Promise Protocol v1 — The Guided Tour

Week 1 of 12: The Workshop & Primitives

Your Software Pedagogue

September 23, 2025

Contents

1 This Week's Mission

The goal this week is to lay a rock-solid foundation for the entire project. We will not build any user-facing features. Instead, we will build the "factory" that will produce those features. By the end of this week, you will have a fully automated development environment and a set of core, trustworthy tools that will prevent entire classes of bugs in the weeks to come. This is the most important investment you can make in the project's long-term health and your own development speed.

2 Learning Plan

This week's reading is the largest block, but it provides the essential context for everything that follows. It covers how services communicate and how we operate them.

2.1 TODO Reading Assignment

- ☐ Read **Part I: Communication (Ch 2-5)** in *Understanding Distributed Systems*.
  - **Focus On**: REST principles, HTTP methods/status codes, and especially **Section 5.7: Idempotency**.
- ☐ Read **Part V: Maintainability (Ch 29-33)** in *Understanding Distributed Systems*.
  - **Focus On**: The "Test Pyramid" (**Ch 29**), CI/CD pipeline stages (**Ch 30**), and the difference between metrics, logs, and traces (**Ch 31-32**).

3 Building Plan: A Step-by-Step Guide

Work through these top-level ‘TODO‘ items in order. Use the checklists to track your progress on the sub-tasks.

3.1 TODO Step 1.1: Set up the Monorepo & Workspace

This first task is to create the physical layout for our code. We'll use a monorepo because it makes sharing code between our services (like the API and the web frontend) much simpler.

- ☐ Create the root project folder and initialize a Git repository.
- ☐ Create the core folder structure: `apps/`, `packages/`, and `infra/`.
- ☐ Initialize a package manager workspace (e.g., using ‘npm workspaces‘ or ‘pnpm‘).

3.2 TODO Step 1.2: Implement the CI Pipeline & Health Check

Before we write any real code, we'll write a test that proves our automation works. This test is simple, but it will validate our entire CI/CD pipeline. The goal is to have a pull request automatically trigger a process that builds, deploys, and tests our application in a temporary, isolated environment.

- ☐ Create a barebones API application in `apps/api` with a single `/health` endpoint that returns "ok".

☐ Write the Gherkin feature file for the pipeline test in your acceptance test suite.

```
Feature: Platform Health and CI Pipeline

  Scenario: The CI pipeline can build, deploy, and test the application
    When a change is pushed to a pull request
    Then the CI pipeline should run successfully
    And deploy the application to a temporary "ephemeral" environment
    And run an acceptance test against the ephemeral environment's "/health" endpo
    And the test should receive a 200 OK response with the body "ok"
```

☐ Create the CI configuration file (e.g., '.github/workflows/ci.yml') with stages for linting, testing, building, deploying, and acceptance testing.

```
name: ci
on: [pull_request]
jobs:
  lint_unit:
    # ... steps to checkout, install, lint, and run unit tests
  build:
    # ... steps to build a Docker image
  deploy_ephemeral:
    # ... steps to run a script that deploys the Docker image to a temporary envir
  acceptance_test:
    needs: deploy_ephemeral
    # ... steps to run the acceptance test against the ephemeral URL
```

☐ Write the single acceptance test described in the Gherkin scenario.

3.3 TODO Step 1.3: Enforce the Layered Architecture

Now we enforce the "Clean Workshop" principle. We'll add an automated rule to prevent dependencies from crossing the architectural layers incorrectly. This is an application of the **Dependency Inversion Principle**: our business logic should not depend on technical details; the details should depend on the business logic.

☐ Install a dependency analysis tool like 'dependency-cruiser'.

☐ Create a rule that makes it an error for your 'domain' package to import anything from your 'infrastructure' package.

```
// .dependency-cruiser.js
module.exports = {
  forbidden: [
    {
      name: "domain-must-be-pure",
      severity: "error",
      from: { path: "^packages/domain" },
      to: { path: "^packages/infrastructure" } // Or any other layer
    },
    // ... other rules
  ]
};
```

☐ Run the check in your CI pipeline's ‘lint_unit‘ job to enforce the rule automatically.

3.4 TODO Step 1.4: Build the Core Primitives

Finally, we build the core, reusable tools that will protect us from common bugs. These are **Value Objects**: small, immutable objects defined by their value, not their identity. They are like perfectly calibrated wrenches that you can always trust.

3.4.1 TODO Build the ‘Money‘ Primitive

The Problem: Computers are famously bad at floating-point math (e.g., in JavaScript, `0.1 + 0.2` is not `0.3`). Handling money with floats will lead to rounding errors. **The Solution**: We will represent all monetary values as integers of the smallest currency unit (e.g., cents). Our ‘Money‘ object will encapsulate this logic.

☐ Create a ‘Money‘ class in `packages/domain/shared/money.ts`.

☐ Store the amount as a private integer (‘amountInCents‘).

☐ Add immutable methods like ‘add(other: Money)‘, ‘subtract(other: Money)‘, and ‘multiply(factor: number)‘, which must **return a new Money object**.

☐ Write comprehensive, table-driven unit tests to cover all financial scenarios.

```
Feature: Money Value Object

  Scenario Outline: Money math is always correct
    Given money <amount1>
    And money <amount2>
    When I <operation> them
    Then the result is <expected>

    Examples:
      | amount1 | operation | amount2 | expected |
      | €10.00  | add       | €5.00   | €15.00   |
      | €10.00  | subtract  | €5.50   | €4.50    |
      | €1.10   | multiply  | 1.5     | €1.65    |
```

3.4.2 TODO Build the ‘Clock‘ Primitive

The Problem: Code that calls ‘Date.now()‘ is impossible to test reliably for time-sensitive logic. You can’t fast-forward, rewind, or freeze time. **The Solution**: We treat time as an external dependency. We define a ‘Clock‘ interface and "inject" it wherever we need the current time. For production, we use a ‘SystemClock‘; for tests, we use a ‘FixedClock‘.

- ☐ Create a ‘Clock‘ interface in `packages/domain/shared/clock.ts`.

  ```
  export interface Clock {
    now(): Date;
  }
  ```

- ☐ Create a ‘SystemClock‘ that implements the interface by calling ‘new Date()‘.

- ☐ Create a ‘FixedClock‘ that takes a ‘Date‘ in its constructor and always returns that same date.

- ☐ Write a unit test that uses the ‘FixedClock‘ to test a piece of time-sensitive logic.

  ```
  // Example of a unit test using the FixedClock
  it('should do something at a specific time', () => {
    const frozenTime = new Date('2025-10-10T10:00:00Z');
    const clock = new FixedClock(frozenTime);
  ```

```
  // Pass the clock to the object under test
  const myService = new MyTimeSensitiveService({ clock });

  // ... run the test
});
```