TRAINING GUIDE FOR EDUMATCHER V0.77.0

Running and Operating EduMatcher

A hands-on and self-paced training guide for EduMatcher.

EduMatcher Training Guide
For EduMatcher Version: v0.77.0
Copyright © Johan Persson
Generated on November 8, 2026
Prepared with LaTeX from Markdown sources for duplex printing. Sources for documentation available at https://github.com/johan162/EduMatcher.

Training Guide

`edumatcher`, Version: v0.77.0

A hands-on self-study programme for EduMatcher

Johan Persson <johan162@gmail.com>

Generated on November 8, 2026

Chapters

1 Training Guide 1

I Foundations 7

2 Installation & Setup 11

3 Configuring & Starting Up 25

4 Setting Up Market-Maker Liquidity 39

5 The First Trade 53

II Trading mechanics 57

6 Amending Orders 61

7 Order Types Deep Dive 67

8 Time-in-Force & Sessions 73

9 Auctions 79

10 Cancelling & Managing Orders 85

III Liquidity, risk and results 89

11 Market Making 93

12 Combo Orders 103

13 Risk Controls 119

14 P&L & Clearing 129

15 Market Data & Drop Copy 143

16 AI Traders & News 149

IV Running a full session 155

17 Statistics & Reporting 159

18 Persistence & Recovery 165

19 Capstone Scenario 169

20 Exchange Observer Processes 175

21 Advanced Admin Operations 181

22 Closing the Day: Settlement (`pm-eod`) 185

V Integration and advanced topics 193

23 Drop-Copy Replay & Recovery Patterns 197

24 Automation with CommandClient & MM Bot Tuning 203

25 RALF Post-Trade Gateway Protocol Training 209

26 CALF Market-Data Gateway Protocol Training 217

27 API Gateway REST/WebSocket Training 227

28 Market Index (`pm-index`) 241

29 ALF TCP Gateway Protocol Training 253

30 BALF TCP Gateway Protocol Training 265

31 IPO Valuation (`pm-valuation`) **277**

Contents

1 Training Guide **1**
- 1.1 How to use this guide . . . 1
- 1.2 Choose your path . . . 3
- 1.3 Training plan . . . 4
- 1.4 After the training . . . 6

I Foundations **7**

2 Installation & Setup **11**
- 2.1 Objective . . . 11
- 2.2 Exercise 0: Read How an Exchange Works . . . 11
- 2.3 Choosing how to install . . . 11
- 2.4 Exercise 1: Containers — the whole system in one command . . . 12
- 2.5 Alternate Routes . . . 15
- 2.6 Exercise 2: Find where your files live . . . 18
- 2.7 Exercise 3: Understand the deployed configuration . . . 19
- 2.8 Exercise 4: Customise your installation . . . 19
- 2.9 Exercise 5: Confirm your entry points . . . 21
- 2.10 Troubleshooting . . . 22
- 2.11 Summary . . . 23
- 2.12 Before You Continue . . . 23
- 2.13 Reflection . . . 23
- 2.14 Further Reading . . . 24

3 Configuring & Starting Up **25**
- 3.1 Objective . . . 25
- 3.2 Prerequisites . . . 25
- 3.3 Background . . . 25
- 3.4 Exercise 1: Create a Minimal Configuration . . . 26
- 3.5 Exercise 2: Generate a Config with pm-config-gen . . . 27
- 3.6 Exercise 3: Validate the Config with pm-cverifier . . . 28
- 3.7 Exercise 4: Practice Fixing Verifier Findings . . . 28
- 3.8 Exercise 5: Compile and Deploy the Config with pm-config-deploy . . . 29

3.9 Exercise 6: Start the Engine . . . 30
3.10 Exercise 7: Start the Scheduler . . . 30
3.11 Exercise 8: Connect a Gateway . . . 31
3.12 Exercise 9: Verify the Setup . . . 31
3.13 Exercise 10: Connect the Operator Console . . . 32
3.14 Exercise 11: Inspect Enriched SYMBOLS Metadata . . . 33
3.15 Exercise 12: Break — and Recognise — the Artifact's Hash Checks . . . 34
3.16 Exercise 13: Weekday / Weekend / Holiday Scheduling . . . 35
3.17 Summary . . . 37
3.18 Reflection . . . 38
3.19 Further Reading . . . 38

4 Setting Up Market-Maker Liquidity 39
4.1 Objective . . . 39
4.2 Prerequisites . . . 39
4.3 Background . . . 39
4.4 Exercise 1: Add MM Gateways to Configuration . . . 40
4.5 Exercise 2: Connect the AAPL Market Maker . . . 43
4.6 Exercise 3: Quote MSFT and TSLA . . . 43
4.7 Exercise 4: Verify Liquidity from the Operator Console . . . 43
4.8 Exercise 5: Inspect Quote State with QLEGS . . . 44
4.9 Exercise 6: Run the Equivalent Bot Workflow (Optional) . . . 44
4.10 Exercise 7: Drive the Bot from a Config File (Optional) . . . 45
4.11 Exercise 8: One Bot, Three Symbols, Three Sets of Terms (Optional) . . . 46
4.12 Returning to the manual-quote state . . . 50
4.13 Summary . . . 51
4.14 Reflection . . . 51
4.15 Further Reading . . . 52

5 The First Trade 53
5.1 Objective . . . 53
5.2 Prerequisites . . . 53
5.3 Exercise 1: Start Clearing . . . 53
5.4 Exercise 2: Buy at Market — Lift the Ask . . . 54
5.5 Exercise 3: Sell at Market — Hit the Bid . . . 54
5.6 Exercise 4: Place a Limit Buy Order . . . 55
5.7 Exercise 5: Place a Limit Sell and Get a Fill . . . 55
5.8 Exercise 6: Check Order State . . . 55
5.9 Exercise 7 (Optional): Observe Automatic Re-Quoting with pm-mm-bot . . . 56
5.10 Key Concepts Learned . . . 56
5.11 Reflection . . . 56
5.12 Further Reading . . . 56

II Trading mechanics 57

6 Amending Orders 61
6.1 Objective . . . 61
6.2 Prerequisites . . . 61

6.3 Exercise 1: Place a Resting Order to Amend . . . 61
6.4 Exercise 2: Amend Quantity Down . . . 62
6.5 Exercise 3: Amend Price . . . 62
6.6 Exercise 4: Amend Both Price and Quantity . . . 62
6.7 Exercise 5: Attempt an Invalid Amendment . . . 62
6.8 Exercise 6: Amend After Partial Fill . . . 63
6.9 Exercise 7: See Queue Priority Change . . . 63
6.10 Key Rules . . . 65
6.11 Reflection . . . 65
6.12 Further Reading . . . 65

7 Order Types Deep Dive 67
7.1 Objective . . . 67
7.2 Prerequisites . . . 67
7.3 Deterministic Trigger Setup . . . 67
7.4 Exercise 1: Stop Order . . . 68
7.5 Exercise 2: Stop-Limit Order . . . 68
7.6 Exercise 3: Fill-or-Kill (FOK) . . . 69
7.7 Exercise 4: Immediate-or-Cancel (IOC) . . . 69
7.8 Exercise 5: Iceberg Order . . . 69
7.9 Exercise 6: Trailing Stop . . . 70
7.10 Order Type Summary . . . 70
7.11 Reflection . . . 71
7.12 Further Reading . . . 71

8 Time-in-Force & Sessions 73
8.1 Objective . . . 73
8.2 Prerequisites . . . 73
8.3 Background . . . 73
8.4 Exercise 1: Observe Session Phases . . . 74
8.5 Freeze/Advance Procedure (Deterministic Session Control) . . . 74
8.6 Exercise 2: DAY Order Expiry . . . 75
8.7 Exercise 3: GTC Order Survives . . . 75
8.8 Exercise 4: ATO Order . . . 76
8.9 Exercise 5: ATC Order . . . 76
8.10 Exercise 6: Rejected TIF . . . 76
8.11 Summary . . . 77
8.12 Reflection . . . 77
8.13 Further Reading . . . 77

9 Auctions 79
9.1 Objective . . . 79
9.2 Prerequisites . . . 79
9.3 Background . . . 79
9.4 Exercise 1: Set Up for an Opening Auction . . . 79
9.5 Exercise 2: Trigger the Auction Uncrossing . . . 80
9.6 Exercise 3: Unfilled Auction Orders . . . 81
9.7 Exercise 4: Closing Auction . . . 81

9.8 Exercise 5: GTC Orders in Auctions . . . 82
9.9 Equilibrium Price Rules . . . 82
9.10 Reflection . . . 82
9.11 Further Reading . . . 83

10 Cancelling & Managing Orders **85**
10.1 Objective . . . 85
10.2 Prerequisites . . . 85
10.3 Exercise 1: Cancel a Resting Order . . . 85
10.4 Exercise 2: Cancel a Partially Filled Order . . . 86
10.5 Exercise 3: Compare STATUS and ORDERS . . . 86
10.6 Exercise 4: Inspect Resting Order Details . . . 86
10.7 Exercise 5: Cancel a Non-Existent Order . . . 87
10.8 Exercise 6: Cancel Another Gateway's Order . . . 87
10.9 Exercise 7: Admin Kill Switch . . . 87
10.10 Exercise 8: Inspect SYMBOLS Metadata from Gateway . . . 87
10.11 Order Lifecycle Summary . . . 88
10.12 Reflection . . . 88
10.13 Further Reading . . . 88

III Liquidity, risk and results **89**

11 Market Making **93**
11.1 Objective . . . 93
11.2 Prerequisites . . . 93
11.3 Exercise 1: Submit a Two-Sided Quote . . . 94
11.4 Exercise 2: Inspect Quote Legs with QLEGS . . . 94
11.5 Exercise 3: Get Filled and Observe Inactivation . . . 95
11.6 Exercise 4: Re-quote After Inactivation . . . 95
11.7 Exercise 5: Replace a Quote Without Cancelling . . . 95
11.8 Exercise 6: Explicit Cancel . . . 96
11.9 Exercise 7: Cancel a Single Leg Directly — and See Why You Shouldn't . . . 96
11.10 Exercise 8: Check Quote Bootstrap State (QBOOT) . . . 97
11.11 Exercise 9: Compare Inactivation Policies . . . 98
11.12 Exercise 10: Disconnect and Bulk-Cancel Behaviour . . . 99
11.13 Key Takeaways . . . 100
11.14 Reflection . . . 100
11.15 Further Reading . . . 100

12 Combo Orders **103**
12.1 Objective . . . 103
12.2 Prerequisites . . . 103
12.3 Background . . . 104
12.4 Exercise 1: Simple Two-Leg Combo . . . 105
12.5 Exercise 2: Atomicity at Entry — One Short Leg Blocks Every Leg . . . 105
12.6 Exercise 3: Leg Risk Returns Once the Combo Rests . . . 106
12.7 Exercise 4: Cancel a Resting Combo . . . 107
12.8 Exercise 5: OCO — One-Cancels-Other . . . 107

12.9 Exercise 6: OCO — Protect an Existing Long Position . . . 107
12.10 Exercise 7: OCO — A Partial Fill Is Not a Completed Choice . . . 108
12.11 Exercise 8: OCO — Breakout Entry and Protective-Exit Variants . . . 109
12.12 Exercise 9: Size a Pairs Trade and Close It . . . 110
12.13 Exercise 10: Three-Leg Basket, Then Scale the Process . . . 110
12.14 Exercise 11: Execution Cost — Depth, MARKET, IOC and FOK Legs . . . 111
12.15 Exercise 12: Conditional Combo Legs Are Not a Synchronized Trigger . . . 112
12.16 Exercise 13: Parent Cancel, Child Failure and Self-Match Prevention . . . 112
12.17 Exercise 14: Validation and TIF — When Should Nothing Be Posted? . . . 114
12.18 Exercise 15: Advanced Operations — Recovery and Strategy Liquidity . . . 115
12.19 When to Use Combos vs OCO . . . 116
12.20 Reflection . . . 116
12.21 Further Reading . . . 117

13 Risk Controls **119**
13.1 Objective . . . 119
13.2 Prerequisites . . . 119
13.3 Background . . . 119
13.4 Exercise 1: Configure Price Collars . . . 120
13.5 Exercise 2: Trigger a Static Collar Rejection . . . 120
13.6 Exercise 3: Configure Circuit Breakers . . . 120
13.7 Exercise 4: Trigger a Circuit Breaker . . . 121
13.8 Exercise 5: Open the Admin Console . . . 122
13.9 Exercise 6: Manual Symbol Halt and Resume . . . 122
13.10 Exercise 7: Exchange-Wide Halt with pm-admin-cli . . . 123
13.11 Exercise 8: Query and Manage with pm-admin-cli . . . 123
13.12 Exercise 9: Exchange-Wide Halt from the Admin Console . . . 123
13.13 Exercise 10: Kill Switch . . . 123
13.14 Exercise 11: Named Risk Levels — One Policy, Many Symbols . . . 124
13.15 Risk Control Summary . . . 125
13.16 Starting over . . . 126
13.17 Reflection . . . 127
13.18 Further Reading . . . 127

14 P&L & Clearing **129**
14.1 Objective . . . 129
14.2 Prerequisites . . . 130
14.3 Background . . . 130
14.4 Part A — Intraday exercises . . . 131
14.5 Interlude — Closing the trading day . . . 135
14.6 Part B — End-of-day exercises . . . 136
14.7 Part C — Ongoing operations . . . 138
14.8 Key Formulas . . . 139
14.9 pm-clearing-cli verb reference . . . 139
14.10 Reflection . . . 141
14.11 Further Reading . . . 141

15 Market Data & Drop Copy **143**

15.1 Objective . . . 143
15.2 Prerequisites . . . 143
15.3 Background . . . 143
15.4 Exercise 1: Open a Live Book Viewer . . . 143
15.5 Exercise 2: Run the Cross-Gateway Order Monitor . . . 144
15.6 Exercise 3: Capture Events with pm-audit . . . 144
15.7 Exercise 4: Confirm the Drop-Copy Feed Is Bound . . . 144
15.8 Exercise 5: Watch the Drop-Copy Feed with `pm-dc-spy` . . . 144
15.9 Exercise 5b: Subscribe to the Raw Socket . . . 145
15.10 Exercise 6: Compare Public Events and Drop-Copy Purpose . . . 146
15.11 Exercise 7: Launch the Market Board . . . 146
15.12 Key Architecture . . . 146
15.13 Reflection . . . 146
15.14 Further Reading . . . 147

16 AI Traders & News **149**
16.1 Objective . . . 149
16.2 Prerequisites . . . 149
16.3 Background . . . 149
16.4 Exercise 1: Deploy an Example with AI Traders . . . 150
16.5 Exercise 2: One AI Trader . . . 151
16.6 Exercise 3: The Swarm . . . 151
16.7 Exercise 4: Price and Value . . . 151
16.8 Exercise 5: An Earnings Surprise . . . 152
16.9 Exercise 6: A Rumour That Turns Out True . . . 152
16.10 Exercise 7: A Rumour That Is Retracted . . . 152
16.11 Exercise 8: A Breakthrough and a Fire . . . 153
16.12 Exercise 9: Stop the Swarm . . . 153
16.13 Classroom Tips . . . 153
16.14 Reflection . . . 154
16.15 Further Reading . . . 154

IV Running a full session **155**

17 Statistics & Reporting **159**
17.1 Objective . . . 159
17.2 Prerequisites . . . 159
17.3 Background . . . 159
17.4 Exercise 1: Start the Statistics Service . . . 159
17.5 Exercise 2: Generate Some Trading Activity . . . 160
17.6 Exercise 3: Query the Daily OHLCV Summary . . . 160
17.7 Exercise 4: Read the VWAP Column . . . 160
17.8 Exercise 5: Query Trade Log . . . 161
17.9 Exercise 6: Multi-Symbol Summary . . . 161
17.10 Exercise 7: Export Data . . . 161
17.11 What Gets Recorded . . . 161
17.12 Summary . . . 162

17.13Reflection . . . 162
17.14Further Reading . . . 162

18 Persistence & Recovery **165**
18.1 Objective . . . 165
18.2 Prerequisites . . . 165
18.3 Background . . . 165
18.4 Exercise 1: Locate the Data Directory . . . 166
18.5 Exercise 2: Create a GTC Order . . . 166
18.6 Exercise 3: Restart the Engine . . . 166
18.7 Exercise 4: DAY Order Survives a Same-Day Restart . . . 167
18.8 Exercise 5: Persist Statistics . . . 167
18.9 Exercise 6: Audit to Disk . . . 167
18.10Summary . . . 168
18.11Reflection . . . 168
18.12Further Reading . . . 168

19 Capstone Scenario **169**
19.1 Objective . . . 169
19.2 Prerequisites . . . 169
19.3 Scenario . . . 169
19.4 Exercise 1: Generate a Fresh Config . . . 170
19.5 Exercise 2: Verify and Deploy . . . 170
19.6 Exercise 3: Start the Exchange Stack . . . 171
19.7 Exercise 4: Connect Gateways . . . 171
19.8 Exercise 5: Open the Market and Provide Liquidity . . . 172
19.9 Exercise 6: Generate Trades and Amendments . . . 172
19.10Exercise 7: Trigger an Operator Action . . . 172
19.11Exercise 8: Inspect P&L, Audit, and Stats . . . 173
19.12Exercise 9: Persistence Check . . . 173
19.13Final Review Questions . . . 173
19.14Further Reading . . . 174

20 Exchange Observer Processes **175**
20.1 Objective . . . 175
20.2 Prerequisites . . . 175
20.3 Background . . . 175
20.4 Exercise 1: Start the Baseline Exchange . . . 176
20.5 Exercise 2: Watch One Book with pm-viewer . . . 176
20.6 Exercise 3: Monitor Orders Across Gateways with pm-orders . . . 176
20.7 Exercise 4: Record the Event Stream with pm-audit . . . 177
20.8 Exercise 5: Build Persistent Statistics with pm-stats . . . 177
20.9 Exercise 6: Launch the Scrolling Ticker . . . 178
20.10Exercise 7: Launch the Multi-Symbol Board . . . 178
20.11Exercise 8: Track Positions and P&L with pm-clearing . . . 178
20.12Exercise 9: Compare the Views . . . 178
20.13Summary . . . 179
20.14Reflection . . . 179

20.15Further Reading . . . 179

21 Advanced Admin Operations 181
21.1 Objective . . . 181
21.2 Prerequisites . . . 181
21.3 Background . . . 181
21.4 Exercise 1: Disconnect a Gateway with KICK . . . 182
21.5 Exercise 2: Cancel One MM Quote with QCANCEL . . . 182
21.6 Exercise 3: Clear One Symbol Book with CANCEL_SYM . . . 182
21.7 Exercise 4: Force Session State for Controlled Drills . . . 183
21.8 Exercise 5: Run an Incident Micro-Runbook . . . 183
21.9 Summary . . . 184
21.10Reflection . . . 184
21.11Further Reading . . . 184

22 Closing the Day: Settlement (`pm-eod`) 185
22.1 Objective . . . 185
22.2 Prerequisites . . . 185
22.3 Background . . . 185
22.4 Exercise 1: Check the starting point . . . 186
22.5 Exercise 2: Clear the backlog . . . 186
22.6 Exercise 3: Ask too early . . . 187
22.7 Exercise 4: Write the roster . . . 187
22.8 Exercise 5: Trade a day . . . 188
22.9 Exercise 6: Close the day and settle it . . . 188
22.10Exercise 7: Read what the traders receive . . . 189
22.11Exercise 8: Run it again . . . 189
22.12Exercise 9: A broken roster . . . 190
22.13Exercise 10: Trading after the close . . . 190
22.14Summary . . . 191
22.15Reflection . . . 191
22.16Further Reading . . . 192

V Integration and advanced topics 193

23 Drop-Copy Replay & Recovery Patterns 197
23.1 Objective . . . 197
23.2 Prerequisites . . . 197
23.3 Background . . . 197
23.4 Exercise 1: Subscribe and Print Sequence Numbers . . . 198
23.5 Exercise 2: Detect a Synthetic Gap . . . 198
23.6 Exercise 3: Validate Current Replay Constraint . . . 200
23.7 Exercise 4: Recovery Workflow with Audit + Stats . . . 201
23.8 Exercise 5: Design a Consumer Gap Policy . . . 201
23.9 Summary . . . 201
23.10Reflection . . . 201
23.11Further Reading . . . 202

24 Automation with CommandClient & MM Bot Tuning **203**
24.1 Objective . . . 203
24.2 Prerequisites . . . 203
24.3 Background . . . 203
24.4 Exercise 1: Run a Minimal CommandClient Session . . . 204
24.5 Exercise 2: Script a Safe Symbol-Protection Runbook . . . 204
24.6 Exercise 3: Automate Gateway Exposure Cleanup . . . 205
24.7 Exercise 4: Tune pm-mm-bot Startup Reliability . . . 205
24.8 Exercise 5: Empty-Book Bootstrap Drill . . . 206
24.9 Exercise 6: Build a Combined Automation Flow . . . 206
24.10Summary . . . 207
24.11Reflection . . . 207
24.12Further Reading . . . 207

25 RALF Post-Trade Gateway Protocol Training **209**
25.1 Objective . . . 209
25.2 Prerequisites . . . 209
25.3 Background . . . 210
25.4 Exercise 1: Prepare a Config That Enables RALF . . . 210
25.5 Exercise 2: Use the Python Example as a CLEARING Consumer . . . 211
25.6 Exercise 2B: Manual Handshake Probe with nc . . . 211
25.7 Optional Exercise 3: Use the C Example as a DROP_COPY Consumer . . . 211
25.8 Exercise 4: Run an AUDIT Session and Compare View Semantics . . . 212
25.9 Exercise 5: Protocol Control Messages (PING, UNSUB, EXIT) . . . 212
25.10Support Libraries and Example Clients . . . 213
25.11Exercise 6: Replay and Recovery with LASTSEQ . . . 213
25.12Exercise 7: Error Conditions and Operational Interpretation . . . 213
25.13Suggested Lab Pattern for the Three External Parties . . . 214
25.14Summary . . . 214
25.15Reflection . . . 214
25.16Further Reading . . . 215

26 CALF Market-Data Gateway Protocol Training **217**
26.1 Objective . . . 217
26.2 Prerequisites . . . 217
26.3 Background . . . 218
26.4 Exercise 1: Prepare a Config That Enables CALF . . . 218
26.5 Exercise 2: Use the Python Example as a TOP/TRADE Consumer . . . 219
26.6 Exercise 2B: Manual Handshake Probe with nc . . . 219
26.7 Optional Exercise 3: Use the C Example Subscriber . . . 220
26.8 Exercise 3B: `pm-calf-spy` — the Read-Only Diagnostic CLI . . . 220
26.9 Exercise 4: State Channel and Wildcard Behavior . . . 221
26.10Exercise 4B: Auction Results and Circuit-Breaker Detail . . . 221
26.11Exercise 4C: Indicative Auction Price (`INDIC`) . . . 222
26.12Exercise 4D: Automated Corridor Expansion (ACE) . . . 223
26.13Exercise 4E: Reference Data — Tick Decimals (`REF`) . . . 223
26.14Exercise 5: Protocol Control Messages (PING, UNSUB, EXIT) . . . 224
26.15Exercise 6: Replay and Recovery with RESUME . . . 224

26.16Exercise 7: Error Conditions and Operational Interpretation . . . 225
26.17Support Libraries and Example Clients . . . 225
26.18Summary . . . 226
26.19Reflection . . . 226
26.20Further Reading . . . 226

27 API Gateway REST/WebSocket Training 227
27.1 Objective . . . 227
27.2 Prerequisites . . . 227
27.3 Background . . . 228
27.4 Exercise 1: Generate a Single API Gateway Config . . . 229
27.5 Exercise 2: Start the API Gateway Process . . . 230
27.6 Exercise 3: Explore Swagger with a Bearer Key . . . 230
27.7 Exercise 4: Submit and Cancel an Order over REST . . . 231
27.8 Exercise 5: Query Reference Data . . . 231
27.9 Exercise 6: Use the Python REST Example . . . 232
27.10Optional Exercise 7: Use the C REST Example . . . 232
27.11Exercise 8: Query Order and Trade History . . . 233
27.12Exercise 9: Observe WebSocket Events . . . 233
27.13Exercise 10: Admin REST — Session Control, Risk, and Cross-Gateway Orders . . . 235
27.14Exercise 11: Configure Multiple Logical API Gateways . . . 236
27.15Exercise 12: Write a Python CLI Client for LIMIT Order Entry . . . 237
27.16Support Libraries and Example Clients . . . 238
27.17Summary . . . 238
27.18Reflection . . . 238
27.19Handoff for Chapter 26 . . . 239

28 Market Index (`pm-index`) 241
28.1 Objective . . . 241
28.2 Prerequisites . . . 241
28.3 Background . . . 242
28.4 Exercise 1: Configure an Index . . . 242
28.5 Exercise 2: Start `pm-index` . . . 243
28.6 Exercise 3: Query the Live Index Level . . . 244
28.7 Exercise 4: Watch the Index Move . . . 245
28.8 Exercise 5: Apply a Stock Split . . . 245
28.9 Exercise 6: Apply a Cash Dividend . . . 246
28.10Exercise 7: Add and Remove a Constituent . . . 247
28.11Exercise 8: Query History with `pm-index-cli` and `pm-stats-cli` . . . 247
28.12Exercise 9: Export and Analyse Data . . . 248
28.13Exercise 10: Two Indices . . . 249
28.14Exercise 11: History Query via the Gateway . . . 250
28.15Summary . . . 251
28.16Reflection . . . 252
28.17See Also . . . 252

29 ALF TCP Gateway Protocol Training 253
29.1 Objective . . . 253

29.2 Prerequisites . . . 253
29.3 Background . . . 254
29.4 Exercise 1: Generate a Config That Enables pm-alf-gwy . . . 254
29.5 Exercise 2: Verify the Port Is Listening with CLI Tools . . . 255
29.6 Exercise 3: Manual Handshake Probe with nc . . . 256
29.7 Exercise 3B: telnet and Non-Interactive One-Liner . . . 256
29.8 Exercise 4: Submit and Cancel an Order over nc . . . 257
29.9 Exercise 5: Use the Python Interactive Client . . . 257
29.10 Exercise 6: Use the Python Library Directly . . . 258
29.11 Optional Exercise 7: Build and Run the C Client . . . 259
29.12 Exercise 8: Control Commands and the Kill-Switch . . . 260
29.13 Exercise 9: Error Conditions and Operational Interpretation . . . 261
29.14 Support Libraries and Example Clients . . . 262
29.15 Summary . . . 263
29.16 Reflection . . . 263
29.17 Handoff for Chapter 28 . . . 263
29.18 Further Reading . . . 263

30 BALF TCP Gateway Protocol Training 265
30.1 Objective . . . 265
30.2 Prerequisites . . . 265
30.3 Background . . . 266
30.4 Exercise 1: Generate a Config That Enables `pm-balf-gwy` . . . 266
30.5 Exercise 2: Verify the Port Is Listening with CLI Tools . . . 267
30.6 Exercise 3: Manual Binary `LOGON` Probe with Python . . . 267
30.7 Exercise 3B: Non-Interactive Reachability Check for CI . . . 268
30.8 Exercise 4: Submit and Cancel an Order over BALF . . . 269
30.9 Exercise 4B: Amend an Order and Receive Fills . . . 271
30.10 Exercise 5: Validate Parser Libraries (Python and Optional C) . . . 273
30.11 Exercise 6: Duplicate Session Policy and Heartbeat Behaviour . . . 274
30.12 Exercise 7: Error Conditions and Operational Interpretation . . . 274
30.13 Support Libraries and Example Clients . . . 275
30.14 Summary . . . 275
30.15 Reflection . . . 275
30.16 Where to go next . . . 276
30.17 Further Reading . . . 276

31 IPO Valuation (`pm-valuation`) 277
31.1 Objective . . . 277
31.2 Prerequisites . . . 277
31.3 Background . . . 278
31.4 Exercise 1: A First Report . . . 278
31.5 Exercise 2: Where the Value Comes From . . . 279
31.6 Exercise 3: The What-If Loop . . . 280
31.7 Exercise 4: How Sure Are We? . . . 280
31.8 Exercise 5: A Postponed IPO . . . 280
31.9 Exercise 6: Save and Hand Out a Case . . . 281
31.10 Exercise 7: Let the Market Decide . . . 281

31.11Exercise 8: No Market Maker — the Crowd Sets the Price 282
31.12Summary 283
31.13Reflection 283
31.14See Also 284

CHAPTER 1

Training Guide

Welcome to the EduMatcher self-study training programme. This hands-on guide takes you from a cold start to confidently operating every major feature of the exchange, one exercise at a time, with a checkpoint after each step.

Who it is for	Students, self-learners and classes who learn best by doing
What it assumes	Basic command-line and YAML skills. No finance background is needed
How it relates to the other books	Each chapter links to the Participant Guide, Operator's Guide or Protocols and Clients for the explanation, and to the Reference Manual for details — the exercises do not repeat them

If you are new to finance, first read How an Exchange Works, a non-technical introduction to what an exchange does. If you have not run EduMatcher at all yet, the Quick Start Guide is the gentler first hour; this guide then goes deeper.

1.1 How to use this guide

The 30 chapters (00–29) form **one sequence**. Each chapter's *Prerequisites* section names what it needs, and for almost every chapter that is everything before it: later exercises reuse the configuration, the processes and the orders that earlier ones set up. Work through the chapters in order and stop when you have covered what you need — Choose your path below shows where. Three chapters can be taken early:

- **18 — Exchange Observer Processes** needs only chapters 00–03. Taking it straight after chapter 03 gives you the viewers and monitors for every later chapter.
- **26 — Market Index** needs only chapters 00–03 (plus `pm-stats` from chapter 15 for Exercise 9).
- **29 — IPO Valuation** needs only chapter 00 for its first six exercises; the last two also need 00–03 and 07.

1.1.1 What you need

- **EduMatcher installed.** Any of the routes in the Operator's Guide Installation chapter works; chapter 00 walks through them and helps you pick.
- **Several terminals.** The exchange is a set of processes, and this guide asks you to watch them side by side. A multiplexer (`tmux`, `screen`) or split panes helps, but separate windows are fine.
- **A text editor and a working folder.** You will write YAML files (configurations, bot settings) and keep them in one folder. Chapter 00 sets it up; on the container route it matters, because the container has no editor and its own file system is thrown away on every restart.

1.1.2 The two consoles

This is the one thing worth knowing before chapter 00, because mixing them up is the most common early mistake:

	`pm-alf-console`	`pm-admin`
You are	a **trader**	the **exchange operator**
Prompt shown in this guide	`[TRADER01]>`	`[GW_ADMIN|ADMIN]>`
Typical commands	`NEW`, `AMEND`, `CANCEL`, `STATUS`, `ORDERS`, `POS`, `QUOTE`, `QLEGS`	`BOOK`, `HALT_SYM`, `CANCEL_SYM`, `KILL|GW=`, `SESSION|STATE=`, `GATEWAYS`
Can see the whole order book	No	Yes

A prompt containing `|ADMIN]>` always means the `pm-admin` terminal. Anything else is a trader console. Commands are **not** interchangeable between them — a `BOOK` typed into a trader console just answers `Unknown command.`

The configuration you build in chapter 01 calls the operator `GW_ADMIN`. The integration chapters 23–28 each generate a fresh configuration whose operator is `OPS01`, the same name the bundled example configurations and the Quick Start use. The role, `ADMIN`, is what matters.

1.1.3 Keeping the market open

Your configuration has sessions enabled, so the engine starts every run in `CLOSED` and rejects orders until the trading day is opened. Chapter 01 shows how to open it by hand from the operator console — `SESSION|STATE=PRE_OPEN`, then `SESSION|STATE=CONTINUOUS` — and the exercises assume you do this after every engine restart. If an order comes back `REJECTED … code=MARKET_CLOSED`, this is why.

1.1.4 After every engine restart

Many exercises stop `pm-engine` and start it again to pick up a new configuration. A restarted engine knows nothing about the consoles that were connected to the old one. When either console detects that its engine connection was lost, it prints `Engine connection was lost; logging out.` and exits; **consoles do not reconnect by themselves**:

After each engine restart, start the consoles again, then reopen the market. A quote or order that is still resting from before the restart is still there.

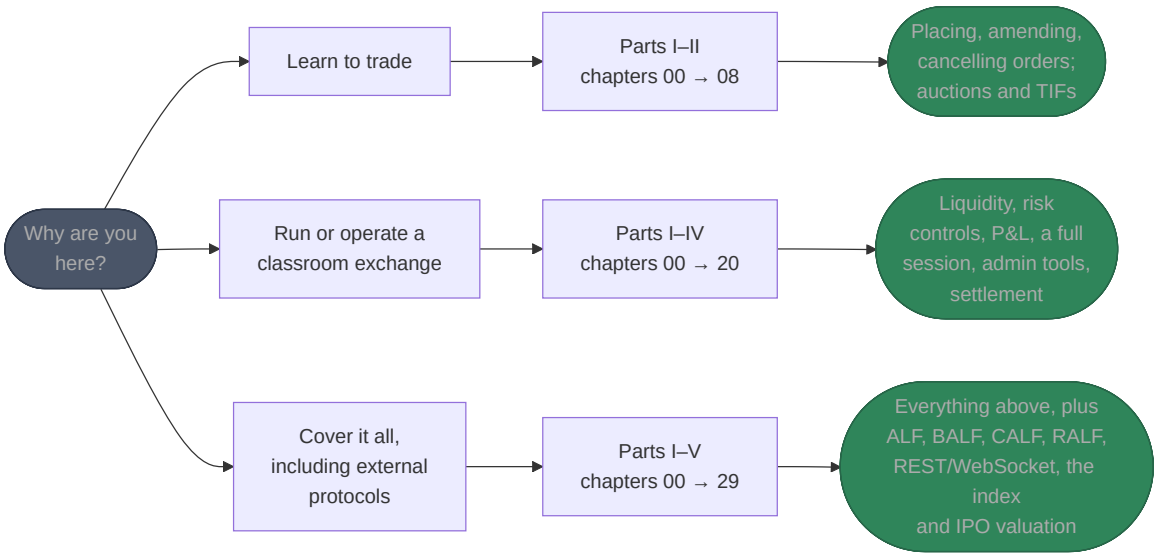
Expected health diagnostics

When monitoring is disabled, processes send no health beacons. If monitoring is configured but `pm-monitor` is not running, the rate-limited `pm-monitor … not accepting beacons` diagnostic is DEBUG and appears only with `-vv`. The same DEBUG-level policy applies to dropped engine heartbeats during a restart. The console still prints its logout message when it loses the engine connection; restart the console after restarting the engine.

1.1.5 Conventions

- `[TRADER01]>` — type this at that trader console's prompt.
- `[GW_ADMIN|ADMIN]>` — type this at the operator console's prompt.
- A plain `$` or a bare command — type this in a normal shell.
- `[output]` — expected output; exact wording may vary between versions.
- :material-checkbox-blank-outline: — a checkpoint. Verify it before moving on; if it fails, the next exercise will not behave as described.

1.2 Choose your path



Your goal	Work through	What you'll have
Learn to trade — place, amend, and cancel orders; understand fills, TIFs, order types, and auctions	Parts I–II (00–08)	Everything a working trader needs day to day

Your goal	Work through	What you'll have
Run or operate a classroom exchange — liquidity, risk controls, P&L, a full session, admin tools, end-of-day settlement	Parts I–IV (00–20)	Everything above, plus the operator's toolkit
Cover everything, including external protocols	Parts I–V (00–29)	The full training programme

1.3 Training plan

The five parts follow the chapter numbers, so reading the book front to back is reading the chapters in order.

1.3.1 Part I — Foundations

#	Chapter	You will be able to	Background reading
00	Installation & Setup	Install EduMatcher and know where its files live	Installation, Quick Start Guide
01	Configuring & Starting Up	Author, verify, deploy a configuration; start the engine, scheduler and both consoles	The Configuration Workflow, Config Verifier, Running the Exchange
02	Setting Up Market-Maker Liquidity	Seed a two-sided book so there is something to trade against	Market Making, Market-Maker Bot
03	The First Trade	Submit orders, read fills, follow an order's lifecycle	ALF Console, Order Types

1.3.2 Part II — Trading mechanics

#	Chapter	You will be able to	Background reading
04	Amending Orders	Change price and quantity, and predict the effect on queue priority	Order Amendment
05	Order Types Deep Dive	Use MARKET, STOP, STOP_LIMIT, FOK, IOC, ICEBERG and TRAILING_STOP	Order Types
06	Time-in-Force & Sessions	Choose a TIF and predict how each session phase treats it	Auctions & Scheduling
07	Auctions	Run an auction and compute the equilibrium price by hand	Auctions & Scheduling — Equilibrium price

#	Chapter	You will be able to	Background reading
08	Cancelling & Managing Orders	Inspect and cancel resting orders; use the operator's bulk-cancel tools	ALF Console, The Admin Console and Exchange Commands

1.3.3 Part III — Liquidity, risk and results

#	Chapter	You will be able to	Background reading
09	Market Making	Run a two-sided quote by hand and inspect its legs	Market Making
10	Combo Orders	Size and close pairs/baskets; test OCO exits, conditional legs, failure cascades and recovery	Combo Orders
11	Risk Controls	Configure and trigger collars, circuit breakers, halts and the kill switch	Risk Controls
12	P&L & Clearing	Read positions, VWAP cost and realized/unrealized P&L	Positions and P&L
13	Market Data & Drop Copy	Explain the market-data and drop-copy feeds and watch them	Drop Copy, DC Gateway
14	AI Traders & News	Run an AI market with a hidden true value and move it with news	AI Traders

1.3.4 Part IV — Running a full session

#	Chapter	You will be able to	Background reading
15	Statistics & Reporting	Query OHLCV, trades and snapshots for analysis	Statistics and Reporting
16	Persistence & Recovery	Say what survives a restart, and verify it	Persistence, Audit Trail
17	Capstone Scenario	Run a full session end to end, combining everything above	Running the Exchange
18	Exchange Observer Processes	Compare the book, tape, order monitor, audit trail, stats and clearing views side by side	Processes, Environment and Ports
19	Advanced Admin Operations	Use `KICK`, `QCANCEL`, `CANCEL_SYM` and manual session overrides	The Admin Console and Exchange Commands
20	Closing the Day	Settle a trading day with `pm-eod`, read the statements, and recover from a missed day, a broken roster and trading after the close	Closing the Day

1.3.5 Part V — Integration and advanced topics

Chapters 21, 23–25, 27 and 28 connect *other software* to the exchange over its wire protocols and APIs; 22 automates the operator, 26 adds a market index and 29 prices an IPO.

#	Chapter	You will be able to	Background reading
21	Drop-Copy Replay & Recovery	Detect sequence gaps and recover a consumer	Drop Copy — Replay
22	Automation & MM Bot Tuning	Script operator workflows and tune `pm-mm-bot`	The Admin Console and Exchange Commands, Market-Maker Bot
23	RALF Post-Trade Protocol	Write a clearing, drop-copy or audit consumer	RALF Gateway, RALF Protocol
24	CALF Market-Data Protocol	Write a market-data consumer with snapshots and replay	CALF Gateway, CALF Protocol
25	API Gateway REST/WebSocket	Trade and query over REST, and stream over WebSocket	API Gateway, REST API Reference
26	Market Index (`pm-index`)	Run an index and apply corporate actions without disturbing its level	Market Index, Index Administration (`pm-index-admin-cli`)
27	ALF TCP Gateway	Speak ALF over a raw socket	ALF Gateway, ALF Protocol
28	BALF TCP Gateway	Speak the binary order-entry protocol	BALF Gateway, BALF Protocol
29	IPO Valuation	Price an IPO, then list it and let the opening auction judge the price	IPO Valuation, Listing a New Symbol

1.4 After the training

Use the Participant Guide, the Operator's Guide, the Reference Manual and the Glossary for day-to-day reference.

Part I

Foundations

Chapters 00–03. Install EduMatcher, write and deploy a configuration, start the exchange, give it market-maker liquidity and make the first trade.

CHAPTER 2

Installation & Setup

2.1 Objective

Get a working EduMatcher, understand **where its files live**, and know **how to change them**. There are four ways to install; you need exactly one, and **Containers is the recommended default** — it needs no Python setup, gives you the exchange plus six web applications in a single command, and is what the rest of this chapter assumes unless you have a specific reason to choose otherwise. By the end of this chapter you will have an exchange you can start and stop, a deployed configuration, and a clear picture of which directory holds your trades and logs.

Background reading

- Installation - Quick Start Guide

2.2 Exercise 0: Read How an Exchange Works

Not strictly required, but it gives you a mental model of the core components and data flows in an exchange, which makes the rest of the training far more intuitive. Especially recommended if you are new to how exchanges work under the hood, or have no financial background.

You can refer back to it at any time; the concepts get clearer as you see them in action.

2.3 Choosing how to install

Route	You need	You get	Best for
A — Containers (recommended)	Podman or Docker	The exchange **and six web applications**, in one command	Most students: no Python to install, the fastest path to a live market, and works through every chapter that follows

Route	You need	You get	Best for
B — pipx	Python 3.14	`pm-*` commands on your PATH	You would rather work outside a container, or Podman/Docker is not available to you
C — Multipass VM	Multipass	A Linux VM with `pm-*` inside it	Workshops; keeping your own machine untouched; a snapshot you can reset
D — Poetry checkout	Python 3.14, Git	The repository plus dev dependencies	Changing EduMatcher itself

How this affects the rest of the training

From Chapter 01 onward you start and stop **individual processes by hand**, one per terminal — that is how the exercises teach you what each process does. Routes B and C give you that directly, in your own terminals.

Route A gives you the same thing with one extra step: the containerised exchange starts its processes for itself, so before following Chapter 01 you open a shell *inside* the container and stop them first. Exercise 1 below shows exactly how, and it is a one-time step per session — after that, every `pm-*` command in the chapters works unchanged. This is why Route A is still the recommended default even though the chapters are demonstrated as bare commands. The differences are prefixing your first command with `./edumatcher.sh shell`, and where your own files live: see "Where to keep your working files" below.

Pick one route below, then continue from Exercise 2 — the remaining exercises apply to all of them.

2.4 Exercise 1: Containers — the whole system in one command

You need Podman or Docker Desktop installed and running before you start. If neither is available, the installer stops immediately and tells you so; there is nothing else to prepare.

2.4.1 Where everything will be stored

Before running an installer, know what it creates. **Everything lives in one directory**, `~/.edumatcher` by default. Nothing is written anywhere else: no system paths, no service registered, no change to your PATH. Deleting that directory removes the installation.

Path	Holds
`~/.edumatcher/data`	**Every trade, order book, log and database the exchange produces.** It is mounted into the containers, so it lives on your disk rather than inside a container, and survives stop, start and update
`~/.edumatcher/config`	An engine configuration of your own, once you supply one
`~/.edumatcher/.env`	Your settings — version, configuration, ports, timezone

Path	Holds
`~/.edumatcher/compose.yaml`, `compose.zmq.yaml`	The container definitions. You will not normally edit these
`~/.edumatcher/edumatcher.sh`	The command you drive everything with

Use `--dir` to put it somewhere else; the layout underneath is the same.

2.4.2 Install

```
curl -fsSL https://raw.githubusercontent.com/johan162/EduMatcher/main/deployment/curl/install.sh | bash
```

Security-first variant

To read the script before running it: `bash curl -fsSL https://raw.githubusercontent.com/johan162/EduMa -o install.sh less install.sh bash install.sh`
The installer checks for Podman or Docker, resolves the newest release, downloads four small support files (`compose.yaml`, `compose.zmq.yaml`, `edumatcher.sh`, `.env.example`), pulls seven images (the exchange and the six web applications) and starts them. Nothing is compiled on your machine.
When it finishes you have a complete exchange **and** six web applications:

Application	URL	What it is
TapeDeck market display (`pm-terminal`)	http://localhost:8090	Live order books, trades and market data
Log Operator Console (`pm-log-ui`)	http://localhost:8091	The centralized log, searchable
Configuration GUI (`config-gui`)	http://localhost:8092	Author an `engine_config.yaml` in your browser
Trading GUI (`pm-trading-ui`)	http://localhost:8093	Submit and manage orders as a participant
Order Book Viewer (`pm-book`)	http://localhost:8094	One symbol's full order book, statistics and trade tape — `pm-viewer` in the browser
Operator Dashboard (`pm-dash`)	http://localhost:8095	Whether every part of the exchange is healthy, and how busy it is, live and over time
REST API docs	http://localhost:8080/docs	Swagger UI for the `desk` API gateway

Open TapeDeck. You should see order books with live quotes: the bundled `s10-basic` configuration seeds each symbol with a resting market-maker bid and ask the moment `pm-engine` starts, so there is a two-sided book before anyone has traded.

If a port is already taken

The REST API ports (`8080`, `8081`) are fixed; the six web applications use `8090`–`8095` by default, and you can move any of them with the `*_GUI_PORT` settings in `.env` (Exercise 4). If another program on your machine already uses one of these ports, `./edumatcher.sh start` fails with a bind error naming that port. Free it, move the conflicting service, or change the port in `.env`, then run `./edumatcher.sh start` again.

Running a second install on the same machine

Every container and its ports are fixed by name, so a second `curl | bash` into a different `--dir` will refuse to start once it reaches the point of creating the exchange container, naming the other install's data directory and telling you to stop it first (`./edumatcher.sh stop` in that other directory, or `make -C <repo>/deployment/docker down-all` for a source checkout). Stop one before starting the other; the two are not designed to run side by side.

2.4.3 Driving it

Everything runs from the install directory through one script:

```
cd ~/.edumatcher
./edumatcher.sh status              # containers, plus the exchange process table
./edumatcher.sh urls                # the table above, with your ports
./edumatcher.sh logs terminal-gui   # follow one service
./edumatcher.sh mounts              # which directory is behind each container path
./edumatcher.sh stop                # stop everything; your data is kept
./edumatcher.sh start               # bring it back
```

2.4.4 Running `pm-*` commands for the rest of the training

Every `pm-*` command is inside the exchange container, already on the PATH. The control script puts you there:

```
cd ~/.edumatcher
./edumatcher.sh shell
```

Inside that shell, `pm-engine`, `pm-alf-console`, `pm-config-deploy` and the rest work exactly as the training chapters describe, and `EDUMATCHER_DATA_DIR` is already set to `/data`.

To run a single command without staying inside, pass it along:

```
./edumatcher.sh shell pm-opctl-cli list
```

One difference matters. The container **starts the processes for you** — the `mm-demo` profile (the full process set plus a slow market-maker bot) is already running, which is why TapeDeck showed a live market. The training chapters assume you start them yourself, so stop them first:

```
pm-opctl-cli list           # see what is running
pm-opctl-cli stop           # stop the profile; the container stays up
pm-engine --verbose         # now follow the chapters
```

For the exercises that need several terminals, run `./edumatcher.sh shell` once in each of them.

2.4.5 Where to keep your working files

From Chapter 01 on you write YAML files (`engine_config.yaml`, `mm_aapl.yaml`, …) and run commands that read them. On the container route this needs one deliberate step, because the container is not a normal machine: it opens in `/`, it has **no text editor**, and anything you save in its own file system is deleted when `./edumatcher.sh restart` or `stop` removes the container. The one folder that your computer and the container share, and that survives, is the data directory. So keep your files there:

```
# on your computer: make the folder, and edit files in it with your own editor
mkdir -p ~/.edumatcher/data/training

# in every container shell (./edumatcher.sh shell), before running anything:
cd /data/training
```

`~/.edumatcher/data/training` on your computer and `/data/training` inside the container are the same folder. Whenever a chapter says "in your working directory", it means this one. Check with `ls /data/training` inside the container.

Docker on Linux

With Docker (not Podman) the container writes as `root`, so a file that a command *creates* there (for example `generated.yaml`) belongs to `root` on your computer. You can read it, but to edit it run `sudo chown -R "$USER" ~/.edumatcher/data/training` first.
On Routes B, C and D simply make a folder of your own and work in it:

```
mkdir -p ~/training && cd ~/training
```

Getting back to a running market

`pm-opctl-cli start` restarts the whole profile whenever you want the web applications populated again.
:material-checkbox-blank-outline: **Checkpoint:** http://localhost:8090 shows order books, and `./edumatcher.sh shell pm-engine --version` prints a version number.

2.5 Alternate Routes

Skip this section if you took Route A. It covers three routes for specific situations: working outside a container, a disposable sandbox, and contributing to EduMatcher itself. Each still ends at the same place — `pm-engine --version` resolving and a data directory you can name — so Exercise 2 onward works the same regardless of which one you picked.

2.5.1 Route B: pipx — commands on your own machine

Choose this when you would rather not run a container, or Podman/Docker is not available to you.

Requirement	Notes
Python 3.14 or later	Check with `python --version`
`pipx`	Installs command-line applications into isolated environments
Several terminals	Or `tmux` / `screen`; one process per pane is normal

```
# macOS with Homebrew
brew install pipx
pipx ensurepath

# Linux, or macOS without Homebrew
python -m pip install --user pipx
python -m pipx ensurepath
```

pip install pipx on its own often fails

On a system Python (most Linux distributions since 2023, including Debian and Ubuntu), a bare `pip install pipx` refuses with `error: externally-managed-environment`. The `--user` flag above installs into your own home directory instead of the system site-packages, which is both what avoids that error and the setup pipx itself recommends.
`pipx ensurepath` edits your shell's startup file but cannot update the *current* shell. **Open a new terminal** before continuing, then install EduMatcher into it:

```
pipx install edumatcher
```

Verify:

```
pm-engine --version
```

command not found: pm-engine

Almost always the same cause: you are still in the terminal that was open when you ran `pipx ensurepath`. Open a new one (or `exec $SHELL`) and try again before suspecting anything else.
Then bootstrap your workspace:

```
pm-setup
```

`pm-setup` creates the data directory, compiles a bundled example configuration (`s10-basic` unless you say otherwise) and installs it as the deployed artifact, then prints the one environment variable you need. The output looks like this — yours will show your own home directory and shell:

```
pm-setup — EduMatcher session initialisation
================================================
  ✓ Created data directory:          /Users/you/.local/share/edumatcher
    ✓ Example config 's10-basic' compiled to: /Users/you/.local/share/edumatcher/ref_data/engine_config.json
    10 symbol(s) ready to trade.

  Shell environment snippet — add to your shell profile:
  (~/.zshrc)
  ------------------------------------------------
  export EDUMATCHER_DATA_DIR="/Users/you/.local/share/edumatcher"
  ------------------------------------------------

  This is the only variable to set. Every process derives its
  configuration, database and log paths from it, so they cannot
  drift apart.
  ...
```

Add the `export` line to `~/.zshrc` (macOS default shell) or `~/.bashrc` (Linux, or bash on macOS) — `pm-setup` names the right file for your shell in its own output — then reload it:

```
source ~/.zshrc     # or source ~/.bashrc — whichever pm-setup named
```

Re-running pm-setup

`pm-setup --force` replaces an already-deployed configuration with the bundled example. `pm-setup --config s10-nominal` picks a different one. `pm-setup --no-config` creates only the data directory.
:material-checkbox-blank-outline: **Checkpoint:** `pm-engine --version` prints a version, and `echo $EDUMATCHER_DATA_DIR` prints a path — **in a new terminal**, not just the one where you ran `pm-setup`.

Want the web applications on this route?

They are not in the Python package, but if Podman or Docker is available you can run them as ready-built containers next to your `pipx` exchange: start the exchange with `EDUMATCHER_GATEWAY_BIND_HOST=0.0.0.0 pm-opctl-cli start`, then `curl -fsSL https://raw.githubusercontent.com/johan162/EduMatcher/main/deployment/curl/install.sh | bash -s -- --guis`. The Operator's Guide chapter Installation has the details.

2.5.2 Route C: Multipass VM — a disposable Linux sandbox

A ready-to-run Linux environment without installing Python tooling on your host. Install Multipass from multipass.run, verify it, then bootstrap:

```
multipass version

curl -fsSL https://raw.githubusercontent.com/johan162/EduMatcher/main/deployment/vm/curl_setup_vm.sh | bash -s --
```

If `multipass version` fails with `command not found`, the install did not complete or your shell has not picked up its PATH change yet — open a new terminal and retry before moving on.

This launches a VM (default name `ems`), installs the runtime inside it, runs `pm-setup` for you, takes a snapshot named `clean`, and prints the VM's address and API keys.

```
multipass shell ems
pm-engine --version
```

Inside the VM, `pm-*` commands work exactly as the chapters describe. The whole exchange can be started at once with `pm-opctl-cli start`, or process by process as the exercises do it.

```
multipass stop ems              # pause the VM
multipass start ems             # resume
multipass delete --purge ems    # remove it entirely
```

The snapshot is the point

Provisioning takes a snapshot named `clean`. After an exercise leaves the order books in a mess, restore it with the `-d` (destructive) flag, which skips the confirmation prompt and discards everything since: `bash multipass restore -d ems.clean`
:material-checkbox-blank-outline: **Checkpoint:** `multipass shell ems` works and `pm-engine --version` succeeds inside the VM.

2.5.3 Route D: Poetry checkout — for changing EduMatcher itself

Choose this only if you intend to modify EduMatcher's own source, not to run it as a student.

```
git clone https://github.com/johan162/EduMatcher.git
cd EduMatcher
poetry config virtualenvs.in-project true
poetry install --with dev,docs
```

Every command is then prefixed with `poetry run`, e.g. `poetry run pm-engine`. The data directory is the repository's own `src/data/` unless you set `EDUMATCHER_DATA_DIR`.

2.6 Exercise 2: Find where your files live

One idea underlies everything in this chapter: **one data directory is one exchange**. Every process reads its configuration from there and writes its trades, logs and databases back to it. Two processes pointed at different data directories are two different exchanges that cannot see each other.

Find yours:

If you installed with containers:

```
cd ~/.edumatcher
./edumatcher.sh mounts
```

The directory is `~/.edumatcher/data` on your disk, mounted as `/data` inside the container. Both names refer to the same files — that is why the Log Operator Console's health page says `/backend-data/log.db` while on your machine the file is `~/.edumatcher/data/log.db`.

If you installed with pipx, the VM, or Poetry:

```
echo $EDUMATCHER_DATA_DIR
ls -la "$EDUMATCHER_DATA_DIR"
```

`EDUMATCHER_DATA_DIR` is **the only location variable there is**. If it is unset, EduMatcher falls back to `~/.local/share/edumatcher` for an installed copy, or the repository's `src/data/` in a Poetry checkout.

Whichever route you took, the directory holds the same things:

Inside the data directory	What it is
`ref_data/engine_config.json`	The **compiled artifact every process actually reads**. Never edit it by hand
`ref_data/engine_config.yaml`	The authored source it was compiled from, kept for provenance
`log.db`	The centralized log, written by `pm-log-srv`
`stats.db`	Trade and market statistics, written by `pm-stats`
`clearing.db`	Positions and P&L
`audit.log`, `audit_index.db`	The audit trail and its index
`gtc_orders.json`, `book_stats.json`	Resting orders and book state across restarts
`emo/`	One log file per process, plus the process manager's state

Relative paths inside a configuration, such as `data/stats.db`, resolve under this directory — so they mean the same file no matter which directory you started a process from.

:material-checkbox-blank-outline: **Checkpoint:** you can name your data directory, and `ref_data/` inside it contains both `engine_config.json` and `engine_config.yaml`.

2.7 Exercise 3: Understand the deployed configuration

EduMatcher separates **the file you author** from **the file the exchange runs**:

```
your engine_config.yaml   →   pm-config-deploy   →   ref_data/engine_config.json
        (edit this)              (validates,              (every process
                                  compiles)                reads this)
```

No process takes a path to a configuration. There is exactly one deployed artifact per data directory, which is what makes it impossible to start two processes on different configurations by accident.

Look at what your install deployed:

If you installed with containers:

```
./edumatcher.sh shell pm-config-show
cat ~/.edumatcher/data/ref_data/engine_config.yaml
```

If you installed with pipx, the VM, or Poetry:

```
pm-config-deploy --show             # the paths of the source and compiled files
pm-config-show                      # the configuration as a terminal dashboard
cat "$EDUMATCHER_DATA_DIR/ref_data/engine_config.yaml"
```

You should see a `symbols:` section and a `participants:` section. This is the same *kind* of file you will author yourself in the next chapter — there, though, you keep your own copy under version control rather than editing the deployed one.

EduMatcher ships thirty ready-made configurations: 1, 3, 10, 30 or 150 order books (`s1` … `s150`), each as a `basic`, `nominal` or `complex` variant, and each of those also as a `-nomm` variant whose books start empty, with no market-maker quotes. See Example Engine Configs for what each contains.

:material-checkbox-blank-outline: **Checkpoint:** you can display the deployed configuration and see its symbols.

2.8 Exercise 4: Customise your installation

2.8.1 Choosing a different configuration

If you installed with containers:

```
cd ~/.edumatcher
./edumatcher.sh config s10-nominal      # a bundled example
./edumatcher.sh config ./my-market.yaml # or one of your own
./edumatcher.sh restart
```

A file of your own is copied into `~/.edumatcher/config/` and deployed on every start, so editing it and restarting is the whole edit-test loop.

If you installed with pipx, the VM, or Poetry:

```
pm-config-deploy ~/configs/classroom.yaml
```

Or give a scenario its own data directory, which isolates its statistics and logs as well:

```
export EDUMATCHER_DATA_DIR=~/sessions/classroom
pm-config-deploy ~/configs/classroom.yaml
```

2.8.2 Settings — the `.env` file (containers only)

`~/.edumatcher/.env` holds every choice about a running install. Compose reads it automatically because it sits beside `compose.yaml`; there is no other configuration file, and `compose.yaml` itself contains only `${VARIABLE}` references into it. The installer writes it once and then **keeps your copy**, so updating does not discard your edits.

Variable	Default	What it does
`EM_VERSION`	*(the installed release)*	Which release to run. All seven images carry this tag, so one value pins the whole system
`GHCR_OWNER`	`johan162`	The registry namespace images come from. Change only for a fork
`EM_CONFIG`	`s10-basic`	Which bundled example the exchange deploys
`EM_CONFIG_FILE`	*(empty)*	Set when you run a configuration of your own; non-empty wins over `EM_CONFIG`
`EM_PROFILE`	`mm-demo`	Which processes start: `mm-demo` (the full set plus a market-maker bot), `ai-swarm` (`mm-demo` with its market maker quoting around the market model's values, plus the model and twenty AI traders), `default` (the full set), `mini` or `micro`
`TZ`	`Europe/Stockholm`	Container timezone — match the trading calendar in your configuration
`BIND_ADDR`	`127.0.0.1`	Which host interface the published ports listen on — **the setting that decides whether the exchange is on your network**
`EM_ZMQ`	`0`	`1` also publishes the raw ZeroMQ bus, and tells the engine and `pm-index` to bind the container interface so host tools can attach

Variable	Default	What it does
`EDUMATCHER_GATEWAY_BIND_HOST`	`0.0.0.0`	Bind host for the gateways *inside* the container. This is what makes them reachable from the GUI containers; it is not a host-exposure setting
`TERMINAL_GUI_PORT`	`8090`	Host port for TapeDeck (`pm-terminal`)
`LOG_GUI_PORT`	`8091`	Host port for the Log Operator Console (`pm-log-ui`)
`CONFIG_GUI_PORT`	`8092`	Host port for the Configuration GUI (`config-gui`)
`TRADER_GUI_PORT`	`8093`	Host port for the Trading GUI (`pm-trading-ui`)
`BOOK_GUI_PORT`	`8094`	Host port for the Order Book Viewer (`pm-book`)
`DASH_GUI_PORT`	`8095`	Host port for the Operator Dashboard (`pm-dash`)

Change any of them and run `./edumatcher.sh restart`. Two have their own commands, because they need more than an edit — `./edumatcher.sh config` keeps `EM_CONFIG` and `EM_CONFIG_FILE` consistent, and `./edumatcher.sh update` pulls images after changing `EM_VERSION`.

BIND_ADDR=0.0.0.0 puts an unauthenticated exchange on your network

It is the right setting for a classroom where students connect to the instructor's machine, and the wrong one on a network you do not control: the protocol gateways have no password. The default keeps everything on this machine.

Two different 0.0.0.0s

`0.0.0.0` appears twice above and means two different things. On `BIND_ADDR` it is a real exposure decision: it opens the published ports to your LAN. On `EDUMATCHER_GATEWAY_BIND_HOST` it is not — that address lives inside the container's private network, where the only route in is a port `BIND_ADDR` published. Widening it lets the GUI containers reach the exchange and nothing else. The installation chapter works through all three places an address appears.
:material-checkbox-blank-outline: **Checkpoint:** you have switched to a different bundled configuration and seen the symbols change.

2.9 Exercise 5: Confirm your entry points

Verify the commands the coming chapters use. Prefix each with `./edumatcher.sh shell` on the container route, or run them directly otherwise:

```
pm-engine --help
pm-scheduler --help
pm-alf-console --help
pm-config-deploy --help
pm-config-gen --help
pm-mm-bot --help
```

Each should print usage information without errors.

pm-mm-bot is available

It is part of the installed command set. Chapter 02 starts with manual market-maker quotes so you understand quote mechanics first, then introduces the equivalent bot-based workflow.
:material-checkbox-blank-outline: **Checkpoint:** all commands respond to `--help`.

2.10 Troubleshooting

A quick index of the problems most students hit, and where above each one is handled in full:

Symptom	Cause	Fix
Installer exits immediately with "Neither podman nor docker is installed"	No container engine on this machine	Install Podman or Docker Desktop, then re-run the install command
`./edumatcher.sh start` fails with a bind error naming a port	Something else on your machine already uses `8080` or `8090`–`8095`	Free the port or stop the other service, then `./edumatcher.sh start` again
Starting a second install fails, naming another install's data directory	Container names and ports are fixed, so two installs cannot run side by side	Stop the other one (`./edumatcher.sh stop` in its directory), then retry — see the warning in Exercise 1
`pip install pipx` fails with `externally-managed-environment`	Modern Linux distributions block installing into the system Python	Use `python -m pip install --user pipx` instead, as Route B shows
`command not found: pm-engine` right after `pipx install edumatcher`	`pipx ensurepath` only takes effect in a *new* shell	Open a new terminal (or `exec $SHELL`) and try again
Poetry `pm-engine --version` is older than `pyproject.toml`	The project virtualenv still has stale editable-install metadata	From the repository root run `poetry install --with dev,docs`, then retry with `poetry run pm-engine --version`
`pm-engine --version` works in one terminal but not another	`EDUMATCHER_DATA_DIR` was exported in one shell but never added to your shell profile	Add the `export` line from `pm-setup`'s output to `~/.zshrc` or `~/.bashrc`, then open a new terminal

Symptom	Cause	Fix
`multipass version` reports `command not found`	Multipass did not finish installing, or your shell has not picked up its PATH change	Reinstall from multipass.run, then open a new terminal
Container route: chapter exercises see a market that already has orders in it	The `mm-demo` profile is still running from install	`pm-opctl-cli stop` inside `./edumatcher.sh shell`, as the "Before You Continue" checklist below requires

If something goes wrong that is not on this list, `./edumatcher.sh logs` (container route) or the relevant process's own `--verbose` output is the next place to look, and Running the Exchange has a fuller troubleshooting section for problems that show up after startup.

2.11 Summary

You now have:

- EduMatcher installed — Containers by default, or one of the three alternate routes for a reason of your own — and you know which one you chose.
- A data directory you can name, holding the deployed configuration and everything the exchange writes.
- A deployed configuration you can display, swap for another bundled example, or replace with your own.
- A way to run `pm-*` commands for the chapters that follow.

2.12 Before You Continue

Confirm every item before starting Chapter 01 — each is a prerequisite that chapter assumes without re-explaining:

- ☐ A configuration is deployed, and you can display it.
- ☐ You can name your data directory, and `ref_data/` inside it contains both `engine_config.json` and `engine_config.yaml`.
- ☐ `pm-engine --version` and `pm-engine --help` both resolve **in the shell you will use for the next chapter** — for pipx that means `EDUMATCHER_DATA_DIR` is in your shell profile, not just this terminal; for containers it means a shell opened with `./edumatcher.sh shell`.
- ☐ On the container route only: `pm-opctl-cli stop`, so the next chapter can start the processes itself.
- ☐ You have a working folder (`~/training`, or `~/.edumatcher/data/training` on the container route, which you reach as `/data/training` inside `./edumatcher.sh shell`) and a text editor to write YAML files in it.

2.13 Reflection

Why does EduMatcher refuse to let a process take a path to a configuration file, insisting instead on one deployed artifact per data directory? What would break in Chapter 16 (Persistence & Recovery) if two terminals ended up pointing at two different data directories?

And if you took the container route: the Log Operator Console reports its database as `/backend-data/log.db` while the exchange reports the same file as `/data/log.db`. Why do two names for one file make the system

safer rather than more confusing?

2.14 Further Reading

- Installation — the reference version of this chapter, with every flag, directory and build option
- Quick Start Guide — the short version of this chapter
- Running the Exchange
- Processes, Environment and Ports

Next: 01 — Configuring & Starting Up

CHAPTER 3

Configuring & Starting Up

3.1 Objective

By the end of this chapter you will have a working exchange with at least one gateway and three tradeable symbols, ready to accept orders. In addition you have become familiar with the three tools:

- **`pm-config-gen`** used to automatically generate configuration file based on options and flags.
- **`pm-cverifier`** used to verify an existing (possibly hand-crafterd) configuration file for errors or missing settings
- **`pm-config-deploy`** used to compile a verified configuration file and install it as the artifact `pm-engine/pm-sche` actually read

Background reading

- The Configuration Workflow - Config Verifier - Running the Exchange

3.2 Prerequisites

- EduMatcher installed and `pm-setup` completed (see 00 — Installation & Setup).

3.3 Background

EduMatcher requires two essential processes:

1. **pm-engine** — the matching engine.
2. **pm-scheduler** — drives session phase transitions.

A **gateway** (`pm-alf-console`) connects traders to the engine.

Neither process reads `engine_config.yaml` directly, and neither accepts a config path on its command line. `engine_config.yaml` is the file *you* author and edit; what they actually read is a **compiled artifact** at `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json`, produced by `pm-config-deploy` (see 00 — Installation & Setup for where that directory comes from). This chapter's exercises follow that exact pipeline: author → verify → deploy → start.

The artifact carries its own tamper and staleness checks: every load recomputes a SHA-256 over the artifact's own content and refuses to start if it no longer matches what was recorded at compile time, and every process separately checks the artifact's recorded hash of the *source* YAML against the YAML currently on disk, warning (but still starting) if you edited the source and forgot to redeploy. Exercise 12 walks through both.

3.4 Exercise 1: Create a Minimal Configuration

Create a file called `engine_config.yaml` in your working folder (the one you set up in chapter 00; on the container route that is `~/.edumatcher/data/training` on your computer, which you reach as `/data/training` after `cd /data/training` inside `./edumatcher.sh shell`). Every file this chapter and the next ones create or edit — `generated.yaml`, `schedule.yaml`, `mm_aapl.yaml` and the rest — lives in that same folder, and every command that reads one is run from it.

```
symbols:
  AAPL:
    tick_decimals: 2
    last_buy_price: 150.00
    last_sell_price: 150.00
  MSFT:
    tick_decimals: 2
    last_buy_price: 420.00
    last_sell_price: 420.00
  TSLA:
    tick_decimals: 2
    last_buy_price: 250.00
    last_sell_price: 250.00

participants:
  - id: TRADER01
    description: "Alice — first trader"
    role: TRADER
  - id: TRADER02
    description: "Bob — second trader"
    role: TRADER
  - id: GW_ADMIN
    description: "Exchange operator"
    role: ADMIN
  - id: ADMIN_CLI
    description: "One-shot command-line operator"
    role: ADMIN
```

Use `GW_ADMIN` for the interactive `pm-admin` console and `ADMIN_CLI` for `pm-admin-cli`. Each configured participant ID can have only one live connection; attempting to use `GW_ADMIN` from both tools at once is refused with `Gateway already connected: GW_ADMIN`.

Why tick_decimals, not tick_size

A symbol's price granularity is given as a **number of decimal places**, not a tick size: `tick_decimals: 2` means prices move in units of 0.01. The engine matches on integer ticks internally, which is why it is expressed this way.
`last_buy_price` / `last_sell_price` seed the reference price used by the price-collar checks. Without them

a symbol has no reference and its collar is never enforced — which is exactly the trap Chapter 11 Exercise 2 depends on avoiding.
Keys the engine does not recognise — `tick_size`, `last_price`, `description` on a *symbol* — are silently ignored rather than rejected, so a typo here fails quietly. `pm-cverifier` in Exercise 3 is how you catch that.
:material-checkbox-blank-outline: **Checkpoint:** file saved, YAML is valid (no tabs!).
This file is only the **authored source** — nothing reads it yet. `pm-engine` will not see any of this until you compile and install it with `pm-config-deploy` in Exercise 5.

3.5 Exercise 2: Generate a Config with pm-config-gen

Instead of writing YAML by hand, you can use the `pm-config-gen` helper to scaffold a configuration. Try generating an equivalent config:

```
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --symbol-opts AAPL:tick_decimals=2 \
  --symbol-opts MSFT:tick_decimals=2 \
  --symbol-opts TSLA:tick_decimals=2 \
  --participants TRADER01:TRADER TRADER02:TRADER GW_ADMIN:ADMIN ADMIN_CLI:ADMIN \
  --static-band 0.10 \
  --dynamic-band 0.05 \
  --sessions-enabled \
  --output generated.yaml --force
```

This writes `generated.yaml` beside your hand-written file — it does **not** replace it — with:

- Three symbols (AAPL, MSFT, TSLA) with 2-decimal tick precision.
- Two trader gateways and two admin gateways (one for each operator client).
- Static and dynamic price collars pre-configured.
- Session schedule enabled (PRE_OPEN → CONTINUOUS → CLOSED).

Inspect the generated file:

```
cat generated.yaml
```

Compare it with your hand-written `engine_config.yaml`. Two differences matter. The generated file spells out many settings you left to their defaults, with a comment on each — a good way to discover what can be configured. And it has **no reference prices** (`last_buy_price` / `last_sell_price`): `pm-config-gen` cannot know what a stock is worth, and the verifier warns about it (`C001`) — run `pm-cverifier generated.yaml` once you have met the verifier in the next exercise. Price collars and several later exercises need those prices, so **keep using your hand-written `engine_config.yaml`** for the rest of the training; `generated.yaml` is for reference.

Dry-run mode

Add `--dry-run` to preview the output without writing a file: `bash pm-config-gen --symbols AAPL MSFT TSLA --participants TRADER01 --dry-run`

Adding market-maker gateways

You can include MM gateways in the same command: `bash pm-config-gen \ --symbols AAPL MSFT TSLA \ --participants TRADER01 TRADER02 GW_ADMIN:ADMIN ADMIN_CLI:ADMIN MM_AAPL_01:MARKET_MAKER \ --enforce-mm-obligations \ --output mm-example.yaml --force`
Chapter 02 explains what a market maker needs in the configuration.
:material-checkbox-blank-outline: **Checkpoint:** `generated.yaml` has the same three symbols and four participants as your hand-written `engine_config.yaml`.

3.6 Exercise 3: Validate the Config with pm-cverifier

Before starting runtime processes, verify the file:

```
pm-cverifier engine_config.yaml
```

Check the verdict and exit code:

```
echo $?
```

Expected for this chapter config:

- Verdict is `WARN`, with one warning, `M032`: no read-only API credential is defined under `api_gateways`. That credential is what the browser dashboards use; this configuration starts no API gateway until Chapter 25, so you can leave the warning alone for now. Below it you will see a few `INFO` advisories (`M004`, `C003`, `C004`, `C007`) — suggestions, not problems.
- Exit code is `0` when there are no warnings/errors, `1` when warnings exist, and `2` when hard errors exist. Here it is `1`, because of `M032`.

Run it on `generated.yaml` as well and compare: that file has the same `M032`, plus the `C001` warning about the missing reference prices that Exercise 2 told you to expect.

CI-style validation

Treat warnings as failures and emit machine-readable output: `bash pm-cverifier --strict --format json engine_config.yaml`

Focus only on actionable items

Hide info-level advisories while iterating: `bash pm-cverifier --level warn engine_config.yaml`
:material-checkbox-blank-outline: **Checkpoint:** you can run `pm-cverifier`, read the verdict, and interpret its exit code.

3.7 Exercise 4: Practice Fixing Verifier Findings

Create a temporary broken config and use `pm-cverifier` to diagnose it:

```
cp engine_config.yaml engine_config.bad.yaml
```

Edit `engine_config.bad.yaml` and intentionally introduce two issues:

1. Remove the `GW_ADMIN` gateway entry.
2. Set one symbol's `tick_decimals` to an invalid value like `12`.

Run verifier:

```
pm-cverifier engine_config.bad.yaml
```

You should see only one finding:

- `S010` error (invalid `tick_decimals`).

Where is the missing admin gateway? The verifier works in layers — YAML syntax, then the schema of every field (`S` codes), then the rules *between* fields (`M` codes), then completeness advice (`C` codes) — and it stops after the first layer that finds an error, because later checks would be judging a file it already knows is broken. Fix `tick_decimals` and run it again: now you see

- `M013` warning (no gateway with role `ADMIN`), alongside the `M032` warning from Exercise 3.

Both `M013` and the `M032` you already know are now reported. Put the `GW_ADMIN` entry back and rerun until only `M032` remains.

:material-checkbox-blank-outline: **Checkpoint:** you can reproduce a verifier finding, map it to a check code, and clear it by fixing the config.

3.8 Exercise 5: Compile and Deploy the Config with pm-config-deploy

Verified YAML is still just a file on disk — no process reads it until it is compiled and installed. Check that it compiles without installing anything:

```
pm-config-deploy --check engine_config.yaml
```

Expected output:

```
OK — /home/you/training/engine_config.yaml compiles (3 symbol(s), 3 gateway(s))
```

Now actually deploy it:

```
pm-config-deploy engine_config.yaml
```

Expected output:

```
Compiled /home/you/training/engine_config.yaml
      to /Users/you/.local/share/edumatcher/ref_data/engine_config.json
   3 symbol(s), 3 gateway(s).
   Restart any running processes to pick it up.
```

Confirm where it landed:

```
pm-config-deploy --show
ls -la "$EDUMATCHER_DATA_DIR/ref_data"
```

Expected behavior:

- `--check` validates and reports symbol/gateway counts but writes nothing
- deploying overwrites the previous `ref_data/engine_config.json` atomically — a process reading it mid-deploy sees either the old configuration or the new one, never a half-written file
- if `engine_config.yaml` fails validation, deploy fails and prints `[ERROR] ...` followed by `Nothing was deployed.`; any previous deployment is left untouched
- deploying does **not** restart any already-running `pm-engine`/`pm-scheduler` — they must be (re)started to

pick up the new artifact, which Exercise 6 does next

:material-checkbox-blank-outline: **Checkpoint:** you can explain, in one sentence, the difference between `engine_config.yaml` and `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json`.

3.9 Exercise 6: Start the Engine

Open a terminal and run:

```
pm-engine --verbose
```

`pm-engine` takes no config path on its command line — it always reads whatever is currently deployed at `$EDUMATCHER_DATA_DIR/ref_data/engine_config.json`, which is exactly the artifact Exercise 5 just installed.

`--verbose` shows the engine's informational log lines; without it the engine prints only warnings and errors. Expected output includes lines like these (timestamps and paths will differ):

```
... INFO edumatcher.engine.main - Loaded config from .../ref_data/engine_config.yaml  (3 symbol(s): AAPL, MSFT, TS
... INFO edumatcher.engine.main - Session handling: enabled (startup state: CLOSED)
... INFO edumatcher.engine.main - Listening on PULL=tcp://127.0.0.1:5555  PUB=tcp://127.0.0.1:5556
```

`Session handling: enabled` means the engine starts in the `CLOSED` phase and waits to be told when the day begins: your configuration does not set `sessions_enabled`, and its default is `true`.

If monitoring is configured but `pm-monitor` is not running, the `pm-monitor … not accepting beacons` diagnostic is DEBUG and appears only when you start a process with `-vv`. It is rate-limited and does not affect the exchange.

The stable way to confirm the engine actually loaded your config, independent of log wording, is to query it from a gateway once connected (Exercise 8) with `SYMBOLS` — if it lists `AAPL`, `MSFT`, and `TSLA`, the engine started correctly regardless of what the startup banner said.

:material-checkbox-blank-outline: **Checkpoint:** engine is running without errors.

3.10 Exercise 7: Start the Scheduler

In a **second terminal**:

```
pm-scheduler --verbose
```

Like the engine, the scheduler prints only warnings and errors unless you pass `--verbose`. It runs **one day's** timetable from the deployed configuration and exits when the day ends. Your configuration has no `schedule:` block, so the built-in weekday timetable applies (pre-open 09:00, opening auction 09:25, continuous 09:30, closing auction 16:00, closed 16:05; Saturdays, Sundays and bank holidays closed). What you see therefore depends on when you start it:

- inside trading hours it first *catches up*, sending the transitions that were already due, and then waits for the next one:

  ```
  ... INFO edumatcher.scheduler.main - Catching up 3 transition(s) to reach the current phase
  ```

- on a weekend it reports that the day is closed and exits at once;
- outside hours on a weekday it catches up straight to `CLOSED`.

To skip the wait and drive the sequence immediately, restart it with `--now`:

```
pm-scheduler --now --delay 5 --verbose
```

Expected output (timestamps trimmed; between the transitions you also see a `Waiting 5s for …` line, and a harmless `pm-monitor` warning may follow `Done.`):

```
... INFO edumatcher.scheduler.main - --now mode: sending all transitions with 5.0s delays
... INFO edumatcher.scheduler.main - -> PRE_OPEN
... INFO edumatcher.scheduler.main - -> OPENING_AUCTION
... INFO edumatcher.scheduler.main - -> CONTINUOUS
... INFO edumatcher.scheduler.main - -> CLOSING_AUCTION
... INFO edumatcher.scheduler.main - -> CLOSED
... INFO edumatcher.scheduler.main - Done.
```

`--now` starts the day's sequence from this moment instead of the configured wall-clock times; `--delay 5` gives you five seconds between phases.

–now runs the whole day

`--now` does not stop at continuous trading: it carries on through the closing auction to `CLOSED`, about half a minute later, and exits. That is fine for watching the phases go by, but afterwards the market is closed and orders are rejected with `MARKET_CLOSED`. Exercise 10 shows how to open it again by hand, which is how the following chapters keep it open.

If nothing was deployed

`pm-scheduler` refuses to guess a schedule the engine has never seen. If Exercise 5 was skipped, it exits immediately with a fatal error naming `pm-config-deploy`, instead of silently falling back to a built-in timetable.
:material-checkbox-blank-outline: **Checkpoint:** you ran `pm-scheduler --now` and saw it send all five transitions, ending with `Done.`

3.11 Exercise 8: Connect a Gateway

In a **third terminal**:

```
pm-alf-console --id TRADER01
```

You should see:

```
Gateway TRADER01 connected.   — Alice — first trader Type HELP for commands.  Tab=complete  ↑↓=history  Ctrl-A/E=

[TRADER01]>
```

Try typing `ORDERS` — it should report no resting orders for this gateway.

:material-checkbox-blank-outline: **Checkpoint:** gateway prompt is interactive and connected.

3.12 Exercise 9: Verify the Setup

From the gateway prompt, confirm the three symbols are available by attempting a tiny limit order:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=0.01|TIF=DAY
```

If you have just done Exercise 7, the scheduler's `--now` day ended in `CLOSED`, so you will see this first:

```
[<time>] REJECTED  <order_id> code=MARKET_CLOSED  Market is closed
```

That is the engine working as designed, not a fault. Open the market in the next exercise (the last part of it, "Open the market for the chapters ahead"), then come back and send the same order again. Now you should see an acknowledgement (the order rests since no matching ask exists):

```
[<time>] ACK       <order_id>  order accepted
```

Repeat for `MSFT` and `TSLA` to confirm all three books are active.

:material-checkbox-blank-outline: **Checkpoint:** all three symbols accept orders (after the market is open).

3.13 Exercise 10: Connect the Operator Console

Two different consoles — this trips everyone up once

EduMatcher has **two** interactive consoles, and they accept different commands. Mixing them up is the most common early mistake, because the error is just `Unknown command`.

	`pm-alf-console`	`pm-admin`
Who it is	A **trader**	The **exchange operator**
Prompt	`[TRADER01]>`	`[GW_ADMIN|ADMIN]>`
Commands	`NEW`, `AMEND`, `CANCEL`, `STATUS`, `ORDERS`, `POS`, `QUOTE`, `QLEGS`, `SYMBOLS`, `SESSION`, `INDEX`	`BOOK`, `ORDERS|GW=`, `HALT`, `HALT_SYM`, `RESUME_SYM`, `CANCEL_SYM`, `KILL|GW=`, `KICK`, `QCANCEL`, `SESSION|STATE=`, `SESSION_STATUS`, `SCHEDULE`, `GATEWAYS`, `VOLUME`
Sees the whole book?	No — only its own orders	Yes

A trader cannot inspect the order book, halt a symbol, or move the session. Those are operator powers, and they live in `pm-admin`.
Throughout this guide, a prompt of `[GW_ADMIN|ADMIN]>` means "type this in the `pm-admin` terminal"; anything else means the trader console.
In a **fourth terminal**, start the operator console. Note this is `pm-admin`, **not** `pm-alf-console`:

```
pm-admin --id GW_ADMIN
```

`GW_ADMIN` must be a gateway with `role: ADMIN` in your configuration — your hand-written file from Exercise 1 has one.

Now look at the order book, which no trader console can do:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

You should see a book snapshot (possibly with the 1-lot bid from Exercise 9).

Try two more operator queries:

```
[GW_ADMIN|ADMIN]> GATEWAYS
[GW_ADMIN|ADMIN]> SESSION_STATUS
```

`GATEWAYS` lists every configured gateway and whether it is connected; `SESSION_STATUS` reports the current

trading phase.

3.13.1 Open the market for the chapters ahead

Chapters 02–05 need the market in continuous trading, whatever the time of day. Stop `pm-scheduler` (Ctrl-C in its terminal) and move the phase yourself:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
[GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
[GW_ADMIN|ADMIN]> SESSION_STATUS
```

`SESSION_STATUS` should now report `CONTINUOUS`. The engine only accepts the phase changes of a real trading day, in order — from `CLOSED` the only legal step is `PRE_OPEN` — so it takes two commands. Chapter 06 explains this "freeze and advance" technique properly; whenever a later exercise says the market must be open, this is how you open it.

Now go back to Exercise 9 and send the three 1-lot test orders (AAPL, MSFT, TSLA) if you have not yet.

3.13.2 Clear the test orders

Those three 1-lot bids at 0.01 are still resting, and they **survive engine restarts**. Left in place, they would show up in the order books and in `ORDERS` in every later chapter. Remove them with the operator's kill switch, which cancels everything one gateway has resting (chapter 08 explains it):

```
[GW_ADMIN|ADMIN]> KILL|GW=TRADER01
```

It answers `KILL OK TRADER01 orders=3 quotes=0`; the cancellations also show on the `TRADER01` console as `CANCELLED … reason=KILL_SWITCH`.

Engine restarts disconnect your consoles

Several exercises below (and in later chapters) restart `pm-engine`. When the engine connection is detected as lost, each console prints `Engine connection was lost; logging out.` and exits. Consoles do not reconnect by themselves, so start them again before continuing. See "After every engine restart" in the guide introduction.
:material-checkbox-blank-outline: **Checkpoint:** `pm-admin` is connected, `BOOK|SYM=AAPL` shows the book, and you can state which of the two consoles a given command belongs to.

Keep both terminals open from here on

Every later chapter assumes you have a trader console *and* an operator console available. Leaving both running saves restarting them — except after an engine restart, when each must be started again (see above).

3.14 Exercise 11: Inspect Enriched SYMBOLS Metadata

From any connected gateway:

```
[TRADER01]> SYMBOLS
```

Besides the symbol names, the table shows each symbol's tick size and its market-maker obligation settings — whether they are enforced, the maximum spread in ticks and the minimum quoted quantity:

```
| #    | Symbol     | Tick | MM Enforced | Max Spread | Min Qty |
| 1    | AAPL       | 0.01 |     NO      |         20 |     100 |
```

:material-checkbox-blank-outline: **Checkpoint:** you can read the tick size and the market-maker settings of each symbol from the `SYMBOLS` output.

3.15 Exercise 12: Break — and Recognise — the Artifact's Hash Checks

The compiled artifact records two independent SHA-256 hashes in its `meta` block: `content_sha256` (the artifact against itself) and `source_sha256` (the artifact against the `engine_config.yaml` it was built from). This exercise reproduces both failures on purpose, so you recognise the real message the first time you see it instead of the first time it costs you an afternoon. See Configuration — Verifying the Deployed Artifact for the full explanation this exercise is drilling.

Part A — trip the content-digest check (an error).

In the engine's terminal, stop it with `Ctrl-C`. Then hand-edit the *deployed* artifact directly — something `pm-config-deploy` never does and nothing should:

```
python3 -c "
import json
p = '$EDUMATCHER_DATA_DIR/ref_data/engine_config.json'
data = json.load(open(p))
data['engine']['symbols']['AAPL']['tick_decimals'] = 4
json.dump(data, open(p, 'w'), indent=2)
"
```

Start the engine again in the same terminal:

```
pm-engine
```

Expected: the engine refuses to start. It prints a Python traceback; read its **last line**, which reports that the compiled config's payload no longer matches its recorded content digest and names `pm-config-deploy`, not manual editing, as the fix.

Recover by recompiling the (untouched) source that is still sitting next to the artifact, then start the engine again:

```
pm-config-deploy engine_config.yaml
pm-engine
```

(The engine restart disconnected your consoles; restart them before using them again.)

A second, different message can come from the same digest check

The content-digest check actually distinguishes two separate causes for the same mismatch, and reports each with a different message. What you just triggered — editing the payload's *values* — is reported as `compiled config has been modified since it was compiled (payload digest ... does not match the recorded ...)`. If instead the artifact's on-disk bytes are untouched but a newer/older build of EduMatcher declares a different set of configuration fields than the artifact was compiled with, you get a different message: `compiled config is intact but was compiled against a different configuration schema ... Nothing has been edited — run pm-config-deploy to recompile it.` Both are fixed the

same way (`pm-config-deploy`), but only the second one means nobody touched anything — the check tells the two apart by separately hashing the payload exactly as it sits on disk, so don't go hunting for an edit that never happened if you see that second message after, say, upgrading EduMatcher itself without redeploying.

Part B — trip the source-staleness check (a warning).

Leave this engine running. In another terminal, edit the *authored* YAML and deliberately skip the deploy step — this only changes the file on disk, it does not affect the already-running engine:

```
sed -i.bak 's/tick_decimals: 2/tick_decimals: 3/' engine_config.yaml   # changes every symbol: each has its own t
```

Now stop the engine (`Ctrl-C`) and start it again without redeploying:

```
pm-engine
```

Expected: the engine **starts anyway**, but logs a warning that `engine_config.yaml` has changed since the running configuration was compiled and that the exchange is still serving the previous one. Confirm with `SYMBOLS` from a gateway (restart your trader console first, as the engine was restarted) that the symbols still report the old tick size (`0.01`, not `0.001`) — the warning told the truth, nothing was silently applied.

Restore the file and redeploy to clear the warning:

```
mv engine_config.yaml.bak engine_config.yaml
pm-config-deploy engine_config.yaml
```

Why the two checks fail differently

A hand-edited artifact is refused outright because a process cannot tell a malicious edit from a benign one, and either way the file no longer means what it claims to. A stale *source* file is only a warning because the deployed artifact is still perfectly valid — you just have an edit sitting uncommitted next to it. `pm-config-deploy` is the only fix for either case.

:material-checkbox-blank-outline: **Checkpoint:** you have seen both the content-digest error and the source-staleness warning fire for real, and can state which of the two blocks startup and which only warns.

3.16 Exercise 13: Weekday / Weekend / Holiday Scheduling

Exercise 2's `--sessions-enabled` config (and your hand-written one) used the built-in default schedule — the same five times every day, Monday through Friday, with Saturday, Sunday and bank holidays all `CLOSED`. Real exchanges are rarely that uniform: some run a short Friday, some trade weekends, and all of them close (or shorten hours) on bank holidays that fall on an otherwise-ordinary weekday. `pm-config-gen` can generate all three variations, and this exercise walks through each. The generated files go to a separate `schedule.yaml`, so your hand-written `engine_config.yaml` — the one with reference prices that later chapters need — stays untouched.

Resolving which schedule applies to a given calendar day always happens in this order:

1. If the day is a bank holiday in the configured `locale.country`, the `holidays:` block applies — or `CLOSED`, if no `--holidays` block was generated.
2. Otherwise, the day's own entry applies: `mon..fri` (from `weekdays:`, unless overridden by `--weekend`) or `sat/sun` (from `weekend:`, only if `--weekend` was passed).

Any day left with no resolved entry — including Saturday and Sunday by default — is `CLOSED` all day, exactly like today's config already leaves them.

Add a weekend session.

By default `pm-config-gen` leaves Saturday and Sunday `CLOSED`. Pass `--weekend` to give them their own (shorter) session:

```
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --symbol-opts AAPL:tick_decimals=2 \
  --symbol-opts MSFT:tick_decimals=2 \
  --symbol-opts TSLA:tick_decimals=2 \
  --participants TRADER01:TRADER TRADER02:TRADER GW_ADMIN:ADMIN ADMIN_CLI:ADMIN \
  --static-band 0.10 \
  --dynamic-band 0.05 \
  --sessions-enabled \
  --weekend \
  --weekend-pre-open 10:00 \
  --weekend-opening-auction 10:25 \
  --weekend-continuous 10:30 \
  --weekend-closing-auction 14:00 \
  --weekend-closing-end 14:05 \
  --output schedule.yaml --force
```

Inspect the generated `schedule:` block — Saturday and Sunday now carry their own five transition times instead of being absent.

Add a holiday session, and set the country.

Pass `--holidays` to give bank holidays a schedule of their own instead of `CLOSED` (a common real-world case is a reduced half-day rather than a full close). This command is the previous one plus the `--holidays…` and `--country` flags, so the file it writes has all three features:

```
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --symbol-opts AAPL:tick_decimals=2 \
  --symbol-opts MSFT:tick_decimals=2 \
  --symbol-opts TSLA:tick_decimals=2 \
  --participants TRADER01:TRADER TRADER02:TRADER GW_ADMIN:ADMIN ADMIN_CLI:ADMIN \
  --static-band 0.10 \
  --dynamic-band 0.05 \
  --sessions-enabled \
  --weekend \
  --weekend-pre-open 10:00 \
  --weekend-opening-auction 10:25 \
  --weekend-continuous 10:30 \
  --weekend-closing-auction 14:00 \
  --weekend-closing-end 14:05 \
  --holidays \
  --holidays-pre-open 10:00 \
  --holidays-opening-auction 10:25 \
  --holidays-continuous 10:30 \
  --holidays-closing-auction 13:00 \
  --holidays-closing-end 13:05 \
  --country DE \
  --output schedule.yaml --force
```

`--country` is what tells `pm-scheduler` *which* calendar of bank holidays to check a date against before it decides whether the `holidays:` block even applies — it generates the `locale.country` key in the config's `locale:` block. It accepts a country name (`"Sweden"`) or an ISO 3166-1 alpha-2 code (`"DE"`), and it does **only** that — it has no effect on the session timezone (that is the sibling `locale.session_timezone` key, generated by `--session-timezone`, out of scope here). Get it wrong and the consequence is silent, not a crash: an unrecognised value makes `pm-cverifier` raise a `M026` warning, and at runtime `pm-scheduler` logs a warning and falls back to `"Sweden"` — so a Germany-based deployment left at the default, or given a typo'd country name, will silently follow the Swedish holiday calendar instead, trading straight through a local bank holiday it should have been closed for (or vice versa). There is no separate validation step for this — run `pm-cverifier` (Exercise 3) after any change to `locale.country` or the schedule and read its output, don't just deploy.

Verify the country was picked up. The summary should read `Sessions enabled — 7 of 7 days scheduled, holidays overridden`, and there should be no `M026` (the `M032` and `C001` warnings are the ones you already know, and this file has no reference prices, as in Exercise 2):

```
pm-cverifier schedule.yaml
```

Deploy it:

```
pm-config-deploy schedule.yaml
```

Restart `pm-engine` (`Ctrl-C`, then start it again) so that it reads the new artifact, restart `pm-admin` (it was disconnected by the restart), and ask the operator console for the schedule the engine is really running:

```
[GW_ADMIN|ADMIN]> SCHEDULE
```

`SCHEDULE` prints the weekly table — five transition times for each of Mon–Sun and a final `Holidays` column. Saturday and Sunday now read 10:00 / 10:25 / 10:30 / 14:00 / 14:05 and the `Holidays` column 10:00 / 10:25 / 10:30 / 13:00 / 13:05. (The trader console's `SESSION` only reports the current phase, not the timetable.)

When you have seen it, put your training configuration back and restart the engine, because `schedule.yaml` has no reference prices and Chapter 11 needs them:

```
pm-config-deploy engine_config.yaml
```

Restart `pm-engine` once more (`Ctrl-C`, then `pm-engine`), restart your trader and operator consoles, and reopen the market with `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS`, as at the end of Exercise 10.

weekend: and sat:/sun: are mutually exclusive

`--weekend` generates a single shared block for both days. There is no `pm-config-gen` flag for giving Saturday and Sunday *different* hours — that requires hand-editing the generated `sat:/sun:` keys directly in `engine_config.yaml` (see Session Scheduling — Configuring the schedule). A config that sets both `weekend:` and an individual `sat:/sun:` block is rejected at load — they say two different things about the same day. :material-checkbox-blank-outline: **Checkpoint:** generated config has a `weekend:` block, a `holidays:` block, and `locale.country: DE`; `pm-cverifier` reports no `M026` warning; `SCHEDULE` in `pm-admin` shows the weekend and `Holidays` columns filled in; and your hand-written `engine_config.yaml` is the deployed configuration again.

3.17 Summary

You now have:

- A configuration file defining 3 symbols and 3 gateways.

- A repeatable verifier workflow (`pm-cverifier`) to catch config problems before startup.
- A compiled, deployed configuration (`pm-config-deploy`) that `pm-engine` and `pm-scheduler` actually read.
- A running engine, scheduler, and at least one trader gateway.
- Confirmation that all symbols accept orders.
- Firsthand experience of the artifact's content-digest error and source-staleness warning, and which fix (`pm-config-deploy`) clears both.
- A schedule with weekend trading, a bank-holiday session, and the correct `locale.country` for its holiday calendar — and the ability to tell, from `pm-cverifier`, when `locale.country` isn't one.

3.18 Reflection

Why does the engine, scheduler, and each gateway all run as **separate processes** connected over ZMQ sockets, instead of one monolithic program? What would you lose (or gain) operationally if the scheduler crashed while the engine kept running?

`pm-engine` and `pm-scheduler` never accept a config path, and neither reads `engine_config.yaml` directly. What problem does forcing every process through one compiled artifact (`pm-config-deploy`) solve that letting each process parse its own copy of the YAML would not?

Exercise 12's two checks fail at different severities — one refuses to start, the other only warns. Why is a hand-edited *artifact* treated as more serious than a stale *source* file, given that both mean the running configuration is not what `engine_config.yaml` currently says?

An unrecognised `locale.country` doesn't stop the config from deploying — it only warns, both from `pm-cverifier` (M026) and from `pm-scheduler` at runtime, then silently falls back to Sweden's holiday calendar. Why might a design choose "warn and fall back" here instead of refusing to deploy, the way an invalid or incomplete schedule block does? What's different about getting `locale.country` wrong versus getting the schedule times wrong?

3.19 Further Reading

- The Configuration Workflow
- Configuration — Verifying the Deployed Artifact
- Config Verifier (`pm-cverifier`)
- Session Scheduling and Auctions
- Session Scheduling — Bank holidays and weekends
- Running the Engine
- Gateways and How You Connect
- ALF Console (pm-alf-console)
- Message Types (system.symbols)

Next: 02 — Setting Up Market-Maker Liquidity

CHAPTER 4

Setting Up Market-Maker Liquidity

4.1 Objective

Configure market-maker gateways and use manual `QUOTE` commands to provide two-sided liquidity for all three symbols. You will also compare this manual workflow with `pm-mm-bot`, which automates the same lifecycle — including how to quote several symbols from one process on different terms each, and how to drive the whole desk from a committed config file instead of a long CLI invocation.

Background reading

- Market Making - Market-Maker Bot

4.2 Prerequisites

- Chapters 00–01 completed.
- `pm-engine` running and the market open in `CONTINUOUS` (chapter 01, "Open the market for the chapters ahead").
- The operator console connected (for book checks), and your working folder as your current directory (`cd /data/training` on the container route).

Which exercises chapter 03 needs

Chapter 03 trades against the **manual quotes** you create in Exercises 1–5. Exercises 6–8 replace them with market-maker bots and change the configuration, so they are **optional for now**: do them after chapter 03, or use "Returning to the manual-quote state" at the end of this chapter to get back.

4.3 Background

A market maker posts simultaneous buy (bid) and sell (ask) prices. Without one, the order book is empty and no trader can get an immediate fill.

Manual first, automation second

`pm-mm-bot` is available, but this chapter starts with manual `pm-alf-console` + `QUOTE` so you can see quote lifecycle and operator tools directly before using automation.

4.4 Exercise 1: Add MM Gateways to Configuration

Extend your `engine_config.yaml` `participants` list:

```
participants:
  # ... existing TRADER01, TRADER02, GW_ADMIN entries ...

  - id: MM_AAPL_01
    description: "AAPL market-maker"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL

  - id: MM_MSFT_01
    description: "MSFT market-maker"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL

  - id: MM_TSLA_01
    description: "TSLA market-maker"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
```

Declaring a `MARKET_MAKER` gateway obliges you to seed a quote for every symbol it makes a market in. Add a `market_maker_quotes` entry under each symbol naming its market maker:

```
symbols:
  AAPL:
    tick_decimals: 2
    last_buy_price: 150.00
    last_sell_price: 150.00
    market_maker_quotes:
      - gateway_id: MM_AAPL_01
        bid_price: 149.95
        ask_price: 150.05
        bid_qty: 500
        ask_qty: 500
  MSFT:
    tick_decimals: 2
    last_buy_price: 420.00
    last_sell_price: 420.00
    market_maker_quotes:
      - gateway_id: MM_MSFT_01
        bid_price: 419.90
        ask_price: 420.10
        bid_qty: 300
        ask_qty: 300
  TSLA:
    tick_decimals: 2
    last_buy_price: 250.00
    last_sell_price: 250.00
    market_maker_quotes:
      - gateway_id: MM_TSLA_01
        bid_price: 249.75
        ask_price: 250.25
        bid_qty: 200
        ask_qty: 200
```

These are the prices and sizes the manual `QUOTE` commands below use too, so the quote you send replaces a seed that looks exactly the same.

Skip the seeds and the deploy is refused

Without them the verifier raises **M001 (ERROR)** — *"Symbol 'AAPL' has no market_maker_quotes entry for MARKET_MAKER gateway(s) MM_AAPL_01, MM_MSFT_01, MM_TSLA_01"* — and `pm-config-deploy` refuses to install a configuration with any error. This is the verifier doing its job: a market maker with nothing to quote is a configuration mistake, not a runtime one.

pm-config-gen can generate the seeds instead of hand-writing them

`pm-config-gen` (Chapter 01) will emit a `market_maker_quotes` stub for every `MARKET_MAKER` gateway automatically — you don't have to hand-write the block above. Two ways to use it:

```
# Stub only: emits gateway_id/bid_qty/ask_qty/tif, but bid_price and
# ask_price are left as `null` for you to fill in by hand
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --participants TRADER01 TRADER02 GW_ADMIN:ADMIN ADMIN_CLI:ADMIN \
    MM_AAPL_01:MARKET_MAKER MM_MSFT_01:MARKET_MAKER MM_TSLA_01:MARKET_MAKER \
  --enforce-mm-obligations \
  --output mm-stub.yaml --force

# Fully seeded: also fills bid_price/ask_price from a random midpoint in
# the given range, rounded to each symbol's tick grid
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --participants TRADER01 TRADER02 GW_ADMIN:ADMIN ADMIN_CLI:ADMIN \
    MM_AAPL_01:MARKET_MAKER MM_MSFT_01:MARKET_MAKER MM_TSLA_01:MARKET_MAKER \
  --enforce-mm-obligations \
  --seed-mm-mid-range 100:400 \
  --output mm-seeded.yaml --force
```

These write to their own files on purpose: generated files have no reference prices, so writing them over your hand-written `engine_config.yaml` would silently lose what later chapters need.

The first form is a trap worth knowing about: `pm-cverifier` will report **0 errors** on the stub — `market_maker_quotes` entries exist, so M001 doesn't fire, and the required keys are all present (just `null`), so the schema check doesn't fire either. The verifier cannot see that `null` is unusable. It's `pm-config-deploy` that catches it, at compile time, with `Symbol 'AAPL': market_maker_quotes[0] is invalid` — because loading the artifact tries `float(None)`. `pm-config-gen` itself warns you about this up front with `[HINT] Fill all market_maker_quotes bid_price/ask_price values before starting pm-engine`, right after it writes the file — that hint is the moment to either fill the two prices in by hand or rerun with `--seed-mm-mid-range` instead.

If you never want seeded quotes at all (a deliberately empty book for a symbol), `--no-mm-seed-quotes` sets `require_mm_seed_quotes: false` and suppresses M001 for that config entirely — no stub is emitted and no price is expected.

Check, deploy, then restart the engine — editing the YAML alone changes nothing, because every process reads the compiled artifact:

```
pm-cverifier engine_config.yaml       # expect 0 errors (the M032 warning is fine)
pm-config-deploy engine_config.yaml
cp engine_config.yaml engine_config.ex1.yaml   # keep a copy of this state
```

The copy is for the optional Exercise 8, which changes `engine_config.yaml`.

Then restart `pm-engine` to pick up the new gateways (`Ctrl-C`, then `pm-engine --verbose`). A restarted engine starts in `CLOSED` again. Your consoles log out when they detect the lost connection; start `pm-admin` and your trader console again, then reopen the market from the operator console (`SESSION|STATE=PRE_OPEN`, then `SESSION|STATE=CONTINUOUS`).

Now look at the books before any market maker has connected:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

They are **already two-sided**: at start-up the engine places each seed quote from `market_maker_quotes` on the market maker's behalf (the engine log shows `MM quote SEED-MM_AAPL_01-AAPL-1 …`). The seeds are what keeps the books from being empty the moment the exchange starts, and the exercises below take them over.

:material-checkbox-blank-outline: **Checkpoint:** `pm-cverifier` reports 0 errors, the deploy succeeds, and the engine logs show 6 gateways loaded.

4.5 Exercise 2: Connect the AAPL Market Maker

In a new terminal:

```
pm-alf-console --id MM_AAPL_01
```

At the prompt, submit a two-sided quote:

```
[MM_AAPL_01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=AAPL-MM-001
```

Expected output should include a quote acknowledgement and active status. Because the engine already placed a seed quote for this market maker, your `QUOTE` replaces it, and you see the old one go first:

```
[<time>] CANCELLED <seed bid leg id> reason=QUOTE_REPLACED
[<time>] CANCELLED <seed ask leg id> reason=QUOTE_REPLACED
[<time>] QUOTE CANCELLED  SEED-MM_AAPL_01-AAPL-1  Replaced by new quote
[<time>] QUOTE ACK  AAPL-MM-001  bid=<order id> ask=<order id>
[<time>] QUOTE ACTIVE  AAPL-MM-001
```

A market maker holds at most one active quote per symbol; sending another always replaces the previous one.

:material-checkbox-blank-outline: **Checkpoint:** AAPL quote acknowledged and active.

4.6 Exercise 3: Quote MSFT and TSLA

Open one terminal per MM gateway:

```
pm-alf-console --id MM_MSFT_01
pm-alf-console --id MM_TSLA_01
```

Submit quotes:

```
[MM_MSFT_01]> QUOTE|SYM=MSFT|BID=419.90|ASK=420.10|BID_QTY=300|ASK_QTY=300|TIF=DAY|QUOTE_ID=MSFT-MM-001
[MM_TSLA_01]> QUOTE|SYM=TSLA|BID=249.75|ASK=250.25|BID_QTY=200|ASK_QTY=200|TIF=DAY|QUOTE_ID=TSLA-MM-001
```

:material-checkbox-blank-outline: **Checkpoint:** all three market makers report active quotes.

4.7 Exercise 4: Verify Liquidity from the Operator Console

From the **operator console** (`pm-admin`) — a trader cannot inspect the whole book:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

You should see a two-sided book with the MM's bid and ask (the prices are the same as the seed's, so the book looks the same as before — what changed is who owns the quote). Repeat for MSFT and TSLA.

```
[GW_ADMIN|ADMIN]> BOOK|SYM=MSFT
[GW_ADMIN|ADMIN]> BOOK|SYM=TSLA
```

:material-checkbox-blank-outline: **Checkpoint:** all three books show two-sided liquidity.

4.8 Exercise 5: Inspect Quote State with QLEGS

From each market-maker gateway, inspect quote legs:

```
[MM_AAPL_01]> QLEGS|SYM=AAPL|SHOW=ALL
[MM_MSFT_01]> QLEGS|SYM=MSFT|SHOW=ALL
[MM_TSLA_01]> QLEGS|SYM=TSLA|SHOW=ALL
```

`QLEGS` shows the bid and ask leg order IDs, prices, remaining quantities, and fill flags. With `SHOW=ALL` you also see the legs of the seed quote that your `QUOTE` replaced, marked `CANCELLED`; `SHOW=ACTIVE` hides them. This is the operator view that helps you reconcile fills after restart or partial execution.

Interpretation guide:

- `Rem` is open quantity still resting in the book.
- `Filled` is already executed quantity on that leg.
- `Leg status` is one of `NEW`, `PARTIAL`, `FILLED`, `CANCELLED`, `EXPIRED`, or `PENDING` — a resting, untouched leg shows `NEW`, not `RESTING`.

:material-checkbox-blank-outline: **Checkpoint:** QLEGS shows both quote legs for each symbol.

4.9 Exercise 6: Run the Equivalent Bot Workflow (Optional)

The manual quote sequence above can be automated with one bot per symbol — independent processes, each free to crash, restart, or be tuned without touching the others. (Exercise 8 below shows the alternative: one process covering several symbols at once, each on its own terms.)

First, **exit the three market-maker consoles** (`EXIT` in each). A bot connects under the same gateway ID as the console (`MM_AAPL_01`, …) and the engine allows only one connection per ID, so while a console is open the bot is refused with `auth rejected: Gateway already connected: MM_AAPL_01`. Leaving the consoles also cancels their quotes (`CANCEL_QUOTES_ONLY` in the configuration), so the books are left without a market maker and the bots need a starting price. Give each one a range to start in with `--initial_min` and `--initial_max`; it draws its first mid-price at random inside it. Then, in three terminals:

```
pm-mm-bot --symbol AAPL --gap 0.10 --qty 500 --initial_min 149 --initial_max 151
pm-mm-bot --symbol MSFT --gap 0.16 --qty 300 --initial_min 419 --initial_max 421
pm-mm-bot --symbol TSLA --gap 0.20 --qty 200 --initial_min 249 --initial_max 251
```

Add `--verbose` to any of these to follow what the bot does; by default it prints only warnings and errors. The quotes now sit around the random mid-price the bot chose, not at the prices of Exercises 2–3: check with `BOOK|SYM=AAPL` in the operator console.

The bot connects using the gateway ID `MM_<SYMBOL>_<id-suffix>` (default suffix `01`), quotes symmetrically around the current mid-price at `--gap`, reissues after fills, and reprices when the mid drifts. It also runs a `QBOOT` request at startup so it can adopt an already-active quote of its own gateway — the engine's seed quote, for instance, when no console is holding the ID — instead of creating a duplicate, and periodically re-runs `QLEGS` to reconcile leg state in case an engine reply was ever dropped.

The 20-tick limit on –gap

The gaps above are not arbitrary. Every symbol has a maximum quoted spread, `mm_max_spread_ticks`, which is **20 ticks** unless you configure otherwise — the `Max Spread` column of `SYMBOLS` in Chapter 01. At `tick_decimals: 2` that is 0.20, and a bot started with a wider `--gap` refuses to quote that symbol:

```
excluded from quoting: --gap exceeds mm_max_spread_ticks obligation.
```

Gateway ID must already exist in your config

`pm-mm-bot` does not create a gateway — it connects under the ID it computes (`MM_AAPL_01`, `MM_MSFT_01`, `MM_TSLA_01` by default) and expects that ID to already be present in `engine_config.yaml` from Exercise 1. If your config used different gateway IDs, either rename them to match this pattern or pass `--id-suffix` to the bot so the computed ID lines up. A mismatch here causes the engine to reject the bot's connection.
Quick primer:

- `QBOOT` asks the engine whether a gateway+symbol already has an active quote slot (for example after a crash/restart).
- `QLEGS` reconciles leg-level state (order IDs, remaining, fills) so the bot can adopt or replace safely instead of duplicating quotes.

See the detailed walkthrough in 09 — Market Making, and 22 — Automation with CommandClient & MM Bot Tuning for the advanced runtime flags (bootstrap timeout, QLEGS reconciliation interval, and similar) that this chapter deliberately leaves out.
:material-checkbox-blank-outline: **Checkpoint:** explain what the bot automates compared with your manual `QUOTE` workflow.

4.10 Exercise 7: Drive the Bot from a Config File (Optional)

A bot's whole configuration can live in a version-controlled YAML file instead of a long command line. Create `mm_aapl.yaml`:

```yaml
version: 1

gateway:
  label: AAPL
  id_suffix: "01"

defaults:
  strategy: symmetric
  gap: 0.10
  qty: 500
  tif: DAY
  drift_ticks: 3

symbols:
  AAPL:
    initial_min: 149
    initial_max: 151
```

The `symbols:` block names AAPL and gives it a starting range (the bot needs a reference price on an empty book, as in Exercise 6). Run the bot from the file instead of typing the flags (exit any other process connected as `MM_AAPL_01` first):

```
pm-mm-bot --config mm_aapl.yaml --verbose
```

The file has four blocks, and the split between them is the thing worth learning:

Block	Holds	Can it differ per symbol?
`gateway:`	Gateway identity and the two engine socket addresses	**No** — one process, one gateway, one pair of sockets
`logging:`	Log level and destination	**No** — one process, one log stream
`defaults:`	Quoting settings applied to every symbol	These *are* the defaults
`symbols:`	Which symbols to quote, and each one's overrides	**Yes** — see Exercise 8

A symbol's block holds its overrides: here only the start range, so AAPL is quoted on the house defaults. A symbol with nothing to override is written `AAPL: {}` — the empty braces are required YAML, not decoration, because the value of a symbol key is its override block.

An explicit CLI flag still overrides the file, so you can keep one committed file per desk for a class and override a single value for a one-off run:

```
pm-mm-bot --config mm_aapl.yaml --gap 0.15
```

No symbol needs to appear on the CLI at all as long as the file's `symbols:` block names one. The bot fails fast with a usage error if *neither* the CLI nor the file provides a symbol, rather than silently picking a default.

Three pricing strategies ship today

`strategy: symmetric` (the default, so the line above is not strictly needed) quotes symmetrically around the tracked mid at `gap`. `strategy: inventory_skew` instead shifts both quotes to work an accumulated position back toward flat, and requires a `max_position:` alongside it — the net position at which the skew saturates. `strategy: passive` keeps its quotes behind other traders' orders instead of in front of them (see the passive strategy). Naming any other strategy is a startup failure, not a silent fallback.

A typo in the file fails fast, and tells you what you meant

An unknown key is rejected at startup rather than silently ignored, with a suggestion where there is an obvious one:

```
invalid config file: config file mm_aapl.yaml: unknown key 'gapp' under symbols.AAPL (did you mean 'gap'?)
```

Putting a real key in the wrong block gets its own message rather than "unknown key" — try moving `gap:` into `gateway:` and see what it says.

Quote the id_suffix

`id_suffix: 01` without quotes is the *number* 1 to YAML, which would make the gateway `MM_AAPL_1` rather than `MM_AAPL_01`. The loader refuses it outright rather than guessing; `"01"` is correct.
:material-checkbox-blank-outline: **Checkpoint:** you have started a bot from a config file alone, then overridden one value from the CLI and confirmed the CLI value won.

4.11 Exercise 8: One Bot, Three Symbols, Three Sets of Terms (Optional)

Exercise 6 ran three separate `pm-mm-bot` processes, one per symbol. But nothing on the engine side requires a `MARKET_MAKER` gateway to quote only one symbol: the engine's `QuoteIndex` keys every active quote by

`(gateway_id, symbol)` and tracks a *set* of such keys per gateway, so one gateway can hold AAPL's quote and MSFT's quote and TSLA's quote at the same time — and each of those symbols can be quoted on its own terms.

Stop the three bots from Exercise 6 if you ran them (Ctrl+C each — a bot cancels its own quote when it exits) and register one gateway that will quote all three symbols instead:

```
participants:
  - id: MM_TECH_01
    description: "AAPL+MSFT+TSLA market-maker"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
```

The seed rule is per symbol, not per gateway

`pm-cverifier`'s **M001** check fires for a symbol that has *no* `market_maker_quotes` entry at all; it does not check which gateway is meant to quote it. The seeds from Exercise 1 therefore already satisfy it, and the configuration verifies clean without a seed for `MM_TECH_01`. Add one anyway — a second item in each symbol's `market_maker_quotes` list, naming `MM_TECH_01` — because a seed is also what gives a bot something to start from (next paragraph). If you regenerate with `pm-config-gen`, pass `--participants ... MM_TECH_01:MARKET_MAKER` and it seeds all three symbols against it.
For AAPL the list becomes (and the same for MSFT and TSLA, with their own prices and sizes):

```
    market_maker_quotes:
      - gateway_id: MM_AAPL_01
        bid_price: 149.95
        ask_price: 150.05
        bid_qty: 500
        ask_qty: 500
      - gateway_id: MM_TECH_01
        bid_price: 149.95
        ask_price: 150.05
        bid_qty: 500
        ask_qty: 500
```

Check and deploy as before, restart the engine, **restart `pm-admin`** (the engine restart disconnected it, and `KILL` would answer `Gateway not connected: GW_ADMIN`) and reopen the market. The old market makers' seed quotes are still in the books, so clear them from the operator console and leave `MM_TECH_01` the only quoter:

```
[GW_ADMIN|ADMIN]> KILL|GW=MM_AAPL_01
[GW_ADMIN|ADMIN]> KILL|GW=MM_MSFT_01
[GW_ADMIN|ADMIN]> KILL|GW=MM_TSLA_01
```

Now start one bot covering all three symbols on identical terms:

```
pm-mm-bot --verbose --symbols AAPL,MSFT,TSLA --label TECH --gap 0.10 --qty 500
```

`--label TECH` is what makes the gateway ID `MM_TECH_01` instead of the default `MM_AAPL_MSFT_TSLA_01` the bot would otherwise derive by joining every symbol — useful once the symbol list is longer than a couple of entries.

Watch the log output. The startup lines report the gateway, the symbol list, and then **one line per symbol** with that symbol's resolved settings:

```
[MM_TECH_01] starting: gateway=MM_TECH_01 symbols=AAPL,MSFT,TSLA
[MM_TECH_01] [AAPL] strategy=symmetric gap=0.1 qty=500 tif=DAY drift_ticks=3
[MM_TECH_01] [MSFT] strategy=symmetric gap=0.1 qty=500 tif=DAY drift_ticks=3
[MM_TECH_01] [TSLA] strategy=symmetric gap=0.1 qty=500 tif=DAY drift_ticks=3
```

A little further down, each symbol reports `adopted existing quote SEED-MM_TECH_01-AAPL-2 …` (and the same for MSFT and TSLA): the bot found its own seed quote through `QBOOT` and took it over instead of quoting afresh. Every per-symbol decision afterwards (a quote sent, a fill, a reprice) is tagged `[AAPL]`, `[MSFT]`, or `[TSLA]` — one process, three independent quoting lifecycles.

A bot needs a reference price to start

Stop the bot with Ctrl+C and its quotes are cancelled — the books are empty again. Start it once more and it exits with `startup failed: no reference price available`. A bot prices around a reference: the book's mid, the last trade, a quote of its own gateway that `QBOOT` finds, or, failing all of those, a starting range you give it with `initial_min` / `initial_max` (`--initial_min` / `--initial_max` on the command line). The variants below all state a range per symbol so that they start on an empty book; see Bootstrap: starting with an empty book. The bot then draws its first mid at random inside the range, so the exact prices in your books will differ from run to run.

4.11.1 Giving each symbol its own terms

Quoting a volatile name and a liquid one with the same spread and size is rarely what you want. `--symbol` may be **repeated**, and every quoting flag after it applies to that symbol until the next `--symbol`:

```
pm-mm-bot --verbose --label TECH --qty 500 \
  --symbol AAPL --gap 0.10 --initial_min 149 --initial_max 151 \
  --symbol MSFT --gap 0.16 --qty 300 --initial_min 419 --initial_max 421 \
  --symbol TSLA --gap 0.20 --qty 200 --drift-ticks 5 --initial_min 249 --initial_max 251
```

Flags **before** the first `--symbol` (here `--label` and `--qty 500`) are the defaults for everything; flags **after** one apply to that symbol alone. The startup lines now differ per symbol — check them against what you typed:

```
[MM_TECH_01] [AAPL] strategy=symmetric gap=0.1 qty=500 tif=DAY drift_ticks=3
[MM_TECH_01] [MSFT] strategy=symmetric gap=0.16 qty=300 tif=DAY drift_ticks=3
[MM_TECH_01] [TSLA] strategy=symmetric gap=0.2 qty=200 tif=DAY drift_ticks=5
```

That command line is already at the edge of readable, which is exactly what the config file is for. The same desk as `mm_tech.yaml`:

```
version: 1

gateway:
  label: TECH
  id_suffix: "01"

defaults:
  strategy: symmetric
  gap: 0.10
  qty: 500
  tif: DAY
  drift_ticks: 3

symbols:
  AAPL:                    # house defaults, plus a starting range
    initial_min: 149
    initial_max: 151

  MSFT:                    # wider, smaller
    gap: 0.16
    qty: 300
    initial_min: 419
    initial_max: 421

  TSLA:                    # wider still, smaller still, less twitchy,
    gap: 0.20              # and working its inventory back toward flat
    qty: 200
    drift_ticks: 5
    strategy: inventory_skew
    max_position: 5000
    initial_min: 249
    initial_max: 251
```

```
pm-mm-bot --verbose --config mm_tech.yaml
```

Compare the startup lines with the CLI version above: the same three spreads and sizes, expressed as a file instead of a command line — plus a per-symbol `strategy` for TSLA that would have made the command line longer still.

Worked examples ship with the docs

`docs/examples/mm-bot/` contains `single-symbol.yaml`, `tech-desk.yaml` (a similar desk with every option commented), and `uniform-desk.yaml` showing the `symbols: [AAPL, MSFT, TSLA]` list shorthand for when no symbol needs its own block.

4.11.2 Which setting wins?

With both a file and a command line in play, each symbol's settings are resolved from the most specific source that supplies one:

Rank	Source	Example
1	The symbol's own CLI scope	`--symbol MSFT --gap 0.16`
2	The symbol's block in the config file	`symbols.MSFT.gap`
3	The global CLI scope	`--gap 0.18` before the first `--symbol`
4	The file's `defaults:` block	`defaults.gap`
5	The built-in default	`0.10`

Rank 2 sitting above rank 3 is the one to think about, and it is deliberate. Try both and read the startup lines:

```
# Widens only AAPL — the one symbol whose block asks for no gap of its own.
# MSFT keeps 0.16 and TSLA keeps 0.20, because their blocks named a gap.
pm-mm-bot --verbose --config mm_tech.yaml --gap 0.18

# Changes TSLA too, because this names it. (0.18 is narrower than its 0.20, but
# the point is that the symbol's own CLI scope wins.)
pm-mm-bot --verbose --config mm_tech.yaml --symbol TSLA --gap 0.18
```

A value someone deliberately wrote into a reviewed file for one symbol is not un-tuned by a throwaway gateway-wide flag. Overriding it is still one short command — you just have to say which symbol you mean.

Confirm the books from the operator console:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
[GW_ADMIN|ADMIN]> BOOK|SYM=MSFT
[GW_ADMIN|ADMIN]> BOOK|SYM=TSLA
```

The spreads should now differ visibly between the three symbols — 0.10, 0.16 and 0.20 — that is the per-symbol `gap` reaching the book. (If you adopted the seeds instead of starting from a range, the first spreads are the seeds' own; force the bot to quote on its own terms with `QCANCEL|GW=MM_TECH_01|SYM=AAPL` — the bot notices the cancel and reissues at its configured gap.)

A struggling symbol doesn't take the others down with it

If one symbol's `gap` violates its own `mm_max_spread_ticks` obligation (20 ticks unless configured), that symbol alone is excluded from quoting at startup while the rest of `MM_TECH_01`'s symbols keep quoting normally — the whole process only exits if *every* symbol fails. Try it: change TSLA's `gap` in `mm_tech.yaml` to `0.50` and restart the bot. Its log says

```
[MM_TECH_01] [TSLA] excluded from quoting: --gap exceeds mm_max_spread_ticks obligation (0.500000 > 20 * 0.01000
```

and AAPL and MSFT quote as before.

:material-checkbox-blank-outline: **Checkpoint:** one `pm-mm-bot` process is quoting all three symbols with three different spreads and sizes, its startup lines report each symbol's own settings, its log lines are tagged per symbol, and all three books show two-sided liquidity at visibly different spreads.

4.12 Returning to the manual-quote state

Chapter 03 needs the state of Exercises 1–5: `MM_AAPL_01` connected with the manual quote `AAPL-MM-001`. If you have since run the bots of Exercises 6–8 (or simply want a clean start), get back to it:

1. Stop every `pm-mm-bot` (`Ctrl-C`; a bot cancels its quotes on exit).
2. Restore the configuration from Exercise 1 and deploy it:

```
cp engine_config.ex1.yaml engine_config.yaml
pm-config-deploy engine_config.yaml
```

3. Restart `pm-engine`, then restart `pm-admin`, and reopen the market with `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS`.
4. Connect `pm-alf-console --id MM_AAPL_01` (and the MSFT and TSLA ones if you want them) and send the `QUOTE` commands of Exercises 2 and 3 again.
5. In `pm-admin`, `BOOK|SYM=AAPL` shows 500 bid at 149.95 and 500 ask at 150.05. Orders that a trader placed earlier (in chapter 03, say) survive the engine restart and are still in the book; clear them with `KILL|GW=TRADER01` (and `KILL|GW=TRADER02`) if you want the clean state.

4.13 Summary

You now have:

- Market-maker gateways configured for all symbols, seeded either by hand or with `pm-config-gen --seed-mm-mid-range`.
- Manual `QUOTE` liquidity in AAPL, MSFT, and TSLA.
- Familiarity with `QLEGS` as the quote-leg inspection tool.
- A clear picture of what `pm-mm-bot` automates, and how to drive it from either the CLI or a committed config file.
- Experience running one `pm-mm-bot` process across multiple symbols, giving each symbol its own spread, size and strategy, and seeing per-symbol failure isolation in action.
- A working understanding of which setting wins when a config file and a command line disagree.

4.14 Reflection

If no market maker were quoting a symbol, what would happen to a marketable order sent by a regular trader in Chapter 03? Why does the training guide insist you set up liquidity *before* any trading exercises rather than letting students discover an empty book on their own?

`pm-config-gen`'s stub `market_maker_quotes` entry (no `--seed-mm-mid-range`) passes `pm-cverifier` with 0 errors, yet still fails at `pm-config-deploy`. Why do you think the verifier and the compile-time loader disagree here — what would it take for the verifier to catch a `null` bid/ask price itself, and can you think of a reason the two checks are allowed to diverge like this rather than the verifier being made strict enough to catch everything `pm-config-deploy` would?

Exercise 8 put three symbols behind one gateway ID and one process, each with its own spread and size. Given that the engine places no limit on how many symbols a `MARKET_MAKER` gateway may quote, and that per-symbol tuning no longer requires per-symbol processes, what's actually left driving the choice between "one process, many symbols" and "one process per symbol" in a real deployment? Think about failure blast radius, whether the scenario needs the symbols to be *independently attributable* to different market makers, and process/operational overhead — is there a symbol count where you'd expect to switch from one pattern to the other?

The precedence table in Exercise 8 puts a per-symbol *file* value above a gateway-wide *CLI* flag, which

reverses the usual "the command line always wins" rule. Argue the other side: what would break, or become surprising, if `--gap 0.30` did override every symbol's `gap` including the ones a reviewed file had deliberately tuned? Which behaviour would you rather explain to someone at 09:29 on a trading morning?

4.15 Further Reading

- Market Making
- The Market-Maker Bot (`pm-mm-bot`)
- Market-Maker Bot — Per-symbol settings
- Market-Maker Bot CLI Reference
- Market-Maker Bot — Config File
- Market-Maker Bot — Precedence
- Market-Maker Bot — Per-symbol failure isolation
- ALF Console (pm-alf-console)
- ALF Protocol Reference
- Config Verifier (`pm-cverifier`)
- 22 — Automation with CommandClient & MM Bot Tuning

Next: 03 — The First Trade

CHAPTER 5

The First Trade

5.1 Objective

Execute your first buy and sell trades against the market-maker quotes and understand fill messages, clearing records, order IDs, and the order book lifecycle.

Background reading

- ALF Console - Order Types

5.2 Prerequisites

- Engine running from chapters 01–02, with the market open in `CONTINUOUS`.
- The manual AAPL quote from chapter 02 active: `MM_AAPL_01` bidding 149.95 and asking 150.05, 500 shares each side, and its console still open. (With `pm-mm-bot` quoting instead, the prices and sizes below will differ. If you ran the optional bot exercises 6–8 of chapter 02, use "Returning to the manual-quote state" at the end of that chapter first.)
- `TRADER01` gateway and the operator console open **and connected to the current engine**. If you restarted the engine since you started them, restart them too (see "After every engine restart" in the introduction).
- One spare terminal for the clearing process.

Check the starting point. In `pm-admin`, `BOOK|SYM=AAPL` should show exactly one bid, 500 at 149.95, and one ask, 500 at 150.05. Anything else (for example a 1-lot bid at 0.01) is left over from an earlier exercise: clear it with `KILL|GW=TRADER01` in `pm-admin` before you begin, and see chapter 01's "Clear the test orders".

5.3 Exercise 1: Start Clearing

In a new terminal, start the clearing process before you trade:

```
pm-clearing --print-every 1
```

`pm-clearing` subscribes to executed trades, updates per-gateway positions and P&L, and writes batched results to `clearing.db` (SQLite) in the data directory — not a CSV file. It starts silent and prints its

table after a trade has been processed (it works in batches, so allow it up to about twenty seconds). Use `pm-clearing-cli --format csv ...` afterward if you want a CSV export. `--print-every 1` makes it print its P&L summary after every trade; by default it prints only every 100 trades. Leave `pm-clearing` running while you work through this chapter.

Start recorders before the trading

`pm-clearing` only sees trades that happen while it is running. Started after a trade, it never learns about it, and its positions will be wrong from then on.
:material-checkbox-blank-outline: **Checkpoint:** clearing is running and waiting for trades.

5.4 Exercise 2: Buy at Market — Lift the Ask

From `TRADER01`, send a market buy order for AAPL:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

Expected output (the console's fill line reports the order ID, quantity, price and remaining/status — not the symbol or side, since you already know those from the order you just sent):

```
[<time>] FILL      <order_id>  qty=100 @150.05  remaining=0  [FILLED]
```

The order matched against the MM's ask at 150.05.

Now switch to the `pm-clearing` terminal. After a short delay it prints a P&L summary: `TRADER01` is long 100 AAPL at 150.05, and `MM_AAPL_01` is short 100 at the same price.

Look at the market maker's console too:

```
[<time>] CANCELLED <bid leg id> reason=QUOTE_LEG_FILLED
[<time>] QUOTE INACTIVE_ASK_FILLED  AAPL-MM-001
[<time>] FILL      <ask leg id>  qty=100 @150.05  remaining=400  [PARTIAL]
```

Chapter 02 configured `MM_AAPL_01` with `quote_refresh_policy: INACTIVATE_ON_ANY_FILL`: the moment either side of its quote is hit, the engine **pulls the other side** and marks the quote inactive, while the remaining 400 on the hit side keep resting. This protects a market maker from being hit again before it has had time to reprice — and it means AAPL now has no bid at all.

:material-checkbox-blank-outline: **Checkpoint:** you received a fill confirmation and can see the trade in `pm-clearing`.

5.5 Exercise 3: Sell at Market — Hit the Bid

With no bid in the book, a market sell would have nothing to trade against — try it and the order is accepted and immediately `CANCELLED`. So let the market maker re-quote first, as a real one would after a fill:

```
[MM_AAPL_01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=AAPL-MM-002
```

The new quote replaces the old one (its 400-share remainder is cancelled). Now sell:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=100
```

Expected output, in the same format as Exercise 2:

```
[<time>] FILL      <order_id>  qty=100 @149.95  remaining=0  [FILLED]
```

This time the market maker's **ask** is pulled, and 400 shares remain bid at 149.95.

:material-checkbox-blank-outline: **Checkpoint:** sell fill at the bid price, and you can explain why the market maker had to re-quote first.

5.6 Exercise 4: Place a Limit Buy Order

Place a buy order below the current bid — it should rest on the book:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=149.80|TIF=DAY
```

Expected: `ACK <order id>  order accepted`. The order rests, because its price is below the best ask (it is not marketable).

Verify in the **operator console** — a trader cannot see the whole book. (It reads the current book from the engine each time you ask, however long the console has been open.)

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

You should see two bids: the market maker's 400 at 149.95 on top, and your 200 at 149.80 below it — the book is sorted best price first.

:material-checkbox-blank-outline: **Checkpoint:** limit order visible in the book.

5.7 Exercise 5: Place a Limit Sell and Get a Fill

Now from `TRADER02` (open a second gateway if not already):

```
pm-alf-console --id TRADER02
```

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=450|PRICE=149.80|TIF=DAY
```

TRADER02 will sell 450 shares at 149.80 *or better*. The engine fills it against the best bids first — **price priority**:

- 400 shares at **149.95** against the market maker, a better price than TRADER02 asked for (this is *price improvement*);
- the last 50 shares at **149.80** against TRADER01.

TRADER01 receives a partial fill (200 → 150 remaining). TRADER02's console reports the whole order filled at the average price, `qty=450 @149.9333…` — (400 × 149.95 + 50 × 149.80) / 450.

:material-checkbox-blank-outline: **Checkpoint:** cross-gateway fill confirmed on both sides.

5.8 Exercise 6: Check Order State

From TRADER01, inspect the partially filled order:

```
[TRADER01]> ORDERS
```

Expected: the order from Exercise 4 appears with its quantity, the remaining quantity (`Qty` 200, `Rem` 150) and status `PARTIAL`. `ORDERS` lists the day's orders for this gateway, so the filled market orders of Exercises 2–3 appear in the list as well, with status `FILLED`.

:material-checkbox-blank-outline: **Checkpoint:** ORDERS shows partial fill state correctly.

Optional: If you have `pm-viewer` running on AAPL, you should see the bid size drop from 200 to 150 after the fill.

5.9 Exercise 7 (Optional): Observe Automatic Re-Quoting with pm-mm-bot

If you are running `pm-mm-bot` for AAPL (instead of manual quoting), your market order in Exercise 2 should trigger automatic re-quoting. Run `BOOK` again in the operator console:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

The MM should have a fresh two-sided quote (possibly at a slightly different mid if the trade moved the reference price).

:material-checkbox-blank-outline: **Checkpoint:** MM has re-quoted after being filled.

5.10 Key Concepts Learned

- **Market orders** execute immediately against the best available price.
- **Limit orders** rest on the book until a matching counterparty arrives.
- **Fills** generate fill messages to both buyer and seller.
- **Clearing** consumes executed trades and turns them into positions and P&L.
- **Partial fills** leave the remainder resting.
- **Order IDs** are used for AMEND and CANCEL operations.

5.11 Reflection

Why does a MARKET order carry no `PRICE` field at all, while a LIMIT order requires one? What risk would a MARKET order expose you to in a thin book that a LIMIT order would protect you from?

5.12 Further Reading

- Your First Trade
- The Order Book
- ALF Console (pm-alf-console)

Next: 04 — Amending Orders

Part II

Trading mechanics

Chapters 04–08. Amend and cancel orders, use every order type and time in force, and run an auction by hand.

CHAPTER 6

Amending Orders

6.1 Objective

Learn to modify resting orders — change price, quantity, or both — and understand how amendments affect queue priority.

Background reading

- Order Amendment (AMEND) - ALF Console

6.2 Prerequisites

- Exchange running with two-sided liquidity from previous chapters (manual MM gateways or `pm-mm-bot`).
- `TRADER01`, `TRADER02`, the operator console and the `MM_AAPL_01` console (Exercise 7 uses it) connected to the current engine — restart any that were open before the last engine restart.
- `TRADER01` connected with at least one resting limit order.

6.3 Exercise 1: Place a Resting Order to Amend

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=300|PRICE=419.50|TIF=DAY|TAG=AMD-ORDER-001
```

Note the `order_id` returned.

`TAG` identifies the order and is echoed on later lifecycle events. `RTAG` is different: it identifies one amend request against that order.

`AMEND` accepts an optional `RTAG=<request-tag>`. The tag identifies this one amend request and is echoed on the `AMENDED` response or rejected ACK, which is useful when several changes are outstanding for the same order.

:material-checkbox-blank-outline: **Checkpoint:** order resting; `ORDERS` confirms qty=300, price=419.50.

6.4 Exercise 2: Amend Quantity Down

Reduce the order to 200 shares:

```
[TRADER01]> AMEND|ID=<order_id>|QTY=200|RTAG=AMD-DOWN-001
```

Expected: `AMENDED <order_id>  price=419.5 qty=200 remaining=200 rtag=AMD-DOWN-001`. There is no `(priority reset)` note, because priority was kept.

Priority preserved

Reducing quantity does **not** lose time priority — your order keeps its place in the queue.
Why: you are not jumping ahead of anyone; you are only reducing your own claim at the same price level.
:material-checkbox-blank-outline: **Checkpoint:** `ORDERS` shows qty=200, same price.

6.5 Exercise 3: Amend Price

Move the order to a more aggressive price:

```
[TRADER01]> AMEND|ID=<order_id>|PRICE=419.70|RTAG=AMD-PRICE-001
```

Expected: `AMENDED <order_id>  price=419.7 qty=200 remaining=200 (priority reset) rtag=AMD-PRICE-001`.

Priority lost

A price change **always** loses time priority — the order moves to the back of the queue at the new price level.
Why: a new price is treated as a new offer, so queue fairness requires re-entering at the back.

What 'time priority' actually means: arrival_seq, not timestamp

Queue order is decided by **`arrival_seq`**, a counter the *engine* assigns the moment it admits an order or a re-priced/re-queued amendment. It is **not** the `timestamp` field the client sends with the order — that field just records when the client says it built the order, and the book never uses it to decide who goes first. "Losing priority" on this page always means "the engine assigned a new `arrival_seq`," never "the timestamp changed." See Order Types — Priority Rules for the full explanation.
:material-checkbox-blank-outline: **Checkpoint:** `ORDERS` shows price=419.70.

6.6 Exercise 4: Amend Both Price and Quantity

```
[TRADER01]> AMEND|ID=<order_id>|PRICE=419.60|QTY=150|RTAG=AMD-BOTH-001
```

:material-checkbox-blank-outline: **Checkpoint:** both fields updated in one command.

6.7 Exercise 5: Attempt an Invalid Amendment

Try setting quantity to zero:

```
[TRADER01]> AMEND|ID=<order_id>|QTY=0|RTAG=AMD-BAD-001
```

Expected: `REJECTED … code=QTY_OUT_OF_RANGE rtag=AMD-BAD-001  Quantity must be positive`. The rejection carries a stable code (`REJECT_CODE` on the wire) as well as the text reason, and echoes your `rtag`.

Try amending a non-existent order:

```
[TRADER01]> AMEND|ID=INVALID123|PRICE=100.00|RTAG=AMD-MISSING-001
```

Expected: `REJECTED INVALID123 code=ORDER_NOT_FOUND rtag=AMD-MISSING-001  Order not found.`

:material-checkbox-blank-outline: **Checkpoint:** both invalid amendments rejected with clear errors.

6.8 Exercise 6: Amend After Partial Fill

To amend a partially filled order you first need one. TSLA is a good symbol for it, as long as you control exactly what is resting there. From the operator console clear it (this cancels every order and quote on TSLA, and stop any `pm-mm-bot` quoting it), then have `TRADER02` offer 200 shares:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=TSLA
[TRADER02]> NEW|SYM=TSLA|SIDE=SELL|TYPE=LIMIT|QTY=200|PRICE=250.00|TIF=DAY
```

1. From `TRADER01`, place a buy for 500 at the same price. Only 200 shares are on offer, so it fills 200 and the other 300 rest:

   ```
   [TRADER01]> NEW|SYM=TSLA|SIDE=BUY|TYPE=LIMIT|QTY=500|PRICE=250.00|TIF=DAY
   ```

   Expected: `FILL … qty=200 @250.0  remaining=300  [PARTIAL]`. Note the order ID of this buy.

2. Amend it. `QTY=` in an `AMEND` is the order's new **total** quantity, not the remainder, and 200 of it are already filled. Try to cut it back to the filled amount, or below it:

   ```
   [TRADER01]> AMEND|ID=<order_id>|QTY=200|RTAG=AMD-PARTIAL-001
   [TRADER01]> AMEND|ID=<order_id>|QTY=150|RTAG=AMD-PARTIAL-002
   ```

   Both are rejected with `code=QTY_OUT_OF_RANGE  New quantity must exceed already-filled quantity.`

3. Now raise the total to 700, which leaves 500 shares resting (700 − 200 filled):

   ```
   [TRADER01]> AMEND|ID=<order_id>|QTY=700|RTAG=AMD-PARTIAL-003
   ```

   Expected: `AMENDED … price=250.0 qty=700 remaining=500 (priority reset).`

Note

The new qty must be **strictly greater than** the already-filled quantity. Setting it equal to the filled quantity is rejected — use `CANCEL` if you want the remainder gone. You cannot amend below what has already been executed.
:material-checkbox-blank-outline: **Checkpoint:** the amendment to 700 is accepted on the partially filled order, and the two amendments at or below the filled quantity are rejected.

6.9 Exercise 7: See Queue Priority Change

The rules below are easy to state and easy to disbelieve. This exercise makes them visible, using two traders queued at the same price.

Step 0 — start from an empty book. Anything already resting on AAPL at a better price would trade first and spoil the experiment, so clear the symbol from the operator console (this cancels every order and quote on it) and stop any `pm-mm-bot` quoting AAPL:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
```

The seller in this exercise is the market maker's console, `MM_AAPL_01`. A market maker may send ordinary orders too, and using a third participant keeps either trader from selling into its own bid — a *self-trade*, which real exchanges prevent.

Step 1 — build a queue. From `TRADER01`, then `TRADER02`, place the same buy at the same price. `TRADER01` is now *ahead* in the queue because it arrived first:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.50|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.50|TIF=DAY
```

Confirm both are resting at 149.50 in the operator console:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

Step 2 — reduce quantity and prove priority survives. From `TRADER01`, amend *down* to 50:

```
[TRADER01]> AMEND|ID=<TRADER01 order id>|QTY=50|RTAG=AMD-KEEP-001
```

Now sell 50 into the bid:

```
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=50|PRICE=149.50|TIF=DAY
```

Because `TRADER01` kept its place, `TRADER01` is filled and `TRADER02` is untouched. Check who filled with `ORDERS` on each trader console.

Step 3 — increase quantity and prove priority is lost. Clear the book again so step 2's leftovers (TRADER02's untouched bid is still there, and already first in line) cannot decide the outcome, then rebuild the queue exactly as in step 1:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.50|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.50|TIF=DAY
```

Amend `TRADER01` *up*. The console confirms with `(priority reset)`:

```
[TRADER01]> AMEND|ID=<TRADER01 order id>|QTY=200|RTAG=AMD-LOSE-001
```

Sell 100 into the bid again:

```
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.50|TIF=DAY
```

This time `TRADER02` fills, because `TRADER01`'s increase sent it to the back of the queue at that price.

:material-checkbox-blank-outline: **Checkpoint:** in step 2 `TRADER01` filled; in step 3 `TRADER02` filled. You have now *observed* the rule in the table below rather than taking it on trust.

Use the full order ID

`AMEND` and `CANCEL` need the **complete** 32-character order ID. The acknowledgement line shows it in full; in a narrow terminal the `ORDERS` table shortens the ID column, so widen the window or copy the ID from the acknowledgement.

Use one RTAG per request

`RTAG` is not the order ID. It is a request label, so use a fresh value for each amend you want to correlate in logs, terminal output, or tests.

6.10 Key Rules

Change	Priority Impact	Why
Quantity down	Priority preserved (`arrival_seq` unchanged)	You are asking for less; nobody behind you is disadvantaged
Quantity up	Priority lost (new `arrival_seq`)	The extra quantity never queued — keeping your slot would jump it ahead of orders that arrived earlier
Price change (any direction)	Priority lost (new `arrival_seq`)	A different price is a different queue
Both price and qty	Priority lost (new `arrival_seq`)	As above

Reminder

"Priority" throughout this table is governed by the engine-assigned `arrival_seq`, never by the order's client-supplied `timestamp` field.

6.11 Reflection

Why does reducing quantity preserve time priority, but increasing it does not? What unfair advantage would a trader gain if increasing quantity kept their original priority in a busy book?

6.12 Further Reading

- ALF Console (pm-alf-console)
- ALF Protocol — AMEND

Next: 05 — Order Types Deep Dive

CHAPTER 7

Order Types Deep Dive

7.1 Objective

Explore every order type beyond basic LIMIT and MARKET — STOP, STOP_LIMIT, FOK, IOC, ICEBERG, and TRAILING_STOP — through practical exercises.

Background reading

- Order Types

7.2 Prerequisites

- Chapters 01–04 completed.
- `TRADER01`, `TRADER02`, the operator console and the `MM_AAPL_01` console connected to the current engine (Exercise 5 also uses `MM_TSLA_01`).
- The MSFT market-maker quote from chapter 02 still resting (Exercise 4 trades against its 300-share ask at 420.10). If `BOOK|SYM=MSFT` shows otherwise, send it again from `MM_MSFT_01`: `QUOTE|SYM=MSFT|BID=419.90|AS`

7.3 Deterministic Trigger Setup

Stop and stop-limit orders only trigger when the market actually **trades** through their trigger price — quotes alone never trigger them. To make the exercises below behave exactly as described, build the AAPL book yourself before each one, with the market maker's console placing ordinary limit orders:

1. Clear AAPL from the operator console (this cancels every order and quote on it), and stop any `pm-mm-bot` quoting AAPL:

   ```
   [GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
   ```

2. Give the book two bids, one above and one below the trigger level you will use (149.50):

   ```
   [MM_AAPL_01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.60|TIF=DAY
   [MM_AAPL_01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=149.30|TIF=DAY
   ```

3. Make a trade **above** the trigger, so the last traded price is 149.60 and a sell stop at 149.50 is not

triggered the moment you place it:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.60|TIF=DAY
```

Now only the 200-share bid at 149.30 is left in the book.

7.4 Exercise 1: Stop Order

A stop order becomes a market order when the trigger price is reached.

After the setup above, place a stop-sell — the order a trader holding AAPL would use to limit the loss if the price falls:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP|QTY=100|STOP=149.50|TIF=DAY
```

The order is acknowledged and then lies dormant: it is not in the visible book. Now let the market fall through 149.50 — `TRADER02` sells into the 149.30 bid:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.30|TIF=DAY
```

That trade at 149.30 is at or below the stop, so the stop triggers and becomes a market sell, which fills against the remaining 100 bid at 149.30:

```
[<time>] FILL      <order_id>  qty=100 @149.3  remaining=0  [FILLED]
```

Interpretation: 149.50 was the **trigger**, not the execution price. A stop guarantees that you get out, not at what price — in a falling market the fill can be well below the stop.

:material-checkbox-blank-outline: **Checkpoint:** the stop triggered on a trade below 149.50 and filled at 149.30.

7.5 Exercise 2: Stop-Limit Order

Like a stop, but becomes a limit order (not a market order) when triggered. Repeat the setup, but with the lower bid at **149.00** instead of 149.30, so that the market will *gap* through your limit:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.60|TIF=DAY
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=149.00|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.60|TIF=DAY
```

Place the stop-limit:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=STOP_LIMIT|QTY=100|STOP=149.50|PRICE=149.40|TIF=DAY
```

`STOP=149.50` controls **when** the order is activated; `PRICE=149.40` is the **worst price** you will accept once it is. Now trade through both levels:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.00|TIF=DAY
```

The trade at 149.00 triggers the stop, and a limit sell at 149.40 is placed — but the only bid left is at 149.00, below your limit, so nothing fills. The order now **rests** on the ask side at 149.40 (check with `BOOK|SYM=AAPL` and `ORDERS`). A plain stop would have sold at 149.00; the stop-limit protected the price and gave up the certainty of getting out.

:material-checkbox-blank-outline: **Checkpoint:** the stop-limit triggered and rests as a limit sell at 149.40.

Before the next exercise, cancel the resting stop-limit and let the market maker quote AAPL again:

```
[TRADER01]> CANCEL|ID=<stop-limit order id>
[MM_AAPL_01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=AAPL-MM-005
```

7.6 Exercise 3: Fill-or-Kill (FOK)

FOK demands the entire quantity in a single all-or-nothing execution (which may sweep several resting orders) or cancels:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=FOK|QTY=1000|PRICE=150.10
```

The market maker offers only 500, so the ask side doesn't have 1000 shares at or below 150.10, and the order is rejected at once with `code=INSUFFICIENT_LIQUIDITY` — nothing trades.

Try with a smaller qty that the MM can fill:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=FOK|QTY=100|PRICE=150.10
```

:material-checkbox-blank-outline: **Checkpoint:** large FOK rejected; small FOK filled.

7.7 Exercise 4: Immediate-or-Cancel (IOC)

IOC fills as much as possible immediately, then cancels the rest:

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=IOC|QTY=1000|PRICE=420.20
```

If only 300 are available at the ask, you get 300 filled and 700 cancelled.

:material-checkbox-blank-outline: **Checkpoint:** partial fill + cancellation of remainder.

7.8 Exercise 5: Iceberg Order

An iceberg shows only a visible "peak" quantity while hiding the reserve. First give TSLA a known book: chapter 04 left orders of its own on it, and its market-maker quote was cleared there. Clear TSLA and have the market maker quote it again:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=TSLA
[MM_TSLA_01]> QUOTE|SYM=TSLA|BID=249.75|ASK=250.25|BID_QTY=200|ASK_QTY=200|TIF=DAY|QUOTE_ID=TSLA-MM-005
```

Then place the iceberg:

```
[TRADER01]> NEW|SYM=TSLA|SIDE=BUY|TYPE=ICEBERG|QTY=1000|PRICE=249.70|VISIBLE=100|TIF=DAY
```

The book shows only 100 visible. When those 100 fill, another 100 automatically appears until the full 1000 is done.

Check the book:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=TSLA
```

You should see the market maker's 200-share bid at 249.75 and, below it, a 100-share bid at 249.70 — not 1000 — next to the market maker's 200 ask at 250.25. (The iceberg's price is one tick below the market maker's bid on purpose: `BOOK` adds up all orders at one price, so at 249.75 you would see the market maker's 200 plus your 100.)

:material-checkbox-blank-outline: **Checkpoint:** only peak quantity visible in the book.

7.9 Exercise 6: Trailing Stop

A trailing stop follows the market by a fixed offset. To watch it work you control every trade, as in the setup above. Clear AAPL again and create a last traded price of 150.00:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY
```

Now place the trailing stop. No `STOP=` is given, so the engine derives the initial trigger from the last traded price minus `TRAIL`: $150.00 - 0.20 = \mathbf{149.80}$.

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=TRAILING_STOP|QTY=100|TRAIL=0.20|TIF=DAY
```

Without a trade, no automatic trigger

If AAPL has no trade history, a `TRAILING_STOP` without an explicit `STOP=` is **rejected** — pass `STOP=<price>` to set the initial trigger yourself, e.g. `STOP=149.80`.
Put a bid in the book for the stop to sell into once it fires (it becomes a market sell), well below the trigger so it does not interfere:
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.70|TIF=DAY
Let the price rise. A trade at 150.50 drags the stop up to 150.30 (150.50 − 0.20); nothing happens visibly:
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.50|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.50|TIF=DAY
Let it reverse. A trade at 150.30 reaches the (raised) stop. Had the stop stayed at 149.80 this trade would not have touched it:
[MM_AAPL_01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.30|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.30|TIF=DAY
`TRADER01`'s console now reports the stop's market sell, filled against the 149.70 bid:
[] FILL qty=100 @149.7 remaining=0 [FILLED]
The stop only moves **up** (for a sell) or **down** (for a buy) — never against you — and, like a plain stop, it guarantees the exit, not the price: it was triggered at 150.30 and sold at 149.70.
:material-checkbox-blank-outline: **Checkpoint:** the trailing stop sat unfilled while the price rose, then sold at 149.70 after the reversal to 150.30.

7.10 Order Type Summary

Type	Execution	Rests?	Key Parameter
MARKET	Immediate, best price	No	—
LIMIT	At price or better	Yes	`PRICE`
STOP	Dormant → market on trigger	Until triggered	`STOP`

Type	Execution	Rests?	Key Parameter
STOP_LIMIT	Dormant → limit on trigger	Until triggered, then yes	`STOP`, `PRICE`
FOK	All-or-nothing, immediate	No	`QTY`, `PRICE`
IOC	Fill what you can, cancel rest	No	`QTY`, `PRICE`
ICEBERG	Hidden reserve, visible peak	Yes	`VISIBLE`
TRAILING_STOP	Dynamic stop follows market	Until triggered	`TRAIL`

7.11 Reflection

Why does STOP_LIMIT exist at all, given that a plain STOP order guarantees execution once triggered? What real trading scenario would make a trader prefer a STOP_LIMIT's "might not fill" risk over a STOP's "might fill at a bad price" risk?

7.12 Further Reading

- Order Types
- ALF Protocol — Single-leg Orders

Next: 06 — Time-in-Force & Sessions

CHAPTER 8

Time-in-Force & Sessions

8.1 Objective

Understand how Time-in-Force (TIF) controls order lifetime across session phases, and how to use the scheduler to drive the exchange through a full trading day.

Background reading

- Auctions & Scheduling

8.2 Prerequisites

- Chapters 01–05 completed.
- `pm-engine`, at least one trader gateway and the operator console running. Exercise 1 runs `pm-scheduler` once to show the phases; after that the chapter controls them by hand (see the Freeze/Advance Procedure below).

8.3 Background

Session phases progress as:

```
PRE_OPEN → OPENING_AUCTION → CONTINUOUS → CLOSING_AUCTION → CLOSED
```

Different TIF values control when orders are active and when they expire.

TIF	Meaning	Valid During
`DAY`	Lives until the end of the trading day (expires at CLOSED)	PRE_OPEN, OPENING_AUCTION, CONTINUOUS, CLOSING_AUCTION
`GTC`	Good-Till-Cancelled; survives session boundaries	Any
`ATO`	At-The-Open only; participates in opening auction	OPENING_AUCTION

TIF	Meaning	Valid During
ATC	At-The-Close only; participates in closing auction	CLOSING_AUCTION

A DAY order's "end of session" is not the same as "the engine process exits"

The table above describes the session-scheduler's rule: `pm-scheduler` driving the session into `CLOSED` cancels every resting `DAY` order at that moment, published as `order.expired`. That is unchanged.
Separately, an **engine restart is not itself a day boundary**. If you stop and restart `pm-engine` mid-session (a config reload, a crash recovery) while still on the same business day, a resting `DAY` order now survives that restart, exactly like a `GTC` order would — it is only discarded, at the *next* startup, once its own timestamp is from a prior business day. See Persistence for the full restart-vs-session-close distinction, and chapter 16 — Persistence & Recovery for a hands-on exercise.

8.4 Exercise 1: Observe Session Phases

`pm-scheduler --now` replays the whole day, and the engine only accepts the transitions in order, so it can show you every phase only when it starts from `CLOSED`. If `SESSION_STATUS` in the operator console says anything else (after chapter 05 it says `CONTINUOUS`), close the day by hand first:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSING_AUCTION
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

Entering `CLOSED` expires every resting `DAY` order, which includes the market makers' quotes from chapter 02, so the books are empty afterwards. That is fine for the rest of this chapter and for chapter 07, which build their own orders. Now start the scheduler in rapid-fire mode to see all transitions quickly:

```
pm-scheduler --now --delay 5
```

Watch the transitions (`--delay 5` leaves five seconds between them, long enough to query the phase after each one; the whole day takes about 25 seconds). The trader console does **not** subscribe to session broadcasts — it only answers an explicit `SESSION` query — so use the operator console, which does:

```
[GW_ADMIN|ADMIN]> SESSION_STATUS
```

Run it after each transition, or watch the audit trail in another terminal for the `session.state` events. The audit logger prints in batches (at least every 10 seconds), so a line can appear a few seconds after the transition. It also prints order-book checkpoints at every phase change, so filter for the lines you want:

```
pm-audit --terminal | grep '\[session.state\]'
```

```
[2026-10-31T11:40:59.5+00:00] [session.state] [seq=3 eseq=40 msg=… cause=… chain=…] {"state": "OPENING_AUCTION",
[2026-10-31T11:41:01.5+00:00] [session.state] [seq=4 eseq=52 msg=… cause=… chain=…] {"state": "CONTINUOUS", "prev
```

:material-checkbox-blank-outline: **Checkpoint:** you see at least one session state transition.

8.5 Freeze/Advance Procedure (Deterministic Session Control)

The exercises below need the exchange sitting in a **specific, known** phase before you place an order — otherwise a TIF exercise may pass or fail purely by timing luck. Instead of running `pm-scheduler` continuously, use the admin gateway to freeze on a phase and advance only when you are ready:

1. Do **not** start `pm-scheduler --now` (which free-runs through all phases). Start the engine and gateways as usual, but leave the scheduler stopped. With `sessions_enabled: true` and no scheduler, the engine starts in **`CLOSED`** and stays there — so every order is rejected with "Market is closed" until you move it yourself in step 2. That is expected, not a fault.

2. From the admin gateway, move to the exact phase you need for the next exercise. The engine only accepts the transitions in the session sequence, and from `CLOSED` the only legal move is to `PRE_OPEN` — so to reach `CONTINUOUS` go in two steps:

   ```
   [GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
   [GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
   ```

   An illegal jump (for example `CLOSED` straight to `CONTINUOUS`) is silently ignored by the engine, and the console just prints `TIMEOUT` with no reason. If you see that, run `SESSION_STATUS` and check which phase you are really in.

3. Confirm the change took effect with:

   ```
   [GW_ADMIN|ADMIN]> SESSION_STATUS
   ```

4. Run the TIF exercise's orders while frozen in that phase.

5. When the exercise calls for a transition (e.g., DAY expiry at `CLOSED`, or `ATO`/`ATC` at auction phases), advance deliberately:

   ```
   [GW_ADMIN|ADMIN]> SESSION|STATE=CLOSING_AUCTION
   [GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
   ```

Using `SESSION|STATE=` this way means each exercise below runs in a reproducible phase instead of racing a free-running scheduler.

8.6 Exercise 2: DAY Order Expiry

During CONTINUOUS (freeze there per the procedure above: the scheduler's day in Exercise 1 ended in `CLOSED`, so send `SESSION|STATE=PRE_OPEN` and then `SESSION|STATE=CONTINUOUS`), place a DAY order:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=148.00|TIF=DAY
```

Advance the session to CLOSED. The engine also accepts `CONTINUOUS` → `CLOSED` directly, but this exercise goes through the closing auction, the way a real day ends:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSING_AUCTION
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

Watch the trader console right after the second transition: the order is reported `EXPIRED`.

Operational note: DAY is best for intraday intent where stale overnight orders must not persist.

:material-checkbox-blank-outline: **Checkpoint:** DAY order expired at session close.

8.7 Exercise 3: GTC Order Survives

Exercise 2 left the market `CLOSED`, where every order is rejected. Open a new day first (`SESSION|STATE=PRE_OPEN`, then `SESSION|STATE=CONTINUOUS`) and place a GTC order:

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=415.00|TIF=GTC
```

Close the day again (`SESSION|STATE=CLOSING_AUCTION`, then `SESSION|STATE=CLOSED`). This order survives the CLOSED phase and will still be resting when the next session opens. `ORDERS` on `TRADER01` now shows the DAY order from Exercise 2 as `EXPIRED` and this one still `NEW`.

Operational note: GTC is used for standing instructions that remain valid until explicitly cancelled.

:material-checkbox-blank-outline: **Checkpoint:** after the session reaches `CLOSED`, `ORDERS` still lists the GTC order as `NEW`.

8.8 Exercise 4: ATO Order

The market is `CLOSED` after Exercise 3. Open a new day to `PRE_OPEN` (`SESSION|STATE=PRE_OPEN`), then freeze in OPENING_AUCTION (`[GW_ADMIN|ADMIN]> SESSION|STATE=OPENING_AUCTION` — from PRE_OPEN, this is a valid direct transition), and place an At-The-Open order:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=150.50|TIF=ATO
```

ATO is rejected during PRE_OPEN, not just outside auction hours

The engine only accepts `TIF=ATO` orders while the session state is exactly `OPENING_AUCTION` — `PRE_OPEN` accepts orders generally (it is not `CLOSED`), but an `ATO` order submitted while still in `PRE_OPEN` is rejected with `"ATO orders only accepted during opening auction"`. Make sure you have actually transitioned to `OPENING_AUCTION` (confirm with `SESSION_STATUS`) before running this exercise.
This order only participates in the opening auction. If not filled during the auction, it expires the moment the session leaves `OPENING_AUCTION` (e.g. when CONTINUOUS begins).
:material-checkbox-blank-outline: **Checkpoint:** ATO order participates in the auction or expires.

8.9 Exercise 5: ATC Order

Exercise 4 left the session in `OPENING_AUCTION`; move on to `CONTINUOUS` (`SESSION|STATE=CONTINUOUS`). First try an At-The-Close order during CONTINUOUS:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.00|TIF=ATC
```

It is rejected: `ATC orders only accepted during closing auction`. An `ATC` order is not parked until the close — it can only be entered *during* the closing auction. Freeze there and send it again:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSING_AUCTION
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.00|TIF=ATC
```

Now it is accepted and waits in the auction; it can only trade when the closing auction uncrosses, on the move to `CLOSED`.

:material-checkbox-blank-outline: **Checkpoint:** ATC is rejected in CONTINUOUS and accepted in CLOSING_AUCTION.

8.10 Exercise 6: Rejected TIF

Exercise 5 left the session in `CLOSING_AUCTION`. Take it through `CLOSED` and a fresh `PRE_OPEN` back to `CONTINUOUS` (`SESSION|STATE=CLOSED`, `SESSION|STATE=PRE_OPEN`, `SESSION|STATE=CONTINUOUS`), then try submitting an ATO order:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=ATO
```

Expected: rejection — `ATO orders only accepted during opening auction`. The phase must be **exactly** `OPENING_AUCTION`; `PRE_OPEN` is not enough, even though `PRE_OPEN` accepts orders generally. This matches the note in the summary below and the rule you saw in Exercise 4.

:material-checkbox-blank-outline: **Checkpoint:** engine rejects out-of-phase TIF correctly.

8.11 Summary

Phase	Accepts	Order expiry at this boundary
PRE_OPEN	DAY, GTC (not ATO — see Exercise 4)	—
OPENING_AUCTION	DAY, GTC, ATO	ATO (unfilled, on **exit** to any other phase)
CONTINUOUS	DAY, GTC (not ATO or ATC)	—
CLOSING_AUCTION	DAY, GTC, ATC	ATC (unfilled, on **exit** to CLOSED)
CLOSED	— (no new orders accepted)	DAY (all, on entry to CLOSED)

Notes:

- The four values above are the whole of `TIF=`: **DAY, GTC, ATO, ATC**. `IOC` and `FOK` are *order types* (`TYPE=`), not time-in-force values — a common confusion, since all six describe "how long does this live".
- `ATO` is only accepted while the session state is exactly `OPENING_AUCTION` — not during `PRE_OPEN`, even though `PRE_OPEN` accepts orders generally.
- `ATC` is only accepted while the session state is exactly `CLOSING_AUCTION`.
- Unfilled `ATO`/`ATC` orders are expired the moment the session *leaves* their respective auction phase (not on any other transition). The `order.expired` event reports `Opening auction ended` for ATO and `Closing auction ended` for ATC.
- All resting `DAY` orders (regardless of when they were placed) are expired the moment the session transitions *into* `CLOSED`, with reason `Trading day ended`.

8.12 Reflection

Why does GTC survive every phase transition while DAY does not? If you were running an overnight risk desk, what would go wrong if all your hedge orders were accidentally submitted as DAY instead of GTC?

8.13 Further Reading

- Auctions & Scheduling
- Persistence
- A Full Trading Day

Next: 07 — Auctions

CHAPTER 9

Auctions

9.1 Objective

Run opening and closing auctions, understand equilibrium price calculation, and observe how auction orders are collected and matched in a single uncrossing event.

Background reading

- Equilibrium price - What are auctions?

9.2 Prerequisites

- Chapters 01–06 completed.
- `pm-engine`, both trader consoles, the market-maker gateway and the operator console running, and **no** `pm-scheduler` (you drive the phases by hand).

9.3 Background

Auctions concentrate liquidity at a single price. All orders are collected during the auction phase, then matched at the **equilibrium price** — the price that maximises the quantity traded.

Key properties:

- No partial information leakage (indicative prices may be shown).
- All fills happen at the same price.
- Prevents manipulation via timing advantages.

9.4 Exercise 1: Set Up for an Opening Auction

Stop the scheduler and use manual phase control instead (see 06 — Freeze/Advance Procedure). You want to be sitting in `OPENING_AUCTION` — **not** `PRE_OPEN` — before sending the `TIF=ATO` orders below: the engine only accepts `ATO` orders while the session state is exactly `OPENING_AUCTION` and rejects them during `PRE_OPEN` even

though PRE_OPEN accepts orders generally. PRE_OPEN can only be entered from CLOSED, so first check where you are:

```
[GW_ADMIN|ADMIN]> SESSION_STATUS
```

Chapter 06 left the market in CONTINUOUS. From there, close the day first — SESSION|STATE=CLOSING_AUCTION, then SESSION|STATE=CLOSED (CLOSED alone is also accepted from CONTINUOUS; a request for PRE_OPEN from CONTINUOUS is ignored and the console only prints TIMEOUT). From CLOSED go on:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
```

Before the auction starts, empty the AAPL book. The market-maker quotes from chapter 02 would otherwise take part in the auction too, and the calculation in Exercise 2 assumes only the four orders below. Stop any pm-mm-bot and AI traders, then cancel everything resting on AAPL — orders and quotes — and check the book is empty:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
[GW_ADMIN|ADMIN]> SESSION|STATE=OPENING_AUCTION
```

From two gateways, enter orders while in OPENING_AUCTION:

TRADER01:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=300|PRICE=150.50|TIF=ATO
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=150.20|TIF=ATO
```

TRADER02:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=400|PRICE=149.80|TIF=ATO
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.30|TIF=ATO
```

Add one order that cannot cross, for Exercise 3 — a bid for MSFT far below any offer (the MSFT market maker, when it is quoting, asks about 420):

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=400.00|TIF=ATO
```

:material-checkbox-blank-outline: **Checkpoint:** orders accepted, resting in auction book.

9.5 Exercise 2: Trigger the Auction Uncrossing

When the session leaves OPENING_AUCTION for CONTINUOUS, all crossing orders match at the equilibrium price. Nothing has traded yet — the orders from Exercise 1 are collected, not matched. Work out the result by hand first, then fire the uncrossing:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
```

For the exact orders entered in Exercise 1, here is the full calculation (verified against engine/auction.py):

Candidate price	Cumulative buy qty (bids price)	Cumulative sell qty (asks price)	Matched qty	Surplus
149.80	500 (300+200)	400	**400**	100
150.20	500	400	400	100
150.30	300	500	300	200
150.50	300	500	300	200

Maximum matched quantity is 400, tied between 149.80 and 150.20 with equal surplus (100). The engine scans candidate prices from lowest to highest and only replaces the current best on a **strict** improvement, so the **first** price reached with the best (qty, surplus) pair wins. That means the equilibrium price here is **149.80** — the lower of the two tied prices — not a value in between.

Expected: all fills print execution price `149.80`, for a total matched quantity of 400 shares. TRADER01's second order fills 100 of its 200 shares and reports `[PARTIAL]`; its unfilled 100, and TRADER02's sell at 150.30, then expire, because an `ATO` order only lives for the opening auction.

What to observe: identify the one common execution price printed across all fill events, and confirm it equals `149.80` and the total filled quantity equals `400` (300 from TRADER01's first order + 100 from the second, against TRADER02's 400-share sell). On `TRADER01` you see:

```
FILL      <first order id>   qty=300 @149.8  remaining=0    [FILLED]
FILL      <second order id>  qty=100 @149.8  remaining=100  [PARTIAL]
EXPIRED   <second order id>  (DAY order — trading day ended)
```

:material-checkbox-blank-outline: **Checkpoint:** all fills report execution price `149.80` and matched quantity `400`.

9.6 Exercise 3: Unfilled Auction Orders

An ATO order that does not cross expires when the auction ends, instead of carrying on into continuous trading. You entered one in Exercise 1: the MSFT bid at 400.00. The uncrossing in Exercise 2 left it unfilled — no MSFT seller was anywhere near 400 (the day's market-maker quotes were gone after chapter 06's close, and a quote at about 420 would be far above 400 anyway) — and the console reported it `EXPIRED` in the same burst as the AAPL fills. Confirm it:

```
[TRADER01]> ORDERS
```

The MSFT order is listed with status `EXPIRED`. (The console's explanation reads *Opening auction ended*, identifying why the `ATO` remainder expired.

:material-checkbox-blank-outline: **Checkpoint:** the unmatched ATO order expired at the start of CONTINUOUS.

9.7 Exercise 4: Closing Auction

Move the session to the closing auction:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSING_AUCTION
```

Then enter:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.80|TIF=ATC
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.60|TIF=ATC
```

Close the day; on the transition to CLOSED the auction uncrosses and fills are generated:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

Both orders fill in full at one price, **150.60**: the candidates 150.60 and 150.80 both match 100 shares with no surplus, and the engine keeps the lower of two tied prices, exactly as in Exercise 2. Each console prints `FILL … qty=100 @150.6 remaining=0 [FILLED]`.

Remember to reopen the market (`PRE_OPEN`, then `CONTINUOUS`) before the next chapter.

:material-checkbox-blank-outline: **Checkpoint:** closing auction produces fills.

9.8 Exercise 5: GTC Orders in Auctions

GTC orders participate in auctions alongside ATO/ATC orders. Prove it for the opening auction. After Exercise 4 the market is `CLOSED`; move to `PRE_OPEN` (which accepts GTC but not ATO), clear AAPL with `CANCEL_SYM|SYM=AAPL`, and rest a GTC bid:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=150.00|TIF=GTC
```

Move to `OPENING_AUCTION`, offer 50 shares at the same price as an ATO order, and uncross:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=OPENING_AUCTION
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=50|PRICE=150.00|TIF=ATO
[GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
```

Both consoles report `FILL … qty=50 @150.0 [FILLED]`: the GTC bid, placed before the auction began, was matched in it. The same holds in the closing auction — rest a GTC bid during `CONTINUOUS`, send an `ATC` sell at the same price after `SESSION|STATE=CLOSING_AUCTION`, and they trade when the session moves to `CLOSED`.

:material-checkbox-blank-outline: **Checkpoint:** the GTC bid filled in the opening auction against an ATO order.

9.9 Equilibrium Price Rules

The auction algorithm scans candidate prices from lowest to highest and selects the price that:

1. **Maximises traded volume** — most shares can match.
2. **Minimises surplus** — least shares left unexecuted at that price.
3. **Ties resolved by scan order** — if multiple prices tie on both volume and surplus, the engine keeps the first (lowest) one it found, since it only replaces the current best on a strict improvement. There is no separate "nearest to last trade" tie-break — the tie always resolves to the lower of the tied candidate prices.

How real exchanges break the tie

Most real exchanges use more steps after rules 1 and 2. If the surplus is on the buy side at every tied price, they pick the *highest* tied price (buyers are pressing, so the price moves up); if it is on the sell side, the lowest; and only if there is no clear pressure do they fall back to the price closest to a reference price, such as the last trade. In this chapter's example the surplus is on the buy side, so a typical exchange would uncross at 150.20, not 149.80. EduMatcher keeps the simpler rule so the result is easy to compute by hand — a good discussion point for the Reflection below.

9.10 Reflection

Why does an auction match all crossing orders at a single equilibrium price, instead of walking the book and filling each order at its own limit price (as CONTINUOUS trading does)? What problem would arise at the open if every order filled at its own price instead?

9.11 Further Reading

- Auctions & Scheduling
- Auction Equilibrium Concepts
- Order Types

Next: 08 — Cancelling & Managing Orders

CHAPTER 10

Cancelling & Managing Orders

10.1 Objective

Master the `STATUS`, `ORDERS`, and `CANCEL` commands for understanding gateway state, inspecting resting orders, and managing the order lifecycle.

Background reading

- ALF Console - The Admin Console and Exchange Commands

10.2 Prerequisites

- Chapters 01–07 completed.
- At least one connected trader gateway with resting orders to manage.

10.3 Exercise 1: Cancel a Resting Order

Place an order and then cancel it:

```
[TRADER01]> NEW|SYM=TSLA|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=248.00|TIF=DAY|TAG=CXL-ORDER-001
```

Note the `order_id`, then:

```
[TRADER01]> CANCEL|ID=<order_id>|RTAG=CXL-REST-001
```

Expected: `CANCELLED <order_id> rtag=CXL-REST-001`.

`RTAG` identifies this cancel request. The engine echoes it on the `CANCELLED` response or rejected ACK, which makes retries and concurrent cancel tests easier to audit. The original order `TAG` is echoed separately, so a client can tell which order was acted on and which cancel request completed.

:material-checkbox-blank-outline: **Checkpoint:** order cancelled; `ORDERS` no longer lists it as resting.

10.4 Exercise 2: Cancel a Partially Filled Order

You need an order that fills only in part, so control what is on offer. From the operator console clear AAPL (stop any `pm-mm-bot` quoting it), and have `TRADER02` offer 300 shares:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=300|PRICE=150.10|TIF=DAY
```

1. Place a large limit buy at the same price. It takes the 300 and the other 700 rest:

   ```
   [TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=1000|PRICE=150.10|TIF=DAY
   ```

   Expected: `FILL … qty=300 @150.1  remaining=700  [PARTIAL]`.

2. Cancel the remainder:

   ```
   [TRADER01]> CANCEL|ID=<order_id>|RTAG=CXL-PARTIAL-001
   ```

Expected: `CANCELLED <order_id> rtag=CXL-PARTIAL-001`. The unfilled 700 are gone; the 300 that filled stay executed. `ORDERS` shows the order as `CANCELLED` with `Qty` 1000 and `Rem` 700 — the quantity that was still open when it was cancelled, not the filled part.

:material-checkbox-blank-outline: **Checkpoint:** cancel succeeds; filled qty preserved.

10.5 Exercise 3: Compare STATUS and ORDERS

Place a fresh resting order so the gateway has something to inspect:

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=25|PRICE=419.00|TIF=DAY
```

First run `STATUS`:

```
[TRADER01]> STATUS
```

`STATUS` gives a quick gateway summary: gateway ID, whether it is authenticated, known symbols (`–` until you have run `SYMBOLS` once in this console), cached order counts by lifecycle state (for example `CANCELLED=3, NEW=1`), cached and active quote legs from quote submissions made by this console, and position symbols. Ordinary order fills are not quote legs, and these counts are not a market-wide quote total. It is useful for a fast health check, but it is not the detailed order table.

Now inspect the actual orders:

```
[TRADER01]> ORDERS
```

`ORDERS` is the gateway command for detailed order inspection. It displays order ID, symbol, side, type, TIF, price, quantity, remaining quantity, and status for orders belonging to your gateway.

Use `STATUS` when you need a quick health pulse. Use `ORDERS` when you are debugging one specific order lifecycle.

:material-checkbox-blank-outline: **Checkpoint:** `STATUS` shows summary counts; `ORDERS` shows the full order row and ID.

10.6 Exercise 4: Inspect Resting Order Details

```
[TRADER01]> ORDERS
```

Find your order in the table. The response includes:

- `symbol`, `side`, `type`, `price`, `qty`
- remaining quantity
- `status` (NEW, PARTIAL, FILLED, CANCELLED, REJECTED, EXPIRED)
- `tif`, last update time

:material-checkbox-blank-outline: **Checkpoint:** `ORDERS` returns full details for each resting order.

10.7 Exercise 5: Cancel a Non-Existent Order

```
[TRADER01]> CANCEL|ID=DOES_NOT_EXIST|RTAG=CXL-MISSING-001
```

Expected: rejection, shown in the console as `REJECTED DOES_NOT_EXIST code=ORDER_NOT_FOUND rtag=CXL-MISSING-001 Order not found` (`REJECT_CODE` is the field's name on the wire).

:material-checkbox-blank-outline: **Checkpoint:** error message returned cleanly.

10.8 Exercise 6: Cancel Another Gateway's Order

Try cancelling an order belonging to TRADER02:

```
[TRADER01]> CANCEL|ID=<trader02_order_id>|RTAG=CXL-NOTOWNER-001
```

Expected: rejection — you can only cancel your own orders: `code=NOT_OWNER Cannot cancel an order owned by another gateway`.

:material-checkbox-blank-outline: **Checkpoint:** cross-gateway cancel rejected.

10.9 Exercise 7: Admin Kill Switch

From the admin gateway, cancel all orders for a specific gateway:

```
[GW_ADMIN|ADMIN]> KILL|GW=TRADER01
```

The console reports `KILL OK TRADER01 orders=<n> quotes=<n>` — the number of orders and of market-maker quote legs cancelled — and each cancelled order appears on `TRADER01`'s console as `CANCELLED <order_id> reason=KILL_SWITCH`.

Or cancel all orders for a specific symbol across all gateways:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=TSLA
```

which answers `CANCEL_SYM OK TSLA orders=<n> quotes=<n>`. Both commands also clear market-maker quotes, which is why they are the quickest way to empty a book before an experiment.

:material-checkbox-blank-outline: **Checkpoint:** admin commands successfully cancel orders.

10.10 Exercise 8: Inspect SYMBOLS Metadata from Gateway

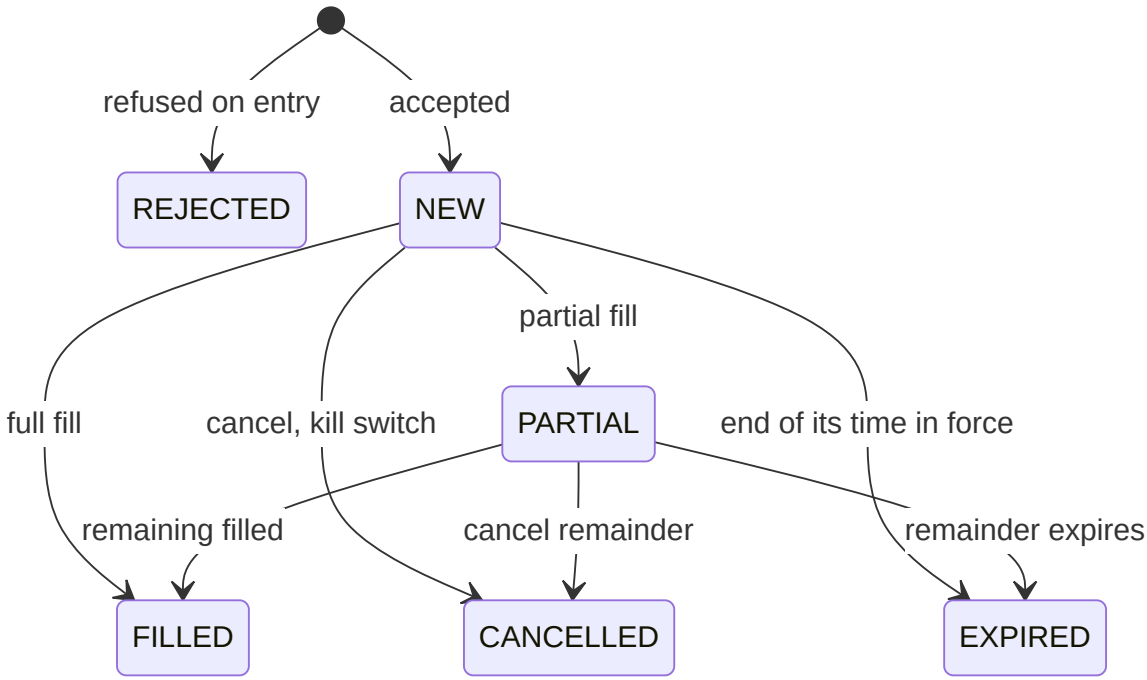
Request the symbol catalog:

```
[TRADER01]> SYMBOLS
```

The table shows each symbol's tick size and its market-maker obligation settings (enforced or not, maximum spread in ticks, minimum quoted quantity) — the parts of the configuration a trader needs while trading.

:material-checkbox-blank-outline: **Checkpoint:** you can explain how `SYMBOLS` complements config-file inspection during operations.

10.11 Order Lifecycle Summary



`EXPIRED` is the engine ending an order because its time in force ran out — a `DAY` order at the close, an `ATO`/`ATC` order after its auction — as opposed to `CANCELLED`, which is someone's decision.

10.12 Reflection

Why does the operator's `KILL|GW=<gateway_id>` cancel every order for an entire gateway rather than requiring one CANCEL per order? In what operational scenario (think: a runaway bot or a disconnected trader) does that bulk-kill behavior matter most?

10.13 Further Reading

- ALF Console (pm-alf-console)
- Message Types (system.symbols)
- ALF Protocol — Cancellation Semantics

Next: 09 — Market Making

Part III

Liquidity, risk and results

Chapters 09–14. Quote as a market maker, trade combo orders, trigger the risk controls, read P&L, watch the market-data and drop-copy feeds, and let AI traders generate order flow.

CHAPTER 11

Market Making

11.1 Objective

Understand the QUOTE command from a human operator's perspective, explore quote lifecycle, inactivation policies, QLEGS inspection, and MM obligations.

Background reading

- Market Making

11.2 Prerequisites

Add a manual MM gateway to your config:

```
participants:
  # ... everything from Chapter 02 ...
  - id: MM_MANUAL_01
    description: "Manual market-maker for training"
    role: MARKET_MAKER
    disconnect_behaviour: CANCEL_QUOTES_ONLY
    quote_refresh_policy: INACTIVATE_ON_ANY_FILL
```

It needs no `market_maker_quotes` seed of its own: the seeds from Chapter 02 already satisfy the verifier for every symbol, and the engine places only the seeds that are written down.

Deploy the change, then restart the engine — editing the YAML alone changes nothing, because every process reads the compiled artifact:

```
pm-config-deploy engine_config.yaml
```

Then restart `pm-engine`, restart your open consoles (`pm-admin`, the trader consoles — see "After every engine restart" in the introduction), reopen the market from the operator console (`SESSION|STATE=PRE_OPEN`, then `SESSION|STATE=CONTINUOUS`) and connect:

```
pm-alf-console --id MM_MANUAL_01
```

In this chapter `MM_MANUAL_01` should be the only liquidity on AAPL — otherwise your trader's orders will

trade with whichever market maker offers the better price, and the fills below will not be yours. Stop any `pm-mm-bot` quoting AAPL, and clear the symbol from the operator console:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
```

11.3 Exercise 1: Submit a Two-Sided Quote

```
[MM_MANUAL_01]> QUOTE|SYM=AAPL|BID=149.90|ASK=150.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q001
```

Expected:

```
QUOTE ACK   Q001  bid=<bid_id> ask=<ask_id>
QUOTE ACTIVE  Q001
```

Note both leg IDs — you'll need them to identify fills.

:material-checkbox-blank-outline: **Checkpoint:** quote acknowledged and active.

11.4 Exercise 2: Inspect Quote Legs with QLEGS

```
[MM_MANUAL_01]> QLEGS|SYM=AAPL|SHOW=ALL
```

Expected: a `Quote legs` table with one row per leg. The columns are:

Column	Meaning
`Symbol`	The instrument
`Quote`	Your `QUOTE_ID` — the label you chose when submitting
`Leg`	`BUY` (the bid leg) or `SELL` (the ask leg)
`Order`	The leg's full order ID
`Price`	The leg's price
`Qty`	Original quantity of the leg
`Rem`	Still resting
`Filled`	Already executed
`Filled?`	`YES` / `NO` — a quick scan column
`Leg status`	The order status of that leg
`Quote status`	The parent quote's state
`Time`	Last event time for the leg

Two things are worth noticing straight away, because they are the ones people misread:

- **`Leg status` and `Quote status` are different enums.** A leg is `NEW`, `PARTIAL`, `FILLED`, `CANCELLED`, `EXPIRED` or `PENDING`. The *quote* is `ACTIVE`, `INACTIVE_BID_FILLED`, `INACTIVE_ASK_FILLED` or `CANCELLED`. There is no `ACTIVE` leg status — a resting, untouched leg shows `NEW`.
- **The `Order` column holds the full order ID** — the same IDs the `QUOTE ACK` line printed, ready for a `CANCEL` (but see Exercise 7 before you cancel a single leg).

Try the filters — `SHOW=` accepts `ALL`, `ACTIVE` or `RECENT`:

```
[MM_MANUAL_01]> QLEGS|SYM=AAPL|SHOW=ACTIVE
[MM_MANUAL_01]> QLEGS|SYM=AAPL|SHOW=RECENT
```

`ACTIVE` keeps legs that are still working (an active status, or any remaining quantity); `RECENT` shows the complement — legs that are done. Right now `ACTIVE` should list both of your legs and `RECENT` should be empty; after Exercise 3 that flips.

:material-checkbox-blank-outline: **Checkpoint:** `QLEGS` shows both a `BUY` and a `SELL` row for `Q001`, each with `Leg status NEW` and `Quote status ACTIVE`, and you can say why no leg shows a status of `ACTIVE`.

11.5 Exercise 3: Get Filled and Observe Inactivation

From TRADER01, buy into the MM's ask:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

Back at MM_MANUAL_01, you should see:

```
CANCELLED <bid_id> reason=QUOTE_LEG_FILLED
QUOTE INACTIVE_ASK_FILLED  Q001
FILL      <ask_id>  qty=100 @150.1  remaining=400  [PARTIAL]
```

Under `INACTIVATE_ON_ANY_FILL`, the first fill on either leg inactivates the quote and cancels the **other** leg; the hit leg's unfilled remainder (here 400 at 150.10) keeps resting until you re-quote or cancel.

:material-checkbox-blank-outline: **Checkpoint:** fill + sibling cancel + INACTIVE status.

11.6 Exercise 4: Re-quote After Inactivation

Submit a fresh quote:

```
[MM_MANUAL_01]> QUOTE|SYM=AAPL|BID=149.92|ASK=150.08|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q002
```

:material-checkbox-blank-outline: **Checkpoint:** new quote active; QLEGS shows fresh legs.

11.7 Exercise 5: Replace a Quote Without Cancelling

You can replace directly — the engine handles the swap:

```
[MM_MANUAL_01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q003
```

Expected (each line also carries a `[HH:MM:SS.mmm]` time; the two `CANCELLED` lines are the two legs of `Q002` being replaced):

```
CANCELLED <Q002 bid id> reason=QUOTE_REPLACED
CANCELLED <Q002 ask id> reason=QUOTE_REPLACED
QUOTE CANCELLED  Q002  Replaced by new quote
QUOTE ACK   Q003  bid=<bid_id> ask=<ask_id>
QUOTE ACTIVE  Q003
```

(In Exercise 4 the same thing happened more quietly: the engine cancelled the 400 shares of `Q001`'s ask that were still resting, with one `CANCELLED … reason=QUOTE_REPLACED` line.)

:material-checkbox-blank-outline: **Checkpoint:** old quote cancelled, new quote active in one step.

11.8 Exercise 6: Explicit Cancel

```
[MM_MANUAL_01]> QUOTE_CANCEL|SYM=AAPL
```

Expected:

```
CANCELLED <bid_id>
CANCELLED <ask_id>
QUOTE CANCELLED  Q003  Cancelled by participant
QUOTE CXL OK AAPL  legs cancelled=2
```

Note

`QUOTE_CANCEL` is keyed by symbol, not by `quote_id`.
:material-checkbox-blank-outline: **Checkpoint:** quote fully cancelled.

11.9 Exercise 7: Cancel a Single Leg Directly — and See Why You Shouldn't

`QLEGS` gives you each leg's full order ID. Nothing stops you from feeding that ID straight into a plain `CANCEL`, bypassing `QUOTE_CANCEL` entirely. Try it.

Submit a fresh quote and note the bid's full order ID from the `QUOTE ACK` line:

```
[MM_MANUAL_01]> QUOTE|SYM=AAPL|BID=149.90|ASK=150.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q004
```

Now cancel *only* the bid leg, by order ID, with plain `CANCEL` — not `QUOTE_CANCEL`:

```
[MM_MANUAL_01]> CANCEL|ID=<bid_id>
```

Expected:

```
CANCELLED <bid_id>
```

That's it — no `QUOTE CANCELLED` line, no sibling cancel. Now inspect both views:

```
[MM_MANUAL_01]> QLEGS|SYM=AAPL|SHOW=ALL
[MM_MANUAL_01]> QBOOT|SYM=AAPL
```

You should see the bid leg's `Leg status` as `CANCELLED`, but the ask leg is still `NEW` and resting, and **`Quote status` on both rows still reads `ACTIVE`** — `QBOOT` agrees, still showing `Q004` as the active slot for this symbol. Compare this with Exercise 3: a fill on one leg makes the engine cancel the sibling and publish `QUOTE INACTIVE_*_FILLED` for you. A manual `CANCEL` on one leg does not — the engine treats it as an ordinary, independent order cancel and never consults the quote it belongs to.

This is a real trap, not just an exercise

The engine has no special handling for cancelling a single quote leg by order ID. Only `QUOTE_CANCEL` (whole quote, both legs) and a fill (subject to `quote_refresh_policy`) keep the quote's state consistent. If you cancel one leg manually, the other leg keeps resting — silently carrying one-sided market risk — while your own state and the engine's have quietly diverged. Always use `QUOTE_CANCEL|SYM=<symbol>` to pull a quote, never a bare `CANCEL` on one of its legs.

Clean up before moving on:
[MM_MANUAL_01]> QUOTE_CANCEL|SYM=AAPL
:material-checkbox-blank-outline: **Checkpoint:** you have seen a single-leg `CANCEL` leave the sibling resting and both `Quote status` rows stuck on `ACTIVE`, and can explain why `QUOTE_CANCEL` is the only safe way to pull a whole quote.

11.10 Exercise 8: Check Quote Bootstrap State (QBOOT)

After submitting a quote, inspect the bootstrap state:

```
[MM_MANUAL_01]> QBOOT|SYM=AAPL
```

This shows the current active quote slot for your gateway+symbol — useful for verifying what the engine thinks your active quote is.

Why this matters operationally: on bot restart, a previous instance might have left an active quote in the book. `QBOOT` prevents blind re-quoting by showing whether a slot is already active so the new process can adopt or replace safely instead of creating duplicates.

The engine, not just your bot, can restart between quotes

Quote legs now persist across an **engine** restart the same way any other resting order does: unconditionally if `TIF=GTC`, or if `TIF=DAY` and the restart happens on the same business day. So `QBOOT` right after an engine restart can legitimately show your `Q001` slot still `ACTIVE` with the same leg IDs, even though you never resubmitted the quote — the engine restored it and re-registered it in its own `QuoteIndex`. Only a business-day rollover (or an explicit `QUOTE_CANCEL`/fill) clears a `TIF=DAY` quote slot. See Persistence — Impact of a Business-Day Change for the full mechanics.
The reply is a `Quote bootstrap` table, one row per active quote:

Column	Meaning
`Symbol`	The instrument
`Quote`	The `QUOTE_ID` of the quote occupying the slot
`State`	The quote state — `ACTIVE` for a live quote (an inactivated or cancelled quote is not listed)
`Bid` / `Ask`	The two quoted prices, or `-` if that side has none
`BidRem` / `AskRem`	Remaining quantity on each leg

When there is nothing to report — no active quote for that gateway and symbol — you get a single dim line instead of a table:
No active quote bootstrap entries returned.
That empty response is the useful one on startup: it means the slot is free and you can quote without first cancelling something.
Run it in both states to see the difference. With a quote active, as `Q001` was after Exercise 1, you get a row. Once a fill has inactivated the quote, as in Exercise 3, `QBOOT` reports **no entry at all** — the slot is free again — even though `QLEGS|SYM=AAPL|SHOW=ALL` still lists the quote's legs with `Quote status INACTIVE_ASK_FILLED`. That is the difference between the two commands: `QBOOT` answers "is there a live quote I must not duplicate?", `QLEGS` keeps the history.

SYM= is optional

`QBOOT` on its own asks about every symbol you have a quote on; `QBOOT|SYM=AAPL` narrows it to one. Use the bare form on a bot restart, when you do not yet know which slots are occupied.
:material-checkbox-blank-outline: **Checkpoint:** you have seen both responses — a table with a `State` of `ACTIVE`, and the "No active quote bootstrap entries" line — and can say which one tells a restarting bot it is safe to quote.

11.11 Exercise 9: Compare Inactivation Policies

Policy	Behaviour
`INACTIVATE_ON_ANY_FILL`	Sibling cancelled on any fill (even partial)
`INACTIVATE_ON_FULL_FILL`	Sibling cancelled only when the filled leg is fully consumed
`NEVER_INACTIVATE`	No automatic sibling cancel; the MM manages both legs itself

Reading the table is not the same as believing it. Change the policy and watch the difference — this is the exercise that makes the three names mean something.

1. Edit `MM_MANUAL_01` in your configuration to `quote_refresh_policy: INACTIVATE_ON_FULL_FILL`, then deploy, restart `pm-engine`, reopen the market, reconnect `MM_MANUAL_01`, and clear AAPL again with `CANCEL_SYM|SYM=AAPL`:

```
pm-config-deploy engine_config.yaml
```

2. Quote again, then take only *part* of one leg:

```
[MM_MANUAL_01]> QUOTE|SYM=AAPL|BID=149.90|ASK=150.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q010
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.10|TIF=DAY
```

3. Inspect both views:

```
[MM_MANUAL_01]> QLEGS|SYM=AAPL|SHOW=ALL
[MM_MANUAL_01]> QBOOT|SYM=AAPL
```

Under `INACTIVATE_ON_ANY_FILL` (Exercise 3) this partial fill would have pulled the bid and inactivated the quote. Under `INACTIVATE_ON_FULL_FILL` the ask leg shows `PARTIAL` with `Rem` 400, the bid leg is still `NEW`, and the quote is still `ACTIVE` — `QBOOT` lists it with `BidRem` 500 and `AskRem` 400.

4. Now consume the rest of the ask (another 400) and watch the quote flip to `INACTIVE_ASK_FILLED`: the bid leg is cancelled with `reason=QUOTE_LEG_FILLED` and `QBOOT` now reports no active entry.

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=400|PRICE=150.10|TIF=DAY
```

:material-checkbox-blank-outline: **Checkpoint:** you can state, for each of the three policies, what happens to the sibling leg on a *partial* fill — and you have observed at least two of them.

Set the policy back

Later chapters assume `INACTIVATE_ON_ANY_FILL`. Restore it and redeploy before moving on, or note that you changed it.

11.12 Exercise 10: Disconnect and Bulk-Cancel Behaviour

Two more ways a quote gets pulled that have nothing to do with `QUOTE_CANCEL` or a fill: your gateway disconnecting, and the self-service kill switch. Both matter for risk — a quote left resting after your process dies is the one-sided exposure Exercise 7 warned about, just caused by a dropped connection instead of a stray `CANCEL`.

Part A — disconnect behaviour. Recall the Prerequisites config:

```
disconnect_behaviour: CANCEL_QUOTES_ONLY
```

Quote a symbol, add a plain resting order too, then disconnect the gateway (Ctrl+C or close the console):

```
[MM_MANUAL_01]> QUOTE|SYM=AAPL|BID=149.90|ASK=150.10|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=Q020
[MM_MANUAL_01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=300.00|TIF=DAY
```

Disconnect `MM_MANUAL_01` (Ctrl+C), then reconnect the same way as in the Prerequisites and check both:

```
pm-alf-console --id MM_MANUAL_01
```

```
[MM_MANUAL_01]> QBOOT|SYM=AAPL
[MM_MANUAL_01]> ORDERS
```

`ORDERS` has no symbol filter — it lists everything resting under this gateway. Expected: the `AAPL` quote is gone (`QBOOT` reports no active entry for it), but the `MSFT` limit order is still listed by `ORDERS`, resting exactly as you left it. `CANCEL_QUOTES_ONLY` — the default — cancels only quotes on disconnect; ordinary resting orders are untouched. There are two other values: `CANCEL_ALL` additionally sweeps every plain resting order too, and `LEAVE_ALL` cancels nothing at all, leaving both quotes and orders resting across the disconnect. Which one you want is a risk decision, not a technical one — `LEAVE_ALL` trusts the MM to reconcile via `QBOOT`/`QLEGS` on reconnect (Exercise 8's note on surviving quotes); `CANCEL_QUOTES_ONLY` assumes a disconnected quoting engine can no longer manage its own risk and should not keep resting prices in the book; `CANCEL_ALL` extends that same assumption to every order, not just quotes.

Part B — self-service `KILL`. `QUOTE_CANCEL` is symbol-scoped and only touches quotes. When you need to pull everything for your own gateway in one shot — every active quote *and* every plain resting order, across every symbol — use `KILL`:

```
[MM_MANUAL_01]> KILL
```

Expected:

```
[<time>] KILL ACK  orders=<n> quote_legs=<n>
```

`quote_legs` counts individual legs, not quote records, so a single two-sided quote contributes up to 2 to that count. Scope it to one symbol with `KILL|SYM=<symbol>` instead of clearing every symbol at once.

This is not the admin KILL|GW=...

`KILL` on its own is self-service — a gateway can only kill its *own* exposure this way. An admin pulling another participant's exposure uses `KILL|GW=<gateway_id>` instead, covered in 19 — Advanced Admin Operations.
:material-checkbox-blank-outline: **Checkpoint:** you have seen a quote disappear on disconnect while a plain order survived under `CANCEL_QUOTES_ONLY`, and used `KILL` to clear all of your own gateway's exposure in one command.

11.13 Key Takeaways

- The QUOTE command creates two linked limit orders (bid + ask).
- `quote_id` identifies the logical quote; legs are separate order IDs.
- `QLEGS` inspects leg-level state; `QBOOT` inspects the quote slot. They report different enums: legs are `NEW/PARTIAL/FILLED/…`, quotes are `ACTIVE/INACTIVE_*_FILLED/CANCELLED`.
- Use QBOOT first during startup, then QLEGS for leg-level reconciliation.
- Replacement quotes don't require explicit cancel first.
- `order.fill` **does** include `quote_id` for a quote-leg fill, so you can correlate a fill back to the quote directly as well as via the leg order IDs.
- A plain `CANCEL` on one leg's order ID does **not** cancel the sibling or update `Quote status` — only `QUOTE_CANCEL`, or a fill under your `quote_refresh_policy`, keep the quote consistent. Never cancel a single leg manually.
- On disconnect, `disconnect_behaviour` decides what happens to your exposure: `CANCEL_QUOTES_ONLY` (default) cancels quotes only, `CANCEL_ALL` also cancels plain resting orders, `LEAVE_ALL` cancels neither.
- `KILL` (optionally `KILL|SYM=<symbol>`) is the self-service bulk cancel — every quote and every plain order for your own gateway in one command; `KILL|GW=<gateway_id>` is the admin equivalent for another gateway.

11.14 Reflection

Why does the engine tie the bid and ask legs of a quote together with a `quote_id` at all, rather than treating a market maker's two orders as completely independent? What could go wrong for a market maker's risk exposure if one leg filled and the sibling stayed resting under a `NEVER_INACTIVATE` policy?

Why do you think the engine cascades a sibling cancel on a *fill* but not on a manual `CANCEL` of one leg? What would have to change internally for single-leg `CANCEL` to be quote-aware, and can you think of a reason the engine might deliberately *not* want that — i.e., a case where cancelling one leg without touching the other is exactly what you'd want?

11.15 Further Reading

- Market Making
- The Market-Maker Bot (`pm-mm-bot`)
- Market-Maker Bot CLI Reference
- ALF Console (pm-alf-console)

- MM Quotes Concept
- ALF Protocol Reference
- Advanced Admin Operations — admin-side KILL and QCANCEL

Next: 10 — Combo Orders

CHAPTER 12

Combo Orders

12.1 Objective

Build and close multi-leg strategies, size their hedges, and distinguish entry checks from lifetime guarantees. Use OCO (One-Cancels-Other) for alternative entries and exits, then test partial fills, stops, failure cascades, session boundaries and recovery.

Background reading

- Combo Orders - OCO

12.2 Prerequisites

- Chapters 01–09 completed.
- Two trader gateways connected so you can stage opposing liquidity.
- An admin console for clearing the training books.
- For the stop-trigger labs, a third configured and connected trader, `TRADER03`, to print a trigger trade without changing `TRADER01`'s position. Add it to the `participants:` list of your `engine_config.yaml`, check, deploy, restart `pm-engine`, restart your open consoles and reopen the market (see "After every engine restart" in the introduction):

```
  - id: TRADER03
    description: "Carol — third trader"
    role: TRADER
```

  then connect with `pm-alf-console --id TRADER03`.
- For the three-leg basket, `TSLA` configured alongside `AAPL` and `MSFT`.

12.2.1 Lab discipline

Use an **isolated classroom exchange**, not a shared trading session. Stop market-maker bots and AI traders. Keep the session in `CONTINUOUS` with the scheduler stopped, using the freeze/advance procedure. The session exercise explicitly changes this.

Exercises 1–4 build on each other. Before Exercise 5 and before **each independent run or variant** in Exercises 6–15, clear the affected books:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=MSFT
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=TSLA
```

Only use the TSLA command if that symbol is configured. These commands cancel orders, **not positions or trade history**. Record positions before and after each lab and measure changes, rather than assuming a flat account. Price collars and circuit breakers must allow the example prices (AAPL 148–153, MSFT near 420 and TSLA near 250); have the instructor align the configuration before starting rather than treating a risk rejection as a strategy result.

Copy commands without the console prompt. Replace `<...>` placeholders with actual IDs. Use a fresh `COMBO_ID` or `OCO_ID` on every repeat. Save the child IDs from acknowledgements: `ORDERS` lists every order the console has seen — the children of a combo or OCO included — with its latest status, but there is no parent-combo status row. Use fill/cancel messages and parent status messages as evidence; the operator console's `ORDERS|GW=<gateway>` shows what the engine really has resting, and `pm-audit --terminal` is useful alongside the consoles.

12.2.2 Progression

Exercises	Skill
1–4	Entry atomicity, resting leg risk and parent cancellation
5–8	OCO entries/exits, partial fills, breakouts and trailing protection
9–10	Hedge ratios, closing a pair and three-to-ten-leg baskets
11–12	Depth, execution instructions and conditional combo legs
13–15	Failure handling, self-match prevention, sessions and recovery

12.3 Background

A **combo order** bundles two to ten legs on distinct symbols, each with its own side and quantity, and submits them as one message with a shared `COMBO_ID`.

`COMBO_TYPE=AON` (all-or-none) gives you an atomicity guarantee — but it is narrower than it first sounds, and the exact shape of it is the main thing this chapter teaches:

The experiments below begin with ordinary limit legs, external liquidity, continuous matching and no halt. AON is a submission-time liquidity check, not transaction rollback if a later child encounters SMP or another execution restriction; Exercises 12–14 explore those boundaries.

AON is atomic at entry, not for the combo's lifetime

When you submit an AON combo the engine checks, at that instant, whether *every* leg can fill completely. In the controlled setup above, there are three outcomes:

- **All legs can fill** → all legs execute together. No partial.
- **A leg is short, and another would trade at once** → the combo is **rejected**: its first leg with `INSUFFICIENT_LIQUIDITY`, and the combo `FAILED`. Resting the leg that would trade would leave its book crossed.
- **A leg is short, and no leg would trade → no leg matches at all**. Every leg is posted to its book as an ordinary resting order.

That third outcome is where the nuance lives. Those resting legs are now normal orders. Nothing stops another participant from hitting one of them later, and when that happens the combo moves to `PARTIALLY_MATCHED` — one leg filled, the others still resting. **You have leg risk again.**

In this controlled all-limit setup, a depth shortfall cannot make the combo execute only part of its intended entry. It can still *end up* half-filled while it rests. Exercises 2 and 3 show both halves of that.

The combo's lifecycle states are `PENDING`, `PARTIALLY_MATCHED`, `MATCHED`, `FAILED` (a leg was cancelled, rejected or expired, so the siblings cascade-cancel), `CANCELLED` and `REJECTED`.

12.4 Exercise 1: Simple Two-Leg Combo

To see the combo **fill atomically**, first guarantee liquidity on both legs with explicit counter-orders, and nothing else. Market-maker quotes from earlier chapters would get in the way — TRADER02's MSFT buy at 420.50, for example, would trade straight away with a market maker's cheaper ask instead of resting. So stop any `pm-mm-bot`, clear both symbols from the operator console, and then place the counter-orders:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=MSFT
```

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.10|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=420.50|TIF=DAY
```

Now submit the combo — buy AAPL and sell MSFT atomically:

```
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=PAIR-001|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0.
```

Because both counter-orders above match the combo's leg prices exactly, the engine can fill both legs simultaneously.

:material-checkbox-blank-outline: **Checkpoint:** the trader console shows a fill on each leg (with a `COMBO PARTIALLY_MATCHED PAIR-001` line between them) followed by `COMBO MATCHED PAIR-001`, and `BOOK|SYM=AAPL` / `BOOK|SYM=MSFT` in the operator console show both counter-orders gone.

Observation: when the pre-check passes, both legs execute in the same pass — this is the case the AON guarantee is designed for.

12.5 Exercise 2: Atomicity at Entry — One Short Leg Blocks Every Leg

Now check the pre-check: when liquidity is missing on **one** leg, *no* leg matches, even the one that could have — and the combo is rejected:

1. Exercise 1 used up the AAPL sell, so place a new one: `[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY`
2. Do **not** place any MSFT buy at 420.50 — cancel or avoid resting MSFT liquidity at that price so the second leg has nothing to match against.
3. Submit a new combo with a fresh ID:

```
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=PAIR-002|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY|
```

4. The AAPL leg is rejected with `INSUFFICIENT_LIQUIDITY` and the combo `FAILED`; the MSFT leg is never posted. Check `BOOK|SYM=AAPL` in the operator console and confirm the AAPL sell at 150.10 is **still resting, unfilled**, proving the combo did not execute the AAPL leg alone even though a matching counter-order existed for it.

:material-checkbox-blank-outline: **Checkpoint:** the combo did not execute the AAPL leg even though a matching counter-order was sitting there, and nothing of the combo rests.

Operational rationale: without the entry pre-check you would pick up accidental directional inventory the moment you submitted an unbalanced combo. And the AAPL leg could not rest instead: a buy at 150.10 resting against a sell at 150.10 would leave the book crossed.

12.6 Exercise 3: Leg Risk Returns Once the Combo Rests

A combo with no leg that can trade yet rests with both legs in their books. This is the case people are surprised by, so cause it deliberately.

1. Submit a combo whose AAPL leg bids below TRADER02's AAPL sell at 150.10, and confirm it is resting and `PENDING`:

```
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=PAIR-003|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0.
[TRADER01]> ORDERS
```

You should see both child legs — the AAPL buy at 150.00 and the MSFT sell at 420.50 — with status `NEW`.

2. From `TRADER02`, hit **only the MSFT leg**:

```
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=420.50|TIF=DAY
```

3. Look at what happened to the combo:

```
[TRADER01]> ORDERS
```

The MSFT fill message says `FILLED`, and `ORDERS` now lists that leg as `FILLED`. The AAPL leg is still `NEW`, resting. The combo has moved to `PARTIALLY_MATCHED`.

`TRADER01` is now short 50 MSFT with no offsetting AAPL position — precisely the leg risk the combo was supposed to prevent. The AON guarantee was never violated: it applied at submission, and at submission nothing executed.

:material-checkbox-blank-outline: **Checkpoint:** the combo is `PARTIALLY_MATCHED` with one leg filled and one resting, and you can explain why this does not contradict "all-or-none".

What a real desk does about this

Because a resting AON combo can be picked apart, desks do not leave them resting. They either submit only when both sides are already fillable, or they cancel the whole combo the moment it fails to execute at entry:

```
[TRADER01]> CANCEL|COMBO_ID=PAIR-003
```

That cancels every remaining leg. Fills that already happened are **not** reversed — cancel the combo before someone hits a leg, not after.

Clean up before continuing:

[TRADER01]> CANCEL|COMBO_ID=PAIR-003

12.7 Exercise 4: Cancel a Resting Combo

Exercise 3's clean-up already cancelled `PAIR-003`. Submit one more combo that cannot fill — there is no MSFT bid at 420.50 now, and the AAPL leg bids below the AAPL sell — so it rests, then cancel it as a whole:

```
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=PAIR-004|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0.
[TRADER01]> CANCEL|COMBO_ID=PAIR-004
```

All legs are cancelled together, and the console reports `COMBO CANCELLED  PAIR-004`.

:material-checkbox-blank-outline: **Checkpoint:** full combo cancellation confirmed.

12.8 Exercise 5: OCO — One-Cancels-Other

Link two independent orders so that when one **fully fills** or otherwise ends (cancel, rejection or expiry), the other is automatically cancelled. A partial fill alone does not end the pair; Exercise 7 tests this distinction.

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=OCO-AAPL-ENTRY|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=BUY|LEG1_TYPE=LIMIT|LEG1_PRICE=
```

Make the first leg fill by selling into it from `TRADER02`:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=149.50|TIF=DAY
```

The 149.50 leg fills, and the console reports the sibling at 148.00 cancelled: `OCO CANCEL OCO-AAPL-ENTRY sibling=…  OCO sibling … reached FILLED`. (If the AAPL book still holds asks at or below 149.50 from earlier exercises, the OCO's first leg fills on entry instead — clear AAPL with `CANCEL_SYM` first for a clean run.)

To pull an OCO pair yourself before either leg fills, cancel it by its `OCO_ID` rather than cancelling each leg individually. `OCO-AAPL-ENTRY` has already completed (cancelling it now answers `OCO CXL REJ … ORDER_NOT_FOUND`), so submit a fresh pair and cancel that:

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=OCO-AAPL-ENTRY-2|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=BUY|LEG1_TYPE=LIMIT|LEG1_PRIC
[TRADER01]> CANCEL|OCO_ID=OCO-AAPL-ENTRY-2
```

Both legs are cancelled: `OCO CXL OK OCO-AAPL-ENTRY-2  legs cancelled=2`.

:material-checkbox-blank-outline: **Checkpoint:** filling one OCO leg cancels the other.

12.9 Exercise 6: OCO — Protect an Existing Long Position

A common exit pattern is a take-profit plus a stop-loss. Both legs are **SELL**, not different sides: each is an alternative way to close the same long position. First acquire 100 shares on the cleared AAPL book:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY
```

Now submit the exit pair:

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=BRACKET-AAPL-001|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRI
```

Run both branches separately, acquiring a fresh 100-share position and using a fresh OCO ID on the second run:

- **Take profit:** `TRADER02` buys 100 AAPL at 151.00. The resting limit sells all 100; the stop is cancelled. Gross profit is 100.
- **Stop loss:** `TRADER02` posts a bid for 101 AAPL at 149.00, then `TRADER03` sells 1 at 149.00. That print triggers the stop, whose 100-share market sell consumes the remaining bid; the take-profit is cancelled. Gross loss is 100.

Quotes merely moving are not enough: these stop exercises create an actual trade at the trigger price.

This is not an attached, reduce-only bracket

The OCO does not create or reserve the long position, wait for a separate entry order, or resize itself after partial fills. A stop becomes a market order: 149.00 is a trigger, **not** a guaranteed execution price. If liquidity is absent, it may execute partly or not at all. You must reconcile the resulting position yourself. :material-checkbox-blank-outline: **Checkpoint:** demonstrate both exits, identify the sibling cancellation, and calculate P&L from actual fills rather than from the stop trigger.

12.10 Exercise 7: OCO — A Partial Fill Is Not a Completed Choice

Predict first: after 40 shares of a 100-share leg fill, does the sibling remain at 100, resize to 60, or disappear?

On a cleared AAPL book:

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=OCO-PARTIAL-01|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=BUY|LEG1_TYPE=LIMIT|LEG1_PRICE=
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=40|PRICE=149.50|TIF=DAY
[TRADER01]> ORDERS
```

The first leg has 60 remaining and is `PARTIAL`. The sibling still bids for **100**, not 60. OCO has not cancelled or resized it.

1. Sell another 60 at 149.50 from `TRADER02`. The first leg becomes `FILLED` and the 148.00 sibling is cancelled.
2. Reset and repeat with a fresh ID, but after the first 40-share fill, cancel the **partially filled child** with `CANCEL|ID=<child_id>`. Both remaining interests disappear. The 40 bought shares remain.
3. Reset and repeat once more, replacing the child cancel with `CANCEL|OCO_ID=<new_oco_id>`. Compare the position change, not just the cancellation messages.

Explain why the higher of two ordinary buy limits usually fills first. These are not two independent opportunities to buy 100 shares each. Nor does the link impose a shared cumulative-fill budget while both legs remain nonterminal. For a partially filled exit OCO, you must manage the remaining position and outstanding quantities yourself.

:material-checkbox-blank-outline: **Checkpoint:** record 40 filled, 60 remaining on one leg and 100 on its sibling; prove that cancellation does not undo the 40-share fill.

12.11 Exercise 8: OCO — Breakout Entry and Protective-Exit Variants

12.11.1 A. Choose a direction only after a breakout

Unlike Exercise 6, this pair has **opposite sides**. It expresses an alternative entry, not protection of an existing position.

First establish a last trade of 150.00 with a one-share sell from `TRADER02` and a one-share buy from `TRADER03`. Then submit:

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=BREAKOUT-01|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=BUY|LEG1_TYPE=STOP|LEG1_STOP=151.0
```

- **Up-break run:** `TRADER02` offers 101 at 151.00; `TRADER03` buys 1 at 151.00. The buy stop consumes the other 100 and cancels the sell stop. `TRADER01`'s position change is +100 AAPL.
- **Down-break run:** reset, print 150.00 again and recreate the OCO. `TRADER02` bids for 101 at 149.00; `TRADER03` sells 1 at 149.00. The sell stop consumes the other 100 and cancels the buy stop. The position change is −100 AAPL.

Repeat with only 1 share at the trigger price, leaving no liquidity for the stop. Observe that a triggered market order can end unfilled and still end its sibling. **OCO cancellation does not prove a position was acquired.**

12.11.2 B. Price protection versus execution risk

Acquire a fresh 100-share long at 150.00, as in Exercise 6. Use this exit:

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=EXIT-LIMIT-01|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRICE=
```

`TRADER02` bids for 101 at 148.50 and `TRADER03` sells 1 there. The gap print triggers the stop-limit, but its sell limit of 148.90 cannot hit 148.50. It remains working, as does the take-profit. You still own the 100 shares. Cancel the group and compare this with the market-stop outcome.

12.11.3 C. Trail a profitable position

Acquire another 100 shares at 150.00 and submit:

```
[TRADER01]> NEW|TYPE=OCO|OCO_ID=EXIT-TRAIL-01|SYM=AAPL|QTY=100|TIF=DAY|LEG1_SIDE=SELL|LEG1_TYPE=LIMIT|LEG1_PRICE=
```

1. `TRADER02` offers 1 at 152.00; `TRADER03` buys it. The sell trailing trigger ratchets up to 151.00 (152.00 − 1.00).
2. `TRADER02` bids for 101 at 151.00; `TRADER03` sells 1 there. The stop sells 100 at the remaining bid and cancels the 153.00 exit.
3. Explain why the trigger must not move back down on a price decline.

The explicit initial `STOP` removes ambiguity about initialization. Without it, a trailing stop needs an existing last trade from which to derive its trigger.

:material-checkbox-blank-outline: **Checkpoint:** distinguish entry versus exit OCO, stop-market versus stop-limit, and fixed versus trailing triggers. For each run, report actual fills, remaining interest and position change.

12.12 Exercise 9: Size a Pairs Trade and Close It

Exercise 1's 100 AAPL versus 50 MSFT is not dollar-neutral: at 150 and 420, the notionals are 15,000 versus 21,000. A combo supplies execution mechanics, **not** the hedge ratio or a profitable trading signal.

Choose a gross notional of 42,000 on each side. That requires buying 280 AAPL at 150 and selling 100 MSFT at 420:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=280|PRICE=150.00|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=420.00|TIF=DAY
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=NEUTRAL-IN-01|COMBO_TYPE=AON|TIF=DAY|TAG=PAIR-LAB|LEG_COUNT=2|LEG0.SYM=AAPL|LE
```

After both fill, close the **actual filled quantities** with a new, opposite-side combo:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=280|PRICE=152.00|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=422.00|TIF=DAY
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=NEUTRAL-OUT-01|COMBO_TYPE=AON|TIF=DAY|TAG=PAIR-LAB|LEG_COUNT=2|LEG0.SYM=AAPL|L
```

Calculate before reading the answer:

- Long AAPL: $280 \times (152 - 150) = +560$.
- Short MSFT: $100 \times (420 - 422) = -200$.
- Gross strategy P&L: **+360**, before fees. The combined position change over entry and exit is zero.

Equal notionals are not equal market sensitivities. Discuss how different volatility, beta and correlation could invalidate this hedge. Prices moving together do not guarantee a profit, and a pairs trade is not risk-free arbitrage.

`TAG` is a client correlation label; it does not link the two combo parents or make the exit automatic. Cancelling an already filled entry cannot close the position.

:material-checkbox-blank-outline: **Checkpoint:** justify the 280:100 ratio, prove both parents matched, and reconcile four leg fills to +360 gross P&L.

12.13 Exercise 10: Three-Leg Basket, Then Scale the Process

Model a hypothetical relative-value signal: buy AAPL against a basket of MSFT and TSLA. Use unequal quantities to balance cash notionals: $100 \times 150 = 25 \times 420 + 18 \times 250 = 15{,}000$.

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=25|PRICE=420.00|TIF=DAY
[TRADER02]> NEW|SYM=TSLA|SIDE=BUY|TYPE=LIMIT|QTY=18|PRICE=250.00|TIF=DAY
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=BASKET-01|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=3|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0
```

1. Verify all three fills and `MATCHED`.
2. Reset and repeat, but stage only **17 TSLA**, not 18. Predict the result before submitting. No leg may trade at entry in this controlled all-limit setup; the parent becomes `FAILED`. All counter-orders remain.
3. Reset and submit the same basket to **empty** books. It rests. Fill only 10 AAPL from `TRADER02`: the parent becomes `PARTIALLY_MATCHED`, even though no entire leg has finished. Record the remaining quantities. Cancel the parent and identify the residual +10 AAPL position.

Instructor extension — up to ten legs: configure seven more distinct symbols with known tick sizes and approved price ranges. Have students build a leg worksheet (symbol, side, quantity, limit, notional, available depth), then expand the all-limit basket to `LEG_COUNT=10` and `LEG0` through `LEG9`. Run it once with full liquidity and once with one missing share on the last leg. Also try `LEG_COUNT=11`: the gateway must refuse

it. A leg per symbol is mandatory; repeating a symbol is not a way to construct an options butterfly or a two-sided quote.

As a strategy variation, design a sector rotation basket with several buys and sells. A calendar-spread analogy requires two separately configured symbols; this lab does not add futures expiry, options payoffs or contract multipliers to the exchange.

:material-checkbox-blank-outline: **Checkpoint:** submit a three-leg basket, diagnose a one-share shortfall, and explain how checking each book scales to the supported two-to-ten-leg range.

12.14 Exercise 11: Execution Cost — Depth, MARKET, IOC and FOK Legs

An AON pre-check needs the **whole quantity**, not merely a visible best price. On cleared books stage this ladder:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=40|PRICE=150.00|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=60|PRICE=150.10|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=20|PRICE=420.50|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=30|PRICE=420.40|TIF=DAY
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=DEPTH-01|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY|LEG0.
```

Calculate the leg VWAPs from executions: AAPL **150.06**, MSFT **420.44**. The entry receives 21,022 − 15,006 = **6,016 cash** before fees, but it also creates a short liability. That receipt is not a profit.

Repeat with a fresh ID and restaged ladder for each variant:

Variant	Command changes	Prediction in this controlled setup
Price limit too tight	AAPL `LEG0.PRICE=150.00`	Only 40 eligible AAPL; no leg trades; parent `FAILED`
One share short	Stage 59 instead of 60 at 150.10	Only 99 AAPL; no leg trades; parent `FAILED`
Market legs	Both `.TYPE=MARKET`; remove both `.PRICE` fields	Full depth is available; both fill, but no leg price cap
IOC legs	Both `.TYPE=IOC`; keep both prices	Both fill completely when the pre-check passes
FOK legs	Both `.TYPE=FOK`; keep both prices	Both fill completely when the pre-check passes

Repeat the short-liquidity run with IOC and FOK legs too. Do **not** import the single-order IOC lesson ("take 99 and cancel 1") into an AON combo: the parent pre-check blocks execution here. Finally, submit each immediate type to empty books; unlike passive limits, those legs cannot wait, so the combo fails instead of remaining `PENDING`.

There is **no parent net-price limit**. Choose each leg's cap yourself and compute the resulting worst-case cash flow. `COMBO_TYPE` currently supports only `AON`, not a ratio-fill, net-debit or net-credit execution mode.

:material-checkbox-blank-outline: **Checkpoint:** reconcile quantities and VWAPs across multiple price levels, and explain the interaction between parent AON and each child's execution instruction.

12.15 Exercise 12: Conditional Combo Legs Are Not a Synchronized Trigger

The engine accepts `STOP` and `STOP_LIMIT` combo legs, but their triggers belong to their individual symbol books. Use them to study conditional exposure, **not** as a promise of an atomic future hedge.

Establish a one-share AAPL trade at 150.00 between `TRADER02` and `TRADER03`. Leave no other interest, then submit:

```
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=CONDITIONAL-01|COMBO_TYPE=AON|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|LEG0.SIDE=BUY
```

1. Confirm the parent is pending. The AAPL stop is dormant; the MSFT limit is working. There was no immediate opposing liquidity.
2. `TRADER02` offers 101 AAPL at 151.00; `TRADER03` buys 1 at 151.00. The print releases the 100-share buy stop, which fills from the remaining offer. MSFT has not filled. Observe `PARTIALLY_MATCHED`.
3. `TRADER02` sells 50 MSFT at 420.00. The parent now reaches `MATCHED`. Explain why this is two separate exposures in time, not an atomic event at the AAPL trigger.

Stop-limit variant: reset and re-establish AAPL's last trade at 150. Replace leg 0 with `LEG0.TYPE=STOP_LIMIT|LI`
Trigger with a one-share print at **151.20**, leaving an offer there. The released buy limit cannot pay 151.20, so it remains unfilled. Fill the MSFT leg and record the resulting unhedged position; then cancel the combo.

Do not generalize the all-limit entry experiment

A stop's trigger and subsequent liquidity are separate from the combo's submission-time depth check. Neither the parent ACK nor AON means that all conditional legs will trigger together, or that a stop will fill. The same caution applies to later auctions, halts and SMP interactions.
:material-checkbox-blank-outline: **Checkpoint:** demonstrate both dormant and triggered legs, and distinguish trigger price, limit price and actual fill price.

12.16 Exercise 13: Parent Cancel, Child Failure and Self-Match Prevention

12.16.1 A. Cancel one child, fail the strategy

Submit `PAIR-004`'s passive limits from Exercise 4 with a fresh ID to empty books. Save the child order IDs. Instead of cancelling the parent, send:

```
[TRADER01]> CANCEL|ID=<aapl_child_id>
```

The parent becomes `FAILED` and its MSFT sibling is cancelled. Contrast this with `CANCEL|COMBO_ID=...`, which produces `CANCELLED`. Repeat after a 10-share AAPL fill: failure cancels the remaining interest, not that fill.

12.16.2 B. AON entry preflight and later child matching use SMP differently

Stage an AAPL offer **owned by TRADER01**, and an MSFT bid owned by `TRADER02`:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY
[TRADER02]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=420.00|TIF=DAY
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=SMP-01|COMBO_TYPE=AON|SMP=CANCEL_AGGRESSOR|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|
```

The combo-level `SMP` applies to every child. For an AON entry, the preflight counts only liquidity the child

could actually trade against under that policy. With `CANCEL_AGGRESSOR`, the AAPL offer counts as **zero** even though 100 shares sit at the exact price: the combo is accepted, then its first leg is rejected with `INSUFFICIENT_LIQUIDITY` and the parent shows `COMBO FAILED SMP-01 Leg 0 (AAPL) REJECTED`. The MSFT child is not posted, there is no trade, and both the AAPL offer and external MSFT bid remain.

Reset and send the same combo with `SMP=CANCEL_RESTING`, then `SMP=CANCEL_BOTH`: with any policy other than `NONE` the outcome is the same, because the entry check discounts own liquidity before any SMP action could run; it does not cancel the resting offer. Now reset once more and send it with `SMP=NONE`, or omit `SMP=` when your gateway default is `NONE` (as in the configuration you wrote in chapter 01): both legs fill, the AAPL leg *against your own offer*, and the parent reaches `MATCHED`. That is a self-trade, not hedge liquidity. Do not equate an accepted parent, displayed depth or a `MATCHED` parent with genuine external execution.

Omitting `SMP=` uses the gateway's configured default (`smp_action` in the participant's block; the bundled example configurations set `CANCEL_AGGRESSOR`, a hand-written one like yours sets nothing, which means `NONE`). OCO has no per-request SMP field: its legs use the gateway default.

This entry preflight is different from a combo child that rests and later becomes marketable. To see the regular SMP actions at execution time, reset the books and submit an AON combo with its AAPL buy at 149.00, its MSFT sell at 420.00, and `SMP=CANCEL_RESTING`. After both children rest, place your own AAPL sell at 150.00, then amend the saved combo-child ID to 150.00:

```
[TRADER01]> NEW|TYPE=COMBO|COMBO_ID=LATER-01|COMBO_TYPE=AON|SMP=CANCEL_RESTING|TIF=DAY|LEG_COUNT=2|LEG0.SYM=AAPL|
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00
[TRADER01]> AMEND|ID=<aapl_child_id>|PRICE=150.00
```

The amended combo child is now the aggressor. `CANCEL_RESTING` cancels your own offer, then leaves the amended child and its MSFT sibling resting; the parent remains `PENDING` and no trade prints. Repeat from clean books with `NONE`, `CANCEL_AGGRESSOR`, and `CANCEL_BOTH`: `NONE` permits the AAPL self-trade and moves the parent to `PARTIALLY_MATCHED`; `CANCEL_AGGRESSOR` cancels the amended child and its sibling but leaves your offer; `CANCEL_BOTH` cancels the offer and both combo children. The two cancel policies leave the parent `FAILED`. These policies differ when an actual child is matched, even though the AON entry preflight gives the same result for all non-`NONE` policies.

12.16.3 C. Reprice a child without re-entering the parent

On empty books submit a fresh passive combo buying 100 AAPL at 149.00 and selling 50 MSFT at 420.00. Then `TRADER02` offers 100 AAPL at 150.00. Amend the saved AAPL child ID:

```
[TRADER01]> AMEND|ID=<aapl_child_id>|PRICE=150.00|RTAG=COMBO-REPRICE-01
```

The amended child can fill alone, moving the parent to `PARTIALLY_MATCHED`. Its sibling remains working. Complete it with a 50-share MSFT buy from `TRADER02`, or cancel the parent and reconcile the AAPL exposure.

There is no parent-wide atomic amend command. A child reprice is an ordinary order amendment, not a resubmission of the original AON package. Apply Amending Orders's priority rules, and do not assume the other child limits were changed.

:material-checkbox-blank-outline: **Checkpoint:** explain why AON entry preflight rejects own-only liquidity under any non-`NONE` policy, why an amended resting combo child applies its own SMP policy when it becomes the aggressor, and why cascade cancellation cannot roll back a previous execution.

12.17 Exercise 14: Validation and TIF — When Should Nothing Be Posted?

12.17.1 A. Reject malformed intent before trading

Starting from a valid passive combo command, change **one thing at a time**. Use a new label for every attempt and preserve the error message:

Change	What to establish
`LEG_COUNT=1` or `11`	Gateway refuses the supported-range violation
Use AAPL for both legs	Engine refuses duplicate symbols
Remove a required limit `.PRICE`	Engine refuses it: `COMBO REJ … Leg 0: LIMIT requires price`
Supply a price between that symbol's valid ticks	Gateway refuses an off-grid price; each symbol has its own tick scale
`LEG0.TYPE=TRAILING_STOP` or `ICEBERG`	Unsupported combo leg shape is refused
Set a leg quantity to zero	Invalid quantity is refused
Use an unconfigured symbol	Submission is refused, not partially posted

Combo legs support LIMIT, MARKET, IOC, FOK, STOP and STOP_LIMIT. OCO also supports TRAILING_STOP, but **not ICEBERG**. For OCO, try an immediate `MARKET`, `IOC` and `FOK` first leg against staged sufficient liquidity, paired with a passive second leg: when the first ends at entry, the second must not trade. Repeat IOC with too little liquidity and FOK with none; compare actual fills before sibling cancellation.

12.17.2 B. DAY/GTC and auction-only interest

With the scheduler stopped and sessions enabled, submit passive combos under two fresh labels, one `TIF=DAY` and one `TIF=GTC`, to empty books. Advance from `CONTINUOUS` through `CLOSING_AUCTION` to `CLOSED`. The DAY children expire and the parent fails; GTC interest survives. Cancel the GTC combo after reopening.

Now use Auctions's manual procedure:

1. In `OPENING_AUCTION`, submit an all-limit `TIF=ATO` combo to empty books. Stage counter-interest on only one symbol, then advance to `CONTINUOUS`. Record auction fills and expiries of unmatched ATO children, the parent outcome and any residual position.
2. Repeat in `CLOSING_AUCTION` with `TIF=ATC`, then advance to `CLOSED`.
3. In a no-matching phase, compare passive LIMIT legs with a combo containing MARKET, IOC or FOK. Those immediate instructions cannot wait for an auction and fail as child orders; passive interest can wait.

Do not expect a cross-symbol auction to rerun the original combo depth check as a synchronized transaction. Each book uncrosses separately. Distinguish a structural parent rejection from an accepted parent whose child later fails, and distinguish `TIF=DAY` from `LEG0.TYPE=IOC`: TIF controls lifetime; the leg type controls execution.

12.17.3 C. A halted hedge leg

Using Risk Controls's manual halt procedure, halt MSFT on an empty book with `HALT_SYM|SYM=MSFT`. Submit a fresh passive two-limit combo to empty books; its MSFT child can rest, but must not match while halted. Then, on freshly cleared books, repeat with an MSFT MARKET, IOC or FOK child: that child cannot rest and the parent fails.

Use the leg rejection to distinguish `INSTRUMENT_HALTED` from an ordinary liquidity shortfall. Resume with `RESUME_SYM|SYM=MSFT` and inspect actual fills; do not assume resuming reconstructs an atomic hedge. Restore `CONTINUOUS` and clear all interest before leaving this lab.

:material-checkbox-blank-outline: **Checkpoint:** separate gateway errors, parent rejection, child failure and expiry, using the ACKs and status events rather than a single "accepted" line.

12.18 Exercise 15: Advanced Operations — Recovery and Strategy Liquidity

This lab extends Persistence & Recovery. Use a dedicated data directory, the same configuration on restart, and a clean stop. Do not hand-edit persistence files to manufacture a result.

12.18.1 A. Recover the relationship, not just the orders

1. On empty books submit a passive two-limit combo with `COMBO_ID=RECOVER-GTC-01` and `TIF=GTC`. Also submit a passive OCO with `TIF=GTC`. Record parent labels and all child IDs.
2. Stop and restart the engine using the same data directory. Reconnect the gateways and restore `CONTINUOUS` if sessions are enabled. The restart also places the configured market-maker seed quotes again, and they would take the fill in step 4. Pull them with `KILL|GW=MM_AAPL_01`, `KILL|GW=MM_MSFT_01` and `KILL|GW=MM_TSLA_01` — not `CANCEL_SYM`, which would cancel your restored GTC orders as well.
3. Cancel the restored combo by `COMBO_ID`; both working children must disappear, proving the parent-child relationship was restored.
4. Fill one restored OCO leg completely. Its restored sibling must cancel, proving that restoring two orders also restored their OCO relationship.
5. Repeat with a **partially filled GTC combo**, recording filled quantities before restart. Complete its remaining legs afterwards and check the resulting parent status; do not resubmit the original full quantities.

Comparison: repeat the passive combo with `TIF=DAY` and restart on the same business day, without a session-close transition first. Individual DAY child orders can survive, but the **DAY combo parent is not persisted**. Do not assume `CANCEL|COMBO_ID=...` can manage those restored plain orders; reconcile and cancel them by their child IDs. This is different from the DAY expiry exercise at session close.

12.18.2 B. Provide liquidity for another participant's strategy

As `TRADER02`, post the opposite of Exercise 9's entry as a passive live combo: sell 280 AAPL at 150 and buy 100 MSFT at 420. With both legs resting, submit the entry from `TRADER01`. Both parents can complete through ordinary book matching; there is no separate combo order book.

Repeat, but have `TRADER03` take just part of one resting provider leg first. Recompute the depth before `TRADER01` submits. The provider's remaining pair is no longer a guaranteed hedge package. This is why a strategy liquidity provider needs per-leg inventory monitoring.

Instructor configuration extension: use `market_maker_combos` to seed passive strategy interest at startup. Verify and deploy the configuration before restarting. Inspect the resulting child orders and compare them with a live-submitted combo. Seeded combos have the internal owner `MM`, not the live `TRADER02` identity; a trader's `CANCEL|COMBO_ID=...` cannot cancel them. Coordinate cleanup with the admin. Do not treat startup seeding as a promise of entry-time atomic execution.

:material-checkbox-blank-outline: **Checkpoint:** prove restored GTC relationships still work, explain the DAY-parent limitation, and distinguish live combo liquidity from configuration-owned seeds.

12.19 When to Use Combos vs OCO

Use Case	Mechanism
Spread / pairs trade (buy A + sell B)	Combo
Hedging (must have both sides or neither)	Combo — but see the caveat below
Take-profit + stop-loss (only want one to execute)	OCO
Multiple entries at different prices (only want one)	OCO

AON is not a lifetime execution transaction

The entry liquidity check prevents the ordinary all-limit, short-depth case from executing only one leg. Resting fills, conditional triggers, amendments, auctions and execution restrictions can still leave leg risk. Stage external liquidity, check each leg's eligibility, and reconcile actual fills before cancelling or unwinding. Cancellation never rolls back an execution.

12.20 Reflection

An AON combo is described as "all-or-none", yet a combo can reach a state where one leg is filled and another is still resting. Explain how both statements are true at once, and name the moment at which the guarantee applies.

Why does a Combo check liquidity at entry while OCO reacts to a child reaching a terminal state? How do partial fills change the risk in each case, and why does the distinction matter for a pairs trade versus a take-profit/stop-loss pair?

For that comparison, state the assumptions under which the entry liquidity check is sufficient. Would your answer change with SMP, conditional legs or a later child amendment?

12.20.1 Strategy design challenge

Choose a pairs trade, a three-symbol rotation, or an OCO-protected long. Give another student your strategy and ask them to break its assumptions using one of the failure cases above.

Submit a short worksheet containing:

1. The thesis and a losing scenario — not just the expected profit.
2. Symbols, sides, quantities, tick-valid prices and the hedge-ratio argument.
3. Entry and exit commands, including who supplies the opposing liquidity.

4. Predicted ACKs, fills, cancellations and parent states for a successful run, a one-share liquidity shortfall and a partial-fill interruption.
5. Actual fills and position changes, gross P&L, and a plan for residual exposure if the strategy cannot complete.
6. What remains live at session close or restart, and who can cancel it.

Completion standard: the peer must reproduce the successful run and one failure case. A parent ACK, an empty `ORDERS` list or a sibling cancellation alone is not evidence that the intended strategy executed.

12.21 Further Reading

- Combo Orders
- ALF Protocol — OCO and Combo Orders

Next: 11 — Risk Controls

CHAPTER 13

Risk Controls

13.1 Objective

Configure and trigger the exchange's safety mechanisms: price collars, circuit breakers, symbol halts, and the kill switch. You will use both the interactive `pm-admin` console and the one-shot `pm-admin-cli` tool.

Background reading

- Risk Controls - The Admin Console and Exchange Commands

13.2 Prerequisites

- Chapters 01–10 completed.
- `GW_ADMIN` configured with `role: ADMIN` and connected.

13.3 Background

Risk controls prevent erroneous or manipulative orders from distorting the market:

- **Price collars** — reject orders outside static/dynamic price bands.
- **Circuit breakers** — auto-halt a symbol after violent price moves.
- **Symbol halt** — manually halt trading on one instrument.
- **Exchange halt** — halt all trading.
- **Kill switch** — cancel all orders for a specific gateway.

Administrative controls can be sent through an admin gateway session, but the preferred operator tools are:

- `pm-admin --id GW_ADMIN` — interactive admin console with tab completion.
- `pm-admin-cli --id ADMIN_CLI <command>` — one command, one response, useful for scripts, demos, and operational runbooks.

Both require a gateway configured with `role: ADMIN`. The chapter 01 configuration uses distinct IDs so both tools can stay connected at once: `GW_ADMIN` for the console and `ADMIN_CLI` for the CLI. The engine allows only one live connection per participant ID.

13.4 Exercise 1: Configure Price Collars

Collars are configured per symbol under `symbols.<SYM>.collar` (a direct override) or inherited from a named `risk_controls.levels` profile via `symbols.<SYM>.level`. Add a direct collar override to `engine_config.yaml`:

```
symbols:
  AAPL:
    collar:
      static_band_pct: 0.10    # ±10% from reference price
      dynamic_band_pct: 0.05   # ±5% from last traded price
  MSFT:
    collar:
      static_band_pct: 0.10
      dynamic_band_pct: 0.05
```

Both fields are fractions in `(0, 1)`, not whole percentages — `0.10` means 10%. Those built-in defaults (`static_band_pct: 0.20`, `dynamic_band_pct: 0.02`) fill in a field that is *missing from a `collar:` block that is present*. Leave the whole `collar:` section out and the symbol gets **no collar at all** — not a default one. `enforce_collars` (top-level, defaults to `true`) must also not be set to `false`.

A collar needs a reference price to compare against

The engine only registers a collar for a symbol once a reference price resolves — from `last_buy_price`/`last_sell_price` in the configuration, or from persisted `book_stats.json`. If those are unset, the collar is silently never enforced and the rejection below will not fire. Chapter 01's configuration seeds them; if you generated yours with `pm-config-gen`, pass `--seed-last-prices-from-mm` or set the two fields by hand.
Deploy the change, then restart the engine — editing the YAML alone changes nothing, because every process reads the compiled artifact:

```
pm-config-deploy engine_config.yaml
```

Then restart `pm-engine`.

:material-checkbox-blank-outline: **Checkpoint:** engine loads risk control configuration.

13.5 Exercise 2: Trigger a Static Collar Rejection

Place an order far outside the allowed range:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=200.00|TIF=DAY
```

If the reference price is 150.00 and the static collar is ±10%, anything above 165.00 or below 135.00 is rejected.

Expected: `REJECTED … code=COLLAR_BREACH  STATIC_COLLAR_BREACH: price 20000 ticks is outside [13500, 16500] ticks (±10% from reference 15000)` (prices are shown in ticks of 0.01).

:material-checkbox-blank-outline: **Checkpoint:** out-of-range order rejected with collar error.

13.6 Exercise 3: Configure Circuit Breakers

Circuit-breaker trigger percentages and halt durations come from a global ladder, `circuit_breaker_defaults`, applied to every symbol unless a symbol defines its own `circuit_breaker.levels` override (see the `TSLA` pattern

in The Configuration Workflow). Add a two-level ladder:

```
circuit_breaker_defaults:
  reference_window_ns: 300000000000   # 5 minutes — window the move is measured over
  levels:
    L1:
      price_shift_pct: 0.05           # halt if price moves ±5% within the window
      halt_duration_ns: 30000000000   # 30 second halt
```

`enforce_circuit_breakers` (top-level, defaults to `true`) must also not be set to `false`.

Deploy the change, then restart the engine — editing the YAML alone changes nothing, because every process reads the compiled artifact:

```
pm-config-deploy engine_config.yaml
```

Then restart `pm-engine`.

:material-checkbox-blank-outline: **Checkpoint:** circuit breaker config loaded.

13.7 Exercise 4: Trigger a Circuit Breaker

A circuit breaker does not compare a trade with the previous one. It compares it with the **average price of the trades in the last `reference_window_ns`** (5 minutes here) — on a fresh start, the opening reference price of 150.00. A gradual climb therefore has to go further than 5% before it trips, while the 5% dynamic collar from Exercise 1 stops any single jump of more than 5%. Walk the price up in steps.

First give the two traders the book to themselves: stop any `pm-mm-bot` and clear AAPL so no market-maker quote trades in between:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=AAPL
```

Then, for each price in turn — 154.00, 158.00, 161.00, 164.50 — let TRADER02 offer 100 shares and TRADER01 buy them:

```
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=154.00|TIF=DAY
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=154.00|TIF=DAY
```

Trade at	Reference (average of earlier trades in the window)	Move	Result
154.00	150.00 (opening reference)	2.7%	trades
158.00	(150 + 154) / 2 = 152.00	3.9%	trades
161.00	(150 + 154 + 158) / 3 = 154.00	4.5%	trades
164.50	(150 + 154 + 158 + 161) / 4 = 155.75	5.6%	trades, then **halts AAPL**

The trade at 164.50 still happens; the halt applies from that moment on. The engine log records it (in `pm-engine --verbose`, or the Log Operator Console):

```
CIRCUIT BREAKER HALT AAPL: level=L1 trigger=16450, ref=15575 ticks, corridor=[14017, 17132] ticks (+/-10.0%)
```

During the halt, **market-maker quote legs are cancelled**, no matching takes place, `MARKET`, `FOK` and `IOC` orders are rejected, and `LIMIT` orders are accepted and rest: a halt is the collection phase of a *reopening auction*, which uncrosses when the 30-second halt ends.

Verification drill, while the halt lasts:

1. Send a market order — it is rejected with `code=CIRCUIT_BREAKER_ACTIVE`:

   ```
   [TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
   ```

2. Send a limit order — it is accepted and rests, waiting for the reopening:

   ```
   [TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=160.00|TIF=DAY
   ```

:material-checkbox-blank-outline: **Checkpoint:** AAPL halted by the circuit breaker after the fourth step; a market order is rejected and a limit order rests until the halt ends.

13.8 Exercise 5: Open the Admin Console

In a new terminal, start the interactive admin console:

```
pm-admin --id GW_ADMIN
```

At the prompt, inspect the exchange:

```
[GW_ADMIN|ADMIN]> HELP
[GW_ADMIN|ADMIN]> SYMBOLS
[GW_ADMIN|ADMIN]> SESSION_STATUS
[GW_ADMIN|ADMIN]> GATEWAYS
```

Use `BOOK|SYM=AAPL` to confirm the console can query market state:

```
[GW_ADMIN|ADMIN]> BOOK|SYM=AAPL
```

:material-checkbox-blank-outline: **Checkpoint:** `pm-admin` authenticates as `GW_ADMIN` and can show symbols, session status, gateways, and book state.

13.9 Exercise 6: Manual Symbol Halt and Resume

From the `pm-admin` console:

```
[GW_ADMIN|ADMIN]> HALT_SYM|SYM=MSFT
```

Try trading MSFT from TRADER01:

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=MARKET|QTY=100
```

Expected: rejection — `code=INSTRUMENT_HALTED`. (A *limit* order would be accepted and rest for the reopening, as in Exercise 4.)

Resume:

```
[GW_ADMIN|ADMIN]> RESUME_SYM|SYM=MSFT
```

:material-checkbox-blank-outline: **Checkpoint:** halt prevents trading; resume restores it.

13.10 Exercise 7: Exchange-Wide Halt with pm-admin-cli

Use the one-shot CLI form when you want an operation to run from a script or checklist without opening an interactive console.

Use a separate identity for each operator client

The interactive console remains connected as `GW_ADMIN`; one-shot commands use `ADMIN_CLI`. If both tools use `GW_ADMIN`, the second connection is refused with `Gateway already connected: GW_ADMIN`.

```
pm-admin-cli --id ADMIN_CLI halt
```

All symbols stop matching. Confirm status:

```
pm-admin-cli --id ADMIN_CLI session-status
pm-admin-cli --id ADMIN_CLI symbols
```

Then resume:

```
pm-admin-cli --id ADMIN_CLI resume
```

:material-checkbox-blank-outline: **Checkpoint:** `pm-admin-cli` halts and resumes the exchange without entering a REPL.

13.11 Exercise 8: Query and Manage with pm-admin-cli

Try the read-only commands first:

```
pm-admin-cli --id ADMIN_CLI book --sym AAPL
pm-admin-cli --id ADMIN_CLI orders --gw TRADER01
pm-admin-cli --id ADMIN_CLI gateways
pm-admin-cli --id ADMIN_CLI volume
pm-admin-cli --id ADMIN_CLI schedule
```

Now halt and resume one symbol through the CLI:

```
pm-admin-cli --id ADMIN_CLI halt-sym --sym TSLA
pm-admin-cli --id ADMIN_CLI resume-sym --sym TSLA
```

:material-checkbox-blank-outline: **Checkpoint:** you can choose `pm-admin` for interactive operation and `pm-admin-cli` for repeatable one-shot commands.

13.12 Exercise 9: Exchange-Wide Halt from the Admin Console

```
[GW_ADMIN|ADMIN]> HALT
```

All symbols stop matching. Then:

```
[GW_ADMIN|ADMIN]> RESUME
```

:material-checkbox-blank-outline: **Checkpoint:** full halt and resume works across all symbols.

13.13 Exercise 10: Kill Switch

Cancel all orders for a misbehaving gateway:

```
[GW_ADMIN|ADMIN]> KILL|GW=TRADER02
```

Expected: `KILL OK TRADER02 orders=<n> quotes=<n>` — all of TRADER02's resting orders cancelled.

Or scope to a single symbol:

```
[GW_ADMIN|ADMIN]> KILL|GW=TRADER02|SYM=AAPL
```

The same operation as a one-shot CLI command is:

```
pm-admin-cli --id ADMIN_CLI kill --gw TRADER02 --sym AAPL
```

:material-checkbox-blank-outline: **Checkpoint:** kill switch cancels targeted orders.

13.14 Exercise 11: Named Risk Levels — One Policy, Many Symbols

Exercise 1 gave each symbol its own inline `collar:`. With 150 symbols that is 150 copies of the same numbers. A **risk level** is a named collar profile that symbols share: define it once under `risk_controls.levels`, pick a `default_level` for symbols that name none, and let a volatile or thinly traded symbol opt into a looser one with `level:`.

Replace the inline collars of AAPL and MSFT by this (keep `TSLA` and the other symbols as they are, with no `collar:` and no `level:`):

```
risk_controls:
  default_level: L2
  levels:
    L1:                       # loose: for volatile names
      collar:
        static_band_pct: 0.30
        dynamic_band_pct: 0.05
    L2:                       # tight: the house default
      collar:
        static_band_pct: 0.05
        dynamic_band_pct: 0.02

symbols:
  MSFT:
    level: L1                 # MSFT opts into the loose profile
```

AAPL and TSLA name no level, so both get `default_level: L2`. Check and deploy.

For repeatable predictions, reset the exchange's working state before this exercise. A same-business-day engine restart restores resting `DAY` orders (and quote legs) as well as persisted reference and last-trade prices. Deleting only `book_stats.json` resets the prices but leaves those orders, so a second run would not start from the same book. After deploying the configuration, stop every EduMatcher process (`pm-opctl-cli stop` for managed processes and `Ctrl-C` for any manually started ones), then run:

```
pm-opctl-cli clear --state
```

This removes persisted orders and prices along with statistics, clearing, and index state, but preserves configuration, logs, audit data, and settlement records. It does not remove the deployed configuration or require redeployment. See Starting over for the reset scopes. Restart the exchange and its consoles, reopen the

market, and continue below.

The reference prices are now 150, 420 and 250 again (from your configuration) and there is no last trade. **Predict** the outcome of each order before sending it. Use buys below the market, which rest instead of trading, so that only the *static* band is being tested: **Predict** the outcome of each order before sending it. Use buys below the market, which rest instead of trading, so that only the *static* band is being tested:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=143.00|TIF=DAY
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=142.00|TIF=DAY
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=300.00|TIF=DAY
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=290.00|TIF=DAY
[TRADER01]> NEW|SYM=TSLA|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=240.00|TIF=DAY
[TRADER01]> NEW|SYM=TSLA|SIDE=BUY|TYPE=LIMIT|QTY=1|PRICE=237.00|TIF=DAY
```

Order	Band that applies	Result
AAPL 143.00	L2 (default), ±5% of 150 = 142.50–157.50	accepted
AAPL 142.00	same	`COLLAR_BREACH` — price 14200 ticks is outside [14250, 15750]
MSFT 300.00	L1, ±30% of 420 = 294–546	accepted
MSFT 290.00	same	`COLLAR_BREACH` — outside [29399, 54600]
TSLA 240.00	L2 (default), ±5% of 250 = 237.50–262.50	accepted
TSLA 237.00	same	`COLLAR_BREACH` — outside [23750, 26250]

MSFT's 290.00 would have been rejected under L2 as well, and AAPL's 142.00 would have passed under L1: the level is the whole difference. An inline `collar:` on a symbol still overrides its level field by field, which is how `TSLA` in The Configuration Workflow sets only a tighter `dynamic_band_pct`.

:material-checkbox-blank-outline: **Checkpoint:** you predicted all six results before sending the orders, and can say which symbols the single line `default_level: L2` governs.

13.15 Risk Control Summary

Control	Scope	Trigger	Effect
Static collar	Per symbol (inline or via a named level)	Order price vs reference	Order rejected
Dynamic collar	Per symbol	Order price vs last trade	Order rejected
Circuit breaker	Per symbol	Price move vs. average of recent trades	Symbol halted, then reopened by auction
Symbol halt	Per symbol	`pm-admin` / `pm-admin-cli`	Trading paused

Control	Scope	Trigger	Effect
Exchange halt	All symbols	`pm-admin` / `pm-admin-cli`	All trading paused
Kill switch	Per gateway	`pm-admin` / `pm-admin-cli`	All orders cancelled

13.16 Starting over

Stop every EduMatcher process before resetting data. The clear command only warns when it detects running `pm-*` processes; it does not block deletion, and `--yes` bypasses the confirmation prompt. Live processes may recreate or write files during or immediately after a reset.

Reset	Command/effect	Preserved
Reset only remembered reference/last-trade prices	Stop `pm-engine`, remove `$EDUMATCHER_DATA_DIR/book_stats.json` (container shell: `/data/book_stats.json`), then restart the engine.	Orders, quotes, clearing/statistics history, logs, audit, and configuration.
Clear exchange working state	After stopping every process, run `pm-opctl-cli clear --state`. This removes `book_stats.json`, persisted GTC orders and combos, `stats.db`, `clearing.db`, and `indexes/`. Use `--yes` only when intentionally skipping the confirmation.	Configuration, audit/log data, EOD records, settlements, and backups.
Clear all managed data	After stopping every process, run `pm-opctl-cli clear --all`. This removes all `--state` data plus the audit trail/index, log database/fallback logs, monitor metrics, operator process tracking, and EOD database/settlement/backup data.	Deployed configuration under `ref_data/` and `emo-config.yaml`.

Neither clear mode removes the deployed configuration or requires redeployment; restart the processes after clearing and they will load the existing deployed configuration. `--all` permanently deletes EOD balances and settlement records, so use it only when those records are intentionally disposable. For a fresh exchange with money records retained, use `--state`.

The reset-scope regression test creates disposable files and directories for both modes, verifies exactly which paths are removed, and confirms that configuration remains. Active-process warning and non-interactive refusal behavior is covered separately by the CLI tests.

13.17 Reflection

Why does a circuit breaker halt the whole symbol rather than just rejecting new orders like the collars do? What failure mode (think: a runaway algorithmic trader or a bad data feed) is a circuit breaker specifically designed to stop that per-order collars cannot?

13.18 Further Reading

- Risk Controls
- Controlling the Exchange
- Processes, Environment and Ports
- Drop Copy
- A Full Trading Day

Next: 12 — P&L & Clearing

CHAPTER 14

P&L & Clearing

14.1 Objective

Understand how the DB-backed clearing process tracks positions, computes VWAP average cost, and reports realized and unrealized P&L per trader per symbol. Gain hands-on proficiency with every `pm-clearing-cli` command verb — during the trading day, at the close, and afterwards — and learn which of them are meaningful in which of those three states.

This chapter has three phases — check which one you are in

Clearing is the one subsystem whose most important behaviour only happens when the exchange **stops**. End-of-day marks are applied when the engine broadcasts `system.eod`, and it broadcasts that exactly once, during a graceful shutdown. So this chapter deliberately runs the exchange, then closes it, then keeps working on the data it left behind.
Each part below is labelled with the state it needs. If an exercise returns nothing, the first thing to check is whether the engine is in the state that part assumes.

Part	Engine	Why
A — Intraday (Ex 1–10)	**Running**, with trading activity	You are watching positions and P&L change as trades arrive
B — End of day (Ex 12–16)	**Stopped** — cleanly	The EOD sentinel and the official daily rollup only exist after `system.eod`
C — Ongoing operations (Ex 17–18)	Either	Retention and restart behaviour are independent of a live session

`pm-clearing` itself keeps running throughout — it is the *engine* that stops between Part A and Part B.

Background reading

- Positions and P&L

14.2 Prerequisites

- Chapters 01–11 completed.
- Live trading activity available (manual trading, AI traders, or both).

Dates in this chapter

The commands use `$(date +%F)` — today's date — rather than a fixed one, so they return your own session's rows. If a query comes back empty, check with `pm-clearing-cli dates` which trading dates actually have data.

14.3 Background

14.3.1 What pm-clearing stores

`pm-clearing` subscribes to `trade.executed` events and persists all state in `clearing.db` (SQLite WAL mode). There is no CSV artifact — everything is in the database.

Table	Purpose
`trade_events`	Append-only trade audit log
`gateway_symbol_positions`	Running live position per gateway/symbol
`gateway_daily_summary`	Daily rollup aggregates
`session_events`	EOD sentinel rows written on `system.eod`
`gateway_sessions`	Gateway connect/disconnect history

14.3.2 What pm-clearing maintains per position

- **net_qty** — signed quantity (positive = long, negative = short, 0 = flat)
- **avg_cost** — VWAP of entry prices (per unit; never multiplied by quantity)
- **realized_pnl** — profit/loss locked in by closing or crossing trades
- **unrealized_pnl** — paper profit/loss: `net_qty × (mark_price − avg_cost)`

14.3.3 Position state machine

A position starts flat and transitions as fills arrive:

```
Flat → Long  (BUY fill opens a long)
Flat → Short (SELL fill opens a short)
Long → Flat  (SELL fill closes the full position, close_qty == net_qty)
Long → Short (SELL fill exceeds net_qty — cross-zero)
Short → Flat (BUY fill closes the full position)
Short → Long (BUY fill exceeds abs(net_qty) — cross-zero)
```

A cross-zero fill realizes P&L on the closing portion and sets `avg_cost` to the fill price for the newly-opened side.

14.4 Part A — Intraday exercises

Engine: running

Everything in Part A assumes a live exchange with the market open and a market maker quoting AAPL — the market orders below need someone to trade with. Chapters 05–11 cleared AAPL several times; re-quote it from `MM_AAPL_01` (the chapter 02 `QUOTE` command) or start `pm-mm-bot --symbol AAPL --gap 0.10 --qty 500`. Prefer the bot: a manual quote under `INACTIVATE_ON_ANY_FILL` pulls its other side on the first fill (Chapter 03), so with a manual quote you must re-send `QUOTE` before every market order that needs the opposite side — the sell orders in Exercises 5 and 6 would otherwise find no bid and be cancelled unfilled.

14.4.1 Exercise 1: Start the clearing service

If `pm-clearing` is still running from chapter 03, keep it and skip to Exercise 2 — two copies would record every trade twice. Otherwise:

```
pm-clearing
```

`pm-clearing` logs at WARNING level by default, so it starts silently. To see that it is up, add `-v`:

```
pm-clearing -v
```

The last startup line reads (prefix and timestamp vary):

```
pm-clearing waiting for pm-engine events
```

:material-checkbox-blank-outline: **Checkpoint:** clearing service is running.

14.4.2 Exercise 2: Check DB health

In a second terminal:

```
pm-clearing-cli health
```

Expected: `pm-clearing process: OK`, `clearing ZMQ ping: OK` and `health: OK`. The database line says `skipped` — the database itself is only checked when you ask for it:

```
pm-clearing-cli health --extended --verbose
```

Expected: `clearing database: OK`, followed by the DB path, row counts for each table, the last trade and last flush timestamps, and `journal mode: wal`.

Run `health --extended --verbose` again after a few trades to see the row counts grow.

:material-checkbox-blank-outline: **Checkpoint:** `health` reports `OK`, and `--extended --verbose` shows WAL journal mode.

14.4.3 Exercise 3: Build a long position and query it

From TRADER01:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=200
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

Query the position:

```
pm-clearing-cli positions --gateway TRADER01 --symbol AAPL
```

Confirm (if TRADER01 already held AAPL from earlier chapters, add that to the figures):

- `net_qty` 300
- `avg_cost` is the VWAP of the two fills (not the last price alone)
- `buy_qty` is 300, `sell_qty` is 0

Query all open positions across every gateway and symbol:

```
pm-clearing-cli positions
```

:material-checkbox-blank-outline: **Checkpoint:** `positions` shows the AAPL long for TRADER01 with correct VWAP avg_cost.

14.4.4 Exercise 4: Inspect the raw trade event log

```
pm-clearing-cli trades --gateway TRADER01 --symbol AAPL --limit 10
```

Each row is one matched fill. Observe: `id`, `trade_date`, `symbol`, `quantity`, `price`, `tick_decimals`, `buy_gateway_id`, `sell_gateway_id`, `aggressor_side`.

Switch format to JSON:

```
pm-clearing-cli --format json trades --gateway TRADER01 --symbol AAPL --limit 3
```

Export all trades for a date to CSV for a spreadsheet:

```
pm-clearing-cli --format csv trades --date "$(date +%F)" > trades_today.csv
```

:material-checkbox-blank-outline: **Checkpoint:** you can retrieve fills in table, JSON, and CSV formats.

14.4.5 Exercise 5: Realize P&L by partially closing

Sell part of the AAPL position:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=100
```

Expected accounting:

```
realized_pnl += (sell_price - avg_cost) x 100
net_qty = 200
```

Query:

```
pm-clearing-cli pnl --gateway TRADER01 --symbol AAPL
```

Confirm `realized_pnl` is non-zero and `total_pnl = realized_pnl + unrealized_pnl`.

Query the exchange-wide P&L across every gateway in one row per gateway:

```
pm-clearing-cli pnl
```

:material-checkbox-blank-outline: **Checkpoint:** `pnl` shows realized P&L on the partial close.

14.4.6 Exercise 6: Cross-zero position

This is the most complex accounting path. Starting from the 200-share AAPL long:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=300
```

This single order closes all 200 long shares **and** opens a 100-share short.

Expected accounting after the cross:

```
realized_pnl += (sell_price - avg_cost) x 200   ← the closing portion
net_qty      = -100                              ← new short side
avg_cost     = sell_price                        ← reset to open price of new side
```

Verify:

```
pm-clearing-cli positions --gateway TRADER01 --symbol AAPL
pm-clearing-cli pnl      --gateway TRADER01 --symbol AAPL
```

Close the short:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

```
pm-clearing-cli pnl --gateway TRADER01 --symbol AAPL
```

Position should return to flat (`net_qty = 0`) with `realized_pnl` reflecting both the original long close and the short round-trip.

:material-checkbox-blank-outline: **Checkpoint:** cross-zero P&L is realized correctly on the closing portion.

14.4.7 Exercise 7: Exposure — net and gross notional

With a few positions open across symbols:

```
pm-clearing-cli exposure
```

Default sort is `gross_notional` (largest exposure first). Try other sorts:

```
pm-clearing-cli exposure --sort total_pnl
pm-clearing-cli exposure --sort net_notional
```

Fields include `net_qty`, `mark_price`, `net_notional`, `gross_notional`, `realized_pnl`, `unrealized_pnl`, `total_pnl`.

This is the clearing house **risk concentration view**: a large `gross_notional` with small `net_notional` means a participant is running offsetting positions that could unwind rapidly.

:material-checkbox-blank-outline: **Checkpoint:** you can rank all positions by exposure and P&L.

14.4.8 Exercise 8: Gateway-level P&L summary

To see every participant's P&L in a single query:

```
pm-clearing-cli gateways
```

Output: one row per gateway with `realized_pnl_total`, `unrealized_pnl_total`, `total_pnl`, `net_qty_total`.

Filter to one participant:

```
pm-clearing-cli gateways --gateway TRADER01
```

Export for downstream reporting:

```
pm-clearing-cli --format csv gateways > gateway_pnl.csv
pm-clearing-cli --format json gateways > gateway_pnl.json
```

This is the clearing house **top-level risk snapshot**: one row per participant, usable without knowing which symbols they traded.

:material-checkbox-blank-outline: **Checkpoint:** `gateways` returns a P&L summary row for each active participant.

14.4.9 Exercise 9: Symbol-level clearing totals

Query clearing activity per symbol across all gateways:

```
pm-clearing-cli symbols
```

Sort by traded volume or P&L:

```
pm-clearing-cli symbols --sort traded_qty
pm-clearing-cli symbols --sort realized_pnl
```

Fields include `symbol`, `traded_qty`, `traded_notional`, `realized_pnl`, `open_net_qty`, `open_unrealized_pnl`.

`traded_qty` and `traded_notional` here are **gross participant-side activity**: a trade is included once for the buyer and once for the seller. For example, one 800-share AAPL trade contributes 800 to each gateway's `daily` row and 1,600 to the symbol total. This is not market volume; use `pm-stats-cli daily` for volume counted once per executed trade.

:material-checkbox-blank-outline: **Checkpoint:** `symbols` shows per-symbol gross participant activity and P&L across all participants; `pm-stats-cli daily` counts each market trade once.

14.4.10 Exercise 10: Normalized vs raw-output

By default, price-derived fields are divided by `10^tick_decimals` to produce display-currency units. Use `--raw-output` to see the raw integer tick values:

```
pm-clearing-cli positions --gateway TRADER01 --symbol AAPL
pm-clearing-cli --raw-output positions --gateway TRADER01 --symbol AAPL
```

Rules: - `avg_cost` **is** normalized like the other price columns. It holds fractional ticks, so without normalization it would render 100x off next to `mark_price` - Fields like `mark_price`, `realized_pnl`, `unrealized_pnl`, `buy_notional`, `sell_notional` are normalized by default

Use `--raw-output` when piping into scripts that expect tick integers, or when debugging a suspected normalization issue.

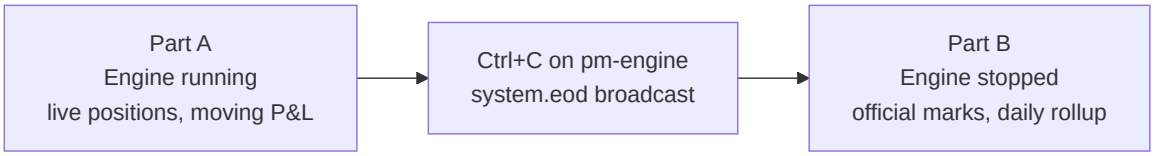
:material-checkbox-blank-outline: **Checkpoint:** you can explain which fields are normalized and why `--raw-outpu` exists.

14.5 Interlude — Closing the trading day

Everything in Part A was a *running* view: positions and unrealized P&L that change with the next trade. Part B is about the *official* numbers — the ones a clearing house would settle on. Those do not exist yet, and this section is how you create them.

14.5.1 Why the engine has to stop

The engine broadcasts `system.eod` from its **graceful shutdown path** and nowhere else. There is no "run EOD now" command: closing the session moves the phase to `CLOSED`, but the end-of-day marks are applied when the engine exits cleanly. That single message is what turns a live position into a settled one.



14.5.2 Exercise 11: Close the day

Do these in order. The order matters: `pm-clearing` must still be running when the engine sends `system.eod`, or the message has nobody to receive it.

1. Note your open positions, so you can compare them with the settled numbers afterwards:

```
pm-clearing-cli positions > /tmp/positions-before-eod.txt
cat /tmp/positions-before-eod.txt
```

2. Close the session from the operator console:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

3. Stop the engine cleanly. Press `Ctrl+C` in the `pm-engine` terminal — **not** `kill -9`. Wait for the shutdown messages; the engine saves GTC state, book statistics, and then broadcasts EOD.

4. Leave `pm-clearing` running. It receives `system.eod` and applies the marks. Watch its terminal: you should see it flush and write.

A hard kill skips EOD entirely

`kill -9` gives the engine no chance to run its shutdown path, so no `system.eod` is ever sent, no marks are applied, and every exercise in Part B returns nothing. If that happens, restart the engine, generate a trade or two, and close it properly this time.
:material-checkbox-blank-outline: **Checkpoint:** the engine has exited cleanly, `pm-clearing` is still running, and its log shows it handled the EOD message.

14.6 Part B — End-of-day exercises

Engine: stopped (cleanly). pm-clearing still running.

If you skipped the Interlude, go back and do it — none of the rows these exercises query exist until the engine has broadcast `system.eod`.
End-of-day (EOD) is when the clearing house finalizes marks, settles daily P&L, and prepares tomorrow's opening state. `pm-clearing` handles this automatically when the engine sends `system.eod`, but as a clearing operator you need to verify and audit the result.

14.6.1 What happens at EOD

When `pm-clearing` receives `system.eod`:

1. **Force-flushes** any buffered trades immediately
2. Applies official EOD **mark-to-market** using last-trade price (or mid-price if no trade occurred) to update `mark_price` and `unrealized_pnl`
3. Writes updated positions to `gateway_symbol_positions` so `end_unrealized_pnl` in `gateway_daily_summary` reflects the official EOD mark
4. Inserts an `EOD` sentinel row into `session_events`

14.6.2 Exercise 12: Verify the EOD sentinel

The sentinel is your proof that marks were applied — the first thing to check before trusting any settled number:

```
pm-clearing-cli eod --limit 5
```

Each row shows the timestamp and a `payload_json` containing the mark prices applied. The `eod` sentinel is proof that marks were applied.

:material-checkbox-blank-outline: **Checkpoint:** `eod` returns at least one row with mark prices in `payload_json`.

14.6.3 Exercise 13: Inspect the daily rollup, and compare it with Part A

Query the official daily summary for today:

```
pm-clearing-cli daily --date "$(date +%F)"
```

Now compare `end_unrealized_pnl` here with the `unrealized_pnl` you saved in Exercise 11:

```
pm-clearing-cli positions
diff /tmp/positions-before-eod.txt <(pm-clearing-cli positions) || true
```

The numbers may differ, and understanding *why* is the point of this part. Before EOD, `unrealized_pnl` was marked against whatever the last trade happened to be at that instant. After EOD it is marked against the official close — the last trade of the session, or the mid if the symbol never traded. Only the second one is a number a clearing house would settle on.

:material-checkbox-blank-outline: **Checkpoint:** you can point at one position whose mark changed at EOD,

and say which price each version was marked against.

Filter to one gateway:

```
pm-clearing-cli daily --date "$(date +%F)" --gateway TRADER01
```

Key fields: - `traded_qty`, `traded_notional` — this gateway's activity for the day; across gateways the same trade contributes once to each counterparty - `buy_qty`, `sell_qty`, `buy_notional`, `sell_notional` — side breakdown - `end_net_qty`, `end_avg_cost`, `end_unrealized_pnl` — official EOD position state

Export the day's summary for settlement reporting:

```
pm-clearing-cli --format csv daily --date "$(date +%F)" > daily_settlement_$(date +%F).csv
```

Query across multiple days:

```
pm-clearing-cli daily --from "$(date +%F)" --to "$(date +%F)" --gateway TRADER01
```

:material-checkbox-blank-outline: **Checkpoint:** daily rollup contains `end_*` fields populated by the EOD mark pass.

14.6.4 Exercise 14: Browse available trading dates

```
pm-clearing-cli dates
```

Add volume and net-amount totals per date:

```
pm-clearing-cli dates --with-totals
```

Filter by symbol to see on which days that symbol traded:

```
pm-clearing-cli dates --symbol AAPL
```

:material-checkbox-blank-outline: **Checkpoint:** you can navigate which dates have clearing data.

14.6.5 Exercise 15: Reconciliation check

After EOD, verify that raw `trade_events` aggregates match `gateway_daily_summary`:

```
pm-clearing-cli reconcile --from "$(date +%F)" --to "$(date +%F)"
```

Expected: - `OK — no discrepancies found.` when consistent - Rows showing side / date / gateway / symbol / quantity-diff / notional-diff if not

Reconcile across the last week (the `date -d` form is GNU `date`; on macOS use `date -v-7d +%F`):

```
pm-clearing-cli reconcile --from "$(date -d '7 days ago' +%F)" --to "$(date +%F)"
```

If discrepancies appear, use `trades` to investigate the affected gateway/symbol/date.

:material-checkbox-blank-outline: **Checkpoint:** you can run a full reconciliation and interpret the result.

14.6.6 Exercise 16: Session history

Query gateway connect and disconnect events recorded during the session:

```
pm-clearing-cli sessions
```

Show only sessions that have not yet disconnected (still open):

```
pm-clearing-cli sessions --connected-only
```

This is the operational audit trail for: who connected, when, and whether they disconnected cleanly or the engine was killed unexpectedly.

:material-checkbox-blank-outline: **Checkpoint:** `sessions` returns at least one row per gateway that connected today.

14.7 Part C — Ongoing operations

Engine: either

Retention and restart behaviour do not depend on a live session. You can do these with the exchange still down, or after restarting it.

14.7.1 Exercise 17: Data retention and pruning

`pm-clearing` prunes old `trade_events` rows on startup and at every close of the market. The default window is 90 days. Control it with `--retention-days`:

```
# Start clearing with 30-day retention
pm-clearing --retention-days 30
```

Use `pm-clearing-cli prune` for on-demand pruning without restarting:

```
# Dry run — see how many rows would be deleted
pm-clearing-cli prune --days 30 --dry-run

# Actually prune and VACUUM
pm-clearing-cli prune --days 30
```

Use `--dry-run` first to avoid unintended data loss. `prune` is the only `pm-clearing-cli` verb that writes to the database.

:material-checkbox-blank-outline: **Checkpoint:** you can run a dry-run prune and interpret the row count.

14.7.2 Exercise 18: Reopen the exchange and confirm the settled day survives

The last thing a clearing operator needs to trust is that yesterday's settled numbers are still there tomorrow, and that a new session does not disturb them.

1. Restart the exchange — engine, scheduler, gateways and market makers, as in Chapter 01 (restart your consoles too, and reopen the market). `pm-clearing` can keep running throughout, or be restarted; both work, because the state lives in `clearing.db`, not in memory.

2. Confirm the closed day is intact:

```
pm-clearing-cli dates
pm-clearing-cli daily --date "$(date +%F)"
```

The row you inspected in Exercise 13 should be unchanged.

3. Trade once, then look again. Submit a single order from a trader console and let it fill, then:

```
pm-clearing-cli positions
pm-clearing-cli daily --date "$(date +%F)"
```

`positions` moves — that is the new session's live state. The settled `end_unrealized_pnl` from the closed day does not change retroactively; the rollup accumulates the new activity into the current trading date.

:material-checkbox-blank-outline: **Checkpoint:** the settled daily row survives the restart, and you can explain which numbers are historical (fixed) and which are live (moving).

This is the whole mental model of the chapter

`positions` and `pnl` answer *"where do we stand right now?"* and change with every fill. `daily`, `eod` and `sessions` answer *"what did we settle for that day?"* and are written once. Knowing which question a verb answers tells you whether the engine needs to be running for it to mean anything.

14.8 Key Formulas

Metric	Formula
Average cost (long)	$\frac{\sum(\text{buy_price} \times \text{buy_qty})}{\sum \text{buy_qty}}$
Realized P&L (closing sell)	$(\text{sell_price} - \text{avg_cost}) \times \text{qty_closed}$
Unrealized P&L (long)	$(\text{mark_price} - \text{avg_cost}) \times \text{net_qty}$
Unrealized P&L (short)	$(\text{avg_cost} - \text{mark_price}) \times \lvert\text{net_qty}\rvert$
Cross-zero realized	$(\text{fill_price} - \text{avg_cost}) \times \lvert\text{old_net_qty}\rvert$

14.9 pm-clearing-cli verb reference

The **Answers** column is the one to internalise: a *live* verb changes with the next fill, a *settled* verb is written once and then never moves.

Verb	Answers	What it returns	Key options
`gateways`	live	One row per gateway: total realized, unrealized, total P&L	`--gateway`, `--limit`

Verb	Answers	What it returns	Key options
`positions`	live	Full live position state per gateway/symbol	`--gateway`, `--symbol`, `--limit`
`pnl`	live	Focused P&L view (no qty/notional detail)	`--gateway`, `--symbol`, `--limit`
`exposure`	live	Net/gross notional exposure, sorted by size	`--gateway`, `--symbol`, `--sort`, `--limit`
`health`	live	DB row counts, last flush, WAL mode	—
`trades`	historical	Raw trade-level audit log	`--gateway`, `--symbol`, `--date`, `--from`, `--to`, `--limit`
`symbols`	historical	Symbol-level volume, notional, P&L	`--date`, `--from`, `--to`, `--sort`, `--limit`
`dates`	historical	Available trading dates	`--gateway`, `--symbol`, `--from`, `--to`, `--limit`, `--with-totals`
`sessions`	historical	Gateway connect/disconnect history	`--gateway`, `--from`, `--to`, `--limit`, `--connected-only`
`daily`	**settled**	Daily rollup + EOD snapshots	`--gateway`, `--symbol`, `--date`, `--from`, `--to`, `--limit`
`eod`	**settled**	EOD sentinel rows with mark prices	`--from`, `--to`, `--limit`
`reconcile`	historical	Raw vs summary discrepancies	`--gateway`, `--symbol`, `--from`, `--to`, `--retention-days`
`prune`	maintenance	Delete old `trade_events` + VACUUM (**writes**)	`--days`, `--dry-run`

The two **settled** verbs are the ones that return nothing until the engine has shut down cleanly at least once — that is the whole reason this chapter has an Interlude in the middle.

Global options apply to all verbs and must be given **before** the verb: `--format table|json|csv`, `--no-header`, `--raw-output`, `--datapath PATH`, `--db-name NAME`.

```
# checkdocs: ignore  — the second line is a deliberate counter-example
pm-clearing-cli --format json positions      # correct
pm-clearing-cli positions --format json      # argparse error: unrecognized arguments
```

14.10 Reflection

Why does realized P&L use the trade price at the moment of the closing fill, while unrealized P&L keeps recalculating against the current mark? In the cross-zero scenario, why is realized P&L computed only on the closing portion and not the new opening portion?

At end-of-day, which three `pm-clearing-cli` commands would you run in order to: (1) confirm the EOD mark was applied, (2) export the official daily settlement file, and (3) check the audit integrity of the day's clearing?

An operator reports that `daily` returns no rows for yesterday, even though `trades` clearly shows yesterday's activity. Give two different explanations that are both consistent with that evidence, and say which command you would run to tell them apart.

14.11 Further Reading

- Positions and P&L
- Messages
- Statistics and Reporting
- Your First Trade
- 20 — Closing the Day — turning these numbers into a daily settlement with `pm-eod`

Next: 13 — Market Data & Drop Copy

CHAPTER 15

Market Data & Drop Copy

15.1 Objective

Observe real-time market data with the current EduMatcher tools and understand where the engine's drop-copy feed fits into the architecture.

Background reading

- Drop Copy - DC Gateway - DC Spy CLI

15.2 Prerequisites

- Chapters 01–12 completed.
- Engine and at least one trader gateway active so events are visible.

15.3 Background

EduMatcher publishes two event streams from `pm-engine`:

- **PUB :5556** — primary market data: books, trades, session state, order events.
- **PUB :5557** — drop-copy feed: per-participant fill events for risk, compliance, and back-office consumers.

There is no separate `pm-drop-copy` process. The engine binds the drop-copy feed itself when it starts.

15.4 Exercise 1: Open a Live Book Viewer

Start a live viewer for AAPL:

```
pm-viewer --symbol AAPL --depth 10
```

In a trader gateway, place or cancel a resting order and watch the viewer update:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.80|TIF=DAY
```

:material-checkbox-blank-outline: **Checkpoint:** `pm-viewer` changes when the book changes.

15.5 Exercise 2: Run the Cross-Gateway Order Monitor

Start the order monitor:

```
pm-orders
```

Now place orders from both `TRADER01` and `TRADER02`. The monitor should show resting order state across gateways.

:material-checkbox-blank-outline: **Checkpoint:** `pm-orders` shows orders from multiple gateways.

15.6 Exercise 3: Capture Events with pm-audit

Start the audit logger in terminal mode:

```
pm-audit --terminal
```

Execute a trade:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

Watch `pm-audit` print the resulting events. This is the easiest training-safe way to observe the event stream without writing a custom ZMQ subscriber.

:material-checkbox-blank-outline: **Checkpoint:** audit output shows the trade/order lifecycle events.

15.7 Exercise 4: Confirm the Drop-Copy Feed Is Bound

Restart `pm-engine --verbose` (and then your trader and operator consoles, and reopen the market) and look for this startup line in the engine output (preceded by the usual timestamp, level and logger name):

```
Drop copy PUB bound on port 5557
```

That confirms the drop-copy publisher is active. It is intended for external risk/compliance subscribers and publishes topics such as `drop_copy.event.<gateway_id>`.

:material-checkbox-blank-outline: **Checkpoint:** you can identify the drop-copy socket in engine startup output.

15.8 Exercise 5: Watch the Drop-Copy Feed with `pm-dc-spy`

`pm-audit`, `pm-viewer` and `pm-orders` all read the *public* feed on port 5556 — none of them show the drop-copy feed. EduMatcher ships two tools for that, and it is worth knowing both:

Tool	What it is	Use it for
`pm-dc-spy`	A CLI that subscribes to the engine's drop-copy PUB socket (5557)	Looking at the feed yourself, right now

Tool	What it is	Use it for
`pm-dc-gwy`	A **gateway process** serving the DC1 protocol over TCP 5590	Letting an *external* system consume drop copy, the way a real back office would

Start the spy:

```
pm-dc-spy --format human
```

Then submit a trade from a trader console and watch the fill events appear.

:material-checkbox-blank-outline: **Checkpoint:** `pm-dc-spy` prints a fill event for a trade you caused, and you can say why an external consumer would use `pm-dc-gwy` on 5590 instead of subscribing to 5557 directly.

15.9 Exercise 5b: Subscribe to the Raw Socket

Understanding what the tools do for you is worth one level down. Connect a minimal ZMQ subscriber to the same socket `pm-dc-spy` reads:

The snippet needs the `zmq` Python package (`pyzmq`); in a container shell or a Poetry checkout use `python`/`poetry run python`, and with `pipx` use `pipx runpip edumatcher`'s interpreter or `pip install pyzmq` in a virtual environment.

```
python - <<'PY'
import json
import zmq

ctx = zmq.Context()
sock = ctx.socket(zmq.SUB)
sock.connect("tcp://127.0.0.1:5557")
sock.subscribe(b"drop_copy.event.")

print("listening on drop_copy.event.*")
for _ in range(10):
    topic, payload = sock.recv_multipart()
    msg = json.loads(payload)
    print(topic.decode(), "seq=", msg.get("seq"), "gateway=", msg.get("gateway_id"),
            "event_type=", msg.get("event_type"), "trade_ids=", msg.get("trade_ids"))
PY
```

While this is running, execute a trade from `TRADER01` in another terminal:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

You should see one `drop_copy.event.<gateway_id>` message per participant on each side of the trade, stamped with a `seq` that counts that gateway's own events (each gateway has its own counter starting at 1, so the two sides of one trade carry numbers from different streams). Each message also carries `trade_ids`, the durable public trade ID that lets a risk or clearing consumer correlate its private execution with `trade.executed`.

:material-checkbox-blank-outline: **Checkpoint:** you received at least one drop-copy message directly from port 5557, distinct from anything `pm-audit` printed on port 5556.

15.10 Exercise 6: Compare Public Events and Drop-Copy Purpose

Execute another trade and compare what each consumer is for:

Table 15.2: material-checkbox-blank-outline: **Checkpoint:** explain why drop-copy is separate from the public market-data stream.

Consumer	Source	Purpose
`pm-viewer`	PUB :5556	Human-readable book view
`pm-orders`	PUB :5556	Cross-gateway resting order monitor
`pm-audit --terminal`	PUB :5556	Full event stream for inspection/logging
External drop-copy client	PUB :5557	Per-participant fill feed for risk/compliance

15.11 Exercise 7: Launch the Market Board

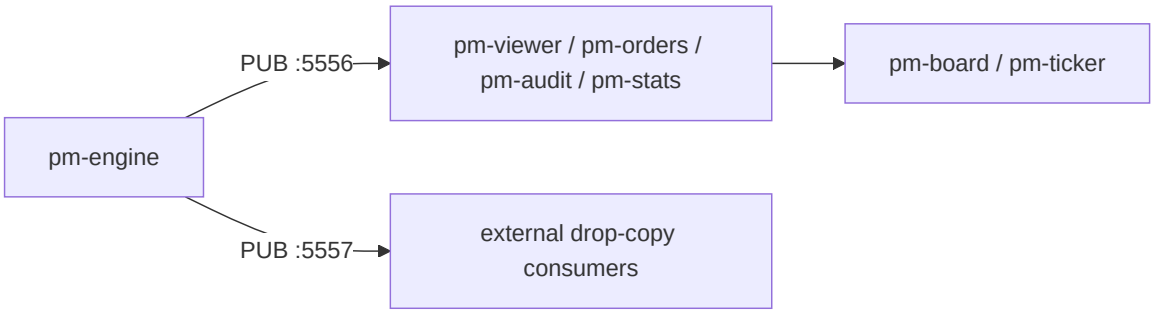
Start the multi-symbol dashboard:

```
pm-board --rows 8 --interval 10
```

`pm-board` is a pure market-data subscriber: it builds its table from the live `book.*` and `trade.executed` topics only and does not read `stats.db`. It auto-rotates pages every `--interval` seconds; Enter, PgUp and PgDn page manually.

:material-checkbox-blank-outline: **Checkpoint:** board shows AAPL, MSFT, and TSLA in one view.

15.12 Key Architecture



15.13 Reflection

Why does the engine publish drop-copy events on a **separate** socket (`:5557`) rather than mixing them into the same `:5556` PUB feed that viewers, orders, audit, and stats all subscribe to? What operational problem

would arise for a compliance drop-copy consumer if it shared a socket with high-volume book/viewer traffic?

15.14 Further Reading

- Messages
- Drop Copy
- Drop-Copy Gateway (`pm-dc-gwy`)
- Drop-Copy Spy CLI (`pm-dc-spy`)
- Processes, Environment and Ports
- CALF Protocol Reference
- Market Data Feed

Next: 14 — AI Traders & News

For a fuller hands-on tour of every viewer and observer process, see 18 — Exchange Observer Processes.

CHAPTER 16

AI Traders & News

16.1 Objective

Run a swarm of AI traders over a market whose prices have a hidden *true value* behind them, then move that market with news: trade an earnings surprise, decide what to do with a rumour, watch a retraction unwind, and see how a breakthrough and a factory fire move two rival chipmakers.

Background reading

- AI Traders - Running the AI-Trader Swarm - The Market Model (`pm-market-sim`)

16.2 Prerequisites

- Chapters 01–13 completed.
- The `s150-nominal` example deployed (Exercise 1). It registers the AI traders `AI001`–`AI020`, the student desks `TRADER01`–`TRADER10`, the instructor `OPS01` (ADMIN) and the market maker `MM01`, and it ships the market model's `market_sim.yaml` and the swarm's `swarm.yaml`.

16.3 Background

Three processes make the AI market:

- **`pm-market-sim`** — the market model. Every symbol gets a *true value* that moves like a company's value does: with the market, with its sector, on its own, and now and then in a jump. Students never see it. It also publishes **news**: headlines that move the values.
- **`pm-ai-swarm`** — runs many AI traders, each logged on through `pm-alf-gwy` as its own participant (`AI001`, `AI002`, …) exactly like a student desk. `pm-ai-trader` runs a single one.
- **`pm-news`** — the instructor's news desk: publish a headline now, or play a timed scenario.

Each AI trader is a *preset*: what it trades on (its strategy), how it places orders, how fast, and its risk limits. `pm-ai-trader --list-presets` prints them all:

Preset	Trades on	Character
`noise-retail`	nothing in particular	Small random orders resting near the touch
`patient-liquidity`	nothing in particular	Small orders waiting deep in the book, giving it depth
`market-taker`, `iceberg-seller`	nothing in particular	Takes liquidity with MARKET orders; hides size behind icebergs
`scalper`, `trend-follower`, `block-taker`	momentum	Buys what has been rising, with IOC, marketable or fill-or-kill orders and protective stops
`contrarian`, `auction-player`	reversion	Fades moves; trades the opening and closing auctions
`value-investor`, `institutional`	the true value	Buys below its own (imperfect) estimate of the value, sells above it
`news-trader`	headlines	Trades a headline within seconds, hard, in its direction

The value and news traders are why prices end up near the values and react to news; the noise and momentum traders are why they never sit exactly on them. A headline has a public *sentiment* (−1 bad to 1 good) and a secret *impact* on the value. A **rumour** carries a *credibility* and moves no value until it is **confirmed**; a **retracted** rumour never moves anything.

16.4 Exercise 1: Deploy an Example with AI Traders

```
pm-config-deploy --example s150-nominal
```

This **replaces** the configuration you have built since chapter 01. From here on the participants are the ones the example defines — `TRADER01`–`TRADER10`, the instructor `OPS01` (ADMIN), `MM01`, and `AI001`–`AI020` — so `GW_ADMIN`, `TRADER03`, `MM_AAPL_01` and the other gateways of your own configuration no longer exist. Stop the consoles and bots that use them. To return to your own configuration later, deploy `engine_config.yaml` again.

The scenario and swarm files this chapter uses ship with EduMatcher, but not in your current directory: on a container or `pipx` install they live inside the installed package, and only in a repository checkout are they under `docs/examples`. Ask Python where they are, once per terminal, and use `$EX` in the commands below:

```
EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_root().parent)")
ls "$EX/news"          # the four scenarios: chip-race, earnings-surprise, rumour-confirmed, rumour-retracted
```

(In the container shell `python` is the right interpreter; in a Poetry checkout use `poetry run python`; with `pipx` use `~/.local/pipx/venvs/edumatcher/bin/python`.)

Restart `pm-engine` (and reopen the market in a moment), then start the gateway the AI traders log on through, the market model, and a market maker for all 150 symbols that quotes around the model's true values, each in its own terminal:

```
pm-alf-gwy -v
pm-market-sim -v
pm-mm-bot --gateway-id MM01 --all-symbols --strategy passive --gap 0.20 --qty 200 --anchor-sim 0.3
```

Without `--anchor-sim` the market maker keeps refilling its quotes around its own mid, and no amount of informed trading moves the price.

Open the market from the operator console:

```
pm-admin --id OPS01
[OPS01|ADMIN]> SESSION|STATE=PRE_OPEN
[OPS01|ADMIN]> SESSION|STATE=OPENING_AUCTION
[OPS01|ADMIN]> SESSION|STATE=CONTINUOUS
```

`pm-market-sim` starts moving the values when continuous trading starts. Check it:

```
pm-market-sim --status --id OPS01
```

:material-checkbox-blank-outline: **Checkpoint:** `pm-market-sim: RUNNING … symbols=150`.

16.5 Exercise 2: One AI Trader

```
pm-ai-trader --id AI001 --preset noise-retail --symbols AAPL -v
```

The trader logs on, reads the AAPL book and starts resting small orders near the touch. Watch them arrive with `BOOK|SYM=AAPL` in the operator console. Stop it with Ctrl+C: its session closes and the participant's `disconnect_behaviour` (`CANCEL_ALL`) removes its orders.

:material-checkbox-blank-outline: **Checkpoint:** AI001's orders appear in the AAPL book and disappear when you stop it.

16.6 Exercise 3: The Swarm

```
pm-ai-swarm --swarm "$EX/ref_data/s150-nominal-setup/swarm.yaml" -v
```

`swarm.yaml` gives the twenty traders `AI001`–`AI020` a mix of every preset (`per_agent: 150` — every agent trades every symbol, not a slice of them), and sets a budget of about 40 order actions a second for the whole swarm. Once a minute the swarm logs a status line:

```
[w1-AI001-AI020] t=120s session=CONTINUOUS actions=4801 (40.0/s) live_orders=164 fills=1290 rejects=0
```

:material-checkbox-blank-outline: **Checkpoint:** a status line every minute; books changing on symbols nobody in the room is trading.

16.7 Exercise 4: Price and Value

The instructor can see what the students cannot:

```
pm-market-sim --status --id OPS01 --top 5
```

```
symbol        value        mid  mid/value
TSLA         212.40     219.95     +3.55%
...
```

The value traders lean against these gaps; the noise and momentum traders keep opening new ones. Run the command again a few minutes later: the symbols at the top change, and large gaps rarely last.

:material-checkbox-blank-outline: **Checkpoint:** mid/value gaps of a few percent at most, changing from one look to the next.

16.8 Exercise 5: An Earnings Surprise

Every student opens a news display: the **News** screen of the trading GUI, the News view of TapeDeck, `pm-ticker`, or simply

```
pm-news list
```

Then the instructor plays the first scenario:

```
pm-news --id OPS01 play "$EX/news/earnings-surprise.yaml"
```

Sixty seconds into the play (it counts from its start when the market is already in continuous trading) AAPL reports far above expectations. The news is confirmed from the start: AAPL's value jumps about 8% at once. The news traders hit the book within seconds; the value investors follow.

Students try to buy AAPL as soon as they see the headline:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
```

Compare your fill with the AAPL price a minute later.

:material-checkbox-blank-outline: **Checkpoint:** AAPL trades several percent higher within a minute of the headline.

16.9 Exercise 6: A Rumour That Turns Out True

First stop the market maker (Ctrl+C on `pm-mm-bot`; its quotes are withdrawn). It quotes around the true values, and a rumour changes no value: it would sell to every rumour buyer at the old price, and the rumour would leave no trace. Without it, prices are what the traders believe — the AI traders' resting orders keep the books two-sided.

```
pm-news --id OPS01 play "$EX/news/rumour-confirmed.yaml"
```

At 60 seconds a rumour says JPM is a takeover target, credibility 50%. Nothing has happened to JPM's value — any move now is traders betting on the rumour. Discuss before the next step: buy now, or wait? At 240 seconds the bid is confirmed and the value jumps about 12%.

:material-checkbox-blank-outline: **Checkpoint:** JPM moves a little on the rumour and much more on the confirmation.

16.10 Exercise 7: A Rumour That Is Retracted

```
pm-news --id OPS01 play "$EX/news/rumour-retracted.yaml"
```

At 60 seconds a rumour says energy producers face a new windfall tax: a *sector* headline, so every ENERGY symbol (`XOM`, `CVX`, …) is concerned. The values do not move; prices dip only because traders believe it. At 240 seconds it is retracted — in the GUIs the rumour is now struck through — and prices drift back to where

the values are.

:material-checkbox-blank-outline: **Checkpoint:** ENERGY symbols dip after the rumour and are back near their earlier prices a few minutes after the retraction.

16.11 Exercise 8: A Breakthrough and a Fire

Not all news is earnings. *Events* (a fire at a factory, a research breakthrough, a big new contract, a product recall) go one way only, and they are big. Their kind says what happened: `OPERATIONS`, `RESEARCH`, `CONTRACT`, `ESG` or `PRODUCT`. Keep the market maker stopped, and have students watch NVDA and AMD side by side.

```
pm-news --id OPS01 play "$EX/news/chip-race.yaml"
```

At 60 seconds NVDA announces a `RESEARCH` breakthrough: its next chips are 50% faster. The news is confirmed, and NVDA's value jumps about 10%. It is a *symbol* headline, so AMD's value does not move at all. Discuss: should AMD's price fall because its rival got ahead?

At 240 seconds a rumour says AMD's main chip fab is on fire, credibility 70%. Discuss before the next step: sell AMD now, or wait for confirmation? A credibility of 0.7 is high, and a fire never turns out to be good news. At 360 seconds the fire is confirmed and AMD's value drops about 7%.

:material-checkbox-blank-outline: **Checkpoint:** NVDA jumps on the first headline while AMD barely moves; AMD dips on the rumour and falls further on the confirmation.

Restart the market maker afterwards (the command from Exercise 1).

16.12 Exercise 9: Stop the Swarm

Ctrl+C on `pm-ai-swarm` stops every worker; each closes its traders' sessions, which cancels their orders. Stop `pm-market-sim` with Ctrl+C too: it saves its values and continues from them next time.

:material-checkbox-blank-outline: **Checkpoint:** no `AI0xx` orders left in the books; the market maker's quotes remain.

16.13 Classroom Tips

- Your own headlines need no scenario: `pm-news --id OPS01 inject --symbol TSLA --kind PRODUCT --sentim -0.6 --impact -0.05`. A rumour you inject (`--rumour --credibility 0.4`) waits for your `pm-news --id OPS01 confirm N7` or `retract N7` — the id is printed when you inject it.
- `pm-market-sim` also generates news on its own, a handful a day. Set the rates in `market_sim.yaml` to `0` for a class where only your headlines should appear.
- Same `swarm.yaml` and `market_sim.yaml` seeds, same traders and same values: change `seed` in either file for a different market.
- A larger `--anchor-sim` (up to 1) makes the market maker follow the true value faster; then the AI traders, and the students, have less of a gap to trade.
- Running the AI-Trader Swarm covers sizing, logs and daily summaries for longer runs.

16.14 Reflection

Why does a confirmed headline move the price further than a rumour with the same sentiment? Who in the swarm was on the other side of the news traders' orders right after the earnings headline? The market maker quotes around the true value: what would have happened to the price, and to the students' chances of buying AAPL cheaply, if it had not been anchored? An earnings headline can go either way, but a fire can only be bad: how did that change what you did on the AMD rumour?

16.15 Further Reading

- AI Traders
- Running the AI-Trader Swarm
- The Market Model (`pm-market-sim`)
- The Market-Maker Bot (`pm-mm-bot`)
- AI Traders and Market Simulation — a simulated market that runs for weeks, with the AI traders and news on all the time

Next: 15 — Statistics & Reporting

Part IV

Running a full session

Chapters 15–20. Query the statistics, survive a restart, run a complete session end to end, compare the observer processes, use the advanced operator commands and settle the day.

CHAPTER 17

Statistics & Reporting

17.1 Objective

Use the statistics service to record market data and query OHLCV, VWAP, and trade history for analysis and reporting.

Background reading

- Statistics and Reporting

17.2 Prerequisites

- Chapters 01–14 completed.
- `pm-stats` running and at least a few trades executed.

17.3 Background

`pm-stats` subscribes to trade and book events and records them in a SQLite database at `$EDUMATCHER_DATA_DIR/sta` (this resolves relative to whatever data directory you configured in Chapter 00 — it is **not** a fixed `data/stats.db` path in the current working directory). The `pm-stats-cli` tool lets you query this data without writing SQL.

17.4 Exercise 1: Start the Statistics Service

```
pm-stats
```

`pm-stats` logs at `WARNING` by default, so it starts silently. To see its startup messages, run it with `-v`:

```
pm-stats -v
```

The startup lines include `opened stats DB connection path=...`, showing the database path (which reflects your `EDUMATCHER_DATA_DIR`), and `recording market statistics (Ctrl-C to stop)`; two more lines follow it (`recorded tick metadata …`, `requested opening snapshots …`).

To confirm the active path with a stable command instead, run:

```
pm-stats --help
```

and check the default shown for `--db`, or simply verify the file exists after the next exercise:

```
ls -la "$EDUMATCHER_DATA_DIR/stats.db"
```

:material-checkbox-blank-outline: **Checkpoint:** stats service running and `$EDUMATCHER_DATA_DIR/stats.db` exists.

17.5 Exercise 2: Generate Some Trading Activity

Ensure `pm-mm-bot` (or AI traders) is quoting **both AAPL and MSFT** (the third order below is a MSFT order; without a MSFT quote it is cancelled with `INSUFFICIENT_LIQUIDITY`). A manual market-maker quote is pulled after its first fill (Chapter 03), so the second and third orders would find no liquidity if you rely on one. One bot per symbol, each with a start range, as in chapter 02:

```
pm-mm-bot --symbol AAPL --gap 0.10 --qty 500 --initial_min 149 --initial_max 151
pm-mm-bot --symbol MSFT --gap 0.16 --qty 300 --initial_min 419 --initial_max 421
```

(Both need their market-maker gateway, `MM_AAPL_01` and `MM_MSFT_01`, in the configuration and no console connected as it. If you are still on the `s150-nominal` example from chapter 14, use its `MM01` instead, as in that chapter.) Execute a few manual trades:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=50
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=MARKET|QTY=200
```

:material-checkbox-blank-outline: **Checkpoint:** trades executed and recorded by stats service.

17.6 Exercise 3: Query the Daily OHLCV Summary

`pm-stats-cli` has no per-minute bar query — the closest built-in views are `daily` (one OHLCV row per symbol per trading day) and `snapshots` (periodic intraday price points, recorded at the interval set by `pm-stats --snapshot-interval`, default 15 minutes). For today's daily summary:

```
pm-stats-cli daily --symbol AAPL
```

Expected output (table or JSON):

```
date       | symbol | open_price | high_price | low_price | close_price | volume | trade_count | vwap
-----------|--------|------------|------------|-----------|-------------|--------|-------------|-------
2026-06-18 | AAPL   | 150        | 150.10     | 149.95    | 150.05      | 350    | 3           | 150.02
```

Note that `vwap` is already one of the columns in this row — see Exercise 4.

For an intraday price series instead of a single daily summary row, use `snapshots`:

```
pm-stats-cli snapshots --symbol AAPL
```

:material-checkbox-blank-outline: **Checkpoint:** `daily` returns an OHLCV row for AAPL with today's date.

17.7 Exercise 4: Read the VWAP Column

There is no standalone `vwap` command — the volume-weighted average price is a column on the `daily` row you already queried in Exercise 3:

```
pm-stats-cli daily --symbol AAPL
```

Read the `vwap` column from the output. It reflects the volume-weighted average price across all of today's trades for that symbol.

:material-checkbox-blank-outline: **Checkpoint:** you can locate the `vwap` value within `daily` output.

17.8 Exercise 5: Query Trade Log

```
pm-stats-cli trades --symbol AAPL --limit 20
```

Shows up to 20 trades from `trade_log`, **oldest first** (it is a page, not a "latest" list): `ts`, `trade_id`, `symbol`, `price`, `quantity`, `aggressor_side`, `buy_gateway_id`, `sell_gateway_id`. When more rows exist the command ends with a notice such as `[WARN] Output truncated at --limit 20. More rows exist; re-run with --after <cursor>`; copy that cursor to read the next page, or narrow the range with `--from <ISO timestamp>` to start at the recent trades. `trade_id` is a durable `run_seq-counter` string such as `000042-000000001`. `aggressor_side` is `BUY` or `SELL` for a continuous match, and `AUCTION` for an uncross print — so you can classify trades by which side crossed the spread.

:material-checkbox-blank-outline: **Checkpoint:** trade log visible.

17.9 Exercise 6: Multi-Symbol Summary

There is no dedicated `summary` verb. Omit `--symbol` on `daily` to get one row per symbol instead of filtering to one:

```
pm-stats-cli daily
```

Each row shows, per symbol: `open_price`, `high_price`, `low_price`, `close_price`, `volume`, `trade_count`, and `vwap` for the day. This does **not** include current spread (best bid/ask) — for that, use `snapshots` (most recent row per symbol) or query the live book directly with `BOOK|SYM=<symbol>` in the operator console (`pm-admin`).

:material-checkbox-blank-outline: **Checkpoint:** `daily` with no `--symbol` filter returns one row per active symbol.

17.10 Exercise 7: Export Data

```
pm-stats-cli --format csv trades --symbol AAPL > aapl_trades.csv
```

The CSV can be imported into Excel or a Jupyter notebook for further analysis.

:material-checkbox-blank-outline: **Checkpoint:** CSV export generated.

17.11 What Gets Recorded

`pm-stats` writes to four tables (see Statistics and Reporting for the full column reference):

Table	Stored Fields
`trade_log`	`ts`, `trade_id`, `symbol`, `price`, `quantity`, `buy_gateway_id`, `sell_gateway_id` — one row per trade, no aggregation
`price_snapshots`	`ts`, `symbol`, `mid_price`, `best_bid`, `best_ask`, `pct_change` — periodic intraday snapshots at `--snapshot-interval` (default 15 min)
`daily_stats`	`date`, `symbol`, `open/high/low/close_price`, `volume`, `trade_count`, `vwap`, plus open/close bid-ask and largest-trade fields — one row per symbol per day
`order_events`	Per-gateway private order lifecycle events (`ACK`, `FILL`, `AMEND`, `CANCEL`, etc.) — see `pm-stats-cli order-events`

There is no separate table tracking session-phase transitions (`PRE_OPEN`→`CONTINUOUS`, etc.) — `pm-stats` records market data and order lifecycle events only.

17.12 Summary

You can now:

- Configure and start the full exchange stack.
- Provide liquidity with manual MM quotes or `pm-mm-bot`.
- Trade using all order types and TIF values.
- Manage orders (amend, cancel, status).
- Run auctions and understand equilibrium pricing.
- Quote as a market maker with full lifecycle understanding.
- Use combo and OCO orders.
- Configure and trigger risk controls.
- Track P&L and positions.
- Subscribe to market data and drop-copy feeds.
- Generate realistic flow with AI traders.
- Record and query statistics.

17.13 Reflection

Why does `pm-stats` need its own subscriber process and SQLite database instead of the engine writing OHLCV/VWAP data directly? What would happen to engine performance or reliability if it had to compute and serve statistics queries itself, in-process, for every connected client?

17.14 Further Reading

- Statistics and Reporting
- Processes, Environment and Ports
- Persistence
- Market Data Feed

Next: 16 — Persistence & Recovery

For a fuller hands-on tour of every viewer and observer process (including `pm-stats` alongside `pm-viewer`, `pm-orders`, `pm-audit`, and `pm-board`), see 18 — Exchange Observer Processes.

CHAPTER 18

Persistence & Recovery

18.1 Objective

Learn what EduMatcher persists, where persistent files live, and how to verify state across restart scenarios.

Background reading

- Persistence - Audit Trail

18.2 Prerequisites

- Chapters 01–15 completed.
- Consistent `EDUMATCHER_DATA_DIR` configured for all processes.

18.3 Background

EduMatcher stores persistent runtime data under `EDUMATCHER_DATA_DIR`. Common files include:

- `gtc_orders.json` — resting `GTC` orders (any age) **and** resting `DAY` orders from the current business day, including MM quote legs. Written on every checkpoint and at clean shutdown; read back at the next startup.
- `stats.db` — SQLite database written by `pm-stats`.
- `audit.log` — event log if `pm-audit` writes to disk.

A restart is not a day boundary

Despite its name, `gtc_orders.json` also carries resting `TIF=DAY` orders across an **engine restart**, as long as the restart happens on the same business day the order was placed. The day-vs-restart distinction is the subject of Exercise 4 below — see Persistence — Impact of a Business-Day Change for the full rule.

18.4 Exercise 1: Locate the Data Directory

```
echo "$EDUMATCHER_DATA_DIR"
ls -la "$EDUMATCHER_DATA_DIR"
```

If the variable is empty, revisit 00 — Installation & Setup and run `pm-setup`.

:material-checkbox-blank-outline: **Checkpoint:** you can locate the persistent data directory.

18.5 Exercise 2: Create a GTC Order

Start the exchange and place a GTC order away from the market:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=140.00|TIF=GTC
```

Check the order is resting:

```
[TRADER01]> ORDERS
```

:material-checkbox-blank-outline: **Checkpoint:** the GTC order is resting and visible.

18.6 Exercise 3: Restart the Engine

GTC persistence in EduMatcher is unconditional — there is no config flag to enable/disable it — but it only works correctly under these conditions:

- **Prefer a clean stop** (`Ctrl+C` / SIGINT) over `kill -9`. The engine writes `gtc_orders.json` in its graceful shutdown handler. A hard kill skips that write, but the engine *also* checkpoints GTC state periodically while running, so a `kill -9` loses only the orders placed since the last checkpoint — not all of them.
- **Restart with the same `EDUMATCHER_DATA_DIR`** (and thus the same `gtc_orders.json` path) used in Exercise 2 — a different data directory has nothing to restore from.
- **Restart with a config that still lists the same symbol** — on restore, the engine skips any persisted GTC order whose symbol is no longer in `engine_config.yaml`.

With those three conditions met, stop `pm-engine` cleanly, then start it again with the same config and data dir:

```
pm-engine
```

Reconnect `TRADER01` and inspect orders:

```
[TRADER01]> ORDERS
```

The GTC order should be restored — with `pm-engine --verbose` the engine also logs a line at shutdown confirming how many resting orders it saved (`Saved N resting order(s) to ...`; the count covers both GTC and same-day DAY orders), which you can check as a stable confirmation instead of relying on `ORDERS` output alone.

Compare explicitly after restart:

- GTC orders restore from persistence (given the conditions above).
- Stats remain in `stats.db` if `pm-stats` was writing before restart.

:material-checkbox-blank-outline: **Checkpoint:** verify whether the GTC order survives restart.

18.7 Exercise 4: DAY Order Survives a Same-Day Restart

A resting `DAY` order is no longer tied to the engine process — it is tied to the business day. Place a DAY order at a non-marketable price:

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=399.00|TIF=DAY
```

Restart `pm-engine` cleanly, on the **same calendar day**, with the same data directory and config. Reconnect `TRADER01` and check:

```
[TRADER01]> ORDERS
```

The DAY order should still be resting — restored exactly like the GTC order from Exercise 3, because the restart did not cross a business-day boundary.

:material-checkbox-blank-outline: **Checkpoint:** the DAY order is still resting after a same-day restart.

Two separate ways a DAY order disappears

Don't confuse these — they are driven by different things:

- **Session close** (`pm-scheduler` transitioning to `CLOSED`, or you forcing it with `SESSION|STATE=CLOSED`): every resting DAY order is cancelled *live*, with an `order.expired` event, regardless of restart. See 06 — Time-in-Force & Sessions.
- **Engine restart after the business day has rolled over**: a resting DAY order from a prior business day is silently discarded during restore — logged at `INFO`, but with **no** `order.expired` published. There is no live sweep for this; it is only checked once, at the next startup. See Persistence — Impact of a Business-Day Change.

If you want to see the second case instead of the first, you would need to actually cross a calendar date between shutdown and restart — not practical to demonstrate live in this exercise, so take it as read from the Operator's Guide.

18.8 Exercise 5: Persist Statistics

Start `pm-stats`, execute a few trades, and inspect the stats database:

```
pm-stats
pm-stats-cli trades --symbol AAPL --limit 5     # the first five of the day, oldest first
ls -lh "$EDUMATCHER_DATA_DIR"
```

You should see `stats.db` in the data directory once statistics have been recorded.

:material-checkbox-blank-outline: **Checkpoint:** `stats.db` exists and contains recent trades.

18.9 Exercise 6: Audit to Disk

Start audit logging to a file:

```
pm-audit --audit-log-file "$EDUMATCHER_DATA_DIR/audit.log" --terminal
```

Execute a trade, then inspect the log:

```
tail -20 "$EDUMATCHER_DATA_DIR/audit.log"
```

`pm-audit` collects events in a buffer and writes them out in batches, at least every 10 seconds, so wait a few

seconds after the trade before you look (and start `pm-audit` before you trade: it records only what it sees).

:material-checkbox-blank-outline: **Checkpoint:** audit log contains events from your trading session.

18.10 Summary

You now understand:

- Which data belongs in `EDUMATCHER_DATA_DIR`.
- Why GTC and DAY orders behave differently across session boundaries, and why that is a *different* question from how they behave across an engine restart.
- How stats and audit files survive beyond the current terminal session.

18.11 Reflection

A resting `DAY` order and a live MM quote leg both now survive an engine restart on the same business day, restored from `gtc_orders.json` exactly like a `GTC` order would — only a business-day rollover discards them, and only at the next startup. Why does it make sense to key this purely off the *business day*, rather than off the number of times the engine process has restarted? What would go wrong operationally in a classroom setting if a `DAY` order's survival depended instead on "how many restarts have happened since it was placed"?

18.12 Further Reading

- Persistence
- Statistics and Reporting
- Processes, Environment and Ports
- A Full Trading Day

Next: 17 — Capstone Scenario

CHAPTER 19

Capstone Scenario

19.1 Objective

Run a complete exchange session that combines configuration, market making, trading, risk controls, clearing, market data, persistence, and reporting.

Background reading

- Running the Exchange - Processes, Environment and Ports

19.2 Prerequisites

- Chapters 01–16 completed.
- Ability to run multiple terminals/processes simultaneously.

19.3 Scenario

You are running a classroom exchange with three symbols (`AAPL`, `MSFT`, `TSLA`), two human traders, three AI traders, one admin/operator, one market maker per symbol and a spare manual market maker. Your goal is to open the market, provide liquidity, generate trades, trigger a risk event, inspect P&L/statistics, and verify persistence.

This chapter rebuilds the configuration from scratch rather than continuing from Chapter 16's. That is deliberate — an operator should be able to stand a venue up from nothing — but it means Exercise 1 re-declares every gateway the earlier chapters introduced, so nothing you learned to use disappears.

19.4 Exercise 1: Generate a Fresh Config

-force overwrites the configuration you have been building

This command replaces `engine_config.yaml` outright. Everything you added by hand in earlier chapters — `MM_MANUAL_01` from Chapter 09, the `AI001`– `AI003` gateways from Chapter 14, any collar or circuit-breaker tuning from Chapter 11 — is gone unless the command below re-creates it.
Back it up first so you can compare, or return to it afterwards:

```
cp engine_config.yaml engine_config.pre-capstone.yaml
```

The command below deliberately re-declares **every** gateway the earlier chapters introduced, so the capstone exercises a complete venue rather than a subset:

```
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --participants TRADER01:TRADER TRADER02:TRADER GW_ADMIN:ADMIN ADMIN_CLI:ADMIN \
             MM_AAPL_01:MARKET_MAKER MM_MSFT_01:MARKET_MAKER MM_TSLA_01:MARKET_MAKER \
             MM_MANUAL_01:MARKET_MAKER \
             AI001:TRADER AI002:TRADER AI003:TRADER \
  --sessions-enabled \
  --static-band 0.10 \
  --dynamic-band 0.05 \
  --no-mm-seed-quotes \
  --output engine_config.yaml --force
```

`--no-mm-seed-quotes` is not optional decoration. Normally every symbol must have a seeded market-maker quote once a `MARKET_MAKER` gateway exists (the **M001** rule from Chapter 02); this flag records `require_mm_seed_quo` `false` instead, so the venue starts with **empty books** and your market makers open them by hand in Exercise 5 — the way a venue is stood up from nothing. Seeding random prices with `--seed-mm-mid-range` would not work here: the hand-typed quotes below assume AAPL near 150, MSFT near 420 and TSLA near 250, and seeded prices elsewhere would make the collars and circuit breakers reject them.

Open the file and confirm the symbol and gateway sections are present.

The configuration is new, the data directory is not

The data directory still holds what earlier chapters left in it: resting GTC orders (such as chapter 16's bid at 140.00), same-day DAY orders and clearing history. They are restored when the engine starts, so a book may not be as empty as the text assumes. For a truly empty venue, stop every process and run `pm-opctl-cli clear --state --yes` before Exercise 3. It also deletes `clearing.db` and `stats.db`, which is what you want for a fresh venue. See Starting over for the complete reset matrix and preconditions.
:material-checkbox-blank-outline: **Checkpoint:** config contains 3 symbols, 5 traders (`TRADER01`, `TRADER02`, `AI001`–`AI003`), 2 admins, and 4 market makers, `require_mm_seed_quotes: false`, and no `market_maker_quotes`.

19.5 Exercise 2: Verify and Deploy

A generated file is not a running configuration. Verify it, then install it — this is the step that makes everything after it possible:

```
pm-cverifier engine_config.yaml       # expect: 0 ERRORS
pm-config-deploy engine_config.yaml
pm-config-deploy --show               # confirm where it landed
```

Warnings and advisories are fine here; errors are not. Expect the warning `M032` (no read-only API credential — nothing in this venue uses one) and `C001` (no reference prices): the books have no price until the market makers quote, so the static collar only takes effect from the first trades on. If `pm-cverifier` reports an error, fix it before deploying — a refused deploy leaves the *previous* configuration running, which is the confusing case where the exchange starts but has the wrong gateways.

:material-checkbox-blank-outline: **Checkpoint:** `pm-cverifier` reports 0 errors and `pm-config-deploy` completes.

19.6 Exercise 3: Start the Exchange Stack

Use separate terminals:

```
pm-engine --verbose
pm-stats
pm-clearing
pm-audit --terminal
pm-viewer --symbol AAPL
pm-alf-gwy -v
```

`pm-alf-gwy` is the TCP gateway that `pm-ai-trader` logs on through in Exercise 6 (Chapter 14); without it the AI traders report `cannot reach ALF gateway`.

`pm-scheduler` is left out on purpose: you will drive the trading day by hand from the operator console, as in Chapter 06, so the capstone works at any time of day.

:material-checkbox-blank-outline: **Checkpoint:** every process starts cleanly and connects.

19.7 Exercise 4: Connect Gateways

Open gateway terminals:

```
pm-alf-console --id TRADER01
pm-alf-console --id TRADER02
pm-admin        --id GW_ADMIN          # the operator console — a different program
pm-alf-console --id MM_AAPL_01
pm-alf-console --id MM_MSFT_01
pm-alf-console --id MM_TSLA_01
```

`MM_MANUAL_01` and `AI001`–`AI003` are configured but do not need a console of their own yet: `MM_MANUAL_01` is there if you want to repeat Chapter 09's manual quoting against this venue, and the AI gateways are driven by `pm-ai-trader` in Exercise 6 rather than typed at.

Confirm the engine agrees with your configuration:

```
[GW_ADMIN|ADMIN]> GATEWAYS
```

All ten should be listed, with the six above showing as connected.

:material-checkbox-blank-outline: **Checkpoint:** `GATEWAYS` lists all ten configured identities, and the six consoles you opened authenticate.

19.8 Exercise 5: Open the Market and Provide Liquidity

Open the trading day from the operator console. You could stop in the opening auction on the way; here, go straight to continuous trading:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
[GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
```

Then let each market maker quote its symbol:

```
[MM_AAPL_01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=AAPL-CAP-001
[MM_MSFT_01]> QUOTE|SYM=MSFT|BID=419.90|ASK=420.10|BID_QTY=300|ASK_QTY=300|TIF=DAY|QUOTE_ID=MSFT-CAP-001
[MM_TSLA_01]> QUOTE|SYM=TSLA|BID=249.75|ASK=250.25|BID_QTY=200|ASK_QTY=200|TIF=DAY|QUOTE_ID=TSLA-CAP-001
```

Verify with `QLEGS|SHOW=ALL` on each MM gateway (trader console) and `BOOK|SYM=...` in the operator console.

:material-checkbox-blank-outline: **Checkpoint:** every symbol has a live two-sided market.

19.9 Exercise 6: Generate Trades and Amendments

From `TRADER01`:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=419.50|TIF=DAY
[TRADER01]> ORDERS
[TRADER01]> AMEND|ID=<msft_order_id>|PRICE=419.95|RTAG=CAP-AMD-001
```

The amendment improves TRADER01's bid to 419.95 — inside the market maker's 419.90 / 420.10 spread — so it is now the **best bid**. From `TRADER02`, sell at that price; it trades with TRADER01, not with the market maker, whose bid is lower:

```
[TRADER02]> NEW|SYM=MSFT|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=419.95|TIF=DAY
```

Now add background flow, so the later P&L and statistics exercises have more than a handful of hand-typed trades to work with. Start the three AI gateways you configured in Exercise 1, each with a different preset, in their own terminals:

```
pm-ai-trader --id AI001 --preset scalper  --duration 300
pm-ai-trader --id AI002 --preset contrarian    --duration 300
pm-ai-trader --id AI003 --preset noise-retail  --duration 300
```

`--duration 300` stops them after five minutes, which is long enough for the rest of the capstone and short enough that you are not chasing runaway processes later. They print nothing unless you add `-v`; check that they are connected with `GATEWAYS` (`AI001`–`AI003` now show `YES`) and watch `pm-viewer` — the book should get noticeably busier.

:material-checkbox-blank-outline: **Checkpoint:** you have at least one market fill, one amended order, one cross-trader fill, and a visible stream of AI trades in `pm-viewer` and `pm-audit`.

19.10 Exercise 7: Trigger an Operator Action

From the admin gateway:

```
[GW_ADMIN|ADMIN]> HALT_SYM|SYM=TSLA
```

Try to trade TSLA from a trader gateway and confirm it is rejected. Then resume:

```
[GW_ADMIN|ADMIN]> RESUME_SYM|SYM=TSLA
```

:material-checkbox-blank-outline: **Checkpoint:** symbol halt blocks trading and resume restores it.

19.11 Exercise 8: Inspect P&L, Audit, and Stats

Check the observer terminals and run:

```
pm-stats-cli trades --symbol AAPL --limit 10
pm-stats-cli daily
```

(`trades` lists the day's trades oldest first, ten at a time; with the AI traders running there are more, and the output ends with a notice giving the `--after` cursor for the next page.)

(`daily` with no `--symbol` filter gives one OHLCV summary row per symbol — see 15 — Statistics & Reporting for the full `pm-stats-cli` verb reference; there is no separate `summary` verb.)

Explain what each observer showed:

- `pm-clearing`: positions and P&L.
- `pm-audit`: raw event stream.
- `pm-stats-cli`: persisted trade/statistics view.
- `pm-viewer`: current book state.

:material-checkbox-blank-outline: **Checkpoint:** you can trace one trade through all observers.

19.12 Exercise 9: Persistence Check

Place a GTC order, restart the engine, and confirm whether it restores:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=140.00|TIF=GTC
```

Restart `pm-engine` (it comes back in `CLOSED`; reopen it with `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS` if you want to keep trading), reconnect `TRADER01`, then run:

```
[TRADER01]> ORDERS
```

:material-checkbox-blank-outline: **Checkpoint:** you can explain what persisted and what expired.

19.13 Final Review Questions

Answer these without looking at earlier chapters:

1. Which process owns the order book?
2. Which commands create, amend, cancel, and inspect resting orders?
3. Why does a quote have both a `quote_id` and two child order IDs?
4. What is the difference between public market data and drop-copy?
5. Which order types never rest on the book?
6. What happens to DAY vs GTC orders at session close?
7. How do you halt and resume a single symbol?

:material-checkbox-blank-outline: **Checkpoint:** you can answer every question from memory or by using the other books.

Review map:

- Q1: Architecture Overview
- Q2: ALF Console (pm-alf-console)
- Q3: MM Quotes Concept
- Q4: Drop Copy
- Q5: Order Types
- Q6: Time-in-Force & Session Lifecycle
- Q7: Controlling the Exchange

19.14 Further Reading

- How an Exchange Works
- Operator's Guide — Running the Exchange
- Architecture Overview
- Glossary
- Exchange Observer Processes
- Order Book Deep Dive

You have completed the capstone. Finish with 18 — Exchange Observer Processes to compare the different live views of the exchange.

CHAPTER 20

Exchange Observer Processes

20.1 Objective

Run the viewer and observer processes side by side so you can see the same exchange activity from different operational viewpoints: book depth, cross-gateway orders, ticker tape, market board, audit trail, statistics, and clearing/P&L.

Background reading

- Processes, Environment and Ports

20.2 Prerequisites

- Chapters 01–03 completed, with `pm-engine` running, the market open in `CONTINUOUS`, and at least `TRADER01` connected.
- Market-maker liquidity from chapter 02, or another gateway placing opposite-side orders so trades and book updates are visible.
- Several terminals available. These observer processes are intentionally separate so you can start, stop, and compare them independently.

20.3 Background

EduMatcher observers subscribe to the engine's market-data stream. They do not own the order book and they do not submit orders. Their job is to make activity visible from different angles.

Process	Viewpoint	Best for
`pm-viewer`	One symbol's order book	Depth, spread, recent trades
`pm-board`	Multi-symbol dashboard	Large-screen market overview
`pm-ticker`	Scrolling market tape	Compact live price/trade context

Process	Viewpoint	Best for
`pm-orders`	Cross-gateway order monitor	Resting order state across gateways
`pm-audit`	Raw event log	Debugging and replay-style inspection
`pm-stats`	Statistics recorder	Persisting OHLCV, VWAP, trade log
`pm-stats-cli`	Statistics query CLI	Reading persisted stats
`pm-clearing`	Positions and P&L	Settlement and participant exposure

20.4 Exercise 1: Start the Baseline Exchange

Start the engine if it is not already running, and open the market from the operator console if it is closed (chapter 01, "Open the market for the chapters ahead"):

```
pm-engine
```

Connect two trader gateways:

```
pm-alf-console --id TRADER01
pm-alf-console --id TRADER02
```

Create one resting order so observers have a visible book update:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.80|TIF=DAY
```

:material-checkbox-blank-outline: **Checkpoint:** `TRADER01` receives an acknowledgement and `ORDERS` shows the order.

20.5 Exercise 2: Watch One Book with pm-viewer

In a new terminal, start the single-symbol book viewer:

```
pm-viewer --symbol AAPL --depth 10
```

From `TRADER01`, add or cancel another resting order:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=50|PRICE=150.20|TIF=DAY
```

`pm-viewer` should update the bid/ask depth and recent trade area when market data changes.

:material-checkbox-blank-outline: **Checkpoint:** you can identify best bid, best ask, spread, and recent trades in `pm-viewer`.

20.6 Exercise 3: Monitor Orders Across Gateways with pm-orders

Start the cross-gateway order monitor:

```
pm-orders
```

For a narrower view, filter to one gateway:

```
pm-orders --gateway TRADER01
```

Place one order from each trader:

```
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=25|PRICE=419.00|TIF=DAY
[TRADER02]> NEW|SYM=TSLA|SIDE=SELL|TYPE=LIMIT|QTY=10|PRICE=252.00|TIF=DAY
```

Compare this process with the gateway-local `ORDERS` command. `ORDERS` is useful inside one gateway; `pm-orders` is useful when you want an exchange-wide monitor.

:material-checkbox-blank-outline: **Checkpoint:** `pm-orders` shows orders from both gateways, while `ORDERS` shows only the current gateway's cache.

20.7 Exercise 4: Record the Event Stream with pm-audit

Start the audit logger in terminal mode:

```
pm-audit --terminal
```

To also write the log to a known file:

```
pm-audit --terminal --audit-log-file "$EDUMATCHER_DATA_DIR/audit.log"
```

Execute a trade:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=25
```

Watch the raw topics and payloads printed by `pm-audit`. This is the most direct view of what the engine publishes.

:material-checkbox-blank-outline: **Checkpoint:** you can find the order acknowledgement, fill, trade, and book update events in the audit output.

20.8 Exercise 5: Build Persistent Statistics with pm-stats

Start the statistics recorder:

```
pm-stats
```

Execute several trades, then query the database:

```
pm-stats-cli trades --symbol AAPL --limit 10
pm-stats-cli daily --symbol AAPL
```

(`daily` gives the day's OHLCV summary, including a `vwap` column — see 15 — Statistics & Reporting for the full `pm-stats-cli` verb reference; there is no separate `ohlcv`, `vwap`, or `summary` verb.)

`pm-stats` is a long-running subscriber. `pm-stats-cli` is a read-only query tool for the SQLite database written by `pm-stats`.

:material-checkbox-blank-outline: **Checkpoint:** `pm-stats-cli` shows recent trades and at least one aggregate statistic after activity occurs.

20.9 Exercise 6: Launch the Scrolling Ticker

Start the ticker. The book updates it receives are pushed live by the engine; `--db-interval` controls only how often it re-reads the daily aggregates from `stats.db`, so a short value makes those refresh quickly during a lesson:

```
pm-ticker --db-interval 30
```

The ticker combines live book updates with statistics from `stats.db` when they are available. If no symbols appear yet, create a book update or trade in a gateway.

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=20|PRICE=149.70|TIF=DAY
```

:material-checkbox-blank-outline: **Checkpoint:** `pm-ticker` prints periodic lines with symbols, last price or best bid/ask, and statistics when available.

20.10 Exercise 7: Launch the Multi-Symbol Board

Start the market board:

```
pm-board --rows 8 --interval 10
```

The board shows multiple symbols at once and updates as `book.*` and `trade.executed` events arrive. Press Enter to advance pages manually if you have more symbols than rows.

:material-checkbox-blank-outline: **Checkpoint:** `pm-board` shows AAPL, MSFT, and TSLA in a single market overview.

20.11 Exercise 8: Track Positions and P&L with pm-clearing

Start clearing in a separate terminal:

```
pm-clearing
```

Create a trade between two gateways:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=150.00|TIF=DAY
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=50
```

`pm-clearing` consumes `trade.executed` events and maintains per-gateway position and P&L, writing batched results to `clearing.db` (SQLite) in the data directory. Use `pm-clearing-cli --format csv ...` if you want a CSV export.

:material-checkbox-blank-outline: **Checkpoint:** `pm-clearing` shows the changed position or records the trade in its report.

20.12 Exercise 9: Compare the Views

For one trade, identify where each process shows it:

Table 20.2: material-checkbox-blank-outline: **Checkpoint:** explain why no single observer replaces the others.

Question	Process to use
What is the current top of book?	`pm-viewer` or `pm-board`
Which gateways have live/resting orders?	`pm-orders`
What exact messages were published?	`pm-audit`
What trades and aggregates were persisted?	`pm-stats-cli`
What changed in participant positions/P&L?	`pm-clearing`
What is the compact live tape view?	`pm-ticker`

20.13 Summary

You have now run the main exchange observer processes and seen how each one answers a different operational question. In normal use, start only the observers you need for the lesson or scenario you are running.

20.14 Reflection

Why does EduMatcher split viewing responsibilities across several small observer processes (`pm-viewer`, `pm-orders`, `pm-audit`, `pm-stats`, `pm-board`) instead of one all-in-one dashboard? What would you lose in a classroom setting if all of them were combined into a single process that had to be restarted together?

20.15 Further Reading

- Processes, Environment and Ports
- Messages
- Market Data & Drop Copy
- Statistics & Reporting
- Market Data Feed

Next: 19 - Advanced Admin Operations

CHAPTER 21

Advanced Admin Operations

21.1 Objective

Practice advanced exchange-operator controls that are documented but were not covered in earlier training chapters: `KICK`, `QCANCEL`, `CANCEL_SYM`, and manual session overrides via `SESSION|STATE=...`.

Background reading

- The Admin Console and Exchange Commands

21.2 Prerequisites

- Chapters 01-18 completed.
- `GW_ADMIN` configured with `role: ADMIN`.
- Engine and at least one trader and one MM gateway connected.

21.3 Background

Earlier risk-control exercises focused on halt/resume and kill switch. In live operations, admins also need targeted controls for:

- Disconnecting a specific participant (`KICK`).
- Cancelling one MM quote on one symbol without killing the whole gateway (`QCANCEL`).
- Clearing one symbol's resting book across all gateways (`CANCEL_SYM`).
- Forcing session state for controlled drills (`SESSION|STATE=...`).

These controls exist in both:

- Interactive admin console (`pm-admin` / ALF commands).
- One-shot scripting CLI (`pm-admin-cli`).

Use a separate identity for each operator client

The interactive console uses `GW_ADMIN`; one-shot commands use `ADMIN_CLI`. The chapter 01 configuration defines both as ADMIN so the console and CLI can stay connected at once. A duplicate live connection with the same participant ID is refused.

21.4 Exercise 1: Disconnect a Gateway with KICK

From admin console:

```
[GW_ADMIN|ADMIN]> KICK|GW=TRADER02|REASON=Training_disconnect_check
```

Or with CLI:

```
pm-admin-cli --id ADMIN_CLI kick --gw TRADER02 --reason "Training disconnect check"
```

Verification:

1. The `TRADER02` console is **not told** it was kicked; its prompt looks unchanged. Confirm the disconnect from the operator side with `GATEWAYS` (`TRADER02` now shows `no` under *Connected*), and from the trader side by sending any order: it is refused with `code=AUTH_REQUIRED  Gateway not connected: TRADER02`.
2. Leave that console (`EXIT`) and reconnect with a fresh one:

```
pm-alf-console --id TRADER02
```

:material-checkbox-blank-outline: **Checkpoint:** gateway is disconnected and can reconnect cleanly.

21.5 Exercise 2: Cancel One MM Quote with QCANCEL

First ensure an MM has an active quote on AAPL.

From admin console:

```
[GW_ADMIN|ADMIN]> QCANCEL|GW=MM_AAPL_01|SYM=AAPL
```

Or with CLI:

```
pm-admin-cli --id ADMIN_CLI qcancel --gw MM_AAPL_01 --sym AAPL
```

Verify from MM gateway:

```
[MM_AAPL_01]> QLEGS|SYM=AAPL|SHOW=ALL
```

You should see the prior quote legs no longer active.

When to use this instead of manual MM-side `QUOTE_CANCEL`: use `QCANCEL` when the MM process is unresponsive or when operations needs immediate symbol-scoped intervention without relying on the MM terminal.

:material-checkbox-blank-outline: **Checkpoint:** one-symbol MM quote is cancelled without mass-cancelling all gateway orders.

21.6 Exercise 3: Clear One Symbol Book with CANCEL_SYM

Create a few resting orders on TSLA from multiple gateways, then run:

From admin console:

```
[GW_ADMIN|ADMIN]> CANCEL_SYM|SYM=TSLA
```

Or with CLI:

```
pm-admin-cli --id ADMIN_CLI cancel-sym --sym TSLA
```

Verify:

```
pm-admin-cli --id ADMIN_CLI book --sym TSLA
```

Confirm TSLA resting orders were cleared while other symbols remain unchanged.

:material-checkbox-blank-outline: **Checkpoint:** symbol-scoped cancel works across gateways.

21.7 Exercise 4: Force Session State for Controlled Drills

Set session state explicitly. Transitions are validated by the engine: from `CLOSED` the only legal move is `PRE_OPEN`, and `PRE_OPEN` must precede `CONTINUOUS`. The sequence below starts from `CLOSED`; if the market is open (`CONTINUOUS`), the first two requests are ignored and the console or CLI only prints a timeout, so start from the step that matches `SESSION_STATUS` (an illegal jump is silently ignored).

From admin console:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
[GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSING_AUCTION
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

Or with CLI:

```
pm-admin-cli --id ADMIN_CLI session --state PRE_OPEN
pm-admin-cli --id ADMIN_CLI session --state CONTINUOUS
pm-admin-cli --id ADMIN_CLI session --state CLOSING_AUCTION
pm-admin-cli --id ADMIN_CLI session --state CLOSED
pm-admin-cli --id ADMIN_CLI session-status
```

Use this to reproduce auction and close transitions deterministically in training. For example, after reaching `CONTINUOUS`, requesting `CONTINUOUS` again is an invalid transition: the engine does not publish a new `session.state`, so the CLI waits until its timeout. The REST transition endpoint instead returns `409 TRANSITION_REJECTED` with the engine's reason.

:material-checkbox-blank-outline: **Checkpoint:** forced session transitions are visible and verifiable.

21.8 Exercise 5: Run an Incident Micro-Runbook

Simulate a short incident response sequence:

1. Detect suspicious behavior on one gateway.
2. `KICK` the gateway.
3. `QCANCEL` one affected MM quote.
4. `CANCEL_SYM` for one stressed symbol.
5. `SESSION|STATE=CONTINUOUS` to restore normal operation if needed.

Example CLI sequence:

```
pm-admin-cli --id ADMIN_CLI kick --gw TRADER02 --reason "Training incident"
pm-admin-cli --id ADMIN_CLI qcancel --gw MM_AAPL_01 --sym AAPL
pm-admin-cli --id ADMIN_CLI cancel-sym --sym AAPL
pm-admin-cli --id ADMIN_CLI session-status
```

`TRADER02` is now disconnected: leave its console and start a new one before the next chapter, which needs it.

:material-checkbox-blank-outline: **Checkpoint:** you can explain when to use each control and why they are not interchangeable.

21.9 Summary

You can now operate advanced admin controls beyond halt/resume:

- `KICK` for participant disconnection.
- `QCANCEL` for targeted MM quote cancellation.
- `CANCEL_SYM` for symbol-wide book clearing.
- `SESSION|STATE=...` for deterministic phase management.

21.10 Reflection

Why does `CANCEL_SYM` clear an entire symbol's book while `QCANCEL` targets only quote-based orders? In an incident where a single market maker's bot is malfunctioning, which of these tools (or `KICK`, or the Chapter 11 kill switch) would you reach for first, and why is that the *least* disruptive choice available?

21.11 Further Reading

- Controlling the Exchange
- Processes, Environment and Ports
- Risk Controls

Next: 20 — Closing the Day

CHAPTER 22

Closing the Day: Settlement (`pm-eod`)

22.1 Objective

Close a trading day the way an exchange operator does, using `pm-eod`:

- settle every trader's result for the day
- produce the statements and the payment instructions
- check the books add up to zero
- recover from the mistakes that happen in practice: a missed day, a broken roster file, and trading after the close

Background reading

- Closing the Day: Settlement, Statements and Housekeeping — the concepts (equity, settlement, the central pool, zero-sum) and every message `pm-eod` can print - Your End-of-Day Statement — the statement as a trader reads it

22.2 Prerequisites

- Chapters 01–19 completed. Chapter 12 (`pm-clearing`) matters most here: settlement reads what `pm-clearing` recorded.
- The engine running with `TRADER01`, `TRADER02`, `GW_ADMIN` and the market makers from chapter 02 connected, and `pm-clearing` running. If chapter 19 left `TRADER02` kicked, start a new console for it, and re-send a `QUOTE` from `MM_AAPL_01` if its quote was cancelled: Exercise 5 needs liquidity on AAPL.
- **No `pm-scheduler` running.** You move the session phases yourself, as in earlier chapters. A running `--daily` scheduler would close the day at its own time.
- `pm-eod` on your `PATH`. Try `pm-eod --version`; if it is missing in a source checkout, run `poetry install` once.

22.3 Background

At the end of each trading day, `pm-eod run` does five things:

1. **Checks** that the day has really closed and that the clearing books agree.
2. **Works out** each trader's result for the day: the change in their equity (realized + unrealized P&L) since the close of the previous settled day.
3. **Adds** that amount to the trader's running balance in `eod.db`.
4. **Writes** one statement per trader, a summary and a payments file under `$EDUMATCHER_DATA_DIR/settlement/<`
5. **Tidies up:** compresses old logs, takes backups, and archives any audit days `pm-audit` missed.

All amounts together always add up to exactly zero, because every trade has a buyer and a seller.

On a production exchange, `pm-scheduler --on-close "pm-eod run --pending"` runs this by itself 30 seconds after every close. In this chapter you run every step by hand, so that you see each one.

The trading date

`pm-eod` labels each day with its **trading date**: the calendar date in the exchange's timezone (`locale.session_timezone`, default `Europe/Stockholm`). Set it once in the shell you use for this chapter:

```
D=$(TZ=Europe/Stockholm date +%F)   # use your session timezone if you changed it
echo "$D"
```

22.4 Exercise 1: Check the starting point

Confirm that `pm-clearing` is healthy and the market is open:

```
pm-clearing-cli health
```

```
[GW_ADMIN|ADMIN]> SESSION_STATUS
```

If the session is not `CONTINUOUS`, open it as in chapter 01 (`SESSION|STATE=PRE_OPEN`, then `SESSION|STATE=CONTINUOUS`)

Why the market must be open before Exercise 2

If you left the market `CLOSED` at the end of an earlier chapter *today*, today counts as a closed trading day. The next step would then settle today before you have traded in this chapter. Open the market first. :material-checkbox-blank-outline: **Checkpoint:** `pm-clearing-cli health` reports `health: OK` and the session is `CONTINUOUS`.

22.5 Exercise 2: Clear the backlog

The earlier chapters closed trading days while `pm-clearing` was running, so those days are probably still waiting to be settled. Look:

```
pm-eod status
```

The table of runs is empty the first time. If any days are waiting, a line on the error stream lists them:

```
[output]
No rows found.
pending: 2026-11-04
```

Settle them, oldest first:

```
pm-eod run --pending
```

```
[output]
2026-11-04:
  [ok] gate: closed, snapshot present, reconciled
  [ok] settle: settled
  [ok] report: wrote .../settlement/2026-11-04 (no roster: statements are keyed by gateway id)
  [ok] audit: nothing pending
  [ok] logs: no log directory
  [ok] backup: backed up clearing.db, eod.db (…); not present: stats.db
```

Each line is one step: `gate`, `settle` and `report` are the money, and `audit`, `logs` and `backup` are the housekeeping. Your dates, sizes and `not present:` list will differ. `nothing pending` instead of the block above means there was no backlog.

If the backlog run stops on an old day

`--pending` stops at the first day that fails and says `stopping: later dates wait for <date>`, because days are settled in order. Look the `[FAIL]` message up in the operator chapter's Troubleshooting table. On a training machine, an old day whose clearing data is damaged can be cleared away with `pm-opctl-cli clear --state` (with everything stopped). That deletes all working state (clearing history, GTC orders, statistics) but keeps `eod.db`, so the balances settled so far survive. See Starting over for the full reset matrix and the `--all` data-loss warning.
:material-checkbox-blank-outline: **Checkpoint:** `pm-eod run --pending` ended without a `[FAIL]` line, and a second `pm-eod run --pending` prints `nothing pending`.

22.6 Exercise 3: Ask too early

Try to settle today while the market is still open:

```
pm-eod run --date "$D" --dry-run
```

```
[output]
2026-11-05:
  [FAIL] gate: 2026-11-05 has not closed yet
```

The **gate** is the first step, and it protects the money. A day is closed only when its latest session phase is `CLOSED` *and* the market was open earlier that day. `--dry-run` runs the gate and the calculation and writes nothing, so it is always safe to try.

:material-checkbox-blank-outline: **Checkpoint:** the gate refuses today, and `echo $?` right after the command prints `1`.

22.7 Exercise 4: Write the roster

`pm-eod` knows traders only by gateway id. The **roster** connects ids to people. Create `$EDUMATCHER_DATA_DIR/ref_da

```
traders:
  - gateway: TRADER01
    name: Alice
    email: alice@example.org
    opening_balance: 1000.00
  - gateway: TRADER02
    name: Bob
    email: bob@example.org
    opening_balance: 1000.00
```

Leave the market makers out on purpose: you will see what `pm-eod` does with a gateway that is not in the roster.

:material-checkbox-blank-outline: **Checkpoint:** the file exists, and you can say what `opening_balance` does: it is the starting point of the trader's running balance, added whenever a balance is shown.

22.8 Exercise 5: Trade a day

Give both traders something to settle. TRADER01 buys and keeps a position; TRADER02 buys and sells part of it again:

```
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=200
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=100
[TRADER02]> NEW|SYM=AAPL|SIDE=SELL|TYPE=MARKET|QTY=50
```

Look at their positions and P&L before the close:

```
pm-clearing-cli positions --symbol AAPL
pm-clearing-cli pnl --symbol AAPL
```

Predict, before going on: whose day result will be positive and whose negative? (Market orders pay the spread to the market makers, so a trader who bought and sold at once is usually a little down.)

:material-checkbox-blank-outline: **Checkpoint:** both traders show filled trades in `pm-clearing-cli trades --date "$D"`, and you have written down a prediction.

22.9 Exercise 6: Close the day and settle it

1. Close the session from the operator console:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

2. See the close snapshot. At the close `pm-clearing` saves every position at the closing price. This is what is settled:

```
pm-clearing-cli snapshots --date "$D" --gateway TRADER01
```

```
[output]
trade_date | gateway_id | symbol | net_qty | mark_price | tick_decimals | realized_pnl | unrealized_pnl
-----------+------------+--------+---------+------------+---------------+--------------+---------------
2026-11-05 | TRADER01   | AAPL   | 200     | 149        | 2             | -106.667     | -213.333
```

3. Dry run. Now the gate passes, and you see the amounts before anything is written:

```
pm-eod run --date "$D" --dry-run
```

```
[output]
2026-11-05:
  [ok] gate: closed, snapshot present, reconciled
  [ok] settle: would settle: MM_AAPL_01 +275.00, TRADER01 -320.00, TRADER02 +45.00
```

4. Check one number by hand. A trader's amount is the sum over all their symbols of `realized_pnl + unrealized_pnl` in today's snapshot, minus the same sum at the close of the previous *settled* day (`pm-eod status` lists the settled days; with none, it is 0). In the example above, TRADER01 has $-106.667 + (-213.333) = -320.00$ today. Repeat this for TRADER01 with your own numbers, using `pm-clearing-cli snapshots --gateway TRADER01 --date <that day>` for the earlier day.

5. Settle:

```
pm-eod run --pending
```

A plain `--pending` is enough: today is now the only closed day that is waiting.

6. Check the books:

```
pm-eod balances
pm-eod balances --check
```

`--check` should print `OK - every settled date sums to zero`. Add up today's amounts from the dry run yourself: they also come to 0.00.

:material-checkbox-blank-outline: **Checkpoint:** your hand calculation matches TRADER01's amount, the run had no `[FAIL]` line, and `--check` says OK. Was your prediction from Exercise 5 right?

22.10 Exercise 7: Read what the traders receive

```
ls "$EDUMATCHER_DATA_DIR/settlement/$D" "$EDUMATCHER_DATA_DIR/settlement/$D/statements"
cat "$EDUMATCHER_DATA_DIR/settlement/$D/statements/TRADER01.md"
cat "$EDUMATCHER_DATA_DIR/settlement/$D/instructions.csv"
cat "$EDUMATCHER_DATA_DIR/settlement/$D/leaderboard.csv"
cat "$EDUMATCHER_DATA_DIR/settlement/$D/summary.md"
```

Answer from the files:

1. What is TRADER01's headline sentence, and does its amount match the dry run?
2. Does TRADER01's *Open positions at the close* agree with `POS` in TRADER01's console?
3. In `instructions.csv`, which traders `pay` and which `receive`? Whose balance includes the 1000.00 opening balance?
4. How does the market maker's statement differ from TRADER01's, and which warning names it? (Look at the end of `summary.md`.)

:material-checkbox-blank-outline: **Checkpoint:** you can answer all four from the files alone.

22.11 Exercise 8: Run it again

Settlement must never count a day twice. Run today again:

```
pm-eod run --date "$D"
pm-eod status
```

The `settle` line now says `already settled`. The report and housekeeping run again: the statements are

written afresh, and the amounts are the stored ones, not new ones. `pm-eod status` shows today with status `ok`.

:material-checkbox-blank-outline: **Checkpoint:** `pm-eod balances` shows the same numbers as after Exercise 6.

22.12 Exercise 9: A broken roster

Introduce a typical mistake: change TRADER01's opening balance to `1000.005` (three decimals) and run today again:

```
pm-eod run --date "$D"
```

```
[output]
2026-11-05:
  [ok] gate: closed, snapshot present, reconciled
  [ok] settle: already settled
  [FAIL] report: .../ref_data/roster.yaml: line 2: opening_balance has more than 2 decimals
  ...
```

The error names the line. The settlement is safe in the ledger; only the statements could not be written. Now look at the day as a monitoring system would:

```
pm-eod status
pm-eod status -q; echo "exit code $?"
```

Today's status is `report-failed`, and `status -q` (the health check) returns **1**. Fix the file back to `1000.00` and run the day again, **without** `--only`:

```
pm-eod run --date "$D"
pm-eod status -q; echo "exit code $?"
```

Why not –only report?

`pm-eod run --date "$D" --only report` would rewrite the statements too, but a run with `--only` is never recorded, so the day would stay `report-failed` and the health check red. A full run records `ok`.
:material-checkbox-blank-outline: **Checkpoint:** the health check went from exit code 1 back to 0, and today's status is `ok` again.

22.13 Exercise 10: Trading after the close

Sometimes trading resumes on a date that is already settled, for example when a class runs over. Reopen the market and trade. Buy rather than sell here: Exercise 5 ended with a sell at the market makers' bid, so selling again at that same price would not move the closing price and the new amounts would equal the old ones. A buy trades at the offer, which moves it:

```
[GW_ADMIN|ADMIN]> SESSION|STATE=PRE_OPEN
[GW_ADMIN|ADMIN]> SESSION|STATE=CONTINUOUS
[MM_AAPL_01]> QUOTE|SYM=AAPL|BID=149.95|ASK=150.05|BID_QTY=500|ASK_QTY=500|TIF=DAY|QUOTE_ID=AAPL-LATE-001
[TRADER02]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=50
[GW_ADMIN|ADMIN]> SESSION|STATE=CLOSED
```

Now ask `pm-eod` to catch up:

```
pm-eod run --pending
```

It says `nothing pending`. There is **one settlement per trading date**, and today's is already done. The afternoon's trading is in the new close snapshot, but not in the ledger. See the difference:

```
pm-eod run --date "$D" --dry-run
```

The dry run computes today again from the new snapshot and shows different amounts. To replace today's settlement with them, use `--redo`:

```
pm-eod run --date "$D" --redo
pm-eod balances
pm-eod balances --check
```

```
[output]
2026-11-05:
  [ok] gate: closed, snapshot present, reconciled
  [ok] settle: redone, dropped 2026-11-05
  ...
```

-redo drops every later day too

Balances are a running total, so `--redo` deletes the chosen date's settlement **and every later date's** and recomputes the chosen one. The later dates are settled again by the next `pm-eod run --pending`. On a real exchange, always do a `--dry-run` first.
:material-checkbox-blank-outline: **Checkpoint:** after `--redo` the balances include the afternoon's trade, `--check` still says OK, and you can explain why `--pending` did not pick the afternoon up by itself.

22.14 Summary

You can now:

- Settle a trading day with `pm-eod`, and know what the gate checks before any money is written.
- Check a trader's amount by hand from the close snapshots, and check that the day sums to zero.
- Read the statement, `instructions.csv`, the leaderboard and the summary.
- Clear a backlog with `--pending`, recover from a broken roster without leaving the day `report-failed`, and recompute a day with `--redo`.

22.15 Reflection

1. Settlement pays out *paper* profits every day, not only profits locked in by selling. Why is that safer for an exchange than waiting until each position is closed? (The operator chapter's Why settle paper profits at all? box is a hint.)
2. The engine announces `CLOSED` when it starts. Why must `pm-eod` *not* treat that as the day's close? What would a trader's statement say if the exchange were started in the morning and `pm-eod` settled the day at once?
3. `pm-eod` refuses to settle a day while an earlier closed day is still waiting. What would go wrong with the balances if it allowed it?

22.16 Further Reading

- Closing the Day — the scheduler hook, running without a scheduler, backups and restore, troubleshooting
- Your End-of-Day Statement
- `pm-eod` reference
- Positions and P&L

Next: 21 — Drop-Copy Replay & Recovery Patterns

Part V

Integration and advanced topics

Chapters 21–29. Recover a drop-copy consumer, automate the operator, connect over every external protocol, run a market index and price an IPO.

CHAPTER 23

Drop-Copy Replay & Recovery Patterns

23.1 Objective

Learn how to operate drop-copy consumers safely when connectivity gaps occur, how sequence numbers are used for loss detection, and what replay support does and does not exist in the current EduMatcher runtime.

Background reading

- Drop Copy - DC Gateway

23.2 Prerequisites

- Chapters 01-20 completed.
- Engine running with drop-copy PUB enabled on `:5557`.
- Basic Python command-line familiarity.

23.3 Background

Drop copy publishes per-participant fill events on:

- `drop_copy.event.<gateway_id>` (live stream)

Each message carries a `seq` field that increases contiguously within its gateway's stream. Consumers use it to detect gaps and de-duplicate messages. Each gateway has an independent counter, starting at 1; it resets when the engine restarts. A consumer listening to multiple gateways must track the last sequence separately for each `gateway_id`.

Important current limitation:

- Replay exists as an in-process publisher method (`DropCopyPublisher.replay(recipient_id, from_seq, gateway_id=...)`), backed by one circular in-memory buffer of the last 10,000 events across all gateways (roughly 16 minutes of history at a sustained ~10 fills/second).
- There is currently no external ZMQ replay-request handler for downstream clients to call directly — and even once one exists, replay is bounded by that 10,000-event buffer, not unlimited history.

So operationally, reconnect handling is currently:

1. Detect gap by sequence numbers.
2. Recover from available audit/statistics artifacts.
3. Re-establish live stream and continue from new high-water mark.

23.4 Exercise 1: Subscribe and Print Sequence Numbers

Run a minimal drop-copy subscriber:

```
python - <<'PY'
import json
import zmq

ctx = zmq.Context()
sock = ctx.socket(zmq.SUB)
sock.connect("tcp://127.0.0.1:5557")
sock.subscribe(b"drop_copy.event.TRADER01")

print("listening on drop_copy.event.TRADER01")
for _ in range(10):
    topic, payload = sock.recv_multipart()
    msg = json.loads(payload)
    print(topic.decode(), "seq=", msg.get("seq"), "symbol=", msg.get("symbol"))
PY
```

Generate a few trades from gateways while this runs.

:material-checkbox-blank-outline: **Checkpoint:** you can observe increasing `seq` values on live drop-copy events.

23.5 Exercise 2: Detect a Synthetic Gap

Use a subscriber with simple gap detection:

```
python - <<'PY'
import json
import zmq

last = None
ctx = zmq.Context()
sock = ctx.socket(zmq.SUB)
sock.connect("tcp://127.0.0.1:5557")
sock.subscribe(b"drop_copy.event.TRADER01")

print("watching for sequence gaps")
for _ in range(30):
    topic, payload = sock.recv_multipart()
    if topic != b"drop_copy.event.TRADER01":  # prefix match: skip TRADER010...
        continue
    msg = json.loads(payload)
    seq = msg.get("seq")
    if last is not None and seq is not None and seq != last + 1:
        print(f"GAP detected: expected {last + 1}, got {seq}")
    last = seq
PY
```

Restarting the subscriber will not show you a gap

A restart resets `last` to `None`, and the guard `if last is not None` exempts the first message after every restart. That is correct behaviour for a real consumer — on a cold start it has no expectation to violate — but it means the obvious experiment cannot produce a `GAP detected` line.

To actually see the detector fire, drop messages *inside* a single run. This version discards every fifth event it receives, simulating lost datagrams:

```
python - <<'EOF'
import json
import zmq

last = None
seen = 0
ctx = zmq.Context()
sock = ctx.socket(zmq.SUB)
sock.connect("tcp://127.0.0.1:5557")
sock.subscribe(b"drop_copy.event.TRADER01")

print("watching for gaps (deliberately dropping every 5th event)")
for _ in range(30):
    topic, payload = sock.recv_multipart()
    if topic != b"drop_copy.event.TRADER01":  # prefix match: skip TRADER010...
        continue
    seen += 1
    if seen % 5 == 0:          # simulate a lost message
        continue
    msg = json.loads(payload)
    seq = msg.get("seq")
    if last is not None and seq is not None and seq != last + 1:
        print(f"GAP detected: expected {last + 1}, got {seq}")
    last = seq
EOF
```

With trades flowing you should now see a `GAP detected` line roughly every fifth event. That is the whole mechanism: the **consumer**, not the publisher, notices the discontinuity — which is why every consumer you write needs this check.

:material-checkbox-blank-outline: **Checkpoint:** you have seen at least one `GAP detected` line, and can explain why a subscriber restart does not produce one.

23.6 Exercise 3: Validate Current Replay Constraint

Confirm current behavior from docs:

1. Review replay section in Drop Copy.
2. Note that replay is in-process and there is no external replay-request handler wired into the engine loop.

Operational implication:

- Downstream clients cannot request historical replay over ZMQ yet.

Current workaround vs. a future replay API

Today: if a subscriber detects a sequence gap, it must reconstruct lost events itself — cross-referencing `pm-audit`'s append-only log and `pm-stats-cli` trade history (Exercise 4), because there is no message it can send back to the engine asking "resend events N through M".
If/when an external replay-request handler is added: a subscriber would instead publish a request (e.g. on a `drop_copy.replay.<recipient_id>` topic) naming the missing sequence range, and the engine

would push just those events back — removing the need to reconstruct history from audit logs and stats queries. The recovery workflow in Exercise 4 is the practical stand-in for that capability until it exists; the underlying goal (resume with no lost events) is the same either way, only the mechanism differs.
:material-checkbox-blank-outline: **Checkpoint:** you can state the exact replay limitation and its impact on downstream consumers.

23.7 Exercise 4: Recovery Workflow with Audit + Stats

When a gap is detected, run a practical recovery checklist:

1. Use `pm-audit --terminal --audit-log-file ...` to keep an append-only event trail. Note the flag: `--audit-log-file` sets the *audit trail* path; `--log-file` is the process's own operational log and will not contain trade events.
2. Use `pm-stats-cli trades --symbol <sym>` for post-gap trade verification.
3. Reconnect subscriber and continue from latest observed sequence.

Example verification commands:

```
pm-stats-cli trades --symbol AAPL --limit 20
tail -40 "$EDUMATCHER_DATA_DIR/audit.log"
```

:material-checkbox-blank-outline: **Checkpoint:** you can recover operational confidence after a subscriber interruption even without external replay requests.

23.8 Exercise 5: Design a Consumer Gap Policy

Write a short policy for your drop-copy consumer:

- Gap detection rule (strictly sequential or tolerance window).
- Duplicate handling rule (ignore already-processed sequence IDs).
- Recovery source order (audit log first, stats snapshot second, then live stream).
- Alerting threshold (for example, gap size > N triggers operator page).

:material-checkbox-blank-outline: **Checkpoint:** your policy defines deterministic behavior for normal, degraded, and recovery modes.

23.9 Summary

You now understand practical drop-copy reliability operations in EduMatcher:

- How to detect stream loss using `seq`.
- Why external replay is currently limited.
- How to run a safe recovery pattern using available tooling.

23.10 Reflection

Why does the drop-copy feed number every event with a monotonic `seq` instead of relying on subscribers to just count messages received? What silent failure mode would go undetected if `seq` didn't exist and a subscriber's socket briefly dropped messages?

23.11 Further Reading

- Drop Copy
- Messages
- Persistence

Next: 22 - Automation with CommandClient & MM Bot Tuning

CHAPTER 24

Automation with CommandClient & MM Bot Tuning

24.1 Objective

Use EduMatcher's programmatic command client for repeatable admin workflows and practice advanced `pm-mm-bot` runtime tuning for startup and reconciliation.

Background reading

- The Admin Console and Exchange Commands - Market-Maker Bot

24.2 Prerequisites

- Chapters 01-21 completed.
- Engine running with `GW_ADMIN` and at least one MM gateway configured.
- Python environment able to import `edumatcher.commands`.

24.3 Background

The interactive tools are excellent for manual operation, but production-like operations often require deterministic automation:

- One-shot operator runbooks (halt/resume, symbol cleanup, gateway checks).
- Repeatable incident actions with explicit timeouts.
- MM bot tuning for startup reliability in sparse-book conditions.

This chapter combines two advanced surfaces:

1. `ExchangeCommandClient` (programmatic admin command API).
2. Advanced `pm-mm-bot` runtime flags such as bootstrap timeout and QLEGS reconciliation interval.

Close pm-admin and the MM console first

`ExchangeCommandClient("GW_ADMIN")` logs on as `GW_ADMIN`, and the bot in Exercises 4–5 as `MM_AAPL_01`. The engine allows one connection per participant, so leave any open `pm-admin --id GW_ADMIN` and `pm-alf-console --id MM_AAPL_01` (`Ctrl-C`) before running the scripts. If you forget, `connect()` does **not** raise: it returns `accepted: False, reason: 'Gateway already connected: GW_ADMIN'`, read-only calls such as `symbol_list()` still work, and only the commands that change something fail. A script that matters should check `auth["accepted"]`.

24.4 Exercise 1: Run a Minimal CommandClient Session

Execute a short Python script:

```
python - <<'PY'
from edumatcher.commands import ExchangeCommandClient

with ExchangeCommandClient("GW_ADMIN") as client:
    auth = client.connect()
    print("auth:", auth)
    symbols = client.symbol_list()
    print("symbols:", symbols)
    state = client.session_status()
    print("session:", state)
PY
```

:material-checkbox-blank-outline: **Checkpoint:** you can connect, read symbols, and read session status programmatically.

24.5 Exercise 2: Script a Safe Symbol-Protection Runbook

Run a scripted sequence:

```
python - <<'PY'
from edumatcher.commands import ExchangeCommandClient

with ExchangeCommandClient("GW_ADMIN") as c:
    c.connect()
    c.symbol_halt("AAPL")
    c.cancel_symbol("AAPL")
    book = c.book_depth("AAPL")
    print("AAPL bids:", len(book.get("bids", [])), "asks:", len(book.get("asks", [])))
    c.symbol_resume("AAPL")
    print("AAPL resumed")
PY
```

This is easier to run consistently than a manual multi-step console sequence.

:material-checkbox-blank-outline: **Checkpoint:** you can automate halt/cancel/verify/resume for one symbol.

24.6 Exercise 3: Automate Gateway Exposure Cleanup

Use API methods to clear one participant and verify no resting orders remain:

```
python - <<'PY'
from edumatcher.commands import ExchangeCommandClient

target = "TRADER02"

with ExchangeCommandClient("GW_ADMIN") as c:
    c.connect()
    c.kill_switch(target)
    orders = c.order_list(target)
    print("remaining orders for", target, ":", len(orders))
PY
```

Optional extension:

- Add `gateway_kick(target, reason=...)` after `kill_switch` when operational policy requires immediate disconnect.

:material-checkbox-blank-outline: **Checkpoint:** you can explain when to use kill-switch only vs. kill-switch + kick.

24.7 Exercise 4: Tune pm-mm-bot Startup Reliability

Run one bot with explicit startup controls. A bot started on an empty book with no range to draw a price from stops with `startup failed: no reference price available`, and the last trade price saved from an earlier session does not count, so the command gives it a start range:

```
pm-mm-bot \
  --symbol AAPL \
  --gap 0.10 \
  --qty 500 \
  --initial_min 149.00 \
  --initial_max 151.00 \
  --startup-session-timeout-sec 5.0 \
  --bootstrap-timeout-sec 1.0 \
  --qlegs-reconcile-interval-sec 15.0 \
  -v
```

Observe startup logs for:

- Bootstrap reference resolution (existing quote adopted, or a random-range price).
- Session readiness before the first quote.
- QLEGS reconciliation, which is silent while everything agrees and only logs when it finds a mismatch.

Every bot log line is prefixed with the bot's gateway ID (`[<gateway_id>]`), and per-symbol lines add a `[<symbol>]` tag. Representative startup lines (after the usual timestamp/level/logger prefix):

```
[MM_AAPL_01] starting: gateway=MM_AAPL_01 symbols=AAPL
[MM_AAPL_01] [AAPL] strategy=symmetric gap=0.1 qty=500 tif=DAY drift_ticks=3
[MM_AAPL_01] [AAPL] bootstrap from random range: 150.37
[MM_AAPL_01] running symbols=['AAPL'] session=CONTINUOUS
```

The `bootstrap from random range` line only appears when the price came from `--initial_min`/`--initial_max` (Exercise 5); with a live book the reference comes from the book and `-v` shows it as `[<symbol>] reference from book/trade: <price>`. There is no per-quote `INFO` line — use `QLEGS|SYM=AAPL` on the MM gateway to confirm the quote is live.

:material-checkbox-blank-outline: **Checkpoint:** you can identify and tune the timeout knobs that control startup behavior.

Most of these knobs can differ per symbol

`--bootstrap-timeout-sec` and `--qlegs-reconcile-interval-sec` are per-symbol settings: on a multi-symbol bot each symbol may be given its own, either by scoping the flag after that symbol's `--symbol` or from its block in a config file (see 02 — Setting Up Market-Maker Liquidity, Exercise 8). `--startup-session-timeout-sec` is the exception and stays process-wide, because the session phase it waits for is exchange-wide and there is nothing per-symbol to express.
The QBOOT/QLEGS *requests themselves*, and the startup/gap checks they inform, run once per symbol regardless. A `pm-mm-bot --symbols AAPL,MSFT,TSLA -v` run shows three independent `[AAPL]`/`[MSFT]`/`[TSLA]`-tagged QBOOT and QLEGS exchanges in the startup log, not one — so a `--bootstrap-timeout-sec` tuned down aggressively for a fast classroom demo is a budget applied to *each* symbol's round trip in turn, not split across them.

24.8 Exercise 5: Empty-Book Bootstrap Drill

Simulate sparse startup conditions and run bot with explicit bootstrap range:

```
pm-mm-bot --symbol AAPL --initial_min 149.00 --initial_max 151.00 -v
```

Then compare behavior with and without range configured (stop the first bot with `Ctrl-C` before you start the next; it cancels its quote on the way out).

Expected understanding:

- With range: bot can start quoting on fresh books.
- Without any bootstrap source: bot fails fast with clear reason.

Fails fast per symbol, not necessarily for the whole process

On a `--symbols` bot, "without any bootstrap source" excludes *that* symbol from quoting rather than stopping the process outright, as long as at least one other symbol resolves a reference price — see Per-symbol failure isolation. The single-symbol case in this exercise is the special case where there is no "other symbol" left, so the process-level failure you'll observe here is the same behavior applied to a symbol set of one. `--initial_min`/ `--initial_max` are per-symbol settings too: give them once before any `--symbol` and every symbol draws from that one range (expect a different random price per symbol), or scope them to a single symbol to give a volatile name a range of its own.
:material-checkbox-blank-outline: **Checkpoint:** you can choose a bootstrap strategy appropriate for your environment.

24.9 Exercise 6: Build a Combined Automation Flow

Design a short automation script that:

1. Checks session status.
2. Halts one symbol if needed.
3. Clears that symbol's resting orders.
4. Resumes symbol.
5. Verifies top-of-book depth.

Use `ExchangeCommandClient` methods only (no manual prompt commands).

Acceptance criteria — your script passes if all of the following hold:

- **Correct halt response**: the halt call returns a success/ack response for the target symbol (not a generic exception or timeout).
- **Book actually empties**: after the clear step, a book query for that symbol shows zero resting orders on both sides before you resume it.
- **Correct resume response**: the resume call returns success and a subsequent `SYMBOL_STATUS`-style query (or equivalent client method) shows the symbol back in a tradeable state.
- **Idempotent rerun**: running the entire script a second time immediately afterward produces the same end state (symbol active, book empty of the orders your script itself cleared) without raising an error — halting an already-halted symbol or clearing an already-empty book must not crash the script.
- **No orphaned state**: after the script finishes, no test order placed by the script remains resting outside of what step 5 intentionally verifies.

:material-checkbox-blank-outline: **Checkpoint:** your script is deterministic, idempotent, and easy to rerun during drills, and satisfies all five acceptance criteria above.

24.10 Summary

You now have advanced operational coverage for:

- Programmatic admin orchestration with `ExchangeCommandClient`.
- Repeatable incident-response style command sequencing.
- Practical `pm-mm-bot` tuning for startup/bootstrap/reconciliation behavior.

24.11 Reflection

Why does this chapter insist your automation scripts be idempotent (safe to rerun) rather than accepting "runs correctly once" as good enough? Think about an incident-response scenario at 3am — what goes wrong if an on-call engineer reruns a non-idempotent halt/clear/resume script by mistake?

24.12 Further Reading

- Controlling the Exchange
- The Market-Maker Bot (`pm-mm-bot`)
- Market-Maker Bot CLI Reference
- Market-Maker Bot — Per-symbol failure isolation
- 02 — Setting Up Market-Maker Liquidity — one bot, multiple symbols
- Processes, Environment and Ports

You have completed the operator track. The remaining chapters (23–28) cover the external protocols — RALF, CALF, the REST/WebSocket API, the market index, and the ALF/BALF TCP gateways.

CHAPTER 25

RALF Post-Trade Gateway Protocol Training

25.1 Objective

Train on end-to-end usage of the RALF protocol through pm-ralf-gwy using the example parser/subscriber libraries for the three main external party roles:

- `CLEARING`
- `DROP_COPY`
- `AUDIT`

You will practice connection, subscription, live event handling, heartbeats, replay checkpoints, and recovery/error behavior.

Background reading

- RALF Gateway - RALF Protocol

25.2 Prerequisites

- Chapters 01-22 completed.
- Engine running and producing trades.
- `pm-ralf-gwy` running and reachable on TCP (default 5580).
- Support libraries and clients available in `docs/examples/ralf`:
  - ralf_parser.py, ralf_subscriber.py
  - ralf_parser.h, ralf_parser.c, ralf_subscriber.c

Recommended startup terminals:

1. Engine: `pm-engine --verbose`
2. RALF gateway: `pm-ralf-gwy`
3. One or more client terminals for subscriber exercises

Where the example files are, and port 5580

- `docs/examples/ralf` exists in a repository checkout. On a container or `pipx` install use the copy that ships with the package: `EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_root().parent)")` and then `$EX/ralf/…` wherever this chapter says `docs/examples/ralf`. The C example needs `make` and a C compiler, which the container image does not have; the Python subscriber and `nc` are enough for every exercise. - Port 5580 can only have one owner. If the AI-simulation container is running, its own `pm-ralf-gwy` holds the port: yours will fail to start, and `nc 127.0.0.1 5580` would talk to the *simulation's* gateway (you would see symbols you never configured). Stop the simulation first, or pick another port in Exercise 1. - There are no market makers in this chapter's configuration. To create a trade, put a limit order in from each trader, for example `NEW|SYM=AAPL|SIDE=SELL|TYPE=LIMIT|QTY=100|PRICE=150.00|TIF=DAY` on `TRADER01` and then `NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=60|PRICE=150.00|TIF=DAY` on `TRADER02`.

25.3 Background

RALF is a line-oriented protocol over TCP used for post-trade dissemination. Clients connect to pm-ralf-gwy, perform HELLO and SUB, then consume events.

In this chapter, the most important operational ideas are:

- Session establishment with HELLO and WELCOME
- Channel and symbol subscription with SUB and UNSUB
- Liveness with HB and PING/PONG
- Replay with LASTSEQ and REPLAY_MISS handling
- Role/channel design for different external consumers

25.4 Exercise 1: Prepare a Config That Enables RALF

Generate `engine_config.yaml` with a post_trade_gateway section:

```
pm-config-gen \
  --symbols AAPL MSFT \
  --participants TRADER01 TRADER02 OPS01:ADMIN \
  --sessions-enabled \
  --post-trade-gateway \
  --post-trade-bind-address 127.0.0.1 \
  --post-trade-port 5580 \
  --post-trade-allowed-roles CLEARING DROP_COPY AUDIT \
  --output engine_config.yaml --force
```

Back up, check, deploy — then restart

This chapter starts from a fresh configuration (its operator is `OPS01`), and `--force` replaces your `engine_config.yaml`: save a copy first if you want to come back to it (`cp engine_config.yaml engine_config.before-ch23.yaml`). As always, nothing changes until you deploy it and restart the engine: `pm-cverifier engine_config.yaml`, then `pm-config-deploy engine_config.yaml`. Sessions are enabled, so the restarted engine starts in `CLOSED`: open the market from the operator console (`pm-admin --id OPS01`, then `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS`).
Start or restart processes with that config:

```
pm-engine --verbose
pm-ralf-gwy
```

:material-checkbox-blank-outline: Checkpoint: gateway starts and listens on the configured port.

25.5 Exercise 2: Use the Python Example as a CLEARING Consumer

From the example directory:

```
cd docs/examples/ralf
python3 ralf_subscriber.py \
  --host 127.0.0.1 \
  --port 5580 \
  --role CLEARING \
  --channels CLEARING \
  --symbols '*'
```

Observe:

- WELCOME line parsed by the Python library
- subscription acknowledgment flow
- incoming EXEC and EOD messages for CLEARING

:material-checkbox-blank-outline: Checkpoint: subscriber prints parsed EXEC events with expected symbols and sequence values.

25.6 Exercise 2B: Manual Handshake Probe with nc

Before continuing with richer clients, validate a minimal manual protocol flow:

```
nc 127.0.0.1 5580
```

Then send:

```
HELLO|CLIENT=manual01|PROTO=RALF1|ROLE=CLEARING|LASTSEQ=0
SUB|CH=CLEARING|SYM=*
```

Expected behavior:

- `WELCOME|...` after `HELLO`
- `SNAP|...` after `SUB`
- live `EXEC|...` / `EOD|...` once trades are produced

:material-checkbox-blank-outline: Checkpoint: manual nc session can establish, subscribe, and receive at least one post-trade message.

25.7 Optional Exercise 3: Use the C Example as a DROP_COPY Consumer

This C exercise requires `make` and a C compiler, which are not installed in the container image. Skip it there, or build it on a host with those tools. Exercise 2's Python subscriber can also consume RALF channels without a compiler.

Build and run the C subscriber:

```
cd docs/examples/ralf
make
./ralf_subscriber 127.0.0.1 5580 DROP_COPY
```

Observe in output:

- WELCOME fields
- SUB flow
- MSG lines for live traffic

Even though the example subscribes broadly for demonstration, run it with a DROP_COPY role argument and validate that role is accepted.

:material-checkbox-blank-outline: Optional checkpoint: C subscriber connects, parses, and prints live RALF events.

25.8 Exercise 4: Run an AUDIT Session and Compare View Semantics

Run another Python subscriber in a second client terminal:

```
python3 docs/examples/ralf/ralf_subscriber.py \
  --host 127.0.0.1 \
  --port 5580 \
  --role AUDIT \
  --channels AUDIT \
  --symbols '*'
```

Compare outputs across CLEARING, DROP_COPY, and AUDIT clients while generating new trades.

Discussion points:

- channel identifiers in CH
- sequence progression
- symbol filtering behavior

:material-checkbox-blank-outline: Checkpoint: you can explain why separate external roles may consume different channel sets.

25.9 Exercise 5: Protocol Control Messages (PING, UNSUB, EXIT)

Use netcat or telnet for manual control-message testing:

```
nc 127.0.0.1 5580
```

Then send lines:

```
HELLO|CLIENT=manual01|PROTO=RALF1|ROLE=CLEARING|LASTSEQ=0
SUB|CH=CLEARING|SYM=*
PING
UNSUB|CH=CLEARING|SYM=AAPL
```

Expected behavior:

- PING yields `PONG|TS=…`
- UNSUB updates the delivery set and sends no reply of its own: trades in AAPL stop arriving, other symbols continue

EXIT goes both ways

The gateway sends `EXIT|REASON=…|TS=…` when it ends a session (e.g. `EXIT|REASON=idle_timeout`). A client may send a bare `EXIT` to end its own session; the gateway closes after flushing pending output. All three line gateways (`pm-md-gwy`, `pm-ralf-gwy`, `pm-dc-gwy`) behave the same.
:material-checkbox-blank-outline: Checkpoint: you can manually drive and verify protocol control flow, and you can explain what each side's `EXIT` means.

25.10 Support Libraries and Example Clients

Reference implementations used in this training chapter:

- docs/examples/ralf/ralf_parser.py
- docs/examples/ralf/ralf_subscriber.py
- docs/examples/ralf/ralf_parser.h
- docs/examples/ralf/ralf_parser.c
- docs/examples/ralf/ralf_subscriber.c

Use these to bootstrap both quick lab subscribers and production-like integration test harnesses.

25.11 Exercise 6: Replay and Recovery with LASTSEQ

1. Start a subscriber and note the highest SEQ seen.
2. Disconnect client.
3. Generate additional trades.
4. Reconnect with LASTSEQ set to the saved sequence:

```
HELLO|CLIENT=replay01|PROTO=RALF1|ROLE=CLEARING|LASTSEQ=<saved_seq>
```

Observe replay behavior:

- replayed EXEC/EOD events for SEQ greater than LASTSEQ when retained; they arrive immediately after `WELCOME`, before you have sent `SUB`
- `ERR|CODE=REPLAY_MISS|DETAIL=requested seq outside retention` followed by a `SNAP` baseline when outside the retention window (try `LASTSEQ=999`)

:material-checkbox-blank-outline: Checkpoint: you can describe the recovery path for both replay-hit and replay-miss cases.

25.12 Exercise 7: Error Conditions and Operational Interpretation

Test typical protocol errors with malformed lines:

```
SUB|CH=CLEARING|SYM=*                (before HELLO)
HELLO|CLIENT=x|PROTO=RALF1           (missing ROLE)
HELLO|CLIENT=x|PROTO=RALF1|ROLE=BAD
SUB|CH=UNKNOWN|SYM=*
```

The gateway closes the connection after a handshake error, so open a fresh `nc` session for each line.

What you should see for each line, in order: `ERR|CODE=AUTH_REQUIRED|DETAIL=send HELLO first`; `ERR|CODE=AUTH_RE` `required`; `ERR|CODE=ENTITLEMENT_DENIED|DETAIL=role not allowed`; `ERR|CODE=INVALID_CHANNEL|DETAIL=unsupporte` `CH=UNKNOWN` (this one needs a valid `HELLO` first). A valid `HELLO` followed by `SUB|CH=AUDIT|SYM=*` as role `CLEARING` gives `ENTITLEMENT_DENIED` too, because a role may only subscribe to its own channel. `BAD_MESSAGE` and `SLOW_CLIENT` are not easy to trigger by hand.

Map observed ERR codes to operator action:

- AUTH_REQUIRED: client handshake bug or ordering error
- BAD_MESSAGE: parse or field validation issue
- ENTITLEMENT_DENIED: role not allowed by gateway config
- INVALID_CHANNEL: unsupported CH value
- SLOW_CLIENT: consumer cannot keep up with delivery rate

:material-checkbox-blank-outline: Checkpoint: you can convert each ERR code into a practical remediation step.

25.13 Suggested Lab Pattern for the Three External Parties

Run three long-lived sessions in parallel:

1. CLEARING client for full clearing consumption
2. DROP_COPY client for compliance/risk copy workflow
3. AUDIT client for forensic/event journaling use

For each session, record:

- client id and role
- channel set
- last checkpoint sequence
- reconnect policy

This produces a realistic external-party operating model for classroom or integration testing.

25.14 Summary

You have now covered major RALF protocol usage patterns:

- provisioning configuration with pm-config-gen
- connecting external clients through example libraries
- handling role/channel subscriptions
- using control messages and liveness probes
- recovering with LASTSEQ replay semantics
- diagnosing protocol errors operationally

25.15 Reflection

Compare the three roles you just used, from a recovery-priorities standpoint:

- **CLEARING**: consumes trade/settlement events. A gap here directly risks incorrect positions or P&L

— what makes this role the least tolerant of missed messages, and how does LASTSEQ replay (Exercise 6) address that?

- **DROP_COPY**: consumes a compliance/risk copy of activity. Is a short gap here as operationally urgent as a CLEARING gap? Why or why not — consider who consumes this data and on what timescale they act on it.
- **AUDIT**: consumes a forensic/event journal. If AUDIT falls behind or drops messages, is it recoverable after the fact from other sources (e.g. the exchange's own audit log from Chapter 16), and does that change how urgently you'd page an on-call engineer compared to a CLEARING gap?

Write one sentence for each role stating its recovery priority (immediate / same-day / best-effort) and why — this is the kind of judgment call a real operations team makes when deciding which reconnect alerts to treat as incidents versus routine noise.

25.16 Further Reading

- Post-Trade Dissemination
- RALF Protocol Spy (pm-ralf-spy)
- RALF Protocol Appendix
- Protocol Support Library Examples
- Processes, Environment and Ports

CHAPTER 26

CALF Market-Data Gateway Protocol Training

26.1 Objective

Train on end-to-end usage of the CALF protocol through pm-md-gwy using the example parser/subscriber libraries for external market-data consumers.

You will practice connection, subscription, snapshots, live market-data consumption, heartbeats, replay checkpoints, and recovery/error behavior.

Background reading

- CALF Gateway - CALF Protocol

26.2 Prerequisites

- Chapters 01-23 completed.
- Engine running and producing market activity.
- `pm-md-gwy` running and reachable on TCP (default 5570).
- `pm-calf-spy` installed (ships with the `edumatcher` package — no separate build step).
- Support libraries and clients available in `docs/examples/calf`:
  - calf_parser.py, calf_subscriber.py
  - calf_parser.h, calf_parser.c, calf_subscriber.c

Recommended startup terminals:

1. Engine: `pm-engine --verbose`
2. CALF gateway: `pm-md-gwy`
3. One or more client terminals for subscriber exercises

Where the example files are, and port 5570

As in the previous chapter, `docs/examples/calf` exists in a repository checkout; on a container or `pipx` install use `EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_root().parent)")` and `$EX/calf/…`. The C example needs `make` and a C compiler (not in the container image). If the AI-simulation container is running, its own `pm-md-gwy` already holds port

5570. Start `pm-md-gwy` a few seconds after the engine and before the first client: a client started too early gets `Connection refused`. This chapter's configuration has no market makers. To create a trade, enter a limit sell from `TRADER01` and a limit buy at the same price from `TRADER02` (as in the previous chapter).

26.3 Background

CALF is a line-oriented protocol over TCP used for external market-data dissemination. Clients connect to pm-md-gwy, perform HELLO and SUB, then consume stream data.

In this chapter, the most important operational ideas are:

- Session establishment with HELLO and WELCOME, including per-symbol reference data (`REF=SYM:DEC`) for display precision
- Channel and symbol subscription with SUB and UNSUB, across all seven channels: TOP, TRADE, STATE, INDEX, DEPTH, AUCTION, CB
- Baseline snapshots (`SNAP`) plus incremental live events (`MD`, `TRADE`, `STATE`, `AUCTION`, `INDIC`, `CB`)
- Liveness with HB and PING/PONG
- Replay with the standalone `RESUME` command / LASTSEQ, and REPLAY_MISS handling
- Auction uncross results (`AUCTION`, with a `REASON` for what kind of uncross it was) and indicative pricing during the call phase (`INDIC`)
- Circuit-breaker detail (`CB`), including its cause (`SRC`) and Automated Corridor Expansion (`CORRLO`/`CORRHI`/`EXP` closing backstop prints)
- `pm-calf-spy` as the ready-made, read-only way to inspect any of the above without writing a client

26.4 Exercise 1: Prepare a Config That Enables CALF

Generate `engine_config.yaml` with a market_data_gateway section:

```
pm-config-gen \
  --symbols AAPL MSFT \
  --participants TRADER01 TRADER02 OPS01:ADMIN \
  --sessions-enabled \
  --market-data-gateway \
  --market-data-bind-address 127.0.0.1 \
  --market-data-port 5570 \
  --market-data-replay-window-sec 30 \
  --market-data-heartbeat-interval-sec 1 \
  --market-data-idle-timeout-sec 5 \
  --output engine_config.yaml --force
```

Back up, check, deploy — then restart

This chapter starts from a fresh configuration (its operator is `OPS01`), and `--force` replaces your `engine_config.yaml`: save a copy first if you want to come back to it (`cp engine_config.yaml engine_config.before-ch24.yaml`). As always, nothing changes until you deploy it and restart the engine: `pm-cverifier engine_config.yaml`, then `pm-config-deploy engine_config.yaml`. Sessions are enabled, so the restarted engine starts in `CLOSED`: open the market from the operator console (`pm-admin --id OPS01`, then `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS`).
Start or restart processes with that config:

```
pm-engine --verbose
pm-md-gwy
```

:material-checkbox-blank-outline: Checkpoint: gateway starts and listens on the configured port.

26.5 Exercise 2: Use the Python Example as a TOP/TRADE Consumer

From the example directory. `calf_subscriber.py` has no `--channels` flag — it always subscribes to the Cartesian product of `{TOP, TRADE, STATE, DEPTH}` (filtered down to whatever `WELCOME|CH_SUPPORTED=` actually advertises) for the symbols you pass:

```
cd docs/examples/calf
python3 calf_subscriber.py \
  --host 127.0.0.1 \
  --port 5570 \
  --symbols AAPL
```

Observe:

- WELCOME line parsed by the Python library
- SNAP for TOP (and STATE, DEPTH) baseline
- incoming MD and TRADE messages for subscribed symbols, including each trade's durable `TRADE_ID` and `RUN_SEQ`

:material-checkbox-blank-outline: Checkpoint: subscriber prints parsed MD/TRADE events with expected symbol, sequence, and durable trade ID fields.

26.6 Exercise 2B: Manual Handshake Probe with nc

Before continuing with richer clients, validate a minimal manual protocol flow:

```
nc 127.0.0.1 5570
```

Then send:

```
HELLO|CLIENT=manual01|PROTO=CALF1
SUB|CH=TOP,TRADE|SYM=AAPL
```

Expected behavior:

- WELCOME after HELLO
- SNAP for CH=TOP after SUB
- live MD and TRADE once market activity occurs

Every `TRADE` line carries a durable `TRADE_ID` and its `RUN_SEQ`. Keep the trade ID when recording the tape: a `RESUME` response can overlap live traffic, and two lines with the same `TRADE_ID` are the same execution even when their delivery sequence differs.

:material-checkbox-blank-outline: Checkpoint: manual nc session can establish, subscribe, and receive at least one live market-data line.

26.7 Optional Exercise 3: Use the C Example Subscriber

This C exercise requires `make` and a C compiler, which are not installed in the container image. Skip it there, or build it on a host with those tools. Exercise 2's Python subscriber provides the same market-data client workflow.

Build and run the C subscriber:

```
cd docs/examples/calf
make
./calf_subscriber 127.0.0.1 5570
```

Observe in output:

- WELCOME fields
- SUB flow
- MSG lines for live traffic

:material-checkbox-blank-outline: Optional checkpoint: C subscriber connects, parses, and prints live CALF events.

26.8 Exercise 3B: `pm-calf-spy` — the Read-Only Diagnostic CLI

`calf_subscriber.py`/`.c` are example *libraries* to build on; `pm-calf-spy` is a ready-made, read-only diagnostic client for watching the wire directly, with no code to write. It runs fully passively — no reconnect, no auto-recovery, no local state rebuild — so what you see is exactly what the gateway sent, nothing injected or inferred on your behalf.

```
pm-calf-spy --channels TOP,TRADE --symbols AAPL
```

Try a few variations:

```
# Every channel the gateway advertises, every symbol (default)
pm-calf-spy

# JSON output instead of human-readable
pm-calf-spy --channels TOP,TRADE --symbols AAPL --format json

# Show heartbeats too (suppressed by default)
pm-calf-spy --channels STATE --symbols AAPL --show-heartbeats

# Raw wire line alongside the formatted one
pm-calf-spy --channels TOP --symbols AAPL --raw

# One-shot replay of a single stream
pm-calf-spy --resume TOP:AAPL:1042 --channels TOP --symbols AAPL

# Stop after 20 data-carrying lines
pm-calf-spy --channels TRADE --symbols '*' --count 20
```

Now try the wildcard default against a mix of eligible and ineligible channels:

```
pm-calf-spy --channels '*' --symbols '*'
```

Since `SYM=*` is only valid for `TOP`/`TRADE`/`STATE`/`AUCTION`, a naive single `SUB|CH=*|SYM=*` would be rejected out-

right by the gateway. `pm-calf-spy` instead splits the request into one `SUB` for the wildcard-eligible channels and one per concrete symbol for `DEPTH`/`INDEX`/ `CB`, and reports which channels it skipped for the wildcard rather than failing the whole subscription.

:material-checkbox-blank-outline: Checkpoint: you can explain why `pm-calf-spy` runs with reconnect and auto-recovery both off, and why `--channels '*' --symbols '*'` doesn't produce a single rejected `SUB`.

26.9 Exercise 4: State Channel and Wildcard Behavior

`calf_subscriber.py` automatically adds a session-wide `SUB|CH=STATE|SYM=*` on top of its per-symbol subscriptions (pass `--no-state-wildcard` to skip it):

```
python3 docs/examples/calf/calf_subscriber.py \
  --host 127.0.0.1 \
  --port 5570 \
  --symbols AAPL
```

Expected behavior:

- immediate SNAP for CH=STATE,SYM=* (session-wide) plus one for CH=STATE,SYM=AAPL (that symbol's own halt/resume stream)
- STATE updates when session phase or halt/resume transitions occur

:material-checkbox-blank-outline: Checkpoint: you can name the four wildcard-eligible channels (STATE, TOP, TRADE, AUCTION) and explain why DEPTH, INDEX and CB are not.

Note: as of CALF `1.0.0`, `SYM=*` is also valid for `TOP` and `TRADE` (see Exercise 7 below); as of the `AUCTION`/`CB` extension, `SYM=*` is valid for `AUCTION` too. It remains invalid for `INDEX`, `DEPTH`, and `CB`.

Note also: a symbol-level `STATE` subscription (`CH=STATE|SYM=AAPL`) no longer only reports that symbol's own halts and resumes. On every session-wide transition (open, close, phase change) the gateway now fans out a `STATE` event per known, non-halted symbol as well as the `SYM=*` line — so a client watching one instrument learns the exchange opened or closed without also subscribing to `SYM=*`. A halted symbol is skipped by this fan-out; its halt outlives the session phase it began in, and only an explicit resume moves it.

To see this, open two `nc` sessions: one `SUB|CH=STATE|SYM=*`, one `SUB|CH=STATE|SYM=AAPL`. Trigger (or wait for) a session-phase transition and compare — both should print a `STATE` line for the transition, not just the wildcard session.

:material-checkbox-blank-outline: Checkpoint: you can explain why a single-symbol `STATE` subscriber needs this fan-out to know the exchange opened or closed, and why a halted symbol is excluded from it.

26.10 Exercise 4B: Auction Results and Circuit-Breaker Detail

Two channels extend the original five: `AUCTION` (auction uncross results — no baseline `SNAP`, `SYM=*` allowed, mirrors `TRADE`) and `CB` (circuit-breaker halt/resume detail — cached baseline `SNAP`, `SYM=*` **not** allowed, mirrors `DEPTH`/`INDEX`). `calf_subscriber.py` does not drive these two channels, so exercise them manually with `nc`, alongside `STATE` for comparison:

```
nc 127.0.0.1 5570
```

```
HELLO|CLIENT=manual03|PROTO=CALF1
SUB|CH=CB,STATE|SYM=AAPL
SUB|CH=AUCTION|SYM=AAPL
```

Expected behavior:

- immediate `SNAP|CH=CB|SYM=AAPL|STATUS=ACTIVE|...` on subscribe (or `STATUS=HALTED` plus detail if already halted) — `AUCTION` gets no `SNAP`, same as `TRADE`
- an `AUCTION|...` line the next time AAPL's auction uncrosses (open, close, or a re-opening auction after a halt), carrying `EQPX`/`EQQTY`/`TRADES` and, if a residual exists, `IMBSIDE`/`IMBQTY`, plus a `REASON` of `SCHEDULED` (leaving an auction/other non-matching phase), `REOPEN` (a halted symbol reopening), or `RECOVERY` (GTC orders restored at engine startup) — without `REASON` the three would be indistinguishable on the wire
- a `CB|...` line on the next halt (`STATUS=HALTED` with `LEVEL`, `TRIGGERPX`, `REFPX`, `RESUMEAT`, `SRC`) and a matching one on resume (`STATUS=ACTIVE` with `SRC`) — `SRC` is `CB` for an automatic breaker trigger or `ADMIN` for an operator halt; it says what caused the halt, not how it will resume
- a `STATE|...` line for the same halt/resume, independently — compare the two: `STATE` gives you the simple `SESSION=HALTED`/`SESSION=CONTINUOUS` flag, `CB` gives you the operational detail behind it. They fire independently, not in a guaranteed order relative to each other

In a second `nc` session, try `SUB|CH=CB|SYM=*` and confirm it is rejected with `ERR|CODE=INVALID_SYMBOL` — unlike `AUCTION`, `TOP`, `TRADE`, and `STATE`, `CB` never accepts a wildcard symbol.

:material-checkbox-blank-outline: Checkpoint: you can explain why `AUCTION` has no `SNAP` but `CB` does, why `CB` rejects `SYM=*` while `AUCTION` accepts it, and what each `AUCTION|REASON=` value tells you that the rest of the fields don't.

26.11 Exercise 4C: Indicative Auction Price (`INDIC`)

While an opening or closing auction's call phase is running, the gateway publishes a repeated *indicative* uncross price on the same `AUCTION` channel — a different message type, `INDIC`, not a preview of `AUCTION` itself. It says what would happen if the phase ended right now, not what did happen.

```
nc 127.0.0.1 5570
```

```
HELLO|CLIENT=manual04|PROTO=CALF1
SUB|CH=AUCTION|SYM=AAPL
```

Time this around the opening or closing auction window (or trigger one via your session-state controls) and observe:

- repeated `INDIC|CH=AUCTION|SYM=AAPL|...` lines during the call phase, on a fixed interval (`auction_indicative` default 1s) — every reading republishes, even unchanged ones, so a client can tell a stable indicative from a stalled feed
- `INDICQTY` and `IMBQTY` always present; `INDICPX` omitted when the book would not cross at all — that omission is a reading ("nothing would trade"), not a gap
- exactly one final `AUCTION|...` line when the call phase ends. An uncross with no crossing interest is reported too, as `EQQTY=0|TRADES=0`; you will see one at every opening and closing auction, and for a reopening after a halt. To get a non-empty one, enter a crossing sell and buy while the call phase is open (for example `SESSION|STATE=CLOSING_AUCTION`, a sell of 50 at 149.50 and a buy of 40 at 151.00); the indicative then reads `INDICQTY=40|INDICPX=149.5|IMB=SELL` and the final line `EQQTY=40|EQPX=149.5|TRADES=1|REASON=SCHEDULED`

:material-checkbox-blank-outline: Checkpoint: you can explain why `INDIC` and `AUCTION` share a channel but are different message types, and why a missing `INDICPX` must not be rendered as a zero price.

26.12 Exercise 4D: Automated Corridor Expansion (ACE)

A circuit-breaker halt does not always end at its first scheduled `RESUMEAT`. If the indicative uncross price at that instant falls outside the reopening corridor (`CORRLO..CORRHI`), the halt extends: the corridor widens one rung, `RESUMEAT` moves out, and a fresh call phase begins. The `CB` channel carries this as a further `STATUS=HALTED` event on the same stream, not a new one.

```
nc 127.0.0.1 5570
```

```
HELLO|CLIENT=manual05|PROTO=CALF1
SUB|CH=CB,STATE|SYM=AAPL
```

Trigger a halt whose reopening indicative lands outside the corridor (or replay a recorded scenario that does) and observe:

- the initial halt's `SNAP/CB` line carrying `CORRLO`, `CORRHI`, `EXP=0` alongside the familiar `LEVEL/TRIGGERPX/REFPX/RE`
- an extension event: `STATUS=HALTED` again, with `EXP` incremented, a later `RESUMEAT`, a wider `[CORRLO, CORRHI]`, and event-only `INDICPX/ INDICQTY/IMB` describing the call phase that just failed to clear inside the old corridor
- **no accompanying `STATE` line for the extension** — the symbol was halted before and after, so the coarse session state hasn't changed; only `CB` reports it
- eventually a resume (`STATUS=ACTIVE`) once an indicative lands inside the corridor, or — if the trading day ends first — a resume carrying `REASON=CLOSING_BACKSTOP`, `CLAMPED=1`, and `PRINTPX`, meaning the exchange printed at the corridor boundary rather than a price the book discovered

:material-checkbox-blank-outline: Checkpoint: you can explain why a client that ignores `EXP/CORRLO/CORRHI` will wrongly report a halted symbol as overdue to reopen, and what `CLAMPED=1` tells you about a `CLOSING_BACKSTOP` print.

26.13 Exercise 4E: Reference Data — Tick Decimals (`REF`)

Every instrument has a display precision (`tick_decimals`) that a market data client otherwise has no way to discover. `WELCOME` and the `SYMBOLS` reply both carry it as `REF=SYM:DEC,...`, matching the symbol set in `SYMBOLS=`.

```
nc 127.0.0.1 5570
```

```
HELLO|CLIENT=manual06|PROTO=CALF1
SYMBOLS
```

Observe:

- `WELCOME|...|SYMBOLS=...|REF=...` on connect — `REF` covers exactly the symbols listed in `SYMBOLS`, in the same `SYM:DEC` order
- the `SYMBOLS` reply repeats the same pairing: `SYMBOLS|COUNT=n|SYMBOLS=...|REF=...`
- a symbol you configured with a non-default `tick_decimals` (e.g. `4`) reports that value, not the fallback of `2`

:material-checkbox-blank-outline: Checkpoint: you can explain why `REF` rides the handshake/`SYMBOLS` reply

instead of being repeated on every `TOP`/`MD` line, and what a client should assume for a symbol missing from `REF` entirely (older gateway, no capability).

26.14 Exercise 5: Protocol Control Messages (PING, UNSUB, EXIT)

Use netcat for manual control-message testing:

```
nc 127.0.0.1 5570
```

Then send lines:

```
HELLO|CLIENT=manual02|PROTO=CALF1
SUB|CH=TOP|SYM=AAPL
PING
UNSUB|CH=TOP|SYM=AAPL
EXIT
```

Expected behavior:

- PING yields PONG
- UNSUB removes delivery for that stream
- EXIT closes session cleanly

:material-checkbox-blank-outline: Checkpoint: you can manually drive and verify control flow.

26.15 Exercise 6: Replay and Recovery with RESUME

1. Start a subscriber and note the highest SEQ for (TOP, AAPL).
2. Disconnect client.
3. Generate additional market activity.
4. Reconnect with a plain `HELLO`, then send `RESUME` for that stream:

```
HELLO|CLIENT=replay01|PROTO=CALF1
RESUME|CH=TOP|SYM=AAPL|LASTSEQ=<saved_seq>
```

`RESUME` is a command in its own right, sent after `WELCOME`, and repeatable — send one per stream you were following. Older builds carried it as a `RESUME=1` flag on `HELLO`, which could only ever be honoured once per connection; that form is no longer accepted.

Observe replay behavior:

- replayed events for SEQ greater than LASTSEQ when retained. This chapter's configuration keeps only **30 seconds** of replay history (`--market-data-replay-window-sec 30`), so do steps 1–4 promptly; a `RESUME|CH=TOP|…` for events older than that is answered with `REPLAY_MISS` (try `LASTSEQ=1` after a minute to see it) and `TRADE` shows how a replay hit looks
- REPLAY_MISS plus a SNAP baseline when outside the replay window — but on `TOP`/`STATE`/`INDEX`/`DEPTH`/`CB` only. On `TRADE` and `AUCTION` no SNAP follows, because there is no snapshot of a print that already happened.

5. Now repeat step 4 on `TRADE` instead of `TOP`, with a `LASTSEQ` several messages behind. Count the prints you receive against the ones you had already seen.

The reply carries **everything** past `LASTSEQ`, not just what you missed — including the message that re-

vealed the gap. Work out what a tape would look like if the client rendered them all, then read how `docs/examples/calf/calf_subscriber.py` avoids it.

:material-checkbox-blank-outline: Checkpoint: you can describe recovery for both replay-hit and replay-miss cases, say which channels get a SNAP on a miss, and explain why a replay reply must be de-duplicated.

26.16 Exercise 7: Error Conditions and Operational Interpretation

Test typical protocol errors with malformed lines:

```
SUB|CH=TOP|SYM=AAPL                  (before HELLO)
HELLO|CLIENT=x|PROTO=BAD
SUB|CH=UNKNOWN|SYM=AAPL
SUB|CH=DEPTH|SYM=*                   (invalid wildcard usage — DEPTH requires an explicit symbol)
SUB|CH=CB|SYM=*                      (invalid wildcard usage — CB requires an explicit symbol)
```

Note: as of CALF `1.0.0`, `SYM=*` is valid for `TOP`, `TRADE`, and `STATE` (e.g. `SUB|CH=TOP|SYM=*` now succeeds and returns one `SNAP` per known symbol). As of the `AUCTION`/`CB` extension, `SYM=*` is also valid for `AUCTION`. The wildcard is rejected for `INDEX`, `DEPTH`, and `CB`, which always require an explicit id/symbol — see the CALF protocol reference for details.

Map observed ERR codes to operator action:

- AUTH_REQUIRED: client handshake bug or ordering error, or a HELLO with a missing CLIENT or wrong PROTO
- INVALID_CHANNEL: unsupported CH value
- INVALID_SYMBOL: unknown symbol, or `SYM=*` used with `INDEX`/`DEPTH`/`CB`
- REPLAY_MISS: requested resume point outside replay retention
- RATE_LIMITED: client is sending commands too fast; throttle it
- SUB_LIMIT: too many subscriptions on one session; subscribe more narrowly
- SLOW_CLIENT: the consumer could not keep up; its backlog was dropped and the session closed — reconnect and use `RESUME` (Exercise 6)

A session the gateway ends for inactivity gets `EXIT|REASON=idle_timeout` before the socket closes.

:material-checkbox-blank-outline: Checkpoint: you can convert each ERR code into a practical remediation step.

26.17 Support Libraries and Example Clients

Reference implementations used in this training chapter:

- docs/examples/calf/calf_parser.py
- docs/examples/calf/calf_subscriber.py
- docs/examples/calf/calf_parser.h
- docs/examples/calf/calf_parser.c
- docs/examples/calf/calf_subscriber.c
- `pm-calf-spy` (installed CLI — no source to read, but the fastest way to point at a running gateway and see the wire)

Use these to bootstrap both quick lab subscribers and production-like integration test harnesses. Note that `calf_subscriber.py`/`.c` only drive `TOP`, `TRADE`, `STATE`, and `DEPTH` (filtered by `WELCOME|CH_SUPPORTED=`) — they

do not touch `AUCTION`, `INDIC`, or `CB`. Reach for `pm-calf-spy` or a manual `nc` session for those, as this chapter's exercises do.

26.18 Summary

You have now covered major CALF protocol usage patterns:

- provisioning configuration with pm-config-gen
- connecting external clients through example libraries, and through the read-only `pm-calf-spy` CLI
- handling channel/symbol subscriptions and snapshots across all seven channels
- distinguishing channels with a baseline `SNAP` (`TOP`, `STATE`, `INDEX`, `DEPTH`, `CB`) from those without one (`TRADE`, `AUCTION`, `INDIC`)
- reading auction uncross results (`AUCTION`, with `REASON`), indicative pricing during a call phase (`INDIC`), and circuit-breaker operational detail (`CB`, with `SRC` and Automated Corridor Expansion) alongside the simpler `STATE` halt/resume flag and its per-symbol session fan-out
- discovering per-symbol display precision via `REF`
- using control messages and liveness probes
- recovering with `RESUME`/LASTSEQ semantics, including de-duplicating a replay
- correlating each public print by its durable `TRADE_ID` across replay overlap
- diagnosing protocol errors operationally

26.19 Reflection

You have now used both RALF (Chapter 23, post-trade dissemination: CLEARING/DROP_COPY/AUDIT roles) and CALF (this chapter, market-data dissemination: TOP/TRADE/state channels). Before moving on, answer this synthesis question:

If you were building a downstream system, which protocol would you consume and why — a **risk system that needs to know every fill as it happens**, a **market-data terminal displaying live top-of-book**, and a **compliance archive that reconstructs the day's activity after the fact**? For each of those three consumers, state whether it needs RALF, CALF, or both, and justify your answer using what each protocol actually carries (trade/clearing events vs. price/quote state) rather than just their names.

26.20 Further Reading

- Market Data Feed (CALF)
- CALF Protocol Spy (pm-calf-spy)
- CALF Protocol Appendix
- Protocol Support Library Examples
- Processes, Environment and Ports

CHAPTER 27

API Gateway REST/WebSocket Training

27.1 Objective

Train on configuring and using `pm-api-gwy` for REST order entry, reference-data and history queries, admin operations, Swagger/OpenAPI exploration, WebSocket event handling, and multi-process logical separation.

You will practice:

- generating `api_gateways` config with bearer credentials
- starting a named API gateway process
- using REST endpoints with trading and read-only keys
- querying reference data and order/trade history over REST
- exercising admin REST endpoints with a resolved `ADMIN` role
- receiving private and market-data events over WebSocket
- splitting API gateway processes by ALF `gateway_id`

Background reading

- API Gateway - REST API Reference

27.2 Prerequisites

- Chapters 01-24 completed.
- Engine and stats commands available from the current environment.
- REST examples available in `docs/examples/REST`. On a container or `pipx` install that directory is inside the package: set `EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_` and read `docs/examples` as `$EX` in Exercises 6, 7 and 12. Exercise 7's C example is optional: it needs `make` and a C compiler, neither of which is installed in the container image. Exercise 6's Python client provides the equivalent API workflow.
- The `curl` commands below assume the lab config from Exercise 1 and are run from the directory holding `engine_config.yaml`.
- A terminal for `pm-engine`, one for `pm-stats`, one for `pm-api-gwy`, and one or more client terminals.
- Optional: `pm-audit` running, if you want to see a populated reply instead of `AUDIT_INDEX_UNAVAILABLE` from the audited order-lifecycle call in Exercise 10.

- This chapter is a hands-on tour of the normative REST and WebSocket API Reference — keep it open alongside this chapter for the full endpoint-by-endpoint contract (arguments, replies, and error codes) behind every call made here.

Recommended startup order:

1. Generate `engine_config.yaml`.
2. Start `pm-engine --verbose`.
3. Start `pm-stats`.
4. Start `pm-api-gwy --instance desk`.

27.3 Background

`pm-api-gwy` is an HTTP and WebSocket wrapper around the matching engine. It uses bearer tokens from `api_gateways.<NAME>.credentials`. A credential with a non-null `gateway_id` can submit and manage orders for that ALF gateway. A credential with `gateway_id: null` is read-only and can use the session, halts, reference, market-data, history, and public WebSocket paths.

Multiple API gateway processes are configured with top-level `api_gateways`:

```
api_gateways:
  desk:
    port: 8080
    credentials:
      - api_key: key-trader-demo
        gateway_id: TRADER01
  algos:
    port: 8081
    credentials:
      - api_key: key-algo-demo
        gateway_id: ALGO01
```

A non-null `gateway_id` may appear in only one `api_gateways` entry. This keeps private session and event state owned by one API gateway process. Read-only `gateway_id: null` credentials may appear in multiple entries.

27.3.1 Authentication model: API key → gateway_id → engine role

Every REST and WebSocket call carries one bearer token: `Authorization: Bearer <api_key>`. The gateway looks the key up in its own credential table and resolves it to a session holding just `api_key`, `gateway_id`, and `description` — nothing more. Three access levels follow directly from that one resolved `gateway_id`:

Resolved `gateway_id`	Access
`null`	Read-only: session, halts, reference data, history, public market data, `/healthz`
a configured ALF `gateway_id`	Trading: everything read-only gets, plus order/quote/combo/OCO submission and cancellation scoped to that one gateway
a configured ALF `gateway_id` whose **engine** role is `ADMIN`	Everything trading gets, plus every `/api/v1/admin/*` endpoint

The important detail is the third row: the API gateway does **not** store an "is admin" flag on the credential itself. It asks the *engine* what role the resolved `gateway_id` has, and only accepts `ADMIN` for admin endpoints. This is why `OPS01:ADMIN` in Exercise 1 configures the role on the **ALF gateway**, not on the API key — the key is only a pointer to that gateway identity.

See Appendix: REST API Reference — Auth and roles for the normative statement of this model, including how keys are provisioned.

27.4 Exercise 1: Generate a Single API Gateway Config

Generate a local lab config with trading keys and one read-only dashboard key. `--sessions-enabled` lets Exercise 10 drive the session, and `--cb-levels L1:0.05:30` gives every symbol a 5% circuit-breaker level with a 30-minute halt, which the admin halt in the same exercise needs (a symbol with no circuit breaker configured refuses it with `HALT_REJECTED`, `AAPL has no circuit breaker configured`):

```
pm-config-gen \
  --symbols AAPL MSFT \
  --participants TRADER01 TRADER02 OPS01:ADMIN \
  --outstanding-shares AAPL:15400000000 \
  --outstanding-shares MSFT:7430000000 \
  --sessions-enabled \
  --cb-levels L1:0.05:30 \
  --api-gateway \
  --api-gateway-name desk \
  --api-key key-trader-demo:TRADER01 \
  --api-key key-trader02-demo:TRADER02 \
  --api-key key-ops-demo:OPS01 \
  --api-key key-readonly-demo:null \
  --no-api-gateway-generate-keys \
  --api-gateway-host 127.0.0.1 \
  --api-gateway-port 8080 \
  --seed 20260624 \
  --output engine_config.yaml --force
```

Back up, check, deploy — then restart

This chapter starts from a fresh configuration (its operator is `OPS01`), and `--force` replaces your `engine_config.yaml`: save a copy first if you want to come back to it (`cp engine_config.yaml engine_config.before-ch25.yaml`). As always, nothing changes until you deploy it and restart the engine: `pm-cverifier engine_config.yaml`, then `pm-config-deploy engine_config.yaml`.
Inspect the generated section:

```
grep -A40 '^api_gateways:' engine_config.yaml
```

Expected behavior:

- top-level `api_gateways:` exists
- `desk:` has `port: 8080`
- credentials exist for `TRADER01`, `TRADER02`, `OPS01`, and one read-only key

For this training exercise, the command supplies fixed, easy-to-type demo keys. Define shell variables once in each terminal you use for `curl`:

```
KEY_TRADER=key-trader-demo
KEY_OPS=key-ops-demo          # OPS01's resolved engine role is ADMIN
KEY_RO=key-readonly-demo
```

You can also use the Config GUI's API subtab to generate keys for selected participants and choose the generated key length.

Exercise 10 uses `$KEY_OPS` for every admin REST call. Where a tool cannot expand shell variables (the WebSocket sessions in Exercise 9), enter the matching demo key shown above.

:material-checkbox-blank-outline: Checkpoint: the config contains the `key-trader-demo`, `key-ops-demo`, and `key-readonly-demo` credentials bound to the expected gateway identities.

27.5 Exercise 2: Start the API Gateway Process

Start the core processes with the generated config:

```
pm-engine --verbose
pm-stats
pm-api-gwy --instance desk
```

If the config contains only one `api_gateways` entry, `--instance` can be omitted. Use it anyway in labs so the selected process is explicit.

Sessions are enabled, so the engine starts in `CLOSED` and refuses orders until the day is opened. Open it from the operator console (`pm-admin --id OPS01`, then `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS`) before Exercise 4; Exercise 10 shows the REST equivalent.

Expected behavior:

- `pm-api-gwy` binds to `127.0.0.1:8080`
- startup fails if `api_gateways.desk.enabled` is `false`
- startup fails if the same non-null `gateway_id` is assigned to two named entries

:material-checkbox-blank-outline: Checkpoint: `GET /api/v1/status` is reachable once the process is running.

27.6 Exercise 3: Explore Swagger with a Bearer Key

Open Swagger in a browser:

```
http://127.0.0.1:8080/docs
```

Use the Authorize control and enter the trading key `key-trader-demo`. Then run:

```
curl -H "Authorization: Bearer $KEY_TRADER" \
  http://127.0.0.1:8080/api/v1/status
```

Try the same request with a read-only key, then try a write endpoint with the read-only key.

Expected behavior:

- status works for a trading key and answers `403 READ_ONLY` for a read-only key (it reports the caller's gateway, which a read-only key does not have)
- order entry rejects a read-only key
- Swagger lists the same route shapes exposed by `/openapi.json`

:material-checkbox-blank-outline: Checkpoint: you can authenticate in Swagger and explain which key is allowed to submit orders.

27.7 Exercise 4: Submit and Cancel an Order over REST

Submit a limit order with a trading key:

```
curl -X POST http://127.0.0.1:8080/api/v1/orders \
  -H "Authorization: Bearer $KEY_TRADER" \
  -H 'Content-Type: application/json' \
  -d '{
    "symbol": "AAPL",
    "side": "BUY",
    "order_type": "LIMIT",
    "quantity": 100,
    "price": 140.00,
    "tif": "DAY"
  }'
```

The price is well below AAPL's market on purpose, so the order rests instead of trading. Capture the returned `order_id`, then cancel it with a request tag:

```
curl -X DELETE 'http://127.0.0.1:8080/api/v1/orders/ORDER_ID?request_tag=rest-cxl-001' \
  -H "Authorization: Bearer $KEY_TRADER"
```

Expected behavior:

- the submit call returns an accepted response or an engine validation error
- a valid cancel request targets the authenticated credential's `gateway_id`
- `request_tag` is echoed on the cancel response/event and identifies this request, not the original order
- synchronous validation errors include both API `error.code` and canonical `error.reject_code`; when supplied, `client_tag` or `request_tag` is echoed in the error body
- read-only keys cannot submit or cancel orders

:material-checkbox-blank-outline: Checkpoint: you can submit an order and observe its cancel path or explain the engine-side validation error.

27.8 Exercise 5: Query Reference Data

Reference-data endpoints expose compiled, mostly-static configuration: tick sizes, risk bands, index definitions, and the session schedule. They change only when an admin reloads the reference bundle (Exercise 10) — never per trade or per order.

Fetch the full bundle in one call:

```
curl -H "Authorization: Bearer $KEY_RO" \
  http://127.0.0.1:8080/api/v1/reference
```

Then fetch the individual sections used most often:

```
curl -H "Authorization: Bearer $KEY_RO" \
  http://127.0.0.1:8080/api/v1/reference/symbols

curl -H "Authorization: Bearer $KEY_RO" \
  http://127.0.0.1:8080/api/v1/reference/risk

curl -H "Authorization: Bearer $KEY_RO" \
  http://127.0.0.1:8080/api/v1/reference/schedule

curl -H "Authorization: Bearer $KEY_RO" \
  http://127.0.0.1:8080/api/v1/reference/config-version
```

Every reference reply carries `config_version` — a content hash of the compiled bundle. Compare the value returned by `/reference/config-version` against the one embedded in `/reference/symbols`; they must match. A client can poll this one small value instead of every field to notice a config change.

Expected behavior:

- a read-only key can call every `/reference/*` endpoint
- `tick_decimals` for `AAPL`/`MSFT` in `/reference/symbols` matches what you passed to `pm-config-gen`
- `config_version` is identical across every `/reference/*` reply taken at the same point in time

:material-checkbox-blank-outline: Checkpoint: you can explain the difference between `GET /api/v1/reference` (one round-trip) and the per-section endpoints, and what `config_version` is for.

27.9 Exercise 6: Use the Python REST Example

From the REST example directory, run the Python client or adapt it with the key and port from your config:

```
cd docs/examples/REST/python        # or: cd $EX/REST/python
EDUMATCHER_API_URL=http://127.0.0.1:8080 \
EDUMATCHER_API_KEY=$KEY_TRADER \
python3 demo_info.py
```

Use the example source as a reference for adding status, order-entry, and cancel calls to a test harness.

:material-checkbox-blank-outline: Checkpoint: the Python client reaches the gateway and sends authenticated requests.

27.10 Optional Exercise 7: Use the C REST Example

Build and run the C example from the REST example directory:

```
cd docs/examples/REST/c             # or: cd $EX/REST/c
make
EDUMATCHER_API_KEY=$KEY_TRADER ./demo_info
```

Expected behavior:

- the client sends an Authorization header
- status or order-entry responses are printed as JSON
- connection errors identify an unavailable gateway process or wrong port

:material-checkbox-blank-outline: Optional checkpoint: the C example can call the running API gateway

with a configured key.

27.11 Exercise 8: Query Order and Trade History

History endpoints read from `pm-stats`'s database, so `pm-stats` must be running and must have been running while your orders and trades happened — it cannot reconstruct events from before it started.

Submit and fill an order (repeat Exercise 4 with a marketable price, or cross `TRADER01` against `TRADER02`), then query its lifecycle with a trading key:

```
curl -H "Authorization: Bearer $KEY_TRADER" \
  "http://127.0.0.1:8080/api/v1/history/orders?symbol=AAPL&limit=10"

curl -H "Authorization: Bearer $KEY_TRADER" \
  http://127.0.0.1:8080/api/v1/history/orders/ORDER_ID

curl -H "Authorization: Bearer $KEY_TRADER" \
  "http://127.0.0.1:8080/api/v1/history/fills?symbol=AAPL"
```

The public trade tape and daily OHLCV accept any authenticated key, not only a trading key:

```
curl -H "Authorization: Bearer $KEY_RO" \
  "http://127.0.0.1:8080/api/v1/history/trades?symbol=AAPL&limit=5"

curl -H "Authorization: Bearer $KEY_RO" \
  "http://127.0.0.1:8080/api/v1/history/daily?symbol=AAPL"
```

Walk a second page using the cursor from the first reply:

```
curl -H "Authorization: Bearer $KEY_RO" \
  "http://127.0.0.1:8080/api/v1/history/trades?symbol=AAPL&limit=5&after=NEXT_CURSOR"
```

Every history call returns rows **oldest first**, a page at a time. On a data directory that has seen earlier chapters the first page is therefore old traffic, not your new order, and `/history/daily` totals include everything `pm-stats` has recorded today. Look an order up by its id (`/history/orders/ORDER_ID`), narrow the range with `from=<ISO timestamp>` (or `date=YYYY-MM-DD`), or follow `next_cursor` to reach the recent rows.

Expected behavior:

- `/history/orders` and `/history/fills` are scoped to the caller's own gateway — `TRADER01`'s key never sees `TRADER02`'s orders
- `/history/trades` and `/history/daily` are public and show both sides
- every page carries `next_cursor`: a string while `"has_more": true` (pass it back unchanged as `after` for the next page) and `null` on the last page
- if `pm-stats` is not running, every history call returns `503 STATS_DB`

:material-checkbox-blank-outline: Checkpoint: you can explain why `/history/orders` is gateway-scoped but `/history/trades` is not.

27.12 Exercise 9: Observe WebSocket Events

Connect to private events before submitting new orders:

```
python3 -m websockets ws://127.0.0.1:8080/api/v1/ws/private   # needs `pip install websockets`
```

Once connected, the CLI waits for you to type. Send the authentication message as your first input, then press Enter:

```
> {"api_key": "key-trader-demo"}
< {"type": "authenticated", "gateway_id": "TRADER01", "stream_seq": 10}
< {"type": "orders.snapshot", "gateway_id": "TRADER01", ...}
```

(`stream_seq` is the gateway's private-stream counter, so the number differs; the `orders.snapshot` that follows lists your current orders and positions before any live event.)

In another terminal, submit or cancel an order with the same key. Observe the private event stream.

Then connect to a public market-data WebSocket with a read-only key:

```
python3 -m websockets ws://127.0.0.1:8080/api/v1/ws/market-data
```

After authentication, send a subscription control message:

```
> {"api_key": "key-readonly-demo"}
< {"type": "authenticated"}
> {"action": "subscribe", "symbols": ["AAPL"], "channels": ["book", "trades"]}
< {"type": "subscription", "data": {"items": [...], "always": ["session", "circuit_breaker", "news"], "rejected":
```

Each rule can have its own symbols, so one socket can carry an overview plus a focused instrument:

```
> {"action": "subscribe", "items": [
    {"symbols": ["*"],    "channels": ["book", "trades"]},
    {"symbols": ["AAPL"], "channels": ["depth"]}
  ]}
```

The book is also available without a socket. Compare it with what the subscription just sent:

```
curl -s -H "Authorization: Bearer $KEY_RO" \
  "http://127.0.0.1:8080/api/v1/book/AAPL?levels=3"
curl -s -H "Authorization: Bearer $KEY_RO" http://127.0.0.1:8080/api/v1/book
```

Each price comes as a display price and as `*_ticks`. The reply's `seq` is the `seq` of the `book` frame for the same snapshot, so a client that reads REST first and then subscribes can drop frames at or below it.

Expected behavior:

- `GET /book/AAPL` matches the `book` frame the subscription sent, with the same `seq` when the book has not changed since
- `GET /book/NOSUCH` answers `404 UNKNOWN_SYMBOL`
- private events correspond to the authenticated trading gateway
- public market-data access does not require a non-null `gateway_id`
- stale or unknown bearer keys are rejected
- every event carries `topic` and a per-topic `seq`; a jump in `seq` for a topic means your client read too slowly and events were dropped
- `session`, `circuit_breaker` and `news` arrive whether or not you subscribed — the ack lists them under `always`
- an `order.fill` event's `data.liquidity_flag` tells you whether *your* order supplied liquidity (`MAKER`, it was resting) or removed it (`TAKER`, it crossed the spread) — no join against another feed required

:material-checkbox-blank-outline: Checkpoint: you can explain when to use REST responses versus WebSocket events for order outcomes.

27.13 Exercise 10: Admin REST — Session Control, Risk, and Cross-Gateway Orders

Confirm the resolved role first, with `OPS01`'s key from Exercise 1:

```
curl -H "Authorization: Bearer $KEY_OPS" \
  http://127.0.0.1:8080/api/v1/status
```

`gateway_role` must read `ADMIN`; the reply also includes `gateway_count`, a field only ADMIN callers get back. Now try the same idea with a trading key:

```
curl -i -H "Authorization: Bearer $KEY_TRADER" \
  http://127.0.0.1:8080/api/v1/admin/gateways
```

Expect `403 ROLE_DENIED` — a trading key is never accepted on `/admin/*`, regardless of which gateway it is bound to.

Trigger and resume a circuit-breaker halt with the ADMIN key. The symbol is part of the path; the body carries only the optional `level` (and `reason`):

```
curl -X POST http://127.0.0.1:8080/api/v1/admin/symbols/AAPL/halt \
  -H "Authorization: Bearer $KEY_OPS" \
  -H 'Content-Type: application/json' \
  -d '{"level": "L1"}'

curl -H "Authorization: Bearer $KEY_OPS" \
  http://127.0.0.1:8080/api/v1/halts

curl -X POST http://127.0.0.1:8080/api/v1/admin/symbols/AAPL/resume \
  -H "Authorization: Bearer $KEY_OPS"
```

Request a session transition. This call waits for the engine's verdict instead of firing and forgetting. If you have not opened the day yet, send `PRE_OPEN` first; the engine only accepts the legal next phase:

```
curl -i -X POST http://127.0.0.1:8080/api/v1/admin/session/transition \
  -H "Authorization: Bearer $KEY_OPS" \
  -H 'Content-Type: application/json' \
  -d '{"to_state": "CONTINUOUS"}'
```

A successful transition returns `200` with `status: "APPLIED"` and a `command_id`. Requesting a state the engine cannot apply because sessions are disabled returns `409 TRANSITION_REJECTED` with the engine's reason as `message`. A transition that is simply not legal *from the current phase* (for example `CONTINUOUS` when the market is already `CONTINUOUS`) gets no answer from the engine at all: the call waits about three seconds and ends with `503 ENGINE_TIMEOUT`. Check `GET /api/v1/session` first, as for the console.

Look at cross-gateway order visibility:

```
curl -H "Authorization: Bearer $KEY_OPS" \
  "http://127.0.0.1:8080/api/v1/admin/orders?symbol=AAPL"
```

This is the only endpoint that shows every gateway's resting orders in one call; a trading key never sees this. Note the `retention_sec` field in the reply — filled/cancelled/expired orders are evicted from this view after `order_retention_sec` (default one hour, configurable per `api_gateways` instance), while resting orders are never evicted regardless of age.

Fetch one order's full cross-gateway lifecycle from the audit trail:

```
curl -i -H "Authorization: Bearer $KEY_OPS" \
  http://127.0.0.1:8080/api/v1/admin/orders/ORDER_ID/audit
```

This endpoint is optional: it depends on a separate `pm-audit` index that this gateway only *reads*, never writes. If `pm-audit` has not built one, expect `503 AUDIT_INDEX_UNAVAILABLE` — every other admin endpoint keeps working normally.

Finally, cancel resting exposure with the kill-switch family:

```
curl -X POST http://127.0.0.1:8080/api/v1/admin/symbols/AAPL/cancel-orders \
  -H "Authorization: Bearer $KEY_OPS" \
  -H 'Content-Type: application/json' \
  -d '{"reason": "training exercise"}'

curl -X POST http://127.0.0.1:8080/api/v1/admin/gateways/TRADER01/kill-switch \
  -H "Authorization: Bearer $KEY_OPS" \
  -H 'Content-Type: application/json' \
  -d '{"reason": "training exercise"}'
```

Expected behavior:

- a trading key gets `403 ROLE_DENIED` on every `/admin/*` path
- halting a symbol the engine does not know returns `409 HALT_REJECTED` with `message: "Unknown symbol: ..."` — the engine refused, your role is fine
- `/admin/session/transition` blocks for the engine's verdict (applied, or `409` when sessions are disabled; an illegal next phase ends in `503 ENGINE_TIMEOUT`)
- `/admin/orders` returns orders from every gateway, not just `OPS01`'s
- `/admin/orders/{order_id}/audit` either returns the full lifecycle or a clear `503 AUDIT_INDEX_UNAVAILABLE` if no audit index is deployed

:material-checkbox-blank-outline: Checkpoint: you can explain why role resolution happens against the engine rather than being a property of the API key itself.

27.14 Exercise 11: Configure Multiple Logical API Gateways

Generate two API gateway process configs, one for desk trading and one for algorithmic trading:

```
pm-config-gen \
  --symbols AAPL MSFT \
  --participants TRADER01 ALGO01 OPS01:ADMIN \
  --api-gateway-instance desk:TRADER01:8080 \
  --api-gateway-instance algos:ALGO01:8081 \
  --seed 20260624 \
  --output engine_config.yaml --force
```

Start each named process in a separate terminal:

```
pm-api-gwy --instance desk
pm-api-gwy --instance algos
```

Try an invalid duplicate assignment:

```
pm-config-gen \
  --symbols AAPL \
  --participants TRADER01 \
  --api-gateway-instance desk:TRADER01 \
  --api-gateway-instance algos:TRADER01 \
  --dry-run
```

Expected behavior:

- `desk` listens on `8080` and owns `TRADER01`
- `algos` listens on `8081` and owns `ALGO01`
- duplicate non-null `gateway_id` assignment is rejected before runtime

:material-checkbox-blank-outline: Checkpoint: you can run two API gateway processes and explain why a write-capable gateway ID is globally unique across them.

27.15 Exercise 12: Write a Python CLI Client for LIMIT Order Entry

Use the `ApiGatewayClient` library from `docs/examples/REST/python` to write a small script that submits a LIMIT order and prints the engine response.

```
import argparse, json, os, sys
sys.path.insert(0, os.path.join(os.environ.get("EX", "docs/examples"), "REST/python"))
from api_gateway_client import ApiGatewayClient

parser = argparse.ArgumentParser()
parser.add_argument("--side",   required=True, choices=["BUY", "SELL"])
parser.add_argument("--symbol", required=True)
parser.add_argument("--qty",    required=True, type=int)
parser.add_argument("--price",  required=True, type=float)
parser.add_argument("--wait-ack", action="store_true")
args = parser.parse_args()

client = ApiGatewayClient(
    os.environ.get("EDUMATCHER_API_URL", "http://127.0.0.1:8080"),
    os.environ.get("EDUMATCHER_API_KEY", "key-trader-demo"),
)
path = "/api/v1/orders" + ("?wait=ack" if args.wait_ack else "")
result = client.post_json(path, {
    "symbol": args.symbol.upper(),
    "side":   args.side,
    "order_type": "LIMIT",
    "quantity": args.qty,
    "price":  args.price,
})
print(json.dumps(result, indent=2))
```

Save it as `limit_order.py`, set `EDUMATCHER_API_KEY` to a trading key (`export EDUMATCHER_API_KEY=$KEY_TRADER`; the built-in default is only a placeholder) and run it from the repo root (or from anywhere with `EX` set as above):

```
python3 limit_order.py --side BUY --symbol AAPL --qty 100 --price 140.00
python3 limit_order.py --side BUY --symbol AAPL --qty 100 --price 140.00 --wait-ack
```

Expected behavior:

- without `--wait-ack`: `status` is `PENDING` and `event` is null
- with `--wait-ack`: `status` is `ACKED` and `event` contains the engine response
- sending a read-only key returns `403 READ_ONLY`
- omitting `--price` is stopped by the script's own argument parser (`error: the following arguments are required: --price`) before any request is sent; to see the gateway's `400 VALIDATION` for a `LIMIT` order without `price`, post the JSON body with `curl` and leave `price` out

A fully documented `MARKET` order version that follows the same pattern is available at `docs/examples/REST/python/`

:material-checkbox-blank-outline: Checkpoint: your script prints `order_id` and `status` from a live gateway.

27.16 Support Libraries and Example Clients

Reference examples used in this training chapter:

- `docs/examples/REST/python`
- `docs/examples/REST/c`

Use these examples as small integration clients when building course labs, smoke tests, or external adapter prototypes.

27.17 Summary

You can now:

- Generate one or more `api_gateways` process configs with trading, read-only, and ADMIN-resolved bearer credentials.
- Start and reach `pm-api-gwy` over REST, Swagger, and WebSocket.
- Submit, cancel, and observe orders through REST and private WebSocket events.
- Query the reference-data, history, and admin REST surfaces, and explain what each returns and who is allowed to call it.
- Explain why a write-capable ALF `gateway_id` must be unique across every configured `api_gateways` process, and why ADMIN access is resolved from the engine rather than stored on the API key.

See REST and WebSocket API Reference for the full normative contract behind every endpoint used in this chapter.

27.18 Reflection

If two API gateway processes both listed the same `gateway_id` as a read-only (`gateway_id: null`) credential, would that be a problem? Why does the constraint only apply to non-null `gateway_id` values?

`GET /api/v1/status` returns a different `gateway_role` for the same running engine depending on which key you authenticate with. Where does that role actually come from, and what would go wrong if the API gateway instead trusted a role embedded directly in the key?

27.19 Handoff for Chapter 26

Before starting 26 — Market Index (pm-index), you can stop `pm-api-gwy`; it is not required there. Keep `pm-engine` and `pm-stats` running if you want to reuse the same session, or start fresh — Chapter 26 generates its own config with `pm-config-gen`.

CHAPTER 28

Market Index (pm-index)

28.1 Objective

Configure and operate the pm-index calculation process, observe a cap-weighted index updating in real time, apply corporate actions without disrupting the index level using pm-index-admin-cli, and analyse historical data with pm-index-cli and pm-stats-cli.

You will practice:

- generating an index config with pm-config-gen
- starting pm-index and verifying initialisation
- querying the live index level through the INDEX gateway command
- watching the index move as trades execute
- applying a stock split and a cash dividend corporate action with pm-index-admin-cli
- adding and removing index constituents without restarting, with pm-index-admin-cli
- querying structural/audit history with pm-index-cli
- querying level and EOD time-series history with pm-stats-cli
- exporting history to CSV and JSON for offline analysis

Background reading

- Market Index - Index Administration (pm-index-admin-cli)

28.2 Prerequisites

- Chapters 01–03 completed (engine running, at least one gateway, a few trades).
- Every constituent symbol must have outstanding_shares set in symbols:.
- To follow Exercise 9 (level/EOD queries via pm-stats-cli), pm-stats must also be running (see Chapter 15) — it needs to be up *before* the trades happen so it can capture the index.update ticks live.

Recommended startup order for this chapter:

1. Generate engine_config.yaml (Exercise 1).
2. Start pm-engine.
3. Start pm-index.

4. Start `pm-stats` (needed later for Exercise 9's level/EOD history).
5. Connect two gateway terminals (one TRADER, one ADMIN).

28.3 Background

`pm-index` is a **standalone** process — it never sends commands to the engine. It subscribes to trade events published by `pm-engine` on port 5556 and recomputes configured indices in real time.

28.3.1 Cap-weighted formula

Each index level is:

$$\text{level} = \frac{\sum_{\text{sym}} \text{last_price(sym)} \times \text{outstanding_shares(sym)}}{\text{divisor}}$$

The divisor is chosen at first launch so the level equals `base_value`. It changes only when a corporate action alters share structures. This keeps the index level **continuous** across structural events.

28.3.2 Key processes and ports

```
pm-engine  PUB :5556  ──► pm-index  PUB :5558  ──► pm-md-gwy  ──► CALF clients
                                    PULL :5559 ◄── ADMIN gateways
```

ADMIN gateways send corporate-action and history-request commands to `pm-index` on the PULL socket (port 5559).

Two different history stores

`pm-index` writes a JSONL file per index, but it only records **structural/audit events** — `INIT`, `CORP_ACTION`, `ADD_CONSTITUENT`, `DELIST`. It does **not** write intraday level ticks or end-of-day OHLC records to that file. Those are captured instead by `pm-stats`, which subscribes to the live `index.update` broadcast and records every tick (plus the EOD-forced publish) into its own SQLite database. This means `pm-stats` must be **running** while trades occur for level/EOD history to exist later — unlike the JSONL file, which `pm-index-cli` can read at any time, even offline.

28.4 Exercise 1: Configure an Index

Generate a complete config with one index, three constituents, and outstanding shares. `pm-index` also needs a reference price for every constituent, and `pm-config-gen` only writes one by seeding it from a market maker's midpoint: hence the `MM01:MARKET_MAKER` participant and the `--seed-mm-mid-range` / `--seed-last-prices-from-mm` flags. `--no-mm-seed-quotes` keeps the book empty at startup.

```
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --participants TRADER01 OPS01:ADMIN MM01:MARKET_MAKER \
  --outstanding-shares AAPL:15000000000 \
  --outstanding-shares MSFT:7400000000 \
  --outstanding-shares TSLA:3200000000 \
  --seed-mm-mid-range 100:400 \
  --seed-last-prices-from-mm \
  --no-mm-seed-quotes \
  --sessions-enabled \
  --seed 20260625 \
  --index EDU100:"EduMatcher broad index" \
  --index-constituents EDU100:AAPL,MSFT,TSLA \
  --output engine_config.yaml --force
```

Back up, check, deploy — then restart

This chapter starts from a fresh configuration (its operator is `OPS01`), and `--force` replaces your `engine_config.yaml`: save a copy first if you want to come back to it (`cp engine_config.yaml engine_config.before-ch26.yaml`). As always, nothing changes until you deploy it and restart the engine: `pm-cverifier engine_config.yaml`, then `pm-config-deploy engine_config.yaml`. Sessions are enabled, so the restarted engine starts in `CLOSED`: open the market from the operator console (`pm-admin --id OPS01`, then `SESSION|STATE=PRE_OPEN` and `SESSION|STATE=CONTINUOUS`).
Open `engine_config.yaml` and locate the `indices:` block. It should look like:

```
indices:
  - id: EDU100
    description: "EduMatcher broad index"
    base_value: 1000.0
    publish_interval_sec: 1.0
    history_file: data/indexes/EDU100_history.jsonl
    state_file: data/indexes/EDU100_state.json
    constituents:
      - AAPL
      - MSFT
      - TSLA
```

Outstanding shares

Every constituent must have `outstanding_shares` set in `symbols:`. The default `pm-config-gen` behaviour adds this from `--outstanding-shares`. Without it `pm-index` will refuse to start.
:material-checkbox-blank-outline: **Checkpoint:** `engine_config.yaml` contains a valid `indices:` block with all three symbols.

28.5 Exercise 2: Start `pm-index`

In a dedicated terminal, start the index calculation process:

```
pm-index
```

Expected startup output:

`pm-index` logs at `WARNING` by default, so it starts silently. Run it with `-v` to see the startup messages:

```
pm-index -v
```

```
starting pm-index with log level INFO
loaded 1 index runtime config(s)
index ready index_id=EDU100 constituents=3 level=1000.0
```

(Each line is preceded by the usual timestamp, level and logger name.)

The INIT record has been written to $EDUMATCHER_DATA_DIR/indexes/EDU100_history.jsonl (the config's relative data/indexes/... path resolves under the data directory). Verify:

```
cat "$EDUMATCHER_DATA_DIR/indexes/EDU100_history.jsonl"
```

```
{"type": "INIT", "ts_ns": ..., "index_id": "EDU100", "base_value": 1000.0, "divisor": 7007100000.0, "constituents
```

:material-checkbox-blank-outline: **Checkpoint:** pm-index running, INIT record visible in history file.

28.6 Exercise 3: Query the Live Index Level

Connect your TRADER01 gateway and run the INDEX command:

```
[TRADER01]> INDEX
```

pm-index publishes a level when a constituent trades (and once at the end of the day), not on a timer. Until the first trade, INDEX answers No index data received yet. Is pm-index running? — that does not mean pm-index is down. Come back to this exercise's output after Exercise 4's first trade. Expected response (the level will not be 1000: it is computed from the price that traded and the *seeded* reference prices of the other two constituents):

```
[10:00:01.042] EDU100 637.57 +0.00 +0.00% O=637.57 H=637.57 L=637.57 CONTINUOUS
```

Fields shown:

Field	Description
Timestamp	UTC time of the reading
Index ID	EDU100
Level	Current calculated level
Change / %	Change from day open (empty before first open)
O / H / L	Day open, high, low (empty before continuous trading begins)
Session state	Current engine session state

The INDEX command works in pm-alf-console under any authenticated gateway role (TRADER, ADMIN, or read-only). The separate pm-admin operator console has no INDEX command (Unknown command 'INDEX'); use pm-alf-console --id OPS01 when you want it as the operator. It prints only the **most recent** index update this console has received, as a single line, so it is a spot check, not a list of constituents or of indices.

:material-checkbox-blank-outline: **Checkpoint:** INDEX command returns a level for EDU100.

28.7 Exercise 4: Watch the Index Move

Move the session into CONTINUOUS (if the scheduler hasn't already done so — from CLOSED that is PRE_OPEN first, then CONTINUOUS). The books start empty (--no-mm-seed-quotes), so connect pm-alf-console --id MM01 and give each symbol a market, with prices near the seeded ones (print the seeded prices with grep -E "last_(buy|sell)_price" engine_config.yaml), for example QUOTE|SYM=AAPL|BID=<p-1>|ASK=<p+1>|BID_QTY=1000| and the same for MSFT and TSLA. Then execute a few trades:

```
[OPS01|ADMIN]> SESSION|STATE=PRE_OPEN
[OPS01|ADMIN]> SESSION|STATE=CONTINUOUS
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=MARKET|QTY=500
[TRADER01]> NEW|SYM=MSFT|SIDE=BUY|TYPE=MARKET|QTY=300
[TRADER01]> NEW|SYM=TSLA|SIDE=SELL|TYPE=MARKET|QTY=200
```

After each fill, query the index again:

```
[TRADER01]> INDEX
```

Observe the level changing as trades update the constituent prices. The index is recalculated on every trade but published at most once per second (the publish_interval_sec throttle).

Tip

Run INDEX several times in quick succession to see the throttle in action — the level updates, but the published timestamp advances at most once per second.
:material-checkbox-blank-outline: **Checkpoint:** index level changed after trades; querying INDEX shows the new value.

28.8 Exercise 5: Apply a Stock Split

Suppose AAPL announces a 2-for-1 stock split. Without adjustment, the index would drop by ~50% when AAPL's price halves — which is wrong because no value was destroyed.

No gateway command for corporate actions

Unlike INDEX|HISTORY (Exercise 11), corporate actions and constituent changes have **no** pm-alf-console command, no pm-admin/pm-admin-cli subcommand, and no ALF/CALF wire message — there is no CORP_ACTION|... line you can type at a gateway prompt. Instead, use the dedicated pm-index-admin-cli tool, which talks directly to pm-index's PULL socket (port 5559). See Market Index → Applying corporate actions.
Apply the split:

```
pm-index-admin-cli --id OPS01 split \
  --index EDU100 --sym AAPL --ratio 2:1
```

You'll be shown a confirmation prompt before the command is sent. Add -y to skip it, or --dry-run to preview the payload without sending — both are options of the tool itself, so they go **before** the sub-command (pm-index-admin-cli --id OPS01 -y split …). The command blocks for the index.corp_action_ack.OPS01 response and prints the result. pm-index applies the action in-process, publishes an updated index value live, and writes a CORP_ACTION record to the structural/audit history file.

Query the index immediately after, from any gateway terminal:

```
[TRADER01]> INDEX
```

The level should be **unchanged** (or differ by only rounding). Now check that the divisor was adjusted in the history file:

```
grep CORP_ACTION "$EDUMATCHER_DATA_DIR/indexes/EDU100_history.jsonl" | python3 -m json.tool
```

You should see `old_divisor` and `new_divisor` in the record, and the two values should be **approximately equal** (differing only by integer-rounding of the new share count). This is expected: a split doubles AAPL's outstanding shares *and* halves AAPL's price, so AAPL's contribution to the aggregate cap (`shares × price`) is unchanged — no divisor adjustment is needed to keep the index level continuous. Contrast this with `ADD`/`DELIST` (Exercise 7), where the aggregate cap genuinely changes and the divisor must move to compensate.

Apply during PRE_OPEN

Applying a corporate action mid-session means one more trade at the old price may be processed before the divisor update. Best practice is to apply all corporate actions during `PRE_OPEN`.
:material-checkbox-blank-outline: **Checkpoint:** compare the `level` field of the `CORP_ACTION` record to the `INDEX` reading you took just before applying the split — the absolute difference should be at most a few cents (rounding only), never a ~50% jump. `CORP_ACTION` record confirmed written to history.

28.9 Exercise 6: Apply a Cash Dividend

Apply a $2.50 cash dividend for MSFT — same `pm-index-admin-cli` mechanism as Exercise 5, no gateway command:

```
[TRADER01]> INDEX        (before)
```

```
pm-index-admin-cli --id OPS01 dividend \
  --index EDU100 --sym MSFT --amount 2.50
```

```
[TRADER01]> INDEX        (after)
```

A cash dividend reduces the effective price by the dividend amount. The divisor is adjusted to compensate so the index level is preserved.

:material-checkbox-blank-outline: **Checkpoint:** index level preserved across dividend adjustment.

Under the hood: ExchangeCommandClient

`pm-index-admin-cli` is a thin wrapper over the same `ExchangeCommandClient` class `pm-admin-cli` uses internally. If you ever need to script a corporate action directly (e.g. from a test harness), the equivalent code is:

```
from edumatcher.commands import ExchangeCommandClient

client = ExchangeCommandClient("OPS01")
result = client.index_corp_action(
    "EDU100", "CASH_DIVIDEND", "MSFT",
    dividend_per_share=2.50,
)
print("CORP_ACTION result:", result)
client.close()
```

Note `pm-index`'s PULL socket has no `connect()`/auth handshake, unlike the engine socket `pm-admin-cli`

talks to.

28.10 Exercise 7: Add and Remove a Constituent

Constituent changes also use `pm-index-admin-cli` — no gateway command exists for these either.

28.10.1 Add AMZN to the index

Adding a constituent adjusts the divisor so the level does not jump at the moment of addition. You must supply the new shares and a reference price:

```
pm-index-admin-cli --id OPS01 add \
  --index EDU100 --sym AMZN \
  --shares 10500000000 --price 195.00
```

`INDEX` prints only the level, not the constituents, so check the change in the history instead (the `ADD OK` reply also prints the new divisor):

```
pm-index-cli --config engine_config.yaml events --index EDU100 --type ADD_CONSTITUENT
```

AMZN is not traded on this venue, so its price stays at the reference price you gave until an engine feed for it exists; it contributes to the cap at `195.00`. (`delist` needs no price.)

28.10.2 Remove TSLA from the index

```
pm-index-admin-cli --id OPS01 delist --index EDU100 --sym TSLA
```

Confirm it with `pm-index-cli --config engine_config.yaml events --index EDU100 --type DELIST`, and run `INDEX` again: the level should be the same as before the change.

:material-checkbox-blank-outline: **Checkpoint:** AMZN added and TSLA removed without visible discontinuity in the index level.

28.11 Exercise 8: Query History with `pm-index-cli` and `pm-stats-cli`

Two tools cover two different slices of index history:

- `pm-index-cli` reads the JSONL structural/audit file directly from disk — `pm-index` does **not** need to be running. It only knows about `INIT`, `CORP_ACTION`, `ADD_CONSTITUENT`, and `DELIST` records.
- `pm-stats-cli` reads level ticks and end-of-day OHLC rollups from `pm-stats`' SQLite database. Those rows only exist if `pm-stats` was **running** while the ticks happened — it captures them live from the `index.update` broadcast, it does not replay history after the fact.

If you started `pm-stats` back in the Prerequisites step, it has been recording every `EDU100` tick since. If you skipped that step, go start it now — `pm-stats-cli` will simply return no rows for anything that happened before it was running.

28.11.1 List configured indices

```
pm-index-cli --config engine_config.yaml indices
```

28.11.2 View recent intraday level snapshots

Every recorded tick (throttled to `publish_interval_sec`, same as the live `INDEX` command) is available via `pm-stats-cli index-snapshots`:

```
pm-stats-cli index-snapshots --index-id EDU100 --limit 50
```

Sample table output:

```
ts                  | index_id | level   | aggregate_cap       | divisor      | session_state | day_open | day_hi
--------------------+----------+---------+---------------------+--------------+---------------+----------+------
2026-06-25T10:00:01 | EDU100   | 1000.00 | 7007100000000.00    | 7007100000.0 | PRE_OPEN      |          |
2026-06-25T10:01:23 | EDU100   | 1034.82 | 7251000000000.00    | 7007100000.0 | CONTINUOUS    | 1000.00  | 1034.
...
```

You can list which index IDs `pm-stats` has actually recorded with:

```
pm-stats-cli index-ids
```

28.11.3 View EOD records

`pm-index` no longer writes an end-of-day record to its own JSONL file — that file is structural/audit only now. Instead, `pm-stats` treats the EOD-forced publish like any other tick and rolls the day's ticks up into one daily OHLC row. Query it with `pm-stats-cli index-daily`:

```
pm-stats-cli index-daily --index-id EDU100
```

```
date       | index_id | open_level | high_level | low_level | close_level | update_count
-----------+----------+------------+------------+-----------+-------------+-------------
2026-06-25 | EDU100   | 1000.00    | 1052.10    | 987.40    | 1041.55     | 312
```

Add `--wide` to also show the opening/closing aggregate market cap:

```
pm-stats-cli index-daily --index-id EDU100 --wide
```

28.11.4 View all structural events

```
pm-index-cli --config engine_config.yaml events --index EDU100
```

This shows `INIT`, `CORP_ACTION`, `ADD_CONSTITUENT`, and `DELIST` records. You should see the split, dividend, add, and delist from earlier exercises.

Filter to only corporate actions:

```
pm-index-cli --config engine_config.yaml events --index EDU100 --type CORP_ACTION
```

:material-checkbox-blank-outline: **Checkpoint:** `pm-index-cli` returns structural events; `pm-stats-cli index-sna` and `index-daily` return level and EOD rows for `EDU100`.

28.12 Exercise 9: Export and Analyse Data

28.12.1 Export EOD data to CSV

```
pm-stats-cli --format csv index-daily --index-id EDU100 > edu100_eod.csv
cat edu100_eod.csv
```

28.12.2 Export intraday data to JSON

```
pm-stats-cli --format json index-snapshots --index-id EDU100 \
  | python3 -c "
import json, sys
rows = json.load(sys.stdin)
print(f'{len(rows)} snapshots')
if rows:
    print(f'First: {rows[0][\"ts\"]}  level={rows[0][\"level\"]}')
    print(f'Last:  {rows[-1][\"ts\"]}  level={rows[-1][\"level\"]}')
"
```

28.12.3 Date-range query

For a single trading day, `--date` is the simplest filter:

```
pm-stats-cli index-snapshots --index-id EDU100 --date 2026-06-25 --limit 500
```

For an arbitrary time window, use `--from`/`--to` with full ISO-8601 timestamps instead:

```
pm-stats-cli index-snapshots --index-id EDU100 \
  --from 2026-06-25T09:30:00+00:00 \
  --to 2026-06-25T16:00:00+00:00 \
  --limit 500
```

:material-checkbox-blank-outline: **Checkpoint:** CSV and JSON exports contain the expected data.

28.13 Exercise 10: Two Indices

Configure a second, narrower index alongside `EDU100`:

```
pm-config-gen \
  --symbols AAPL MSFT TSLA \
  --participants TRADER01 OPS01:ADMIN MM01:MARKET_MAKER \
  --outstanding-shares AAPL:15000000000 \
  --outstanding-shares MSFT:7400000000 \
  --outstanding-shares TSLA:3200000000 \
  --seed-mm-mid-range 100:400 \
  --seed-last-prices-from-mm \
  --no-mm-seed-quotes \
  --sessions-enabled \
  --seed 20260625 \
  --index EDU100:"EduMatcher broad index" \
  --index-constituents EDU100:AAPL,MSFT,TSLA \
  --index TECH2:"Technology pair" \
  --index-constituents TECH2:AAPL,MSFT \
  --index-base-value TECH2:500.0 \
  --index-interval TECH2:2.0 \
  --output engine_config.yaml --force
```

Deploy the new configuration (`pm-index` reads the compiled artifact, so without this it still loads one index), then restart `pm-index` with `--reset` to clear state and re-initialise. Stop the running `pm-index` first:

```
pm-cverifier engine_config.yaml
pm-config-deploy engine_config.yaml
pm-index --reset
```

-reset discards divisors

`--reset` removes all state files. The divisors for both indices are recomputed from scratch using current reference prices. Use only when you intentionally want a clean slate.
Both indices start at their base value (`pm-index -v` prints `index ready index_id=EDU100 … level=1000.0` and `index ready index_id=TECH2 … level=500.0`), but nothing is published until a constituent trades. Make a trade in AAPL or MSFT, then run `INDEX` in the TRADER terminal:
[TRADER01]> INDEX
[15:52:12.376] EDU100 1001.83 +0.00 +0.00% O=1001.83 H=1001.83 L=1001.83 PRE_OPEN
You see one line, for whichever index published last. `TECH2` is throttled to one publish every 2 seconds, so `EDU100` is usually the one shown. `INDEX` is not a multi-index listing; use `pm-index-cli indices` to list both indices and `pm-stats-cli index-daily` to verify recorded daily data. On startup, `pm-index` requests the engine's current session status and publishes an initial update for each configured index, so a restart during `CONTINUOUS` does not leave the index reporting the default phase until the next transition. Before that status reply arrives, its phase is `UNKNOWN`, not a misleading `PRE_OPEN`.
Use `pm-index-cli` to list both configured indices, and `pm-stats-cli` to confirm `pm-stats` is capturing ticks for both:

```
pm-index-cli --config engine_config.yaml indices
pm-stats-cli index-ids
pm-stats-cli index-daily
```

`pm-index-cli indices` always shows every index in the config. `pm-stats-cli index-daily` with no `--index-id` filter shows the daily rollup for every index `pm-stats` has recorded — useful here to confirm both `EDU100` and `TECH2` are being tracked. `pm-stats-cli index-snapshots`, by contrast, requires an explicit `--index-id` (it returns raw ticks, not a summary, so a single-index scope keeps the output meaningful).

:material-checkbox-blank-outline: **Checkpoint:** `INDEX` shows the most recently received update; `pm-index-cli indices` lists both `EDU100` and `TECH2`; `pm-stats-cli index-daily` confirms daily rows for both.

28.14 Exercise 11: History Query via the Gateway

`pm-index` also accepts history requests directly through the ADMIN gateway. These go over the PULL socket (port 5559) and return records inline:

```
[OPS01]> INDEX|HISTORY|INDEX=EDU100
```

Run it in a `pm-alf-console` session (`pm-alf-console --id OPS01`), not in `pm-admin`, which has no `INDEX` command. Without a prior index update seen by that console it answers `INDEX|HISTORY requires INDEX=<id> or prior index.update.`, so a console you have just started needs the index named: `INDEX|HISTORY|INDEX=EDU100`. With one, it returns the last 30 days of **structural/audit records** — `INIT`, `CORP_ACTION`, `ADD_CONSTITUENT`, `DELIST` — for whichever index you last saw an `INDEX` update for (or pass `INDEX=<id>` explicitly, e.g. `INDEX|HISTORY|I`

```
[OPS01]> INDEX|HISTORY|INDEX=EDU100|FROM=2026-06-25|TO=2026-06-26
```

Returns structural/audit records within the specified date range, newest last. `FROM` is local midnight at the start of its date; date-only `TO` includes the entire date, through the final nanosecond before the next local midnight. `FROM=2026-06-25|TO=2026-06-25` therefore returns records from that day. Exact timestamp bounds

on the history request are inclusive.

No level or EOD ticks here

`INDEX|HISTORY` never returns intraday level ticks or end-of-day OHLC rows — only the same four structural/audit record types you saw in Exercise 8's `pm-index-cli events` output. For level and EOD history, use `pm-stats-cli index-snapshots` / `pm-stats-cli index-daily` instead; the gateway command has no equivalent for that data.

Gateway vs pm-index-cli vs pm-stats-cli

The gateway `INDEX|HISTORY` command is convenient for a quick structural lookup in the operator terminal without leaving your session. For richer filtering, multi-index queries, CSV/JSON export, or scripting workflows, use `pm-index-cli` instead — it reads the structural/audit JSONL file directly without going through the network. For level and EOD time-series data, neither of those tools applies — use `pm-stats-cli`.
:material-checkbox-blank-outline: **Checkpoint:** `INDEX|HISTORY` returns structural/audit records only; date-range filter works.

28.15 Summary

Concept	Command / file
Start index process	`pm-index`
Re-initialise from scratch	`pm-index --reset`
Live level query	`INDEX` (any gateway)
Structural/audit history query (gateway)	`INDEX|HISTORY`, `INDEX|HISTORY|FROM=…|TO=…` — INIT/CORP_ACTION/ADD_CONSTITUENT/DELI only
Stock split (no gateway command — use `pm-index-admin-cli`)	`pm-index-admin-cli --id ID split --index IDX --sym SYM --ratio N:M`
Cash dividend	`pm-index-admin-cli --id ID dividend --index IDX --sym SYM --amount …`
Shares issuance / buy-back	`pm-index-admin-cli --id ID shares --index IDX --sym SYM --new-shares …` (or `--delta …`)
Add constituent	`pm-index-admin-cli --id ID add --index IDX --sym SYM --shares … --price …`
Remove constituent	`pm-index-admin-cli --id ID delist --index IDX --sym SYM`
List configured indices	`pm-index-cli --config … indices`
Structural/audit events	`pm-index-cli --config … events [--type TYPE]`
Intraday level snapshots	`pm-stats-cli index-snapshots --index-id ID`
Daily OHLC (EOD) rollup	`pm-stats-cli index-daily [--index-id ID]`

Concept	Command / file
List indices seen by pm-stats	`pm-stats-cli index-ids`
CSV export (structural events)	`pm-index-cli --config … events --format csv > out.csv`
CSV export (daily OHLC)	`pm-stats-cli --format csv index-daily > out.csv`

28.16 Reflection

Why does a stock split leave the index divisor roughly unchanged while a cash dividend or shares issuance actually changes it? What would happen to the index's continuity (its "don't jump on non-economic events" property) if the divisor were recalculated the same way for all four corporate action types?

28.17 See Also

In this book:

- Chapter 15 — Statistics & Reporting — starting `pm-stats` and querying its SQLite database

Operator's Guide:

- Market Index — Operator's Guide — full reference for config fields, formulas, and history record types
- Index Admin CLI — full `pm-index-admin-cli` subcommand reference, `--dry-run`, and confirmation-prompt behaviour
- pm-index-cli reference — `events`/`indices` subcommands, column descriptions, and output-format options
- Statistics and Reporting — `pm-stats-cli index-daily` / `index-snapshots` / `index-ids` reference
- Engine Configuration — `indices:` YAML field reference

Reference Manual:

- Process Reference — pm-index — socket layout and message types

CHAPTER 29

ALF TCP Gateway Protocol Training

29.1 Objective

Train on end-to-end usage of the ALF protocol through `pm-alf-gwy` using OS CLI tools, the example parser libraries, and the interactive example clients.

You will practise:

- hand-writing an `alf_gateway:` config section (`pm-config-gen` has no `--alf-gateway` flag)
- starting `pm-alf-gwy` and verifying the port is listening using OS tools
- testing connectivity and the full session lifecycle manually with `nc` and `telnet`
- submitting, amending, and cancelling orders over a raw TCP session
- using the Python `alf_parser.py` / `alf_client.py` examples for scripted and interactive access
- building and running the C `alf_client` example
- operating control commands (`PING`, `ORDERS`, `QBOOT`, `KILL`)
- diagnosing error conditions from `ERR|CODE=...|REJECT_CODE=...`

Background reading

- ALF Gateway - ALF Protocol

29.2 Prerequisites

- Chapters 01-26 completed.
- Engine running and configured with at least two gateway IDs.
- `pm-alf-gwy` available in the environment (`poetry run pm-alf-gwy` or installed via `pipx`).
- Support libraries and clients available in `docs/examples/alf` (in a repository checkout; on a container or `pipx` install they are under `$EX/alf`, with `EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_root().parent)")`; the C client needs `make`, a C compiler and the readline headers, so its exercise is optional in the container image):
  - `python/alf_parser.py`, `python/alf_client.py`
  - `c/alf_parser.h`, `c/alf_parser.c`, `c/alf_client.c`, `c/Makefile`

Recommended startup terminals:

1. Engine: `pm-engine --verbose`
2. ALF gateway: `pm-alf-gwy`
3. One or more client terminals for connection exercises

29.3 Background

`pm-alf-gwy` is the TCP gateway for the ALF (Almost FIX) protocol. It accepts multiple simultaneous connections from external bots and scripts written in any language, and translates ALF commands into the same engine ZMQ messages used by the interactive `pm-alf-console` terminal.

In this chapter, the most important operational ideas are:

- Session establishment: `HELLO` → auth round-trip → `WELCOME` with automatic `SYMBOLS` bootstrap
- Authenticated order entry and management: `NEW`, `AMEND`, `CANCEL`, `QUOTE`
- Gateway kill-switch: `KILL` (with optional `SYM=` scope)
- Bootstrap state recovery: `ORDERS` and `QBOOT`
- Liveness: `PING`/`PONG` and unsolicited `HB` heartbeat lines
- Error escalation: per-error `ERR|CODE=...|REJECT_CODE=...` and sliding-window forced disconnect

29.4 Exercise 1: Generate a Config That Enables pm-alf-gwy

Unlike `post_trade_gateway`, `market_data_gateway`, `api_gateways`, and `balf_gateway`, `pm-config-gen` has **no `--alf-gateway*` flags** — there is no CLI generator support for the `alf_gateway:` section. Generate the base config (`--no-mm-seed-quotes` because `MM01` is a market maker and the generator would otherwise emit price-less seed stubs that `pm-config-deploy` refuses), then add `alf_gateway:` by hand:

```
pm-config-gen \
  --symbols AAPL MSFT \
  --participants TRADER01 TRADER02 OPS01:ADMIN MM01:MARKET_MAKER \
  --no-mm-seed-quotes \
  --output engine_config.yaml --force
```

Append the `alf_gateway:` section yourself:

```
alf_gateway:
  enabled: true
  name: "alf-gwy01"
  bind_address: "0.0.0.0"          # all interfaces; use 127.0.0.1 if you do not want other machines to reach it
  port: 5565
  heartbeat_interval_sec: 5
  handshake_timeout_sec: 10
  idle_timeout_sec: 30
```

Inspect the section:

```
grep -A10 '^alf_gateway:' engine_config.yaml
```

See ALF TCP Gateway → Configuration for the full field reference and defaults (the section is optional — omitting it entirely still works, using every default shown there).

Back up, check, deploy — then restart

This chapter starts from a fresh configuration (its operator is `OPS01`), and `--force` replaces your `engine_config.yaml`: save a copy first if you want to come back to it (`cp engine_config.yaml engine_config.before-ch27.yaml`). As always, nothing changes until you deploy it and restart the engine: `pm-cverifier engine_config.yaml`, then `pm-config-deploy engine_config.yaml`.
Start processes with that config:

```
pm-engine --verbose
pm-alf-gwy
```

Orders from earlier chapters are still in the data directory

Resting `DAY` orders and market-maker quote legs from the previous chapters are restored when the engine starts. Exercise 4's "`COUNT=0`" answers and Exercise 8's "kill clears everything" assume `MM01` and the traders start with nothing resting. For a clean venue stop every process and run `pm-opctl-cli clear --state --yes` before starting the engine (this also deletes `clearing.db` and `stats.db`). See Starting over for the reset matrix and preconditions.
:material-checkbox-blank-outline: Checkpoint: `pm-alf-gwy` starts, logs that it is listening on port 5565, and reports a successful engine ZMQ connection.

29.5 Exercise 2: Verify the Port Is Listening with CLI Tools

Before writing any client code, confirm the gateway is actually bound to the configured port using OS tools. This is the fastest way to rule out startup failures, bind-address mismatches, and firewall issues.

macOS:

```
# lsof — shows the process name and PID holding the port
sudo lsof -iTCP:5565 -sTCP:LISTEN

# BSD netstat (ships with macOS)
netstat -an | grep LISTEN | grep 5565
```

Expected: a line with `pm-alf-gwy` (or `Python` if running under Poetry) and `LISTEN`.

Linux:

```
# ss — preferred on modern Linux
ss -tlnp 'sport = :5565'

# lsof
sudo lsof -iTCP:5565 -sTCP:LISTEN

# netstat (older distributions)
netstat -tlnp | grep 5565
```

Both commands should show a process bound to `*:5565` or `127.0.0.1:5565`.

If no output appears:

- the gateway is not running or failed to start — check logs
- `alf_gateway.enabled` may be `false` in the config
- the port may differ from what you expect — re-run `grep -A5 'alf_gateway:' engine_config.yaml`

- `pm-engine` must be running before `pm-alf-gwy` can complete its ZMQ handshake

:material-checkbox-blank-outline: Checkpoint: at least one CLI command shows a process listening on port 5565.

29.6 Exercise 3: Manual Handshake Probe with nc

Use `nc` (netcat) to open a raw TCP connection and type ALF lines by hand. This verifies the full session lifecycle — HELLO, auth, WELCOME, SYMBOLS — with no code.

```
nc 127.0.0.1 5565
```

Type these lines, pressing Enter after each:

```
HELLO|CLIENT=test|PROTO=ALF1|ID=TRADER01
SYMBOLS
EXIT
```

Expected response pattern:

1. `WELCOME|PROTO=ALF1|GW=alf-gwy01|ID=TRADER01|HBINT=5|IDLE=30`
2. Automatic `SYMBOLS|COUNT=N` block with one `SYMBOL|SYM=...|TICK=...` per instrument
3. `END|TYPE=SYMBOLS`
4. `HB|TS=...` every 5 seconds when quiet
5. Connection closes after `EXIT`

Also try `ORDERS` to see the empty bootstrap snapshot:

```
HELLO|CLIENT=test|PROTO=ALF1|ID=TRADER01
ORDERS
EXIT
```

Expected: `ORDERS|COUNT=0|GW=TRADER01` and `END|TYPE=ORDERS`.

:material-checkbox-blank-outline: Checkpoint: `nc` session reaches `WELCOME` and receives a valid `SYMBOLS` block and an `END|TYPE=ORDERS` line.

29.7 Exercise 3B: telnet and Non-Interactive One-Liner

telnet (macOS / Linux):

```
telnet 127.0.0.1 5565
```

Type `HELLO|CLIENT=test|PROTO=ALF1|ID=TRADER01` and press Enter. `telnet` echoes characters locally so the line looks doubled in the terminal — the `WELCOME` response confirms the gateway accepted it. Press `Ctrl-]`, then type `quit` to close.

Non-interactive probe — useful in scripts or CI pipelines:

```
printf 'HELLO|CLIENT=ci|PROTO=ALF1|ID=TRADER01\nSYMBOLS\nEXIT\n' \
  | nc 127.0.0.1 5565
```

Expected: the full session including `WELCOME`, `SYMBOLS` block, and `END|TYPE=SYMBOLS` is printed to stdout and the command exits cleanly.

:material-checkbox-blank-outline: Checkpoint: the one-liner prints `WELCOME` and `END|TYPE=SYMBOLS` without

interaction.

29.8 Exercise 4: Submit and Cancel an Order over nc

Open a new `nc` session and type all commands (press Enter after each line):

```
nc 127.0.0.1 5565
```

```
HELLO|CLIENT=manual|PROTO=ALF1|ID=TRADER01
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=NC-ORDER-001
ORDERS
```

Expected behavior:

- `ACK|ORDER_ID=<uuid>|ACCEPTED=TRUE|...` arrives immediately after `NEW`
- the ACK and later lifecycle events echo `TAG=NC-ORDER-001`
- `ORDERS` shows the resting order with `STATUS=NEW` and `REMAINING=100`
- If `MM01` is quoting, the order may fill immediately and `ORDERS` returns `COUNT=0`

When an order fills, its `FILL` event carries `TRADE_IDS=<id[,id...]>`. Keep those durable public trade IDs with the order record: a swept order can contain several, in match order, and they join the private fill to the public trade tape, CALF, and drop copy without matching on price or time.

`FILL` also carries `LIQUIDITY=<MAKER|TAKER>` — whether this order supplied liquidity by resting on the book (`MAKER`) or removed it by crossing the spread (`TAKER`). If `MM01` filled your order immediately, expect `TAKER`; if you place a resting `LIMIT` and a later order crosses it, that later `FILL` (for the counterparty) says `TAKER` while yours says `MAKER`. It is the same attribution the drop-copy `DC_FILL` line has always reported (see Exercise 8 → DC below, or Market Data & Drop Copy) — you no longer need drop copy just to find out which side of a trade you were on.

Cancel the resting order using the UUID from the `ACK` and a request tag:

```
CANCEL|ID=<uuid>|RTAG=NC-CXL-001
```

Expected: `CANCELLED|ORDER_ID=<uuid>|TAG=NC-ORDER-001|RTAG=NC-CXL-001`.

Try amending before cancelling to see priority rules in action:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=200|PRICE=149.00|TAG=NC-ORDER-002
AMEND|ID=<uuid>|PRICE=149.50|QTY=150|RTAG=NC-AMD-001
```

Expected: `AMENDED|ORDER_ID=...|PRICE=149.5|QTY=150|REMAINING=150|PRIORITY_RESET=TRUE|TAG=NC-ORDER-002|R`

`RTAG` is optional, but it is the easiest way to prove which amend or cancel a later TCP response belongs to.

:material-checkbox-blank-outline: Checkpoint: you can submit, confirm, amend, and cancel an order using only raw TCP and a terminal.

29.9 Exercise 5: Use the Python Interactive Client

`alf_client.py` provides a `pm-alf-console`-style interactive session over TCP, with tab-completion, background event display, and local P&L tracking.

```
cd docs/examples/alf/python
python3 alf_client.py --id TRADER01
```

At the prompt:

```
[TRADER01]> SYMBOLS
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=PY-ORDER-001
[TRADER01]> ORDERS
[TRADER01]> POS
[TRADER01]> STATUS
[TRADER01]> HELP
[TRADER01]> EXIT
```

Expected behavior:

- `POS` shows positions and realized P&L from fills seen by this connection; `POS|GW=TRADER01` queries the engine's position snapshot
- `STATUS` shows session info: gateway name, heartbeat interval, idle timeout
- fills, `HB`, and broadcast `SESSION`/`TRADE` events appear in real time while you type other commands
- Tab completes command verbs and field names
- Command history is saved across sessions to `~/.alf_client_history`
- Both interactive clients send keepalive `PING` automatically; `DC_FILL` is displayed separately without counting its execution a second time

Connect to a remote gateway:

```
python3 alf_client.py --host 10.0.0.5 --port 5565 --id TRADER01
```

:material-checkbox-blank-outline: Checkpoint: interactive Python client connects, shows `WELCOME`, and `ORDERS` and `POS` respond correctly.

29.10 Exercise 6: Use the Python Library Directly

`alf_parser.py` exposes `parse_alf_line`, `build_alf_line`, and the `AlfSession` high-level class for use in scripts.

```
cd docs/examples/alf/python
python3 - <<'EOF'
from alf_parser import parse_alf_line, build_alf_line, AlfSession, AlfMessage

# Parse a line received from the gateway
msg: AlfMessage = parse_alf_line("ACK|ORDER_ID=abc|ACCEPTED=FALSE|REJECT_CODE=ORDER_NOT_FOUND|REASON=missing")
print(msg.msg_type)   # ACK
print(msg.fields)     # {"ORDER_ID": "abc", "ACCEPTED": "FALSE", "REJECT_CODE": "ORDER_NOT_FOUND", ...}

# Build a line to send
line = build_alf_line("NEW", {
    "SYM": "AAPL", "SIDE": "BUY",
    "TYPE": "LIMIT", "QTY": "100", "PRICE": "150.00", "TAG": "NC-ORDER-003",
})
print(repr(line))  # 'NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=NC-ORDER-003\n'

# High-level session: connect, HELLO/WELCOME, consume a complete query
session = AlfSession.connect("127.0.0.1", 5565, "TRADER01")
try:
    print(session.welcome.gw_name)   # alf-gwy01
    session.sock.settimeout(2.0)    # fail visibly if this short exercise stalls
    session.send("ORDERS")
    while True:
        msg = session.recv_msg()
        print(msg.msg_type, msg.fields)
        if msg.msg_type == "END" and msg.fields.get("TYPE") == "ORDERS":
            break
finally:
    session.close()
EOF
```

Expected output: `ACK`, the parsed `fields` dict, the built line string, the gateway name from `WELCOME`, and the complete `ORDERS` response, possibly interleaved with heartbeats or broadcasts.

This short exercise closes after one query. `AlfSession` validates the handshake and buffers TCP lines but does not start keepalives or reconnect. For a persistent application, copy `alf_parser.py` beside your program and use the application integration walkthrough, which includes periodic `PING`, a query deadline, error handling, and reconciliation responsibilities.

:material-checkbox-blank-outline: Checkpoint: `AlfSession.connect` completes the HELLO/WELCOME handshake and `recv_msg` returns a parsed message.

29.11 Optional Exercise 7: Build and Run the C Client

The container image does not include `make`, a C compiler, or readline development headers. Skip this exercise there; Exercise 5's Python interactive client provides the same gateway workflow.

macOS — install Homebrew readline for full callback support if not already present:

```
brew install readline
```

Linux:

```
sudo apt install libreadline-dev   # Debian/Ubuntu
```

Build:

```
cd docs/examples/alf/c
make
```

Run:

```
./alf_client --id TRADER01
```

At the prompt:

```
[TRADER01]> SYMBOLS
[TRADER01]> NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=C-ORDER-001
[TRADER01]> ORDERS
[TRADER01]> EXIT
```

Expected behavior:

- the C client uses `select()` to multiplex the socket and stdin, so gateway events (fills, heartbeats, broadcasts) appear in real time while you type
- readline provides tab completion and persistent command history
- `--no-color` disables ANSI colour output for plain terminals

Try connecting to a remote host:

```
./alf_client --host 10.0.0.5 --port 5565 --id TRADER01
```

:material-checkbox-blank-outline: Optional checkpoint: the C client connects, shows `WELCOME`, and a `NEW` order returns an `ACK`.

29.12 Exercise 8: Control Commands and the Kill-Switch

29.12.1 PING / PONG

```
nc 127.0.0.1 5565
```

```
HELLO|CLIENT=ctrl|PROTO=ALF1|ID=TRADER01
PING
```

Expected: `PONG|TS=<ISO-8601-timestamp>`.

29.12.2 QBOOT — quote bootstrap state (MARKET_MAKER role)

Open a second terminal and connect with the MM01 identity:

```
nc 127.0.0.1 5565
```

```
HELLO|CLIENT=mm|PROTO=ALF1|ID=MM01
QBOOT
```

Expected: `QBOOT|COUNT=0` and `END|TYPE=QBOOT` initially.

Submit a quote then re-check:

```
QUOTE|SYM=AAPL|BID=149.90|ASK=150.10|BID_QTY=500|ASK_QTY=500
QBOOT
```

Expected: `QBOOT|COUNT=1` with one active `QUOTE|...` line, then `END|TYPE=QBOOT`. (If `MM01` still had a quote on AAPL from an earlier chapter you also see a `CANCELLED|…|CANCEL_REASON=QUOTE_REPLACED` line for the old one.)

29.12.3 KILL — cancel all resting orders and active quotes

```
KILL
```

Expected: `KILL_ACK|ACCEPTED=TRUE|ORDERS=N|QUOTES=N`.

Confirm everything was cleared:

```
ORDERS
QBOOT
```

`KILL_ACK` reports non-quote orders in `ORDERS` and quote-origin orders in `QUOTES`. A partially filled quote can be inactive while its unfilled remainder still rests; `KILL` removes that remainder and includes it in `QUOTES`, even after an engine restart.

For example, if 340 shares of a 1,000-share quote leg filled before restart, `ORDERS` may show its 660-share remainder. After `KILL`, the acknowledgement reports `ORDERS=0` and `QUOTES=1` for that remainder. `ORDERS` then returns `ORDERS|COUNT=0|GW=MM01` followed by `END|TYPE=ORDERS`; `QBOOT` is empty because the quote was already inactive. If other orders or quotes are resting, their cancellations are included in the corresponding acknowledgement count.

Scope the kill to one symbol:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=149.00
NEW|SYM=MSFT|SIDE=BUY|TYPE=LIMIT|QTY=50|PRICE=400.00
KILL|SYM=AAPL
ORDERS
```

Expected: only the MSFT order remains.

:material-checkbox-blank-outline: Checkpoint: you can use `PING`, `QBOOT`, and `KILL` (with and without `SYM=`) and interpret the responses.

29.13 Exercise 9: Error Conditions and Operational Interpretation

Test typical protocol errors. Start a fresh `nc` session and send each line before completing the `HELLO` handshake, or with invalid content:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=BAD-AUTH-001    (before HELLO)
HELLO|CLIENT=test|PROTO=WRONG|ID=TRADER01
HELLO|CLIENT=test|PROTO=ALF1|ID=NONEXISTENT
```

After a successful `HELLO`, try:

```
NEW|SYM=AAPL|SIDE=BUY|TYPE=LIMIT|QTY=100|TAG=BAD-MISSING-001          (missing PRICE for LIMIT)
NEW|SYM=UNKNOWN|SIDE=BUY|TYPE=LIMIT|QTY=100|PRICE=150.00|TAG=BAD-SYM-001
QUOTE|SYM=AAPL|BID=150.00|ASK=150.10|BID_QTY=500|ASK_QTY=500   (TRADER role)
```

Map each observed `ERR|CODE=...|REJECT_CODE=...` to its operator remediation:

`CODE`	Typical `REJECT_CODE`	Practical interpretation
`AUTH_REQUIRED`	`AUTH_REQUIRED`	Client sent a non-`HELLO` first line; fix client code ordering
`PROTO_MISMATCH`	`AUTH_FAILED`	Wrong `PROTO=` value; must be exactly `ALF1`

CODE	Typical REJECT_CODE	Practical interpretation
AUTH_FAILED	UNKNOWN	Gateway ID not in participants; add to config and restart engine
GATEWAY_ALREADY_CONNECTED	AUTH_FAILED	Same ID connected from another session; disconnect it first
MISSING_FIELD	MISSING_FIELD	Required field absent for this order type; check command reference
INVALID_VALUE	INVALID_VALUE	Field value fails validation (e.g. PRICE=NaN, QTY=0); fix client
SYMBOL_NOT_CONFIGURED	UNKNOWN_SYMBOL	Unknown symbol; run SYMBOLS to see the configured list
ROLE_DENIED	ROLE_DENIED	Command not allowed for this gateway's role (QUOTE requires MARKET_MAKER)
RATE_LIMITED	RATE_LIMITED	Commands arriving faster than max_commands_per_second; throttle client

CODE is the ALF gateway's local validation category. REJECT_CODE is the canonical order-rejection class used by ALF, REST, and tests. When the rejected line carried TAG or RTAG, the gateway echoes it as TAG on the ERR so a client can correlate the failure.

:material-checkbox-blank-outline: Checkpoint: you can map each ERR|CODE and REJECT_CODE to the correct config or client-code fix.

29.14 Support Libraries and Example Clients

Reference implementations used in this training chapter:

- docs/examples/alf/python/alf_parser.py — protocol library: parse_alf_line, build_alf_line, AlfSession
- docs/examples/alf/python/alf_client.py — interactive client with tab-completion, event display, and P&L tracking
- docs/examples/alf/c/alf_parser.h / alf_parser.c — C protocol library: alf_parse_line, alf_build_line, alf_get_field
- docs/examples/alf/c/alf_client.c — interactive C client using select() + readline

Use these to bootstrap ALF clients in any language that can open a TCP socket and read newline-delimited text. The Python parser/session and C parser libraries do not need readline; only the interactive clients do. The C parser is not a session library: adapt the C client's transport and lifecycle code for a headless application, check parser/builder return codes, and review its fixed capacities. See the reusable-client guide for the integration checklist.

29.15 Summary

You can now:

- Hand-write the `alf_gateway:` config section and start `pm-alf-gwy`.
- Verify a TCP port is listening using `lsof`, `ss`, and `netstat` on macOS and Linux.
- Test the complete ALF session lifecycle manually with `nc` and `telnet` before writing any code.
- Submit, amend, and cancel orders over a raw TCP session and read `ACK`, `AMENDED`, and `CANCELLED` responses.
- Correlate an ALF `FILL` to its public executions through `TRADE_IDS`.
- Use `python3 alf_client.py` for interactive sessions with tab-completion and live event display.
- Script against the gateway using `AlfSession` from `alf_parser.py`.
- Build and run the C `alf_client` for a readline-based interactive session with `select()` multiplexing.
- Operate `PING`, `ORDERS`, `QBOOT`, and `KILL` and interpret their responses.
- Map `ERR|CODE=...|REJECT_CODE=...` values to the correct config or client-side fix.

29.16 Reflection

`pm-alf-gwy` (this chapter) and `pm-api-gwy` (Chapter 25) both allow external clients to submit orders to the same matching engine. Answer the following:

1. A Python bot connecting to `pm-alf-gwy` receives a `FILL` event pushed directly over its persistent TCP connection. A Python bot connecting to `pm-api-gwy` learns about the same fill through a WebSocket `events` stream. Both need to act on fills within milliseconds. Which transport has lower latency overhead and why?

2. A compliance team needs read-only access to every fill for **all** gateway IDs — not just their own. Can they use `pm-alf-gwy` for this? If not, which gateway process should they consume instead and with what role?

3. After a network interruption, an ALF bot reconnects and sends `ORDERS` to recover resting state. The API gateway equivalent is `GET /api/v1/orders`. At the protocol level, which reconnect handler is simpler to implement and why?

29.17 Handoff for Chapter 28

This completes the external gateway protocol series. `pm-engine`, `pm-alf-gwy`, and any other running processes can remain up if you want to continue a live session, or you can stop them and start fresh for the next chapter.

29.18 Further Reading

- ALF TCP Gateway — configuration, session lifecycle, command reference, and troubleshooting
- ALF Protocol Reference — formal wire syntax and full field/enum definitions
- ALF Console (pm-alf-console) — interactive command reference for `pm-alf-console`
- Protocol Support Library Examples
- Processes, Environment and Ports

CHAPTER 30

BALF TCP Gateway Protocol Training

30.1 Objective

Train on end-to-end usage of the BALF protocol through `pm-balf-gwy` using OS CLI tools, a minimal Python client, and the BALF parser reference libraries.

You will practise:

- generating a `balf_gateway:` config with `pm-config-gen`
- starting `pm-balf-gwy` and verifying the port is listening using OS tools
- testing connectivity and session auth with a binary `LOGON` probe script
- submitting and cancelling orders over a raw BALF TCP session
- handling heartbeats (`HEARTBEAT` / `HEARTBEAT_ACK`) correctly
- decoding `LOGON_ACK`, `ORDER_ACK`, and `CANCEL_ACK` responses
- validating frame parsing using the Python and C BALF parser examples
- diagnosing reject/error conditions from BALF reject codes

Background reading

- BALF Gateway - BALF Protocol

30.2 Prerequisites

- Chapters 01-27 completed.
- Engine running and configured with at least two gateway IDs.
- `pm-balf-gwy` available in the environment (`poetry run pm-balf-gwy` or installed via `pipx`).
- Support parser examples available in `docs/examples/balf` (in a repository checkout; on a container or `pipx` install use `$EX/balf`, with `EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_root().parent)")`; Exercise 5's C check is optional and needs a C compiler, which the container image does not have):
  - `balf_parser.py`
  - `balf_parser.c`

Recommended startup terminals:

1. Engine: `pm-engine --verbose`
2. BALF gateway: `pm-balf-gwy`
3. One or more client terminals for connection exercises

30.3 Background

`pm-balf-gwy` is the TCP gateway for BALF (Binary ALF). It accepts simultaneous external client connections and translates fixed-width binary frames into the same engine semantics used by ALF command workflows.

In this chapter, the most important operational ideas are:

- Session establishment: `LOGON` -> auth round-trip -> `LOGON_ACK`
- Authenticated order entry and management: `NEW_ORDER`, `CANCEL_ORDER`, `AMEND_ORDER`
- Fill and lifecycle feedback: `ORDER_ACK`, `CANCEL_ACK`, `AMEND_ACK`, `EXECUTION_REPORT`
- Liveness: `HEARTBEAT`/`HEARTBEAT_ACK` and timeout-driven disconnects
- Binary safety rules: fixed frame size by `msg_type`, little-endian numeric fields, reserved bytes set to zero
- Error interpretation: BALF reject codes in `LOGON_ACK` and `ORDER_ACK`

30.4 Exercise 1: Generate a Config That Enables `pm-balf-gwy`

Generate `engine_config.yaml` with the `balf_gateway:` section enabled:

```
pm-config-gen \
  --symbols AAPL MSFT \
  --participants TRADER01 TRADER02 OPS01:ADMIN MM01:MARKET_MAKER \
  --no-mm-seed-quotes \
  --balf-gateway \
  --balf-port 5560 \
  --balf-bind-address 127.0.0.1 \
  --output engine_config.yaml --force
```

Back up, check, deploy — then restart

This chapter starts from a fresh configuration (its operator is `OPS01`), and `--force` replaces your `engine_config.yaml`: save a copy first if you want to come back to it (`cp engine_config.yaml engine_config.before-ch28.yaml`). As always, nothing changes until you deploy it and restart the engine: `pm-cverifier engine_config.yaml`, then `pm-config-deploy engine_config.yaml`.
Orders and market-maker quotes from earlier chapters are restored when the engine starts. For the empty AAPL book that Exercise 4B assumes, stop every process and run `pm-opctl-cli clear --state --yes` before you start the engine (it also deletes `clearing.db` and `stats.db`). See Starting over for the reset matrix and preconditions.
Inspect the generated section:

```
grep -A20 '^balf_gateway:' engine_config.yaml
```

Expected output includes:

```
balf_gateway:
  name: balf-gwy01
  bind_address: 127.0.0.1
  port: 5560
  heartbeat_interval_sec: 1
  heartbeat_timeout_sec: 5
  idle_timeout_sec: 30
  auth_timeout_sec: 10
  max_connections: 64
```

Start processes with that config:

```
pm-engine --verbose
pm-balf-gwy
```

:material-checkbox-blank-outline: Checkpoint: `pm-balf-gwy` starts, logs that it is listening on port 5560, and reports a successful engine ZMQ connection.

30.5 Exercise 2: Verify the Port Is Listening with CLI Tools

Before writing client code, confirm the gateway is bound to the configured port.

macOS:

```
sudo lsof -iTCP:5560 -sTCP:LISTEN
netstat -an | grep LISTEN | grep 5560
```

Expected: a line with `pm-balf-gwy` (or `Python` under Poetry) and `LISTEN`.

Linux:

```
ss -tlnp 'sport = :5560'
sudo lsof -iTCP:5560 -sTCP:LISTEN
netstat -tlnp | grep 5560
```

If no output appears:

- the gateway is not running or failed to start
- `balf_gateway.enabled` may be `false` in config
- the port may differ from expectation
- `pm-engine` must be running before `pm-balf-gwy` can complete its startup handshake

:material-checkbox-blank-outline: Checkpoint: at least one CLI command shows a process listening on port 5560.

30.6 Exercise 3: Manual Binary `LOGON` Probe with Python

Because BALF is binary, use a short Python probe instead of `nc` or `telnet`. This verifies the full auth lifecycle with no client framework.

```
python3 - <<'EOF'
import socket
import struct

BALF_MAGIC = 0xBA
BALF_VERSION = 0x01
MSG_LOGON = 0x01
MSG_LOGON_ACK = 0x02


def recv_exact(sock, n):
    buf = bytearray()
    while len(buf) < n:
        chunk = sock.recv(n - len(buf))
        if not chunk:
            raise RuntimeError("connection closed")
        buf.extend(chunk)
    return bytes(buf)

header = struct.pack("<BBBBI", BALF_MAGIC, BALF_VERSION, MSG_LOGON, 0, 0)
body = struct.pack("<16sB7s", b"TRADER01", BALF_VERSION, b"\x00" * 7)

sock = socket.create_connection(("127.0.0.1", 5560), timeout=5)
sock.sendall(header + body)

ack = recv_exact(sock, 92)
msg_type = ack[2]
if msg_type != MSG_LOGON_ACK:
    raise RuntimeError(f"expected LOGON_ACK, got 0x{msg_type:02X}")

payload = ack[8:]
gw_raw, accepted, reject_code, msg_len, _, msg_bytes = struct.unpack("<16sBBBB64s", payload)
gw = gw_raw.rstrip(b"\x00").decode("ascii", errors="replace")
msg = msg_bytes[:msg_len].decode("ascii", errors="replace")

print(f"LOGON_ACK gateway={gw!r} accepted={bool(accepted)} code=0x{reject_code:02X} msg={msg!r}")
sock.close()
EOF
```

Expected success pattern:

```
LOGON_ACK gateway='TRADER01' accepted=True code=0x00 msg='gateway=balf-gwy01 hbint=1s'
```

:material-checkbox-blank-outline: Checkpoint: probe receives `LOGON_ACK` with `accepted=True`.

30.7 Exercise 3B: Non-Interactive Reachability Check for CI

Use this one-liner to fail fast in scripts and CI:

```
python3 - <<'EOF'
import socket, struct, sys
sock = socket.create_connection(("127.0.0.1", 5560), timeout=3)
logon = struct.pack("<BBBBI16sB7s", 0xBA, 0x01, 0x01, 0, 0, b"TRADER01", 0x01, b"\x00"*7)
sock.sendall(logon)
ack = sock.recv(92)
ok = len(ack) == 92 and ack[2] == 0x02 and ack[24] == 1
print("BALF_PROBE_OK" if ok else "BALF_PROBE_FAIL")
sock.close()
sys.exit(0 if ok else 1)
EOF
```

Expected: prints `BALF_PROBE_OK` and exits with code `0`.

:material-checkbox-blank-outline: Checkpoint: one-liner succeeds without manual interaction.

30.8 Exercise 4: Submit and Cancel an Order over BALF

Run a minimal BALF client that authenticates, sends `NEW_ORDER`, waits for `ORDER_ACK`, then sends `CANCEL_ORDER` and checks `CANCEL_ACK`.

```
python3 - <<'EOF'
import socket
import struct

BALF_MAGIC = 0xBA
BALF_VERSION = 0x01
PRICE_SCALE = 100_000_000

MSG_LOGON = 0x01
MSG_LOGON_ACK = 0x02
MSG_NEW_ORDER = 0x10
MSG_ORDER_ACK = 0x11
MSG_CANCEL_ORDER = 0x12
MSG_CANCEL_ACK = 0x13
MSG_HEARTBEAT = 0x30
MSG_HEARTBEAT_ACK = 0x31
MSG_LOGOUT = 0x40

FRAME_SIZES = {
    MSG_LOGON_ACK: 92,
    MSG_ORDER_ACK: 60,
    MSG_CANCEL_ACK: 32,
    MSG_HEARTBEAT: 16,
}


def build_header(msg_type, seq_no):
    return struct.pack("<BBBBI", BALF_MAGIC, BALF_VERSION, msg_type, 0, seq_no)


def recv_exact(sock, n):
    buf = bytearray()
    while len(buf) < n:
        chunk = sock.recv(n - len(buf))
        if not chunk:
            raise RuntimeError("connection closed")
        buf.extend(chunk)
    return bytes(buf)


def recv_frame(sock):
    header = recv_exact(sock, 8)
    msg_type = header[2]
    size = FRAME_SIZES[msg_type]
    body = recv_exact(sock, size - 8)
    return msg_type, body

sock = socket.create_connection(("127.0.0.1", 5560), timeout=5)
seq = 0

# LOGON
logon = build_header(MSG_LOGON, 0) + struct.pack("<16sB7s", b"TRADER01", BALF_VERSION, b"\x00" * 7)
sock.sendall(logon)
msg_type, body = recv_frame(sock)
if msg_type != MSG_LOGON_ACK or body[16] != 1:
    raise RuntimeError("LOGON failed")
```

Expected behavior:

- `LOGON_ACK accepted`
- `ORDER_ACK accepted order_id=...`
- `CANCEL_ACK accepted`
- `LOGOUT sent`

:material-checkbox-blank-outline: Checkpoint: you can submit and cancel an order over a raw BALF TCP session.

30.9 Exercise 4B: Amend an Order and Receive Fills

Exercise 4 sent an order and cancelled it. The other half of order entry is what comes back when the order trades. This exercise uses two sessions — `TRADER01` rests a bid, amends it, and `TRADER02` sells into it — and decodes the `AMEND_ACK` and both sides' `EXECUTION_REPORT`. The market must be open (`SESSION|STATE=PRE_OPEN`, then `CONTINUOUS`, from `pm-admin --id OPS01`) and AAPL's book empty (the configuration from Exercise 1 has no seed quotes).

```
python3 - <<'EOF'
import socket
import struct

PRICE_SCALE = 100_000_000
FRAME_SIZES = {0x02: 92, 0x11: 60, 0x13: 32, 0x15: 48, 0x20: 64, 0x30: 16}


def header(msg_type, seq):
    return struct.pack("<BBBBI", 0xBA, 0x01, msg_type, 0, seq)


def recv_exact(sock, n):
    buf = bytearray()
    while len(buf) < n:
        chunk = sock.recv(n - len(buf))
        if not chunk:
            raise RuntimeError("connection closed")
        buf.extend(chunk)
    return bytes(buf)


def next_frame(sock, want):
    """Return the body of the next `want` frame, answering heartbeats."""
    while True:
        h = recv_exact(sock, 8)
        body = recv_exact(sock, FRAME_SIZES[h[2]] - 8)
        if h[2] == 0x30:
            sock.sendall(header(0x31, 0) + body[:8])
            continue
        if h[2] == want:
            return body


def session(gateway):
    sock = socket.create_connection(("127.0.0.1", 5560), timeout=10)
    sock.sendall(header(0x01, 0) + struct.pack("<16sB7s", gateway, 1, b"\x00" * 7))
    assert next_frame(sock, 0x02)[16] == 1, "LOGON rejected"
    return sock


def new_order(sock, seq, client_id, side, price, qty):
    sock.sendall(
        header(0x10, seq)
        + struct.pack("<Q8sqqqIIBBBB", client_id, b"AAPL", round(price * PRICE_SCALE), 0, 0, qty, 0, side, 2, 1, 0
    )
    return struct.unpack_from("<Q", next_frame(sock, 0x11), 8)[0]


def report(label, body):
    cid, oid, px, qty, rem = struct.unpack_from("<QQqII", body)
    print(f"{label}: EXECUTION_REPORT client_order_id={cid} fill={qty}@{px / PRICE_SCALE:.2f} remaining={rem}")


resting = session(b"TRADER01")
taker = session(b"TRADER02")
```

Expected:

```
TRADER01 resting order_id 1
AMEND_ACK accepted=True price=150.50 qty=200 priority_reset=True
TRADER02 (taker): EXECUTION_REPORT client_order_id=7 fill=120@150.50 remaining=0
TRADER01 (resting): EXECUTION_REPORT client_order_id=1 fill=120@150.50 remaining=80
```

Read it against the BALF specification:

- `amend_flags 0b11` says both the price and the quantity changed. `new_quantity` is the new **total**, so 200 means 200 shares in all, as with `AMEND` in Chapter 04.
- `priority_reset=True`: a price change always loses time priority.
- The fill is at the resting order's price, 150.50 (the taker's limit equals it here). Both sides then get their own `EXECUTION_REPORT`; the `client_order_id` in each is the number that session chose (1 and 7). The 80 shares left on the bid would stay on the book until you cancel them with `CANCEL_ORDER`, as in Exercise 4 — but the script ends and closes both connections, and a closing session has its resting orders cancelled by the engine, so the book is empty again and you can run the script a second time with the same result.

:material-checkbox-blank-outline: Checkpoint: you decoded an `AMEND_ACK` with `priority_reset=True` and the two `EXECUTION_REPORT` frames of one trade, and can say why the resting side's `remaining` is 80.

30.10 Exercise 5: Validate Parser Libraries (Python and Optional C)

Run the Python parser self-test to verify your BALF decode tooling. The C parser check below is optional; the container image has no C compiler.

Python parser:

```
cd docs/examples/balf
python3 balf_parser.py
```

Expected output: `balf_parser.py self-test: OK`

C parser:

```
cd docs/examples/balf
cc -std=c11 -Wall -Wextra -pedantic -O2 -I../generated \
   balf_parser.c ../generated/edumatcher_order.c \
   ../generated/edumatcher_msg.c -o balf_parser
./balf_parser
```

Expected output includes:

```
LOGON_ACK gateway_id=TRADER01 accepted=1 reject_code=0 msg=ok
EXECUTION_REPORT order_id=4242 AAPL 25 @ 150.00000000 (FILLED, remaining 0)
LOGON_ACK gateway_id=TRADER01 accepted=0 reject_code=255 msg=SYMBOLS_UNAVAILABLE
  -> SYMBOLS_UNAVAILABLE: engine did not send its symbols in time; reconnect and log on again (check the engine i
CANCEL_ACK ...
AMEND_ACK ...
STOP_TRIGGERED order_id=42 AAPL TRAILING_STOP stop=148.00000000 -> MARKET
balf_parser.c self-test: OK
```

The second `LOGON_ACK` is the one to know: `SYMBOLS_UNAVAILABLE` means the engine accepted the connection but

never sent its symbol list in time. The gateway closes the connection after it, so retry with a new connection.

:material-checkbox-blank-outline: Checkpoint: the Python parser self-test passes; the optional C self-test also passes when a compiler is available.

30.11 Exercise 6: Duplicate Session Policy and Heartbeat Behaviour

Set `duplicate_session_policy` in `balf_gateway:` and observe behaviour by connecting two clients using the same gateway ID.

- With `REJECT_NEW`, second `LOGON` should receive `LOGON_ACK` with reject code `0x02`.
- With `EVICT_OLD`, first client disconnects and second session is accepted.

The default is `REJECT_NEW` (the generated config shows it). A second `LOGON` as `TRADER01` while the first session is open gets `accepted=False`, code `0x02` and the message `Gateway already connected`.

Also verify heartbeat handling:

- Keep a client idle and watch for `HEARTBEAT` frames (one per second here, `heartbeat_interval_sec: 1`).
- Ensure your client replies with `HEARTBEAT_ACK`.
- Skip ACK replies to confirm timeout disconnect after `heartbeat_timeout_sec`: with the generated values (1 s interval, 5 s timeout) a client that logs on and then only reads receives frames at 0, 1, 2, 3 and 4 s, and the gateway closes the connection at 5 s.

:material-checkbox-blank-outline: Checkpoint: observed both duplicate-session behaviour and heartbeat timeout semantics.

30.12 Exercise 7: Error Conditions and Operational Interpretation

Trigger representative reject conditions and map them to remediation:

1. `LOGON` with unknown `gateway_id`
2. `LOGON` with wrong `proto_version`
3. `NEW_ORDER` with unknown symbol
4. `NEW_ORDER` with invalid quantity (`0`)
5. `NEW_ORDER LIMIT` with missing required `price` (send `price=0`)

Use this mapping:

Message	Code	Practical interpretation
`LOGON_ACK`	`0x01`	Gateway ID not configured in engine
`LOGON_ACK`	`0x02`	Gateway already connected (duplicate session)
`LOGON_ACK`	`0x03`	Protocol version mismatch
`ORDER_ACK`	`0x03`	Symbol not configured
`ORDER_ACK`	`0x04`	Market closed

Message	Code	Practical interpretation
`ORDER_ACK`	`0x07`	Symbol halted
`ORDER_ACK`	`0x08`	Rejected by the current session phase
`ORDER_ACK`	`0x0B`	Rejected by a price collar
`ORDER_ACK`	`0x0C`	Invalid field — bad quantity, or a LIMIT order with no price
`ORDER_ACK`	`0xFF`	Other error; inspect reason string

The full code list is in Appendix: BALF Protocol — reject codes. Note that `ORDER_ACK` and `LOGON_ACK` use **different** code spaces: `0x01` on a logon means "gateway not configured", while on an order it means the same thing about the *gateway*, and a bad **symbol** is `0x03`.

:material-checkbox-blank-outline: Checkpoint: you can map BALF reject codes to the correct config or client-code fix.

30.13 Support Libraries and Example Clients

Reference implementations used in this training chapter:

- `docs/examples/balf/balf_parser.py` — Python BALF frame parser and decode helpers
- `docs/examples/balf/balf_parser.c` — C BALF frame parser and decode helpers

Use these as wire-format references when building clients in Python, C, C++, or Rust.

30.14 Summary

You can now:

- Generate a `balf_gateway:` config with `pm-config-gen` and start `pm-balf-gwy`.
- Verify a TCP port is listening using `lsof`, `ss`, and `netstat` on macOS and Linux.
- Probe a BALF session with a minimal binary `LOGON` script.
- Submit and cancel orders using raw BALF frames over TCP.
- Handle heartbeats and sequence progression safely in client code.
- Decode BALF responses and rejects using parser examples.
- Map `LOGON_ACK` and `ORDER_ACK` reject codes to concrete fixes.

30.15 Reflection

`pm-balf-gwy` (this chapter) and `pm-alf-gwy` (Chapter 27) both feed the same matching engine semantics but through different wire formats. Answer the following:

1. For a latency-sensitive strategy written in C++, why might fixed-width BALF framing outperform text ALF parsing in practice?
2. For classroom demos and ad-hoc operations, why is ALF often still a better choice than BALF despite higher parsing overhead?

3. If a BALF client repeatedly disconnects after being idle, which heartbeat settings and client behaviours should you inspect first?

30.16 Where to go next

This completes the BALF training chapter. `pm-engine`, `pm-balf-gwy`, and any supporting processes can remain up if you want to continue live testing, or you can stop them and start fresh for the next chapter.

30.17 Further Reading

- BALF TCP Gateway — configuration, lifecycle, and troubleshooting
- BALF Protocol Reference — formal frame layouts and enum/reference tables
- The Configuration Workflow — `balf_gateway:` generation with `pm-config-gen`
- Protocol Support Library Examples
- Processes, Environment and Ports

CHAPTER 31

IPO Valuation (`pm-valuation`)

31.1 Objective

Price an initial public offering the way an investment bank's equity capital markets desk would: value the company, build the book, choose the offer price, and then list the stock on EduMatcher and let the opening auction show whether the price was right.

You will practice:

- reading a valuation report from the verdict down to the cash flows
- telling a price set by fundamentals from a price set by demand
- using the what-if loop: change one answer, recalculate, compare
- reading a Monte Carlo distribution against the offer price
- finding what it takes to rescue a postponed IPO
- saving a scenario and handing it out as a fully specified case
- listing the priced IPO with `pm-new-symbol` and trading it

Background reading

- IPO Valuation - Listing a New Symbol

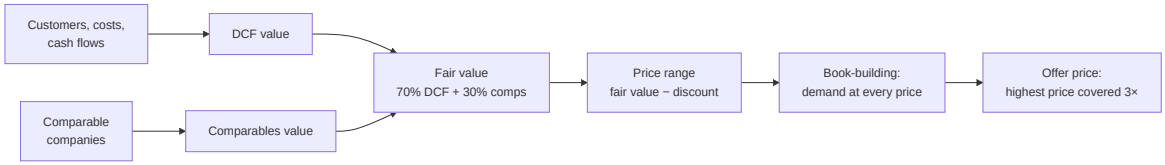
31.2 Prerequisites

- Chapter 00 completed (EduMatcher installed). Exercises 1–6 need nothing else: `pm-valuation` runs without an exchange.
- Exercises 7 and 8 list and trade the stock. They need chapters 01–03 (a running exchange with a market maker and two trader consoles) and the auctions chapter (Auctions), because the new stock opens in the opening auction.
- A terminal at least 120 columns wide: the interview is laid out for it, and printed reports are 100 columns wide.

31.3 Background

An IPO has two prices. **Fair value** is what the company's future cash flows are worth today. The **offer price** is what new investors pay. They differ on purpose: banks price an IPO at a discount to fair value (15% by default) so that the first day's trading ends above the offer price, which rewards the investors who took the risk of buying an unknown stock.

`pm-valuation` computes both:



Coverage is demand divided by the shares offered. A 12× covered book means investors asked for twelve times what is for sale. Banks price at the highest price that is still covered at least 3×, so that most investors receive less than they asked for and buy the rest in the market on day one. That is what drives the **first-day pop**. In Sweden the top of the price range in the prospectus is the **maximum price**: the deal cannot price above it. In the United States it may price up to about 20% above the filed range.

Two classroom cases ship with the tool. Both are Swedish companies listing on Nasdaq Stockholm, and their amounts are in SEK:

Case	Company	Sector
`tornfalk`	Tornfalk Security AB (TORN)	Business software
`halvard`	Halvard Robotics AB (HALV)	Hardware plus service

The American framework

`pm-valuation` follows the Swedish IPO process by default. `--market us` switches to the American one: an S-1 instead of a prospectus, US rates, tax and bank fees, and pricing up to 20% above the range. A case file's amounts are in its own market's currency, so `--market us` on a Swedish case reads its kronor as dollars. See Sweden or the United States.

31.4 Exercise 1: A First Report

Print the Tornfalk report, and export it to a file you can reread:

```
pm-valuation --case tornfalk --no-tui --mode deterministic --export tornfalk.md
```

```
[output]
Tornfalk Security AB (TORN) — IPO valuation  PROCEED

1. Verdict
  PROCEED

   Fair value per share           117.45
   Price range              94.50–105.00
   Offer price                    105.00
   Market capitalisation      16,600.0 m
   Primary raise               4,000.0 m
   Coverage                       11.92×
   Expected first-day pop          29.8%
```

:material-checkbox-blank-outline: The verdict is PROCEED at 105.00 SEK.

Answer from the report:

1. The offer price is 105.00, the top of the range. Why exactly 105.00, and not higher? (Section 13, Pricing.)
2. How much money did the company leave on the table, and who received it?
3. Fair value blends the DCF value and the comparables value. How far apart are they, and why?

Answers

1. 105.00 is the maximum price stated in the prospectus. Private investors applied on that promise, so the deal cannot price above it without a supplement that lets everyone withdraw. Even at 105.00 the book is 11.92× covered: demand would have supported more, and an American deal would have priced above the range. 2. 1,193.1 m SEK, about 1.2 miljarder kronor: the expected first-day pop (29.8%) times the offer price times the 38.1 million shares sold. The new investors receive it, not the company. 3. The DCF gives 92.09 and the comparables 176.62, almost twice as much. Listed software companies trade at 10 times next year's revenue, a multiple that Tornfalk's own forecast cash flows do not justify. The model weights the DCF 70% and the comparables 30%.

31.5 Exercise 2: Where the Value Comes From

Read sections 10 (DCF) and 11 (Bridge and fair value) in `tornfalk.md`.

1. What share of the enterprise value is the terminal value?
2. Fair value is 70% DCF and 30% comparables. Start the interview, set the DCF weight to 100% and press F5:

   ```
   pm-valuation --case tornfalk --mode deterministic --level intermediate
   ```

   Page 9, *DCF weight*: type `100`. (A plain number in a percentage field is percentage points.)

:material-checkbox-blank-outline: Fair value is now 92.09, the DCF value.

What changes

The terminal value is 66.6% of the enterprise value (question 1). With 100% DCF, the range falls to 74.00–82.50 and the deal prices at 82.50, again the maximum price, on an 11.85× book. Investor demand did not change, so the book pushes the price to the top of the range again. The valuation method moved

the range; demand chose the price inside it.

31.6 Exercise 3: The What-If Loop

Start the interview again, so the DCF weight is back to 70%, and press F5 to see the base report:

```
pm-valuation --case tornfalk --mode deterministic
```

Then:

1. Press `b` to go back. Every answer is kept.
2. On page 9, set *Institutional interest* and *Retail interest* to `medium` (Enter opens the pick-list) and *Hype* to `2`.
3. Press F5, then `c` to compare with the previous run.

:material-checkbox-blank-outline: The comparison shows the three inputs you changed.

Which moved more: the offer price, or the first-day pop? Why?

Answer

The price does not move: it stays at the maximum price, 105.00, now on a 5.01× book instead of 11.92×. The pop falls from 29.8% to 16.9%, and the money left on the table from 1,193.1 m to 676.0 m SEK. With fewer excess orders, fewer unfilled investors buy on day one. Sentiment shows up in the aftermarket before it shows up in the price.

31.7 Exercise 4: How Sure Are We?

Run the Monte Carlo simulation (10,000 draws; a few seconds):

```
pm-valuation --case tornfalk --no-tui --mode montecarlo --export tornfalk-mc.md
```

Read section 12 (Monte Carlo).

1. What is the median fair value, and how does it compare with the deterministic base case?
2. What is the probability that fair value is below the offer price?

Answers

1. The median is 110.48 and the mean 112.70, both below the base case 117.45. The driver ranges are skewed, so the base case is not the expected case. 2. 43.8%, close to a coin flip, even though the offer price is 11% below the base-case fair value. A hot book is a statement about demand, not about value. That is the question Exercise 7 puts to the market.

31.8 Exercise 5: A Postponed IPO

```
pm-valuation --case halvard --no-tui --mode deterministic
```

```
[output]
Halvard Robotics AB (HALV) — IPO valuation  POSTPONE
```

:material-checkbox-blank-outline: The verdict is POSTPONE.

1. Why? Read the Verdict and section 13 (Pricing).
2. Management's minimum market cap is derived from the last private round. Start the interview (`pm-valuation --case halvard --mode deterministic --level intermediate`) and, on page 11, lower *Minimum market cap* step by step: `2.6mdr`, `2.5mdr`, `2.4mdr`, `2.3mdr`. Watch the preview after each one. What is the highest minimum that lists the stock, and on what kind of book?

Answers

1. The last private round valued Halvard at 2.7 miljarder SEK post-money, and management will not list below that. The market cap after the IPO includes the 500 mkr raised, so the floor price is (2,700 mkr − 500 mkr) / 20 m pre-IPO shares = 110.00, which moves the range to 110.00–120.50. Its midpoint (115.25) is above fair value (112.68): the bankers need at least a 5% discount to sell the deal, and there is none left. 2. 2.5 mdr lists the stock, but on a thin book (2.63×, below the 3× target). At 2.3 mdr or less the book is covered 3×:

Minimum market cap	Result
2.7 mdr (the last round)	POSTPONE
2.6 mdr	POSTPONE: the discount is 2.2%, still under 5%
2.5 mdr	PROCEED (THIN BOOK) at 100.00, 2.63× covered
2.4 mdr	PROCEED (THIN BOOK) at 95.00, 2.98× covered
2.3 mdr or less	PROCEED at 94.50, 3.01× covered, market cap 2,390 mkr

Listing means accepting a **down round**: a public valuation of 2,390 m SEK against 2.7 miljarder in the last private round. Raising institutional interest does not help (try it): the floor binds before demand does.

31.9 Exercise 6: Save and Hand Out a Case

In the interview, press F9 and save to `halvard-rescue.yaml`. Then write a fully specified version for another group:

```
pm-valuation --load halvard-rescue.yaml --no-tui --mode deterministic \
    --save halvard-full.yaml --with-defaults
```

:material-checkbox-blank-outline: `halvard-rescue.yaml` holds only the answers you typed; `halvard-full.yaml` holds every value, each commented with its source (`you`, `preset`, `derived`, `default`).

Why does the short file stay correct if someone later changes a sector preset, while the full one does not?

31.10 Exercise 7: Let the Market Decide

List Tornfalk on your exchange and trade it.

1. Stop the exchange and list the priced IPO:

```
pm-opctl-cli stop
pm-valuation --case tornfalk --no-tui --mode deterministic --list
pm-opctl-cli start
```

```
[output]
Listed TORN in …/engine_config.yaml: IPO price 105.00, 158095238 shares outstanding
  seed quote MM01: 1000 @ 104.90 / 1000 @ 105.10 (DAY)
Deployed to …/engine_config.json. Start the exchange to open trading.
```

The seed-quote line appears when the deployed configuration has a market maker with seeded quotes (as in chapters 01–03). If you are still on one of the generated configurations of chapters 23–28 (`--no-mm-seed-quotes`), the command prints `no seed quote: the book opens empty` instead, as in Exercise

8. Either way the stock opens in the auction. The exchange must be stopped (every `pm-*` process, not only `pm-engine`) and the data directory must hold no saved state for `TORN`; a leftover `pm-engine` anywhere on the machine makes `--list` stop with `pm-engine is running (pid …)`.

2. During the pre-open, each trader enters orders for TORN at the price they believe in: buyers who missed out in the book, and allocated investors ready to take a profit.

3. Watch the opening auction set the first price (see Auctions).

:material-checkbox-blank-outline: TORN has an opening price.

Compare it with the offer price (105.00) and the report's expected first-day close ($105.00 \times 1.298 = 136.32$). Was the offer price too low, as the pop heuristic says, or too high, as the Monte Carlo suggests in almost one draw out of two?

If –list refuses

The refusals are `pm-new-symbol`'s: the deployed configuration is a bundled example (`--example …`), which is never edited — deploy a copy, as in Exercise 8; the exchange is still running, the data directory holds saved state for TORN from an earlier run, or the configuration has several market-maker gateways and one must be chosen for the seed quote. See Why it refused. For the last one, run the command from the report's Next step section yourself, adding `--mm-gateway-id`.

31.11 Exercise 8: No Market Maker — the Crowd Sets the Price

In Exercise 7 a market maker's seed quote (1000 @ 104.90 / 1000 @ 105.10) framed the opening around the offer price. A real IPO has no such anchor: the first price is whatever the orders in the auction say. List the same stock into a configuration **without** a market maker's quotes and see what the auction does with nothing but your classmates' orders.

`--list` refuses to edit a bundled example, so work on a copy (Chapter 00 says where the examples live; with a container or `pipx` install they are inside the package: replace `docs/examples` below by `$EX`, with `EX=$(python -c "from edumatcher.config_deploy import examples_root; print(examples_root().parent)")`):

```
pm-opctl-cli stop
cp docs/examples/ref_data/s3-nominal-nomm-setup/engine_config.yaml classroom.yaml
pm-config-deploy classroom.yaml
pm-valuation --case tornfalk --no-tui --mode deterministic --list
```

```
[output]
Listed TORN in …/classroom.yaml: IPO price 105.00, 158095238 shares outstanding
  no seed quote: the book opens empty
```

`s3-nominal-nomm` has sessions enabled (the opening auction needs them) and its market maker has no seeded

quotes, so TORN opens on an empty book. Start the exchange, then from the operator console (OPS01) go to PRE_OPEN and on to OPENING_AUCTION. Enter these three orders, all TIF=ATO:

```
[TRADER01]> NEW|SYM=TORN|SIDE=BUY|TYPE=LIMIT|QTY=1000|PRICE=125.00|TIF=ATO
[TRADER02]> NEW|SYM=TORN|SIDE=SELL|TYPE=LIMIT|QTY=600|PRICE=110.00|TIF=ATO
[TRADER02]> NEW|SYM=TORN|SIDE=SELL|TYPE=LIMIT|QTY=600|PRICE=120.00|TIF=ATO
```

Predict the opening price and the fills before moving on, using the equilibrium rule from Auctions. Then open continuous trading with SESSION|STATE=CONTINUOUS.

Answer

At 110.00 only 600 shares are offered against 1000 wanted, so 600 trade. At 120.00 and at 125.00 all 1000 buy shares can trade (1200 are offered, 200 too many). The maximum matched quantity, 1000, is first reached at **120.00**, and the engine keeps the lowest of tied prices, so TORN opens at 120.00: TRADER01 is filled 1000 @120.0; TRADER02 sells 600 + 400 @120.0 and the last 200 of its 120.00 offer expire unfilled, because it is an ATO order. The book is empty again.

Compare the result with the offer price (105.00) and with Exercise 7. A first price of 120.00 is 14% above the offer, which is the "pop" the report predicted (29.8% expected), reached without any quote from the exchange. Change TRADER01's bid to 115.00 and run the day again: which price do you predict now, and does the stock open above or below the offer?

:material-checkbox-blank-outline: **Checkpoint:** TORN listed with no seed quote, opening price predicted as 120.00 before it printed, and you can say why the answer is 120.00 and not 125.00.

31.12 Summary

Task	Command
Interview a classroom case	`pm-valuation --case tornfalk`
Show more fields	… `--level intermediate` (or `advanced`, `expert`), or F3
Print a report	`pm-valuation --case tornfalk --no-tui`
Scenarios only, no Monte Carlo	… `--mode deterministic`
Export the report	… `--export report.md`
Print the report	… `--pdf report.pdf` (or `p` in the report)
Save answers / a full case	F9, or `--save FILE [--with-defaults]`
Start from a saved file	`pm-valuation --load FILE`
Price and list	`pm-valuation --load FILE --no-tui --list`
In the report	`b` back, `c` compare, `e` export, `p` PDF, Tab next section

31.13 Reflection

In Exercise 1 the book was 12× covered at the maximum price, and in Exercise 4 the offer price was above fair value in almost half the draws. Both are true of the same deal. Who is right, the book or the model, and what does your opening auction in Exercise 7 say?

31.14 See Also

- IPO Valuation — Operator's Guide — every key, field format, report section and option
- Listing a New Symbol — what `--list` does to the configuration, and the IPO-day checklist
- Auctions & Scheduling — how the opening price is found
- Market Index — the index inclusion the report predicts