THE NOVADEV BOOK

From standalone programs to generated web applications

```
app ProofExpress {
    project {
        frontend VueVite
        backend Express
        database SQLite
    }
}
```

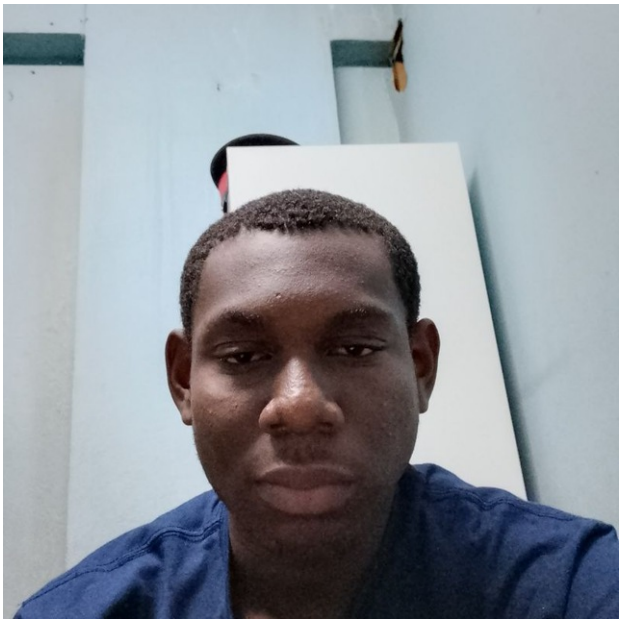
A practical language and compiler course

Author: Aldane Hutchinson

Canonical source: docs/book/

About the author

Aldane Hutchinson



Aldane Hutchinson, author of The NovaDev Book

This book teaches NovaDev in two connected movements: first the standalone language, then the application compiler. The examples are designed for the current Vue/Vite + Express/Node path and the repository's verified source-of-truth boundary.

Reference pointer: Python Crash Course, 3rd edition, by Eric Matthes. It is used only as a pacing and completeness reference; this book's syntax and examples are original.

How to use this book

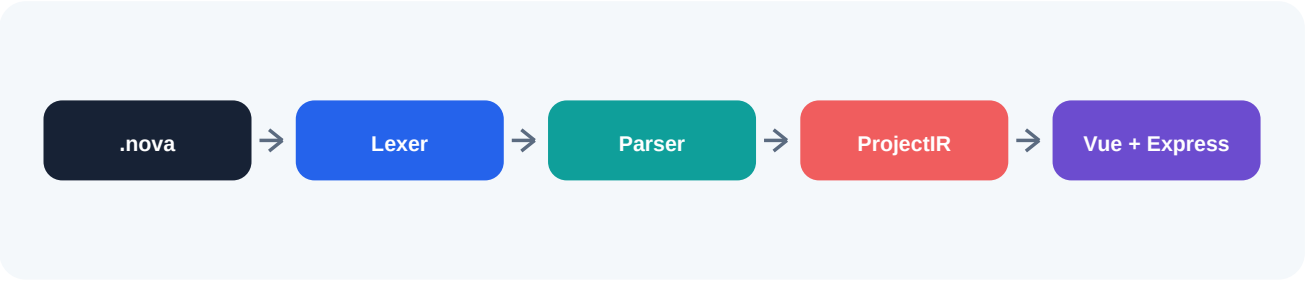
A generated project is a real editable application, not a screenshot or a hidden runtime.

Run each small example, validate application declarations before generating, and keep an evidence ledger for runtime behavior. The website and Vue IDE link to this same Markdown source.

Contents

Chapter 1 - Orientation and the source-of-truth rule

Page 1: What NovaDev is



NovaDev is both a small general-purpose language and a declaration language for web applications. The first side teaches programs that stand on their own. The second side lets a project describe data, screens, behavior, and integration boundaries. The compiler turns those declarations into editable source files; it does not replace the generated application with a hidden runtime.

The most useful mental model is a translation pipeline: source files are lexed, parsed into an AST, converted into ProjectIR, validated, and then handed to the selected generators. A generated project belongs to the developer. Open it, test it, and change it like any other Vue/Vite and Express/Node project.

Chapter 1 - Orientation and the source-of-truth rule

Page 2: Current stack

Current application lessons use Vue/Vite on the frontend, Express/Node on the backend, Tailwind for styling intent, and SQLite through the generated Prisma schema/client contract. This is the stack to use in examples. Flask, SQLAlchemy, Vue CDN, and static HTML are historical targets in this repository, not alternatives learners should select.

When a document disagrees with `docs/SOURCE_OF_TRUTH.md`, the source-of-truth document wins. A historical note is useful when it explains why an old file exists, but it must never look like a current command. This distinction prevents a learner from debugging a target the compiler no longer accepts.

Chapter 1 - Orientation and the source-of-truth rule

Page 3: The learner loop

Every lesson follows a short loop: write a small source file, validate it, run the smallest relevant command, inspect the result, and record evidence. Small feedback beats a large untested project. For standalone code, use the Nova interpreter or CLI. For an application, begin with `validate` and then `build-fullstack`.

The loop also teaches professional habits. Keep generated output separate from source, keep secrets out of examples, read diagnostics before changing code, and rerun the check after every fix. A passing lesson is not a feeling; it is a command result or a visible behavior.

Chapter 1 - Orientation and the source-of-truth rule

Page 4: First exercise

Create `hello.nova` with a `print("Hello from NovaDev")` statement. Run the command that your current CLI exposes for interpreting or validating a standalone file. If the command differs between IDE and terminal, record both commands in your notes rather than silently guessing.

Then change the string, introduce a `let` binding, and print the binding. The goal is not the greeting. The goal is learning where source lives, how a command is selected, and what evidence a successful run produces.

Chapter 1 - Orientation and the source-of-truth rule

Page 5: Checkpoint

Explain the compiler path in your own words. Name the frontend, backend, styling, and database defaults. Explain why a historical Flask example should not be copied into a current lesson. Finally, write one rule for secrets: a real API key belongs in local environment configuration, never in a `.nova` example, generated project, website, or commit.

Chapter 2 - The compiler loop and CLI

Page 1: Commands are contracts

CLI commands are interfaces, not magic incantations. Before running one, identify its input, its output, and whether it changes files. `validate` reads source and reports syntax/semantic status. `build-fullstack` creates an editable project directory. Explain and documentation commands should remain read-only. This habit makes it easier to diagnose failures and safe to repeat builds.

Use paths that exist in the current checkout. The repository keeps the proof fixture under `novadev/examples/apps/proof_express/app.nova`; a copied command from an old document may point at a directory that was deleted during the redo.

Chapter 2 - The compiler loop and CLI

Page 2: Validation first

Validation should be the first application command. It catches malformed declaration blocks, missing fields, unsupported target names, and references to unavailable components before generation creates a confusing directory tree. A clean validation result is necessary but not sufficient: generation and runtime checks still matter.

When validation fails, read the first diagnostic. Later errors may be consequences of one unclosed block. Check line and column numbers against the source, especially inside lists. The parser treats a statement-per-line declaration as clearer than a long comma-filled line.

Chapter 2 - The compiler loop and CLI

Page 3: Building output

The canonical build shape is:

```
project-root/
  frontend/   Vue/Vite source and package manifest
  backend/    Express source, Prisma schema, and package manifest
  render.yaml deployment contract when requested
  README.md   generated run instructions
```

Generated code is intentionally ordinary. The frontend has Vue components and Vite configuration. The backend has Express routes, models, authentication, configuration, and tests. The build should not copy the compiler's local OpenRouter key into any of those directories.

Chapter 2 - The compiler loop and CLI

Page 4: Evidence ledger

Keep a small ledger while working: command, working directory, exit status, and one meaningful output line. For a generated app, include validation, JavaScript syntax checks, frontend build, backend database initialization, health endpoint, authentication rejection and success, a CRUD write, a workflow call, and a protected route. If a host cannot complete dependency installation, mark the runtime row as blocked instead of upgrading a static check into a runtime claim.

Chapter 2 - The compiler loop and CLI

Page 5: Exercise

Take a deliberately broken `.nova` file. Remove a closing brace, run validation, and use the reported location to fix it. Then change `backend Express` to an unsupported backend name and observe the semantic error. Restore the canonical target and generate into a fresh directory. Compare the output tree with the source declaration and explain one generated file's responsibility.

Chapter 3 - Values, variables, and names

Page 1: Values

Programs manipulate values. NovaDev lessons use strings, numbers, booleans, nil, lists, and objects. A literal is a value written directly in source. A variable is a name that refers to a value. Keep the first examples small so you can predict the result before running them.

```
let retries = 3
let ready = true
let label = "import queue"
print(label)
```

The code shows three different kinds of data and one output operation. It also gives every value a useful name. Naming is not decoration; it makes later conditions and function calls readable.

Chapter 3 - Values, variables, and names

Page 2: Binding and reassignment

`let` introduces a binding. If the language surface allows reassignment in the current runtime, use it for changing state; otherwise create a new binding or follow the diagnostic's guidance. Do not assume Python or JavaScript rules apply automatically. NovaDev's parser and interpreter are the authority.

A good exercise is a counter: initialize a number, print it, add one, and print the new value. Ask whether the source is clearer as one mutable name or several named snapshots. The answer is a design choice you can test with the current interpreter.

Chapter 3 - Values, variables, and names

Page 3: Truth and absence

Boolean values express a deliberate yes/no state. Nil expresses the absence of a value. Do not use a magic string such as `"none"` when the program means absence; different values have different comparison and display behavior. Check the current runtime's truthiness rules before teaching a shortcut.

Write three examples: an explicitly true flag, an explicitly false flag, and a missing optional value. Print each one. Then use an `if` to give the missing value a friendly fallback. This creates a bridge from data values to control flow.

Chapter 3 - Values, variables, and names

Page 4: Names as documentation

Names should answer a question: `total`, `customer_email`, and `is_ready` tell the reader more than `x`, `y`, and `flag`. Use snake_case for ordinary program variables unless a surrounding convention requires something else. Avoid names that shadow built-ins or hide whether a value is singular or plural.

Review a small program and rename three bindings without changing behavior. If a rename forces you to edit many unrelated lines, that is evidence the code needs a function or module boundary.

Chapter 3 - Values, variables, and names

Page 5: Checkpoint

Create a five-line program that stores a product name, quantity, price, and availability flag, then prints a sentence assembled from those values. Add one nil-like optional note and choose a fallback. Explain which parts are literals and which parts are bindings. Run it, then intentionally misspell one name and read the diagnostic.

Chapter 4 - Strings and interpolation

Page 1: Text is data

Strings represent text, not commands. Use quotes consistently and keep user-facing text separate from the data it describes. A label, an email address, and a serialized JSON fragment may all be strings but should not be treated as interchangeable. Name the value according to its role.

```
let name = "Ari"
let message = "Welcome, " + name
print(message)
```

Concatenation is explicit. If the current language build supports interpolation syntax, prefer it when it makes a sentence easier to scan, but validate the exact syntax in the interpreter before putting it in a lesson.

Chapter 4 - Strings and interpolation

Page 2: Escaping and boundaries

Quotes inside a string need escaping or a different delimiter supported by the language. Newlines, tabs, and backslashes are also data with special notation. A common bug is closing a string too early, which causes a later parser error at a misleading location.

Practice with a quoted customer name, a path-like string, and a message containing punctuation. Print each value. Then move the punctuation into a separate binding and decide whether the result is clearer or worse. Readability is the point of the exercise.

Chapter 4 - Strings and interpolation

Page 3: Formatting values

Numbers should be formatted at the boundary where people see them. A stored money value is a number; a display value may have two decimal places and a currency symbol. Dates likewise need a display policy. Keep the raw value available for calculations and the formatted value for a label.

The proof project places a JavaScript formatting helper in a custom module. That is a useful boundary: Nova source declares the feature, while the generated Vue application owns browser-specific formatting.

Chapter 4 - Strings and interpolation

Page 4: User input is not trusted text

A string from a form, route parameter, or environment variable is input. Trim it when appropriate, validate its shape, and avoid interpreting it as source code. Never assemble SQL, shell commands, or HTML by blindly concatenating input. Use the generated validators and parameterized database layer.

Write a function that accepts a username and returns a normalized display label. Test an empty string, whitespace, and punctuation. State which transformations are safe and which would be surprising to a user.

Chapter 4 - Strings and interpolation

Page 5: Checkpoint

Build a receipt line from a customer name, quantity, and price. Add a helper that formats the price to two decimals. Then write one invalid-input case and describe the error message a user should see. This exercise connects strings, numbers, custom helpers, and validation without needing an application scaffold.

Chapter 5 - Lists, objects, and indexing

Page 1: Lists

Lists preserve order and hold multiple values. Use them for rows, items, steps, and other sequences. A list should usually have one conceptual element type even if the runtime permits mixtures. Give the list a plural name and give each element a singular name when iterating.

```
let colors = ["red", "blue", "gold"]
print(colors[0])
```

Indexing is a boundary: an index outside the list must be handled according to the current runtime's error behavior. Do not hide an out-of-range bug with an arbitrary fallback until you understand why it happened.

Chapter 5 - Lists, objects, and indexing

Page 2: Objects

Objects group named fields. They are useful for configuration, records, and small messages. A field name is part of the data contract. Keep it stable when another function or generated route consumes the object.

```
let customer = { name: "Ari", status: "active" }
print(customer.name)
```

Dot access communicates a known field. Index-style access is useful when the field name is itself stored in a variable, if the current language surface supports it.

Chapter 5 - Lists, objects, and indexing

Page 3: Nested data

Real data is nested: a response may contain a list of objects, and an order may contain a customer reference. Start with a readable shape, then write one accessor at a time. When a nested value can be absent, check the parent before reaching deeper.

Draw the shape on paper. Mark each list, object, scalar, and optional field. Then write a function that extracts one display value. This simple diagram often reveals whether the data contract is too complicated for one function.

Chapter 5 - Lists, objects, and indexing

Page 4: Transformations

Collection transformations should state what they do: filter inactive rows, map names to labels, or total amounts. If the language's standard library provides collection helpers, use the documented names. Otherwise a loop is clearer than inventing an API.

Make a list of three order objects. Produce a second list containing only pending orders. Then calculate the total for those rows. Print both the count and total, and include a case where the filtered list is empty.

Chapter 5 - Lists, objects, and indexing

Page 5: Checkpoint

Model a small inventory object with a list of product records. Write one function that finds a product by name and another that returns products with quantity below a threshold. Add a missing-product case. Explain why the functions return data rather than printing from deep inside the logic.

Chapter 6 - Numbers, booleans, comparisons, and arithmetic

Page 1: Arithmetic

Arithmetic is easiest to trust when units are explicit. A price and a quantity multiply into a subtotal; a percentage is not the same as a whole number. Name intermediate values so the formula can be inspected.

```
let unit_price = 12.5
let quantity = 3
let subtotal = unit_price * quantity
print(subtotal)
```

Decide how the program handles decimals, rounding, and invalid quantities. A display helper should not silently change the stored calculation.

Chapter 6 - Numbers, booleans, comparisons, and arithmetic

Page 2: Comparisons

Comparisons produce booleans. Equality asks whether two values match; ordering asks whether one value is before, after, less than, or greater than another. Compare compatible values and make the comparison's business meaning visible in the variable name.

`is_overdue`, `has_stock`, and `meets_minimum` read like decisions. A naked expression in a long condition forces the reader to recompute the meaning each time.

Chapter 6 - Numbers, booleans, comparisons, and arithmetic

Page 3: Boolean composition

Logical operators combine decisions. Use parentheses when the precedence is not obvious. A condition such as “paid and shipped, or staff override” should be grouped so the exception is unmistakable.

Write truth-table examples for a checkout rule. Test paid/unpaid, shipped/unshipped, and an override role. Then simplify only after the tests make the intended behavior clear.

Chapter 6 - Numbers, booleans, comparisons, and arithmetic

Page 4: Guarding invalid math

Division by zero, negative quantities, and nonnumeric form values are not edge trivia; they are normal input cases. Validate before arithmetic and return a useful error or a safe result. Keep validation close to the boundary where invalid data enters.

Create a `percentage_of` function. Define its behavior for a zero denominator, a negative amount, and a percentage above one hundred. Write the rule in prose before writing the condition.

Chapter 6 - Numbers, booleans, comparisons, and arithmetic

Page 5: Checkpoint

Implement a shipping decision from subtotal, country, and expedited flag. Use named boolean expressions, test at least four combinations, and print the final price. Explain which values are domain data and which are derived decisions. This is the same discipline used later in route guards and workflow validation.

Chapter 7 - Conditionals and loops

Page 1: If blocks

An `if` block makes a decision. Its condition should be a boolean expression or a clearly named boolean. Keep the branch small; move multi-step work into a function so the decision remains readable.

```
if balance > 0 {
    print("Payment available")
} else {
    print("Payment required")
}
```

Use the exact block and comparison syntax accepted by the current parser. A teaching example is only useful when a learner can validate it.

Chapter 7 - Conditionals and loops

Page 2: Multiple branches

Several mutually exclusive states can be expressed with `if`/`else if`/`else` when the language surface supports them. Prefer a small number of named categories over a deeply nested tree. If a state has more than a few branches, consider a lookup object or a function per state.

Describe the precedence of checks in prose. For an order, “cancelled” may override “late,” while “paid” and “shipped” may be independent facts. Code should reflect that domain order.

Chapter 7 - Conditionals and loops

Page 3: While loops

A `while` loop repeats while a condition is true. Every loop needs a progress argument: what changes so the condition eventually becomes false? Without one, the program can hang.

Write a countdown and prove that the value changes on every iteration. Add a maximum-iteration guard when processing external input. A guard is not a substitute for correctness, but it limits damage when assumptions fail.

Chapter 7 - Conditionals and loops

Page 4: For-style iteration

Use the current language's supported loop form to visit a list. The body should perform one conceptual operation. If it grows, extract a function. Avoid mutating the list being traversed unless the runtime and the algorithm make that safe.

Practice by printing each pending order, then count them. Add an empty-list case. Compare the loop with a collection helper if the standard library offers one, and choose the clearer form.

Chapter 7 - Conditionals and loops

Page 5: Checkpoint

Create a retry loop for a fictional operation. It should stop on success, stop after three failures, and report the final state. Do not call a real service. The exercise is about state transitions, exit conditions, and diagnostics-the same pieces a workflow runner needs.

Chapter 8 - Functions and return values

Page 1: A function has a job

A function names one transformation or decision. Parameters are its inputs; the return value is its output. A function that both changes a database and formats a UI label has two jobs and is difficult to test.

```
function subtotal(price, quantity) {
    return price * quantity
}
```

Read the signature as a contract. Ask what values are valid and what happens when a value is missing or invalid.

Chapter 8 - Functions and return values

Page 2: Return early

Guard clauses make invalid cases visible near the top of a function. Return an error value or raise the language's supported error when the contract cannot be met. Do not continue with a half-valid record and hope a later layer catches it.

Write a `require_positive_quantity` helper and use it before multiplying. Test zero, a negative value, and a valid value. Keep the error message actionable: identify the field and the allowed rule.

Chapter 8 - Functions and return values

Page 3: Pure and effectful functions

A pure function depends only on its inputs and has no external side effect. A workflow step, file write, or database insert is effectful. Keep pure calculation separate from effectful orchestration. That split lets you test pricing rules without starting a server.

For the proof project, `formatOrderTotal` is a small frontend helper. Creating an order is an effectful backend operation. The form gathers input, the workflow validates, and the database layer writes.

Chapter 8 - Functions and return values

Page 4: Composition

Useful programs are compositions of small functions. One function parses, another validates, another transforms, and a final function performs the effect. Name the intermediate value and keep the order explicit.

Draw a pipeline for a form submission. Mark which stages can fail. Decide where each failure becomes a user-facing message and where it becomes a server log. This diagram becomes a route and workflow design later.

Chapter 8 - Functions and return values

Page 5: Checkpoint

Write three functions for a tiny invoice: validate a line, calculate a line total, and calculate the invoice total. Add one invalid line test. Then refactor the main program so it calls the functions rather than duplicating arithmetic. State which functions are pure and why.

Chapter 9 - Errors, diagnostics, and debugging

Page 1: Read the first error

A parser diagnostic gives a location and a reason. Start there. A missing brace can make every later line look wrong, so fixing the first structural problem often removes many secondary messages. Keep the source open beside the diagnostic and count from the reported line.

Do not “fix” an error by deleting the feature you meant to learn. Reduce the example to the smallest failing form, validate it, and add pieces back one at a time.

Chapter 9 - Errors, diagnostics, and debugging

Page 2: Syntax versus semantics

Syntax asks whether the program has a valid shape. Semantics asks whether the names, targets, fields, and references make sense. A `table` with a missing field reference is syntactically valid but semantically unusable. Treat both checks as necessary.

For application source, validate after changing a table, page, workflow, route, or target. A clean build does not mean a route is authorized or a database write is correct; it means the compiler could produce the requested artifact.

Chapter 9 - Errors, diagnostics, and debugging

Page 3: Runtime failures

Runtime failures occur after parsing: a missing package, an invalid environment variable, a database migration failure, or an unhandled request. Capture the command, exit code, and relevant log line. Never paste a secret-bearing environment dump into an issue.

Separate code failures from host failures. A registry timeout or a missing native module may block a local proof run without proving the generated source is wrong. Record the distinction so the next run starts with the right hypothesis.

Chapter 9 - Errors, diagnostics, and debugging

Page 4: Deterministic fallbacks

NovaDev's optional AI assistant is not a required compiler dependency. If no key is configured, if the request fails, or if the model returns malformed content, deterministic generation continues. This is a reliability pattern: optional quality improvements must not become required correctness paths.

Write a test that stubs the assistant response and another that disables it with `NOVADEV_AI_ASSIST=0`. The test should prove that source generation still returns a result when the assistant is unavailable.

Chapter 9 - Errors, diagnostics, and debugging

Page 5: Debugging checkpoint

Create three failures: a missing brace, an unsupported backend target, and a generated-project package-install failure. For each, write the first useful diagnostic, the smallest next action, and what evidence would close the issue. This is a professional debugging record, not just an exercise answer.

Chapter 10 - Classes, composition, and OOP

Page 1: Why objects exist

Classes group data and behavior when a concept has a stable identity and several operations. A `Customer` may have contact data and methods for display or eligibility. Do not create a class merely because a language supports classes; a small function and object may be clearer.

Start with the behavior you need. Then decide whether a class owns it. This reverses the common beginner mistake of designing a hierarchy before identifying a responsibility.

Chapter 10 - Classes, composition, and OOP

Page 2: Encapsulation

Encapsulation means callers use a small public surface while internal representation remains changeable. A class can validate its state in a constructor or method. A service can hide how it calls a database. The goal is a stable contract, not secrecy for its own sake.

Write a `Money` concept with an amount and currency. Decide whether invalid currency belongs in construction, validation, or a separate parser. Document the decision and test it.

Chapter 10 - Classes, composition, and OOP

Page 3: Composition

Composition builds a larger object from smaller collaborators. An order service may receive a validator, repository, and notifier. Each collaborator has one job. Composition keeps behavior replaceable and avoids a giant base class with unrelated hooks.

Map the proof project to collaborators: a model layer persists records, an auth module issues tokens, routes translate HTTP, and the Vue page renders state. The compiler generates these boundaries even when the source declaration is compact.

Chapter 10 - Classes, composition, and OOP

Page 4: Inheritance

Inheritance can express a true “is a” relationship, but it couples subclasses to base-class details. Prefer composition when behavior varies independently. If a lesson uses `extends`, explain which contract is inherited and why a collaborator would not be clearer.

Create a small notification hierarchy only as a comparison exercise. Then rewrite it with a `send` function and a strategy object. Compare the number of files, tests, and surprising dependencies.

Chapter 10 - Classes, composition, and OOP

Page 5: Checkpoint

Design a class or object for a cart. It must add an item, remove an item, and calculate a total. Write invariants: quantity cannot be negative and an unknown item cannot be removed silently. Explain whether the design is better as a class, a module of functions, or a combination.

Chapter 11 - Modules, use, and package boundaries

Page 1: A module is a boundary

Modules let a file own a concept and expose a deliberate interface. Keep private helpers private when the language/module system permits it. A module should have a reason for changing. If every feature edits the same file, the boundary is not doing enough work.

Use a small module for invoice math and import it into a command file. Name the import by the concept it provides, not by a vague abbreviation.

Chapter 11 - Modules, use, and package boundaries

Page 2: Imports are dependencies

An import is a dependency edge. It tells the reader what a module needs and makes cycles visible. Keep the dependency direction simple: user interface to application API, route to service, service to model. A lower layer should not reach upward into a page component.

Draw the module graph before adding an import that feels convenient. A graph with many arrows in both directions is a design warning.

Chapter 11 - Modules, use, and package boundaries

Page 3: Application modules

The generated project separates frontend custom JavaScript from backend server modules. A custom browser helper should not read server secrets. A backend module may validate and persist data, but it should not assume a browser DOM exists.

Declare custom code intentionally and review the generated destination. The compiler's responsibility is to preserve the boundary; the developer's responsibility is to keep the code safe inside it.

Chapter 11 - Modules, use, and package boundaries

Page 4: Packages

A package is an external dependency with its own version, license, and failure modes. Add one because the project needs a capability, not because a tutorial mentioned it. Pin core toolchain versions in generated manifests so repeated builds do not silently change behavior.

Audit packages before deployment. A small dependency tree is easier to install, secure, and explain. Do not put backend-only packages in the frontend manifest or vice versa.

Chapter 11 - Modules, use, and package boundaries

Page 5: Checkpoint

Split a simple library into `math`, `formatting`, and `main` modules. Give each module one public function. Add a fake package dependency in a design note, then remove it and implement the small behavior locally. Explain the tradeoff between reuse and operational cost.

Chapter 12 - The standard-library toolbox

Page 1: Choose a library by the job

The standard library should provide boring, dependable building blocks: math, dates, JSON, file access, SQLite helpers, and HTTP clients. Start with the job-parse a date, persist a local record, read JSON-then select the namespace documented by the current repository. Do not invent Python or JavaScript names by memory.

A library call is still code with inputs, outputs, and errors. Wrap it in a function when the rest of the program should not know the library's details.

Chapter 12 - The standard-library toolbox

Page 2: Files

File operations have a scope and a risk. Read and write inside an application-owned directory. Use explicit paths, validate user-provided names, and avoid recursive deletion unless the destination has been resolved and reviewed. A “delete files” lesson should teach a dry run or a confirmation boundary.

Write a small report file with a known name. Read it back, handle a missing-file case, and leave unrelated files untouched. Record the absolute path only in local output, not in a portable lesson.

Chapter 12 - The standard-library toolbox

Page 3: JSON

JSON is a boundary format. Parse it, validate the shape, and serialize only data the consumer needs. Do not assume a successful parse means a valid domain object. Check required fields and types.

Create a JSON settings object, save it, reload it, and reject a version with a missing required key. Keep secrets out of fixture JSON; use placeholders and environment configuration.

Chapter 12 - The standard-library toolbox

Page 4: Dates and HTTP

Dates need a timezone and a format policy. HTTP needs a timeout, status handling, and a response-size policy. A request that works once on a laptop is not a reliable integration. Put the client behind a function so tests can stub it.

Use a fictional or local endpoint for practice. Never put a real key in a lesson. Test success, a non-success status, a timeout, and malformed JSON.

Chapter 12 - The standard-library toolbox

Page 5: Checkpoint

Build a local report pipeline: read JSON rows, filter by date, compute a total, and write a Markdown summary. Add a missing input and an invalid row case. List which operations are pure and which touch the filesystem. This is the same pipeline used by application jobs.

Chapter 13 - App and project declarations

Page 1: The application shell

An `app` declaration gives a project a name and a boundary for its members. Inside it, project settings, tables, pages, workflows, routes, modules, and seeds describe one application. Keep a proof project small enough that every declaration can be traced to generated output.

```
app ProofExpress {
    project {
        frontend VueVite
        backend Express
    }
}
```

Statement-per-line form is easier to diagnose than a single dense line.

Chapter 13 - App and project declarations

Page 2: Project settings

The project block selects the frontend, backend, database, structure, styling, and mode. Current application lessons use `VueVite`, `Express`, `SQLite`, `VueExpress`, and `Tailwind`. Unsupported targets should fail clearly rather than silently selecting a legacy generator.

A project setting is a promise to the generator. If a setting has no current generator behavior, it belongs in a design note, not a lesson example.

Chapter 13 - App and project declarations

Page 3: Features

Features describe cross-cutting needs such as validation, CORS, security headers, authentication, uploads, Prisma, and testing. A feature should map to a real generated dependency or file. Review the generated package manifests to make sure frontend and backend dependencies remain separated.

Use the smallest feature set that proves the lesson. Extra features create extra install time and extra failure surface.

Chapter 13 - App and project declarations

Page 4: Seeds and fixtures

Seeds make a generated project observable. They should be fictional, deterministic, and safe to repeat. Never seed a real password, token, or API key. A development password in a local fixture must be clearly non-production and excluded from deployment instructions.

The proof project seeds an admin account, customer, and order so authentication, relation shape, and list endpoints can be exercised.

Chapter 13 - App and project declarations

Page 5: Checkpoint

Write an app declaration with canonical project settings, one fictional seed, and one custom module. Validate it. Make a table or page reference a nonexistent name, observe the semantic diagnostic, and repair it. Explain how the source remains compact while the output becomes an ordinary project.

Chapter 14 - Modes and ProjectIR

Page 1: Modes are input policy

Modes can provide defaults for an application domain, but they do not excuse unsupported output. Custom mode is useful when the source should state its own project decisions. In all modes, the compiler still produces one ProjectIR representation before generation.

Ask what a mode changes: defaults, components, styling, or validation. If the answer is unclear in source, read the mode registry rather than inferring from a website card.

Chapter 14 - Modes and ProjectIR

Page 2: ProjectIR as a contract

ProjectIR is the compiler's normalized model. It contains canonical targets, tables, fields, pages, workflows, routes, auth, features, modules, seeds, and relationships. Generators consume ProjectIR; they should not reparse `.nova` text with regular expressions.

This separation lets validation happen once and lets documentation describe the same model that code generation uses.

Chapter 14 - Modes and ProjectIR

Page 3: Relationships

A field such as `customer Customer required` expresses a relationship from `Order` to `Customer`. The generator turns that intent into schema information and relation metadata. The generated API still needs a concrete payload policy; decide whether a form sends an ID, a natural key, or a nested object.

Write the relationship in plain language before coding: “each order belongs to one customer.” Then document the reverse query only if the generator supports it.

Chapter 14 - Modes and ProjectIR

Page 4: Diagnostics as design feedback

Semantic diagnostics are not merely compiler scolding. “Missing field” means the page and table contracts disagree. “Unsupported backend” means the project is asking for a generator that does not exist. Fix the model, not just the message.

A good diagnostic includes a source location. Preserve that location when changing analyzer code so IDE output can point to the right line.

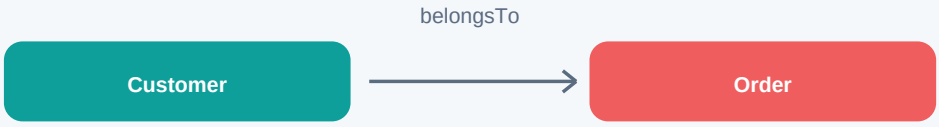
Chapter 14 - Modes and ProjectIR

Page 5: Checkpoint

Draw an input-to-output diagram for the proof project. Label AST, ProjectIR, frontend generation, backend generation, and docs generation. Pick one field and follow it through the table declaration, IR, Prisma schema, API model, and Vue form. Note every conversion and validation boundary.

Chapter 15 - Tables, relations, SQLite, and Prisma

Page 1: Table design



Tables describe durable records. Give each table an identity field, name required fields, and distinguish text, email, money, numbers, dates, booleans, and secure values. The type is a contract for validation and generated schema, not a decorative label.

```
table Customer {
    id auto
    name text required
    email email unique required
}
```

One field per line makes attribute parsing unambiguous.

Chapter 15 - Tables, relations, SQLite, and Prisma

Page 2: Required and unique

Required prevents incomplete records; unique prevents duplicates for a field that is an identity in the domain. An email may be unique for an account but not necessarily for a mailing-list event. Make the business rule explicit before adding the attribute.

Test missing values and duplicates at the API boundary. A frontend required marker improves experience, but the backend must enforce the rule again.

Chapter 15 - Tables, relations, SQLite, and Prisma

Page 3: Relations

The proof project has `Order.customer Customer required`. The relation is represented in ProjectIR and the generated schema. Decide how the API receives it. A simple initial contract can accept a customer ID or a stable name in a seed-only workflow; production code should prefer a validated ID.

Seed the related row before the dependent row. A database constraint should fail loudly when the relation is invalid rather than creating an orphan record.

Chapter 15 - Tables, relations, SQLite, and Prisma

Page 4: SQLite and Prisma

SQLite is a local file database useful for development and small deployments with honest persistence expectations. Prisma supplies a schema, client, and database operations. Generated code should keep the database URL in environment configuration and run its schema initialization as an explicit build/start step.

Do not describe SQLite as a multi-instance production database without a deployment-specific persistence plan. A mounted disk, backup policy, and single-writer assumption are operational decisions.

Chapter 15 - Tables, relations, SQLite, and Prisma

Page 5: Checkpoint

Design `Account`, `Customer`, and `Order` tables with one relationship, one unique email, one secure password field, and one money field. Write three valid seeds and two invalid test cases. Explain what belongs in Nova source, Prisma schema, backend validation, and frontend form feedback.

Chapter 16 - Pages, forms, and Vue/Vite UI

Page 1: A page is a user journey

A page declaration names a screen and its components. Start with the user question: what should this page let someone see or do? A table renders records; a form gathers input; a hero or card explains context. Keep the page focused.

The generated frontend is a normal Vue/Vite application. It can be opened and edited by a Vue developer. The `.nova` file expresses intent; the generated component is where interaction details live.

Chapter 16 - Pages, forms, and Vue/Vite UI

Page 2: Forms

A form needs fields, labels, validation, submit behavior, and a result state. The source declaration can identify the entity and workflow. The generated Vue code should still show loading, success, and failure states. A disabled submit button is not enough evidence that the request is safe.

Practice with the order form: customer, total, and status. Decide what a user sees when total is invalid and what the backend returns when a required field is missing.

Chapter 16 - Pages, forms, and Vue/Vite UI

Page 3: Tables

A table component needs a source and columns. Use human-readable labels and stable field names. Empty state, loading state, and error state are part of the component contract. A table that renders a blank page on a failed API request is not complete.

Add a row action only when its authorization and confirmation behavior are defined. A delete button is a security and data-integrity decision, not just a UI element.

Chapter 16 - Pages, forms, and Vue/Vite UI

Page 4: Vite configuration

Vite builds the frontend and supplies the development experience. API base URLs belong in environment variables when frontend and backend are separate. Do not embed a machine-specific localhost URL in generated source. A proxy may be configured for local development through explicit environment settings.

Build the frontend and inspect the output directory. Verify the generated JavaScript has no syntax errors and that the page has a mount element before calling the frontend usable.

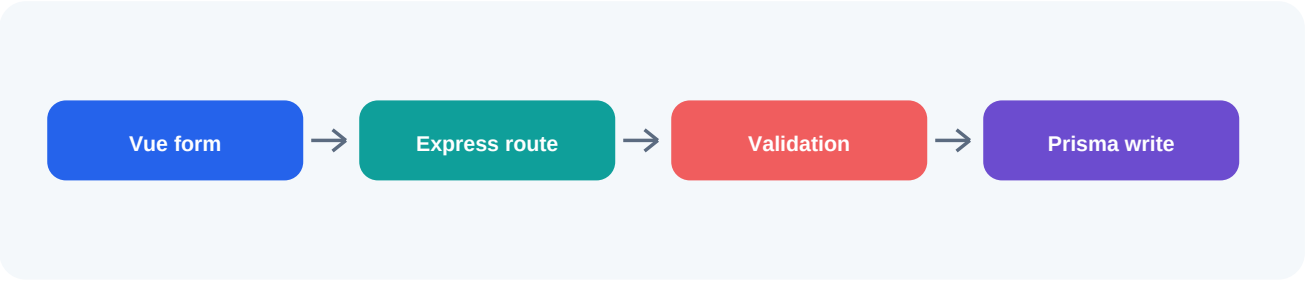
Chapter 16 - Pages, forms, and Vue/Vite UI

Page 5: Checkpoint

Sketch a page with a heading, order form, and order table. List every visible state. Map each state to a Vue ref or computed value, an API call, and a user message. Then compare the sketch with the generated page and record one improvement you would make by editing Vue source.

Chapter 17 - Workflows as application behavior

Page 1: A workflow is a named action



A workflow gives a multi-step business action a stable name. It may accept an input entity, create or update records, validate fields, use a custom function, or notify another boundary. Naming the action helps the frontend and backend agree on what a submit button means.

The workflow should be idempotent or have a duplicate policy when users can click twice. “Create order” needs to answer whether two identical submissions create two orders.

Chapter 17 - Workflows as application behavior

Page 2: Validation order

Validate cheap shape rules before database work. Validate required fields, email syntax, positive money, and allowed status values. Then check relations and uniqueness. Finally perform the write. This order produces clearer errors and avoids partial changes.

The backend must repeat security-sensitive validation even when the Vue form already checks it. Client validation is for feedback; server validation is for trust.

Chapter 17 - Workflows as application behavior

Page 3: API contract

The generated Express route is `/api/workflows/:workflow`. The workflow slug is normalized from its declaration name. The response should identify success, the workflow, created records, and a useful record reference. A failure should have a status code and error message, not an empty 200 response.

Before teaching a workflow, inspect generated route ordering. A generic `POST /:resource` route must not capture `/workflows/:workflow` first.

Chapter 17 - Workflows as application behavior

Page 4: Side effects

Notifications, emails, and external calls are side effects. Make them explicit and observable. If a database write succeeds but a notification fails, choose whether to retry, compensate, or report partial success. Do not hide an external call inside a display helper.

Use a fake notifier in exercises. Record the event payload and test that the workflow calls it once. This prepares the code for a queue without requiring a live service.

Chapter 17 - Workflows as application behavior

Page 5: Checkpoint

Design `CreateOrder`: validate customer and total, write an order, and return the created row. Add a duplicate-submission rule and one invalid relation case. Write the request and response JSON by hand. Then trace the same action from the Vue form to the Express route to Prisma.

Chapter 18 - Routes, authentication, and roles

Page 1: Routes

A route has a method, path, input policy, and response contract. Use nouns for resources and action paths for workflows. Keep `/api/health` small and dependency-light so deployment checks can call it. A protected route should fail without credentials before it attempts private work.

The proof route `GET /api/admin/orders` requires authentication. It is a useful acceptance case even though the generated starter response remains intentionally declared-route metadata.

Chapter 18 - Routes, authentication, and roles

Page 2: Authentication

Authentication answers who the caller is. The generated Express auth module uses a password hash and signed token flow. Passwords are never returned in public records. Secrets come from environment variables, not source files.

Use a fictional seeded development account only for local checks. Rotate credentials before any shared environment. A token in a browser is still sensitive data; protect it according to the deployment's threat model.

Chapter 18 - Routes, authentication, and roles

Page 3: Authorization

Authorization answers what the caller may do. A role such as `admin` is a policy input, not proof that the user should see every record. Apply the policy in backend routes and database queries. A hidden frontend button is not authorization.

Test three states: no token, valid token with insufficient role, and valid token with the required role. Make the expected status codes part of the API contract.

Chapter 18 - Routes, authentication, and roles

Page 4: CORS and headers

CORS controls which browser origins may call an API. Configure it through an environment variable and use secure headers middleware. Avoid allowing every origin with credentials in a production configuration. Rate limits and request validation reduce abuse but do not replace authorization.

Security configuration should be visible in generated files and explained in the README. A developer should know which values must be set before running the app.

Chapter 18 - Routes, authentication, and roles

Page 5: Checkpoint

Write an auth matrix for the proof app: public health, public schema, public order list only if the page policy allows it, protected admin route, login, and write operations. For each row list method, path, credential, role, status, and data exposure. Review it for accidental password or token leakage.

Chapter 19 - Custom JavaScript, Python boundaries, and security

Page 1: Custom code is an escape hatch

Custom code exists for behavior the declarative vocabulary cannot express. It should have a named module, a language, a destination, and a small interface. The source declaration should not become a second undocumented compiler.

In the proof project, a JavaScript helper formats an order total. It is browser-adjacent and belongs in the generated app's custom module area. It does not read the OpenRouter key or reach into Prisma.

Chapter 19 - Custom JavaScript, Python boundaries, and security

Page 2: Browser versus server

Browser code can render, collect input, and call public API endpoints. Server code can validate trusted boundaries, access the database, and use server secrets. Mixing those responsibilities creates leaks. Put a secret read in the backend only and return the minimum data needed by the browser.

If a requested custom Python module has no current Express integration path, document that boundary rather than pretending the generator imports it into Node. A future bridge can be designed separately.

Chapter 19 - Custom JavaScript, Python boundaries, and security

Page 3: Validation and escaping

Validate values at boundaries and escape output for its context. HTML, SQL, shell commands, URLs, and JSON each have different rules. Parameterized database calls and framework escaping are safer than handcrafted concatenation.

Use allow-lists for enum-like statuses and file extensions. Enforce maximum lengths and upload sizes. A custom module should make those rules easy to find, not conceal them in a clever one-liner.

Chapter 19 - Custom JavaScript, Python boundaries, and security

Page 4: Secrets

The compiler's local `.env` contains optional configuration. It is ignored by Git and loaded only by the compiler process. Generated applications receive their own `.env.example` with placeholders such as `NOVA_SECRET_KEY`; they do not receive the OpenRouter key.

If a key has ever been pasted into chat, source, a log, or a screenshot, treat it as exposed and rotate it. A key being present in a local file is not a reason to print or test it.

Chapter 19 - Custom JavaScript, Python boundaries, and security

Page 5: Checkpoint

Review a custom helper and label every input as trusted or untrusted. Move one server-only operation out of browser code. Add a validation rule and a safe error message. Finally, write a short security note that names the secret source, the generated destination, and the redaction policy.

Chapter 20 - Jobs, manual runs, and the scheduler boundary

Page 1: What a current job is

The parser accepts `job` declarations and the current Express generator exposes the declared catalog at `GET /api/jobs`. A job is metadata plus a name that can be selected by a manual request. This is useful for proving that a job was declared and that the API contract exists.

It is not yet a durable queue, distributed worker, or calendar scheduler. Documentation must keep that distinction visible.

Chapter 20 - Jobs, manual runs, and the scheduler boundary

Page 2: Manual execution

`POST /api/jobs/:name/run` is the current manual entry point. A successful response records an in-process queued/completed transition. An unknown name returns an error. This endpoint is a starting contract for a future worker, not evidence that business work has happened outside the process.

Test the catalog, known job, and unknown job. Include a process restart in the acceptance plan to show that in-process status is not durable.

Chapter 20 - Jobs, manual runs, and the scheduler boundary

Page 3: Designing a real job

For a production report, define input parameters, idempotency key, schedule source, retry policy, timeout, observability, and storage of results. A reminder job needs a recipient policy and an opt-out path. A billing check needs a dry-run mode and a reconciliation record.

Write this design before adding a scheduler. It prevents a `setInterval` loop from becoming an unreviewed production system.

Chapter 20 - Jobs, manual runs, and the scheduler boundary

Page 4: Historical Python scheduler note

Older repository artifacts contain a retired Python `backend/jobs.py` sketch and mention `NOVA_RUN_JOBS`. They are not generated by the current Vue/Vite + Express path. They remain useful as historical design context only. Do not copy them into a current project or teach them as shipped behavior.

If a future implementation adds a Node worker, it should have a new verified contract, tests, and deployment guidance before it enters the main curriculum.

Chapter 20 - Jobs, manual runs, and the scheduler boundary

Page 5: Checkpoint

Create a job design for an inventory alert. Define the catalog entry, manual request, input validation, idempotency, retry policy, and result record. Mark which parts the current generator provides and which remain future work. This is the honest professional answer to an automation requirement.

Chapter 21 - Worked project: ProofExpress, part one

Page 1: Project brief

ProofExpress is a small order dashboard used to exercise the compiler contract. It has accounts, customers, and orders. A customer belongs to an order through the `customer` field. The page displays orders and submits a `CreateOrder` workflow. An admin route demonstrates authentication.

Keep the brief small. The purpose is evidence: two tables would prove a relation, but three tables also let the auth record remain separate from business data.

Chapter 21 - Worked project: ProofExpress, part one

Page 2: Source shape

Begin with canonical settings and one feature block. Write each table field on its own line, then declare the workflow, page, route, custom module, and seeds. Validate after each conceptual addition. This makes it easy to identify whether a failure came from syntax, semantic references, or generated code.

The page uses `form orders` and `table orders` so its component source matches the ProjectIR resource name. Case and resource naming are part of the contract.

Chapter 21 - Worked project: ProofExpress, part one

Page 3: Data model

`Account` has a unique email, secure password, and role. `Customer` has a required name and unique email. `Order` has a required customer relation, money total, and status. The seed order comes after its customer. The password is only a fixture input; public responses must omit it.

Draw the schema and list the invalid cases: missing email, duplicate email, missing customer, invalid total, and missing status.

Chapter 21 - Worked project: ProofExpress, part one

Page 4: Page contract

The Orders page has a form and table. The form fields are customer, total, and status; its submit action names `CreateOrder`. The table columns are the same visible business fields. A generated Vue page should provide loading and error feedback even if the source declaration stays compact.

The custom formatter returns a display string, not a database value. That distinction prevents presentation logic from changing persistence.

Chapter 21 - Worked project: ProofExpress, part one

Page 5: Part-one checkpoint

Run validation and record the result. Confirm the IR contains three tables and a `belongsTo` relationship. Confirm generated output has frontend and backend directories, a Prisma schema, an Express server, auth code, and the custom JavaScript module. Do not claim the server works until packages and runtime checks complete.

Chapter 22 - Worked project: ProofExpress, part two

Page 1: Generated backend tour

The backend configuration reads `PORT`, `DATABASE_URL`, `NOVA_SECRET_KEY`, and CORS settings from the environment. The server exposes a health route, schema route, auth routes, CRUD resource routes, workflow routes, and job routes. The Prisma schema describes the SQLite models.

Read generated files in dependency order: config, schema, models, auth, routes, server. This mirrors the runtime path and makes a failure easier to localize.

Chapter 22 - Worked project: ProofExpress, part two

Page 2: Acceptance sequence

The sequence is deliberate: health, schema, list, unauthenticated protected route, login, authenticated protected route, CRUD create, workflow create, and list again. A workflow response should show the created row. The final list should show database-backed state rather than a hardcoded sample.

Use a test database path and a test secret. Never run the fixture password against a shared environment.

Chapter 22 - Worked project: ProofExpress, part two

Page 3: Frontend tour

The frontend build produces Vite assets. The app module mounts Vue, configures routing and stores when used, and loads page components. A separate API base URL is environment-driven. The generated page is editable; adding a field label or empty state should be ordinary Vue work.

A static asset build is not a browser acceptance test. Open the built page or run a preview server and watch the network request for the page's API call.

Chapter 22 - Worked project: ProofExpress, part two

Page 4: Operational contract

`render.yaml` is generated when deployment metadata is requested. It uses a distinct build and start command, a health path, environment variables, and an honest SQLite disk strategy. This book does not claim a live Render deployment; deployment credentials and external state are intentionally out of scope for this curriculum pass.

Local proof and deployment proof are separate rows in the evidence ledger.

Chapter 22 - Worked project: ProofExpress, part two

Page 5: Part-two checkpoint

Write a one-page runbook for ProofExpress. Include install commands, environment names, database initialization, health check, login payload, protected route, workflow payload, and cleanup. Mark every step that depends on npm/network availability. This runbook is more valuable than a screenshot because another developer can repeat it.

Chapter 23 - Testing and acceptance evidence

Page 1: Test layers

Unit tests cover pure functions. Integration tests cover a module boundary such as a model or route. Acceptance checks cover the user-visible sequence. Keep the layers distinct so a passing unit test does not hide a broken server startup.

The generated backend includes a Jest/Supertest contract when testing is enabled. The exact test result belongs in the evidence ledger, not in a README copied from a different project.

Chapter 23 - Testing and acceptance evidence

Page 2: API tests

An API test should assert status, response shape, and a meaningful side effect. Test health, login failure, login success, protected route rejection, protected route success, validation failure, resource create, and workflow create. Use an isolated database and deterministic seed data.

Do not assert an implementation detail when the public contract is enough. A route can change internal modules without invalidating a good API test.

Chapter 23 - Testing and acceptance evidence

Page 3: Frontend tests

The first frontend check is build success. The next is a browser or preview check that mounts the app and observes a real API request. Test loading, empty, error, and success states. A page that only looks correct with a mocked response is not proven connected.

Keep browser tests free of real credentials. Use a test account or a mocked auth boundary with the scope clearly named.

Chapter 23 - Testing and acceptance evidence

Page 4: Failure-oriented checks

Tests should include failures: invalid email, missing required field, duplicate unique value, no token, wrong role, unknown resource, unknown workflow, and unknown job. Failures are part of the product contract. They also reveal whether generic routes capture specialized paths.

Record status codes and error bodies. A vague 500 makes both users and operators guess.

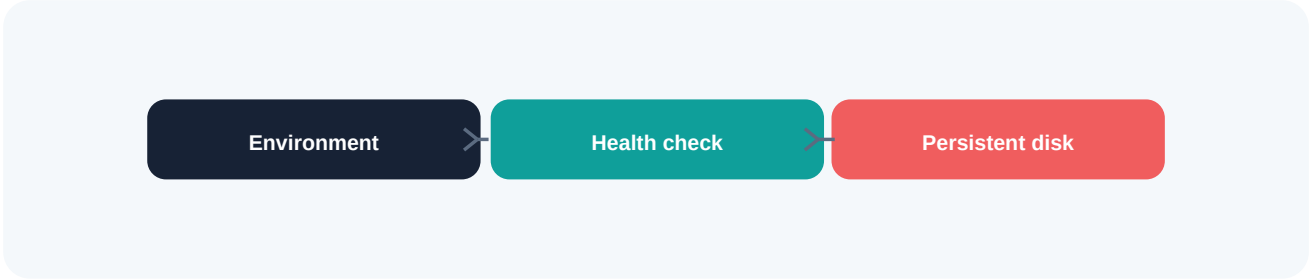
Chapter 23 - Testing and acceptance evidence

Page 5: Checkpoint

Create an acceptance table with columns: scenario, command/request, expected result, observed result, and status. Fill it for the worked project. Leave blocked rows visibly blocked. This is how a team distinguishes “generated” from “works.”

Chapter 24 - Deployment contracts and honest operations

Page 1: Build versus start



A production build prepares assets and database client artifacts. A start command launches the server and reads environment configuration. They are different phases. A development command that watches files is not a production start command.

The backend listens on `PORT`; it must not assume a fixed laptop port. The frontend API base is configurable when it is deployed separately.

Chapter 24 - Deployment contracts and honest operations

Page 2: Health

A health endpoint should be cheap and predictable. Decide whether it means process alive or database ready; document the distinction. A deployment platform can use `/api/health` for process-level checks, while a deeper readiness check can validate database connectivity separately.

Never make the health route depend on a third-party API or a slow report job.

Chapter 24 - Deployment contracts and honest operations

Page 3: Persistence

SQLite persistence is honest only when the deployment provides a durable disk and the application has an appropriate single-instance/storage model. Otherwise data may disappear on restart or cannot support concurrent writers. Say this in the README and choose a different database when the product needs more.

Backups, migrations, and restore tests are operational features. A generated YAML file cannot replace them.

Chapter 24 - Deployment contracts and honest operations

Page 4: Secrets and origins

Set secrets through the deployment environment. Use a generated secret for signing tokens, a controlled CORS origin list, and a database URL appropriate to the platform. Do not commit `.env`, print environment maps, or copy the compiler's OpenRouter key into a deployed app.

Rotate keys after accidental exposure. This includes keys pasted into chat or screenshots.

Chapter 24 - Deployment contracts and honest operations

Page 5: Checkpoint

Review a deployment plan and mark each item verified, configured, or out of scope. Include build command, start command, port, health path, database persistence, CORS, auth secret, logs, and rollback. The correct professional answer can be “not deployed yet” when external credentials were not provided.