Enterprise Test Framework

Team Training & Onboarding Guide

Cucumber BDD • Kafka Events • Synthetic Data • Contract Testing • Performance

5 Phases • 20 Tasks • All Skill Levels

From test strategy to team enablement

★ **Novice** ★★ **Intermediate** ★★★ **Advanced** ★★★★ **Expert**

Quality Engineering Team

February 2026

Table of Contents

Tip: In Google Docs, use View → Show Outline for one-click navigation to any section.

SECTION I

Framework Onboarding & Maintenance

Setup • Training • Progressive Tasks • Solutions • Team Enablement

This section onboards new team members from zero to productive across all framework layers.

How to Use This Guide

This guide serves the entire QE team. Find your role below, start at the recommended phase, and work forward. Each task builds on the previous ones, with clear verification checkpoints so you know when you have mastered the material before moving on.

Where to Start (by Role)

Role	Start → End	Duration	Focus
Junior QA / Manual Tester	Phase 0 → Phase 1	3-5 days	Framework navigation, Gherkin literacy, safe modifications
QA Engineer (1-3 yrs)	Phase 0 (skim) → Phase 2	3-4 days	Creating generators, feature files, step definitions
SDET (3-7 yrs)	Phase 1 (skim) → Phase 3	2-3 days	Independent creation across all layers, domain depth
Sr. SDET / Lead (7+ yrs)	Phase 2 (skim) → Phase 4	1-2 days	Architecture, strategy docs, team enablement
Director / Principal	Phase 3 → Phase 4	1 day	Strategic artifacts, cross-layer integration, leadership

Complete Task Progression

Task	Level	Layer	Title	Time
0.1	★ NOVICE	Foundation	Clone, Build, and Explore the Project	20 min
0.2	★ NOVICE	Foundation	Read and Understand a Feature File	15 min
0.3	★ NOVICE	Foundation	Trace a Step from Feature to Java Code	20 min
0.4	★ NOVICE	Data Generation	Generate Synthetic Data and Inspect Output	15 min
1.1	★ NOVICE	Data Generation	Add a New Field to the User Model	20 min
1.2	★ NOVICE	Functional (API)	Add a Scenario Using Existing Steps	15 min
1.3	★★ INTERMEDIATE	Data Generation	Add a Symbol to the Market Data Watchlist	15 min
2.1	★★ INTERMEDIATE	Data Generation	Create a Payment Generator from Scratch	30-45 min
2.2	★★ INTERMEDIATE	Functional (API)	Write Feature File AND Step Definitions for Payments	45 min
2.3	★★ INTERMEDIATE	Kafka Events	Create Kafka Event Tests for Payments	30-45 min
2.4	★★ INTERMEDIATE	Database	Write Database Assertions for Payment Records	30 min
3.1	★★★ ADVANCED	Data Generation	Build an Options Contract Generator with OCC Symbology	45-60 min

3.2	★★★ **ADVANCED**	Kafka Events	Kafka Event Chain for Options Clearing (Novation)	45-60 min
3.3	★★★ **ADVANCED**	Database	Database Tests for Position Management	45 min
3.4	★★★ **ADVANCED**	Contract Testing	Pact Contract Tests for Clearing API	30-45 min
3.5	★★★ **ADVANCED**	Performance	JMeter Load Test for Clearing Throughput	30-45 min
4.1	★★★★ **EXPERT**	Full Stack	End-to-End Clearing Lifecycle Test	60 min
4.2	★★★★ **EXPERT**	Full Stack	Design a Test Data Strategy Document	45-60 min
4.3	★★★★ **EXPERT**	Full Stack	Create a Framework Extension Guide for Your Team	45 min

Prerequisites

Repository: https://github.com/crhoads1024/cucumber-kafka-test-framework

Key Rule: Complete every verification checkpoint before moving to the next task. If you cannot check off all items, re-read the hints section or ask your team lead for help.

Environment Setup

Follow these instructions to install all required tools. Pick your operating system and shell, then work through each section in order. At the end, run the verification commands to confirm everything is ready.

1. Install Java 17 (or newer)

We recommend Eclipse Temurin (formerly AdoptOpenJDK) — it is free, open-source, and widely used in enterprise environments. Amazon Corretto and Oracle JDK 17 also work.

macOS (zsh or bash)	

Option A:Homebrew	`# Install Homebrew if you don't have it:`
	`  /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
	
	`# Install Java 17:`
	`  brew install --cask temurin@17`
	
	`# Homebrew handles PATH automatically. Verify:`
	`  java -version`

Option B:SDKMAN	`# Install SDKMAN (manages multiple Java versions):`
	`  curl -s "https://get.sdkman.io" | bash`
	`  source "$HOME/.sdkman/bin/sdkman-init.sh"`
	
	`# Install Java 17:`
	`  sdk install java 17.0.13-tem`
	
	`# SDKMAN sets JAVA_HOME automatically. Verify:`
	`  java -version`

Set JAVA_HOME(if needed)	`# Find where Java is installed:`
	`  /usr/libexec/java_home -v 17`
	
	`# Add to ~/.zshrc (zsh) or ~/.bashrc (bash):`
	`  export JAVA_HOME=$(/usr/libexec/java_home -v 17)`
	`  export PATH="$JAVA_HOME/bin:$PATH"`
	
	`# Reload your shell:`
	`  source ~/.zshrc    # or source ~/.bashrc`

Windows (PowerShell)	

Option A:winget	`# Open PowerShell as Administrator and run:`
	`  winget install EclipseAdoptium.Temurin.17.JDK`
	
	`# Close and reopen PowerShell, then verify:`
	`  java -version`

Option B:Chocolatey	`# Install Chocolatey first (if not installed):`
	`# See https://chocolatey.org/install`
	
	`# Then install Java:`
	`  choco install temurin17 -y`
	
	`# Close and reopen PowerShell, then verify:`

	`java -version`

Option C:Manual	`1. Download from: https://adoptium.net/temurin/releases/?version=17`
	2. Run the .msi installer
	3. IMPORTANT: Check 'Set JAVA_HOME variable' during installation
	4. Check 'Add to PATH' during installation
	`5. Close and reopen PowerShell, then verify: java -version`

Set JAVA_HOME(if needed)	`# Check if already set:`
	`echo $env:JAVA_HOME`
	
	`# If empty, set it (adjust path to your install location):`
	`[Environment]::SetEnvironmentVariable("JAVA_HOME", "C:\Program Files\Eclipse Adoptium\jdk-17", "User")`
	`[Environment]::SetEnvironmentVariable("Path", $env:Path + ";$env:JAVA_HOME\bin", "User")`
	
	`# Close and reopen PowerShell for changes to take effect`

Linux (bash)	

Ubuntu/Debian	`sudo apt update`
	`sudo apt install -y temurin-17-jdk`
	
	`# If temurin package not available, add the repo first:`
	`sudo apt install -y wget apt-transport-https gpg`
	`wget -qO - https://packages.adoptium.net/artifactory/api/gpg/key/public | sudo gpg --dearmor -o /etc/apt/trusted.gpg.d/adoptium.gpg`
	`echo "deb https://packages.adoptium.net/artifactory/deb $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/adoptium.list`
	`sudo apt update && sudo apt install -y temurin-17-jdk`

Set JAVA_HOME	`# Add to ~/.bashrc:`
	`export JAVA_HOME=/usr/lib/jvm/temurin-17-jdk-amd64`
	`export PATH="$JAVA_HOME/bin:$PATH"`
	
	`source ~/.bashrc`
	`java -version`

2. Install Maven

Maven 3.8+ is required. It must be able to find your Java installation via JAVA_HOME.

macOS	

Homebrew	`brew install maven`
	
	`# Verify:`
	`mvn -version`
	`# Should show: Apache Maven 3.9.x and Java version: 17.x`

Windows (PowerShell)	

winget	`winget install Apache.Maven`
	
	`# Close and reopen PowerShell, then verify:`

	`mvn -version`

Chocolatey	`choco install maven -y`
	
	`# Close and reopen PowerShell, then verify:`
	`mvn -version`

Manual	`1. Download from: https://maven.apache.org/download.cgi (Binary zip archive)`
	2. Extract to C:\Program Files\Apache\maven
	3. Set environment variables in PowerShell:
	`[Environment]::SetEnvironmentVariable("M2_HOME", "C:\Program Files\Apache\maven", "User")`
	`[Environment]::SetEnvironmentVariable("Path", $env:Path + ";C:\Program Files\Apache\maven\bin", "User")`
	`4. Close and reopen PowerShell: mvn -version`

Linux (bash)	

Ubuntu/Debian	`sudo apt update`
	`sudo apt install -y maven`
	
	`# Verify:`
	`mvn -version`

3. Install Git

All Platforms	

macOS	`# Git comes with Xcode Command Line Tools:`
	`xcode-select --install`
	
	`# Or via Homebrew:`
	`brew install git`

Windows	`winget install Git.Git`
	
	`# Or download from: https://git-scm.com/download/win`
	`# IMPORTANT: During install, select 'Git from the command line and also from 3rd-party software'`
	`# This ensures git works in PowerShell, not just Git Bash`

Linux	`sudo apt update && sudo apt install -y git`

Configure(all platforms)	`# Set your identity (required for commits):`
	`git config --global user.name "Your Name"`
	`git config --global user.email "your.email@company.com"`
	
	`# Verify:`
	`git --version`
	`git config --list`

4. Install Docker Desktop

Docker is needed for running Kafka, PostgreSQL, and WireMock locally. It is not required for Phase 0-1 tasks but is needed from Phase 2 onward.

All Platforms

macOS	`brew install --cask docker`
	
	`# Or download from: https://www.docker.com/products/docker-desktop/`
	`# Choose Apple Silicon (M1/M2/M3/M4) or Intel based on your chip:`
	`uname -m    # arm64 = Apple Silicon, x86_64 = Intel`
	
	`# Launch Docker Desktop from Applications`
	`# Wait for the whale icon in the menu bar, then verify:`
	`docker --version`
	`docker compose version`

Windows	`winget install Docker.DockerDesktop`
	
	`# Or download from: https://www.docker.com/products/docker-desktop/`
	`# IMPORTANT: WSL 2 backend is recommended (installer will prompt)`
	`# After install, launch Docker Desktop and wait for it to start`
	
	`# Verify in PowerShell:`
	`docker --version`
	`docker compose version`

Linux	`# Follow official docs: https://docs.docker.com/engine/install/ubuntu/`
	`sudo apt update`
	`sudo apt install -y docker.io docker-compose-v2`
	`sudo usermod -aG docker $USER`
	
	`# Log out and back in, then verify:`
	`docker --version`
	`docker compose version`

Note	The command is 'docker compose' (with a space), NOT 'docker-compose' (with a hyphen). The hyphenated version is the old v1 tool and is no longer installed by default.

5. IDE Setup

Either IntelliJ IDEA or VS Code works. IntelliJ has stronger built-in Java and Maven support. VS Code requires extensions but is lighter weight and free.

IntelliJ IDEA (Recommended for Java)	

Install	Download IntelliJ IDEA Community (free) from:
	`https://www.jetbrains.com/idea/download/`
	
	`macOS: brew install --cask intellij-idea-ce`
	`Windows: winget install JetBrains.IntelliJIDEA.Community`

Open Project	1. File > Open > select the test-framework root folder (the one with pom.xml) 2. IntelliJ will detect Maven and ask to import — click 'Trust Project' 3. Wait for indexing to complete (progress bar in bottom right) `4. Verify: open any .java file — no red underlines means dependencies resolved`

ConfigureProject SDK	This is the most important step. IntelliJ needs to know which JDK to use.
	
	1. Open File > Project Structure (Cmd+; on Mac / Ctrl+Alt+Shift+S on Win)
	2. In the left panel, click 'Project'
	3. Under 'SDK', click the dropdown:
	`  • If temurin-17 appears — select it. Done.`
	`  • If not, click 'Add SDK' > 'Download JDK'`

	- Version: 17 - Vendor: Eclipse Temurin (AdoptOpenJDK HotSpot) - Click Download. IntelliJ installs it for you. 4. Under 'Language level', select '17 - Sealed types, always-strict floating-point' 5. Click 'Apply', then 'OK'
ConfigureMaven JDK	IntelliJ uses its own Maven runner that can pick a different JDK than your project. 1. Open Settings (Cmd+, on Mac / Ctrl+Alt+S on Win) 2. Navigate to: Build, Execution, Deployment > Build Tools > Maven > Runner 3. Under 'JRE', select 'Project SDK (temurin-17)' or 'Use Project JDK' 4. Click Apply If Maven builds still fail inside IntelliJ: 5. Go to: Build, Execution, Deployment > Build Tools > Maven > Importing 6. Set 'JDK for importer' to 'Use Project JDK' 7. Click Apply, then right-click pom.xml > Maven > Reimport
Verify SDKis Working	1. Open any Java file (e.g., User.java) 2. Check the bottom-right status bar — it should show 'Java 17' 3. If it shows a different version, click it and select temurin-17 4. Open the Maven tool window (right sidebar > Maven icon) 5. Click the refresh icon (circular arrows) to reload 6. Expand 'shared > Lifecycle' and double-click 'compile' 7. The Run window should show BUILD SUCCESS with Java 17
Useful Plugins	Go to Settings > Plugins and install: • Cucumber for Java (enables Gherkin syntax highlighting and step navigation) • Gherkin (feature file support) • Docker (optional: manage containers from IDE) After installing plugins, restart IntelliJ
Key Shortcuts	Cmd+Shift+F (Mac) / Ctrl+Shift+F (Win): Search all files Cmd+Click (Mac) / Ctrl+Click (Win): Jump to definition Cmd+Shift+T (Mac) / Ctrl+Shift+T (Win): Jump to test for class Right-click feature file > Run: Execute Cucumber scenarios

Visual Studio Code	

Install	`Download from: https://code.visualstudio.com/`
	
	`macOS: brew install --cask visual-studio-code`
	`Windows: winget install Microsoft.VisualStudioCode`
	`Linux: sudo snap install --classic code`

RequiredExtensions	Open VS Code, press Cmd+Shift+X (Mac) or Ctrl+Shift+X (Win) and install:
	
	• Extension Pack for Java (by Microsoft) — ESSENTIAL
	`Includes: Language Support, Debugger, Test Runner, Maven, Project Manager`
	• Cucumber (Gherkin) Full Support (by Alexander Krechik)
	`Enables: syntax highlighting, step definition navigation, auto-complete`
	• Docker (by Microsoft) — optional but helpful
	
	You can also install from the terminal:

```
  code --install-extension vscjava.vscode-java-pack
  code --install-extension alexkrechik.cucumberautocomplete
  code --install-extension ms-azuretools.vscode-docker
```

Find YourJDK Path

VS Code needs the exact path to your JDK installation. Find it first:

macOS (zsh/bash):

```
  /usr/libexec/java_home -v 17
  # Example output:
/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home
```

Windows (PowerShell):

```
  Get-ChildItem 'C:\Program Files\Eclipse Adoptium\' -Directory
  # Example output: jdk-17.0.13.11-hotspot
  # Full path: C:\Program Files\Eclipse Adoptium\jdk-17.0.13.11-hotspot

  # OR if installed via Chocolatey:
  Get-ChildItem 'C:\Program Files\Java\' -Directory
```

Linux (bash):

```
  readlink -f $(which java) | sed 's|/bin/java||'
  # Example output: /usr/lib/jvm/temurin-17-jdk-amd64
```

ConfigureJDK inVS Code

1. Open Command Palette: Cmd+Shift+P (Mac) / Ctrl+Shift+P (Win)
2. Type: 'Preferences: Open User Settings (JSON)' and press Enter
3. Add these settings (use the path you found above):

macOS example:

```
  {
    "java.jdt.ls.java.home":
"/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home",
    "java.configuration.runtimes": [
      {
        "name": "JavaSE-17",
        "path":
"/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home",
        "default": true
      }
    ]
  }
```

Windows example:

```
  {
    "java.jdt.ls.java.home": "C:\\Program Files\\Eclipse
Adoptium\\jdk-17.0.13.11-hotspot",
    "java.configuration.runtimes": [
      {
        "name": "JavaSE-17",
        "path": "C:\\Program Files\\Eclipse
Adoptium\\jdk-17.0.13.11-hotspot",
        "default": true
      }
    ]
  }
```

Linux example:

```
  {
    "java.jdt.ls.java.home": "/usr/lib/jvm/temurin-17-jdk-amd64",
    "java.configuration.runtimes": [
      {
        "name": "JavaSE-17",
        "path": "/usr/lib/jvm/temurin-17-jdk-amd64",
        "default": true
```

	`    }`
	`  ]`
	`}`

Why BothSettings?	• java.jdt.ls.java.home — tells the Java Language Server (the engine that provides
	`  code completion, error checking, and navigation) which JDK to use. This is the most`
	`  important setting. Without it, VS Code shows 'Java runtime could not be located.'`
	
	• java.configuration.runtimes — tells VS Code which JDK to use when RUNNING and
	`  COMPILING your project. The 'name' must match the Java version in pom.xml.`
	`  Our pom.xml specifies Java 17, so the name must be 'JavaSE-17'.`
	
	• Setting 'default: true' means this JDK is used for all projects unless overridden.

ConfigureMaven inVS Code	Add these to the same settings.json file:
	
	`{`
	`  "java.configuration.maven.userSettings": "~/.m2/settings.xml",`
	`  "maven.executable.path": "/usr/local/bin/mvn",`
	`  "java.import.maven.enabled": true`
	`}`
	
	Windows — adjust the Maven path:
	`  "maven.executable.path": "C:\\Program Files\\Apache\\maven\\bin\\mvn.cmd"`
	
	If you installed Maven via winget/choco, you can usually omit maven.executable.path
	as it will already be on your system PATH.

ConfigureCucumber	`# Add to the same settings.json for step definition navigation:`
	`  {`
	`    "cucumberautocomplete.steps": [`
	`      "functional-tests/src/test/java/**/*.java"`
	`    ],`
	`    "cucumberautocomplete.syncfeatures": "functional-tests/src/test/resources/features/**/*.feature"`
	`  }`
	
	This lets you Ctrl+Click on a step in a .feature file to jump to its Java definition.

Completesettings.json	Here is a complete example combining all settings (macOS):
	
	`  {`
	`    "java.jdt.ls.java.home": "/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home",`
	`    "java.configuration.runtimes": [`
	`      {`
	`        "name": "JavaSE-17",`
	`        "path": "/Library/Java/JavaVirtualMachines/temurin-17.jdk/Contents/Home",`
	`        "default": true`
	`      }`
	`    ],`
	`    "java.configuration.maven.userSettings": "~/.m2/settings.xml",`
	`    "java.import.maven.enabled": true,`
	`    "cucumberautocomplete.steps": [`
	`      "functional-tests/src/test/java/**/*.java"`
	`    ],`

	`"cucumberautocomplete.syncfeatures": "functional-tests/src/test/resources/features/**/*.feature"`
	`}`

Open Project	1. File > Open Folder > select the test-framework root 2. VS Code will detect pom.xml and activate Java extensions 3. Wait for 'Java: Ready' in the bottom status bar (may take 1-2 minutes) 4. If you see 'Java runtime could not be located' — recheck java.jdt.ls.java.home `5. Open Terminal (Ctrl+`) and run: mvn compile -pl shared,data-generator -am`

Verify SDKis Working	`1. Open any .java file (e.g., shared/src/.../model/User.java)`
	2. Check the bottom status bar — it should show 'Java 17' or 'JavaSE-17'
	3. Hover over an import statement — you should see Javadoc tooltip (not an error)
	4. Open a .feature file — step text should be syntax-highlighted in green/blue
	5. Ctrl+Click on a step (e.g., 'the response status should be') — it should jump to ApiSteps.java
	
	If the status bar shows the wrong Java version:
	`• Cmd+Shift+P > 'Java: Configure Java Runtime'`
	`• Under 'Project JDKs', change the JDK to temurin-17`
	`• Click 'Apply' and wait for VS Code to re-index`

Troubleshoot	• 'Java runtime could not be located' → java.jdt.ls.java.home path is wrong. Re-run the
	`'Find Your JDK Path' step above and paste the exact output.`
	`• Red squiggles everywhere → Maven import failed. Open terminal, run: mvn compile -pl shared -am`
	`Then Cmd+Shift+P > 'Java: Clean Java Language Server Workspace' > Restart.`
	• Cucumber steps not linking → Check cucumberautocomplete.steps path matches your project.
	`• 'Build failed' in terminal → Likely wrong Java version. Run: java -version in VS Code terminal.`
	`If it shows Java 8 or 11, your system PATH has an older JDK. Fix JAVA_HOME (Section 1).`

Key Shortcuts	Cmd+Shift+F (Mac) / Ctrl+Shift+F (Win): Search all files F12: Go to definition Cmd+P (Mac) / Ctrl+P (Win): Quick open file by name Ctrl+`: Toggle integrated terminal Cmd+Shift+P (Mac) / Ctrl+Shift+P (Win): Command palette

6. Install Modules to Local Maven Repository

This is a multi-module Maven project. The functional-tests module depends on shared and data-generator, but Maven cannot find them unless they are installed into your local repository (~/.m2/repository). This step is required before running ANY tests.

Install Shared Modules (all platforms)	

Why ThisIs Needed	Maven resolves dependencies by looking in two places:
	`1. Remote repositories (Maven Central, etc.)`
	`2. Your local repository (~/.m2/repository/)`
	
	Our shared and data-generator modules are NOT published to Maven Central.
	`They only exist in this project. 'mvn install' compiles them and copies` the resulting JAR files into ~/.m2/repository so that functional-tests, contract-tests, and perf-tests can find them as dependencies.

	Without this step, you get:
	`  Could not find artifact`
	`com.enterprise.testing:shared:jar:1.0.0-SNAPSHOT`

Command	`# From the project root directory (where the root pom.xml lives):`
	`  mvn install -pl shared,data-generator -am -DskipTests`
	
	`# Breakdown:`
	`#   -pl shared,data-generator   = only build these two modules`
	`#   -am                         = also build any modules they depend on`
	`#   -DskipTests                 = skip tests during install (faster)`
	
	`# Should print: BUILD SUCCESS for both modules`

When toRe-Run	`Re-run mvn install after ANY of these changes:`
	• You modify a model class in shared/ (e.g., User.java, Trade.java)
	• You modify a generator in data-generator/ (e.g., TradeGenerator.java)
	• You add a new class to shared/ or data-generator/
	`• You pull new changes from git that touch shared/ or data-generator/`
	
	You do NOT need to re-run install for changes to:
	• Feature files (.feature)
	• Step definitions in functional-tests/
	• JMeter test plans

Alternative:Use -am flag	Instead of running install separately, you can add -am to any test command: `  mvn test -pl functional-tests -am -Dcucumber.filter.tags="@smoke"` The -am flag tells Maven to build upstream dependencies automatically. This is slower (rebuilds shared + data-generator every time) but convenient. `Recommendation: run 'mvn install' once after cloning and after pulling` changes, then use the regular test commands without -am for speed.

7. Test Target Architecture (What the Tests Connect To)

When you run tests, they send real HTTP requests to an API server. If no server is running, you get 'Connection refused.' This section explains what each environment uses as a test target and how to run one locally.

Environment	Test Target	How Tests Connect	Who Uses It
Local Dev	WireMock (stubbed API) Runs in Docker on port 8080	Tests hit http://localhost:8080 WireMock returns pre-defined JSON responses	Every developer, daily work
CI Pipeline	WireMock in Docker (same stubs, automated)	GitHub Actions starts Docker Compose before running tests	Every PR, every merge to main
QA / Staging	Real deployed services (microservices on internal cluster)	Tests point at QA URL: https://api.qa.internal	QA team, integration testing
UAT / Pre-Prod	Near-production services with synthetic market data	Tests point at UAT URL: https://api.uat.internal	Performance tests, release sign-off

How the Base URL is Configured

Config File	The target server URL is set in FrameworkConfig.java:
	`  shared/src/.../config/FrameworkConfig.java`
	
	`  public String getBaseUrl() {`
	`      return get("app.base.url", "http://localhost:8080");`
	`  }`
	
	`The default is http://localhost:8080 (WireMock's port).`
	This means: if you start WireMock, tests work with zero config changes.

OverridePriority	FrameworkConfig checks three places in order (first match wins):
	
	1. Environment variable: APP_BASE_URL
	`    (dots become underscores, uppercased)`
	
	2. System property: -Dapp.base.url=...
	`    (passed on the mvn command line)`
	
	3. Properties file: framework.properties
	`    (in shared/src/main/resources/)`
	
	`4. Default:  http://localhost:8080`

SwitchEnvironments	`# Local dev (WireMock - default, no override needed):`
	`  mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`
	
	`# Point at QA environment:`
	`  APP_BASE_URL=https://api.qa.internal mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`
	
	`# Or via system property:`
	`  mvn test -pl functional-tests -Dapp.base.url=https://api.qa.internal -Dcucumber.filter.tags="@smoke"`
	
	`# In CI/CD (GitHub Actions), set as a pipeline variable:`
	`  env:`
	`    APP_BASE_URL: ${{ secrets.QA_API_URL }}`

OtherConnections	The same pattern applies to all infrastructure connections: • Kafka: KAFKA_BOOTSTRAP_SERVERS (default: localhost:9092) `• Database:  DB_URL                   (default: jdbc:postgresql://localhost:5432/testdb)` • DB User: DB_USERNAME (default: testuser) • DB Pass: DB_PASSWORD (default: testpass) All defaults point to Docker Compose services, so starting Docker gives you a fully wired local environment with no config changes.

Start WireMock Locally (Required for Running Tests)

StartWireMock	`# From the project root (where docker-compose.yml lives):`
	`  docker compose up -d wiremock`
	
	`# Verify it started:`
	`  curl http://localhost:8080/api/health`
	`  # Should return: {"status":"UP","service":"clearing-api",...}`

	`# To see all registered stubs:`
	`  curl http://localhost:8080/__admin/mappings | python3 -m json.tool`

Start AllServices	`# For integration tests that need Kafka and Postgres too:`
	`  docker compose up -d`
	
	`# This starts: WireMock (8080), Kafka (9092), Postgres (5432), Zookeeper (2181)`
	`# Wait ~30 seconds for Kafka to fully initialize`
	
	`# Verify all services are running:`
	`  docker compose ps`

Run SmokeTests	`# After WireMock is running:`
	`  mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`
	
	`# If you see 'Connection refused' — WireMock is not running.`
	`# Run: docker compose up -d wiremock`

StopServices	`# Stop all services when done:`
	`  docker compose down`
	
	`# Stop but keep data (Postgres tables persist):`
	`  docker compose stop`
	
	`# Full cleanup (remove volumes, start fresh):`
	`  docker compose down -v`

What AreWireMockStubs ?	WireMock is a 'fake' API server. It reads JSON files from docker/wiremock/mappings/ and returns pre-defined responses.
	
	Example: when tests POST to /api/trades, WireMock returns a 201 with a realistic trade response (tradeId, status: EXECUTED, etc.)
	
	Stubs are defined in:
	`  docker/wiremock/mappings/order-api.json`
	`  docker/wiremock/mappings/trade-settlement-api.json`
	
	This is 'service virtualization' — testing without a real backend.
	It is listed as a requirement in the OCC job posting.

8. Verify Your Environment

Run all of these commands. If any fail, go back to the relevant section above.

Verification Checklist (all shells)	

Commands	`  java -version           # Should show: openjdk 17.x.x`
	`  echo $JAVA_HOME         # (bash/zsh) Should show a path`
	`  echo $env:JAVA_HOME     # (PowerShell) Should show a path`
	`  mvn -version            # Should show: Apache Maven 3.8+ and Java 17`
	`  git --version           # Should show: git version 2.x`
	`  docker --version        # Should show: Docker version 2x.x`
	
	`# Clone, install, and verify:`
	`  git clone https://github.com/crhoads1024/cucumber-kafka-test-framework.git`
	`  cd cucumber-kafka-test-framework`
	`  mvn install -pl shared,data-generator -am -DskipTests`
	`  mvn compile -pl functional-tests`

	`# Start WireMock and run smoke tests:`
	`  docker compose up -d wiremock`
	`  curl http://localhost:8080/api/health`
	`  mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`

Troubleshoot	• 'java' not recognized → JAVA_HOME not set or not on PATH. Re-do Step 1.
	• `mvn shows wrong Java version → Multiple JDKs installed. Set JAVA_HOME explicitly.`
	• `Maven downloads fail → Check internet/proxy. Try: mvn compile -U to force refresh.`
	• VS Code says 'Java runtime not found' → Set java.jdt.ls.java.home in settings.
	• docker: command not found → Docker Desktop not installed or not running.
	• 'Could not find artifact shared:jar' → You skipped Step 6. Run:
	`    mvn install -pl shared,data-generator -am -DskipTests`
	• 'Connection refused' on tests → WireMock not running. Run:
	`    docker compose up -d wiremock`
	• `curl localhost:8080 fails → Docker Desktop not running. Launch it first.`
	• `Tests fail with dependency errors after git pull → Re-run the install command.`

✓ If all commands pass and BUILD SUCCESS appears, you are ready to start Phase 0.

Concepts & Technologies You Will Learn

This framework uses many tools, patterns, and concepts that may be new to you. This section explains each one so you have context before encountering them in the tasks. You do not need to memorize everything here — use it as a reference you come back to when you see something unfamiliar.

Concepts are grouped by the phase where you will first encounter them.

Phase 0 — Foundation Concepts

Java & Project Structure	
Maven	**What it is:** A build tool that compiles Java code, downloads dependencies (libraries), runs tests, and packages your project. The pom.xml file is its configuration — think of it like package.json in Node or requirements.txt in Python, but it also defines how to build and test. **Why it matters here:** Every module in this framework has its own pom.xml. The root pom.xml ties them together as a 'multi-module project.' When you run mvn compile, Maven reads pom.xml to know what to build and in what order. **Learn more:** *Search: 'Maven in 5 minutes' on maven.apache.org*
Multi-ModuleProject	**What it is:** A Maven project split into multiple sub-projects (modules) that depend on each other. Our framework has 5 modules: shared, data-generator, functional-tests, contract-tests, and perf-tests. **Why it matters here:** Splitting into modules means each layer can be built and tested independently. The shared module holds models used by everyone. The -pl flag in mvn compile -pl shared,data-generator tells Maven to only build specific modules. **Learn more:** *Search: 'Maven multi-module project structure'*
@JsonProperty	**What it is:** A Jackson annotation that tells the JSON serializer what name to use when converting a Java object to JSON (and back). Jackson is the most popular Java library for reading and writing JSON. **Why it matters here:** Every model in this framework (User, Order, Trade, Settlement) uses @JsonProperty so that generated test data serializes to clean JSON files. Without it, your Java field name 'firstName' would still work, but @JsonProperty makes the mapping explicit and protects against refactoring mistakes. **Learn more:** *Search: 'Jackson @JsonProperty annotation tutorial'*
@JsonIgnoreProperties	**What it is:** A Jackson annotation on a class that says: 'if you encounter JSON fields that don't match any Java field, ignore them instead of throwing an error.' **Why it matters here:** This is what makes the framework backward-compatible. In Task 1.1 you add a 'department' field to User. Without @JsonIgnoreProperties(ignoreUnknown=true), any code that reads the OLD User JSON (without department) would crash. With it, the missing field is silently skipped. **Learn more:** *Search: 'Jackson @JsonIgnoreProperties ignoreUnknown'*
Getters &Setters	**What it is:** Java methods that read (get) and write (set) private fields. Example: getFirstName() returns the firstName field, setFirstName("Chris") sets it. This is called 'encapsulation' — the field is private, but controlled access is public. **Why it matters here:** Every model class follows this pattern. Jackson uses getters/setters to convert between Java objects and JSON. When you see user.getEmail(), it is reading the private email field through its getter. **Learn more:** *Search: 'Java getter setter explained'*

BDD & Cucumber

BDD (BehaviorDrivenDevelopment)	**What it is:** A testing approach where tests are written in plain English (or close to it) before or alongside the code. The format is Given-When-Then: Given some setup, When an action happens, Then assert a result. **Why it matters here:** BDD bridges the gap between QA, developers, and product owners. Everyone can read a .feature file and understand what is being tested. This is why the OCC job posting specifically lists BDD as a required skill. **Learn more:** *Search: 'BDD Cucumber tutorial for beginners'*
Cucumber	**What it is:** A testing framework that reads .feature files written in Gherkin syntax (Given/When/Then) and executes matching Java methods. Each line in a feature file maps to a Java method via annotations like @Given, @When, @Then. **Why it matters here:** Cucumber is the engine that connects business-readable scenarios to executable test code. When you write 'Then the response status should be 200', Cucumber finds the Java method annotated with that exact text and runs it. **Learn more:** *Search: 'Cucumber Java tutorial step by step'*
Gherkin	**What it is:** The language used in .feature files. It has specific keywords: Feature, Scenario, Given, When, Then, And, But, Background, Scenario Outline, Examples. It is designed to be human-readable. **Why it matters here:** You will write Gherkin starting in Phase 1. The most common mistake is writing steps that are too technical. Good Gherkin reads like a user story, not like code. **Learn more:** *Search: 'Gherkin syntax reference'*
Tags (@smoke,@functional,@negative)	**What it is:** Labels on scenarios or features that let you run specific subsets. @smoke runs quick health checks. @negative runs error-handling tests. You can combine them: mvn test -Dcucumber.filter.tags="@smoke and @trade". **Why it matters here:** Tags are how you control what runs in CI/CD. A pull request might only run @smoke tests (fast). A nightly build runs everything. A release candidate runs @e2e (end-to-end). Understanding tags is essential for pipeline design. **Learn more:** *Search: 'Cucumber tags filtering'*
StepDefinitions	**What it is:** Java methods annotated with @Given, @When, or @Then that contain the actual test logic. The annotation text must exactly match the step text in the .feature file. Parameters like {string} and {int} capture values from the step text. **Why it matters here:** In Phase 0 you trace steps from feature files to Java code. In Phase 1 you reuse existing steps. In Phase 2 you write your own. Understanding this mapping is the core SDET skill. **Learn more:** *Search: 'Cucumber step definitions Java example'*
Background	**What it is:** A Gherkin keyword that defines steps that run before EVERY scenario in a feature file. It replaces duplicated Given steps across scenarios. **Why it matters here:** Most feature files in this framework use Background to load synthetic test data. Without it, every scenario would repeat the same data-loading step. Background keeps features DRY (Don't Repeat Yourself).

Test Data

SyntheticData	**What it is:** Fake but realistic data generated by code rather than manually created or copied from production. Libraries like Datafaker produce realistic names, addresses, emails, and numbers. **Why it matters here:** The data-generator module creates synthetic users, orders, trades, and settlements. Each time you run it, you get fresh data. This eliminates test data dependencies and makes tests repeatable across environments. **Learn more:** *Search: 'Datafaker Java library'*

SeededRandom	**What it is:** When you create a Random object with a seed number (e.g., new Random(42)), it produces the same sequence of 'random' numbers every time. This makes data generation reproducible. **Why it matters here:** Seeded generators are critical for debugging. If a test fails with seed 42, you can re-run with the same seed to reproduce the exact same data. Without seeding, failures can be impossible to reproduce. **Learn more:** *Search: 'Java Random seed reproducibility'*
TestDataRegistry	**What it is:** The central hub class that orchestrates all generators and stores generated data by scenario name. You call registry.generateScenario("order-happy-path", 1, 2) and it creates users, orders, events, and optionally trades/settlements. **Why it matters here:** Every test layer reads from the same registry, ensuring all layers test against consistent data. This is the key architectural pattern: generate once, consume everywhere.

Phases 1–2 — API, Events, and Database Concepts

API Testing	
REST API	**What it is:** A web service that accepts HTTP requests (GET, POST, PUT, DELETE) and returns JSON responses. GET retrieves data, POST creates data, PUT updates data, DELETE removes data. **Why it matters here:** This framework tests REST APIs by sending requests and asserting on the response status code and body content. When you see POST /api/trades, that means 'send trade data to the trade creation endpoint.' **Learn more:** *Search: 'REST API basics for testers'*
REST Assured	**What it is:** A Java library for testing REST APIs. It provides a fluent syntax: given().contentType(JSON).body(data).post("/api/trades"). It handles HTTP connections, serialization, and response parsing. **Why it matters here:** All API step definitions in this framework use REST Assured. When TradeSteps.java calls given().post(endpoint), REST Assured sends the HTTP request and captures the response for assertions. **Learn more:** *Search: 'REST Assured tutorial Java'*
HTTP StatusCodes	**What it is:** Numbers returned by APIs that indicate success or failure. 200 = OK, 201 = Created, 400 = Bad Request (your fault), 404 = Not Found, 409 = Conflict, 500 = Server Error (their fault). **Why it matters here:** Every API test asserts a status code first. If you expect 201 and get 500, the server has a bug. If you expect 400 and get 200, validation is broken. Status codes are the first line of defense. **Learn more:** *Search: 'HTTP status codes cheat sheet'*
AssertJ	**What it is:** A Java assertion library that provides readable assertions: assertThat(price).isGreaterThan(BigDecimal.ZERO). It produces clear failure messages: 'Expected price to be greater than 0 but was -5.00'. **Why it matters here:** All step definitions use AssertJ for assertions instead of JUnit's basic assertEquals. The fluent syntax makes test code readable and failure messages actionable. **Learn more:** *Search: 'AssertJ introduction'*

Kafka & Event-Driven Architecture	
Apache Kafka	**What it is:** A distributed messaging system where services publish events to 'topics' and other services consume them. Think of it like a bulletin board: producers post messages, consumers read them. **Why it matters here:** In this framework, trade execution publishes TRADE_EXECUTED to the trade.events topic. The settlement service consumes that event and creates a settlement. Tests verify that the right events appear on the right topics. **Learn more:** *Search: 'Apache Kafka explained simply'*
Topics	**What it is:** Named channels in Kafka where events are published. Examples: trade.events, settlement.events, payment.events. Each topic holds events of a specific type. **Why it matters here:** Feature files reference topics in steps like: 'Kafka consumers are listening on topics: trade.events'. The test subscribes to that topic and waits for expected events to appear.
Events /Messages	**What it is:** JSON objects published to Kafka topics. Each event has a type (TRADE_EXECUTED), a payload (trade details), a source (which service sent it), and a correlation ID (to link related events).

	Why it matters here: The TradeKafkaEventGenerator creates expected events from generated trades. Tests compare actual Kafka events against these expected events to verify the system works correctly.
Event-DrivenArchitecture	**What it is:** A system design where services communicate by producing and consuming events instead of calling each other directly. This makes services loosely coupled — they don't need to know about each other. **Why it matters here:** OCC's clearing system is event-driven. A trade event triggers clearing, which triggers margin calculation, which triggers settlement. Testing this chain is a key skill for this role. **Learn more:** *Search: 'Event-driven architecture for beginners'*

Database Testing	
SQL	**What it is:** Structured Query Language for reading and writing data in relational databases. SELECT reads data, INSERT adds data, UPDATE modifies data, DELETE removes data. **Why it matters here:** Database step definitions use SQL queries like: SELECT * FROM trades WHERE trade_id = ? to verify that API operations actually persisted data correctly. **Learn more:** *Search: 'SQL basics tutorial'*
Foreign Key(FK)	**What it is:** A database constraint that enforces relationships between tables. If settlements.trade_id REFERENCES trades(trade_id), you cannot create a settlement for a trade that doesn't exist. **Why it matters here:** The init-db.sql file defines these constraints. Tests verify referential integrity: every settlement must reference a valid trade. This catches orphaned records.
ParameterizedQueries	**What it is:** SQL queries that use ? placeholders instead of string concatenation. Example: SELECT * FROM trades WHERE trade_id = ? with the value passed separately. This prevents SQL injection attacks. **Why it matters here:** Even in test code, always use parameterized queries. The DatabaseHelper class provides queryRow(sql, params...) for this purpose. Never concatenate user input into SQL strings. **Learn more:** *Search: 'SQL injection prevention parameterized queries'*
Docker	**What it is:** A tool that runs applications in isolated containers. This framework uses Docker to run PostgreSQL (database), Kafka (messaging), and WireMock (API mocking) locally. **Why it matters here:** Running docker compose up from the docker/ directory starts all infrastructure. This means every developer gets an identical test environment regardless of their operating system. **Learn more:** *Search: 'Docker for beginners'*

Phases 3–4 — Advanced & Domain Concepts

Contract Testing	
ContractTesting	**What it is:** A testing approach where the consumer of an API defines a 'contract' (what requests it sends and what responses it expects), and the provider verifies it can fulfill that contract. This catches breaking changes before deployment. **Why it matters here:** The contract-tests module uses Pact to define contracts and JSON Schema to validate event formats. If the clearing API changes its response format, the contract test fails before any functional test runs. **Learn more:** *Search: 'Pact contract testing introduction'*
Pact	**What it is:** A contract testing framework. The consumer writes a test that generates a Pact file (a JSON contract). The provider runs that Pact file to verify it can produce the expected responses. **Why it matters here:** ClearingApiContractTest.java generates Pact files in target/pacts/. These files can be shared with the provider team (e.g., via a Pact Broker) so they know what consumers expect. **Learn more:** *Search: 'Pact JVM consumer test tutorial'*
JSON Schema	**What it is:** A standard for describing the structure of JSON data: what fields are required, what types they should be, what values are allowed. Think of it as a blueprint for JSON. **Why it matters here:** The contract-tests module validates Kafka events against JSON schemas. If an event is missing a required field or has the wrong type, the schema test catches it. **Learn more:** *Search: 'JSON Schema tutorial'*
Type Matchingvs ExactMatching	**What it is:** In Pact, type matching (stringType, decimalType) checks that a field exists and has the right data type, but ignores the actual value. Exact matching (stringValue) checks the exact value. **Why it matters here:** Use type matching for IDs, timestamps, and amounts (values change between runs). Use exact matching for statuses, enums, and flags (values must be specific). Getting this wrong makes contracts either too brittle or too loose.

Performance Testing	
JMeter	**What it is:** An open-source load testing tool. It simulates multiple users hitting an API simultaneously to measure response times, throughput, and error rates under load. **Why it matters here:** The perf-tests module contains JMeter test plans (.jmx files) that use synthetic data from CSV files. The check-perf-thresholds.py script fails the CI build if response times exceed defined limits. **Learn more:** *Search: 'JMeter getting started tutorial'*
Throughput	**What it is:** The number of requests a system can handle per second. A clearing system might need to handle 10,000 trades per second during market open. **Why it matters here:** JMeter reports measure throughput. If a code change drops throughput from 5,000 to 500 requests/second, the performance test catches the regression.
p99 Latency	**What it is:** The response time that 99% of requests are faster than. If p99 = 500ms, then 99 out of 100 requests complete in under 500ms. The remaining 1% are the 'tail latency.' **Why it matters here:** p99 matters more than average in financial systems. An average of 50ms is useless if 1% of requests take 10 seconds — those slow requests might miss settlement deadlines.

Financial Domain (OCC-Specific)	
OCC (OptionsClearingCorporation)	**What it is:** The world's largest equity derivatives clearing organization. OCC clears all US-listed options contracts and certain futures. It guarantees that both sides of every trade are honored. **Why it matters here:** This entire framework is designed around OCC's domain. The trade generators, settlement logic, and clearing event chains model how OCC actually operates. **Learn more:** *Search: 'OCC what does the options clearing corporation do'*
Novation	**What it is:** The process where OCC inserts itself as the counterparty to every trade. If Trader A sells to Trader B, after novation: Trader A's counterparty is OCC, and Trader B's counterparty is OCC. This eliminates counterparty risk. **Why it matters here:** Task 3.2 tests the novation event chain. The NOVATION_COMPLETE event must show OCC as the central counterparty. This is OCC's core function and will need to be validated.. **Learn more:** *Search: 'OCC novation clearing process'*
T+1Settlement	**What it is:** US equities and options settle one business day after the trade date (T+1, where T = trade date). This means if you buy stock on Monday, the shares and cash actually transfer on Tuesday. Crypto settles T+0 (same day). **Why it matters here:** The SettlementGenerator calculates settlement dates by adding business days and skipping weekends. Tests verify that settlement dates are correct, which is critical for clearing operations.
OCC OptionSymbology	**What it is:** The standard format for option contract identifiers: AAPL 240315C00150000. This encodes the underlying (AAPL), expiration date (2024-03-15), option type (C=Call), and strike price ($150.00). **Why it matters here:** Task 3.1 builds a generator that creates valid OCC symbols. Parsing and generating these symbols correctly is a daily task at OCC.
Margin	**What it is:** Collateral required to hold a position. Initial margin is required to open a position. Maintenance margin is the minimum required to keep it open. If your account falls below maintenance margin, you get a margin call. **Why it matters here:** Task 3.2 includes MARGIN_CALCULATED and MARGIN_CALL_ISSUED events. OCC calculates margin requirements for every clearing member daily. Incorrect margin calculations are a systemic risk.
BigDecimal	**What it is:** A Java class for precise decimal arithmetic. Unlike double (which has floating-point errors: 0.1 + 0.2 = 0.30000000000000004), BigDecimal is exact. All financial calculations must use BigDecimal. **Why it matters here:** Every price, amount, and fee in this framework uses BigDecimal. If you see double used for money anywhere, that is a bug. This is a non-negotiable rule in financial software. **Learn more:** *Search: 'Java BigDecimal vs double financial calculations'*

CI/CD & DevOps	
CI/CD	**What it is:** Continuous Integration / Continuous Delivery. CI automatically builds and tests code when you push to a repository. CD automatically deploys tested code to environments. Together, they catch bugs early and ship faster. **Why it matters here:** The .github/workflows/test-pipeline.yml file defines the CI pipeline. When you push code, GitHub Actions runs: compile → generate data → functional tests → contract tests → performance tests. If any step fails, the pipeline stops. **Learn more:** *Search: 'CI/CD explained simply'*

GitHubActions	**What it is:** GitHub's built-in CI/CD platform. Workflows are defined in YAML files. Each workflow has jobs, and each job has steps that run shell commands or use pre-built actions. **Why it matters here:** The framework's pipeline runs inside Ubuntu containers on GitHub's servers. This is why the MarketDataProvider needs fallback data — those containers may not be able to reach Yahoo Finance. **Learn more:** *Search: 'GitHub Actions quickstart'*
DockerCompose	**What it is:** A tool for defining and running multi-container Docker applications. The docker-compose.yml file describes all services (Kafka, PostgreSQL, WireMock) and their configuration in one place. **Why it matters here:** Running docker compose up starts the entire test infrastructure. Running docker compose down tears it down. This ensures every developer has an identical environment.
PipelineArtifacts	**What it is:** Files produced by one CI pipeline stage that are consumed by another. The data-generator produces JSON files that functional-tests reads. These files are passed between stages as 'artifacts.' **Why it matters here:** The TestDataRegistry.persistToDisk() method writes to a directory that the CI pipeline uploads as an artifact. The next job downloads it and loads the data. This decouples data generation from test execution.

Phase 0: Getting Oriented

Start here if you are new to the team, new to Java, or have never worked with BDD test frameworks. These tasks require no coding — you will explore the existing framework, run tests, and understand the architecture before writing anything.

Task 0.1 • Clone, Build, and Explore the Project ★ NOVICE | **Foundation** | ⏱ 20 min	
Objective	Get the framework running on your machine and understand the project structure.
Prerequisites	▸ Git installed ▸ Java 17+ installed (run: java -version) ▸ Maven installed (run: mvn -version) ▸ An IDE (IntelliJ IDEA Community is free)
Steps	**1.** Clone the repo: git clone https://github.com/crhoads1024/cucumber-kafka-test-framework.git **2.** Open the project in your IDE and let Maven download dependencies **3.** Run: mvn install -pl shared,data-generator -am -DskipTests (this installs shared modules to your local Maven repository) **4.** Run: mvn compile -pl functional-tests (this verifies functional-tests can find the installed modules) **5.** Open pom.xml at the root — identify the 5 modules listed under <modules> **6.** Open shared/src/.../model/ — read User.java and Order.java. Note the @JsonProperty annotations **7.** Open data-generator/src/.../generators/UserGenerator.java — read the generate() method **8.** Draw a diagram (paper is fine) showing: shared → data-generator → functional-tests

Code	`git clone`
	`https://github.com/crhoads1024/cucumber-kafka-test-framework.git`
	`cd cucumber-kafka-test-framework`
	
	`# IMPORTANT: Install shared modules to local Maven repo first:`
	`mvn install -pl shared,data-generator -am -DskipTests`
	
	`# Then verify downstream modules can resolve them:`
	`mvn compile -pl functional-tests`
	`# Both should print BUILD SUCCESS`

Verify	✓ mvn install completes with BUILD SUCCESS for both shared and data-generator ✓ mvn compile -pl functional-tests succeeds with no 'Could not find artifact' errors ✓ You can name all 5 modules and explain each in one sentence ✓ You can explain what @JsonProperty does ✓ Your diagram shows the dependency flow between modules
If Stuck	💡 *If Maven fails, check: java -version shows 17+ and mvn -version shows 3.8+* 💡 *If you see 'Could not find artifact com.enterprise.testing:shared' it means mvn install was skipped — run it first* 💡 *IntelliJ tip: right-click pom.xml > Maven > Reimport to refresh dependencies*

Task 0.2 • Read and Understand a Feature File

★ NOVICE | Foundation | ⏱ 15 min

Objective	Learn Gherkin syntax by reading an existing feature file. No coding required.
Prerequisites	▸ Task 0.1 completed
Steps	1. Open functional-tests/src/test/resources/features/order-api.feature 2. Identify: the Feature name, the Background section, each Scenario 3. For each scenario, identify the Given (setup), When (action), Then (assertion) steps 4. Notice the @functional, @smoke, and @negative tags — these control which tests run 5. Now open features/trade-execution.feature and compare the structure 6. Write down: What is the difference between @smoke and @negative scenarios? 7. Write down: Why does Background exist? What would happen without it?
Verify	✓ You can explain Given/When/Then in your own words ✓ You understand Background runs before EVERY scenario ✓ You can explain what tags do and how they filter execution ✓ You identified at least 2 differences between order and trade features
If Stuck	💡 *Think of Given as 'arrange', When as 'act', Then as 'assert' — same as AAA pattern* 💡 *Tags work like labels: @smoke = quick checks, @negative = error cases*

Task 0.3 • Trace a Step from Feature to Java Code

★ NOVICE | Foundation | ⏱ 20 min

Objective	Understand how Cucumber connects .feature files to Java step definitions.
Prerequisites	▸ Tasks 0.1 and 0.2 completed
Steps	1. Open order-api.feature and find: Then the response status should be {int} 2. Open functional-tests/src/.../steps/ApiSteps.java 3. Find the method with @Then("the response status should be {int}") 4. Read the method body — it asserts the status code from TestContext 5. Open SharedTestContext.java — understand why it exists (shares state between step classes) 6. Open TestContext.java — note the fields: lastApiResponse, currentDataSet, kafkaConsumer 7. Trace another step: Given synthetic test data is loaded for scenario {string} → find it in CommonSteps.java
Verify	✓ You can explain: .feature step text → @Then annotation → Java method ✓ You understand TestContext shares state between step definition classes ✓ You can find any step definition by searching for its annotation text
If Stuck	💡 *In IntelliJ, Cmd+Shift+F searches all files — search for the step text* 💡 *{string} captures quoted text, {int} captures numbers — they become method parameters*

Task 0.4 • Generate Synthetic Data and Inspect Output

★ NOVICE | Data Generation | ⏱ 15 min

Objective	Run the data generator and examine what it produces.
Prerequisites	▸ Task 0.1 completed
Steps	1. Run the data generator from terminal (see code block below) 2. Open the generated-data/ directory that was created 3. Open order-happy-path.json — find the user, orders, and expectedEvents sections 4. Open trade-happy-path.json — find trades, settlements, and marketSnapshots 5. Compare a trade price against the marketSnapshot for the same symbol 6. Open jmeter-users.csv — this is what the performance test layer consumes

Code	`mvn exec:java -pl data-generator \`
	`  -Dexec.mainClass="com.enterprise.testing.datagen.DataGeneratorMain"`
	
	`cat generated-data/trade-happy-path.json | python3 -m json.tool | head -80`

Verify	✓ generated-data/ contains JSON files for each scenario ✓ Trade prices are near the bid/ask from the market snapshot ✓ jmeter-users.csv has headers and data rows
If Stuck	💡 *If mvn exec:java fails, make sure you ran mvn compile first* 💡 *python3 -m json.tool pretty-prints JSON for readability*

Phase 1: Guided Modifications

Now you will make small, safe changes to existing code. Each task modifies one or two files and has a clear before/after you can verify. If something breaks, git checkout -- <file> to undo.

Task 1.1 • Add a New Field to the User Model ★ **NOVICE** | **Data Generation** | ⏱ 20 min	
Objective	Add a 'department' field to User and populate it in the generator.
Prerequisites	▸ Phase 0 completed
Steps	**1.** Open shared/src/.../model/User.java **2.** Add: @JsonProperty("department") private String department; **3.** Add getter and setter following the existing pattern **4.** Open data-generator/src/.../generators/UserGenerator.java **5.** In generate(), add: user.setDepartment(randomDepartment()); **6.** Add a private method returning one of: Trading, Operations, Risk, Compliance, Technology **7.** Re-run the data generator and verify 'department' appears in JSON output

Code	`// In User.java, add after the 'tier' field:`
	`@JsonProperty("department")`
	`private String department;`
	
	`public String getDepartment() { return department; }`
	`public void setDepartment(String d) { this.department = d; }`

Verify	✓ User.java compiles without errors ✓ Generated JSON includes "department" for each user ✓ Values are one of the 5 valid departments ✓ Existing tests still pass
If Stuck	💡 *Follow the exact pattern of the 'tier' field* 💡 *Use random.nextInt(5) to index into a String array of departments*
InterviewTalking Point	*Extending shared models is a daily task. This proves you can add to a framework without breaking existing functionality.*

Task 1.2 • Add a Scenario Using Existing Steps ★ **NOVICE** | **Functional (API)** | ⏱ 15 min	
Objective	Write a new Gherkin scenario using only existing step definitions — no Java.
Prerequisites	▸ Phase 0 completed
Steps	**1.** Open features/order-api.feature **2.** After the last scenario, add a new @functional scenario **3.** Write a Given step that loads synthetic data for "order-happy-path" **4.** Write a When step that sends GET to /api/orders **5.** Write a Then step asserting status 200

	6. All steps already exist in ApiSteps.java — you are reusing them **7.** Verify the feature file has no syntax errors in your IDE

Code	`@functional`
	`Scenario: List all orders returns success`
	`  Given synthetic test data is loaded for scenario "order-happy-path"`
	`  When I send a GET request to "/api/orders"`
	`  Then the response status should be 200`

Verify	✓ No Cucumber syntax errors in the .feature file ✓ Every step matches an existing Java step definition ✓ The scenario tests exactly one behavior
If Stuck	💡 *Copy an existing scenario structure and modify the details* 💡 *Step text must match the @Given/@When/@Then annotations EXACTLY*
InterviewTalking Point	*Writing BDD scenarios is a collaboration skill. Product owners read these.*

Task 1.3 • Add a Symbol to the Market Data Watchlist

★★ **INTERMEDIATE** | **Data Generation** | ⏱ 15 min

Objective	Add NVDA to the watchlist and verify it flows through trade generation.
Prerequisites	▸ Phase 0 completed
Steps	**1.** Open data-generator/src/.../market/MarketDataProvider.java **2.** Add "NVDA" to DEFAULT_SYMBOLS in the equities section **3.** Add NVDA to the fallback data map with an approximate price **4.** Open DataGeneratorMain.java and add NVDA to a generateTradeScenario() call **5.** Re-run the data generator **6.** Verify NVDA trades exist in the generated JSON with realistic prices
Verify	✓ NVDA appears in DEFAULT_SYMBOLS ✓ Fallback data includes a reasonable NVDA price ✓ Generated trades for NVDA have realistic prices ✓ Settlement records exist for NVDA trades
If Stuck	💡 *Check NVDA price on finance.yahoo.com for a reasonable fallback* 💡 *Follow the exact pattern of existing entries in the fallback Map*
InterviewTalking Point	*Adding new instruments is routine at OCC when products launch.*

Phase 2: Supported Creation

You will create new files from scratch, but with templates and guidance. By the end of this phase, you can independently create test scenarios across all layers.

Task 2.1 • Create a Payment Generator from Scratch ★★ INTERMEDIATE | **Data Generation** | ⏱ 30-45 min	
Objective	Build a complete new generator class following established patterns.
Prerequisites	▸ Phase 1 completed ▸ Comfortable reading existing generator code
Steps	1. Create shared/src/.../model/Payment.java: paymentId, orderId, amount, method (CREDIT_CARD/WIRE/ACH), status, processedAt 2. Create an enum PaymentMethod.java in the same package 3. Create data-generator/src/.../generators/PaymentGenerator.java 4. Add default and seeded constructors (copy pattern from OrderGenerator) 5. Implement generate(Order order) that creates a Payment linked to the order 6. Make distribution realistic: 60% credit card, 25% ACH, 15% wire 7. Wire into TestDataRegistry: add payments to SyntheticDataSet

Code	`public class PaymentGenerator {`
	`    private final Faker faker;`
	`    private final Random random;`
	
	`    public Payment generate(Order order) {`
	`        Payment p = new Payment();`
	`        p.setOrderId(order.getId());`
	`        p.setAmount(order.getTotalAmount());`
	`        p.setMethod(randomMethod());`
	`        return p;`
	`    }`
	`}`

Verify	✓ Payment.java serializes to valid JSON ✓ Seeded generator produces identical output across runs ✓ Payment amounts match order totals ✓ Generated JSON includes a payments array
If Stuck	💡 *Copy UserGenerator.java as a starting template* 💡 *Seeded pattern: new Faker(new Random(seed))*
InterviewTalking Point	*Every new entity type OCC needs follows this generator pattern.*

Task 2.2 • Write Feature File AND Step Definitions for Payments ★★ INTERMEDIATE | **Functional (API)** | ⏱ 45 min	
Objective	Create both BDD scenarios and the Java step definitions that implement them.
Prerequisites	▸ Task 2.1 completed

Steps	1. Create features/payment-processing.feature with @functional @payment tags 2. Background: load data for "order-happy-path" 3. Scenario 1: Process payment for existing order → POST /api/payments → 201 4. Scenario 2: Payment amount matches order total 5. Scenario 3: Reject payment with invalid order ID → 400 6. Scenario 4: Reject duplicate payment → 409 Conflict 7. Create steps/PaymentSteps.java with step definitions 8. Use TestContext pattern (see TradeSteps.java for reference)
Verify	✓ Feature file has no syntax errors ✓ Every step has a corresponding Java method ✓ PaymentSteps uses SharedTestContext.get() pattern ✓ New steps don't duplicate existing ApiSteps
If Stuck	💡 *TradeSteps.java is your template: constructor gets context, @Given loads data, @When calls API, @Then asserts* 💡 *Reuse 'the response status should be {int}' from ApiSteps*
InterviewTalking Point	*Writing features AND step definitions together is the core SDET workflow.*

Task 2.3 • Create Kafka Event Tests for Payments

★★ INTERMEDIATE | Kafka Events | ⏱ 30-45 min

Objective	Define the Kafka event lifecycle for payments and test it.
Prerequisites	▸ Task 2.2 completed
Steps	1. Define lifecycle: PAYMENT_INITIATED → PAYMENT_AUTHORIZED → PAYMENT_SETTLED (or PAYMENT_DECLINED) 2. Create features/payment-events.feature with @kafka @payment tags 3. Background: Kafka consumers on "payment.events" topic 4. Scenario 1: Payment produces PAYMENT_INITIATED within 10 seconds 5. Scenario 2: Successful payment events in order: INITIATED → AUTHORIZED → SETTLED 6. Scenario 3: Declined payment produces PAYMENT_DECLINED 7. Scenario 4: No PAYMENT_SETTLED for declined payments (negative test) 8. All steps reuse existing KafkaSteps — no new Java code needed
Verify	✓ All scenarios use existing Kafka step definitions ✓ Event types follow a realistic payment flow ✓ Negative test uses 'no event should appear' step
If Stuck	💡 *Review trade-kafka-events.feature for exact step text patterns* 💡 *The 'within X seconds' and 'no event' steps already exist*
InterviewTalking Point	*Kafka event testing validates microservice communication. At OCC, every clearing action produces downstream events.*

Task 2.4 • Write Database Assertions for Payment Records

★★ **INTERMEDIATE** | **Database** | ⏱ 30 min

Objective

Create database tests that verify payment data persists correctly.

Prerequisites

▸ Task 2.2 completed

Steps

1. Add a payments table to docker/init-db.sql: payment_id, order_id, amount, method, status, processed_at
2. Create features/payment-database.feature with @database @payment tags
3. Background: load data and establish database connection
4. Scenario 1: Payment record exists after API submission
5. Scenario 2: Payment amount matches the order total in DB
6. Scenario 3: Payment status updates from PENDING to COMPLETED
7. Add step definitions in PaymentSteps.java using DatabaseHelper.queryRow()

Code

```
-- Add to docker/init-db.sql:
CREATE TABLE IF NOT EXISTS payments (
    payment_id    VARCHAR(64) PRIMARY KEY,
    order_id      VARCHAR(64) NOT NULL REFERENCES orders(id),
    amount        DECIMAL(12,2) NOT NULL,
    method        VARCHAR(32) NOT NULL,
    status        VARCHAR(32) DEFAULT 'PENDING',
    processed_at  TIMESTAMP
);
```

Verify

✓ SQL table definition compiles (test with psql if Docker is running)
✓ DB assertions use parameterized queries (not string concatenation)
✓ Payment amount in DB matches order total

If Stuck

💡 *Look at DatabaseSteps.java for the queryRow() pattern*
💡 *Always use parameterized SQL: WHERE payment_id = ? not WHERE payment_id = '" + id + "'"*

InterviewTalking Point

Database integrity testing catches data corruption before it reaches production reconciliation.

Phase 3: Independent Creation

Less hand-holding. You are given objectives and expected outcomes, but the approach is yours. This is the level expected of mid-senior SDETs.

Task 3.1 • Build an Options Contract Generator with OCC Symbology ★★★ ADVANCED | Data Generation | ⏱ 45-60 min	
Objective	Create a generator for options contracts using the real OCC symbol format: AAPL240315C00150000.
Prerequisites	▸ Phase 2 completed ▸ Understand: call = right to buy, put = right to sell
Steps	1. Create model/trade/OptionContract.java: symbol, underlying, expirationDate, optionType (CALL/PUT), strikePrice, premium, contractSize (100) 2. Create generators/trade/OptionContractGenerator.java 3. Use MarketDataProvider to fetch real underlying price for strike derivation 4. Generate strikes around market price: ITM (-10% to 0%), ATM (0%), OTM (0% to +10%) 5. Format OCC symbols: AAPL + YYMMDD + C/P + strike*1000 padded to 8 digits 6. Generate realistic premiums: ITM options cost more than OTM options 7. Wire into TestDataRegistry with generateOptionsScenario()
Verify	✓ OCC symbol parses: AAPL240315C00150000 → AAPL, 2024-03-15, CALL, $150.00 ✓ Strike prices cluster around the real market price ✓ ITM premiums > ATM premiums > OTM premiums ✓ Contract size defaults to 100 shares
If Stuck	💡 *OCC symbol format: 6-char underlying (padded) + YYMMDD + C or P + 8-digit strike (price * 1000)* 💡 *Use BigDecimal for all prices — never double for financial calculations*
InterviewTalking Point	*OCC clears ALL US-listed options. Knowing their symbology format demonstrates domain depth that generic automation skills don't.*

Task 3.2 • Kafka Event Chain for Options Clearing (Novation) ★★★ ADVANCED | Kafka Events | ⏱ 45-60 min	
Objective	Define and test the clearing event chain including novation — OCC's core function.
Prerequisites	▸ Task 3.1 completed
Steps	1. Define the event sequence: TRADE_RECEIVED → TRADE_VALIDATED → NOVATION_COMPLETE → MARGIN_CALCULATED → TRADE_CLEARED 2. Create features/options-clearing-events.feature with @kafka @options tags 3. Scenario: Submit options trade → verify TRADE_RECEIVED within 10s 4. Scenario: Verify full event ordering on clearing.events topic 5. Scenario: Margin failure produces MARGIN_CALL_ISSUED event

	6. Scenario: Cancelled trade does NOT produce NOVATION_COMPLETE (negative) 7. Create JSON schema: contract-tests/src/.../resources/schemas/clearing-event.schema.json 8. Write schema validation test in contract-tests
Verify	✓ Events follow correct lifecycle ordering ✓ NOVATION_COMPLETE payload includes both counterparties AND OCC as central counterparty ✓ MARGIN_CALCULATED includes initialMargin and maintenanceMargin ✓ JSON schema validates all event types
If Stuck	💡 *Novation means OCC becomes buyer to every seller and seller to every buyer* 💡 *Study the existing order-event.schema.json for JSON Schema structure*
InterviewTalking Point	*Novation is OCC's core function. Demonstrating you understand this clearing flow is a major differentiator.*

Task 3.3 • Database Tests for Position Management

★★★ ADVANCED | **Database** | ⏱ 45 min

Objective	Write tests verifying position tracking and settlement integrity after clearing.
Prerequisites	▸ Phase 2 database task completed
Steps	1. Add positions table to init-db.sql: account_id, symbol, quantity, avg_price, market_value, unrealized_pnl 2. Create features/clearing-database.feature with @database @options tags 3. Scenario 1: BUY trade creates a long position in positions table 4. Scenario 2: Round-trip (BUY then SELL) leaves position quantity at 0 5. Scenario 3: Settlement table has T+1 date for equity options 6. Scenario 4: clearing_house = 'OCC' for all options settlements 7. Create steps/PositionSteps.java using DatabaseHelper
Verify	✓ Position quantity matches cumulative trades ✓ Average price = (qty1*price1 + qty2*price2)/(qty1+qty2) ✓ Settlement dates skip weekends ✓ FK constraint: settlement.trade_id references trades.trade_id
If Stuck	💡 *Position quantity: BUY adds, SELL subtracts* 💡 *Use BigDecimal for avg price calculation, never floating point*
InterviewTalking Point	*Position management and reconciliation is core to clearing operations.*

Task 3.4 • Pact Contract Tests for Clearing API

★★★ ADVANCED | **Contract Testing** | ⏱ 30-45 min

Objective	Write consumer-driven contract tests for a clearing service API.

Prerequisites	▸ Read existing OrderApiContractTest.java
Steps	1. Create contract-tests/src/.../api/ClearingApiContractTest.java 2. Contract 1: POST /api/clearing/submit → 202 (clearingId, status: ACCEPTED) 3. Contract 2: GET /api/clearing/{id} → 200 (full clearing record) 4. Contract 3: GET /api/clearing/{id}/margin → margin requirements 5. Contract 4: GET /api/clearing/notfound → 404 6. Use PactDslJsonBody with type matching for amounts 7. Run tests and verify Pact files in target/pacts/
Verify	✓ Pact files generated in target/pacts/ ✓ Type matching used for numeric fields ✓ Each interaction has a meaningful provider state ✓ Both success and error contracts exist
If Stuck	💡 *Type matching: .decimalType("amount") instead of .decimalValue("amount", 100.00)* 💡 *Provider states describe preconditions: 'trade XYZ is pending clearing'*
InterviewTalking Point	*Contract testing prevents breaking changes between microservices in OCC's distributed clearing system.*

Task 3.5 • JMeter Load Test for Clearing Throughput

★★★ ADVANCED | Performance | ⏱ 30-45 min

Objective	Create a JMeter test plan simulating peak clearing volume.
Prerequisites	▸ JMeter installed or Docker available
Steps	1. Create perf-tests/test-plans/clearing-load-test.jmx 2. POST /api/clearing/submit with parameterized trade data from CSV 3. Configure: 100 threads, 60s ramp-up, 300s duration 4. Add JSON extractor to capture clearingId from response 5. Add follow-up GET /api/clearing/{clearingId} to simulate polling 6. Set thresholds: avg < 200ms, p99 < 1000ms, error rate < 1% 7. Add JMeter CSV generation to TestDataRegistry.persistToDisk()
Verify	✓ JMeter plan runs in CLI mode without errors ✓ CSV data consumed correctly ✓ Threshold script fails when limits breached ✓ Report HTML generated with per-endpoint breakdown
If Stuck	💡 *Use ${__P(param,default)} for CLI parameters in JMX* 💡 *Copy the structure of order-load-test.jmx as a template*
InterviewTalking Point	*OCC processes over 10 billion contracts annually. Performance testing at scale is essential.*

Phase 4: Expert — Architecture & Leadership

These tasks go beyond individual test writing. They require architectural thinking, cross-layer integration, and the ability to design systems that other team members will use. This is Director/Lead-level work.

Task 4.1 • End-to-End Clearing Lifecycle Test

★★★★ EXPERT | **Full Stack** | ⏱ 60 min

Objective	Create a single scenario that exercises ALL framework layers in one flow.
Prerequisites	▸ Phase 3 completed
Steps	1. Create features/e2e-clearing-lifecycle.feature with @e2e @options @kafka @database 2. Single scenario flowing through every layer: 3. → Load synthetic options trade data (real AAPL market price) 4. → Submit trade via API → verify 201 5. → Verify TRADE_RECEIVED Kafka event on clearing.events 6. → Verify NOVATION_COMPLETE with OCC as counterparty 7. → Query DB: trades table has correct record 8. → Query DB: settlements table has T+1 and status CLEARING 9. → Query DB: positions table updated 10. → Verify SETTLEMENT_COMPLETED Kafka event 11. Implement any missing step definitions 12. Run with: mvn test -Dcucumber.filter.tags="@e2e"
Verify	✓ Single scenario passes touching API, Kafka, and DB ✓ Data is consistent across all layers (same tradeId, same amounts) ✓ Market data matches the real Yahoo Finance snapshot ✓ @e2e tag runs ONLY this scenario
If Stuck	💡 *This is integration, not new code — stitch existing steps together* 💡 *The hardest part is data consistency: make sure tradeId flows through all layers*
InterviewTalking Point	*At Director level, OCC wants to see you can architect tests that validate an entire distributed system end-to-end.*

Task 4.2 • Design a Test Data Strategy Document

★★★★ EXPERT | **Full Stack** | ⏱ 45-60 min

Objective	Write a 2-page strategy doc explaining the synthetic data approach, suitable for presenting to VP Engineering.
Prerequisites	▸ All previous phases completed
Steps	1. Create a markdown or Word document covering: 2. → Problem: why static test data fails for clearing systems (market prices change daily, settlement dates shift, new instruments launch) 3. → Solution: hybrid approach — real market data + synthetic trade activity

	4. → Architecture: MarketDataProvider → Generators → TestDataRegistry → CI/CD artifacts 5. → Benefits: environment-agnostic, reproducible (seeded), CI/CD pipeline integration 6. → Risk mitigation: fallback data, cache TTL, offline mode for restricted networks 7. → Metrics: data freshness, generation time, coverage by instrument type 8. → Roadmap: AI-generated test scenarios, ML-based trade distribution modeling 9. Keep it executive-friendly: diagrams over code, outcomes over implementation
Verify	✓ Non-technical stakeholder can understand the strategy ✓ Covers both current state and roadmap ✓ Addresses risk (network failures, stale data) ✓ Includes measurable success criteria
If Stuck	💡 *The README architecture diagram is your starting point* 💡 *Focus on why, not how — executives care about risk reduction and velocity*
InterviewTalking Point	*Directors must communicate technical strategy to non-technical leadership. This document is something you could literally present at OCC.*

Task 4.3 • Create a Framework Extension Guide for Your Team

★★★★ EXPERT | **Full Stack** | ⏱ 45 min

Objective	Write internal documentation that enables other SDETs to add new entity types, test layers, and CI pipeline stages without your help.
Prerequisites	▸ Deep understanding of all framework layers
Steps	1. Create a CONTRIBUTING.md in the repo root covering: 2. → How to add a new model (checklist: Java class, JSON annotations, add to SyntheticDataSet) 3. → How to add a new generator (checklist: constructor pattern, seeded variant, wire to registry) 4. → How to add a new feature file (naming convention, tag strategy, Background pattern) 5. → How to add step definitions (TestContext pattern, reuse vs. new steps decision tree) 6. → How to add a Kafka event type (schema, generator, feature, contract test — 4 files) 7. → How to add a database table (init-db.sql, feature, steps — 3 files) 8. → How to add a CI pipeline stage (GitHub Actions job, artifact passing) 9. Include a decision tree: 'I need to test X → which files do I create?'
Verify	✓ A Phase 2 SDET could follow your guide to add a new entity without asking questions ✓ Every checklist is complete (no missing steps that require tribal knowledge) ✓ Decision tree covers at least 5 common scenarios ✓ Guide links to example files in the repo for each pattern
If Stuck	💡 *Test your guide by having someone at Phase 2 level try to follow it*

	💡 *The best documentation answers questions before they are asked*
InterviewTalking Point	*The job description calls for 'reusable asset libraries and knowledge management artifacts.' This IS that deliverable.*

Completion Milestones

✓ **Phase 0 Complete**

You can navigate the codebase, read feature files, trace steps to Java code, and generate test data. You understand the architecture.

✓ **Phase 1 Complete**

You can safely extend existing code: add fields, write scenarios using existing steps, and modify configuration without breaking things.

✓ **Phase 2 Complete**

You can create new generators, feature files, step definitions, Kafka event tests, and database assertions independently with guidance.

✓ **Phase 3 Complete**

You can design and implement test solutions across all framework layers for complex domain-specific scenarios without hand-holding.

✓ **Phase 4 Complete**

You can architect end-to-end test strategies, create documentation that enables the team, and present technical strategy to leadership.

Answer Key — Solution Reference

Use this section AFTER attempting each task. The purpose is not to copy-paste, but to compare your solution against a reference and identify what you may have missed. If your approach differs but passes all verification checkpoints, your solution is equally valid.

⚠ Team leads: consider distributing this section separately so that learners attempt tasks independently first.

Phase 0 Solutions

Task 0.1: Clone, Build, and Explore — SOLUTION	
5 Modules	1. shared — Domain models (User, Order, Trade), config, JSON utilities 2. data-generator — Synthetic data creation using Datafaker + Yahoo Finance 3. functional-tests — Cucumber BDD tests (API, Kafka, database) 4. contract-tests — Pact API contracts + JSON Schema validation 5. perf-tests — JMeter load tests + threshold validation scripts

@JsonProperty	`@JsonProperty("fieldName") maps a Java field to a specific key in JSON.`
	Without it, Jackson uses the Java field name directly.
	Example: @JsonProperty("firstName") private String firstName;
	`  Serializes to: { "firstName": "Chris" }`
	`  Without annotation, a field named 'fName' would serialize as 'fName' not 'firstName'`

DependencyDiagram	shared (models + config) ↓ data-generator (uses shared models to create test data) ↓ functional-tests (uses data-generator output + shared models) ↓ contract-tests (uses shared models for schema validation) ↓ perf-tests (uses generated CSV data from data-generator)

Task 0.2: Read and Understand a Feature File — SOLUTION	

@smoke vs@negative	`@smoke — Quick health-check scenarios that verify core functionality works.`
	`  Run frequently (every deploy, every PR). Should be fast and reliable.`
	`  Example: 'Health check returns 200' — if this fails, nothing else matters.`
	
	`@negative — Scenarios that verify the system correctly REJECTS bad input.`
	`  Example: 'Reject trade with zero quantity' — tests error handling.`
	`  These run in full regression, not necessarily on every quick check.`

WhyBackground?	`Background runs before EVERY scenario in the file. It avoids duplicating`
	common setup steps. Without it, every scenario would need:
	`  Given synthetic test data is loaded for scenario "order-happy-path"`
	
	With Background, you write it once. If you need different data for a specific scenario, override it with a Given step inside that scenario.

Task 0.3: Trace a Step from Feature to Java Code — SOLUTION	
TracingPath	`Feature file step: 'Then the response status should be 201'` ↓ Cucumber scans for @Then("the response status should be {int}")

	↓
	Found in ApiSteps.java:
	`@Then("the response status should be {int}")`
	`public void verifyResponseStatus(int expectedStatus) {`
	`    Response response = context.getLastApiResponse();`
	`    assertThat(response.getStatusCode()).isEqualTo(expectedStatus);`
	`}`
	
	{int} in the annotation becomes the 'int expectedStatus' parameter.
	201 from the feature file is passed as the argument value.

TestContextPurpose	Cucumber creates a NEW instance of each step definition class per scenario. ApiSteps, KafkaSteps, DatabaseSteps are all separate objects. They need to share data (e.g., the API response from a @When step needs to be readable in a @Then step in a different class). SharedTestContext uses ThreadLocal to hold a single TestContext that all step def classes can access. Each class calls: `this.context = SharedTestContext.get();` in its constructor.

Phase 1 Solutions

Task 1.1: Add a New Field to the User Model — SOLUTION

User.javaChanges

```
// Add after the existing 'tier' field:

  @JsonProperty("department")
  private String department;

  public String getDepartment() { return department; }
  public void setDepartment(String department) {
      this.department = department;
  }
```

UserGeneratorChanges

```
// Add inside the generate() method, after setTier():
  user.setDepartment(randomDepartment());

// Add this private method:
  private String randomDepartment() {
      String[] depts = {"Trading", "Operations",
          "Risk", "Compliance", "Technology"};
      return depts[random.nextInt(depts.length)];
  }
```

Key Point

You added a field and generator without modifying any test code.
Existing tests still pass because @JsonIgnoreProperties(ignoreUnknown=true)
on the model classes means extra fields are simply ignored by consumers
that don't know about them. This is backward-compatible extension.

Task 1.2: Add a Scenario Using Existing Steps — SOLUTION

CompleteScenario

```
@functional
Scenario: List all orders returns success
  Given synthetic test data is loaded for scenario "order-happy-path"
  When I send a GET request to "/api/orders"
  Then the response status should be 200
```

Key Point

Every step reuses existing definitions from CommonSteps and ApiSteps.
You wrote zero Java code. This is the power of BDD: once steps exist,
non-developers can compose new test scenarios from them.

Note: you MUST include the Given step even though Background has one,
because your scenario might run independently with a different tag filter.

Task 1.3: Add a Symbol to the Market Data Watchlist — SOLUTION

MarketDataProvider

```
// In DEFAULT_SYMBOLS, add to the equities section:
  "AAPL", "MSFT", "GOOGL", "AMZN", "TSLA", "NVDA",

// In getFallbackSnapshot(), add to the map:
  Map.entry("NVDA", new String[]{"135.00", "NMS", "NVIDIA Corporation"}),
```

DataGeneratorMain

```
// In generateDefaultProfile(), add NVDA to an existing call:
  registry.generateTradeScenario("trade-happy-path",
      List.of("AAPL", "BTC-USD", "NVDA"), 2);
```

Key Point

Adding a symbol required changes in TWO places: the provider (with fallback)
and the scenario definition. The generator, settlement, and Kafka event code
all worked automatically because they are symbol-agnostic. This is the value
of the layered architecture.

Phase 2 Solutions

Task 2.1: Create a Payment Generator from Scratch — SOLUTION

Payment.java

```java
package com.enterprise.testing.shared.model;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;
import java.math.BigDecimal;
import java.time.Instant;
import java.util.UUID;

@JsonIgnoreProperties(ignoreUnknown = true)
public class Payment {
  @JsonProperty("paymentId")
  private String paymentId;
  @JsonProperty("orderId")
  private String orderId;
  @JsonProperty("amount")
  private BigDecimal amount;
  @JsonProperty("method")
  private PaymentMethod method;
  @JsonProperty("status")
  private String status;
  @JsonProperty("processedAt")
  private Instant processedAt;

  public Payment() {
      this.paymentId = "PAY-" + UUID.randomUUID()
          .toString().substring(0,12).toUpperCase();
      this.status = "PENDING";
  }

  // Getters and setters for all fields...
}
```

PaymentMethod.java

```java
package com.enterprise.testing.shared.model;

public enum PaymentMethod {
    CREDIT_CARD, ACH, WIRE_TRANSFER
}
```

PaymentGenerator.java

```java
package com.enterprise.testing.datagen.generators;

import com.enterprise.testing.shared.model.*;
import net.datafaker.Faker;
import java.time.Instant;
import java.util.Random;

public class PaymentGenerator {
  private final Faker faker;
  private final Random random;

  public PaymentGenerator() {
      this.faker = new Faker();
      this.random = new Random();
  }
  public PaymentGenerator(long seed) {
      this.faker = new Faker(new Random(seed));
      this.random = new Random(seed);
  }

  public Payment generate(Order order) {
      Payment p = new Payment();
      p.setOrderId(order.getId());
      p.setAmount(order.getTotalAmount());
      p.setMethod(randomMethod());
      p.setStatus("COMPLETED");
      p.setProcessedAt(Instant.now());
      return p;
  }

  private PaymentMethod randomMethod() {
      int roll = random.nextInt(100);
      if (roll < 60) return PaymentMethod.CREDIT_CARD;
      if (roll < 85) return PaymentMethod.ACH;
      return PaymentMethod.WIRE_TRANSFER;
  }
}
```

Key Point

The seeded constructor is critical for reproducibility. Two runs with the same seed produce identical data. This matters in CI where flaky tests caused by random data waste everyone's time.

Using BigDecimal (not double) for amounts prevents floating-point errors. In financial systems, 0.1 + 0.2 != 0.3 with doubles. OCC would reject this.

Task 2.2: Feature File and Step Definitions for Payments — SOLUTION

payment-processing.feature

```
@functional @payment
Feature: Payment Processing

  Background:
    Given synthetic test data is loaded for scenario "order-happy-path"

  @smoke
  Scenario: Process payment for existing order
    Given a generated payment for the first order
    When I submit the payment to "/api/payments"
    Then the response status should be 201
    And the response body should contain field "paymentId"

  Scenario: Payment amount matches order total
    Given a generated payment for the first order
    Then the payment amount should equal the order total

  @negative
  Scenario: Reject payment with invalid order ID
    When I submit a POST to "/api/payments" with body:
      """
      {"orderId": "INVALID", "amount": 99.99,
       "method": "CREDIT_CARD"}
      """
    Then the response status should be 400

  @negative
  Scenario: Reject duplicate payment for same order
    Given a payment has already been submitted for the first order
    When I submit the same payment again
    Then the response status should be 409
```

Key Point

The @smoke scenario tests the happy path. The two @negative scenarios test error handling. This 3:1 ratio (1 happy, 2 negative) is typical.

Note that 'the response status should be {int}' is reused from ApiSteps. Only the payment-specific steps (loading payment data, submitting) are new.

Task 2.3: Kafka Event Tests for Payments — SOLUTION

payment-events.feature

```
@kafka @payment
Feature: Payment Kafka Events

  Background:
    Given synthetic test data is loaded for scenario "order-happy-path"
    And Kafka consumers are listening on topics:
      | payment.events |

  Scenario: Payment produces PAYMENT_INITIATED event
    Given a payment has been submitted for the first order
    Then within 10 seconds a Kafka event should appear on "payment.events" with:
      | field     | expected           |
      | eventType | PAYMENT_INITIATED  |

  Scenario: Successful payment event ordering
    Given a payment has been submitted for the first order
    Then the Kafka events on "payment.events" should appear in order:
      | PAYMENT_INITIATED  |
      | PAYMENT_AUTHORIZED |
      | PAYMENT_SETTLED    |

  @negative
  Scenario: No PAYMENT_SETTLED for declined payments
    Given a declined payment exists
    Then within 5 seconds a Kafka event should appear on "payment.events" with:
      | field     | expected          |
      | eventType | PAYMENT_DECLINED  |
    And no Kafka event should appear on "payment.events" within 5 seconds
```

Key Point

All step definitions are reused from KafkaSteps.java. Zero new Java code.
This is the payoff of well-designed, generic step definitions.

The negative test verifies absence: 'no Kafka event should appear' uses a timeout-based assertion that waits 5 seconds and fails if an event arrives. This pattern is critical for testing that cancelled/rejected flows do NOT trigger downstream processing.

Task 2.4: Database Assertions for Payment Records — SOLUTION

SQL Table

```
CREATE TABLE IF NOT EXISTS payments (
    payment_id    VARCHAR(64) PRIMARY KEY,
    order_id      VARCHAR(64) NOT NULL REFERENCES orders(id),
    amount        DECIMAL(12,2) NOT NULL,
    method        VARCHAR(32) NOT NULL,
    status        VARCHAR(32) DEFAULT 'PENDING',
    processed_at  TIMESTAMP
);
```

Key Point

The REFERENCES orders(id) creates a foreign key constraint.
This means: you cannot insert a payment for an order that does not exist.
The database enforces data integrity that your tests should also verify.

Always use parameterized queries in step definitions:

```
  db.queryRow("SELECT * FROM payments WHERE payment_id = ?", id)
```

NEVER concatenate strings:

```
  db.queryRow("SELECT * FROM payments WHERE payment_id = '" + id + "'")
```

The second form is vulnerable to SQL injection, even in tests.

Phase 3 Solutions

Task 3.1: Options Contract Generator with OCC Symbology — SOLUTION

OCC SymbolFormat

```
The OCC symbol format is: AAPL  240315C00150000
                          ^^^^  ^^^^^^ ^ ^^^^^^^^
                          |     |      | |
  Underlying (6 chars, right-padded with spaces)
  Expiration (YYMMDD)
  Type (C = Call, P = Put)
  Strike price x 1000, zero-padded to 8 digits
```

Examples:

```
  AAPL  240315C00150000  = AAPL Call, expires 2024-03-15, strike $150.00
  TSLA  250620P00200000  = TSLA Put, expires 2025-06-20, strike $200.00
  SPY   260116C00600000  = SPY Call, expires 2026-01-16, strike $600.00
```

SymbolGenerationCode

```
  public String formatOccSymbol(String underlying,
      LocalDate expiration, String type, BigDecimal strike) {
    String paddedUnderlying = String.format("%-6s", underlying);
    String expStr = expiration.format(
        DateTimeFormatter.ofPattern("yyMMdd"));
    long strikeInt = strike.multiply(
        BigDecimal.valueOf(1000)).longValue();
    String strikeStr = String.format("%08d", strikeInt);
    return paddedUnderlying + expStr + type + strikeStr;
  }
```

Strike PriceDistribution

```
  // Generate strikes around current market price:
  BigDecimal marketPrice = snapshot.getPrice();
  double offset = (random.nextGaussian() * 0.05); // +/- 5%
  BigDecimal strike = marketPrice
      .multiply(BigDecimal.valueOf(1 + offset))
      .setScale(0, RoundingMode.HALF_UP);  // Round to whole $

  // ITM/ATM/OTM classification:
  // For calls:  strike < market = ITM, strike = market = ATM, strike > market =
OTM
  // For puts:   strike > market = ITM, strike = market = ATM, strike < market =
OTM
```

Key Point

BigDecimal.valueOf(1000) for strike encoding, not 1000.0. Floating point would give you 150000.00000001 instead of 150000. In financial systems, one incorrect digit can mean millions of dollars in mismatched positions.

Task 3.2: Kafka Event Chain for Options Clearing (Novation) — SOLUTION

What isNovation?

`When Trader A sells an option to Trader B through an exchange,`
OCC 'novates' the trade: it becomes the buyer to every seller and the seller to every buyer. After novation:

```
  BEFORE:  Trader A  <-->  Trader B
  AFTER:   Trader A  <-->  OCC  <-->  Trader B
```

This eliminates counterparty risk. If Trader B defaults, OCC still honors the obligation to Trader A. This is OCC's core function.

Event Chain

```
  TRADE_RECEIVED       → Trade enters clearing system
  TRADE_VALIDATED      → Symbol, quantity, price verified
  NOVATION_COMPLETE    → OCC inserted as central counterparty
  MARGIN_CALCULATED    → Initial + maintenance margin computed
  TRADE_CLEARED        → Trade fully processed
```

Failure path:

```
  TRADE_RECEIVED → TRADE_VALIDATED → MARGIN_CALCULATED
      → MARGIN_CALL_ISSUED (insufficient margin)
```

NOVATIONPayload

```
  {
    "eventType": "NOVATION_COMPLETE",
    "payload": {
      "tradeId": "TRD-ABC123",
      "originalBuyer": "CITADEL-001",
      "originalSeller": "ACCT-12345",
      "centralCounterparty": "OCC",
      "novatedBuyLeg": "OCC -> ACCT-12345",
      "novatedSellLeg": "CITADEL-001 -> OCC"
    }
  }
```

Key Point

The NOVATION_COMPLETE event must include BOTH original parties AND OCC as the central counterparty. This is what the interviewer will check: do you understand that OCC doesn't just route trades, it BECOMES a party to every trade it clears.

Task 3.3: Database Tests for Position Management — SOLUTION

PositionsTable SQL

```
CREATE TABLE IF NOT EXISTS positions (
    account_id     VARCHAR(64) NOT NULL,
    symbol         VARCHAR(32) NOT NULL,
    quantity       INTEGER NOT NULL DEFAULT 0,
    avg_price      DECIMAL(18,8) NOT NULL DEFAULT 0,
    market_value   DECIMAL(18,2) NOT NULL DEFAULT 0,
    unrealized_pnl DECIMAL(18,2) NOT NULL DEFAULT 0,
    updated_at     TIMESTAMP DEFAULT NOW(),
    PRIMARY KEY (account_id, symbol)
);
```

Average PriceCalculation

```
When adding to a position, average price is weighted:

  existing: 50 shares @ $150.00
  new buy:  30 shares @ $155.00

  new avg = (50*150 + 30*155) / (50+30)
          = (7500 + 4650) / 80
          = $151.875

  // In Java:
  BigDecimal newAvg = existingQty.multiply(existingAvg)
      .add(newQty.multiply(newPrice))
      .divide(existingQty.add(newQty), 8, RoundingMode.HALF_UP);
```

Key Point

The PRIMARY KEY is composite: (account_id, symbol). One account can hold many symbols, and many accounts can hold the same symbol, but each account-symbol pair is unique. This is standard position modeling.

Task 3.4: Pact Contract Tests for Clearing API — SOLUTION

SubmitTradeContract

```
  @Pact(consumer = "clearing-client")
  public V4Pact submitTradeInteraction(PactDslWithProvider builder) {
      return builder
          .given("clearing system is available")
          .uponReceiving("a request to submit trade for clearing")
          .path("/api/clearing/submit")
          .method("POST")
          .body(new PactDslJsonBody()
              .stringType("tradeId")
              .stringType("symbol")
```

```
                .stringValue("side", "BUY")
                .integerType("quantity")
                .decimalType("price"))
            .willRespondWith()
            .status(202)
            .body(new PactDslJsonBody()
                .stringType("clearingId")
                .stringValue("status", "ACCEPTED"))
            .toPact(V4Pact.class);
    }
```

Key Point	Use stringType/decimalType (type matching) for values that change. Use stringValue (exact matching) for values that must be constant. 'status: ACCEPTED' is exact because the consumer depends on this value. 'clearingId' is type-matched because the actual ID doesn't matter, only that it exists and is a string. Provider states like 'clearing system is available' tell the provider what test data to set up before verifying the contract.

Phase 4 Solutions

Task 4.1: End-to-End Clearing Lifecycle Test — SOLUTION

CompleteFeature

```
@e2e @options @kafka @database
Feature: End-to-End Options Clearing Lifecycle

  Scenario: Full clearing flow from trade to settlement
    Given trade test data is loaded for scenario "trade-happy-path"
    And Kafka consumers are listening on topics:
      | trade.events      |
      | settlement.events |
    And a database connection is established

    # API Layer
    Given a generated trade for symbol "AAPL"
    When I submit the trade to "/api/trades"
    Then the response status should be 201

    # Kafka Layer - Trade
    Then within 10 seconds a Kafka event should appear on "trade.events" with:
      | field     | expected       |
      | eventType | TRADE_EXECUTED |

    # Kafka Layer - Settlement
    Then within 10 seconds a Kafka event should appear on "settlement.events"
with:
      | field     | expected           |
      | eventType | SETTLEMENT_CREATED |

    # Database Layer - Trade
    When I query the trades table for the submitted trade ID
    Then the database record should have:
      | column | expected |
      | symbol | AAPL     |
      | status | EXECUTED |

    # Database Layer - Settlement
    When I query the settlements table for the submitted trade ID
    Then the settlement_date should be after the trade_date
    And the clearing_house column should not be null
```

Key Point

This single scenario touches API, Kafka, and database layers. The tradeId flows through all assertions, ensuring end-to-end data consistency.

Comments (# API Layer, # Kafka Layer) make the scenario readable to stakeholders who need to understand the full clearing flow.

The @e2e tag ensures this only runs when explicitly requested:

```
  mvn test -Dcucumber.filter.tags="@e2e"
```

You do NOT want this in your quick @smoke suite — it requires all infra.

Task 4.2: Test Data Strategy Document — SOLUTION

Outline

Your 2-page strategy document should cover:

1. PROBLEM: Static test data fails for clearing systems because:

```
    • Market prices change daily (yesterday's test prices are stale)
    • Settlement dates shift (T+1 means different dates each day)
    • New instruments launch (can't predict future symbols)
    • Environment parity (dev/staging/prod have different data)
```

	2. SOLUTION: Hybrid approach (real market data + synthetic activity) • `Diagram: Yahoo Finance → MarketDataProvider → TradeGenerator → CI artifacts` 3. BENEFITS: Environment-agnostic, reproducible, pipeline-integrated 4. RISK MITIGATION: Fallback data, cache TTL, offline mode 5. METRICS: Generation time, data freshness, coverage by instrument type 6. ROADMAP: AI-generated scenarios, ML-based trade distribution modeling
Key Point	This document is a real deliverable. The OCC job posting asks for 'synthetic test data management' and 'AI/ML infused solutions.' A strategy doc that covers both, written in executive-friendly language with diagrams instead of code, proves Director-level communication.

Task 4.3: Framework Extension Guide (CONTRIBUTING.md) — SOLUTION	

Decision Tree	Include this decision tree in your CONTRIBUTING.md:
	
	'I need to test a new entity type' (e.g., margin calls):
	`1. Create model class in shared/src/.../model/ (1 file)`
	`2. Create generator in data-generator/src/.../generators/ (1 file)`
	`3. Wire into TestDataRegistry (modify 1 file)`
	`4. Create .feature file in functional-tests/.../features/ (1 file)`
	`5. Create step definitions in functional-tests/.../steps/ (1 file)`
	`6. Add DB table to docker/init-db.sql (modify 1 file)`
	`Total: 4 new files, 2 modified files`
	
	'I need to add a Kafka event type':
	`1. Add to TradeKafkaEventGenerator (modify 1 file)`
	`2. Add JSON schema in contract-tests/.../schemas/ (1 file)`
	`3. Add schema test in contract-tests (modify 1 file)`
	`4. Add scenarios to existing or new .feature file (1 file)`
	`Total: 1 new file, 1 modified or new file, 2 modified files`

Key Point	The best CONTRIBUTING.md answers questions before they are asked. Test it by asking a Phase 2 team member to follow it without help. If they get stuck, the guide has a gap. This IS the job at Director level: building systems that enable your team to be independent.

SECTION II

Test Strategy, Execution & Coverage

Architecture • Quality Gates • Pipeline Mapping • Coverage Matrix • Infrastructure

This section defines what we test, when we test it, and how we run it.

Architectural Test Plan

This section defines the test architecture: which tests run at which quality gate, how they map to CI/CD pipeline stages, and the tag taxonomy that controls execution. Every feature file in the framework is classified here so the team understands what runs where and why.

Quality Gate Model

Every test in the framework belongs to exactly one quality gate. Gates are ordered by execution speed and scope. A failure at any gate blocks promotion to the next stage.

Gate	Tag	When It Runs	What It Tests	Target Duration
Gate 0Build	`@build`	Every commit,pre-merge	Compilation, unit tests,static analysis	< 2 min
Gate 1Smoke	`@smoke`	Post-deploy toany environment	Core APIs alive, criticalhappy paths work	< 5 min
Gate 2Functional	`@functional`	Post-deploy toDev/QA	API contracts, businessrules, CRUD operations	< 15 min
Gate 3Integration	`@integration`	Post-deploy toStaging	Kafka events, DB integrity,cross-service flows	< 30 min
Gate 4Regression	`@regression`	Nightly orpre-release	Full scenario coverage,edge cases, negative tests	< 60 min
Gate 5Performance	`@performance`	Weekly orpre-release	Load tests, throughput,latency thresholds	< 30 min
Gate 6E2E	`@e2e`	Pre-releaseonly	Full clearing lifecycle,all layers combined	< 15 min

Tag Taxonomy

Tags serve two purposes: quality gate classification (when it runs) and domain classification (what it tests). Every scenario should have at least one of each.

Quality Gate Tags (when)		
Tag	**Purpose**	**Example Usage**
`@smoke`	Fastest critical-path checks. Must pass in < 5 min.	`-Dcucumber.filter.tags="@smoke"`
`@functional`	Core business logic and API behavior.	`-Dcucumber.filter.tags="@functional"`
`@integration`	Cross-service: Kafka events + DB + API together.	`-Dcucumber.filter.tags="@integration"`
`@regression`	Full coverage including edge cases and negatives.	`-Dcucumber.filter.tags="@regression"`
`@performance`	Load/stress tests with threshold assertions.	`-Dcucumber.filter.tags="@performance"`
`@e2e`	Full stack end-to-end lifecycle tests.	`-Dcucumber.filter.tags="@e2e"`
`@negative`	Tests that verify rejection/error handling.	`-Dcucumber.filter.tags="@negative"`

Domain Tags (what)		
Tag	**Purpose**	**Example Usage**
`@api`	REST API endpoint tests (HTTP request/response).	`-Dcucumber.filter.tags="@api"`
`@trade`	Trade execution domain (equities, crypto).	`-Dcucumber.filter.tags="@trade"`
`@settlement`	Settlement lifecycle (T+1, clearing, fails).	`-Dcucumber.filter.tags="@settlement"`
`@options`	Options-specific (OCC symbology, novation).	`-Dcucumber.filter.tags="@options"`
`@kafka`	Kafka event production and consumption.	`-Dcucumber.filter.tags="@kafka"`
`@database`	Database persistence and integrity checks.	`-Dcucumber.filter.tags="@database"`
`@payment`	Payment processing domain.	`-Dcucumber.filter.tags="@payment"`
`@contract`	Pact consumer-driven contract tests.	`-Dcucumber.filter.tags="@contract"`

Combining Tags (compound filters)		
Tag	**Purpose**	**Example Usage**
`@smoke and @trade`	Only smoke tests for trading domain.	`-Dcucumber.filter.tags="@smoke and @trade"`
`@functional and not @negative`	Happy-path functional tests only.	`-Dcucumber.filter.tags="@functional and not @negative"`
`@kafka or @database`	All integration layer tests.	`-Dcucumber.filter.tags="@kafka or @database"`
`@regression and @settlement`	Full settlement regression suite.	`-Dcucumber.filter.tags="@regression and @settlement"`

CI/CD Pipeline Mapping

Each quality gate maps to a GitHub Actions pipeline stage. Stages run in sequence; a failure blocks all downstream stages.

Pipeline Stage	Job Name	Command	Tags	Trigger	Catches
1. Build & Install	`build-and-install`	`mvn install -pl shared,data-generator -am -DskipTests`	`@build`	Every push, every PR	Compile errors, missing dependencies
2. Generate Data	`generate-data`	`mvn exec:java -pl data-generator`	`N/A`	After install passes	Data generation errors, Yahoo Finance fallback
3. Smoke	`smoke-tests`	`mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`	`@smoke`	After data generated	Core API health, critical paths dead
4. Functional	`functional-tests`	`mvn test -pl functional-tests -Dcucumber.filter.tags="@functional"`	`@functional`	After smoke passes	Business logic, CRUD, validation rules

5. Contract	`contract-tests`	`mvn test -pl contract-tests`	`@contract`	Parallel with functional	API schema drift, Kafka event schema breaks
6. Integration	`integration-tests`	`mvn test -pl functional-tests -Dcucumber.filter.tags="@kafka or @database"`	`@kafka` `@database`	After functional passes	Kafka events missing, DB integrity violations

7. Performance	`perf-tests`	`jmeter -n -t clearing-load-test.jmx`
		`python check-perf-thresholds.py`
			`@performance`	Nightly / pre-release	Latency regressions, throughput drops

8. E2E	`e2e-tests`	`mvn test -pl functional-tests -Dcucumber.filter.tags="@e2e"`	`@e2e`	Pre-release only	Cross-layer data inconsistency

Test Coverage Matrix — Feature Files by Quality Gate

Every scenario in the framework is mapped below to its quality gate(s) and domain. Use this to identify gaps: if a domain has no @smoke tests, it has no fast feedback loop.

Feature File	Scenario	Quality Gate(s)	Domain
`order-api.feature`	Health check endpoint is available	@smoke	@api
	Create a new order successfully	@smoke @functional	@api
	Retrieve an existing order by ID	@functional @regression	@api
	List orders for a customer	@functional @regression	@api
	Reject order with missing required fields	@functional @negative	@api
	Return 404 for non-existent order	@functional @negative	@api
`trade-execution.feature`	Execute a market buy order at real market price	@smoke @functional	@trade
	Execute a crypto trade using live BTC price	@smoke @functional	@trade
	Retrieve trade by ID	@functional @regression	@trade
	List trades for an account	@functional @regression	@trade
	Reject trade with invalid symbol	@functional @negative	@trade
	Reject trade with zero quantity	@functional @negative	@trade
`settlement-lifecycle.feature`	Executed trade creates a settlement record	@smoke @functional	@settlement
	Settlement date is T+1 for US equities	@functional @regression	@settlement
	Crypto trades settle same day (T+0)	@functional @regression	@settlement
	Round-trip trade produces two settlements	@functional @regression	@settlement
	Failed settlement has a failure reason	@functional @negative	@settlement
`kafka-events.feature`	Order creation produces ORDER_CREATED event	@integration @regression	@kafka
	Order confirmation produces multiple events	@integration @regression	@kafka
	Verify event ordering for order lifecycle	@integration @regression	@kafka
	Cancelled order does not produce notification event	@integration @negative	@kafka
`trade-kafka-events.feature`	Trade execution produces TRADE_EXECUTED event	@integration @regression	@kafka @trade
	Settlement creation produces SETTLEMENT_CREATED event	@integration @regression	@kafka @settlement
	Completed settlement produces full event chain	@integration @regression	@kafka @settlement
	Rejected trade does not produce settlement events	@integration @negative	@kafka @trade

database.feature	Order data persists correctly after creation	@integration @regression	@database
	Order items are stored with correct references	@integration @regression	@database
	Order status update reflects in database	@integration @regression	@database
	Audit log captures order events	@integration @regression	@database
	Cancelled order does not leave orphan records	@integration @negative	@database
trade-database.feature	Trade record persists with correct market data	@integration @regression	@database @trade
	Settlement references the correct trade	@integration @regression	@database @settlement
	Trade audit log captures execution details	@integration @regression	@database @trade

Coverage Summary by Gate

Gate	Count	Scenarios Included	Duration
@smoke	**5**	Health check, order create, trade buy, crypto trade, settlement create	< 5 min
@functional	**17**	All @smoke + CRUD, validation, T+1/T+0, round-trips, rejections	< 15 min
@integration	**16**	Kafka event chains, DB persistence, audit logs, FK integrity	< 30 min
@negative	**10**	Invalid input, zero qty, 404s, orphan records, failed settlements	Included above
@regression	**31**	All functional + integration scenarios combined	< 45 min
@performance	**2**	Clearing throughput, order API load test	< 30 min
@e2e	**1**	Full clearing lifecycle (API → Kafka → DB)	< 15 min

Test Target Environments

The framework does not include an application server. It tests AGAINST a target service. If you run tests without a target, you will see 'Connection refused.' This section explains what to point at and how.

Three Layers of Test Targets	
Layer 1:WireMock(Local Dev + CI)	WireMock is a stub server that returns pre-configured responses. It ships with this framework in docker-compose.yml and runs on port 8080. No real application needed — you define what the API SHOULD return and test against that. *This test method is known as 'service virtualization.'* **Use for: local development, CI pipelines, @smoke tests, @functional tests, contract validation.** **Limitation: does not test real business logic, only that your tests work against a defined contract.**
Layer 2:QA/StagingEnvironment	Real microservices deployed on internal infrastructure. Kafka is real, PostgreSQL is real, the clearing engine is real. `Set via environment variable: export BASE_URL=https://clearing-api.qa.internal` **Use for: @integration tests, @regression suite, @e2e lifecycle.** This is where you find real bugs in real services.
Layer 3:UAT / Sandbox(Pre-Production)	Near-production environment with synthetic market data. Performance tests and load tests run here. **Use for: @performance tests, final @e2e validation before release.** This is where you catch latency regressions and throughput degradation.
Why No PublicClearing API?	OCC, DTCC, NSCC — none of them expose public APIs. Clearing infrastructure is proprietary and only accessible to member firms. This is standard for financial services. The framework is designed so that only the config values change between environments, not the test code.

Configuring the Test Target

All target URLs are configured in FrameworkConfig.java using a three-level override chain: environment variable > system property > framework.properties default.

Setting	Property Key	Env Variable	Default	Purpose
API Base URL	`app.base.url`	`APP_BASE_URL`	`http://localhost:8080`	Where REST API calls go
Kafka Brokers	`kafka.bootstrap.servers`	`KAFKA_BOOTSTRAP_SERVERS`	`localhost:9092`	Kafka event consumption
Database URL	`db.url`	`DB_URL`	`jdbc:postgresql://localhost:5432/testdb`	Database assertions
DB Username	`db.username`	`DB_USERNAME`	`testuser`	Database connection
DB Password	`db.password`	`DB_PASSWORD`	`testpass`	Database connection

Override examples:

```
# Option 1: Environment variable (recommended for CI/CD)
  export APP_BASE_URL=https://clearing-api.qa.internal
  mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"
# Option 2: System property (inline)
  mvn test -pl functional-tests -Dapp.base.url=https://clearing-api.qa.internal
-Dcucumber.filter.tags="@smoke"
# Option 3: Default (localhost:8080 — requires WireMock or local server running)
  mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"
```

Running Tests Locally with WireMock

WireMock acts as a fake API server. It reads stub mappings from docker/wiremock/mappings/ and returns pre-configured responses. This lets you run smoke and functional tests without any real services.

WireMock Setup — Step by Step	

Step 1:Start WireMock	`# Make sure Docker Desktop is running, then from the project root:`
	`  docker compose up -d wiremock`
	
	`# Verify it started:`
	`  curl http://localhost:8080/__admin/mappings`
	`# Should return JSON listing all stub mappings`
	
	`# You should see stubs for: /api/trades, /api/settlements,`
	`# /api/clearing, /api/orders, and more`

Step 2:Run Tests	`# WireMock runs on port 8080, which is the default BASE_URL.`
	`# No extra configuration needed — just run:`
	
	`  mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`
	
	`# Tests will hit WireMock at http://localhost:8080 and get`
	`# pre-configured responses matching each endpoint.`

Step 3:Stop When Done	`  docker compose down wiremock`
	
	`# Or stop all services:`
	`  docker compose down`

What StubsExist?	The framework ships with stub mappings in docker/wiremock/mappings/:
	
	• order-api.json — Stubs for order CRUD: create, get by ID, list, 404
	• trade-settlement-api.json — Stubs for:
	`    - POST /api/trades (success + invalid symbol + zero qty)`
	`    - GET /api/trades/{id} and GET /api/trades?accountId=X`
	`    - GET /api/settlements/trade/{id}`
	`    - POST /api/clearing/submit (accepted)`
	`    - GET /api/clearing/{id} (status + novation details)`
	`    - GET /api/clearing/{id}/margin (margin requirements)`

Adding NewStubs	To add stubs for new endpoints: 1. Create a new .json file in docker/wiremock/mappings/ 2. Follow the existing pattern: request matcher + response body `3. Restart WireMock: docker compose restart wiremock` WireMock docs: https://wiremock.org/docs/stubbing/

Full StackLocal Setup	To run integration tests locally (Kafka + DB + WireMock):
	`docker compose up -d`
	
	This starts: WireMock (8080), Kafka (9092), Postgres (5432), Zookeeper (2181)
	
	Then run the full regression:
	`mvn test -pl functional-tests -Dcucumber.filter.tags="@functional or @kafka or @database"`
	
	`When done: docker compose down`

✓ If 'Connection refused' → WireMock is not running. Start it with: docker compose up -d wiremock

Execution Quick Reference

Every test command requires specific infrastructure to be running. The tables below are grouped by what you need to start BEFORE running the tests. Always run 'mvn install' first if you haven't already.

Step 0: Always Run First (no Docker needed)	
Install shared modules	`mvn install -pl shared,data-generator -am -DskipTests`
Generate test data	`mvn exec:java -pl data-generator -Dexec.mainClass="com.enterprise.testing.datagen.DataGeneratorMain"`

Requires: WireMock Only (API stubs)	
⚡ Start Infra First	**`docker compose up -d wiremock`** *Wait ~5 seconds for WireMock to load mappings. Verify: curl http://localhost:8080/__admin/mappings*
Run smoke tests	`mvn test -pl functional-tests -Dcucumber.filter.tags="@smoke"`
Run functional tests	`mvn test -pl functional-tests -Dcucumber.filter.tags="@functional"`
Run API tests only	`mvn test -pl functional-tests -Dcucumber.filter.tags="@api"`
Run trade domain only	`mvn test -pl functional-tests -Dcucumber.filter.tags="@trade and not @kafka and not @database"`
Run settlement domain only	`mvn test -pl functional-tests -Dcucumber.filter.tags="@settlement and not @kafka and not @database"`
Run negatives only	`mvn test -pl functional-tests -Dcucumber.filter.tags="@negative and not @kafka and not @database"`
Run contract tests	`mvn test -pl contract-tests`
⏹ Stop When Done	`docker compose down`

Requires: WireMock + Kafka (event-driven tests)	
⚡ Start Infra First	**`docker compose up -d wiremock kafka zookeeper`** *Wait ~15 seconds for Kafka to register with Zookeeper. Verify: docker compose logs kafka | tail -5 (look for 'started')*
Run Kafka event tests	`mvn test -pl functional-tests -Dcucumber.filter.tags="@kafka"`

Run trade Kafka events	`mvn test -pl functional-tests -Dcucumber.filter.tags="@kafka and @trade"`
Run settlement Kafka events	`mvn test -pl functional-tests -Dcucumber.filter.tags="@kafka and @settlement"`
⏹️ **Stop When Done**	`docker compose down`

Requires: WireMock + PostgreSQL (database tests)	
⚡ **Start Infra First**	**`docker compose up -d wiremock postgres`** *Wait ~10 seconds for Postgres to initialize. Verify: docker compose exec postgres pg_isready*
Run database tests	`mvn test -pl functional-tests -Dcucumber.filter.tags="@database"`
Run trade DB tests	`mvn test -pl functional-tests -Dcucumber.filter.tags="@database and @trade"`
⏹️ **Stop When Done**	`docker compose down`

Requires: Full Stack — WireMock + Kafka + PostgreSQL (all services)	
⚡ **Start Infra First**	**`docker compose up -d`** *Wait ~20 seconds for all services. Verify: docker compose ps (all should show 'running')*
Run integration tests	`mvn test -pl functional-tests -Dcucumber.filter.tags="@kafka or @database"`
Run full regression	`mvn test -pl functional-tests -Dcucumber.filter.tags="@functional or @kafka or @database"`
Run E2E lifecycle	`mvn test -pl functional-tests -Dcucumber.filter.tags="@e2e"`
Run everything	`mvn test -pl functional-tests`
Run performance tests	`cd perf-tests && jmeter -n -t test-plans/clearing-load-test.jmx -l results.jtl -e -o report/`
⏹️ **Stop When Done**	`docker compose down`

One-Liner Cheat Sheet (full stack, start to finish):

```
# Smoke tests (fastest demo):
  docker compose up -d wiremock && sleep 5 && mvn test -pl functional-tests
-Dcucumber.filter.tags="@smoke"
# Full regression (complete demo):
  docker compose up -d && sleep 20 && mvn test -pl functional-tests
# Cleanup:
  docker compose down -v
```