth building.

> **Industry context**
>
> International banks process over 12 billion UPI transactions every month. The engineers who build and maintain these systems use the same Spring Boot, Kafka, and Kubernetes stack you learn in this series. The patterns in this book come from production architectures used by major International financial institutions.

1.2.2 What You Will Own at the End of Volume 1

When you finish the last chapter of this volume, you will have a complete, tested, deployable application. Here is exactly what it contains.

- A Spring Boot 4 application with JWT authentication and role-based access control. Different users see different parts of the application.
- A MySQL 8 database with 17 tables, every one created through Flyway migrations so the schema is always reproducible on any machine.
- Customer registration with PAN validation and a KYC state machine that moves from PENDING to UNDER REVIEW to APPROVED.
- Account opening for savings and current account types, with balance inquiry and mini statement.

- Fund transfer with double-entry ledger entries, six validation checks before any money moves, idempotency enforcement, and support for NEFT, IMPS, UPI, and RTGS channels.
- Beneficiary management with IFSC validation and a 24-hour waiting period before the first transfer to a new beneficiary.
- Bill payment with a biller registry and a standing instruction scheduler that runs recurring payments automatically.
- Notifications sent asynchronously after every transaction.
- Card management with PCI-DSS compliant PIN storage and transaction limit controls.
- Cheque book requests and stop-cheque with leaf-range validation.
- Loan applications with amortisation schedule generation, EMI auto-debit, prepayment, and foreclosure.
- A React 19 frontend with a dedicated page for every feature above.
- An admin portal for approving KYC submissions and controlling accounts.
- Integration tests that run against a real MySQL database using Testcontainers.
- A GitHub Actions CI pipeline that runs all tests on every push, plus a CD pipeline that builds and publishes Docker images.

This book uses exact versions throughout: Spring Boot 4.0, JDK 21, and React 19. Matching the version is the easiest way to make sure every exercise works exactly as described. If a newer version exists when you read this, the concepts are identical. Only specific API names may differ. The GitHub repository at https://github.com/tektutor/software-professional-blueprint-book.git notes any version-specific updates in its README.

1.3 The Technology Stack

Every tool in this book was chosen deliberately. The diagram below shows how the layers sit on top of each other. The table that follows explains what each tool does and why it was chosen.

Every tool here is widely used in International technology companies and has strong community support. You will find job postings that list these exact skills, colleagues who already know them, and a large community to answer questions. Books built on experimental tools go out of date within a year. This one is built to last.

Figure 1.2, Technology stack layers used in the NetBanking portal. Docker and GitHub Actions wrap every layer.

Tool	Version	Role in the project
Spring Boot	4.0	The main application framework. It handles HTTP requests, dependency injection, database access, and security with very little configuration. Version 4 requires JDK 17 minimum and includes stable support for virtual threads.
JDK	21	The Java runtime. Version 21 is the current long-term support release and the version used throughout this book.
MySQL	8.0	The relational database for Volumes 1 and 2. ACID transactions, row-level locking, and JSON column support are all used.

Tool	Version	Role in the project
Flyway	10.x	Database migration tool. Every schema change is written as a numbered SQL file and stored in version control. The database builds itself automatically from these files on first startup.
Spring Security	6.x	Handles authentication and authorisation. JWT tokens are checked on every request. Method-level security controls which roles can call each endpoint.
React	19	The frontend library for the user interface. Functional components and hooks are used throughout. Redux Toolkit is added in Volume 2 for more complex state management.
Testcontainers	1.19+	Starts a real MySQL database inside each integration test. Tests run against the same database behaviour as production.
Docker	26+	Packages the application into a container image. Also used by Testcontainers during testing.
GitHub Actions	-	Runs the CI pipeline on every code push: compile, test, build image, push to registry. The CD pipeline deploys to staging.

Why not Gradle? Why not PostgreSQL?

Maven was chosen over Gradle because its structure is more visible to someone learning, every dependency and plugin is written out explicitly in pom.xml. Gradle is introduced in Volume 3 where its build speed becomes more relevant. MySQL was chosen over PostgreSQL because MySQL is more common in International enterprise environments and easier to find help for. All patterns in this book work identically on PostgreSQL.

1.4 How Each Chapter Is Structured

Every feature chapter delivers one complete working feature, not half a feature you finish three chapters later. This is called vertical slice development. A vertical slice cuts through all layers of the application at once, database, backend, frontend, and tests, for one feature. You always end a

chapter with something you can run.

This is how professional Agile teams work. At the end of a two-week sprint, a team ships working software, not a finished database layer sitting idle while the backend catches up. The diagram below shows the six steps every feature chapter follows, in this order.

Every feature chapter in this book follows these six steps in order

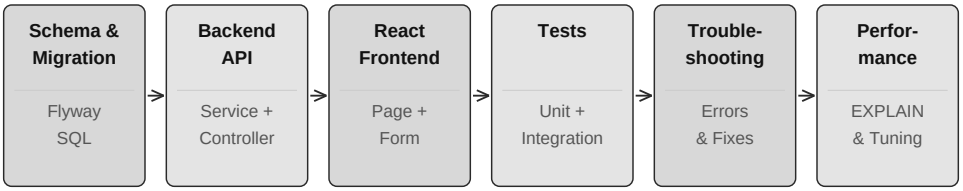


Figure 1.3, The six-step structure used by every feature chapter.

Step	What it covers
Schema and Migration	The database tables for this feature are designed first, then added as a Flyway migration script. The script runs automatically on startup and is stored in version control alongside the code.
Backend API	The JPA entity, Spring Data repository, service layer, and REST controller are built in that order. Request and response objects are mapped with MapStruct.
React Frontend	The React page for this feature is built and connected to the API using Axios. Loading states, error messages, and form validation are included. No feature ships without a working screen.
Tests	Unit tests check the service layer. Slice tests check the repository and controller layers. An integration test runs the whole feature against a real MySQL database using Testcontainers.

Step	What it covers
Troubleshooting	The three to five errors that appear most often when building this feature are shown with their full error message, the cause, and the exact fix.
Performance	The slow SQL query this feature introduces is found using EXPLAIN. The index that fixes it is added. A basic load test is provided.

Foundation chapters, the earlier chapters that build your understanding of Spring Boot before the first feature chapter, follow a shorter version of this structure. They prepare you to use the full six steps confidently when feature chapters begin. By Chapter 12 the structure will feel natural, because you will have followed it four times already.

1.5 The Roadmap of This Volume

Volume 1 is divided into eight parts. Each part builds on the one before it. Work through them in order. The first three parts establish the domain knowledge, tooling, and application foundation. Parts four through eight implement the actual banking features, one per sprint.

Part	Chapters	What you build
Part I, Project Foundation	1 - 3	Domain knowledge, tooling, and Agile practices. You understand the problem before writing a line of code.
Part II, Spring Boot Core	4 - 7	Development environment, Git, JIRA, and database schema. The project structure is in place and connected to MySQL.
Part III, Security and Onboarding	8 - 11	JPA, service layer, REST controllers, and Spring Security. Customers can register and log in.
Part IV, Core Banking Features	12 - 15	Account opening, fund transfer, and beneficiary management. Core banking works end to end.

Part	Chapters	What you build
Part V, Payments and Scheduled Operations	16 - 19	Bill payment, standing instructions, and notifications. Recurring payments run automatically on a schedule.
Part VI, Cards, Cheques, and Loans	20 - 22	PCI-compliant card management, cheque operations, and a full loan lifecycle with EMI scheduling.
Part VII, Account Services and Admin	23 - 24	Statement generation, profile management, and an admin portal for KYC approvals and account controls.
Part VIII, Testing, CI/CD, and Deployment	25	Code coverage, a CI pipeline, Docker, and CD deployment. The application runs in a container.

1.6 Setting Up for Success

Before you move to Chapter 2, put four habits in place. They will save you time throughout the entire series.

1.6.1 Work in a Dedicated Folder

Create one folder on your computer and use it for all four volumes. The codebase grows continuously, the project you start in Chapter 4 is the same project you deploy to Kubernetes in Volume 4. Keeping everything in one place makes that continuity easy to manage.

Create the project folder

```
# macOS or Linux
mkdir -p ~/projects/netbanking
cd ~/projects/netbanking

# Windows PowerShell
New-Item -ItemType Directory $HOME\projects\netbanking
Set-Location $HOME\projects\netbanking
```

1.6.2 Type the Code, Do Not Copy and Paste

Every code listing in this book is complete and correct. Copying and pasting it is tempting. Do not do it. Typing the code, even when you make small mistakes and fix them, builds understanding that copying skips. The small errors you make while typing are often more useful than the code itself, because fixing them shows you exactly why the code is written the way it is.

1.6.3 Commit After Every Section

Git saves snapshots of your code as you work. Each snapshot is called a commit. If your code breaks two sections from now, you do not restart the whole chapter. You go back to the last commit where everything worked and carry on from there.

Build this habit now: finish a numbered section, check that the application still runs, then commit. Write a short message that says what you just did, for example, **feature: add customer registration endpoint**. That message tells you exactly what this snapshot contains when you look back at it weeks later.

After working through five or six chapters, your git log will tell a clear story. Here is what a good log looks like:

Example git log after Chapter 9

```
feature: add SecurityConfig and JWT filter
feature: add CustomerService registration logic
feature: add AccountController and open-account endpoint
fix: correct IFSC validation regex
test: add integration test for login flow
refactor: extract CustomerNumberGen to its own class
```

Anyone reading that log knows exactly what you built and in what order. Compare it to a log full of wip, more stuff, and final version. The well-labelled one is easier to maintain, easier to review, and much easier to debug when something breaks.

This book uses the Conventional Commits convention throughout. The prefix before the colon signals the type of change: feat for a new feature, fix for a bug fix, test for test changes, refactor for restructuring without behaviour change, and docs for documentation updates. Most professional teams use this convention and many CI tools recognise it to generate release notes automatically.

> **Git commit discipline**
>
> Experienced developers commit small, focused changes several times a day, not one large bundle at the end of the week. Each commit should do one thing, and its message should say clearly what that one thing is. A project history full of messages like 'fixed stuff' or 'wip' tells you nothing useful when you need to find where a bug was introduced. Start writing good commit messages now. It takes almost no extra time and saves a lot of it later.

A good commit message uses a short prefix to signal the type of change: feature: for a new feature, fix: for a bug fix, test: for adding or updating tests, refactor: for restructuring code without changing behaviour, and docs: for documentation. This convention is called Conventional Commits and is used by most professional teams. Following it from the start means your project history is readable by anyone, including yourself six months from now.

You can always look at what changed in a commit later by running git show followed by the commit hash. The hash is the long alphanumeric string shown next to each commit in git log. You do not need to type the full hash. The first seven characters are enough to identify a commit uniquely in any normal-sized project.

Useful Git commands

```
# See the last 5 commits
git log --oneline -5

# See what changed in a specific commit
git show a3f92b1

# Go back to how things were at a specific commit
git checkout a3f92b1
```

```
# Come back to the latest code
git checkout main
```

1.6.4 Run the Application After Every Chapter

Every chapter ends with a working application. Run it. Open the React page in a browser. Make an API call. Check that it behaves the way the chapter describes before you move on. Skipping this step is the most common reason developers spend an hour debugging a confusing error that was actually introduced two chapters earlier.

The quickest way to confirm the backend is running correctly is the Actuator health endpoint. With the application started, run this in a terminal:

```
Verify the application is healthy
curl http://localhost:8080/actuator/health

# Application running and MySQL connected:
# {"status":"UP"}

# MySQL not running or wrong password:
# {"status":"DOWN"}
```

A status of UP means Spring Boot started, connected to MySQL, and all 17 database tables exist. A status of DOWN means the database connection failed. Check that MySQL is running and that the password in application-local.yml matches your MySQL installation. Chapter 8 covers this in full detail.

For chapters that include a React frontend, open http://localhost:5173 in your browser after starting the frontend with npm run dev. Every React page shows a loading spinner while it fetches data from the backend, then displays the result. If you see the spinner but no data, check the browser developer console for a network error. The most common cause is the backend not running on port 8080.

End of chapter checklist	Command
Compile succeeds	mvn clean compile
Unit tests pass	mvn test -Dtest=UnitTests
Application starts	mvn spring-boot:run -Dspring-boot.run.profiles=local
Health endpoint responds	curl http://localhost:8080/actuator/health
Integration tests pass	mvn test -Dspring.profiles.active=local -Dtest=NetBankingIntegrationTest

1.7 What Comes Next

The next two chapters finish Part I. Chapter 2 explains how software teams work in practice, Agile, Scrum, JIRA, and how a team decides when a piece of work is truly done. Chapter 3 covers the banking domain: accounts, transactions, double-entry accounting, idempotency, and the payment channels you implement in Chapter 15.

Part II starts the practical setup work. Chapter 4 installs every tool you need on Windows, macOS, and Linux. Chapter 5 sets up Git and GitHub. Chapter 6 builds the JIRA backlog that tracks your work across all four volumes. Chapter 7 designs the complete database schema before a single Java entity is written.

> **How to get the most from this book**
>
> Read each chapter once before you start typing. Knowing where the chapter is going makes each step easier to follow when it references something that comes later. Then go back to the beginning and work through the code, committing after each section. If you get stuck, the Troubleshooting section near the end of each feature chapter covers the errors that come up most often.

The complete source code for this book can be downloaded from: **https://github.com/tektutor/software-professional-blueprint-book.git**