

TaskGraph Reference Guide

Will Woodman

Version 2.0, 2026-01-20

Table of Contents

Preface	1
Who Should Read This Guide	1
How to Use This Guide	1
1. Part I: Foundation	2
Architecture & Design Philosophy	3
2. Overview	4
3. Design Philosophy	5
3.1. Secure by Default	5
3.2. Fail-Fast Validation	5
3.3. Explicit Over Implicit	5
3.4. Composability First	6
4. Architectural Layers	7
4.1. Layer 1: Foundation	7
4.2. Layer 2: Execution	8
4.3. Layer 3: Orchestration	9
4.4. Layer 4: DSL	9
4.5. Layer 5: Application	10
5. Core Concepts	11
5.1. Directed Acyclic Graph (DAG)	11
5.2. Task Dependencies	11
5.3. Promise-Based Execution	11
5.4. Event-Driven Observability	12
6. Execution Model	13
6.1. Task Lifecycle	13
6.2. Concurrency Control	13
6.3. Error Handling	14
7. Design Patterns	15
7.1. Fork-Join Pattern	15
7.2. Pipeline Pattern	15
7.3. Saga Pattern	15
7.4. Conditional Routing	16
8. Performance Considerations	17
8.1. Graph Construction	17
8.2. Task Execution	17
8.3. Memory Management	17
9. Extensibility Points	18
9.1. Custom Task Types	18
9.2. Custom Security Models	18

9.3. Custom Event Listeners	18
9.4. SPI Implementations	18
10. Summary	19
11. Next Steps	20
DSL Approach	21
12. Why DSL Over XML/Annotations?	22
12.1. The Problem with XML Configuration	22
12.2. The Problem with Annotations	22
12.3. The DSL Advantage	23
13. DSL Design Principles	25
13.1. Principle 1: Fluent and Readable	25
13.2. Principle 2: Hierarchical Structure	25
13.3. Principle 3: Sensible Defaults	25
13.4. Principle 4: Type Safety	26
13.5. Principle 5: Composability	26
14. DSL Components	28
14.1. Graph-Level DSL	28
14.2. Task-Level DSL	28
14.3. Closure-Based Configuration	29
14.4. Method Reference Support	30
15. Dynamic Workflow Construction	31
15.1. Loops and Iteration	31
15.2. Conditional Task Creation	31
15.3. Parameterized Workflows	31
15.4. Higher-Order Functions	32
16. DSL Best Practices	33
16.1. Extract Common Configuration	33
16.2. Use Descriptive Task IDs	33
16.3. Group Related Tasks	34
16.4. Leverage Type Safety	34
17. DSL vs Programmatic API	36
17.1. DSL Approach (Recommended)	36
17.2. Programmatic API	36
17.3. Hybrid Approach	36
18. Comparison Summary	37
19. Summary	38
20. Next Steps	39
Foundation Libraries: Pool & Promises	40
21. Overview	41
22. ExecutorPool Framework	42
22.1. Virtual Threads by Default (Java 21+)	42

22.2. ExecutorPoolFactory - Named Pool Instances	43
22.2.1. Default Pool (Virtual Threads)	43
22.2.2. Named Pool Instances	43
22.2.3. Creating Custom Named Pools	44
22.3. When to Use Custom Pools	44
22.4. Pool Configuration (Advanced)	45
22.5. Pool Usage in TaskGraph	46
22.5.1. Default Behavior (Recommended)	46
22.5.2. Graph-Level Pool Override	47
22.5.3. Task-Level Pool Override	47
22.6. Pool Monitoring	48
22.7. Best Practices	48
23. Promise Framework	51
23.1. Core Concepts	51
23.1.1. Promise vs Future	51
23.2. Creating Promises	51
23.2.1. Task Promises	51
23.2.2. Completed Promises	52
23.2.3. Deferred Promises	52
23.3. Promise Composition	52
23.3.1. Sequential Composition	52
23.3.2. Transformation	52
23.3.3. Error Recovery	53
23.3.4. Parallel Composition	53
23.4. Promise Operations	54
23.4.1. Filter	54
23.4.2. Zip	54
23.4.3. FlatMap	54
23.4.4. OnComplete	55
23.5. Timeout and Cancellation	55
23.5.1. Timeout	55
23.5.2. Cancellation	55
23.6. Promise Usage in TaskGraph	56
23.6.1. Task Execution Returns Promise	56
23.6.2. Individual Task Promises	56
23.6.3. Combining Task Promises	57
23.7. Best Practices	57
24. Dataflow Variables	59
24.1. DataflowVariable	59
24.2. Usage Patterns	59
24.2.1. Producer-Consumer	59

24.2.2. Synchronization Point	59
24.3. TaskGraph Usage	60
25. Integration: How It All Works Together	61
25.1. Task Execution Flow	61
25.2. Example: Complete Flow with Virtual Threads	61
26. Performance Considerations	63
26.1. Virtual Threads - No Pool Sizing Required!	63
26.2. Platform Thread Pool Sizing (CPU-Intensive Only)	63
26.3. Promise Overhead	64
26.4. Dataflow Performance	64
27. Debugging and Monitoring	65
27.1. Thread Dumps	65
27.2. Promise Tracing	65
27.3. Pool Metrics	65
28. Summary	67
29. Next Steps	68
Simple Task Execution: Tasks and TasksCollection	69
30. Overview	70
31. Part 1: Tasks Utility	71
31.1. What is Tasks?	71
31.2. Core Execution Patterns	71
31.2.1. Execute All Tasks (Parallel)	71
31.2.2. Race to First Result (Any)	72
31.2.3. Sequential Pipeline (Sequence)	72
31.2.4. Parallel Execution (Non-Blocking)	73
31.3. Shared Context Execution	74
31.4. Single Task Execution	74
31.4.1. Groovy Closure Style	74
31.4.2. Java Lambda Style	75
31.5. Task Closure Parameters	75
31.6. Complete Examples	76
31.6.1. Example 1: Multi-Source Data Aggregation	76
31.6.2. Example 2: Fallback Strategy with Timeout	76
31.6.3. Example 3: ETL Pipeline	77
31.6.4. Example 4: Shared State Coordination	78
31.7. When to Use Tasks	78
32. Part 2: TasksCollection	80
32.1. What is TasksCollection?	80
32.2. Creating a TasksCollection	80
32.3. Task Registration Methods	81
32.3.1. Service Tasks	81

32.3.2. Timer Tasks	81
32.3.3. Business Rule Tasks	82
32.3.4. Subprocess Tasks (Call Activities)	82
32.3.5. Generic Task Registration	83
32.3.6. Register Existing Tasks	83
32.4. Task Discovery	83
32.4.1. Find by ID	83
32.4.2. Find with Filter	83
32.4.3. Get All Task IDs	84
32.4.4. Check if Task Exists	84
32.4.5. Get by Type	84
32.5. Lifecycle Management	84
32.5.1. Start Collection	84
32.5.2. Stop Collection	84
32.5.3. Clear Collection	84
32.6. Promise Chaining API	85
32.6.1. Basic Chain	85
32.6.2. Chain with Initial Value	85
32.6.3. Chain with Handlers	85
32.6.4. Async Chain Execution	85
32.7. Metrics and Monitoring	86
32.7.1. Task Counts	86
32.7.2. Task Count by State	86
32.7.3. Active Tasks	86
32.7.4. Summary Statistics	86
32.8. Complete Examples	87
32.8.1. Example 1: Event-Driven Monitoring System	87
32.8.2. Example 2: Data Processing Pipeline	88
32.8.3. Example 3: Microservices Orchestration	89
32.9. When to Use TasksCollection	91
33. Choosing the Right Abstraction	92
34. Migration Path	93
34.1. Tasks → TasksCollection	93
34.2. TasksCollection → TaskGraph	93
35. Summary	94
36. Next Steps	95
Architectural Layers: From Tasks to TaskGraph	96
37. Overview	97
38. Layer 1: Task	98
38.1. Task Hierarchy	98
38.2. TaskBase: The Foundation	98

38.2.1. Core Properties	98
38.2.2. Lifecycle Methods	99
38.2.3. Cross-Cutting Concerns	99
38.3. Task Types	100
38.3.1. ServiceTask - Generic Action	100
38.3.2. ScriptTask - Embedded Scripts	101
38.3.3. HttpTask - REST APIs	101
38.3.4. SqlTask - Database Operations	101
38.3.5. FileTask - File Operations	102
38.4. Task State Machine	102
38.5. Task Execution Model	103
38.5.1. Synchronous Execution	103
38.5.2. Asynchronous Execution	104
38.5.3. Dependency-Driven Execution	104
39. Layer 2: TaskCollection	105
39.1. Purpose	105
39.2. Creating Collections	105
39.3. Shared Configuration	105
39.4. Collection Operations	106
39.4.1. Start All Tasks	106
39.4.2. Query Collection State	106
39.4.3. Iterate Tasks	106
39.5. Collection Dependencies	107
39.6. Nested Collections	107
39.7. Use Cases	107
39.7.1. Parallel Processing by Domain	108
39.7.2. ETL Pipeline Stages	108
39.7.3. Environment-Specific Tasks	108
40. Layer 3: TaskGraph	110
40.1. Core Responsibilities	110
40.2. Graph Structure	110
40.2.1. Directed Acyclic Graph (DAG)	110
40.2.2. Cycle Detection	111
40.2.3. Missing Dependency Detection	111
40.3. Graph Configuration	111
40.3.1. Basic Configuration	111
40.3.2. Execution Strategies	111
40.4. Dependency Patterns	112
40.4.1. Sequential Pipeline	112
40.4.2. Fan-Out / Fan-In	112
40.4.3. Diamond Pattern	113

40.4.4. Parallel Independent Paths	114
40.5. Execution Control	115
40.5.1. Starting Execution	115
40.5.2. Pausing and Resuming	115
40.5.3. Cancellation	115
40.5.4. Partial Execution	116
40.6. Result Handling	116
40.6.1. GraphResult	116
40.6.2. Accessing Intermediate Results	117
40.7. Event Handling	117
40.7.1. Graph-Level Events	117
40.7.2. Filtering Events	118
40.8. Concurrency Control	118
40.8.1. Graph-Level Limits	118
40.8.2. Task-Level Limits	118
40.8.3. Resource Semaphores	118
40.9. Graph Composition	119
40.9.1. SubGraphs	119
40.9.2. Graph Factories	119
40.10. Performance Optimization	120
40.10.1. Parallel Execution	120
40.10.2. Lazy Task Creation	120
40.10.3. Result Cleanup	121
40.11. Debugging and Visualization	121
40.11.1. Graph Structure Inspection	121
40.11.2. Visualization Export	121
41. Layer Integration Example	123
42. Summary	125
43. Next Steps	126
Extensibility Through SPI	127
44. Overview	128
44.1. Design Philosophy	128
45. SPI Architecture	129
45.1. Core SPI Pattern	129
45.2. Provider Discovery	129
45.3. Provider Registration	129
46. Available SPI Extension Points	131
46.1. Database Providers	131
46.1.1. DatabaseProvider SPI	131
46.1.2. Default Implementation	131
46.1.3. Custom Implementation Example	132

46.1.4. Usage in Tasks	132
46.2. NoSQL Providers	132
46.2.1. NoSqlProvider SPI	132
46.2.2. Provided Implementations	133
46.2.3. Custom Implementation Example	133
46.3. Messaging Providers	134
46.3.1. MessagingProvider SPI	134
46.3.2. Provided Implementations	134
46.3.3. Custom Implementation Example	134
46.4. Storage Providers	135
46.4.1. ObjectStoreProvider SPI	135
46.4.2. Provided Implementations	136
46.4.3. Custom Implementation Example	136
46.5. Serialization Providers	137
46.5.1. SerializationProvider SPI	137
46.5.2. Provided Implementations	137
46.5.3. Custom Implementation Example	137
46.6. Credential Providers	138
46.6.1. CredentialProvider SPI	138
46.6.2. Provided Implementations	138
46.6.3. Custom Implementation Example	138
46.7. HTTP Client Providers	139
46.7.1. HttpClientProvider SPI	139
46.7.2. Provided Implementations	139
46.7.3. Custom Implementation Example	140
47. Creating Custom Task Types	141
47.1. Extending TaskBase	141
47.2. DSL Builder for Custom Tasks	141
47.3. Usage	142
48. Custom Gateway Types	143
48.1. Extending Gateway	143
48.2. DSL Builder	143
48.3. Usage	144
49. Custom Script Base Classes	145
49.1. Creating Custom Base Class	145
49.2. Usage in Scripts	145
50. Provider Selection Strategies	147
50.1. Priority-Based Selection	147
50.2. Explicit Provider Selection	147
50.3. Configuration-Based Selection	147
50.4. Environment-Based Selection	147

51. Testing Custom Providers	149
51.1. Unit Testing Providers	149
51.2. Integration Testing	149
52. Best Practices	150
52.1. Provider Design	150
52.2. Custom Task Design	151
53. Summary	153
54. Next Steps	154
55. Part II: Usage Guide	155
TaskContext: Shared Execution State	156
56. Overview	157
57. Variable Storage	158
57.1. Global Variables	158
57.2. Task-Spaced Variables	158
57.3. Previous Task Result	159
57.4. Task Results by ID	159
57.5. Variable Type Safety	160
58. Configuration Management	161
58.1. Hierarchical Configuration	161
58.2. Configuration Sources	161
58.3. Environment-Specific Configuration	162
58.4. Type-Safe Configuration	162
59. Credential Management	164
59.1. Credential Resolvers	164
59.2. Credential Caching	164
59.3. Structured Credentials	164
59.4. Credential Injection	165
60. Service Registry	166
60.1. Registering Services	166
60.2. Service Lifecycle	166
60.3. Service Factories	167
61. Event Dispatching	169
61.1. Publishing Events	169
61.2. Event Listeners	169
61.3. Typed Event Handlers	169
62. Context Inheritance	171
62.1. Parent-Child Context	171
62.2. Context Isolation	171
63. Advanced Context Features	172
63.1. Context Cloning	172
63.2. Context Snapshots	172

63.3. Context Validation	173
63.4. Context Debugging	173
64. Context Best Practices	175
64.1. Use Global Variables for Shared State	175
64.2. Use Configuration for Environment-Specific Values	175
64.3. Use Credentials for Sensitive Data	176
64.4. Use Service Registry for Shared Resources	176
64.5. Clean Up Task-Scoped Variables	177
65. Context Performance Considerations	178
65.1. Variable Access Performance	178
65.2. Memory Management	178
65.3. Credential Caching	178
66. Summary	180
67. Next Steps	181
Practical Examples and Usage Patterns	182
68. Overview	183
69. Example 1: Simple Data Pipeline (ETL)	184
70. Example 2: Fan-Out / Fan-In Pattern	186
71. Example 3: Conditional Routing	188
72. Example 4: Error Handling with Fallbacks	190
73. Example 5: Parallel Batch Processing	192
74. Example 6: Saga Pattern (Distributed Transaction)	194
75. Example 7: API Orchestration	196
76. Example 8: File Processing Pipeline	198
77. Example 9: Periodic Job with Scheduling	201
78. Example 10: Dynamic Workflow Construction	203
79. Common Patterns Summary	205
80. Next Steps	206
TaskGraph Test Harness	207
81. Overview	208
82. Test Harness Architecture	209
83. Setting Up Tests	210
83.1. Test Dependencies	210
83.2. Basic Test Structure	210
84. Unit Testing Tasks	211
84.1. Testing Individual Tasks	211
84.2. Testing Custom Tasks	212
85. Integration Testing Workflows	213
85.1. Testing Complete Workflows	213
85.2. Using Test Harness	213
86. Failure Injection Testing	215

86.1. Testing Error Handling	215
86.2. Testing Circuit Breaker	216
87. Time-Based Testing	218
87.1. Fast-Forward Time	218
87.2. Testing Scheduled Workflows	219
88. Execution Verification	221
88.1. Verifying Task Execution	221
88.2. Asserting Task Results	222
89. Performance Testing	224
89.1. Measuring Execution Time	224
89.2. Load Testing	224
90. Testing Conditional Logic	226
90.1. Testing Gateways	226
91. Testing Best Practices	228
91.1. Isolate External Dependencies	228
91.2. Use Test Data Builders	228
91.3. Test One Thing Per Test	229
91.4. Use Descriptive Test Names	229
92. Summary	230
93. Next Steps	231
TaskGraphSpec Testing Framework	232
94. Overview	233
94.1. Key Features	233
94.2. Design Philosophy	233
95. Getting Started	234
95.1. Setting Up Your Test Class	234
95.2. Automatic Setup	234
96. Fluent Execution API	235
96.1. Basic Execution Pattern	235
96.2. Configuration Options	235
96.3. Expecting Failures	235
96.4. Convenience Methods	236
97. Verification API	237
97.1. Execution Order Verification	237
97.2. Task Execution Checks	237
97.3. Task State Verification	238
97.4. Workflow Success/Failure	238
98. Input/Output Inspection	240
98.1. Task Output Verification	240
98.2. Task Input Inspection	240
98.3. Result Field Assertions	241

99. Timing and Performance	242
99.1. Task Execution Time	242
99.2. Total Execution Time	242
99.3. Timeout Testing	242
100. Mock Support	244
100.1. Mocking Task Results	244
100.2. Mocking Task Failures	244
100.3. Mock Storage	245
101. Test Data Builders	246
101.1. Fluent Input Builder	246
101.2. Custom Test Data Builders	246
102. Complete Examples	248
102.1. Example 1: Validation Workflow	248
102.2. Example 2: Transformation Pipeline	249
102.3. Example 3: Conditional Workflow	251
103. Best Practices	253
103.1. Write Focused Tests	253
103.2. Use Descriptive Test Names	253
103.3. Organize Tests with given-when-then	254
103.4. Isolate External Dependencies	254
103.5. Test Both Success and Failure Paths	254
103.6. Use Test Data Builders	255
104. Troubleshooting	256
104.1. Common Issues	256
104.1.1. Timeout Errors	256
104.1.2. Task Not Executed	256
104.1.3. Input Not Propagating	256
105. API Reference	257
105.1. TaskGraphSpec Methods	257
105.1.1. Execution Methods	257
105.1.2. Verification Methods	257
105.1.3. Inspection Methods	257
105.1.4. Timing Methods	257
105.1.5. Assertion Helpers	257
105.1.6. Mock Methods	257
105.1.7. Test Data Methods	258
105.2. TaskGraphExecution Methods	258
106. Summary	259
107. Next Steps	260
108. Part III: Operations	261
Security Architecture	262

109. Overview	263
110. Script Sandboxing	264
110.1. Default Sandbox Behavior	264
110.2. Sandbox Configuration	264
110.3. Disabling Sandbox (Not Recommended)	265
110.4. Custom Script Base Classes	265
111. File System Security	267
111.1. Allowed Directories	267
111.2. Path Traversal Prevention	267
111.3. File Size Limits	268
111.4. File Type Validation	268
112. Network Security	269
112.1. URL Validation	269
112.2. SSRF Prevention	269
112.3. DNS Resolution Control	270
112.4. Request Header Control	270
113. Credential Management	272
113.1. Credential Providers	272
113.2. Credential Injection	272
113.3. Credential Masking	273
113.4. Credential Rotation	273
114. Shared Security with ConfigBuilder	274
114.1. Consistent Security Model	274
114.2. Cross-Cutting Security	274
115. Audit Logging	276
115.1. Audit Configuration	276
115.2. Audit Events	276
115.3. Audit Log Format	277
116. Resource Limits	278
116.1. Memory Limits	278
116.2. Execution Time Limits	278
116.3. Concurrency Limits	278
116.4. Rate Limiting	279
117. Security Best Practices	280
117.1. Principle of Least Privilege	280
117.2. Defense in Depth	280
117.3. Secure Defaults	281
117.4. Regular Security Audits	281
118. Summary	283
119. Next Steps	284
Observability and Monitoring	285

120. Overview	286
121. Event System	287
121.1. Event Types	287
121.2. Event Listeners	287
121.3. Typed Event Handlers	288
121.4. Async Event Processing	289
121.5. Event Filtering	289
122. Execution Snapshots	291
122.1. Creating Snapshots	291
122.2. ExecutionSnapshot API	291
122.3. Periodic Snapshots	292
122.4. Snapshot Comparison	293
123. Health Monitoring	294
123.1. HealthStatus API	294
123.2. Health Checks	294
123.3. Health Endpoints	295
124. Metrics Collection	297
124.1. Built-in Metrics	297
124.2. Custom Metrics	297
124.3. Prometheus Integration	298
125. Distributed Tracing	300
125.1. OpenTelemetry Integration	300
125.2. Jaeger Integration	300
125.3. Custom Span Attributes	301
126. Logging Integration	302
126.1. Structured Logging	302
126.2. MDC (Mapped Diagnostic Context)	302
126.3. ELK Stack Integration	303
127. Monitoring Dashboards	304
127.1. Grafana Dashboard	304
127.2. Custom Dashboard	304
128. Alerting	306
128.1. Alert Rules	306
128.2. Alert Channels	306
129. Summary	308
130. Next Steps	309
131. Part IV: Reference	310
TaskBase Reference	311
132. Overview	312
133. Class Hierarchy	313
134. Core Properties	314

134.1. Identity	314
134.2. State Management	314
135. Lifecycle Methods	316
135.1. Starting Execution	316
135.2. Pausing and Resuming	316
135.3. Cancellation	316
136. Retry Policy	318
136.1. Basic Retry Configuration	318
136.2. Selective Retry	318
136.3. Retry Listeners	318
137. Circuit Breaker	320
137.1. Basic Circuit Breaker	320
137.2. Circuit Breaker States	320
137.3. Fallback on Circuit Open	321
138. Timeout Policy	322
138.1. Basic Timeout	322
138.2. Advanced Timeout Configuration	322
139. Idempotency Policy	323
139.1. Basic Idempotency	323
139.2. Custom Idempotency Store	323
140. Concurrency Control	324
140.1. Concurrency Limits	324
140.2. Resource Semaphores	324
140.3. Custom Executor Pool	324
141. Conditional Execution	326
141.1. Simple Condition	326
141.2. Complex Conditions	326
142. Event Handling	327
142.1. Lifecycle Hooks	327
142.2. Custom Events	328
143. Validation	329
143.1. Built-in Validation	329
143.2. Custom Validation	329
144. Error Handling	331
144.1. Error Recovery	331
144.2. Error Propagation	331
145. Result Management	333
145.1. Result Transformation	333
145.2. Result Caching	333
145.3. Result Cleanup	333
146. Metrics and Monitoring	335

146.1. Task Metrics	335
146.2. Custom Metrics	335
147. Best Practices	336
147.1. Use Retry for Transient Failures	336
147.2. Use Circuit Breaker for External Services	336
147.3. Set Appropriate Timeouts	336
147.4. Clean Up Large Results	337
148. Summary	338
149. Next Steps	339
Gateway Types Reference	340
150. Overview	341
151. Exclusive Gateway	342
151.1. Basic Usage	342
151.2. Multiple Conditions	342
151.3. Switch-Style Gateway	343
151.4. Path Validation	343
152. Parallel Gateway	344
152.1. Fork and Join	344
152.2. Dynamic Parallel Paths	344
152.3. Concurrent Execution Control	345
152.4. Timeout for Join	346
153. Inclusive Gateway	347
153.1. Basic Inclusive Gateway	347
153.2. Minimum Paths	347
153.3. Maximum Paths	348
154. Event Gateway	349
154.1. Basic Event Gateway	349
154.2. Message-Based Routing	349
154.3. Timer-Based Events	350
155. Complex Gateway	351
155.1. Custom Routing	351
155.2. State Machine Gateway	352
155.3. Weighted Routing	352
156. Gateway Patterns	354
156.1. Discriminator Pattern	354
156.2. N-out-of-M Join	354
156.3. Synchronizing Merge	355
157. Gateway Error Handling	356
157.1. Error in Path	356
157.2. Gateway Timeout	356
158. Gateway Observability	358

158.1. Gateway Events	358
158.2. Gateway Metrics	358
159. Best Practices	359
159.1. Use Exclusive for Mutually Exclusive Paths	359
159.2. Always Provide Default Path	359
159.3. Use Parallel for Independent Paths	359
159.4. Set Timeouts for Joins	360
160. Summary	361
161. Next Steps	362
Concrete Task Types Reference	363
162. Overview	364
163. Core Tasks	365
163.1. ServiceTask	365
163.2. ScriptTask	365
164. HTTP/REST Tasks	367
164.1. HttpTask	367
164.2. RestTask	368
164.3. GraphQLTask	369
165. Database Tasks	372
165.1. SqlTask	372
165.2. BatchSqlTask	373
165.3. NoSqlTask	374
166. File Operation Tasks	377
166.1. FileTask	377
166.2. CsvTask	378
166.3. JsonTask	379
166.4. XmlTask	380
167. Messaging Tasks	381
167.1. MessagingTask	381
167.2. KafkaTask	381
168. Cloud Service Tasks	383
168.1. S3Task	383
168.2. LambdaTask	384
169. Utility Tasks	385
169.1. DelayTask	385
169.2. LogTask	385
169.3. TransformTask	386
170. Task Configuration Patterns	387
170.1. Combining Features	387
171. Summary	389
172. Next Steps	390

Appendix A: Appendix A: Migration Guide	391
Appendix B: Appendix B: Performance Tuning	392
Appendix C: Appendix C: Troubleshooting	393
Appendix D: Appendix D: API Reference	394
Glossary	395

Preface

Welcome to the TaskGraph Reference Guide. This comprehensive documentation covers the architecture, design, and usage of the TaskGraph workflow orchestration framework.

TaskGraph provides a powerful, type-safe DSL for building complex workflow graphs with support for:

- **Declarative workflow definition** using Groovy DSL
- **Asynchronous execution** with promises and futures
- **Parallel processing** with configurable concurrency
- **Error handling and retry logic** built-in at every layer
- **Observability and monitoring** with event listeners
- **Security by default** with sandboxed execution
- **Extensibility** through SPI and custom implementations

Who Should Read This Guide

- **Developers** implementing workflows and data pipelines
- **Architects** designing distributed systems
- **DevOps Engineers** orchestrating complex operations
- **Library Consumers** extending TaskGraph for custom needs

How to Use This Guide

This guide is structured as a progressive learning experience:

1. **Part I (Chapters 1-5):** Foundation - Architecture, design philosophy, and core concepts
2. **Part II (Chapters 6-8):** Usage - Practical guides and examples
3. **Part III (Chapters 9-10):** Operations - Security, monitoring, and production deployment
4. **Part IV (Chapters 11-13):** Reference - Deep dive into components and task types

You can read sequentially or jump to specific chapters based on your needs.

Chapter 1. Part I: Foundation

Architecture & Design Philosophy

This chapter explores the fundamental architecture and design principles behind TaskGraph. Understanding these concepts will help you make the most of the framework and extend it effectively.

Chapter 2. Overview

TaskGraph is a workflow orchestration framework built on three core pillars:

1. **Type-safe DSL** - Groovy-based declarative workflow definition
2. **Asynchronous execution** - Non-blocking, promise-based execution model
3. **Composability** - Build complex workflows from simple, reusable components

Chapter 3. Design Philosophy

3.1. Secure by Default

TaskGraph follows a "secure by default" philosophy where security controls are enabled automatically:

- Script sandboxing enabled by default
- File access restricted to approved directories
- HTTP requests validate URLs and block SSRF attacks
- Credentials never hardcoded, always from secure sources

```
scriptTask("process-data") {
    sandboxed true // ☐ Default - secure mode
    script '''
        // Safe operations allowed
        def result = data.sum()
        return result

        // Blocked operations (throw SecurityException)
        // "rm -rf /.execute() // ☐ Blocked
    ...
}
```

3.2. Fail-Fast Validation

Configuration errors are caught at build time, not runtime:

- Type checking in DSL
- Validation during graph construction
- Circular dependency detection before execution
- Missing dependency detection upfront

3.3. Explicit Over Implicit

TaskGraph favors explicitness over magic:

- Dependencies explicitly declared
- Error handling explicitly configured
- Retry policies explicitly specified
- No hidden auto-wiring or conventions

```
task("validate-input") { /* ... */ }
task("process") {
    dependsOn "validate-input" // ☐ Explicit dependency
    retryPolicy {
        maxAttempts 3           // ☐ Explicit retry config
        delay 1000
    }
}
```

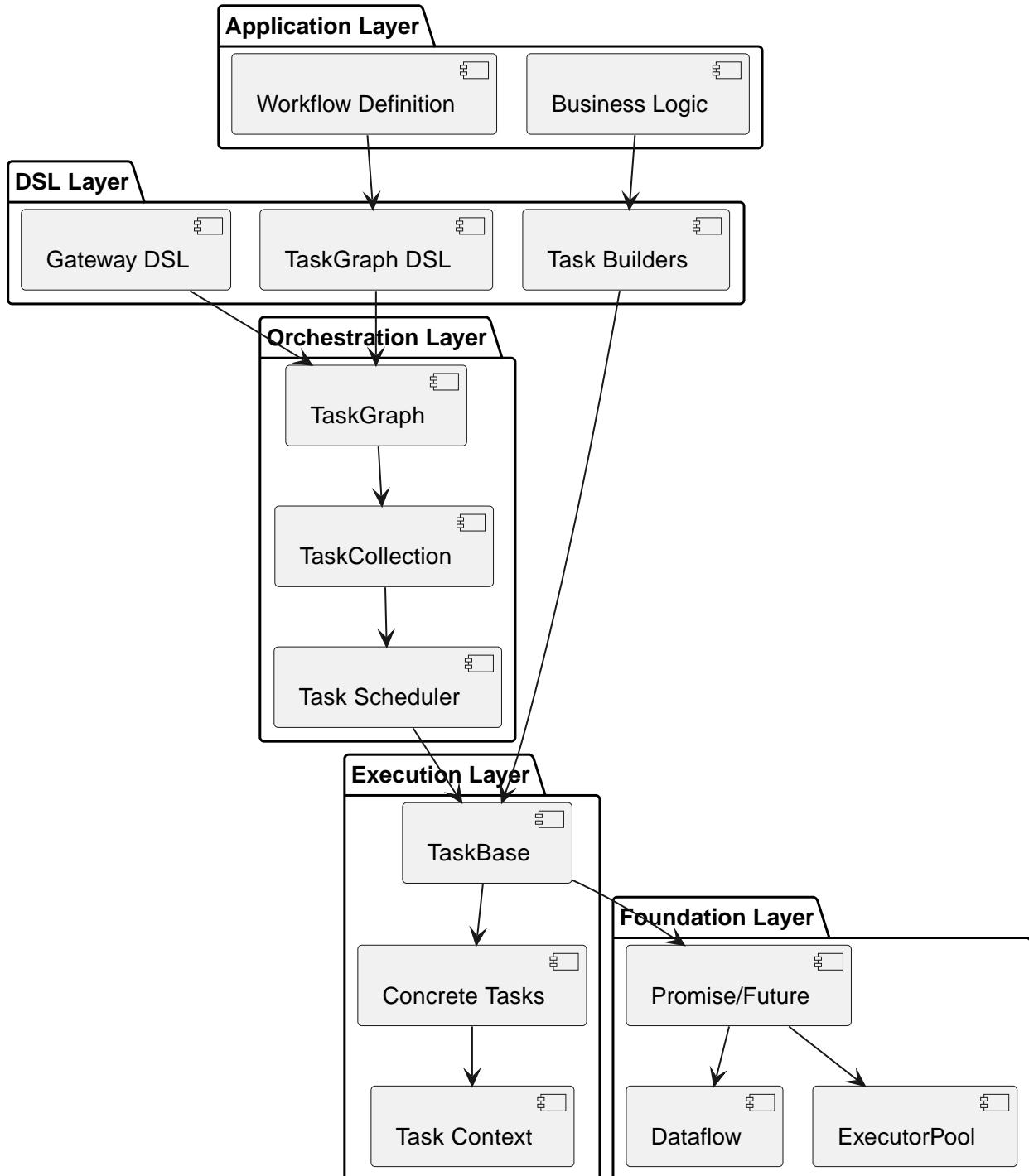
3.4. Composability First

Workflows are composed from reusable building blocks:

- Tasks are composable units
- TaskCollections group related tasks
- SubGraphs enable workflow reuse
- Gateways enable conditional logic

Chapter 4. Architectural Layers

TaskGraph is organized in distinct layers, each with clear responsibilities:



4.1. Layer 1: Foundation

The foundation layer provides core concurrency primitives:

ExecutorPool ([org.softwood.pool](#))

Managed thread pools with configurable concurrency limits.

Promise/Future ([org.softwood.promise](#))

Non-blocking asynchronous execution with composable operations.

Dataflow ([org.softwood.dataflow](#))

Single-assignment variables for coordinating concurrent tasks.

```
// Foundation primitives used internally
def pool = ExecutorPoolFactory.defaultPool()
def promise = Promises.task(pool) {
    // Async work
}
promise.then { result ->
    // Handle result
}
```

4.2. Layer 2: Execution

The execution layer implements task lifecycle and execution:

TaskBase

Abstract base class providing cross-cutting concerns:

- Retry logic
- Circuit breaker
- Timeout handling
- Idempotency
- Event emission
- Error handling

Concrete Task Types

Specific implementations for common operations:

- [ScriptTask](#) - Execute scripts (sandboxed)
- [HttpTask](#) - HTTP/REST calls
- [SqlTask](#) - Database operations
- [FileTask](#) - File operations
- ... and 15+ more

TaskContext

Execution context providing:

- Variable bindings
- Configuration access
- Credential resolution

- Event dispatcher
- Shared state

4.3. Layer 3: Orchestration

The orchestration layer manages workflow execution:

TaskGraph

The main workflow container:

- Builds directed acyclic graph (DAG)
- Resolves dependencies
- Schedules task execution
- Manages lifecycle
- Emits graph-level events

TaskCollection

Groups related tasks:

- Logical grouping
- Shared configuration
- Collective operations

Task Scheduler

Coordinates task execution:

- Dependency resolution
- Parallel execution
- Resource management
- Failure handling

4.4. Layer 4: DSL

The DSL layer provides declarative workflow definition:

TaskGraph DSL

Fluent builder for workflow definition:

```
def workflow = TaskGraph.build {
    task("step1") { /* ... */ }
    task("step2") {
        dependsOn "step1"
    }
}
```

Task Builders

Type-safe builders for each task type:

```
httpTask("fetch-data") {  
    url "https://api.example.com/data"  
    method POST  
    json { /* request body */ }  
}
```

Gateway DSL

Decision and routing logic:

```
exclusiveGateway("route-by-status") {  
    condition { ctx -> ctx.status == "active" }  
    whenTrue { task("process-active") }  
    whenFalse { task("process-inactive") }  
}
```

4.5. Layer 5: Application

The application layer is your workflow logic:

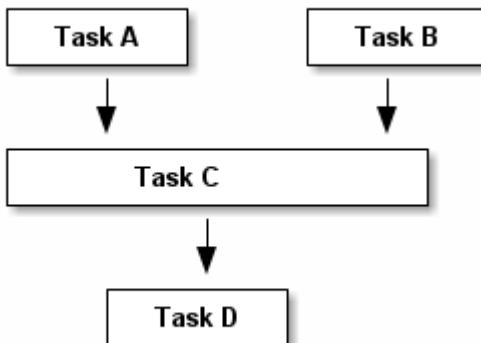
- Business rules
- Data transformations
- Integration points
- Custom task implementations

Chapter 5. Core Concepts

5.1. Directed Acyclic Graph (DAG)

TaskGraph workflows are represented as DAGs:

- **Nodes** are tasks
- **Edges** are dependencies
- **No cycles** ensures termination
- **Multiple roots** supported (parallel entry points)



5.2. Task Dependencies

Dependencies are explicitly declared:

```
task("A") { /* ... */ }
task("B") { /* ... */ }
task("C") {
    dependsOn "A", "B" // C runs after A and B complete
}
```

Dependency types:

- **Sequential** - `dependsOn` creates ordering
- **Parallel** - Tasks without dependencies run concurrently
- **Conditional** - Gateways enable dynamic dependencies

5.3. Promise-Based Execution

Tasks return promises for asynchronous execution:

```

def taskPromise = task.start() // Returns Promise<Result>

taskPromise.then { result ->
    // Handle success
}.onError { error ->
    // Handle failure
}

def finalResult = taskPromise.get() // Block until complete

```

Benefits:

- **Non-blocking** - Efficient resource utilization
- **Composable** - Chain operations with `then`, `map`, etc.
- **Error propagation** - Automatic error handling
- **Cancellation** - Support for cancelling work

5.4. Event-Driven Observability

TaskGraph emits events at every lifecycle stage:

```

workflow.addListener(new GraphEventListener() {
    void onEvent(GraphEvent event) {
        switch (event.type) {
            case TASK_STARTED:
                log.info("Task started: ${event.taskEvent.taskName}")
                break
            case TASK_COMPLETED:
                log.info("Task completed in ${event.taskEvent.executionTimeMs}ms")
                break
        }
    }
})

```

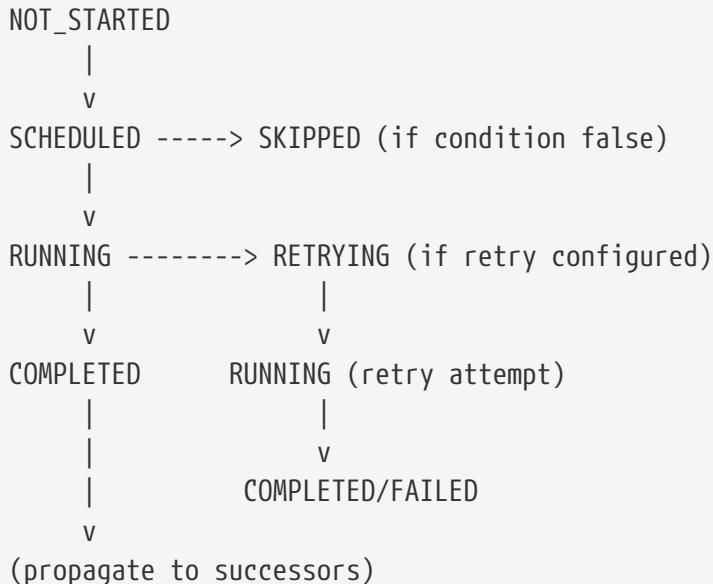
Event types:

- Graph lifecycle (started, completed, failed)
- Task lifecycle (scheduled, started, completed, failed)
- Resilience events (circuit opened, rate limited, retrying)
- Resource events (warnings, exhaustion)

Chapter 6. Execution Model

6.1. Task Lifecycle

Each task progresses through defined states:



6.2. Concurrency Control

TaskGraph provides multiple levels of concurrency control:

Graph-level:

```
def graph = TaskGraph.build {
    maxConcurrency 10 // Limit total concurrent tasks
}
```

Task-level:

```
task("intensive-work") {
    concurrencyLimit 2 // Max 2 instances concurrently
}
```

Resource-based:

```
task("db-query") {
    resource "database-pool", max: 5 // Semaphore on shared resource
}
```

6.3. Error Handling

Multi-layered error handling approach:

1. Task-level retry:

```
task("flaky-operation") {  
    retryPolicy {  
        maxAttempts 3  
        delay 1000  
        exponentialBackoff true  
    }  
}
```

2. Circuit breaker:

```
task("external-api") {  
    circuitBreaker {  
        failureThreshold 5  
        timeout 3000  
        halfOpenAfter 60000  
    }  
}
```

3. Fallback tasks:

```
task("primary") { /* ... */ }  
task("fallback") { /* ... */ }  
  
exclusiveGateway("try-primary") {  
    try { task("primary") }  
    onError { task("fallback") }  
}
```

4. Graph-level handling:

```
def result = workflow.start()  
    .onError { error ->  
        // Handle graph-level failure  
        notifyOps(error)  
        rollback()  
    }  
    .get()
```

Chapter 7. Design Patterns

7.1. Fork-Join Pattern

Execute tasks in parallel and join results:

```
def workflow = TaskGraph.build {
    task("fetch-users") { /* ... */ }
    task("fetch-orders") { /* ... */ }
    task("fetch-products") { /* ... */ }

    task("aggregate") {
        dependsOn "fetch-users", "fetch-orders", "fetch-products"
        action { ctx ->
            def users = ctx.taskResult("fetch-users")
            def orders = ctx.taskResult("fetch-orders")
            def products = ctx.taskResult("fetch-products")
            return merge(users, orders, products)
        }
    }
}
```

7.2. Pipeline Pattern

Chain transformations sequentially:

```
def workflow = TaskGraph.build {
    task("extract") { /* read data */ }
    task("transform") {
        dependsOn "extract"
        action { ctx -> transform(ctx.prev) }
    }
    task("load") {
        dependsOn "transform"
        action { ctx -> load(ctx.prev) }
    }
}
```

7.3. Saga Pattern

Distributed transaction with compensating actions:

```
def workflow = TaskGraph.build {
    saga("booking-saga") {
        step("reserve-flight") {
```

```

        action { /* reserve */ }
        compensate { /* cancel reservation */ }
    }
    step("charge-card") {
        action { /* charge */ }
        compensate { /* refund */ }
    }
    step("confirm-booking") {
        action { /* confirm */ }
        compensate { /* cancel */ }
    }
}
}

```

7.4. Conditional Routing

Route based on runtime conditions:

```

exclusiveGateway("check-priority") {
    condition { ctx -> ctx.global("priority") == "high" }
    whenTrue {
        task("fast-path")
    }
    whenFalse {
        task("normal-path")
    }
}

```

Chapter 8. Performance Considerations

8.1. Graph Construction

- **Lightweight** - Graph construction is fast, $O(n)$ where $n = \text{tasks}$
- **Cached** - Task instances reused across executions
- **Immutable** - Graph structure immutable after build

8.2. Task Execution

- **Parallel by default** - Tasks run concurrently when possible
- **Resource-aware** - Respects concurrency limits and semaphores
- **Lazy evaluation** - Tasks only execute when dependencies met

8.3. Memory Management

- **Streaming** - Large datasets processed in chunks
- **Cleanup** - Task results cleared after downstream tasks complete
- **Configurable** - Result retention policy customizable

```
task("process-large-file") {
    streaming true // Process in chunks
    cleanupResults true // Clear results after use
}
```

Chapter 9. Extensibility Points

TaskGraph provides multiple extension mechanisms:

9.1. Custom Task Types

Extend `TaskBase` for custom behavior:

```
class MyCustomTask extends TaskBase<MyResult> {
    @Override
    protected MyResult performTask(Object previousValue) {
        // Custom logic
    }
}
```

9.2. Custom Security Models

Provide custom script base classes:

```
scriptTask("secure-script") {
    customScriptBaseClass MyCompanyScriptBase
    // Your security model enforced
}
```

9.3. Custom Event Listeners

Implement observability integrations:

```
class PrometheusListener implements GraphEventListener {
    void onEvent(GraphEvent event) {
        // Emit Prometheus metrics
    }
}
```

9.4. SPI Implementations

Provide alternative implementations:

- Database providers (`SqlProvider`, `NoSqlProvider`)
- Messaging providers (`MessagingProvider`)
- Storage providers (`ObjectStoreProvider`)
- Serialization providers

Chapter 10. Summary

TaskGraph's architecture provides:

- **Type-safe workflows** - Compile-time checking
- **Asynchronous execution** - Efficient resource use
- **Composable design** - Build complex from simple
- **Secure by default** - Production-ready security
- **Observable** - Event-driven monitoring
- **Extensible** - SPI for custom needs
- **Testable** - Built-in test harness

The layered architecture ensures separation of concerns while maintaining flexibility for extension and customization.

Chapter 11. Next Steps

- **Chapter 2** - Learn why we chose a DSL approach
- **Chapter 3** - Deep dive into Pool and Promise foundations
- **Chapter 4** - Understand the Task/Collection/Graph layers

DSL Approach

This chapter explains why TaskGraph uses a Groovy DSL rather than XML configuration or annotations, and demonstrates the power and flexibility this provides.

Chapter 12. Why DSL Over XML/Annotations?

TaskGraph chose a Groovy-based DSL approach for workflow definition. Let's explore why this was the right choice.

12.1. The Problem with XML Configuration

Traditional workflow frameworks often use XML configuration:

```
<!-- ☐ XML Configuration - Verbose and Error-Prone -->
<workflow id="data-pipeline">
    <task id="extract" type="http">
        <property name="url" value="https://api.example.com/data"/>
        <property name="method" value="GET"/>
        <dependency ref="validate-credentials"/>
    </task>
    <task id="transform" type="script">
        <property name="language" value="groovy"/>
        <property name="script"><![CDATA[
            data.collect { it.value * 2 }
        ]]></property>
        <dependency ref="extract"/>
    </task>
    <task id="load" type="database">
        <property name="query" value="INSERT INTO results VALUES (?)"/>
        <dependency ref="transform"/>
    </task>
</workflow>
```

Problems with XML:

- ☐ **Verbose** - 20+ lines for simple workflow
- ☐ **No type safety** - Errors only at runtime
- ☐ **No IDE support** - No autocomplete, no refactoring
- ☐ **Not programmable** - Can't use variables, loops, functions
- ☐ **Hard to test** - Requires parsing XML in tests
- ☐ **Poor readability** - Obscured by tags and attributes

12.2. The Problem with Annotations

Some frameworks use Java annotations:

```
// ☐ Annotation Configuration - Rigid and Limited
```

```

@Workflow(id = "data-pipeline")
public class DataPipeline {

    @Task(id = "extract", type = "http")
    @HttpConfig(url = "https://api.example.com/data", method = "GET")
    @DependsOn("validate-credentials")
    public HttpResult extract() {
        // Implementation
    }

    @Task(id = "transform", type = "script")
    @DependsOn("extract")
    public List<Data> transform(HttpResult input) {
        // Limited to Java - no dynamic scripting
    }

    @Task(id = "load", type = "database")
    @DependsOn("transform")
    public void load(List<Data> data) {
        // Implementation
    }
}

```

Problems with Annotations:

- **Inflexible** - Configuration in code, can't change at runtime
- **Verbose** - Lots of boilerplate
- **Limited expressiveness** - Annotations have syntax limitations
- **Tight coupling** - Workflow structure tied to class structure
- **Hard to compose** - Can't easily build from smaller pieces

12.3. The DSL Advantage

TaskGraph uses a Groovy DSL that combines the best of both worlds:

```

// □ DSL - Concise, Type-Safe, and Powerful
def workflow = TaskGraph.build {
    httpTask("extract") {
        url "https://api.example.com/data"
        method GET
        dependsOn "validate-credentials"
    }

    scriptTask("transform") {
        dependsOn "extract"
        script '''
            data.collect { it.value * 2 }
        '''
    }
}

```

```
    ...  
}  
  
sqlTask("load") {  
    dependsOn "transform"  
    query "INSERT INTO results VALUES (?)"  
}  
}
```

Advantages:

- **Concise** - 15 lines vs 30+ in XML
- **Type-safe** - Compile-time checking
- **IDE-friendly** - Autocomplete, refactoring, navigation
- **Programmable** - Full Groovy language available
- **Testable** - Easy to test in unit tests
- **Readable** - Clear intent, minimal noise

Chapter 13. DSL Design Principles

13.1. Principle 1: Fluent and Readable

The DSL reads like natural language:

```
task("send-email") {  
    to "user@example.com"  
    subject "Welcome!"  
    body "Thank you for signing up"  
    sendWhen { ctx -> ctx.global("email-verified") }  
}
```

Reading aloud: "Task 'send-email' sends to 'user@example.com' with subject 'Welcome!' and body 'Thank you...' when email is verified."

13.2. Principle 2: Hierarchical Structure

The DSL reflects the natural hierarchy:

```
TaskGraph.build {  
    task("parent") {  
        retryPolicy {  
            maxAttempts 3  
            delay 1000  
        }  
    }  
}
```

Each level has appropriate scope and configuration options.

13.3. Principle 3: Sensible Defaults

Most configuration is optional with sensible defaults:

```
// Minimal configuration - uses defaults  
httpTask("fetch") {  
    url "https://api.example.com/data"  
    // Defaults: method=GET, timeout=30s, retries=0  
}  
  
// Full configuration - override defaults  
httpTask("fetch") {  
    url "https://api.example.com/data"  
    method POST
```

```

    timeout 6000
    retryPolicy {
        maxAttempts 3
        delay 1000
    }
}

```

13.4. Principle 4: Type Safety

The DSL leverages Groovy's static typing:

```

httpTask("api-call") {
    url "https://api.example.com" // □ String
    method POST // □ Enum (HttpMethod.POST)
    timeout 30000 // □ Integer (milliseconds)

    // url 12345 // □ Compile error: wrong type
    // method "INVALID" // □ Compile error: invalid method
}

```

IDE provides autocomplete and type checking:

[ide autocomplete] | *images/ide-autocomplete.png*

13.5. Principle 5: Composability

DSL elements compose naturally:

```

// Define reusable pieces
def retryConfig = {
    maxAttempts 3
    delay 1000
    exponentialBackoff true
}

def securityHeaders = {
    header "Authorization", "Bearer ${token}"
    header "X-API-Key", apiKey
}

// Compose into tasks
httpTask("api-call-1") {
    url "https://api1.example.com"
    retryPolicy(retryConfig)
    headers(securityHeaders)
}

```

```
httpTask("api-call-2") {  
    url "https://api2.example.com"  
    retryPolicy(retryConfig)  
    headers(securityHeaders)  
}
```

Chapter 14. DSL Components

14.1. Graph-Level DSL

The top level defines the workflow structure:

```
def workflow = TaskGraph.build {  
    // Graph configuration  
    maxConcurrency 10  
    id "my-workflow"  
    failFast true  
  
    // Task definitions  
    task("step1") { /* ... */ }  
    task("step2") { /* ... */ }  
  
    // Add listeners  
    addListener(myListener)  
}
```

Available at graph level:

- `maxConcurrency` - Limit concurrent tasks
- `id` - Workflow identifier
- `failFast` - Stop on first failure
- `addListener` - Register event listeners
- `task()` - Define tasks
- Gateway builders (covered in Chapter 12)

14.2. Task-Level DSL

Each task type has specific DSL methods:

```
httpTask("api-call") {  
    // HTTP-specific  
    url "https://api.example.com"  
    method POST  
    json { name: "Alice", age: 30 }  
    auth {  
        bearer token  
    }  
  
    // Common to all tasks  
    dependsOn "previous-task"  
    retryPolicy { /* ... */ }
```

```
    timeout 30000
    concurrencyLimit 5
}
```

Common task configuration:

- `dependsOn` - Task dependencies
- `retryPolicy` - Retry configuration
- `circuitBreaker` - Circuit breaker config
- `timeout` - Task timeout
- `concurrencyLimit` - Concurrent instances
- `resource` - Shared resource semaphore
- `condition` - Conditional execution
- `idempotencyPolicy` - Idempotency config

Task-specific configuration:

- HttpTask: `url, method, headers, auth, json, formData`
- SqlTask: `query, parameters, datasource`
- ScriptTask: `language, script, bindings, sandboxed`
- FileTask: `sources, operation, securityConfig`

14.3. Closure-Based Configuration

Many DSL methods accept closures for complex configuration:

```
task("complex") {
    // Closure for retry policy
    retryPolicy {
        maxAttempts 3
        delay 1000
        exponentialBackoff true
        retryOn(IOException.class, TimeoutException.class)
    }

    // Closure for circuit breaker
    circuitBreaker {
        failureThreshold 5
        timeout 30000
        halfOpenAfter 60000
    }

    // Closure for action
    action { ctx ->
        def input = ctx.prev
```

```
    return processData(input)
}
}
```

14.4. Method Reference Support

The DSL supports method references for cleaner code:

```
class DataProcessor {
    def extract(ctx) { /* ... */ }
    def transform(ctx) { /* ... */ }
    def load(ctx) { /* ... */ }
}

def processor = new DataProcessor()

TaskGraph.build {
    task("extract", processor::extract)
    task("transform", processor::transform) {
        dependsOn "extract"
    }
    task("load", processor::load) {
        dependsOn "transform"
    }
}
```

Chapter 15. Dynamic Workflow Construction

The DSL is programmable - you can use Groovy's full power:

15.1. Loops and Iteration

```
def workflow = TaskGraph.build {
    // Generate tasks dynamically
    def regions = ['us-east', 'eu-west', 'ap-south']

    regions.each { region ->
        httpTask("fetch-${region}") {
            url "https://api.${region}.example.com/data"
        }
    }

    // Aggregate all results
    task("aggregate") {
        dependsOn regions.collect { "fetch-${it}" }
        action { ctx ->
            regions.collect { ctx.taskResult("fetch-${it}") }
        }
    }
}
```

15.2. Conditional Task Creation

```
def workflow = TaskGraph.build {
    task("process") { /* ... */ }

    // Add email notification only in production
    if (System.getenv("ENVIRONMENT") == "production") {
        mailTask("notify") {
            dependsOn "process"
            to "ops@example.com"
            subject "Processing completed"
        }
    }
}
```

15.3. Parameterized Workflows

```
def buildDataPipeline(String source, String destination, int parallelism) {
    return TaskGraph.build {
        maxConcurrency parallelism
```

```

task("extract-${source}") {
    // Extract from source
}

task("load-${destination}") {
    dependsOn "extract-${source}"
    // Load to destination
}
}

// Create multiple pipelines
def pipeline1 = buildDataPipeline("s3", "postgres", 10)
def pipeline2 = buildDataPipeline("kafka", "elasticsearch", 5)

```

15.4. Higher-Order Functions

```

// Reusable workflow builder
def withRetryAndCircuitBreaker(Closure taskDef) {
    return {
        taskDef.delegate = delegate
        taskDef()

        retryPolicy {
            maxAttempts 3
            delay 1000
        }
        circuitBreaker {
            failureThreshold 5
            timeout 30000
        }
    }
}

// Use the builder
TaskGraph.build {
    task("resilient-api-call", withRetryAndCircuitBreaker {
        httpTask {
            url "https://api.example.com"
        }
    })
}

```

Chapter 16. DSL Best Practices

16.1. Extract Common Configuration

Don't repeat yourself - extract common patterns:

```
// ☐ Repetitive
httpTask("api1") {
    auth { bearer token }
    header "Content-Type", "application/json"
    timeout 30000
    retryPolicy { maxAttempts 3; delay 1000 }
}

httpTask("api2") {
    auth { bearer token }
    header "Content-Type", "application/json"
    timeout 30000
    retryPolicy { maxAttempts 3; delay 1000 }
}

// ☐ Extracted common config
def apiDefaults = {
    auth { bearer token }
    header "Content-Type", "application/json"
    timeout 30000
    retryPolicy { maxAttempts 3; delay 1000 }
}

httpTask("api1") {
    url "https://api1.example.com"
    apiDefaults.delegate = delegate
    apiDefaults()
}

httpTask("api2") {
    url "https://api2.example.com"
    apiDefaults.delegate = delegate
    apiDefaults()
}
```

16.2. Use Descriptive Task IDs

Task IDs should describe what the task does:

```
// ☐ Poor task IDs
task("task1") { /* ... */ }
```

```

task("task2") { /* ... */ }
task("task3") { /* ... */ }

// □ Descriptive task IDs
task("validate-user-input") { /* ... */ }
task("fetch-customer-data") { /* ... */ }
task("calculate-invoice-total") { /* ... */ }

```

16.3. Group Related Tasks

Use logical grouping for clarity:

```

TaskGraph.build {
    // Data extraction phase
    task("extract-users") { /* ... */ }
    task("extract-orders") { /* ... */ }
    task("extract-products") { /* ... */ }

    // Transformation phase
    task("transform-users") {
        dependsOn "extract-users"
    }
    task("transform-orders") {
        dependsOn "extract-orders"
    }
    task("transform-products") {
        dependsOn "extract-products"
    }

    // Loading phase
    task("load-data-warehouse") {
        dependsOn "transform-users", "transform-orders", "transform-products"
    }
}

```

16.4. Leverage Type Safety

Use enums and constants instead of strings:

```

import static org.softwood.dag.task.HttpMethod.*

// □ Type-safe
httpTask("api-call") {
    method POST // Enum
}

// □ String-based (error-prone)

```

```
httpTask("api-call") {  
    method "POST" // Typos not caught: "PSOT", "post", etc.  
}
```

Chapter 17. DSL vs Programmatic API

TaskGraph provides both DSL and programmatic APIs:

17.1. DSL Approach (Recommended)

```
def workflow = TaskGraph.build {
    task("step1") { /* ... */ }
    task("step2") {
        dependsOn "step1"
    }
}
```

When to use: * Building workflows declaratively * Rapid prototyping * Configuration-driven workflows * Most use cases

17.2. Programmatic API

```
def workflow = new TaskGraph()
def task1 = new ServiceTask("step1", "Step 1", ctx)
def task2 = new ServiceTask("step2", "Step 2", ctx)
task2.addPredecessor(task1)
workflow.addTask(task1)
workflow.addTask(task2)
```

When to use: * Dynamic workflow generation at runtime * Complex conditional logic * Integration with external systems * Advanced customization

17.3. Hybrid Approach

Mix both styles as needed:

```
// Start with DSL
def workflow = TaskGraph.build {
    task("step1") { /* ... */ }
}

// Add tasks programmatically
if (needsExtraProcessing) {
    def extraTask = new CustomTask("extra", "Extra", ctx)
    workflow.addTask(extraTask)
    workflow.getDependencyGraph().addEdge("step1", "extra")
}
```

Chapter 18. Comparison Summary

Aspect	XML	Annotations	DSL
Verbosity	High	Medium	Low
Type Safety	None	Compile-time	Compile-time
IDE Support	Poor	Good	Excellent
Programmability	None	Limited	Full Groovy
Testability	Hard	Medium	Easy
Flexibility	Low	Low	High
Learning Curve	Easy	Medium	Easy
Runtime Changes	Possible	Not possible	Possible

Chapter 19. Summary

The DSL approach provides:

- **Concise syntax** - Less boilerplate
- **Type safety** - Compile-time checking
- **IDE support** - Autocomplete and refactoring
- **Programmable** - Full Groovy power
- **Testable** - Easy unit testing
- **Flexible** - Dynamic construction

This makes TaskGraph workflows easy to write, maintain, and evolve.

Chapter 20. Next Steps

- **Chapter 3** - Understand the foundation libraries (Pool & Promises)
- **Chapter 7** - See practical examples of DSL usage

Foundation Libraries: Pool & Promises

This chapter explores the foundational concurrency libraries that power TaskGraph: the ExecutorPool and Promise frameworks. Understanding these primitives is key to understanding TaskGraph's asynchronous execution model.

Chapter 21. Overview

TaskGraph is built on two fundamental concurrency libraries located in the `org.softwood.pool` and `org.softwood.promise` packages:

- **ExecutorPool** - Managed thread pools with lifecycle control
- **Promise/Future** - Asynchronous value containers with composition support

These libraries provide the low-level primitives that TaskGraph uses to orchestrate concurrent workflow execution.

Chapter 22. ExecutorPool Framework

The ExecutorPool framework (`org.softwood.pool.*`) provides managed thread pools with automatic resource management.

22.1. Virtual Threads by Default (Java 21+)

IMPORTANT: TaskGraph uses Java Virtual Threads by default - no configuration required!

TaskGraph leverages Java 21+ virtual threads as the natural, optimal choice for concurrent task execution:

```
// Default behavior - uses virtual threads automatically
def workflow = TaskGraph.build {
    task("my-task") {
        action { performWork() }
    }
}

// Internally uses virtual thread executor - nothing to configure!
workflow.start() // Tasks execute on virtual threads
```

Why Virtual Threads?

- **Lightweight** - Millions of virtual threads vs thousands of platform threads
- **No blocking penalty** - Blocking operations don't tie up OS threads
- **Natural scalability** - Perfect for I/O-heavy workflows
- **No tuning needed** - Works optimally out-of-the-box

The default pool uses virtual threads automatically:

```
// This is what TaskGraph does by default (you don't need to do this!)
def pool = ExecutorPoolFactory.defaultPool()
// Returns: Executors.newVirtualThreadPerTaskExecutor()
// Auto-detected when running on Java 21+

// Your workflows just work with virtual threads:
def workflow = TaskGraph.build {
    // All 1000 tasks run concurrently on virtual threads - no problem!
    (1..1000).each { i ->
        httpTask("api-call-${i}") {
            url "https://api.example.com/item/${i}"
        }
    }
}
```

22.2. ExecutorPoolFactory - Named Pool Instances

While virtual threads are the default, you can create custom named pool instances for specific use cases:

22.2.1. Default Pool (Virtual Threads)

```
// Get the default virtual thread pool
def pool = ExecutorPoolFactory.defaultPool()

// On Java 21+: Returns virtual thread executor
// On Java 17-20: Returns ForkJoinPool.commonPool()

// Use in TaskGraph (already the default!)
def workflow = TaskGraph.build {
    executorPool pool // Unnecessary - this is already the default

    task("my-task") { action { work() } }
}
```

22.2.2. Named Pool Instances

Create named pool instances for specific purposes:

```
// Create named pools for different workload types
class MyPools {
    // Virtual thread pool for I/O operations (recommended)
    static final ExecutorPool IO_POOL =
        ExecutorPoolFactory.newVirtualThreadPool("io-operations")

    // Virtual thread pool for API calls (recommended)
    static final ExecutorPool API_POOL =
        ExecutorPoolFactory.newVirtualThreadPool("api-calls")

    // Platform thread pool for CPU-intensive work (if needed)
    static final ExecutorPool CPU_POOL =
        ExecutorPoolFactory.newFixedThreadPool(
            Runtime.availableProcessors(),
            "cpu-intensive"
        )

    // Named scheduled pool for periodic tasks
    static final ExecutorPool SCHEDULER =
        ExecutorPoolFactory.newScheduledThreadPool(2, "scheduler")
}

// Use named pools in workflows
def workflow = TaskGraph.build {
```

```

httpTask("api-call") {
    executorPool MyPools.API_POOL // Use named API pool
    url "https://api.example.com/data"
}

task("cpu-work") {
    executorPool MyPools.CPU_POOL // Use named CPU pool
    action { cpuIntensiveCalculation() }
}
}

```

22.2.3. Creating Custom Named Pools

```

// Virtual thread pool (recommended for most use cases)
def ioPool = ExecutorPoolFactory.newVirtualThreadPool("database-operations")

// Platform thread pools (for specific needs)
def fixedPool = ExecutorPoolFactory.newFixedThreadPool(10, "fixed-workers")
def cachedPool = ExecutorPoolFactory.newCachedThreadPool("dynamic-workers")
def singlePool = ExecutorPoolFactory.newSingleThreadExecutor("sequential-processor")

// Scheduled pool
def scheduledPool = ExecutorPoolFactory.newScheduledThreadPool(5, "scheduled-tasks")
scheduledPool.schedule({ println "Delayed" }, 10, TimeUnit.SECONDS)
scheduledPool.scheduleAtFixedRate({ println "Periodic" }, 0, 5, TimeUnit.SECONDS)

// All pools implement ExecutorPool interface
interface ExecutorPool {
    String getName() // Pool name for monitoring
    Future<?> submit(Runnable task)
    <T> Future<T> submit(Callable<T> task)

    // Lifecycle
    void shutdown()
    List<Runnable> shutdownNow()
    boolean awaitTermination(long timeout, TimeUnit unit)

    // Statistics
    int getActiveCount()
    int getPoolSize()
    long getCompletedTaskCount()
}

```

22.3. When to Use Custom Pools

Use the default virtual thread pool (already configured) when:

- * ☐ Tasks perform I/O operations (HTTP, database, file operations)
- * ☐ Tasks block on external resources
- * ☐ You have many concurrent tasks (hundreds to millions)
- * ☐ You want zero-configuration optimal performance

Consider custom named pools for:

- * **CPU-intensive work** - Fixed platform thread pool sized to CPU cores
- * **Sequential processing** - Single thread executor for ordered execution
- * **Scheduled tasks** - Scheduled thread pool for cron-like operations
- * **Resource isolation** - Separate pools for different subsystems
- * **Monitoring** - Named pools for better observability

```
// Example: Different pools for different workloads
def workflow = TaskGraph.build {
    // I/O tasks use default virtual threads (optimal)
    httpTask("fetch-data") {
        // No executorPool specified - uses default virtual threads
        url "https://api.example.com/data"
    }

    sqlTask("query-db") {
        // No executorPool specified - uses default virtual threads
        query "SELECT * FROM users"
    }

    // CPU-intensive task uses platform thread pool
    task("crunch-numbers") {
        executorPool MyPools.CPU_POOL // Custom platform thread pool
        action {
            // Computation-heavy work
            complexMathematicalCalculation()
        }
    }

    // Ordered processing uses single thread executor
    task("sequential") {
        executorPool MyPools.SEQUENTIAL_POOL
        action {
            // Must be processed in order
            processInOrder()
        }
    }
}
```

22.4. Pool Configuration (Advanced)

For the rare cases where you need custom platform thread pool configuration:

```
import org.softwood.pool.*

// Custom platform thread pool configuration
// (Most users should use virtual threads instead!)
def config = PoolConfig.builder()
    .poolName("custom-platform-pool")
    .corePoolSize(10)
```

```

    .maxPoolSize(50)
    .keepAliveTime(60, TimeUnit.SECONDS)
    .queueCapacity(1000)
    .threadNamePrefix("workflow-")
    .rejectionPolicy(RejectionPolicy.CALLER_RUNS)
    .build()

def customPool = ExecutorPoolFactory.create(config)

// Use in specific tasks that need platform threads
task("legacy-blocking-io") {
    executorPool customPool
    action { legacyBlockingOperation() }
}

```

Configuration options for platform thread pools:

- `poolName` - Named identifier for monitoring
- `corePoolSize` - Minimum threads kept alive
- `maxPoolSize` - Maximum threads allowed
- `keepAliveTime` - How long excess threads stay alive
- `queueCapacity` - Bounded queue size (0 = unbounded)
- `threadNamePrefix` - Name pattern for threads
- `rejectionPolicy` - What to do when queue is full

Rejection Policies:

Policy	Behavior
ABORT	Throw <code>RejectedExecutionException</code> (default)
CALLER_RUNS	Execute in the calling thread (provides backpressure)
DISCARD	Silently discard the task
DISCARD_OLDEST	Discard oldest queued task, retry



Virtual thread pools don't need these configurations - they scale automatically!

22.5. Pool Usage in TaskGraph

TaskGraph uses the default virtual thread pool automatically, but you can override at different levels:

22.5.1. Default Behavior (Recommended)

Just use TaskGraph - virtual threads work automatically:

```
// This is all you need - virtual threads are used automatically!
def workflow = TaskGraph.build {
    task("step1") { action { fetchData() } }
    task("step2") { action { processData() } }
}

// Tasks run on virtual threads - perfect for I/O and blocking operations
workflow.start()
```

22.5.2. Graph-Level Pool Override

Override the pool for an entire graph (rarely needed):

```
// Only if you have specific requirements
def namedPool = ExecutorPoolFactory.newVirtualThreadPool("my-workflow-pool")

def workflow = TaskGraph.build {
    executorPool namedPool // All tasks use this pool

    task("step1") { action { work1() } }
    task("step2") { action { work2() } }
}
```

22.5.3. Task-Level Pool Override

Individual tasks can use different named pools:

```
// Create named pools for different purposes
def ioPool = ExecutorPoolFactory.newVirtualThreadPool("io-operations")
def cpuPool = ExecutorPoolFactory.newFixedThreadPool(
    Runtime.availableProcessors(),
    "cpu-intensive"
)

TaskGraph.build {
    // I/O task uses named virtual thread pool
    task("io-bound") {
        executorPool ioPool // Named virtual thread pool
        action { performIO() }
    }

    // CPU task uses platform thread pool
    task("cpu-bound") {
        executorPool cpuPool // Platform thread pool for CPU work
        action { cpuIntensiveCalculation() }
    }
}
```

```

// This task uses the default virtual thread pool
task("default") {
    action { normalWork() } // Uses default virtual threads
}

```

22.6. Pool Monitoring

Monitor named pool instances for observability:

```

// Create named pool for monitoring
def pool = ExecutorPoolFactory.newVirtualThreadPool("api-operations")

// Get pool name
println "Pool: ${pool.name}" // "api-operations"

// Get statistics (varies by pool type)
println "Active count: ${pool.activeCount}"
println "Completed tasks: ${pool.completedTaskCount}"

// For platform thread pools only:
def platformPool = ExecutorPoolFactory.newFixedThreadPool(10, "platform-pool")
println "Pool size: ${platformPool.poolSize}"
println "Queue size: ${platformPool.queue.size()}"

```

22.7. Best Practices

1. Use virtual threads by default (no configuration needed!):

```

// ☐ Best practice - just use TaskGraph
def workflow = TaskGraph.build {
    task("my-task") { action { work() } }
}
// Virtual threads handle everything optimally!

```

2. Create named pools for monitoring and isolation:

```

// ☐ Good - named pools for observability
def apiPool = ExecutorPoolFactory.newVirtualThreadPool("external-api-calls")
def dbPool = ExecutorPoolFactory.newVirtualThreadPool("database-operations")
def scheduledPool = ExecutorPoolFactory.newScheduledThreadPool(2, "scheduled-tasks")

// Pool names appear in logs, metrics, and monitoring dashboards

```

3. Use platform thread pools only for CPU-intensive work:

```

// ☐ Good - platform threads for CPU work only
def cpuPool = ExecutorPoolFactory.newFixedThreadPool(
    Runtime.availableProcessors(),
    "cpu-intensive"
)

task("crunch-numbers") {
    executorPool cpuPool // CPU-bound work
    action { complexCalculation() }
}

// ☐ Bad - platform threads for I/O (use virtual threads instead!)
def badPool = ExecutorPoolFactory.newFixedThreadPool(100, "io-tasks")

```

4. Size platform thread pools appropriately (virtual threads don't need sizing):

- **Virtual threads** - No sizing needed! Use for I/O and blocking operations
- **CPU-bound platform threads** - `Runtime.availableProcessors()` or `CPUs + 1`
- **Scheduled tasks** - Small fixed number (2-5 threads)

5. Always shut down custom pools (virtual threads auto-clean up on JVM exit):

```

// Custom platform thread pool requires manual shutdown
def customPool = ExecutorPoolFactory.newFixedThreadPool(10, "custom")
try {
    // Use pool
} finally {
    customPool.shutdown()
    customPool.awaitTermination(30, TimeUnit.SECONDS)
}

// Virtual thread pools automatically clean up
def vtPool = ExecutorPoolFactory.newVirtualThreadPool("vt-pool")
// No shutdown needed - cleaned up on JVM exit

```

6. Don't worry about pool exhaustion with virtual threads:

```

// ☐ With virtual threads - no problem!
def workflow = TaskGraph.build {
    // 10,000 concurrent tasks? No problem with virtual threads!
    (1..10000).each { i ->
        httpTask("api-${i}") {
            url "https://api.example.com/item/${i}"
        }
    }
}

```

```
// ☺ With platform threads - would need careful sizing
// Don't do this - use virtual threads instead!
def platformPool = ExecutorPoolFactory.newFixedThreadPool(10000, "bad-idea")
```

Chapter 23. Promise Framework

The Promise framework (`org.softwood.promise.*`) provides asynchronous value containers with functional composition.

23.1. Core Concepts

23.1.1. Promise vs Future

Future (Java standard)

Blocking placeholder for a future value.

```
Future<String> future = executor.submit({ "Hello" })
String result = future.get() // Blocks thread
```

Promise (TaskGraph)

Non-blocking, composable async value.

```
Promise<String> promise = Promises.task(pool) { "Hello" }
promise.then { result ->
    println result // Non-blocking callback
}

// Virtual thread pools automatically clean up
def vtPool = ExecutorPoolFactory.newVirtualThreadPool("vt-pool")
// No shutdown needed - cleaned up on JVM exit
```

23.2. Creating Promises

23.2.1. Task Promises

Execute async work in a pool:

```
import org.softwood.promise.Promises

def pool = ExecutorPoolFactory.defaultPool()

def promise = Promises.task(pool) {
    // Async work
    Thread.sleep(1000)
    return "Result"
}
```

23.2.2. Completed Promises

Return an already-resolved value:

```
// Success
def success = Promises.successful("immediate value")

// Failure
def failure = Promises.failed(new RuntimeException("error"))
```

23.2.3. Deferred Promises

Manual control over completion:

```
def deferred = Promises.deferred()

// Complete later
Thread.start {
    Thread.sleep(1000)
    deferred.resolve("value")
    // or: deferred.reject(error)
}

def promise = deferred.promise
```

23.3. Promise Composition

23.3.1. Sequential Composition

Chain operations with `then`:

```
Promises.task(pool) {
    fetchUser(123)
}.then { user ->
    fetchOrders(user.id)
}.then { orders ->
    calculateTotal(orders)
}.then { total ->
    println "Total: $total"
}
```

23.3.2. Transformation

Transform values with `map`:

```

Promises.task(pool) {
    fetchData()
}.map { data ->
    data.toUpperCase() // Transform
}.map { upper ->
    upper.size() // Transform again
}

```

23.3.3. Error Recovery

Handle errors with `recover` or `fallbackTo`:

```

// Recover from error
Promises.task(pool) {
    riskyOperation()
}.recover { error ->
    log.error("Failed: $error")
    return "fallback value"
}

// Fallback to alternative promise
def primary = Promises.task(pool) { fetchFromPrimary() }
def secondary = Promises.task(pool) { fetchFromSecondary() }

primary.fallbackTo(secondary)

```

23.3.4. Parallel Composition

Wait for multiple promises:

```

def p1 = Promises.task(pool) { fetchUsers() }
def p2 = Promises.task(pool) { fetchOrders() }
def p3 = Promises.task(pool) { fetchProducts() }

// All must succeed
Promises.all([p1, p2, p3]).then { results ->
    def (users, orders, products) = results
    println "Fetched: ${users.size()} users, ${orders.size()} orders"
}

// First to complete
Promises.race([p1, p2, p3]).then { first ->
    println "First result: $first"
}

// All settled (success or failure)
Promises.allSettled([p1, p2, p3]).then { results ->

```

```

results.each { result ->
    if (result.isSuccess()) {
        println "Success: ${result.value}"
    } else {
        println "Failed: ${result.error}"
    }
}
}

```

23.4. Promise Operations

23.4.1. Filter

Filter based on predicate:

```

Promises.task(pool) {
    fetchNumber()
}.filter { n ->
    n > 0 // Only accept positive numbers
}.then { positive ->
    println "Positive: $positive"
}.recover { error ->
    println "Filtered out or error"
}

```

23.4.2. Zip

Combine two promises:

```

def p1 = Promises.task(pool) { fetchUser() }
def p2 = Promises.task(pool) { fetchProfile() }

p1.zip(p2) { user, profile ->
    return [user: user, profile: profile]
}.then { combined ->
    println combined
}

```

23.4.3. FlatMap

Chain async operations:

```

Promises.task(pool) {
    fetchUserId()
}.flatMap { userId ->
    // Return another promise
}

```

```

    Promises.task(pool) {
        fetchUser(userId)
    }
}.then { user ->
    println user
}

```

23.4.4. OnComplete

Execute side effects (success or failure):

```

promise
    .onComplete { result ->
        println "Completed: $result"
    }
    .onSuccess { value ->
        println "Success: $value"
    }
    .onError { error ->
        log.error("Error: $error")
    }

```

23.5. Timeout and Cancellation

23.5.1. Timeout

Fail promise if not completed in time:

```

Promises.task(pool) {
    slowOperation()
}.timeout(5, TimeUnit.SECONDS)
.recover { error ->
    if (error instanceof TimeoutException) {
        log.warn("Timed out after 5s")
        return "default"
    }
    throw error
}

```

23.5.2. Cancellation

Cancel ongoing work:

```

def promise = Promises.task(pool) {
    longRunningOperation()
}

```

```

// Cancel after 2 seconds
Thread.start {
    Thread.sleep(2000)
    promise.cancel()
}

promise.onError { error ->
    if (error instanceof CancellationException) {
        println "Cancelled"
    }
}

```

23.6. Promise Usage in TaskGraph

TaskGraph uses promises internally for all task execution:

23.6.1. Task Execution Returns Promise

```

def workflow = TaskGraph.build {
    task("step1") {
        action { "result" }
    }
}

// Start returns Promise<GraphResult>
def promise = workflow.start()

// Non-blocking callback
promise.then { result ->
    println "Workflow completed"
    println "Results: ${result.taskResults}"
}

// Or block if needed
def result = promise.get()

```

23.6.2. Individual Task Promises

```

def task = workflow.getTask("step1")
def taskPromise = task.start()

taskPromise.then { result ->
    println "Task result: $result"
}

```

23.6.3. Combining Task Promises

```
def task1 = workflow.getTask("fetch-users")
def task2 = workflow.getTask("fetch-orders")

def p1 = task1.start()
def p2 = task2.start()

Promises.all([p1, p2]).then { results ->
    println "Both completed: $results"
}
```

23.7. Best Practices

1. Avoid blocking operations:

```
// ☹ Bad - blocks thread
def result = promise.get()

// ☺ Good - non-blocking
promise.then { result ->
    // Handle result
}
```

2. Always handle errors:

```
promise
    .then { result -> /* ... */ }
    .onError { error -> log.error("Failed", error) }
```

3. Use composition over callbacks:

```
// ☹ Callback hell
promise.then { r1 ->
    Promises.task(pool) {
        process(r1)
    }.then { r2 ->
        Promises.task(pool) {
            save(r2)
        }.then { r3 ->
            println r3
        }
    }
}

// ☺ Composed
```

```
promise
    .flatMap { r1 -> Promises.task(pool) { process(r1) } }
    .flatMap { r2 -> Promises.task(pool) { save(r2) } }
    .then { r3 -> println r3 }
```

4. Set timeouts for external operations:

```
Promises.task(pool) {
    callExternalAPI()
}.timeout(30, TimeUnit.SECONDS)
```

Chapter 24. Dataflow Variables

The `org.softwood.dataflow` package provides single-assignment variables for coordinating concurrent tasks.

24.1. DataflowVariable

A variable that can be written once and read many times:

```
import org.softwood.dataflow.DataflowVariable

def var = new DataflowVariable()

// Write once
Thread.start {
    Thread.sleep(1000)
    var.bind("value") // Single assignment
}

// Read blocks until value available
println var.get() // Waits for binding
```

24.2. Usage Patterns

24.2.1. Producer-Consumer

```
def result = new DataflowVariable()

// Producer
Thread.start {
    def data = fetchData()
    result.bind(data)
}

// Multiple consumers
10.times { i ->
    Thread.start {
        def data = result.get() // All get same value
        process(data, i)
    }
}
```

24.2.2. Synchronization Point

```
def barrier = new DataflowVariable()
```

```

// Workers wait for signal
def workers = (1..10).collect { i ->
    Thread.start {
        barrier.get() // Wait for start signal
        doWork(i)
    }
}

// Coordinator starts work
Thread.sleep(1000)
barrier.bind("GO") // Release all workers

```

24.3. TaskGraph Usage

TaskGraph uses dataflow variables internally for dependency coordination:

```

// TaskGraph implementation (simplified)
class TaskBase {
    DataflowVariable result = new DataflowVariable()

    void execute() {
        // Wait for dependencies
        predecessors.each { pred ->
            pred.result.get() // Blocks until predecessor completes
        }

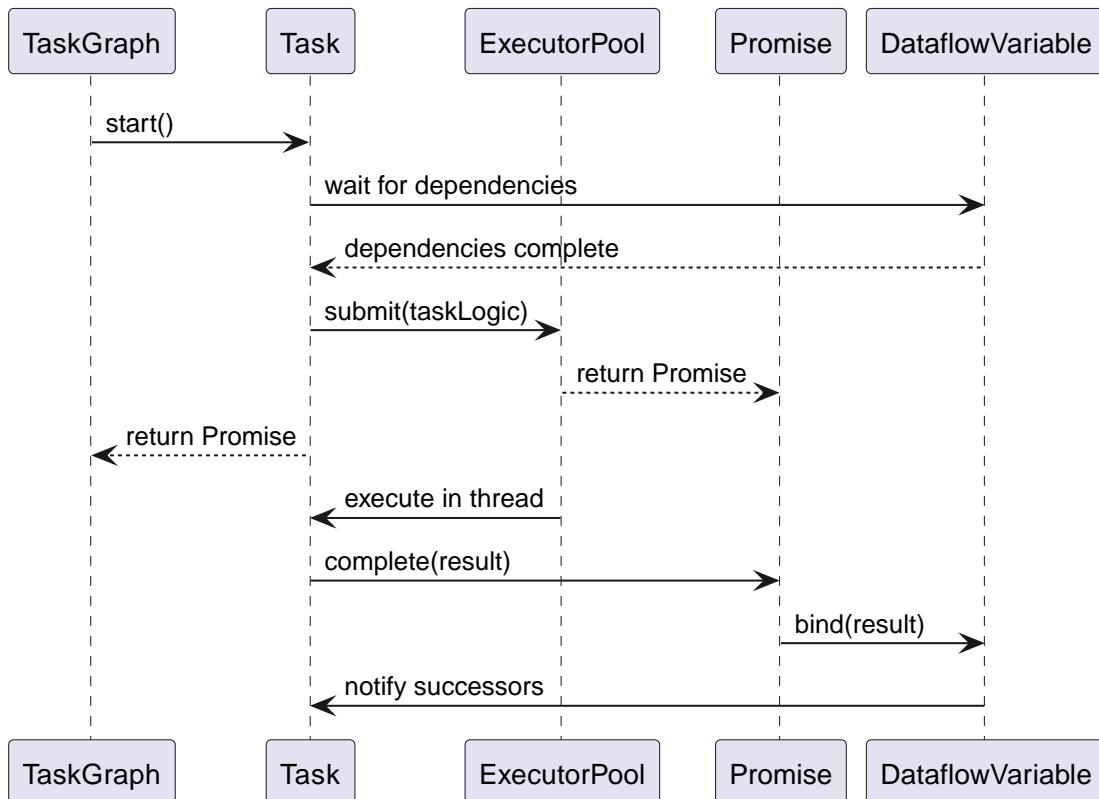
        // Execute task
        def value = performTask()

        // Publish result
        result.bind(value)
    }
}

```

Chapter 25. Integration: How It All Works Together

25.1. Task Execution Flow



25.2. Example: Complete Flow with Virtual Threads

```
// 1. No pool creation needed - virtual threads by default!
// 2. Define workflow - uses virtual threads and promises automatically
def workflow = TaskGraph.build {
    // No executorPool config needed!

    task("fetch") {
        action {
            // Executes on virtual thread, returns Promise
            fetchData()
        }
    }

    task("process") {
        dependsOn "fetch"
        action { ctx ->
            // Waits for "fetch" via DataflowVariable
            def data = ctx.prev
```

```
// Process and return Promise (on virtual thread)
processData(data)
    }
}
}

// 3. Start returns Promise<GraphResult>
def promise = workflow.start()

// 4. Compose with promise operations
promise
    .timeout(60, TimeUnit.SECONDS)
    .then { result ->
        println "Success: ${result.taskResults}"
    }
    .onError { error ->
        log.error("Failed", error)
    }

// Or block for result
def result = promise.get()
```

Chapter 26. Performance Considerations

26.1. Virtual Threads - No Pool Sizing Required!

Virtual threads scale automatically - no tuning needed:

```
// ☐ Perfect - handles any number of concurrent tasks
def workflow = TaskGraph.build {
    // 10,000 concurrent HTTP calls? No problem!
    (1..10000).each { i ->
        httpTask("api-$i") {
            url "https://api.example.com/item/$i"
        }
    }
}

// All tasks run concurrently on virtual threads
// No pool exhaustion, no careful sizing - it just works!
```

Virtual threads are optimal for: * HTTP/REST API calls * Database operations * File I/O * Message queue operations * Any blocking operations

26.2. Platform Thread Pool Sizing (CPU-Intensive Only)

Only needed for CPU-bound work:

```
// CPU-intensive: size to CPU cores
def cpuPool = ExecutorPoolFactory.newFixedThreadPool(
    Runtime.availableProcessors(),
    "cpu-work"
)

task("computation") {
    executorPool cpuPool
    action { cpuIntensiveCalculation() }
}
```

Don't use platform threads for I/O:

```
// ☐ Bad - wasteful, limits scalability
def badPool = ExecutorPoolFactory.newFixedThreadPool(1000, "io-tasks")

// ☐ Good - virtual threads for I/O (the default!)
def workflow = TaskGraph.build {
    // No pool config - uses virtual threads automatically
    httpTask("api") { url "..." }
```

```
}
```

26.3. Promise Overhead

Promises have minimal overhead:

- **Creation:** ~100 nanoseconds
- **Composition:** ~50 nanoseconds per operation
- **Memory:** ~200 bytes per promise

For very high-throughput scenarios (millions of ops/sec), consider pooling or reusing promises.

26.4. Dataflow Performance

DataflowVariable operations:

- **Bind:** O(1) time, notifies all waiters
- **Get (bound):** O(1) immediate return
- **Get (unbound):** Blocks until bound

Chapter 27. Debugging and Monitoring

27.1. Thread Dumps

Identify pool exhaustion:

```
// Get thread dump
def threadDump = Thread.allStackTraces
    .findAll { thread, stack ->
        thread.name.startsWith("workflow-")
    }

threadDump.each { thread, stack ->
    println "${thread.name} - ${thread.state}"
    stack.take(5).each { println " $it" }
}
```

27.2. Promise Tracing

Enable detailed promise logging:

```
System.setProperty("promise.trace", "true")

Promises.task(pool) {
    operation()
}.then { result ->
    // Logs: Promise[123] completed with: result
    result
}
```

27.3. Pool Metrics

Export pool metrics:

```
def pool = ExecutorPoolFactory.defaultPool()

def metrics = [
    active: pool.activeCount,
    poolSize: pool.poolSize,
    coreSize: pool.corePoolSize,
    maxSize: pool.maximumPoolSize,
    completed: pool.completedTaskCount,
    queueSize: pool.queue.size(),
    largestPoolSize: pool.largestPoolSize
]
```

```
println JsonOutput.toJson(metrics)
```

Chapter 28. Summary

The foundation libraries provide:

- **ExecutorPool** - Managed thread pools with lifecycle control
- **Promise** - Non-blocking async values with composition
- **Dataflow** - Single-assignment coordination primitives
- **Integration** - Seamless TaskGraph integration
- **Performance** - Low-overhead, high-throughput
- **Observability** - Built-in monitoring and debugging

These primitives are the building blocks that enable TaskGraph's powerful asynchronous orchestration capabilities.

Chapter 29. Next Steps

- **Chapter 4** - Learn how Tasks, TaskCollections, and TaskGraph build on these foundations
- **Chapter 6** - Understand how TaskContext provides execution context
- **Appendix B** - Performance tuning guidelines

Simple Task Execution: Tasks and TasksCollection

Before diving into the full TaskGraph orchestration framework, this chapter introduces two simpler abstractions for task execution: the lightweight `Tasks` utility for ad-hoc task execution, and `TasksCollection` for managing coordinated task sets with shared context. These provide gentle on-ramps to understanding task-based programming.

Chapter 30. Overview

The TaskGraph framework offers multiple levels of abstraction, each suited for different use cases:

Abstraction	Complexity	Use Case
Tasks (this chapter)	Minimal	Ad-hoc execution, simple pipelines, one-off operations
TasksCollection (this chapter)	Light	Coordinated task sets, event-driven systems, shared state
TaskGraph (later chapters)	Full	Complex workflows, dependencies, orchestration

This chapter covers the first two—lightweight alternatives that don't require graph dependencies or complex orchestration.

Chapter 31. Part 1: Tasks Utility

31.1. What is Tasks?

`Tasks` is a static utility class providing simple methods for executing independent tasks without TaskGraph overhead. Think of it as similar to `Dataflows` but for task execution.

Key Characteristics:

- **No graph structure** - Just execute tasks and combine results
- **Shared TaskContext** - Tasks can coordinate via a common context
- **Promise-based** - All operations return promises
- **Groovy DSL** - Clean, fluent syntax
- **Auto-wrapping** - Non-promise results automatically wrapped

31.2. Core Execution Patterns

31.2.1. Execute All Tasks (Parallel)

Wait for ALL tasks to complete and collect results:

```
import org.softwood.dag.Tasks

def results = Tasks.all { ctx ->
    task("fetch-users") {
        // Fetch user data
        [1, 2, 3]
    }

    task("fetch-orders") {
        // Fetch order data
        [101, 102, 103]
    }

    task("fetch-products") {
        // Fetch product data
        ["A", "B", "C"]
    }
}

// results = [[1, 2, 3], [101, 102, 103], ["A", "B", "C"]]
println "Users: ${results[0]}"
println "Orders: ${results[1]}"
println "Products: ${results[2]}
```

Behavior:

- All tasks execute in parallel
- Returns List of results in definition order
- Blocks until ALL tasks complete
- If any task fails, exception propagates

31.2.2. Race to First Result (Any)

Execute tasks and return the FIRST to complete:

```
def winner = Tasks.any { ctx ->
    task("primary-api") {
        sleep(100) // Slow primary
        "primary-data"
    }

    task("fallback-api") {
        sleep(10) // Fast fallback
        "fallback-data"
    }

    task("cache") {
        sleep(5) // Fastest
        "cached-data"
    }
}

// winner = "cached-data" (fastest)
println "Got result from: ${winner}"
```

Use Cases:

- Fallback strategies (try multiple sources)
- Timeout patterns (race with timeout task)
- Performance optimization (try multiple algorithms)

Note: Other tasks continue executing but results are ignored.

31.2.3. Sequential Pipeline (Sequence)

Chain tasks where each receives the previous result:

```
def result = Tasks.sequence { ctx ->
    task("parse") {
        "42" // String input
    }

    task("convert") { prev ->
```

```

        prev.toInt() // receives "42", returns 42
    }

    task("double") { prev ->
        prev * 2 // receives 42, returns 84
    }

    task("format") { prev ->
        "Result: $prev" // receives 84, returns "Result: 84"
    }
}

println result // "Result: 84"

```

Behavior:

- Tasks execute sequentially (one after another)
- Each task receives previous task's result as `prev`
- Final task's result is returned
- Short-circuits on first failure

31.2.4. Parallel Execution (Non-Blocking)

Start tasks in parallel and get promises immediately:

```

def promises = Tasks.parallel { ctx ->
    task("long-running-1") {
        sleep(1000)
        "Result 1"
    }

    task("long-running-2") {
        sleep(1000)
        "Result 2"
    }
}

// Do other work while tasks run...
println "Tasks started, doing other work..."

// Wait for results when needed
def results = promises.collect { it.get() }
println "All done: ${results}"

```

Difference from `all()`:

- `all()` - Blocks until complete, returns results

- **parallel()** - Returns promises immediately, non-blocking

31.3. Shared Context Execution

Execute tasks that share state via TaskContext globals:

```
def ctx = Tasks.withContext { ctx ->
    task("init") {
        ctx.globals.config = [timeout: 5000, retries: 3]
        ctx.globals.results = []
        "initialized"
    }

    task("work-1") {
        def config = ctx.globals.config
        ctx.globals.results << "work-1 done with timeout ${config.timeout}"
        "work-1"
    }

    task("work-2") {
        def config = ctx.globals.config
        ctx.globals.results << "work-2 done with ${config.retries} retries"
        "work-2"
    }
}

// Access shared context after execution
println ctx.globals.config
// [timeout: 5000, retries: 3]

println ctx.globals.results
// ["work-1 done with timeout 5000", "work-2 done with 3 retries"]
```

Use Cases:

- Accumulating results from multiple tasks
- Sharing configuration across tasks
- Collecting metrics or logs
- Coordinating via shared state

31.4. Single Task Execution

For simple one-off task execution:

31.4.1. Groovy Closure Style

```

def result = Tasks.execute { task ->
    task.action { ctx, prev ->
        // Your task logic
        def data = fetchData()
        processData(data)
    }
}

println result

```

Auto-Wrapping:

Non-Promise returns are automatically wrapped:

```

def result = Tasks.execute { task ->
    task.action { ctx, prev ->
        "Hello" // String automatically wrapped in Promise
    }
}

// result = "Hello"

```

31.4.2. Java Lambda Style

```

import java.util.function.Function

def result = Tasks.execute({ ServiceTask task ->
    task.action { ctx, prev -> "Result" }
    return task
} as Function)

println result

```

31.5. Task Closure Parameters

Task closures can accept 0, 1, or 2 parameters:

```

// No parameters - standalone task
task("independent") {
    fetchFromDatabase()
}

// One parameter - receives previous result
task("transform") { prev ->
    prev.toUpperCase()
}

```

```
// Two parameters - receives context and previous result
task("advanced") { ctx, prev ->
    def config = ctx.globals.config
    processWithConfig(prev, config)
}
```

The DSL automatically detects the closure's parameter count and calls it appropriately.

31.6. Complete Examples

31.6.1. Example 1: Multi-Source Data Aggregation

```
// Fetch from multiple sources in parallel, combine results
def data = Tasks.all { ctx ->
    task("fetch-db") {
        database.query("SELECT * FROM users")
    }

    task("fetch-api") {
        httpClient.get("https://api.example.com/users")
    }

    task("fetch-cache") {
        cache.get("users")
    }
}

def combined = [
    database: data[0],
    api: data[1],
    cache: data[2]
]

println "Fetched ${combined.database.size()} from DB"
println "Fetched ${combined.api.size()} from API"
println "Fetched ${combined.cache.size()} from cache"
```

31.6.2. Example 2: Fallback Strategy with Timeout

```
import java.util.concurrent.TimeoutException

def result = Tasks.any { ctx ->
    task("primary") {
        try {
            // Try primary source
            httpClient.get("https://primary.api.com/data")
        } catch (Exception e) {
```

```

        throw e // Let other tasks win
    }
}

task("secondary") {
    sleep(1000) // Wait before trying secondary
    httpClient.get("https://secondary.api.com/data")
}

task("timeout") {
    sleep(5000) // 5 second timeout
    throw new TimeoutException("All sources timed out")
}
}

println "Got data from fastest source: ${result}"

```

31.6.3. Example 3: ETL Pipeline

```

def result = Tasks.sequence { ctx ->
    task("extract") {
        println "Extracting data..."
        rawData = database.query("SELECT * FROM source")
        rawData
    }

    task("transform") { prev ->
        println "Transforming ${prev.size()} records..."
        prev.collect { record ->
            [
                id: record.id,
                name: record.name.toUpperCase(),
                processed: true,
                timestamp: System.currentTimeMillis()
            ]
        }
    }

    task("load") { prev ->
        println "Loading ${prev.size()} records..."
        database.batchInsert("destination", prev)
        prev.size()
    }
}

println "ETL complete: processed ${result} records"

```

31.6.4. Example 4: Shared State Coordination

```
def ctx = Tasks.withContext { ctx ->
    // Initialize shared state
    ctx.globals.total = 0
    ctx.globals.errors = []

    task("process-batch-1") {
        try {
            def count = processBatch(1)
            synchronized(ctx.globals) {
                ctx.globals.total += count
            }
        } catch (Exception e) {
            ctx.globals.errors << e
        }
    }

    task("process-batch-2") {
        try {
            def count = processBatch(2)
            synchronized(ctx.globals) {
                ctx.globals.total += count
            }
        } catch (Exception e) {
            ctx.globals.errors << e
        }
    }

    task("process-batch-3") {
        try {
            def count = processBatch(3)
            synchronized(ctx.globals) {
                ctx.globals.total += count
            }
        } catch (Exception e) {
            ctx.globals.errors << e
        }
    }

    println "Total processed: ${ctx.globals.total}"
    if (ctx.globals.errors) {
        println "Errors: ${ctx.globals.errors.size()}"
    }
}
```

31.7. When to Use Tasks

- Use Tasks when:

- Running a few independent tasks
- Simple parallel or sequential execution
- One-off operations or scripts
- Prototyping or testing task logic
- You don't need dependency management

□ **Don't use Tasks when:**

- Tasks have complex dependencies
- Need conditional routing (gateways)
- Require workflow orchestration
- Need execution graph visualization
- Want automatic retry/timeout per task

For those cases, use TaskGraph instead.

Chapter 32. Part 2: TasksCollection

32.1. What is TasksCollection?

`TasksCollection` is a lightweight registry and coordinator for tasks that need to share context but don't require graph dependencies.

Key Features:

- **Shared TaskContext** - All tasks coordinate via common context
- **Named registry** - Tasks discoverable by ID
- **Lifecycle management** - Start/stop tasks as a unit
- **Auto-start support** - Timers and business rules auto-start
- **Promise chaining** - Fluent API for task chains
- **Metrics** - Built-in task state tracking

Comparison:

Feature	Tasks	TasksCollection
Task registry	No	Yes (named lookup)
Shared context	Yes (implicit)	Yes (explicit)
Lifecycle	Execute once	Start/stop, long-running
Discovery	No	Yes (find by ID, type, filter)
Metrics	No	Yes (state counts, stats)
Chaining API	No	Yes (fluent chains)

32.2. Creating a TasksCollection

Use the static builder method:

```
import org.softwood.dag.TasksCollection

def tasks = TasksCollection.tasks {
    serviceTask("fetch") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                fetchDataFromAPI()
            }
        }
    }

    serviceTask("transform") {
        action { ctx, prev ->
```

```

        ctx.promiseFactory.executeAsync {
            transformData(prev)
        }
    }

    serviceTask("save") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                saveToDatabase(prev)
            }
        }
    }
}

println tasks // TasksCollection[tasks=3, running=false, active=0]

```

32.3. Task Registration Methods

32.3.1. Service Tasks

Standard executable tasks:

```

tasks.serviceTask("process-order") {
    action { ctx, prev ->
        ctx.promiseFactory.executeAsync {
            // Process order logic
            [orderId: prev.id, status: "processed"]
        }
    }
}

```

32.3.2. Timer Tasks

Tasks that execute on a schedule (auto-start):

```

import java.time.Duration

tasks.timer("heartbeat") {
    interval Duration.ofSeconds(10)
    action { ctx ->
        ctx.promiseFactory.executeAsync {
            println "Heartbeat: ${System.currentTimeMillis()}"
            sendHeartbeat()
        }
    }
}

```

```
// Auto-starts when collection starts
tasks.start()
```

32.3.3. Business Rule Tasks

Tasks that evaluate conditions and react (auto-start):

```
tasks.businessRule("fraud-check") {
    when { signal "transaction-received" }

    evaluate { ctx, data ->
        // Return true if rule passes, false if fails
        data.amount < 10000 && data.riskScore < 0.8
    }

    onTrue { ctx, data ->
        println "Transaction approved: ${data}"
        tasks.find("approve-transaction").execute(
            ctx.promiseFactory.createPromise(data)
        )
    }

    onFalse { ctx, data ->
        println "Transaction flagged: ${data}"
        tasks.find("flag-for-review").execute(
            ctx.promiseFactory.createPromise(data)
        )
    }
}
```

32.3.4. Subprocess Tasks (Call Activities)

Tasks that delegate to subworkflows:

```
tasks.callActivity("process-payment") {
    subworkflow { paymentWorkflow }

    inputMapper { parentContext, inputData ->
        // Map data to subprocess input
        [
            amount: inputData.total,
            paymentMethod: inputData.method
        ]
    }

    outputMapper { subResult ->
        // Map subprocess result back
    }
}
```

```
        [transactionId: subResult.txnId, success: subResult.status == "ok"]
    }
}
```

32.3.5. Generic Task Registration

Register any task type:

```
import org.softwood.dag.task.TaskType

tasks.task("custom", TaskType.SCRIPT) {
    script "println 'Hello from script'"
}
```

32.3.6. Register Existing Tasks

Add already-created tasks:

```
def myTask = TaskFactory.createServiceTask("existing", "label", ctx)
myTask.action { ctx, prev -> "result" }

tasks.register(myTask)
```

32.4. Task Discovery

32.4.1. Find by ID

```
def task = tasks.find("process-order")
if (task) {
    println "Found task: ${task.id}"
}
```

32.4.2. Find with Filter

```
import org.softwood.dag.task.TaskState

// Find all completed tasks
def completed = tasks.findAll { it.state == TaskState.COMPLETED }

// Find all service tasks
def serviceTasks = tasks.findAll { it instanceof ServiceTask }

// Find tasks with specific prefix
def apiTasks = tasks.findAll { it.id.startsWith("api-") }
```

32.4.3. Get All Task IDs

```
def ids = tasks.getTaskIds()
println "Registered tasks: ${ids}"
```

32.4.4. Check if Task Exists

```
if (tasks.contains("critical-task")) {
    println "Critical task is registered"
}
```

32.4.5. Get by Type

```
import org.softwood.dag.task.TimerTask

def timers = tasks.getTasksByType(TimerTask)
println "Found ${timers.size()} timer tasks"
```

32.5. Lifecycle Management

32.5.1. Start Collection

Starts all auto-start tasks (timers, business rules):

```
tasks.start()
println "TasksCollection started"

// Timers now ticking, business rules listening for signals
```

32.5.2. Stop Collection

Stops all running tasks:

```
tasks.stop()
println "TasksCollection stopped"

// Timers stopped, business rules deactivated
```

32.5.3. Clear Collection

Stop and remove all tasks:

```
tasks.clear()
println "TasksCollection cleared"

// All tasks removed, context cleared
```

32.6. Promise Chaining API

Create sequential execution chains with fluent API:

32.6.1. Basic Chain

```
tasks.chain("fetch", "transform", "save")
    .run()
    .get() // Wait for completion
```

32.6.2. Chain with Initial Value

```
def result = tasks.chain("validate", "process", "store")
    .run([userId: 123, action: "purchase"])
    .get()

println "Chain result: ${result}"
```

32.6.3. Chain with Handlers

```
tasks.chain("step1", "step2", "step3")
    .onComplete { result ->
        println "Chain completed successfully: ${result}"
    }
    .onError { error ->
        println "Chain failed: ${error.message}"
    }
    .run(initialData)
```

Handler Behavior:

- **onComplete** - Called when chain succeeds
- **onError** - Called when any task fails (error still propagates)

32.6.4. Async Chain Execution

Chains return promises, allowing async patterns:

```

// Start chain, don't wait
def promise = tasks.chain("long", "running", "chain")
    .onComplete { println "Done!" }
    .run()

// Do other work...
doOtherWork()

// Wait when needed
def result = promise.get()

```

32.7. Metrics and Monitoring

32.7.1. Task Counts

```

println "Total tasks: ${tasks.taskCount}"
println "Auto-start tasks: ${tasks.autoStartCount}"
println "Active tasks: ${tasks.activeCount}"
println "Completed tasks: ${tasks.completedCount}"
println "Failed tasks: ${tasks.failedCount}"

```

32.7.2. Task Count by State

```

import org.softwood.dag.task.TaskState

def scheduled = tasks.getTaskCountByState(TaskState.SCHEDULED)
def running = tasks.getTaskCountByState(TaskState.RUNNING)
def completed = tasks.getTaskCountByState(TaskState.COMPLETED)

println "Scheduled: $scheduled, Running: $running, Completed: $completed"

```

32.7.3. Active Tasks

```

def activeTasks = tasks.getActiveTasks()
activeTasks.each { task ->
    println "Active: ${task.id} (${task.state})"
}

```

32.7.4. Summary Statistics

```

def stats = tasks.getStats()
println stats
// [total: 10, autoStart: 2, running: 3, completed: 5,

```

```
// failed: 1, scheduled: 1, skipped: 0]
```

32.8. Complete Examples

32.8.1. Example 1: Event-Driven Monitoring System

```
import java.time.Duration
import org.softwood.dag.task.SignalTask

def monitoring = TasksCollection.tasks {

    // Timer sends heartbeat every 10 seconds
    timer("heartbeat-sender") {
        interval Duration.ofSeconds(10)
        action { ctx ->
            ctx.promiseFactory.executeAsync {
                def data = [
                    timestamp: System.currentTimeMillis(),
                    source: "monitoring"
                ]
                SignalTask.sendSignalGlobal("heartbeat", data)
                println "Heartbeat sent: ${data.timestamp}"
            }
        }
    }

    // Business rule monitors heartbeats
    businessRule("heartbeat-monitor") {
        when { signal "heartbeat" }

        evaluate { ctx, data ->
            // Check if heartbeat is recent enough
            def age = System.currentTimeMillis() - data.timestamp
            age < 15000 // Must be within 15 seconds
        }

        onTrue { ctx, data ->
            println "Heartbeat OK: ${data.timestamp}"
        }

        onFalse { ctx, data ->
            println "ALERT: Heartbeat too old!"
            find("send-alert").execute(
                ctx.promiseFactory.createPromise(data)
            )
        }
    }

    // Alert service task
}
```

```

    serviceTask("send-alert") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                sendAlert("Heartbeat timeout detected", prev)
                println "Alert sent"
            }
        }
    }

// Start the monitoring system
monitoring.start()
println "Monitoring system started"

// Run for a while...
Thread.sleep(60000)

// Stop monitoring
monitoring.stop()
println "Monitoring system stopped"

```

32.8.2. Example 2: Data Processing Pipeline

```

def pipeline = TasksCollection.tasks {

    serviceTask("extract") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                println "Extracting data..."
                database.query("SELECT * FROM raw_data WHERE processed = false")
            }
        }
    }

    serviceTask("validate") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                println "Validating ${prev.size()} records..."
                prev.findAll { it.isValid() }
            }
        }
    }

    serviceTask("transform") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                println "Transforming ${prev.size()} records..."
                prev.collect { record ->
                    [

```

```

        id: record.id,
        data: processData(record.data),
        processed_at: System.currentTimeMillis()
    ]
}
}
}

serviceTask("enrich") {
    action { ctx, prev ->
        ctx.promiseFactory.executeAsync {
            println "Enriching ${prev.size()} records..."
            prev.collect { record ->
                record + fetchAdditionalData(record.id)
            }
        }
    }
}

serviceTask("load") {
    action { ctx, prev ->
        ctx.promiseFactory.executeAsync {
            println "Loading ${prev.size()} records..."
            database.batchInsert("processed_data", prev)
            prev.size()
        }
    }
}
}

// Execute pipeline as a chain
def result = pipeline.chain("extract", "validate", "transform", "enrich", "load")
    .onComplete { count ->
        println "Pipeline completed: processed ${count} records"
    }
    .onError { error ->
        println "Pipeline failed: ${error.message}"
    }
    .run()
    .get()

println "Final result: ${result}"

```

32.8.3. Example 3: Microservices Orchestration

```

def services = TasksCollection.tasks {

    serviceTask("user-service") {

```

```

        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                httpClient.get("http://user-service/users/${prev.userId}")
            }
        }

    serviceTask("order-service") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                httpClient.get("http://order-service/orders?userId=${prev.userId}")
            }
        }
    }

    serviceTask("inventory-service") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                httpClient.get("http://inventory-service/stock/${prev.productId}")
            }
        }
    }

    serviceTask("payment-service") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                httpClient.post("http://payment-service/charge", prev)
            }
        }
    }

    serviceTask("notification-service") {
        action { ctx, prev ->
            ctx.promiseFactory.executeAsync {
                httpClient.post("http://notification-service/send", prev)
            }
        }
    }
}

// Execute order fulfillment workflow
def order = [userId: 123, productId: 456, amount: 99.99]

// Chain: get user → check inventory → charge payment → send notification
def result = services.chain(
    "user-service",
    "inventory-service",
    "payment-service",
    "notification-service"
)
.onComplete { notification ->

```

```
    println "Order fulfilled: ${notification}"  
}  
.run(order)  
.get()
```

32.9. When to Use TasksCollection

□ Use TasksCollection when:

- Tasks need to coordinate via shared context
- Want named registry for task lookup
- Need lifecycle management (start/stop)
- Using timers or business rules (event-driven)
- Want to chain tasks dynamically at runtime
- Need metrics on task execution
- Building event-driven or reactive systems

□ Don't use TasksCollection when:

- Tasks have complex dependencies (use TaskGraph)
- Need conditional routing/gateways (use TaskGraph)
- Want dependency graph visualization (use TaskGraph)
- Need automatic dependency resolution (use TaskGraph)

Chapter 33. Choosing the Right Abstraction

Use Case	Tasks	TasksCollection	TaskGraph
Quick script/one-off	<input type="checkbox"/> Perfect	Overkill	Overkill
Parallel data fetch	<input type="checkbox"/> Great	<input type="checkbox"/> Good	Overkill
Sequential pipeline	<input type="checkbox"/> Great	<input type="checkbox"/> Great (chains)	<input type="checkbox"/> Good
Event-driven system	<input type="checkbox"/> No	<input type="checkbox"/> Perfect	<input type="checkbox"/> Possible
Complex dependencies	<input type="checkbox"/> No	<input type="checkbox"/> No	<input type="checkbox"/> Required
Conditional routing	<input type="checkbox"/> No	<input type="checkbox"/> Manual	<input type="checkbox"/> Built-in
Long-running services	<input type="checkbox"/> No	<input type="checkbox"/> Perfect	<input type="checkbox"/> Good
Workflow orchestration	<input type="checkbox"/> No	<input type="checkbox"/> Limited	<input type="checkbox"/> Perfect

Chapter 34. Migration Path

As your needs grow, you can migrate between abstractions:

34.1. Tasks → TasksCollection

When you need registry and lifecycle:

```
// Before: Tasks (ad-hoc)
Tasks.all { ctx ->
    task("t1") { "A" }
    task("t2") { "B" }
}

// After: TasksCollection (managed)
def tasks = TasksCollection.tasks {
    serviceTask("t1") {
        action { ctx, prev -> ctx.promiseFactory.executeAsync { "A" } }
    }
    serviceTask("t2") {
        action { ctx, prev -> ctx.promiseFactory.executeAsync { "B" } }
    }
}
tasks.start()
```

34.2. TasksCollection → TaskGraph

When you need dependencies and orchestration:

```
// Before: TasksCollection (manual chain)
tasks.chain("fetch", "transform", "save").run()

// After: TaskGraph (declarative dependencies)
def workflow = TaskGraph.build {
    serviceTask("fetch") { ... }
    serviceTask("transform") { ... }
    serviceTask("save") { ... }

    chainVia("fetch", "transform", "save")
}
workflow.run()
```

Chapter 35. Summary

This chapter introduced two lightweight task execution abstractions:

Tasks Utility:

- □ Static utility for ad-hoc execution
- □ Patterns: all, any, sequence, parallel
- □ Auto-wrapping of non-Promise results
- □ Shared context support
- □ Perfect for scripts and simple use cases

TasksCollection:

- □ Named registry with task lookup
- □ Lifecycle management (start/stop)
- □ Auto-start timers and rules
- □ Fluent promise chaining API
- □ Built-in metrics and monitoring
- □ Perfect for event-driven systems

Both provide simpler alternatives to full TaskGraph orchestration when you don't need complex dependencies or workflow management.

Chapter 36. Next Steps

- **Chapter 4 (Layers)** - Understanding TaskGraph layered architecture
- **Chapter 7 (Examples)** - More complex workflow examples using TaskGraph
- **Chapter 11 (TaskBase)** - Deep dive into task types and capabilities
- **Chapter 12 (Gateways)** - Conditional routing and decision logic

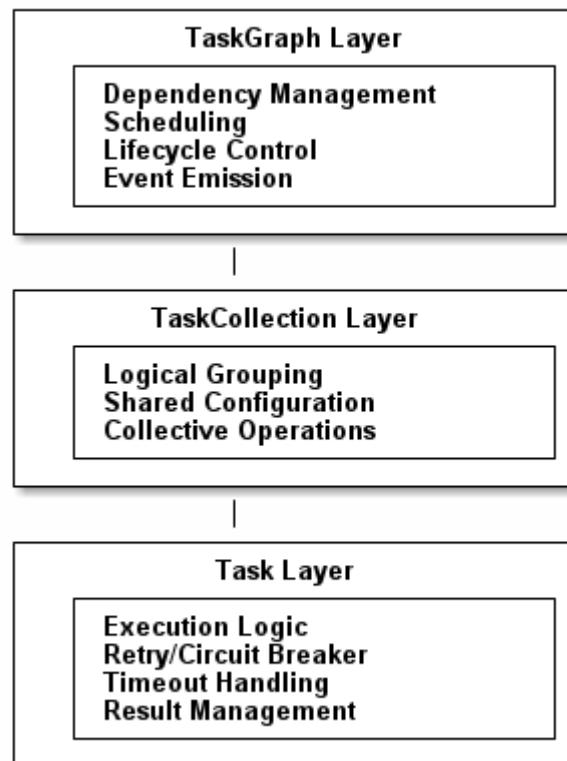
Architectural Layers: From Tasks to TaskGraph

This chapter explores the three architectural layers of the TaskGraph framework: Tasks (individual units of work), TaskCollections (logical groupings), and TaskGraph (complete workflow orchestration). Understanding these layers is essential for building effective workflows.

Chapter 37. Overview

TaskGraph is built in three progressive layers, each adding capabilities:

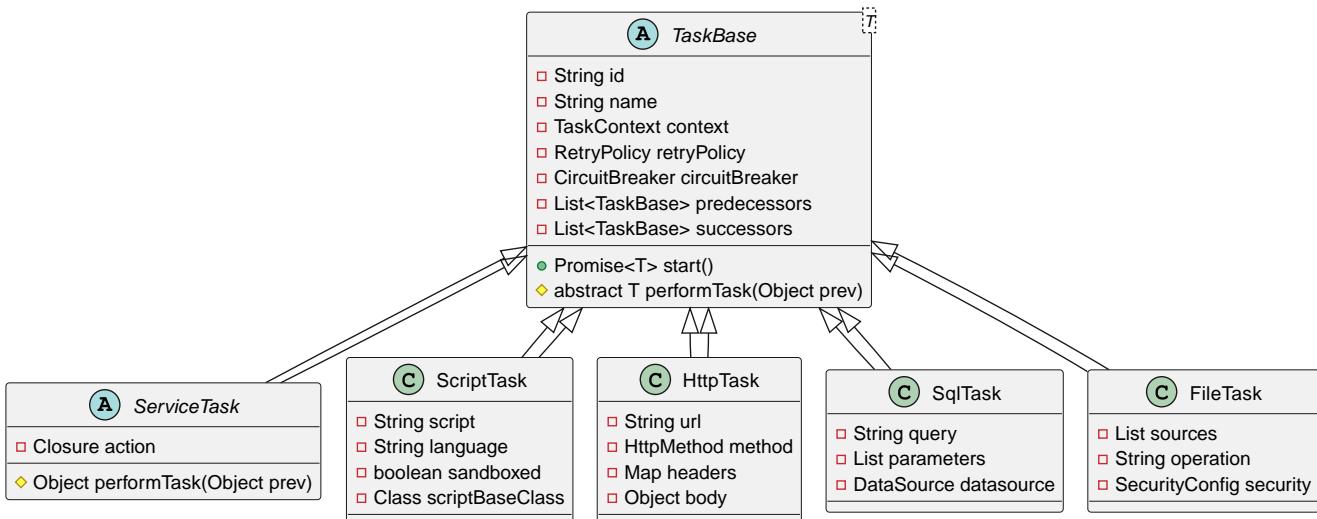
1. **Task Layer** - Individual units of work with execution logic
2. **TaskCollection Layer** - Logical grouping and shared configuration
3. **TaskGraph Layer** - Complete workflow with dependency management and orchestration



Chapter 38. Layer 1: Task

The Task layer represents individual units of work.

38.1. Task Hierarchy



38.2. TaskBase: The Foundation

Every task extends `TaskBase`, which provides:

38.2.1. Core Properties

```
abstract class TaskBase<T> {
    // Identity
    String id           // Unique identifier
    String name         // Human-readable name
    String description  // Optional description

    // Execution context
    TaskContext context // Shared execution context

    // Dependencies
    List<TaskBase> predecessors // Tasks that must complete first
    List<TaskBase> successors   // Tasks that depend on this

    // Resilience
    RetryPolicy retryPolicy // Retry configuration
    CircuitBreaker circuitBreaker // Circuit breaker
    TimeoutPolicy timeoutPolicy // Timeout handling
    IdempotencyPolicy idempotency // Idempotency control

    // State
    TaskState state      // Current execution state
    DataflowVariable result // Async result container
```

```

    // Metrics
    long startTime
    long endTime
    int attemptCount
}

```

38.2.2. Lifecycle Methods

```

abstract class TaskBase<T> {
    // Main execution entry point
    Promise<T> start() {
        // 1. Validate preconditions
        // 2. Wait for dependencies
        // 3. Check conditions
        // 4. Execute with retry/circuit breaker
        // 5. Publish result
        // 6. Notify successors
    }

    // Template method - subclasses implement
    protected abstract T performTask(Object previousValue)

    // Lifecycle hooks
    protected void beforeExecute() { }
    protected void afterExecute(T result) { }
    protected void onError(Throwable error) { }
    protected void onRetry(int attempt, Throwable error) { }
}

```

38.2.3. Cross-Cutting Concerns

TaskBase provides cross-cutting concerns for all tasks:

1. Retry Logic

```

task("flaky-operation") {
    retryPolicy {
        maxAttempts 3
        delay 1000
        exponentialBackoff true
        retryOn(IOException, TimeoutException)
    }
}

```

2. Circuit Breaker

```

task("external-service") {
    circuitBreaker {
        failureThreshold 5          // Open after 5 failures
        timeout 3000                // Consider failure after 30s
        halfOpenAfter 60000          // Try again after 60s
        successThreshold 2          // Close after 2 successes
    }
}

```

3. Timeout Control

```

task("long-running") {
    timeout 60000 // Fail if not complete in 60s
}

```

4. Idempotency

```

task("idempotent-write") {
    idempotencyPolicy {
        enabled true
        keyExtractor { ctx -> "write-${ctx.global('userId')}" }
        ttl 3600 // Cache for 1 hour
    }
}

```

5. Conditional Execution

```

task("conditional") {
    condition { ctx ->
        ctx.global("environment") == "production"
    }
    action { /* only runs in production */ }
}

```

38.3. Task Types

TaskGraph provides 15+ concrete task types:

38.3.1. ServiceTask - Generic Action

Execute arbitrary code:

```

task("custom-logic") {
    action { ctx ->
        def input = ctx.prev

```

```

    // Your logic here
    return processData(input)
}
}

```

38.3.2. ScriptTask - Embedded Scripts

Execute dynamic scripts (sandboxed by default):

```

scriptTask("process") {
    language "groovy"
    sandboxed true
    script '''
        def data = bindings.data
        return data.collect { it * 2 }
    '''
    ...
    bindings([data: [1, 2, 3, 4, 5]])
}

```

38.3.3. HttpTask - REST APIs

Make HTTP/REST calls:

```

httpTask("fetch-users") {
    url "https://api.example.com/users"
    method GET
    header "Authorization", "Bearer ${token}"
    timeout 30000
    validateSSL true
}

```

38.3.4. SqlTask - Database Operations

Execute SQL queries:

```

sqlTask("load-users") {
    datasource myDataSource
    query """
        INSERT INTO users (name, email, created)
        VALUES (?, ?, NOW())
    """
    ...
    parameters { ctx ->
        def user = ctx.prev
        [user.name, user.email]
    }
}

```

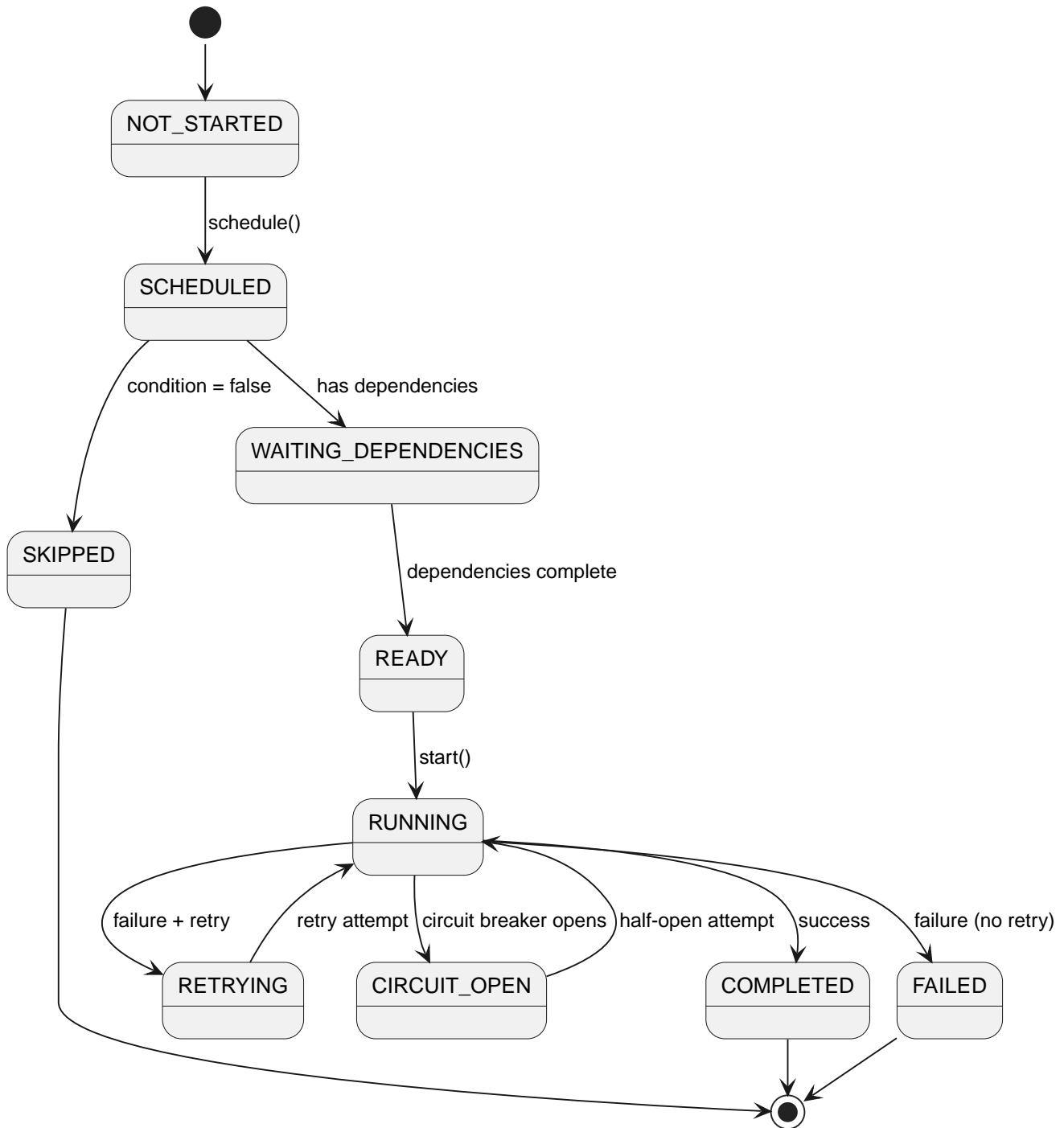
38.3.5. FileTask - File Operations

Read, write, copy, or move files:

```
fileTask("process-files") {  
    operation READ  
    sources(["/data/input.txt"])  
    securityConfig {  
        allowedDirectories(["/data"])  
        maxFileSize 10 * 1024 * 1024 // 10 MB  
    }  
}
```

38.4. Task State Machine

Tasks progress through well-defined states:



38.5. Task Execution Model

38.5.1. Synchronous Execution

Block until complete:

```

def task = new ServiceTask("my-task", ctx) {
    action { "result" }
}

def result = task.start().get() // Blocks

```

```
    println "Result: $result"
```

38.5.2. Asynchronous Execution

Non-blocking with callbacks:

```
task.start()
  .then { result ->
    println "Success: $result"
  }
  .onError { error ->
    log.error("Failed", error)
  }
```

38.5.3. Dependency-Driven Execution

Automatic coordination:

```
def task1 = new ServiceTask("task1", ctx) { action { 1 } }
def task2 = new ServiceTask("task2", ctx) { action { 2 } }
def task3 = new ServiceTask("task3", ctx) {
  action { ctx ->
    // Automatically waits for task1 and task2
    ctx.taskResult("task1") + ctx.taskResult("task2")
  }
}

task3.addPredecessor(task1)
task3.addPredecessor(task2)

def result = task3.start().get() // Returns 3
```

Chapter 39. Layer 2: TaskCollection

TaskCollection provides logical grouping and collective operations.

39.1. Purpose

TaskCollections serve several purposes:

1. **Logical grouping** - Group related tasks
2. **Shared configuration** - Apply config to multiple tasks
3. **Collective operations** - Operate on groups
4. **Namespace isolation** - Avoid task ID collisions

39.2. Creating Collections

```
def workflow = TaskGraph.build {  
    // Define a collection  
    collection("data-extraction") {  
        task("extract-users") { /* ... */ }  
        task("extract-orders") { /* ... */ }  
        task("extract-products") { /* ... */ }  
    }  
  
    // Another collection  
    collection("data-transformation") {  
        task("transform-users") { /* ... */ }  
        task("transform-orders") { /* ... */ }  
        task("transform-products") { /* ... */ }  
    }  
}
```

39.3. Shared Configuration

Apply configuration to all tasks in a collection:

```
collection("api-calls") {  
    // Shared config for all tasks  
    defaults {  
        timeout 30000  
        retryPolicy {  
            maxAttempts 3  
            delay 1000  
        }  
        header "Authorization", "Bearer ${token}"  
    }  
}
```

```

httpTask("fetch-users") {
    url "https://api.example.com/users"
    // Inherits timeout, retry, and auth header
}

httpTask("fetch-orders") {
    url "https://api.example.com/orders"
    // Inherits timeout, retry, and auth header
}
}

```

39.4. Collection Operations

39.4.1. Start All Tasks

```

def collection = workflow.getCollection("data-extraction")

// Start all tasks in parallel
def promises = collection.startAll()

// Wait for all
Promises.all(promises).then { results ->
    println "All extraction tasks complete: $results"
}

```

39.4.2. Query Collection State

```

def collection = workflow.getCollection("data-extraction")

println "Total tasks: ${collection.taskCount}"
println "Completed: ${collection.completedTaskCount}"
println "Failed: ${collection.failedTaskCount}"
println "Running: ${collection.runningTaskCount}"

if (collection.allComplete) {
    println "Collection complete!"
}

```

39.4.3. Iterate Tasks

```

def collection = workflow.getCollection("data-extraction")

collection.tasks.each { task ->
    println "Task: ${task.id} - State: ${task.state}"
}

```

```
}
```

39.5. Collection Dependencies

Collections can depend on other collections:

```
def workflow = TaskGraph.build {
    collection("extract") {
        task("extract-users") { /* ... */ }
        task("extract-orders") { /* ... */ }
    }

    collection("transform") {
        // All tasks in 'transform' depend on all tasks in 'extract'
        dependsOn collection("extract")

        task("transform-users") { /* ... */ }
        task("transform-orders") { /* ... */ }
    }
}
```

39.6. Nested Collections

Collections can be nested:

```
collection("data-pipeline") {
    collection("extract") {
        task("extract-users") { /* ... */ }
        task("extract-orders") { /* ... */ }
    }

    collection("transform") {
        dependsOn collection("extract")
        task("transform-users") { /* ... */ }
        task("transform-orders") { /* ... */ }
    }

    collection("load") {
        dependsOn collection("transform")
        task("load-warehouse") { /* ... */ }
    }
}
```

39.7. Use Cases

39.7.1. Parallel Processing by Domain

```
collection("user-domain") {  
    task("fetch-user-profile") { /* ... */ }  
    task("fetch-user-orders") { /* ... */ }  
    task("fetch-user-preferences") { /* ... */ }  
}  
  
collection("product-domain") {  
    task("fetch-product-catalog") { /* ... */ }  
    task("fetch-product-inventory") { /* ... */ }  
    task("fetch-product-reviews") { /* ... */ }  
}
```

39.7.2. ETL Pipeline Stages

```
collection("extract") {  
    defaults {  
        timeout 6000  
        retryPolicy { maxAttempts 5 }  
    }  
    // Extraction tasks  
}  
  
collection("transform") {  
    dependsOn collection("extract")  
    defaults {  
        maxConcurrency 10  
    }  
    // Transformation tasks  
}  
  
collection("load") {  
    dependsOn collection("transform")  
    defaults {  
        idempotencyPolicy { enabled true }  
    }  
    // Loading tasks  
}
```

39.7.3. Environment-Specific Tasks

```
if (environment == "production") {  
    collection("monitoring") {  
        task("send-metrics") { /* ... */ }  
        task("update-dashboard") { /* ... */ }  
        task("send-alerts") { /* ... */ }  
    }
```

```
    }  
}
```

Chapter 40. Layer 3: TaskGraph

TaskGraph is the top layer providing complete workflow orchestration.

40.1. Core Responsibilities

The TaskGraph layer handles:

1. **Dependency resolution** - Build and validate DAG
2. **Scheduling** - Determine execution order
3. **Concurrency control** - Manage parallel execution
4. **Lifecycle management** - Start, pause, cancel, resume
5. **Event emission** - Publish lifecycle events
6. **Error handling** - Graph-level error strategies
7. **Result aggregation** - Collect and report results

40.2. Graph Structure

40.2.1. Directed Acyclic Graph (DAG)

TaskGraph represents workflows as DAGs:

```
def workflow = TaskGraph.build {  
    task("A") { action { "A" } }  
    task("B") { action { "B" } }  
  
    task("C") {  
        dependsOn "A", "B"  
        action { "C" }  
    }  
  
    task("D") {  
        dependsOn "C"  
        action { "D" }  
    }  
}
```

Visual representation:



40.2.2. Cycle Detection

TaskGraph validates the DAG at build time:

```
// ☐ This will throw CyclicDependencyException
def invalid = TaskGraph.build {
    task("A") {
        dependsOn "B" // A depends on B
    }
    task("B") {
        dependsOn "A" // B depends on A - CYCLE!
    }
}
```

40.2.3. Missing Dependency Detection

```
// ☐ This will throw MissingTaskException
def invalid = TaskGraph.build {
    task("A") {
        dependsOn "B" // B doesn't exist!
    }
}
```

40.3. Graph Configuration

40.3.1. Basic Configuration

```
def workflow = TaskGraph.build {
    id "my-workflow-v1"
    description "Data processing pipeline"

    maxConcurrency 20          // Max parallel tasks
    failFast true              // Stop on first failure
    continueOnError false      // Opposite of failFast

    executorPool myCustomPool // Custom thread pool
}
```

40.3.2. Execution Strategies

Fail-Fast (default)

```
def workflow = TaskGraph.build {  
    failFast true // Stop immediately on first failure  
}
```

Continue on Error

```
def workflow = TaskGraph.build {  
    continueOnError true // Complete all possible tasks  
}
```

Partial Execution

```
// Execute only specific subgraph  
def workflow = TaskGraph.build { /* ... */ }  
  
workflow.start(  
    startTasks: ["task-A", "task-B"], // Entry points  
    endTasks: ["task-X"] // Exit points  
)
```

40.4. Dependency Patterns

40.4.1. Sequential Pipeline

```
TaskGraph.build {  
    task("step1") { action { 1 } }  
    task("step2") {  
        dependsOn "step1"  
        action { ctx -> ctx.prev + 1 }  
    }  
    task("step3") {  
        dependsOn "step2"  
        action { ctx -> ctx.prev + 1 }  
    }  
}  
// Executes: step1 → step2 → step3
```

40.4.2. Fan-Out / Fan-In

```
TaskGraph.build {  
    task("source") { action { fetchData() } }  
  
    // Fan-out: parallel processing  
    task("process-A") {
```

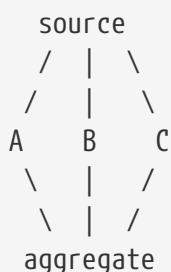
```

dependsOn "source"
action { ctx -> processA(ctx.prev) }
}
task("process-B") {
dependsOn "source"
action { ctx -> processB(ctx.prev) }
}
task("process-C") {
dependsOn "source"
action { ctx -> processC(ctx.prev) }
}

// Fan-in: aggregate results
task("aggregate") {
dependsOn "process-A", "process-B", "process-C"
action { ctx ->
[
    ctx.taskResult("process-A"),
    ctx.taskResult("process-B"),
    ctx.taskResult("process-C")
]
}
}
}
}

```

Visual:



40.4.3. Diamond Pattern

```

TaskGraph.build {
task("start") { action { "data" } }

task("left") {
dependsOn "start"
action { ctx -> processLeft(ctx.prev) }
}

task("right") {
dependsOn "start"
action { ctx -> processRight(ctx.prev) }
}
}

```

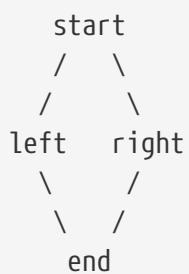
```

    }

    task("end") {
        dependsOn "left", "right"
        action { ctx ->
            merge(
                ctx.taskResult("left"),
                ctx.taskResult("right")
            )
        }
    }
}

```

Visual:



40.4.4. Parallel Independent Paths

```

TaskGraph.build {
    // Path 1: User processing
    task("fetch-users") { action { fetchUsers() } }
    task("process-users") {
        dependsOn "fetch-users"
        action { ctx -> processUsers(ctx.prev) }
    }

    // Path 2: Order processing (independent)
    task("fetch-orders") { action { fetchOrders() } }
    task("process-orders") {
        dependsOn "fetch-orders"
        action { ctx -> processOrders(ctx.prev) }
    }

    // Both paths run in parallel
}

```

Visual:



40.5. Execution Control

40.5.1. Starting Execution

```
def workflow = TaskGraph.build { /* ... */ }

// Async execution (returns Promise)
def promise = workflow.start()

promise.then { result ->
    println "Workflow completed"
    println "Results: ${result.taskResults}"
    println "Duration: ${result.durationMs}ms"
}

// Or block for result
def result = workflow.start().get()
```

40.5.2. Pausing and Resuming

```
def workflow = TaskGraph.build { /* ... */ }

def promise = workflow.start()

// Pause after 5 seconds
Thread.start {
    Thread.sleep(5000)
    workflow.pause()
    println "Workflow paused"

    Thread.sleep(10000)
    workflow.resume()
    println "Workflow resumed"
}

promise.get()
```

40.5.3. Cancellation

```
def workflow = TaskGraph.build { /* ... */ }

def promise = workflow.start()

// Cancel after 5 seconds
```

```

Thread.start {
    Thread.sleep(5000)
    workflow.cancel()
    println "Workflow cancelled"
}

promise.onError { error ->
    if (error instanceof CancellationException) {
        println "Workflow was cancelled"
    }
}

```

40.5.4. Partial Execution

```

// Execute only part of the graph
def result = workflow.start(
    startFrom: ["task-C"],      // Start from task-C (skip A, B)
    stopAt: ["task-E"]          // Stop at task-E (don't execute F, G)
).get()

```

40.6. Result Handling

40.6.1. GraphResult

Workflow execution returns a [GraphResult](#):

```

def result = workflow.start().get()

println "Status: ${result.status}" // SUCCESS, FAILED, PARTIAL
println "Duration: ${result.durationMs}ms"
println "Tasks executed: ${result.executedTaskCount}"
println "Tasks failed: ${result.failedTaskCount}"
println "Tasks skipped: ${result.skippedTaskCount}"

// Access task results
result.taskResults.each { taskId, taskResult ->
    println "$taskId: ${taskResult.value}"
}

// Check specific task
def taskResult = result.getTaskResult("my-task")
if (taskResult.isSuccess()) {
    println "Task succeeded: ${taskResult.value}"
} else {
    println "Task failed: ${taskResult.error}"
}

```

40.6.2. Accessing Intermediate Results

```
def workflow = TaskGraph.build { /* ... */ }

workflow.start().then { result ->
    // Get result from specific task
    def userData = result.getTaskResult("fetch-users").value
    def orderData = result.getTaskResult("fetch-orders").value

    // Process results
    processData(userData, orderData)
}
```

40.7. Event Handling

40.7.1. Graph-Level Events

Listen to workflow lifecycle events:

```
workflow.addListener(new GraphEventListener() {
    void onEvent(GraphEvent event) {
        switch (event.type) {
            case GRAPH_STARTED:
                println "Workflow started"
                break

            case GRAPH_COMPLETED:
                println "Workflow completed in ${event.durationMs}ms"
                break

            case GRAPH_FAILED:
                println "Workflow failed: ${event.error}"
                break

            case TASK_STARTED:
                println "Task started: ${event.taskEvent.taskName}"
                break

            case TASK_COMPLETED:
                println "Task completed: ${event.taskEvent.taskName} in
${event.taskEvent.durationMs}ms"
                break

            case TASK_FAILED:
                println "Task failed: ${event.taskEvent.taskName} -
${event.taskEvent.error}"
                break
        }
    }
})
```

```
    }
})
```

40.7.2. Filtering Events

```
// Only listen to task completion events
workflow.addListener { event ->
    if (event.type == TASK_COMPLETED) {
        def task = event.taskEvent
        println "${task.taskName}: ${task.durationMs}ms"
    }
}
```

40.8. Concurrency Control

40.8.1. Graph-Level Limits

```
def workflow = TaskGraph.build {
    maxConcurrency 10 // Max 10 tasks running simultaneously

    // Define 100 tasks
    (1..100).each { i ->
        task("task-$i") {
            action { performWork(i) }
        }
    }
}
// Only 10 run at once, others queued
```

40.8.2. Task-Level Limits

```
task("database-task") {
    concurrencyLimit 5 // Max 5 instances of this task
    action { queryDatabase() }
}
```

40.8.3. Resource Semaphores

```
TaskGraph.build {
    // Multiple tasks share a resource pool
    task("db-query-1") {
        resource "database-pool", max: 10
        action { query1() }
    }
}
```

```

task("db-query-2") {
    resource "database-pool", max: 10
    action { query2() }
}

task("db-query-3") {
    resource "database-pool", max: 10
    action { query3() }
}
}

// All three tasks share 10 database connections

```

40.9. Graph Composition

40.9.1. SubGraphs

Embed one graph in another:

```

// Define reusable subgraph
def dataExtractionGraph = TaskGraph.build {
    task("extract-users") { /* ... */ }
    task("extract-orders") { /* ... */ }
    task("extract-products") { /* ... */ }
}

// Embed in parent graph
def mainWorkflow = TaskGraph.build {
    task("validate-credentials") { /* ... */ }

    // Embed subgraph
    subgraph("extract-data", dataExtractionGraph) {
        dependsOn "validate-credentials"
    }

    task("process-data") {
        dependsOn "extract-data"
        action { /* ... */ }
    }
}

```

40.9.2. Graph Factories

```

// Factory for creating similar workflows
class WorkflowFactory {
    static TaskGraph createDataPipeline(String source, String target) {
        return TaskGraph.build {

```

```

        task("extract-from-$source") { /* ... */ }
        task("transform") {
            dependsOn "extract-from-$source"
        }
        task("load-to-$target") {
            dependsOn "transform"
        }
    }
}

// Use factory
def s3ToPostgres = WorkflowFactory.createDataPipeline("s3", "postgres")
def kafkaToElastic = WorkflowFactory.createDataPipeline("kafka", "elasticsearch")

```

40.10. Performance Optimization

40.10.1. Parallel Execution

Maximize parallelism by minimizing dependencies:

```

// ☠ Unnecessarily sequential
task("A") { action { fetchA() } }
task("B") {
    dependsOn "A" // B doesn't actually need A's result!
    action { fetchB() }
}

// ☠ Parallel execution
task("A") { action { fetchA() } }
task("B") { action { fetchB() } } // No dependency, runs in parallel

```

40.10.2. Lazy Task Creation

Create tasks dynamically only when needed:

```

TaskGraph.build {
    task("fetch-config") {
        action { fetchConfiguration() }
    }

    task("process") {
        dependsOn "fetch-config"
        action { ctx ->
            def config = ctx.prev

            // Dynamically add tasks based on config
        }
    }
}

```

```

        config.regions.each { region ->
            def regionalTask = new ServiceTask("process-$region", ctx) {
                action { processRegion(region) }
            }
            graph.addTask(regionalTask)
        }
    }
}

```

40.10.3. Result Cleanup

Clean up intermediate results to save memory:

```

task("large-dataset") {
    cleanupResults true // Clear result after successors consume it
    action { fetchLargeDataset() }
}

```

40.11. Debugging and Visualization

40.11.1. Graph Structure Inspection

```

def workflow = TaskGraph.build /* ... */

println "Total tasks: ${workflow.taskCount}"
println "Root tasks: ${workflow.rootTasks}"
println "Leaf tasks: ${workflow.leafTasks}"

// Find task by ID
def task = workflow.getTask("my-task")
println "Predecessors: ${task.predecessors}"
println "Successors: ${task.successors}"

// Topological sort (execution order)
def executionOrder = workflow.topologicalSort()
println "Execution order: $executionOrder"

```

40.11.2. Visualization Export

```

// Export to GraphViz DOT format
def dot = workflow.toDot()
File("workflow.dot").text = dot

// Export to PlantUML
def plantuml = workflow.toPlantUML()

```

```
File("workflow.puml").text = plantuml  
  
// Export to Mermaid  
def mermaid = workflow.toMermaid()  
File("workflow.mmd").text = mermaid
```

Chapter 41. Layer Integration Example

Complete example showing all three layers:

```
import org.softwood.dag.*  
import org.softwood.pool.*  
import org.softwood.promise.*  
  
// Layer 1: Individual tasks (can be used standalone)  
def pool = ExecutorPoolFactory.defaultPool()  
def ctx = new TaskContext()  
  
def task1 = new ServiceTask("fetch", ctx) {  
    action { fetchData() }  
}  
  
def result = task1.start().get() // Standalone execution  
println "Task result: $result"  
  
// Layer 2: Task collections (logical grouping)  
def workflow = TaskGraph.build {  
    collection("extraction") {  
        defaults {  
            timeout 30000  
            retryPolicy { maxAttempts 3 }  
        }  
  
        task("fetch-users") {  
            action { fetchUsers() }  
        }  
        task("fetch-orders") {  
            action { fetchOrders() }  
        }  
    }  
  
    collection("processing") {  
        dependsOn collection("extraction")  
  
        task("process-users") {  
            dependsOn "fetch-users"  
            action { ctx -> processUsers(ctx.prev) }  
        }  
        task("process-orders") {  
            dependsOn "fetch-orders"  
            action { ctx -> processOrders(ctx.prev) }  
        }  
    }  
}  
  
// Layer 3: TaskGraph orchestration
```

```
def graphResult = workflow.start()
    .timeout(300, TimeUnit.SECONDS)
    .then { result ->
        println "Workflow complete!"
        println "Executed ${result.executedTaskCount} tasks in ${result.durationMs}ms"
        return result
    }
    .onError { error ->
        log.error("Workflow failed", error)
        notifyOps(error)
    }
    .get()
```

Chapter 42. Summary

The three-layer architecture provides:

- **Task Layer** - Individual units with cross-cutting concerns
- **TaskCollection Layer** - Logical grouping and shared config
- **TaskGraph Layer** - Complete workflow orchestration
- **Flexibility** - Use any layer independently
- **Composability** - Build complex from simple
- **Separation of concerns** - Clear responsibilities

Each layer builds on the previous one, providing progressively more powerful workflow capabilities.

Chapter 43. Next Steps

- **Chapter 5** - Learn about extensibility through SPI
- **Chapter 6** - Deep dive into TaskContext
- **Chapter 11** - Complete TaskBase reference
- **Chapter 13** - All concrete task types

Extensibility Through SPI

This chapter explores how TaskGraph leverages the Service Provider Interface (SPI) pattern to enable extensibility while providing comprehensive default implementations. You'll learn how to extend TaskGraph for custom needs while benefiting from battle-tested common capabilities.

Chapter 44. Overview

TaskGraph follows a "batteries included but swappable" philosophy:

- **Default implementations** - Production-ready implementations for common scenarios
- **SPI extensibility** - Clean extension points for custom needs
- **No vendor lock-in** - Swap implementations without code changes

44.1. Design Philosophy

Provide Common Capabilities

Most users need database access, HTTP calls, file operations, messaging, etc. TaskGraph provides these out of the box.

Enable Customization

Every organization has unique requirements. SPI allows custom implementations.

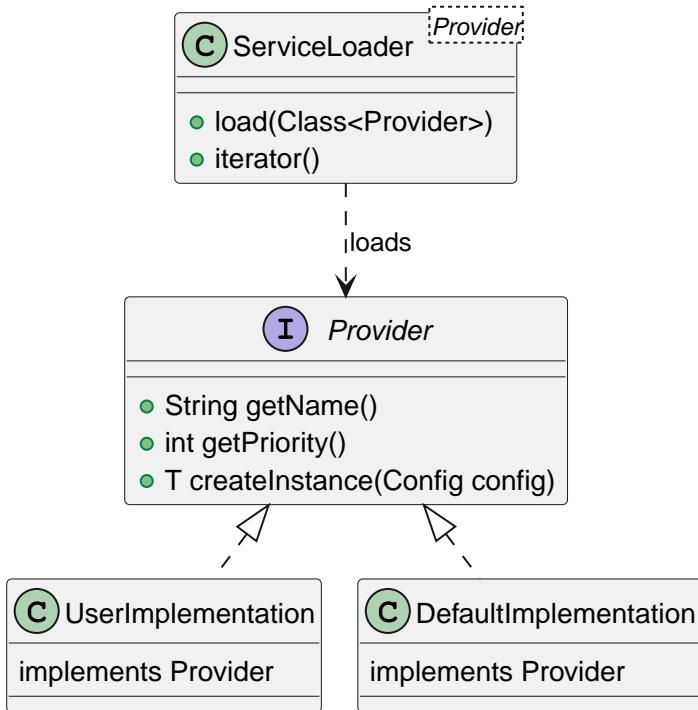
Maintain Compatibility

SPI contracts ensure custom implementations work seamlessly with TaskGraph.

Chapter 45. SPI Architecture

45.1. Core SPI Pattern

TaskGraph uses Java's ServiceLoader mechanism:



45.2. Provider Discovery

TaskGraph automatically discovers and loads providers:

```
// TaskGraph discovers providers at startup
ServiceLoader<DatabaseProvider> loader =
    ServiceLoader.load(DatabaseProvider.class)

// Select provider by name or priority
DatabaseProvider provider = loader.stream()
    .map(ServiceLoader.Provider::get)
    .max(Comparator.comparing(Provider::getPriority))
    .orElse(new DefaultDatabaseProvider())
```

45.3. Provider Registration

Register providers via [META-INF/services](#):

```
src/main/resources/META-INF/services/
    org.softwood.dag.spi.DatabaseProvider
    org.softwood.dag.spi.MessagingProvider
```

```
org.softwood.dag.spi.StorageProvider
```

Example file content:

```
com.mycompany.CustomDatabaseProvider  
com.mycompany.CustomMessagingProvider
```

Chapter 46. Available SPI Extension Points

46.1. Database Providers

46.1.1. DatabaseProvider SPI

```
interface DatabaseProvider {
    // Provider metadata
    String getName()
    int getPriority()

    // Connection management
    Connection getConnection(DatabaseConfig config)
    DataSource getDataSource(DatabaseConfig config)

    // Query execution
    <T> T executeQuery(String sql, List params, RowMapper<T> mapper)
    int executeUpdate(String sql, List params)
    List<Map<String, Object>> executeBatch(String sql, List<List> batchParams)

    // Transaction management
    void beginTransaction()
    void commit()
    void rollback()

    // Health check
    boolean isHealthy()
}
```

46.1.2. Default Implementation

TaskGraph provides a default JDBC provider:

```
class DefaultJdbcProvider implements DatabaseProvider {
    String getName() { "default-jdbc" }
    int getPriority() { 0 } // Lowest priority

    Connection getConnection(DatabaseConfig config) {
        // Standard JDBC connection
        return DriverManager.getConnection(
            config.url,
            config.username,
            config.password
        )
    }

    <T> T executeQuery(String sql, List params, RowMapper<T> mapper) {
```

```

        // JDBC query execution with proper resource management
    }
}

```

46.1.3. Custom Implementation Example

```

class CustomOracleProvider implements DatabaseProvider {
    String getName() { "oracle-optimized" }
    int getPriority() { 100 } // Higher priority

    Connection getConnection(DatabaseConfig config) {
        // Oracle-specific optimizations
        def props = new Properties()
        props.setProperty("oracle.net.CONNECT_TIMEOUT", "5000")
        props.setProperty("oracle.jdbc.ReadTimeout", "30000")

        return OracleDriver.getConnection(
            config.url,
            props
        )
    }

    <T> T executeQuery(String sql, List params, RowMapper<T> mapper) {
        // Oracle-specific query hints and optimizations
        def optimizedSql = addOracleHints(sql)
        // Execute with Oracle-specific features
    }
}

```

46.1.4. Usage in Tasks

```

sqlTask("query-data") {
    // TaskGraph automatically selects highest priority provider
    provider "oracle-optimized" // Or let auto-selection work

    query "SELECT * FROM users WHERE id = ?"
    parameters([userId])
}

```

46.2. NoSQL Providers

46.2.1. NoSqlProvider SPI

```

interface NoSqlProvider {
    String getName()
    int getPriority()
}

```

```

String getType() // "document", "key-value", "column", "graph"

// CRUD operations
<T> T get(String collection, String id, Class<T> type)
void put(String collection, String id, Object value)
void delete(String collection, String id)

// Query operations
<T> List<T> find(String collection, Query query, Class<T> type)
long count(String collection, Query query)

// Bulk operations
<T> void bulkInsert(String collection, List<T> documents)
void bulkUpdate(String collection, List<Update> updates)

// Index management
void createIndex(String collection, IndexDefinition index)
}

```

46.2.2. Provided Implementations

TaskGraph includes providers for:

- **MongoDB** - Document store
- **Redis** - Key-value store
- **Cassandra** - Column family store
- **Neo4j** - Graph database

46.2.3. Custom Implementation Example

```

class CustomDynamoDBProvider implements NoSqlProvider {
    String getName() { "dynamodb" }
    String getType() { "key-value" }

    private AmazonDynamoDB client

    <T> T get(String collection, String id, Class<T> type) {
        def result = client.getItem(
            new GetItemRequest()
                .withTableName(collection)
                .withKey([id: new AttributeValue(id)])
        )
        return mapToObject(result.item, type)
    }

    void put(String collection, String id, Object value) {
        def item = objectToMap(value)
        client.putItem(

```

```

        new PutItemRequest()
            .withTableName(collection)
            .withItem(item)
    )
}
}

```

46.3. Messaging Providers

46.3.1. MessagingProvider SPI

```

interface MessagingProvider {
    String getName()
    int getPriority()
    String getProtocol() // "amqp", "kafka", "jms", etc.

    // Publishing
    void send(String destination, Object message)
    void send(String destination, Object message, Map<String, Object> headers)
    Future<SendResult> sendAsync(String destination, Object message)

    // Consuming
    <T> T receive(String destination, Class<T> type)
    <T> T receive(String destination, long timeout, Class<T> type)
    void subscribe(String destination, MessageHandler handler)

    // Request-Reply
    <T> T sendAndReceive(String destination, Object request, Class<T> responseType)

    // Connection management
    void connect()
    void disconnect()
    boolean isConnected()
}

```

46.3.2. Provided Implementations

- **RabbitMQ** - AMQP messaging
- **Apache Kafka** - Event streaming
- **ActiveMQ** - JMS messaging
- **AWS SQS** - Cloud messaging

46.3.3. Custom Implementation Example

```

class CustomAzureServiceBusProvider implements MessagingProvider {
    String getName() { "azure-servicebus" }
}

```

```

String getProtocol() { "amqp" }

private ServiceBusClient client

void send(String destination, Object message) {
    def sender = client.createSender(destination)
    try {
        sender.sendMessage(
            new ServiceBusMessage(serialize(message))
        )
    } finally {
        sender.close()
    }
}

<T> T receive(String destination, long timeout, Class<T> type) {
    def receiver = client.createReceiver(destination)
    try {
        def message = receiver.receiveMessages(1,
            Duration.ofMillis(timeout)).first()
        return deserialize(message.body, type)
    } finally {
        receiver.close()
    }
}
}

```

46.4. Storage Providers

46.4.1. ObjectStoreProvider SPI

```

interface ObjectStoreProvider {
    String getName()
    int getPriority()
    String getType() // "s3", "blob", "file", etc.

    // Object operations
    void put(String bucket, String key, InputStream data)
    void put(String bucket, String key, byte[] data)
    InputStream get(String bucket, String key)
    void delete(String bucket, String key)
    boolean exists(String bucket, String key)

    // Metadata operations
    ObjectMetadata getMetadata(String bucket, String key)
    void setMetadata(String bucket, String key, ObjectMetadata metadata)

    // Listing operations
    List<ObjectSummary> list(String bucket, String prefix)
}

```

```

// Multipart upload
String initiateMultipartUpload(String bucket, String key)
void uploadPart(String uploadId, int partNumber, byte[] data)
void completeMultipartUpload(String uploadId)
}

```

46.4.2. Provided Implementations

- **AWS S3** - Amazon object storage
- **Azure Blob** - Microsoft blob storage
- **Google Cloud Storage** - Google object storage
- **MinIO** - Self-hosted S3-compatible storage
- **Local File System** - Development/testing

46.4.3. Custom Implementation Example

```

class CustomSwiftProvider implements ObjectStoreProvider {
    String getName() { "openstack-swift" }
    String getType() { "swift" }

    private SwiftClient client

    void put(String bucket, String key, InputStream data) {
        client.put(
            new PutObjectRequest()
                .withContainer(bucket)
                .withObject(key)
                .withInputStream(data)
        )
    }

    InputStream get(String bucket, String key) {
        def obj = client.getObject(bucket, key)
        return obj.inputStream
    }

    List<ObjectSummary> list(String bucket, String prefix) {
        return client.listObjects(bucket)
            .findAll { it.name.startsWith(prefix) }
            .collect { obj ->
                new ObjectSummary(
                    key: obj.name,
                    size: obj.contentLength,
                    lastModified: obj.lastModified
                )
            }
    }
}

```

```
    }  
}
```

46.5. Serialization Providers

46.5.1. SerializationProvider SPI

```
interface SerializationProvider {  
    String getName()  
    int getPriority()  
    List<String> getSupportedFormats() // "json", "xml", "protobuf", etc.  
  
    // Serialization  
    byte[] serialize(Object obj)  
    String serializeToString(Object obj)  
  
    // Deserialization  
    <T> T deserialize(byte[] data, Class<T> type)  
    <T> T deserialize(String data, Class<T> type)  
  
    // Streaming  
    void serialize(Object obj, OutputStream out)  
    <T> T deserialize(InputStream in, Class<T> type)  
  
    // Configuration  
    void configure(Map<String, Object> config)  
}
```

46.5.2. Provided Implementations

- **Jackson** - JSON/XML serialization
- **Gson** - Google JSON library
- **Protocol Buffers** - Binary serialization
- **Avro** - Schema-based serialization
- **Java Serialization** - Fallback

46.5.3. Custom Implementation Example

```
class CustomMsgPackProvider implements SerializationProvider {  
    String getName() { "msgpack" }  
    List<String> getSupportedFormats() { ["msgpack", "messagepack"] }  
  
    private MessagePack msgpack = new MessagePack()  
  
    byte[] serialize(Object obj) {
```

```

    return msgpack.write(obj)
}

<T> T deserialize(byte[] data, Class<T> type) {
    return msgpack.read(data, type)
}
}

```

46.6. Credential Providers

46.6.1. CredentialProvider SPI

```

interface CredentialProvider {
    String getName()
    int getPriority()

    // Credential retrieval
    Credentials getCredentials(String credentialId)
    String getSecret(String secretId)
    Map<String, String> getSecrets(List<String> secretIds)

    // Caching
    void invalidateCache(String credentialId)
    void invalidateAll()

    // Health check
    boolean isAvailable()
}

```

46.6.2. Provided Implementations

- **Environment Variables** - System env vars
- **System Properties** - Java properties
- **AWS Secrets Manager** - Cloud secrets
- **HashiCorp Vault** - Enterprise secrets
- **Azure Key Vault** - Azure secrets

46.6.3. Custom Implementation Example

```

class Custom1PasswordProvider implements CredentialProvider {
    String getName() { "1password" }
    int getPriority() { 50 }

    private OnePasswordClient client
}

```

```

String getSecret(String secretId) {
    // Format: op://vault/item/field
    def parts = secretId.split("/")
    def vault = parts[2]
    def item = parts[3]
    def field = parts[4]

    return client.read(vault, item, field)
}

Credentials getCredentials(String credentialId) {
    def username = getSecret("${credentialId}/username")
    def password = getSecret("${credentialId}/password")
    return new Credentials(username, password)
}
}

```

46.7. HTTP Client Providers

46.7.1. HttpClientProvider SPI

```

interface HttpClientProvider {
    String getName()
    int getPriority()

    // Request execution
    HttpResponse execute(HttpRequest request)
    Future<HttpResponse> executeAsync(HttpRequest request)

    // Streaming
    InputStream executeStreaming(HttpRequest request)

    // Configuration
    void configure(HttpClientConfig config)

    // Connection management
    void closeIdleConnections(long idleTime, TimeUnit unit)
    void shutdown()
}

```

46.7.2. Provided Implementations

- **Apache HttpClient** - Industry standard (default)
- **OkHttp** - Modern HTTP client
- **Java 11 HttpClient** - Built-in client

46.7.3. Custom Implementation Example

```
class CustomHttpClientProvider implements HttpClientProvider {
    String getName() { "custom-http" }
    int getPriority() { 75 }

    private OkHttpClient client

    HttpResponse execute(HttpRequest request) {
        def okRequest = Request.Builder()
            .url(request.url)
            .method(request.method, request.body)
            .build()

        request.headers.each { k, v ->
            okRequest.header(k, v)
        }

        def response = client.newCall(okRequest).execute()

        return new HttpResponse(
            statusCode: response.code(),
            body: response.body().bytes(),
            headers: response.headers().toMultimap()
        )
    }

    Future<HttpResponse> executeAsync(HttpRequest request) {
        // Async execution
        def promise = Promises.deferred()

        client.newCall(okRequest).enqueue(
            new Callback() {
                void onResponse(Call call, Response response) {
                    promise.resolve(mapResponse(response))
                }
                void onFailure(Call call, IOException e) {
                    promise.reject(e)
                }
            }
        )

        return promise.promise
    }
}
```

Chapter 47. Creating Custom Task Types

Beyond SPI providers, you can create custom task types.

47.1. Extending TaskBase

```
class ElasticsearchTask extends TaskBase<SearchResult> {
    String index
    String query
    ElasticsearchClient client

    @Override
    protected SearchResult performTask(Object previousValue) {
        // Implement Elasticsearch-specific logic
        def searchRequest = new SearchRequest(index)
        searchRequest.source(
            new SearchSourceBuilder()
                .query(QueryBuilders.queryStringQuery(query))
        )

        def response = client.search(searchRequest, RequestOptions.DEFAULT)

        return new SearchResult(
            hits: response.hits.hits.collect { it.sourceAsMap },
            totalHits: response.hits.totalHits.value
        )
    }

    // Custom DSL methods
    ElasticsearchTask index(String index) {
        this.index = index
        return this
    }

    ElasticsearchTask query(String query) {
        this.query = query
        return this
    }
}
```

47.2. DSL Builder for Custom Tasks

```
// Add DSL method to TaskGraphBuilder
class CustomTaskGraphBuilder extends TaskGraphBuilder {

    ElasticsearchTask elasticsearchTask(String id,
                                      @DelegatesTo(ElasticsearchTask) Closure
```

```

config) {
    def task = new ElasticsearchTask(id, context)
    config.delegate = task
    config.resolveStrategy = Closure.DELEGATE_FIRST
    config()

    addTask(task)
    return task
}
}

```

47.3. Usage

```

def workflow = TaskGraph.build {
    elasticsearchTask("search-users") {
        index "users"
        query "status:active AND role:admin"

        retryPolicy {
            maxAttempts 3
            delay 1000
        }
    }

    task("process-results") {
        dependsOn "search-users"
        action { ctx ->
            def results = ctx.prev
            processSearchResults(results)
        }
    }
}

```

Chapter 48. Custom Gateway Types

Create custom gateways for specialized routing logic.

48.1. Extending Gateway

```
class WeightedGateway extends Gateway {
    Map<TaskBase, Integer> weights = [:]

    @Override
    TaskBase selectPath(TaskContext context) {
        // Select path based on weighted random selection
        def totalWeight = weights.values().sum()
        def random = new Random().nextInt(totalWeight)

        def cumulative = 0
        for (entry in weights.entrySet()) {
            cumulative += entry.value
            if (random < cumulative) {
                return entry.key
            }
        }

        return weights.keySet().first()
    }

    // DSL methods
    void path(TaskBase task, int weight) {
        weights[task] = weight
    }
}
```

48.2. DSL Builder

```
class CustomTaskGraphBuilder extends TaskGraphBuilder {

    WeightedGateway weightedGateway(String id,
                                    @DelegatesTo(WeightedGateway) Closure config) {
        def gateway = new WeightedGateway(id, context)
        config.delegate = gateway
        config.resolveStrategy = Closure.DELEGATE_FIRST
        config()

        addTask(gateway)
        return gateway
    }
}
```

```
}
```

48.3. Usage

```
def workflow = TaskGraph.build {
    task("fast-path") { action { fastProcess() } }
    task("slow-path") { action { thoroughProcess() } }
    task("medium-path") { action { balancedProcess() } }

    weightedGateway("route") {
        path task("fast-path"), 70      // 70% probability
        path task("medium-path"), 20    // 20% probability
        path task("slow-path"), 10     // 10% probability
    }
}
```

Chapter 49. Custom Script Base Classes

Provide custom script base classes for domain-specific operations.

49.1. Creating Custom Base Class

```
abstract class CompanyScriptBase extends Script {
    // Company-specific utilities available in all scripts

    def fetchDataWarehouse(String query) {
        // Internal data warehouse access
        def client = DataWarehouseClient.instance
        return client.query(query)
    }

    def sendToKafka(String topic, Object message) {
        // Internal Kafka cluster
        def producer = KafkaProducerPool.getProducer()
        producer.send(topic, message)
    }

    def callInternalAPI(String endpoint, Map params) {
        // Internal API with auth handled automatically
        def client = InternalAPIClient.withAuth()
        return client.get(endpoint, params)
    }

    // Restricted operations
    @Override
    void execute() {
        // Prevent file system access
        throw new SecurityException("Direct script execution not allowed")
    }
}
```

49.2. Usage in Scripts

```
scriptTask("company-workflow") {
    customScriptBaseClass CompanyScriptBase

    script '''
        // Company-specific methods available
        def data = fetchDataWarehouse("SELECT * FROM sales")

        def processed = data.collect { row ->
            processRow(row)
        }
    
```

```
    sendToKafka("processed-sales", processed)

    return processed.size()
}

}
```

Chapter 50. Provider Selection Strategies

50.1. Priority-Based Selection

TaskGraph selects providers by priority:

```
class HighPriorityProvider implements DatabaseProvider {
    int getPriority() { 100 } // Selected first
}

class DefaultProvider implements DatabaseProvider {
    int getPriority() { 0 } // Fallback
}
```

50.2. Explicit Provider Selection

```
sqlTask("query") {
    provider "oracle-optimized" // Explicitly select provider
    query "SELECT * FROM users"
}
```

50.3. Configuration-Based Selection

```
// application.properties
taskgraph.database.provider=oracle-optimized
taskgraph.messaging.provider=rabbitmq
taskgraph.storage.provider=aws-s3

// TaskGraph reads config and selects providers
def workflow = TaskGraph.build {
    sqlTask("query") {
        // Uses provider from config
        query "SELECT * FROM users"
    }
}
```

50.4. Environment-Based Selection

```
class EnvironmentAwareProvider implements DatabaseProvider {
    int getPriority() {
        // Higher priority in production
        return System.getenv("ENV") == "production" ? 100 : 0
    }
}
```

}

Chapter 51. Testing Custom Providers

51.1. Unit Testing Providers

```
class CustomProviderTest {
    @Test
    void testProviderDiscovery() {
        def loader = ServiceLoader.load(DatabaseProvider.class)
        def provider = loader.find { it.name == "custom-provider" }

        assert provider != null
        assert provider.priority == 100
    }

    @Test
    void testProviderFunctionality() {
        def provider = new CustomDatabaseProvider()
        def config = new DatabaseConfig(url: "jdbc:custom:...")

        def conn = provider.getConnection(config)
        assert conn != null
        assert conn.isValid(5)
    }
}
```

51.2. Integration Testing

```
class CustomProviderIntegrationTest {
    @Test
    void testInWorkflow() {
        def workflow = TaskGraph.build {
            sqlTask("query") {
                provider "custom-provider"
                query "SELECT * FROM test_table"
            }
        }

        def result = workflow.start().get()
        assert result.status == SUCCESS
        assert result.getTaskResult("query").value != null
    }
}
```

Chapter 52. Best Practices

52.1. Provider Design

1. Single Responsibility

```
// ☐ Good - focused provider
class PostgresProvider implements DatabaseProvider {
    // PostgreSQL-specific implementation
}

// ☐ Bad - too broad
class UniversalDatabaseProvider implements DatabaseProvider {
    // Tries to support all databases
}
```

2. Fail Fast

```
Connection getConnection(DatabaseConfig config) {
    // Validate config immediately
    if (!config.url?.startsWith("jdbc:postgres:")) {
        throw new IllegalArgumentException(
            "Invalid PostgreSQL URL: ${config.url}"
        )
    }
    // Continue with connection
}
```

3. Resource Management

```
class MyProvider implements DatabaseProvider {
    private DataSource dataSource

    void shutdown() {
        // Clean up resources
        if (dataSource instanceof Closeable) {
            dataSource.close()
        }
    }
}
```

4. Health Checks

```
boolean isHealthy() {
    try {
```

```

    def conn = getConnection(config)
    def healthy = conn.isValid(5)
    conn.close()
    return healthy
} catch (Exception e) {
    log.error("Health check failed", e)
    return false
}
}

```

52.2. Custom Task Design

1. Inherit Cross-Cutting Concerns

```

// Extends TaskBase to get retry, circuit breaker, etc.
class CustomTask extends TaskBase<Result> {
    @Override
    protected Result performTask(Object prev) {
        // Focus on task logic
    }
}

```

2. Validate Configuration

```

@Override
void validate() {
    super.validate()

    if (index == null || index.isEmpty()) {
        throw new ValidationException("Index is required")
    }
    if (query == null) {
        throw new ValidationException("Query is required")
    }
}

```

3. Fluent DSL Methods

```

CustomTask index(String index) {
    this.index = index
    return this // Return this for chaining
}

CustomTask query(String query) {
    this.query = query
    return this // Return this for chaining
}

```

}

Chapter 53. Summary

TaskGraph's SPI model provides:

- ☐ **Comprehensive defaults** - Production-ready implementations
- ☐ **Clean extension points** - Well-defined SPI contracts
- ☐ **Multiple providers** - Database, messaging, storage, etc.
- ☐ **Custom tasks** - Create domain-specific tasks
- ☐ **Custom gateways** - Specialized routing logic
- ☐ **Script base classes** - Domain-specific script capabilities
- ☐ **Auto-discovery** - ServiceLoader-based provider discovery
- ☐ **Priority-based selection** - Flexible provider selection

The SPI pattern enables TaskGraph to be both powerful out-of-the-box and infinitely extensible for custom requirements.

Chapter 54. Next Steps

- **Chapter 11** - TaskBase reference with all extension points
- **Chapter 13** - All provided concrete task types
- **Appendix** - Complete SPI reference documentation

Chapter 55. Part II: Usage Guide

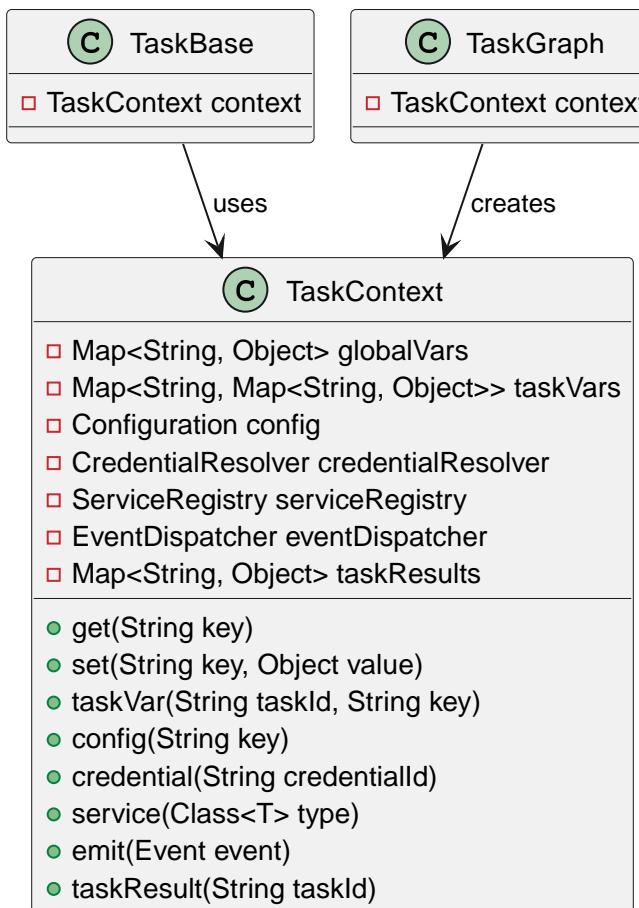
TaskContext: Shared Execution State

This chapter explores TaskContext, the execution environment that provides shared state, configuration, credentials, and services to all tasks in a workflow. Understanding TaskContext is essential for building cohesive, maintainable workflows.

Chapter 56. Overview

TaskContext is the shared execution environment for all tasks in a TaskGraph workflow. It provides:

- **Variable storage** - Task-scoped and global variables
- **Configuration access** - Centralized configuration
- **Credential management** - Secure credential resolution
- **Service registry** - Shared services and resources
- **Event dispatcher** - Event publishing
- **Result storage** - Task result access



Chapter 57. Variable Storage

TaskContext provides hierarchical variable storage.

57.1. Global Variables

Shared across all tasks in the workflow:

```
def workflow = TaskGraph.build {
    // Set global variable at graph level
    global("environment", "production")
    global("userId", 12345)
    global("apiToken", fetchToken())

    task("task1") {
        action { ctx ->
            // Access global variable
            def env = ctx.global("environment")
            def userId = ctx.global("userId")

            println "Running in $env for user $userId"
            return fetchData(userId)
        }
    }

    task("task2") {
        dependsOn "task1"
        action { ctx ->
            // All tasks see same global variables
            def env = ctx.global("environment")
            def userId = ctx.global("userId")

            processData(env, userId)
        }
    }
}
```

57.2. Task-Scooped Variables

Private to a specific task:

```
task("task1") {
    action { ctx ->
        // Set task-scoped variable
        ctx.taskVar("task1", "internalState", someValue)

        // Read task-scoped variable
    }
}
```

```

    def state = ctx.taskVar("task1", "internalState")

    return result
}
}

task("task2") {
    action { ctx ->
        // Cannot access task1's variables
        def state = ctx.taskVar("task1", "internalState") // Returns null
    }
}

```

57.3. Previous Task Result

Access the result of the immediate predecessor:

```

task("fetch") {
    action { fetchUsers() }
}

task("process") {
    dependsOn "fetch"
    action { ctx ->
        // ctx.prev contains result from "fetch"
        def users = ctx.prev
        return users.collect { processUser(it) }
    }
}

```

57.4. Task Results by ID

Access any task's result by ID:

```

task("fetch-users") {
    action { fetchUsers() }
}

task("fetch-orders") {
    action { fetchOrders() }
}

task("join") {
    dependsOn "fetch-users", "fetch-orders"
    action { ctx ->
        // Access specific task results
        def users = ctx.taskResult("fetch-users")
    }
}

```

```
def orders = ctx.taskResult("fetch-orders")

    return joinData(users, orders)
}
}
```

57.5. Variable Type Safety

TaskContext provides type-safe variable access:

```
// Type-safe getters
def userId = ctx.global("userId", Integer.class)
def userName = ctx.global("userName", String.class)
def config = ctx.global("config", Config.class)

// With default values
def timeout = ctx.global("timeout", Integer.class, 30000)
def retries = ctx.global("retries", Integer.class, 3)

// Null-safe access
def optional = ctx.globalOptional("maybeNull")
optional.ifPresent { value ->
    println "Value: $value"
}
```

Chapter 58. Configuration Management

58.1. Hierarchical Configuration

TaskContext supports hierarchical configuration:

```
// application.properties or application.yml
taskgraph:
    database:
        url: jdbc:postgresql://localhost:5432/mydb
        username: dbuser
        pool:
            maxSize: 20
            minSize: 5
    http:
        timeout: 30000
        retries: 3

// Access in tasks
task("query") {
    action { ctx ->
        def dbUrl = ctx.config("taskgraph.database.url")
        def poolMax = ctx.config("taskgraph.database.pool.maxSize")
        def timeout = ctx.config("taskgraph.http.timeout")

        // Use config values
    }
}
```

58.2. Configuration Sources

TaskContext merges configuration from multiple sources:

1. **System properties** (highest priority)
2. **Environment variables**
3. **Configuration files** (application.properties, application.yml)
4. **Programmatic configuration**
5. **Default values** (lowest priority)

```
// Programmatic configuration
def workflow = TaskGraph.build {
    config {
        set("app.timeout", 30000)
        set("app.retries", 3)
        set("app.environment", "production")
```

```

    }

    task("process") {
        action { ctx ->
            def timeout = ctx.config("app.timeout")
            // Use timeout
        }
    }
}

```

58.3. Environment-Specific Configuration

```

// application-dev.yml
taskgraph:
  database:
    url: jdbc:postgresql://localhost:5432/devdb
  http:
    timeout: 60000

// application-prod.yml
taskgraph:
  database:
    url: jdbc:postgresql://prod-server:5432/proddb
  http:
    timeout: 10000

// Select profile
System.setProperty("spring.profiles.active", "prod")

def workflow = TaskGraph.build {
    task("query") {
        action { ctx ->
            // Gets prod config
            def dbUrl = ctx.config("taskgraph.database.url")
            def timeout = ctx.config("taskgraph.http.timeout")
        }
    }
}

```

58.4. Type-Safe Configuration

```

// Define configuration class
@ConfigurationProperties("taskgraph.database")
class DatabaseConfig {
    String url
    String username
    PoolConfig pool
}

```

```
static class PoolConfig {
    int maxSize
    int minSize
    long keepAliveMs
}
}

// Bind configuration
def dbConfig = ctx.config(DatabaseConfig.class)

// Type-safe access
println dbConfig.url
println dbConfig.pool.maxSize
```

Chapter 59. Credential Management

TaskContext provides secure credential resolution.

59.1. Credential Resolvers

Multiple credential sources supported:

```
// Environment variables
def password = ctx.credential("DB_PASSWORD")

// AWS Secrets Manager
def apiKey = ctx.credential("aws:secretsmanager:api-key")

// HashiCorp Vault
def dbCreds = ctx.credential("vault:secret/database/prod")

// Azure Key Vault
def cert = ctx.credential("azure:keyvault:ssl-cert")

// System properties
def token = ctx.credential("sys:api.token")
```

59.2. Credential Caching

Credentials are cached for performance:

```
// First access - fetches from source
def password = ctx.credential("db-password") // ~100ms

// Subsequent access - cached
def password2 = ctx.credential("db-password") // ~1ms

// Invalidate cache
ctx.invalidateCredential("db-password")

// Next access fetches again
def password3 = ctx.credential("db-password") // ~100ms
```

59.3. Structured Credentials

```
// Fetch structured credential
def dbCreds = ctx.credential("database-credentials")

// Structured format: { username: "...", password: "..." }
```

```
def username = dbCreds.username
def password = dbCreds.password

// Use in connection
def conn = DriverManager.getConnection(
    ctx.config("db.url"),
    username,
    password
)
```

59.4. Credential Injection

Automatically inject credentials into tasks:

```
sqlTask("query") {
    credentialId "database-credentials" // Auto-injected

    query "SELECT * FROM users"
}

httpTask("api-call") {
    credentialId "api-credentials" // Auto-injected for auth

    url "https://api.example.com/data"
}
```

Chapter 60. Service Registry

TaskContext maintains a registry of shared services.

60.1. Registering Services

```
def workflow = TaskGraph.build {
    // Register services at graph level
    service(DataSource.class, createDataSource())
    service(HttpClient.class, createHttpClient())
    service(CacheManager.class, createCacheManager())

    task("query") {
        action { ctx ->
            // Access registered service
            def ds = ctx.service(DataSource.class)
            def conn = ds.connection
            // Use connection
        }
    }

    task("api-call") {
        action { ctx ->
            def client = ctx.service(HttpClient.class)
            def response = client.execute(request)
            return response
        }
    }
}
```

60.2. Service Lifecycle

Services can have lifecycle management:

```
interface ManagedService {
    void start()
    void stop()
    boolean isRunning()
}

class ConnectionPoolService implements ManagedService {
    private HikariDataSource dataSource

    void start() {
        dataSource = new HikariDataSource(config)
    }
}
```

```

void stop() {
    dataSource?.close()
}

boolean isRunning() {
    return dataSource != null && !dataSource.isClosed()
}

Connection getConnection() {
    return dataSource.connection
}
}

// Register managed service
def poolService = new ConnectionPoolService()
ctx.registerService(DataSource.class, poolService)

// TaskGraph starts service
poolService.start()

// Use in tasks
task("query") {
    action { ctx ->
        def pool = ctx.service(DataSource.class)
        def conn = pool.connection
        // Use connection
    }
}

// TaskGraph stops service on completion
workflow.onComplete {
    poolService.stop()
}

```

60.3. Service Factories

Lazy service initialization:

```

def workflow = TaskGraph.build {
    // Register service factory
    serviceFactory(DataSource.class) {
        // Created on first access
        createDataSource(ctx.config("database"))
    }

    task("query") {
        action { ctx ->
            // Service created here if not already
            def ds = ctx.service(DataSource.class)
        }
    }
}

```

```
// Use datasource
}
}
}
```

Chapter 61. Event Dispatching

TaskContext provides event publishing capabilities.

61.1. Publishing Events

```
task("process") {
    action { ctx ->
        // Publish custom event
        ctx.emit(new CustomEvent(
            type: "DATA_PROCESSED",
            data: processedData,
            timestamp: System.currentTimeMillis()
        ))
    }
    return processedData
}
```

61.2. Event Listeners

```
// Add event listener
workflow.addListener(new EventListener() {
    void onEvent(Event event) {
        if (event instanceof CustomEvent) {
            println "Custom event: ${event.type}"
            println "Data: ${event.data}"

            // Send to monitoring system
            metricsCollector.record(event)
        }
    }
})
```

61.3. Typed Event Handlers

```
// Type-specific handlers
workflow.on(TaskStartedEvent.class) { event ->
    println "Task started: ${event.taskName}"
}

workflow.on(TaskCompletedEvent.class) { event ->
    println "Task completed: ${event.taskName} in ${event.durationMs}ms"
}
```

```
workflow.on(CustomEvent.class) { event ->
    println "Custom: ${event.type}"
}
```

Chapter 62. Context Inheritance

62.1. Parent-Child Context

SubGraphs inherit parent context:

```
def parentWorkflow = TaskGraph.build {
    global("apiToken", fetchToken())
    global("environment", "production")

    subgraph("child-workflow", childWorkflow) {
        // Child inherits parent globals
    }
}

def childWorkflow = TaskGraph.build {
    task("child-task") {
        action { ctx ->
            // Access parent global variables
            def token = ctx.global("apiToken")
            def env = ctx.global("environment")

            // Child can also set its own
            ctx.global("childVar", "value")
        }
    }
}
```

62.2. Context Isolation

Task-scoped variables are isolated:

```
task("task1") {
    action { ctx ->
        ctx.taskVar("task1", "secret", "sensitive-data")
        // Only task1 can see this
    }
}

task("task2") {
    action { ctx ->
        // Cannot access task1's variables
        def secret = ctx.taskVar("task1", "secret") // null
    }
}
```

Chapter 63. Advanced Context Features

63.1. Context Cloning

Clone context for parallel execution:

```
def baseContext = new TaskContext()
baseContext.global("shared", "value")

// Clone for isolated execution
def context1 = baseContext.clone()
def context2 = baseContext.clone()

// Each clone has independent state
context1.global("instance", "1")
context2.global("instance", "2")

// Both see shared values
assert context1.global("shared") == "value"
assert context2.global("shared") == "value"

// But not each other's modifications
assert context1.global("instance") == "1"
assert context2.global("instance") == "2"
```

63.2. Context Snapshots

Capture context state at a point in time:

```
def workflow = TaskGraph.build {
    task("setup") {
        action { ctx ->
            ctx.global("state", "initialized")
        }
    }

    task("checkpoint") {
        dependsOn "setup"
        action { ctx ->
            // Capture snapshot
            def snapshot = ctx.snapshot()

            // Continue modifying context
            ctx.global("state", "processing")

            // Restore if needed
            if (errorOccurred) {

```

```

        ctx.restore(snapshot)
    }
}
}
}
```

63.3. Context Validation

Validate required context at build time:

```

def workflow = TaskGraph.build {
    // Require globals
    requireGlobal("apiToken")
    requireGlobal("environment")
    requireGlobal("userId")

    // Validate on build
    validate()

    task("process") {
        action { ctx ->
            // Guaranteed to have these globals
            def token = ctx.global("apiToken")
            def env = ctx.global("environment")
        }
    }
}

// ☐ Throws ValidationException if missing
workflow.start()
```

63.4. Context Debugging

```

// Enable context tracing
ctx.enableTracing()

task("debug") {
    action { ctx ->
        // Log context access
        ctx.global("value") // Logs: Context.global(value) = ...

        // Dump context state
        println ctx.dump()
        /*
         * Global Variables:
         *   environment = production
         *   userId = 12345
         */
    }
}
```

```
*   apiToken = ***REDACTED***  
*  
* Task Variables:  
*   task1.state = initialized  
*  
* Configuration:  
*   database.url = jdbc:...  
*   http.timeout = 30000  
*  
* Services:  
*   DataSource = HikariDataSource@abc123  
*   HttpClient = ApacheHttpClient@def456  
*/  
}  
}
```

Chapter 64. Context Best Practices

64.1. Use Global Variables for Shared State

```
// ☀ Good - shared state in globals
def workflow = TaskGraph.build {
    global("userId", 12345)
    global("apiToken", token)

    task("task1") {
        action { ctx ->
            def userId = ctx.global("userId")
            // Use userId
        }
    }
}

// ☠ Bad - duplicated values
task("task1") {
    action {
        def userId = 12345 // Hardcoded
    }
}
```

64.2. Use Configuration for Environment-Specific Values

```
// ☀ Good - externalized config
task("api-call") {
    action { ctx ->
        def url = ctx.config("api.url")
        def timeout = ctx.config("api.timeout")
        // Use config values
    }
}

// ☠ Bad - hardcoded values
task("api-call") {
    action {
        def url = "https://api.example.com" // Hardcoded
    }
}
```

64.3. Use Credentials for Sensitive Data

```
// ☐ Good - secure credential resolution
task("connect") {
    action { ctx ->
        def password = ctx.credential("db-password")
        // Use password
    }
}

// ☐ Bad - hardcoded credentials
task("connect") {
    action {
        def password = "secret123" // ☐ NEVER DO THIS!
    }
}
```

64.4. Use Service Registry for Shared Resources

```
// ☐ Good - shared service
def workflow = TaskGraph.build {
    service(HttpClient.class, createClient())

    task("call1") {
        action { ctx ->
            def client = ctx.service(HttpClient.class)
            client.execute(request1)
        }
    }

    task("call2") {
        action { ctx ->
            def client = ctx.service(HttpClient.class)
            client.execute(request2)
        }
    }
}

// ☐ Bad - creating multiple clients
task("call1") {
    action {
        def client = new HttpClient() // Creates new instance
    }
}
task("call2") {
    action {
        def client = new HttpClient() // Creates another instance
    }
}
```

```
}
```

64.5. Clean Up Task-SScoped Variables

```
task("large-data") {
    action { ctx ->
        def data = fetchLargeDataset()

        // Store temporarily
        ctx.taskVar("large-data", "dataset", data)

        // Process
        processData(data)

        // Clean up
        ctx.removeTaskVar("large-data", "dataset")

        return result
    }
}
```

Chapter 65. Context Performance Considerations

65.1. Variable Access Performance

- **Global variable access:** O(1) - HashMap lookup
- **Task variable access:** O(1) - Nested HashMap lookup
- **Configuration access:** O(1) with caching
- **Credential access:** O(1) with caching (first access may be slow)
- **Service access:** O(1) - Registry lookup

65.2. Memory Management

```
// Large intermediate results consume memory
task("large-result") {
    action {
        return new byte[100 * 1024 * 1024] // 100 MB
    }
}

// Clear result after use
task("process") {
    dependsOn "large-result"
    action { ctx ->
        def data = ctx.prev
        processData(data)

        // Clear from context
        ctx.clearTaskResult("large-result")

        return result
    }
}
```

65.3. Credential Caching

```
// Configure cache TTL
ctx.credentialCache {
    ttl 3600 // Cache for 1 hour
    maxSize 100 // Max 100 credentials
}

// Credentials cached automatically
```

```
def password = ctx.credential("db-password") // Fetches
def password2 = ctx.credential("db-password") // Cached

// Invalidate when needed
ctx.invalidateCredential("db-password")
```

Chapter 66. Summary

TaskContext provides:

- **Variable storage** - Global and task-scoped variables
- **Configuration** - Hierarchical, environment-specific config
- **Credentials** - Secure credential resolution with caching
- **Service registry** - Shared services and resources
- **Event dispatching** - Custom event publishing
- **Result access** - Access task results by ID
- **Type safety** - Type-safe access methods
- **Performance** - Efficient lookups and caching

TaskContext is the glue that binds workflows together, providing a consistent execution environment for all tasks.

Chapter 67. Next Steps

- **Chapter 7** - Practical examples using TaskContext
- **Chapter 9** - Security features in TaskContext
- **Chapter 10** - Observability with events

Practical Examples and Usage Patterns

This chapter provides practical, real-world examples demonstrating common TaskGraph usage patterns. Each example is complete and ready to adapt for your own workflows.

Chapter 68. Overview

This chapter covers common workflow patterns:

- Data pipelines (ETL)
- API orchestration
- Parallel processing
- Conditional workflows
- Error handling patterns
- Retry strategies
- Long-running workflows
- Event-driven workflows

Chapter 69. Example 1: Simple Data Pipeline (ETL)

Extract data from an API, transform it, and load into a database.

```
import org.softwood.dag.*

def etlPipeline = TaskGraph.build {
    // Extract: Fetch data from REST API
    httpTask("extract-users") {
        url "https://api.example.com/users"
        method GET
        header "Authorization", "Bearer ${config('api.token')}"
        retryPolicy {
            maxAttempts 3
            delay 1000
            exponentialBackoff true
        }
    }

    // Transform: Clean and validate data
    scriptTask("transform-users") {
        dependsOn "extract-users"

        script '''
            def users = bindings.users
            return users.collect { user ->
                [
                    id: user.id,
                    name: user.name?.trim(),
                    email: user.email?.toLowerCase(),
                    created: new Date()
                ]
            }.findAll { it.name && it.email }
        '''

        bindings { ctx ->
            [users: ctx.prev]
        }
    }

    // Load: Insert into database
    sqlTask("load-users") {
        dependsOn "transform-users"

        datasource myDataSource
        query """
            INSERT INTO users (id, name, email, created)
        """
    }
}
```

```
VALUES (?, ?, ?, ?, ?)
ON CONFLICT (id) DO UPDATE
SET name = EXCLUDED.name,
    email = EXCLUDED.email
"""

batchParameters { ctx ->
    ctx.prev.collect { user ->
        [user.id, user.name, user.email, user.created]
    }
}
}

// Execute pipeline
def result = etlPipeline.start().get()
println "Loaded ${result.getTaskResult('load-users').rowsAffected} users"
```

Chapter 70. Example 2: Fan-Out / Fan-In Pattern

Process multiple data sources in parallel, then aggregate results.

```
def aggregationWorkflow = TaskGraph.build {
    // Define multiple data sources
    def sources = ['users', 'orders', 'products', 'reviews']

    // Fan-out: Fetch from all sources in parallel
    sources.each { source ->
        httpTask("fetch-${source}") {
            url "https://api.example.com/${source}"
            method GET
            timeout 30000

            circuitBreaker {
                failureThreshold 5
                timeout 30000
            }
        }
    }

    // Fan-in: Aggregate all results
    task("aggregate-data") {
        dependsOn sources.collect { "fetch-${it}" }

        action { ctx ->
            def aggregated = [:]

            sources.each { source ->
                def result = ctx.taskResult("fetch-${source}")
                aggregated[source] = result
            }

            return [
                timestamp: new Date(),
                sources: aggregated.keySet(),
                totalRecords: aggregated.values().sum { it.size() },
                data: aggregated
            ]
        }
    }

    // Store aggregated results
    task("store-results") {
        dependsOn "aggregate-data"

        action { ctx ->
```

```
    def data = ctx.prev
    storageService.save("daily-aggregation", data)
    return "Stored ${data.totalRecords} records"
}
}

def result = aggregationWorkflow.start().get()
println result.getTaskResult("store-results").value
```

Chapter 71. Example 3: Conditional Routing

Route workflow based on runtime conditions.

```
def conditionalWorkflow = TaskGraph.build {
    // Fetch user data
    task("fetch-user") {
        action { ctx ->
            def userId = ctx.global("userId")
            return userService.getUser(userId)
        }
    }

    // Route based on user type
    exclusiveGateway("check-user-type") {
        dependsOn "fetch-user"

        condition { ctx ->
            def user = ctx.prev
            user.type == "premium"
        }

        whenTrue {
            task("premium-processing") {
                action { ctx ->
                    def user = ctx.taskResult("fetch-user")
                    return processPremiumUser(user)
                }
            }
        }

        whenFalse {
            task("standard-processing") {
                action { ctx ->
                    def user = ctx.taskResult("fetch-user")
                    return processStandardUser(user)
                }
            }
        }
    }

    // Merge paths
    task("finalize") {
        dependsOn "premium-processing", "standard-processing"

        action { ctx ->
            // One of these will have executed
            def result = ctx.taskResult("premium-processing") ?:
                ctx.taskResult("standard-processing")
        }
    }
}
```

```
        finalizeProcessing(result)
    }
}
```

Chapter 72. Example 4: Error Handling with Fallbacks

Handle errors gracefully with fallback strategies.

```
def resilientWorkflow = TaskGraph.build {
    // Primary data source
    httpTask("fetch-primary") {
        url "https://primary-api.example.com/data"
        timeout 5000

        circuitBreaker {
            failureThreshold 3
            timeout 30000
        }
    }

    // Fallback to cache
    task("try-cache") {
        condition { ctx ->
            // Only run if primary failed
            ctx.taskResult("fetch-primary")?.isFailure()
        }

        action { ctx ->
            log.warn("Primary source failed, trying cache")
            return cacheService.get("cached-data")
        }
    }

    // Final fallback to secondary source
    httpTask("fetch-secondary") {
        condition { ctx ->
            ctx.taskResult("fetch-primary")?.isFailure() &&
            ctx.taskResult("try-cache")?.isEmpty()
        }

        url "https://secondary-api.example.com/data"
        timeout 10000

        retryPolicy {
            maxAttempts 5
            delay 2000
        }
    }

    // Use whichever source worked
    task("process-data") {
        dependsOn "fetch-primary", "try-cache", "fetch-secondary"
    }
}
```

```
action { ctx ->
    def data = ctx.taskResult("fetch-primary")?.value ?:
        ctx.taskResult("try-cache")?.value ?:
        ctx.taskResult("fetch-secondary")?.value

    if (!data) {
        throw new NoDataException("All sources failed")
    }

    return processData(data)
}
}
```

Chapter 73. Example 5: Parallel Batch Processing

Process large datasets in parallel batches.

```
def batchWorkflow = TaskGraph.build {
    // Fetch all records
    task("fetch-records") {
        action {
            return recordService.findAll() // Returns 10,000 records
        }
    }

    // Create batches
    task("create-batches") {
        dependsOn "fetch-records"

        action { ctx ->
            def records = ctx.prev
            def batchSize = 100

            return records.collate(batchSize) // 100 batches of 100
        }
    }

    // Process batches in parallel
    task("process-batches") {
        dependsOn "create-batches"
        concurrencyLimit 10 // Max 10 batches at once

        action { ctx ->
            def batches = ctx.prev

            def promises = batches.withIndex().collect { batch, index ->
                Promises.task(pool) {
                    log.info("Processing batch ${index + 1}/${batches.size()}")
                    processBatch(batch)
                }
            }

            // Wait for all batches
            return Promises.all(promises).get()
        }
    }

    // Aggregate results
    task("aggregate-results") {
        dependsOn "process-batches"
```

```
action { ctx ->
    def batchResults = ctx.prev

    return [
        totalProcessed: batchResults.sum { it.count },
        successful: batchResults.count { it.success },
        failed: batchResults.count { !it.success },
        duration: System.currentTimeMillis() - startTime
    ]
}
}

def result = batchWorkflow.start().get()
println "Processed ${result.getTaskResult('aggregate-results').totalProcessed} records"
```

Chapter 74. Example 6: Saga Pattern (Distributed Transaction)

Implement compensating transactions for distributed operations.

```
def bookingSaga = TaskGraph.build {
    // Step 1: Reserve flight
    task("reserve-flight") {
        action { ctx ->
            def booking = ctx.global("booking")
            return flightService.reserve(booking.flightId)
        }

        onError { ctx, error ->
            // No compensation needed - reservation not made
            log.error("Flight reservation failed", error)
        }
    }

    // Step 2: Reserve hotel
    task("reserve-hotel") {
        dependsOn "reserve-flight"

        action { ctx ->
            def booking = ctx.global("booking")
            return hotelService.reserve(booking.hotelId)
        }

        onError { ctx, error ->
            // Compensate: Cancel flight
            def flightReservation = ctx.taskResult("reserve-flight")
            flightService.cancel(flightReservation.id)
            log.error("Hotel reservation failed, cancelled flight", error)
        }
    }

    // Step 3: Charge payment
    task("charge-payment") {
        dependsOn "reserve-hotel"

        action { ctx ->
            def booking = ctx.global("booking")
            return paymentService.charge(booking.amount, booking.cardId)
        }

        onError { ctx, error ->
            // Compensate: Cancel hotel and flight
            def hotelReservation = ctx.taskResult("reserve-hotel")
            def flightReservation = ctx.taskResult("reserve-flight")
        }
    }
}
```

```

        hotelService.cancel(hotelReservation.id)
        flightService.cancel(flightReservation.id)

        log.error("Payment failed, cancelled all reservations", error)
    }
}

// Step 4: Confirm booking
task("confirm-booking") {
    dependsOn "charge-payment"

    action { ctx ->
        def flightRes = ctx.taskResult("reserve-flight")
        def hotelRes = ctx.taskResult("reserve-hotel")
        def payment = ctx.taskResult("charge-payment")

        return bookingService.confirm(
            flightRes.id,
            hotelRes.id,
            payment.transactionId
        )
    }

    onError { ctx, error ->
        // Compensate: Refund and cancel all
        def payment = ctx.taskResult("charge-payment")
        paymentService.refund(payment.transactionId)

        def hotelRes = ctx.taskResult("reserve-hotel")
        def flightRes = ctx.taskResult("reserve-flight")

        hotelService.cancel(hotelRes.id)
        flightService.cancel(flightRes.id)

        log.error("Confirmation failed, rolled back all", error)
    }
}
}

```

Chapter 75. Example 7: API Orchestration

Orchestrate multiple API calls with dependencies.

```
def apiOrchestration = TaskGraph.build {
    // Authenticate
    httpTask("authenticate") {
        url "https://auth.example.com/token"
        method POST
        json {
            client_id: config("auth.clientId")
            client_secret: credential("auth-secret")
            grant_type: "client_credentials"
        }
    }

    // Fetch user profile (needs auth)
    httpTask("fetch-profile") {
        dependsOn "authenticate"

        url "https://api.example.com/profile"
        method GET

        header { ctx ->
            def token = ctx.taskResult("authenticate").access_token
            return ["Authorization": "Bearer ${token}"]
        }
    }

    // Fetch user orders (parallel with profile)
    httpTask("fetch-orders") {
        dependsOn "authenticate"

        url "https://api.example.com/orders"
        method GET

        header { ctx ->
            def token = ctx.taskResult("authenticate").access_token
            return ["Authorization": "Bearer ${token}"]
        }
    }

    // Fetch recommendations (parallel)
    httpTask("fetch-recommendations") {
        dependsOn "authenticate"

        url "https://api.example.com/recommendations"
        method GET

        header { ctx ->
```

```

        def token = ctx.taskResult("authenticate").access_token
        return ["Authorization": "Bearer ${token}"]
    }
}

// Combine all data
task("combine-data") {
    dependsOn "fetch-profile", "fetch-orders", "fetch-recommendations"

    action { ctx ->
        return [
            profile: ctx.taskResult("fetch-profile"),
            orders: ctx.taskResult("fetch-orders"),
            recommendations: ctx.taskResult("fetch-recommendations"),
            timestamp: new Date()
        ]
    }
}

// Cache combined data
task("cache-data") {
    dependsOn "combine-data"

    action { ctx ->
        def data = ctx.prev
        cacheService.put("user-data", data, ttl: 3600)
        return "Cached"
    }
}
}

```

Chapter 76. Example 8: File Processing Pipeline

Process files with validation and transformation.

```
def fileProcessor = TaskGraph.build {
    // Read input file
    fileTask("read-input") {
        operation READ
        sources(["/data/input/customers.csv"])

        securityConfig {
            allowedDirectories(["/data/input"])
            maxFileSize 100 * 1024 * 1024 // 100 MB
        }
    }

    // Parse CSV
    task("parse-csv") {
        dependsOn "read-input"

        action { ctx ->
            def content = ctx.prev
            return CSVParser.parse(content)
        }
    }

    // Validate records
    task("validate-records") {
        dependsOn "parse-csv"

        action { ctx ->
            def records = ctx.prev

            return records.collect { record ->
                def errors = []

                if (!record.email?.matches(/.*@.*/)) {
                    errors << "Invalid email"
                }
                if (!record.phone?.matches(/\d{10}/)) {
                    errors << "Invalid phone"
                }

                return [
                    record: record,
                    valid: errors.isEmpty(),
                    errors: errors
                ]
            }
        }
    }
}
```

```

        }
    }

// Split valid/invalid
task("split-records") {
    dependsOn "validate-records"

    action { ctx ->
        def validated = ctx.prev

        return [
            valid: validated.findAll { it.valid }.collect { it.record },
            invalid: validated.findAll { !it.valid }
        ]
    }
}

// Process valid records
task("process-valid") {
    dependsOn "split-records"

    action { ctx ->
        def split = ctx.prev
        def valid = split.valid

        // Transform and enrich
        return valid.collect { record ->
            enrichCustomerData(record)
        }
    }
}

// Write valid records
fileTask("write-valid") {
    dependsOn "process-valid"

    operation WRITE
    destination "/data/output/customers-processed.json"
    content { ctx -> JsonOutput.toJson(ctx.prev) }

    securityConfig {
        allowedDirectories(["/data/output"])
    }
}

// Write error report
task("write-errors") {
    dependsOn "split-records"

    action { ctx ->

```

```
def split = ctx.taskResult("split-records")
def invalid = split.invalid

def report = invalid.collect { record ->
    "${record.record.id}: ${record.errors.join(', ')}"
}.join("\n")

new File("/data/output/errors.txt").text = report
return "${invalid.size()} errors written"
}
```

Chapter 77. Example 9: Periodic Job with Scheduling

Schedule recurring workflow execution.

```
import java.util.concurrent.*

// Define workflow
def dailyReport = TaskGraph.build {
    task("fetch-daily-data") {
        action {
            def yesterday = new Date() - 1
            return dataService.fetchByDate(yesterday)
        }
    }

    task("generate-report") {
        dependsOn "fetch-daily-data"

        action { ctx ->
            def data = ctx.prev
            return reportGenerator.create(data)
        }
    }

    task("send-email") {
        dependsOn "generate-report"

        mailTask {
            to config("report.recipients")
            subject "Daily Report - ${new Date().format('yyyy-MM-dd')}"
            attachments { ctx -> [ctx.prev] }
        }
    }
}

// Schedule daily execution
def scheduler = Executors.newScheduledThreadPool(1)

scheduler.scheduleAtFixedRate(
{
    try {
        log.info("Starting daily report")
        def result = dailyReport.start().get()
        log.info("Daily report completed: ${result.status}")
    } catch (Exception e) {
        log.error("Daily report failed", e)
        notifyOps(e)
    }
}
```

```
},
0,                                // Initial delay
1,                                // Period
TimeUnit.DAYS                      // Unit
)
```

Chapter 78. Example 10: Dynamic Workflow Construction

Build workflow dynamically based on configuration.

```
def buildDynamicWorkflow(Config config) {
    return TaskGraph.build {
        // Create tasks for each data source
        config.dataSources.each { source ->
            httpTask("fetch-${source.id}") {
                url source.url
                method GET
                timeout source.timeout ?: 30000

                if (source.requiresAuth) {
                    header "Authorization", "Bearer
${credential(source.credentialId)}"
                }
            }
        }

        // Create transformation tasks
        config.transformations.each { transform ->
            task("transform-${transform.id}") {
                // Depend on source task
                dependsOn "fetch-${transform.sourceId}"

                action { ctx ->
                    def data = ctx.prev
                    applyTransformation(data, transform.rules)
                }
            }
        }

        // Create loading tasks
        config.destinations.each { dest ->
            task("load-${dest.id}") {
                // Depend on corresponding transformation
                dependsOn "transform-${dest.transformId}"

                action { ctx ->
                    def data = ctx.prev
                    loadToDestination(data, dest)
                }
            }
        }
    }
}
```

```
// Load configuration
def config = ConfigLoader.load("workflow-config.yml")

// Build and execute workflow
def workflow = buildDynamicWorkflow(config)
def result = workflow.start().get()
```

Chapter 79. Common Patterns Summary

Pattern	Use Case	Key Features
ETL Pipeline	Data extraction, transformation, loading	Sequential dependencies, error handling
Fan-Out / Fan-In	Parallel processing, aggregation	Parallel execution, result combination
Conditional Routing	Branch based on runtime data	Exclusive gateways, dynamic paths
Fallback Chain	Resilience, fault tolerance	Multiple data sources, graceful degradation
Batch Processing	Large dataset processing	Parallel batches, concurrency limits
Saga Pattern	Distributed transactions	Compensation, rollback logic
API Orchestration	Multiple API coordination	Authentication, parallel calls
File Processing	File validation, transformation	Read/write, validation, error handling
Scheduled Jobs	Recurring workflows	Periodic execution, error notification
Dynamic Workflows	Config-driven workflows	Runtime construction, flexibility

Chapter 80. Next Steps

- **Chapter 8** - Test these patterns with the test harness
- **Chapter 9** - Add security to workflows
- **Chapter 10** - Monitor workflow execution

TaskGraph Test Harness

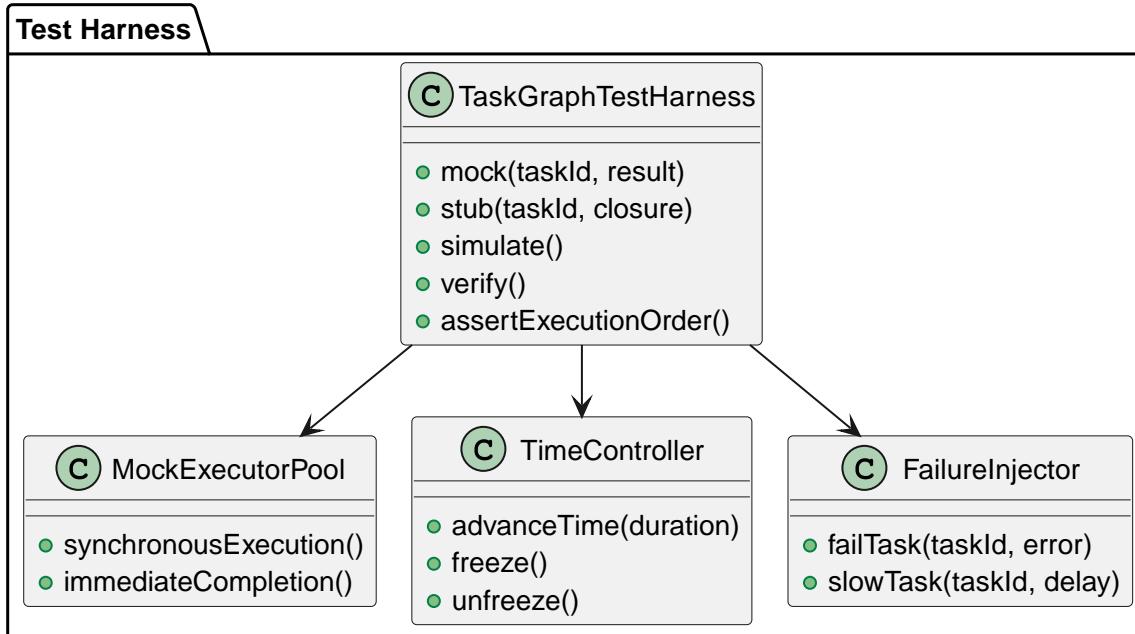
This chapter covers TaskGraph’s comprehensive test harness, designed to make testing workflows easy, fast, and reliable. Learn how to unit test tasks, integration test workflows, and validate complex orchestration logic.

Chapter 81. Overview

TaskGraph provides a dedicated test harness with:

- **Task mocking** - Mock individual tasks for unit testing
- **Workflow simulation** - Simulate execution without side effects
- **Time control** - Fast-forward time for timeout testing
- **Failure injection** - Test error handling paths
- **Execution verification** - Assert execution order and results
- **Performance profiling** - Measure workflow performance

Chapter 82. Test Harness Architecture



Chapter 83. Setting Up Tests

83.1. Test Dependencies

```
// build.gradle
dependencies {
    testImplementation 'org.softwood.dag:taskgraph-test:2.0.0'
    testImplementation 'org.spockframework:spock-core:2.3-groovy-3.0'
}
```

83.2. Basic Test Structure

```
import org.softwood.dag.test.*
import spock.lang.Specification

class WorkflowTest extends Specification {

    TaskGraphTestHarness harness

    def setup() {
        harness = new TaskGraphTestHarness()
    }

    def cleanup() {
        harness.reset()
    }

    def "test workflow execution"() {
        given:
        def workflow = buildWorkflow()

        when:
        def result = harness.execute(workflow)

        then:
        result.status == SUCCESS
        result.executedTaskCount == 5
    }
}
```

Chapter 84. Unit Testing Tasks

84.1. Testing Individual Tasks

```
class TaskUnitTest extends Specification {

    def "test service task logic"() {
        given:
        def context = new TaskContext()
        context.global("userId", 123)

        def task = new ServiceTask("fetch-user", context) {
            action { ctx ->
                def userId = ctx.global("userId")
                return userService.getUser(userId)
            }
        }

        when:
        def result = task.start().get()

        then:
        result != null
        result.id == 123
    }

    def "test task with retry"() {
        given:
        def attemptCount = 0
        def task = new ServiceTask("flaky-task", context) {
            retryPolicy {
                maxAttempts 3
                delay 100
            }

            action {
                attemptCount++
                if (attemptCount < 3) {
                    throw new RuntimeException("Temporary failure")
                }
                return "Success"
            }
        }

        when:
        def result = task.start().get()

        then:
        result == "Success"
    }
}
```

```

        attemptCount == 3
    }

def "test task timeout"() {
    given:
    def task = new ServiceTask("slow-task", context) {
        timeout 1000

        action {
            Thread.sleep(5000)
            return "Should not reach here"
        }
    }

    when:
    task.start().get()

    then:
    thrown(TimeoutException)
}
}

```

84.2. Testing Custom Tasks

```

class CustomTaskTest extends Specification {

    def "test elasticsearch task"() {
        given:
        def mockClient = Mock(ElasticsearchClient)
        def task = new ElasticsearchTask("search", context)
        task.client = mockClient
        task.index("users")
        task.query("status:active")

        when:
        def result = task.start().get()

        then:
        1 * mockClient.search({ SearchRequest req ->
            req.indices() == ["users"]
            req.source().query().queryString().queryString() == "status:active"
        }) >> mockSearchResponse

        result.totalHits == 100
    }
}

```

Chapter 85. Integration Testing Workflows

85.1. Testing Complete Workflows

```
class WorkflowIntegrationTest extends Specification {

    def "test ETL pipeline"() {
        given:
        def workflow = TaskGraph.build {
            httpTask("extract") {
                url "https://api.example.com/data"
            }

            task("transform") {
                dependsOn "extract"
                action { ctx -> transformData(ctx.prev) }
            }

            sqlTask("load") {
                dependsOn "transform"
                datasource testDataSource
                query "INSERT INTO results VALUES (?)"
            }
        }

        when:
        def result = workflow.start().get()

        then:
        result.status == SUCCESS
        result.getTaskResult("extract").value != null
        result.getTaskResult("transform").value != null
        result.getTaskResult("load").rowsAffected > 0
    }
}
```

85.2. Using Test Harness

```
class WorkflowHarnessTest extends Specification {

    TaskGraphTestHarness harness = new TaskGraphTestHarness()

    def "test workflow with mocks"() {
        given:
        def workflow = buildComplexWorkflow()

        // Mock external service calls
    }
}
```

```

harness.mock("fetch-users", [
    [id: 1, name: "Alice"],
    [id: 2, name: "Bob"]
])

harness.mock("fetch-orders", [
    [id: 101, userId: 1, total: 50.00],
    [id: 102, userId: 2, total: 75.00]
])

when:
def result = harness.execute(workflow)

then:
result.status == SUCCESS
harness.verifyExecuted("fetch-users", "fetch-orders", "join-data")
harness.verifyExecutionOrder("fetch-users", "join-data")
harness.verifyExecutionOrder("fetch-orders", "join-data")
}

def "test workflow with stubs"() {
given:
def workflow = buildWorkflow()

// Stub with dynamic behavior
harness.stub("process-data") { ctx ->
    def input = ctx.prev
    return input.collect { it * 2 }
}

when:
def result = harness.execute(workflow)

then:
result.getTaskResult("process-data").value == [2, 4, 6, 8]
}
}

```

Chapter 86. Failure Injection Testing

86.1. Testing Error Handling

```
class ErrorHandlingTest extends Specification {

    TaskGraphTestHarness harness = new TaskGraphTestHarness()

    def "test task failure handling"() {
        given:
        def workflow = buildWorkflow()

        // Inject failure
        harness.failTask("fetch-data", new IOException("Network error"))

        when:
        def result = harness.execute(workflow)

        then:
        result.status == FAILED
        result.getTaskResult("fetch-data").isFailure()
        result.getTaskResult("fetch-data").error instanceof IOException
    }

    def "test retry on failure"() {
        given:
        def workflow = TaskGraph.build {
            task("flaky-task") {
                retryPolicy {
                    maxAttempts 3
                    delay 1000
                }
                action { riskyOperation() }
            }
        }

        def attempts = 0
        harness.stub("flaky-task") {
            attempts++
            if (attempts < 3) {
                throw new RuntimeException("Temporary error")
            }
            return "Success"
        }

        when:
        def result = harness.execute(workflow)

        then:
```

```

        result.status == SUCCESS
        attempts == 3
        harness.getTaskAttempts("flaky-task") == 3
    }

def "test fallback on failure"() {
    given:
    def workflow = TaskGraph.build {
        task("primary") {
            action { fetchFromPrimary() }
        }

        task("fallback") {
            condition { ctx ->
                ctx.taskResult("primary")?.isFailure()
            }
            action { fetchFromFallback() }
        }

        task("process") {
            dependsOn "primary", "fallback"
            action { ctx ->
                def data = ctx.taskResult("primary")?.value ?:
                    ctx.taskResult("fallback")?.value
                return processData(data)
            }
        }
    }

    harness.failTask("primary", new TimeoutException())
    harness.mock("fallback", mockData)

    when:
    def result = harness.execute(workflow)

    then:
    result.status == SUCCESS
    result.getTaskResult("primary").isFailure()
    result.getTaskResult("fallback").isSuccess()
    result.getTaskResult("process").value != null
}
}

```

86.2. Testing Circuit Breaker

```

def "test circuit breaker opens on failures"() {
    given:
    def workflow = TaskGraph.build {
        task("external-api") {

```

```

        circuitBreaker {
            failureThreshold 3
            timeout 30000
        }
        action { callExternalAPI() }
    }
}

// Inject failures
harness.alwaysFail("external-api", new IOException("Service unavailable"))

when: "Execute multiple times"
5.times {
    try {
        harness.execute(workflow)
    } catch (Exception ignored) {}
}

then: "Circuit breaker opens after 3 failures"
harness.getCircuitBreakerState("external-api") == OPEN
harness.getTaskFailureCount("external-api") == 3 // Only 3 attempts, then circuit
open
}

```

Chapter 87. Time-Based Testing

87.1. Fast-Forward Time

```
class TimeBasedTest extends Specification {

    def "test timeout behavior"() {
        given:
        def workflow = TaskGraph.build {
            task("slow-task") {
                timeout 5000
                action {
                    Thread.sleep(10000)
                    return "Result"
                }
            }
        }

        def harness = new TaskGraphTestHarness()
        harness.timeControl.freeze() // Freeze time

        when:
        def promise = harness.executeAsync(workflow)

        // Fast-forward past timeout
        harness.timeControl.advanceTime(6000)

        def result = promise.get()

        then:
        result.status == FAILED
        result.getTaskResult("slow-task").error instanceof TimeoutException
    }

    def "test retry delay"() {
        given:
        def workflow = TaskGraph.build {
            task("retry-task") {
                retryPolicy {
                    maxAttempts 3
                    delay 5000 // 5 second delay between retries
                }
                action { riskyOperation() }
            }
        }

        def harness = new TaskGraphTestHarness()
        harness.timeControl.freeze()
    }
}
```

```

def attempts = 0
harness.stub("retry-task") {
    attempts++
    if (attempts < 3) {
        throw new RuntimeException("Fail")
    }
    return "Success"
}

when:
def promise = harness.executeAsync(workflow)

// Fast-forward through retries
3.times {
    harness.timeControl.advanceTime(5000)
}

def result = promise.get()

then:
result.status == SUCCESS
attempts == 3
// Test completed in milliseconds instead of 10+ seconds
}
}

```

87.2. Testing Scheduled Workflows

```

def "test scheduled execution"() {
    given:
    def workflow = buildWorkflow()
    def harness = new TaskGraphTestHarness()
    harness.timeControl.freeze()

    def executionCount = 0
    def scheduler = harness.mockScheduler()

    scheduler.scheduleAtFixedRate(
        { workflow.start().get(); executionCount++ },
        0,
        3600, // Every hour
        TimeUnit.SECONDS
    )

    when: "Advance 5 hours"
    5.times {
        harness.timeControl.advanceTime(3600, TimeUnit.SECONDS)
    }
}

```

```
    then: "Workflow executed 5 times"
    executionCount == 5
}
```

Chapter 88. Execution Verification

88.1. Verifying Task Execution

```
class ExecutionVerificationTest extends Specification {

    def "verify all tasks executed"() {
        given:
        def workflow = buildComplexWorkflow()
        def harness = new TaskGraphTestHarness()

        when:
        harness.execute(workflow)

        then:
        harness.verifyExecuted("task-a", "task-b", "task-c")
        harness.verifyNotExecuted("optional-task")
    }

    def "verify execution order"() {
        given:
        def workflow = TaskGraph.build {
            task("task-a") { action { "A" } }
            task("task-b") {
                dependsOn "task-a"
                action { "B" }
            }
            task("task-c") {
                dependsOn "task-b"
                action { "C" }
            }
        }
        def harness = new TaskGraphTestHarness()

        when:
        harness.execute(workflow)

        then:
        harness.verifyExecutionOrder("task-a", "task-b", "task-c")
        harness.verifyTaskExecutedBefore("task-a", "task-b")
        harness.verifyTaskExecutedAfter("task-c", "task-b")
    }

    def "verify parallel execution"() {
        given:
        def workflow = TaskGraph.build {
            task("task-a") { action { "A" } }
    }}
```

```

task("task-b") {
    dependsOn "task-a"
    action { "B" }
}

task("task-c") {
    dependsOn "task-a"
    action { "C" }
}

task("task-d") {
    dependsOn "task-b", "task-c"
    action { "D" }
}

def harness = new TaskGraphTestHarness()

when:
harness.execute(workflow)

then:
harness.verifyParallelExecution("task-b", "task-c")
harness.verifyTaskExecutedBefore("task-a", "task-b")
harness.verifyTaskExecutedBefore("task-a", "task-c")
harness.verifyTaskExecutedAfter("task-d", "task-b")
harness.verifyTaskExecutedAfter("task-d", "task-c")
}
}

```

88.2. Asserting Task Results

```

def "verify task results"() {
given:
def workflow = buildWorkflow()
def harness = new TaskGraphTestHarness()

when:
harness.execute(workflow)

then:
harness.assertTaskResult("fetch-data") { result ->
    result != null
    result.size() == 10
    result.every { it.id != null }
}

harness.assertTaskResult("transform") { result ->
    result.every { it.processed == true }
}

```

```
    }  
}
```

Chapter 89. Performance Testing

89.1. Measuring Execution Time

```
class PerformanceTest extends Specification {

    def "measure workflow execution time"() {
        given:
        def workflow = buildComplexWorkflow()
        def harness = new TaskGraphTestHarness()

        when:
        def result = harness.execute(workflow)

        then:
        result.durationMs < 5000 // Must complete in under 5 seconds

        harness.getTaskDuration("task-a") < 1000
        harness.getTaskDuration("task-b") < 2000
    }

    def "profile workflow performance"() {
        given:
        def workflow = buildWorkflow()
        def harness = new TaskGraphTestHarness()
        harness.enableProfiling()

        when:
        harness.execute(workflow)

        then:
        def profile = harness.getProfile()

        println "Total duration: ${profile.totalDuration}ms"
        println "Task breakdown:"
        profile.taskProfiles.each { task ->
            println " ${task.id}: ${task.duration}ms (${task.percentageOfTotal}%)"
        }

        // Assert no task takes more than 50% of total time
        profile.taskProfiles.every { it.percentageOfTotal < 50 }
    }
}
```

89.2. Load Testing

```
def "load test workflow"() {
```

```
given:  
def workflow = buildWorkflow()  
def harness = new TaskGraphTestHarness()  
  
def iterations = 1000  
def maxConcurrency = 50  
  
when:  
def pool = ExecutorPoolFactory.newFixedThreadPool(maxConcurrency)  
def startTime = System.currentTimeMillis()  
  
def futures = (1..iterations).collect {  
    pool.submit {  
        harness.execute(workflow)  
    }  
}  
  
futures*.get() // Wait for all  
def duration = System.currentTimeMillis() - startTime  
  
then:  
def throughput = (iterations / duration) * 1000  
println "Throughput: ${throughput} workflows/second"  
  
throughput > 100 // At least 100 workflows per second  
}
```

Chapter 90. Testing Conditional Logic

90.1. Testing Gateways

```
class GatewayTest extends Specification {

    def "test exclusive gateway routing"() {
        given:
        def workflow = TaskGraph.build {
            task("check") { action { 10 } }

            exclusiveGateway("route") {
                dependsOn "check"

                condition { ctx -> ctx.prev > 5 }
                whenTrue { task("high-path") { action { "HIGH" } } }
                whenFalse { task("low-path") { action { "LOW" } } }
            }
        }

        def harness = new TaskGraphTestHarness()

        when:
        harness.execute(workflow)

        then:
        harness.verifyExecuted("high-path")
        harness.verifyNotExecuted("low-path")
    }

    def "test parallel gateway"() {
        given:
        def workflow = TaskGraph.build {
            task("start") { action { "GO" } }

            parallelGateway("split") {
                dependsOn "start"

                fork {
                    task("path-a") { action { "A" } }
                    task("path-b") { action { "B" } }
                    task("path-c") { action { "C" } }
                }

                join {
                    task("merge") {
                        dependsOn "path-a", "path-b", "path-c"
                        action { ctx -> "MERGED" }
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}

def harness = new TaskGraphTestHarness()

when:
harness.execute(workflow)

then:
harness.verifyParallelExecution("path-a", "path-b", "path-c")
harness.verifyExecuted("merge")
}
}
```

Chapter 91. Testing Best Practices

91.1. Isolate External Dependencies

```
// ☐ Good - mock external services
def "test with mocked services"() {
    given:
    def mockHttpClient = Mock(HttpClient)
    def mockDatabase = Mock(Database)

    def workflow = TaskGraph.build {
        service(HttpClient.class, mockHttpClient)
        service(Database.class, mockDatabase)

        httpTask("fetch") { url "..." }
        sqlTask("store") { query "..." }
    }

    when:
    workflow.start().get()

    then:
    1 * mockHttpClient.execute(_)
    1 * mockDatabase.execute(_)
}
```

91.2. Use Test Data Builders

```
class TestDataBuilder {
    static TaskGraph buildTestWorkflow() {
        return TaskGraph.build {
            task("task-1") { action { "result-1" } }
            task("task-2") {
                dependsOn "task-1"
                action { "result-2" }
            }
        }
    }

    static TaskContext buildTestContext() {
        def context = new TaskContext()
        context.global("userId", 123)
        context.global("apiToken", "test-token")
        return context
    }
}
```

```
// Use in tests
def "test with builder"() {
    given:
    def workflow = TestDataBuilder.buildTestWorkflow()
    def context = TestDataBuilder.buildTestContext()

    // ... test code
}
```

91.3. Test One Thing Per Test

```
// ☐ Good - focused tests
def "test task A executes successfully"() { /* ... */ }
def "test task B depends on task A"() { /* ... */ }
def "test task C handles errors"() { /* ... */ }

// ☐ Bad - testing too much
def "test entire workflow"() {
    // Tests 20 different things
}
```

91.4. Use Descriptive Test Names

```
// ☐ Good - clear intent
def "should retry task up to 3 times on IOException"() { /* ... */ }
def "should skip task when condition is false"() { /* ... */ }
def "should execute tasks B and C in parallel after A completes"() { /* ... */ }

// ☐ Bad - unclear intent
def "test1"() { /* ... */ }
def "testWorkflow"() { /* ... */ }
```

Chapter 92. Summary

The TaskGraph test harness provides:

- **Task mocking** - Mock external dependencies
- **Failure injection** - Test error handling paths
- **Time control** - Fast-forward time for testing
- **Execution verification** - Assert execution order
- **Performance profiling** - Measure workflow performance
- **Integration testing** - Test complete workflows
- **Unit testing** - Test individual tasks

With the test harness, you can confidently test workflows of any complexity.

Chapter 93. Next Steps

- **Chapter 7** - More examples to test
- **Chapter 9** - Test security features
- **Chapter 10** - Test observability features

TaskGraphSpec Testing Framework

This chapter introduces TaskGraphSpec, a fluent, Spock-inspired testing framework for TaskGraph workflows. Learn how to write clean, expressive tests with minimal boilerplate using the given-when-then pattern.

Chapter 94. Overview

TaskGraphSpec provides a dedicated base class for testing TaskGraph workflows with a clean, expressive API inspired by Spock Framework. It eliminates common testing boilerplate and provides powerful verification capabilities.

94.1. Key Features

- **Fluent execution API** - Chain configuration methods for readable tests
- **Automatic task tracking** - No manual instrumentation required
- **Clean async handling** - No Awaitility or manual promise management
- **Execution order verification** - Assert task dependencies and parallelism
- **Input/output inspection** - Examine data flow through tasks
- **Timing verification** - Measure and assert execution performance
- **Mock support** - Stub task behavior for isolated testing
- **Failure testing** - Test error paths with `expectFailure()`

94.2. Design Philosophy

TaskGraphSpec follows these principles:

1. **Spock-style fluency** - Use given-when-then for clear test structure
2. **Minimal boilerplate** - Focus on what you're testing, not test infrastructure
3. **Type-safe** - Leverage Groovy's type system for compile-time safety
4. **JUnit Jupiter compatible** - Works with standard JUnit 5 tooling

Chapter 95. Getting Started

95.1. Setting Up Your Test Class

Extend `TaskGraphSpec` to get access to all testing capabilities:

```
import org.junit.jupiter.api.Test
import org.softwood.dag.test.TaskGraphSpec
import static org.junit.jupiter.api.Assertions.*

class MyWorkflowTest extends TaskGraphSpec {

    @Test
    void "should process data successfully"() {
        // given: A workflow
        def workflow = TaskGraph.build {
            serviceTask("process") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        [result: prev * 2]
                    }
                }
            }
        }

        // when: Workflow executes
        def result = execute(workflow)
            .with(10)
            .await()

        // then: Result is correct
        assertEquals(20, result.result)
        verifyTaskExecuted("process")
    }
}
```

95.2. Automatic Setup

`TaskGraphSpec` automatically provides:

- `ctx` - Fresh `TaskContext` for each test (via `@BeforeEach`)
- `verifier` - `TaskGraphVerifier` for assertions
- `currentExecution` - Execution wrapper for the current workflow

No manual setup or teardown required!

Chapter 96. Fluent Execution API

96.1. Basic Execution Pattern

The core pattern is: `execute(workflow).with(input).await()`

```
@Test
void "demonstrate basic execution"() {
    given: "a simple workflow"
    def workflow = createMyWorkflow()

    when: "workflow executes with input"
    def result = execute(workflow)
        .with([value: 42])
        .await()

    then: "result is returned"
    assertNotNull(result)
}
```

96.2. Configuration Options

Chain configuration methods before calling `await()`:

```
@Test
void "configure execution"() {
    def result = execute(workflow)
        .with(inputData)          // Set input
        .timeout(10000)           // Set timeout (ms)
        .await()                  // Execute and wait
}

// Alternative timeout syntax
def result2 = execute(workflow)
    .with(inputData)
    .timeout(10, TimeUnit.SECONDS)
    .await()
```

96.3. Expecting Failures

Test error handling paths cleanly:

```
@Test
void "handle expected failures"() {
    given: "a workflow that validates input"
```

```

def workflow = createValidationWorkflow()

when: "invalid input provided"
def result = execute(workflow)
    .with(null)
    .expectFailure()
    .await()

then: "failure is captured"
assertTrue(result.failed)
assertNotNull(result.error)
assertEquals("NullPointerException", result.error.type)
}

```

The result map for failures contains:

- **failed** - Boolean flag (always true)
- **error.message** - Error message string
- **error.type** - Exception class simple name
- **error.exception** - The actual exception object
- **executedTasks** - List of task IDs that ran
- **totalTime** - Execution duration in milliseconds

96.4. Convenience Methods

For simple cases, use `executeAndAwait()`:

```

@Test
void "use convenience method"() {
    // One-liner execution
    def result = executeAndAwait(workflow, inputData)

    // Equivalent to:
    // def result = execute(workflow).with(inputData).await()
}

```

Chapter 97. Verification API

97.1. Execution Order Verification

Verify tasks executed in the expected sequence:

```
@Test
void "verify execution order"() {
    given:
    def workflow = TaskGraph.build {
        serviceTask("step-1") {
            action { ctx, prev -> ctx.promiseFactory.executeAsync { "A" } }
        }
        serviceTask("step-2") {
            action { ctx, prev -> ctx.promiseFactory.executeAsync { "B" } }
        }
        serviceTask("step-3") {
            action { ctx, prev -> ctx.promiseFactory.executeAsync { "C" } }
        }

        chainVia("step-1", "step-2", "step-3")
    }

    when:
    execute(workflow).await()

    then: "tasks executed in chain order"
    verifyExecutionOrder("step-1", "step-2", "step-3")
}
```

97.2. Task Execution Checks

Verify which tasks ran or didn't run:

```
@Test
void "verify conditional execution"() {
    given:
    def workflow = createConditionalWorkflow()

    when:
    execute(workflow).with([condition: true]).await()

    then: "only expected path executed"
    verifyTaskExecuted("main-path")
    verifyTaskNotExecuted("alternate-path")
}
```

```

@Test
void "check all executed tasks"() {
    when:
    execute(workflow).await()

    then:
    def executedTasks = getExecutedTasks()
    assertEquals(5, executedTasks.size())
    assertTrue(executedTasks.contains("task-a"))
}

```

97.3. Task State Verification

Assert specific task completion states:

```

@Test
void "verify task states"() {
    when:
    execute(workflow).await()

    then: "all tasks completed successfully"
    assertTaskSucceeded("task-1")
    assertTaskSucceeded("task-2")
    assertTaskSucceeded("task-3")
}

@Test
void "verify partial failure"() {
    when:
    execute(workflow).with(invalidInput).expectFailure().await()

    then: "first task succeeded, second failed"
    assertTaskSucceeded("validate")
    assertTaskFailed("process")
}

```

97.4. Workflow Success/Failure

Check overall workflow outcome:

```

@Test
void "verify overall success"() {
    when:
    execute(workflow).with(validInput).await()

    then:
    verifySuccess()
}

```

```
}

@Test
void "verify overall failure"() {
    when:
    execute(workflow).with(invalidInput).expectFailure().await()

    then:
    verifyFailure()
}
```

Chapter 98. Input/Output Inspection

98.1. Task Output Verification

Examine data produced by each task:

```
@Test
void "verify task outputs"() {
    given:
    def workflow = TaskGraph.build {
        serviceTask("double") {
            action { ctx, prev ->
                ctx.promiseFactory.executeAsync {
                    [value: prev * 2]
                }
            }
        }
        serviceTask("add-ten") {
            action { ctx, prev ->
                ctx.promiseFactory.executeAsync {
                    [value: prev.value + 10]
                }
            }
        }
        chainVia("double", "add-ten")
    }

    when:
    execute(workflow).with(5).await()

    then: "can inspect each task's output"
    def doubleOutput = taskOutput("double")
    assertEquals(10, doubleOutput.value)

    def addOutput = taskOutput("add-ten")
    assertEquals(20, addOutput.value)
}
```

98.2. Task Input Inspection

Check what data each task received:

```
@Test
void "inspect task inputs"() {
    when:
    execute(workflow).with([id: 123, name: "test"]).await()
```

```
then:  
def taskInput = taskInput("process-user")  
assertEquals(123, taskInput.id)  
assertEquals("test", taskInput.name)  
}
```

98.3. Result Field Assertions

Helper methods for common result checks:

```
@Test  
void "assert result fields"() {  
    when:  
    def result = execute(workflow).with(input).await()  
  
    then: "result has expected fields"  
    assertResultHasField(result, "userId")  
    assertResultHasField(result, "processed")  
  
    and: "fields have expected values"  
    assertResultHas(result, "userId", 123)  
    assertResultHas(result, "processed", true)  
}
```

Chapter 99. Timing and Performance

99.1. Task Execution Time

Measure individual task performance:

```
@Test
void "measure task performance"() {
    when:
    execute(workflow).await()

    then: "fast task completes quickly"
    def fastTime = getTaskExecutionTime("fast-task")
    assertTrue(fastTime < 100, "Fast task took ${fastTime}ms")

    and: "slow task takes expected time"
    def slowTime = getTaskExecutionTime("slow-task")
    assertTrue(slowTime >= 1000, "Slow task took ${slowTime}ms")
}
```

99.2. Total Execution Time

Verify overall workflow performance:

```
@Test
void "verify total execution time"() {
    when:
    execute(workflow).await()

    then: "completes within SLA"
    def totalTime = getTotalExecutionTime()
    assertTrue(totalTime < 5000,
               "Workflow took ${totalTime}ms, exceeds 5s SLA")
}
```

99.3. Timeout Testing

Test timeout behavior:

```
@Test
void "test custom timeout"() {
    given: "a workflow with long-running tasks"
    def workflow = createSlowWorkflow()

    when: "executed with generous timeout"
```

```
def result = execute(workflow)
    .timeout(30, TimeUnit.SECONDS)
    .await()

    then: "completes successfully"
    assertNotNull(result)
}

@Test
void "test timeout failure"() {
    given: "a very slow workflow"
    def workflow = createVerySlowWorkflow()

    when: "executed with short timeout"
    execute(workflow)
        .timeout(100) // 100ms
        .await()

    then: "timeout exception thrown"
    thrown(AssertionError) // Wrapped TimeoutException
}
```

Chapter 100. Mock Support

100.1. Mocking Task Results

Stub task behavior for isolated testing:

```
@Test
void "mock external service call"() {
    given:
    def workflow = buildWorkflowWithExternalAPI()

    // Mock the external API task
    mockTask("fetch-from-api", [
        status: "success",
        data: [id: 1, name: "Test"]
    ])

    when:
    def result = execute(workflow).await()

    then: "uses mocked data"
    assertEquals("success", result.status)
    assertEquals(1, result.data.id)
}
```

100.2. Mocking Task Failures

Test error handling by mocking failures:

```
@Test
void "mock task failure"() {
    given:
    def workflow = buildWorkflowWithRetry()

    // Make external call fail
    mockTaskFailure("external-api",
        new IOException("Network timeout"))

    when:
    def result = execute(workflow)
        .expectFailure()
        .await()

    then: "error handled gracefully"
    assertTrue(result.failed)
    assertTrue(result.error.message.contains("Network timeout"))
```

```
}
```

100.3. Mock Storage

Mocks are stored in the TaskContext globals:

- `ctx.globals['mocks']` - Map of taskId → mock result
- `ctx.globals['mockFailures']` - Map of taskId → exception

This allows tasks to check for mocks during execution if designed to support it.

Chapter 101. Test Data Builders

101.1. Fluent Input Builder

Build test input data fluently:

```
@Test
void "use input builder"() {
    given:
    def testData = input()
        .with("userId", 123)
        .with("action", "process")
        .with("timestamp", System.currentTimeMillis())
        .build()

    when:
    def result = execute(workflow).with(testData).await()

    then:
    assertNotNull(result)
}

// Can also use call() syntax
@Test
void "builder with call syntax"() {
    def testData = input()
        .with("key", "value")
        .call() // Same as .build()
}
```

101.2. Custom Test Data Builders

Create domain-specific builders:

```
class CustomerWorkflowTest extends TaskGraphSpec {

    static class CustomerBuilder {
        private Map data = [:]

        CustomerBuilder withId(String id) {
            data.customerId = id
            return this
        }

        CustomerBuilder withName(String name) {
            data.customerName = name
            return this
        }
    }
}
```

```
}

CustomerBuilder withStatus(String status) {
    data.status = status
    return this
}

Map build() { return data }

static CustomerBuilder customer() {
    return new CustomerBuilder()
}

@Test
void "use custom builder"() {
    given:
    def customer = customer()
        .withId("C123")
        .withName("John Doe")
        .withStatus("active")
        .build()

    when:
    def result = execute(customerWorkflow)
        .with(customer)
        .await()

    then:
    assertEquals("active", result.status)
}
}
```

Chapter 102. Complete Examples

102.1. Example 1: Validation Workflow

```
class ValidationWorkflowTest extends TaskGraphSpec {

    static TaskGraph createValidationWorkflow() {
        TaskGraph.build {
            serviceTask("validate-not-null") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        def valid = prev != null
                        [valid: valid, value: prev, step: "null-check"]
                    }
                }
            }

            serviceTask("validate-range") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        def number = prev.value as Double
                        def inRange = number >= 0 && number <= 100
                        [valid: inRange, value: number, step: "range-check"]
                    }
                }
            }

            chainVia("validate-not-null", "validate-range")
        }
    }

    @Test
    void "validate successful input"() {
        given:
        def workflow = createValidationWorkflow()

        when:
        def result = execute(workflow).with("50").await()

        then:
        assertTrue(result.valid)
        assertEquals(50.0, result.value)
        assertEquals("range-check", result.step)
    }

    @Test
    void "validate out of range input"() {
        given:
        def workflow = createValidationWorkflow()
    }
}
```

```

when:
def result = execute(workflow).with("150").await()

then:
assertFalse(result.valid)
assertEquals(150.0, result.value)
}

@Test
void "validate null input"() {
    given:
    def workflow = createValidationWorkflow()

    when:
    def result = execute(workflow).with(null).await()

    then:
    assertFalse(result.valid)
}
}

```

102.2. Example 2: Transformation Pipeline

```

class TransformPipelineTest extends TaskGraphSpec {

    static TaskGraph createTransformWorkflow() {
        TaskGraph.build {
            serviceTask("parse") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        def value = prev instanceof Map ? prev.value : prev
                        [value: Double.parseDouble(value.toString()),
                         step: "parsed"]
                    }
                }
            }

            serviceTask("double-it") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        [value: prev.value * 2, step: "doubled"]
                    }
                }
            }

            serviceTask("add-ten") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {

```

```

                [value: prev.value + 10, step: "added"]
            }
        }

        serviceTask("format") {
            action { ctx, prev ->
                ctx.promiseFactory.executeAsync {
                    [result: "Result: ${prev.value}", step: "formatted"]
                }
            }
        }

        chainVia("parse", "double-it", "add-ten", "format")
    }
}

@Test
void "transform value through pipeline"() {
    given:
    def workflow = createTransformWorkflow()

    when:
    def result = execute(workflow).with("10").await()

    then: "final result is correct"
    assertEquals("Result: 30.0", result.result.toString())

    and: "tasks executed in order"
    verifyExecutionOrder("parse", "double-it", "add-ten", "format")
}

@Test
void "track intermediate values"() {
    given:
    def workflow = createTransformWorkflow()

    when:
    execute(workflow).with("5").await()

    then: "can verify each transformation"
    assertEquals(5.0, taskOutput("parse").value)
    assertEquals(10.0, taskOutput("double-it").value)
    assertEquals(20.0, taskOutput("add-ten").value)
    assertEquals("Result: 20.0", taskOutput("format").result.toString())
}

@Test
void "handle invalid input"() {
    given:
    def workflow = createTransformWorkflow()
}

```

```

when:
def result = execute(workflow)
    .with("not-a-number")
    .expectFailure()
    .await()

then:
assertTrue(result.failed)
assertTrue(result.error.message.contains("NumberFormatException") ||
           result.error.type.contains("NumberFormatException"))
}
}

```

102.3. Example 3: Conditional Workflow

```

class ConditionalWorkflowTest extends TaskGraphSpec {

    static TaskGraph createConditionalWorkflow() {
        TaskGraph.build {
            serviceTask("classify") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        def value = prev instanceof Number ?
                            prev : Double.parseDouble(prev.toString())
                        def category = value < 50 ? "small" :
                            value < 100 ? "medium" : "large"
                        [value: value, category: category]
                    }
                }
            }

            serviceTask("process") {
                action { ctx, prev ->
                    ctx.promiseFactory.executeAsync {
                        def category = prev['category']
                        [result: "${category.capitalize()}": ${prev['value']},
                         category: category,
                         processed: true]
                    }
                }
            }

            chainVia("classify", "process")
        }
    }

    @Test
    void "classify small value"() {

```

```

given:
def workflow = createConditionalWorkflow()

when:
def result = execute(workflow).with(25.0).await()

then:
assertEquals("small", result.category)
assertEquals("Small: 25.0", result.result)
}

@Test
void "classify medium value"() {
    given:
    def workflow = createConditionalWorkflow()

    when:
    def result = execute(workflow).with(75.0).await()

    then:
    assertEquals("medium", result.category)
    assertTrue(result.processed)
}

@Test
void "classify large value"() {
    given:
    def workflow = createConditionalWorkflow()

    when:
    def result = execute(workflow).with(150.0).await()

    then:
    assertEquals("large", result.category)
    verifyTaskExecuted("classify")
    verifyTaskExecuted("process")
}
}

```

Chapter 103. Best Practices

103.1. Write Focused Tests

Each test should verify one specific behavior:

```
// ☐ Good - focused test
@Test
void "should validate email format"() {
    // Test email validation only
}

@Test
void "should save user to database"() {
    // Test database save only
}

// ☐ Bad - testing too much
@Test
void "should validate and save user and send email and log event"() {
    // Testing 4 different things
}
```

103.2. Use Descriptive Test Names

Test names should describe the behavior being tested:

```
// ☐ Good - clear intent
@Test
void "should retry up to 3 times on network error"() { }

@Test
void "should skip optional task when input is null"() { }

@Test
void "should execute parallel tasks concurrently"() { }

// ☐ Bad - unclear intent
@Test
void "test1"() { }

@Test
void "testWorkflow"() { }
```

103.3. Organize Tests with given-when-then

Structure tests for readability:

```
@Test
void "example test structure"() {
    given: "setup context and preconditions"
    def workflow = buildWorkflow()
    def input = prepareTestData()

    when: "perform the action being tested"
    def result = execute(workflow).with(input).await()

    then: "verify the expected outcome"
    assertEquals(expectedValue, result.someField)
    verifyTaskExecuted("some-task")

    and: "additional verifications"
    assertTrue(result.someCondition)
}
```

103.4. Isolate External Dependencies

Mock external services and APIs:

```
@Test
void "mock external dependencies"() {
    given:
    def workflow = buildWorkflowWithExternalCalls()

    // Mock all external dependencies
    mockTask("fetch-from-api", mockApiResponse)
    mockTask("query-database", mockDbResult)
    mockTask("send-email", [sent: true])

    when:
    def result = execute(workflow).await()

    then:
    // Test workflow logic without actual external calls
    assertNotNull(result)
}
```

103.5. Test Both Success and Failure Paths

Always test error handling:

```

@Test
void "handle valid input successfully"() {
    when:
    def result = execute(workflow).with(validInput).await()

    then:
    verifySuccess()
}

@Test
void "handle invalid input gracefully"() {
    when:
    def result = execute(workflow)
        .with(invalidInput)
        .expectFailure()
        .await()

    then:
    verifyFailure()
}

```

103.6. Use Test Data Builders

Create reusable test data builders:

```

class WorkflowTestBase extends TaskGraphSpec {

    protected static Map validCustomer() {
        return [
            id: "C123",
            name: "John Doe",
            email: "john@example.com",
            status: "active"
        ]
    }

    protected static Map invalidCustomer() {
        return [
            id: null,
            name: "",
            email: "invalid-email"
        ]
    }
}

```

Chapter 104. Troubleshooting

104.1. Common Issues

104.1.1. Timeout Errors

If tests timeout unexpectedly:

```
// Increase timeout for slow workflows
execute(workflow)
    .timeout(30, TimeUnit.SECONDS) // Default is 5 seconds
    .await()
```

104.1.2. Task Not Executed

If `verifyTaskExecuted()` fails:

1. Check that task ID matches exactly (case-sensitive)
2. Verify task dependencies are satisfied
3. Check task condition evaluates to true
4. Use `getExecutedTasks()` to see which tasks actually ran

```
when:
execute(workflow).await()

then:
def executed = getExecutedTasks()
println "Executed tasks: ${executed}" // Debug output
```

104.1.3. Input Not Propagating

If task input is null or incorrect:

1. Verify workflow has root tasks (no predecessors)
2. Check that tasks properly pass output to successors
3. Inspect intermediate outputs with `taskOutput()`

```
then:
def firstOutput = taskOutput("first-task")
println "First task output: ${firstOutput}"

def secondInput = taskInput("second-task")
println "Second task input: ${secondInput}"
```

Chapter 105. API Reference

105.1. TaskGraphSpec Methods

105.1.1. Execution Methods

- `execute(workflow)` - Start fluent execution
- `executeAndAwait(workflow, input)` - Convenience method

105.1.2. Verification Methods

- `verifyExecutionOrder(taskIds…)` - Assert task execution order
- `verifyTaskExecuted(taskId)` - Assert task ran
- `verifyTaskNotExecuted(taskId)` - Assert task didn't run
- `verifySuccess()` - Assert workflow succeeded
- `verifyFailure()` - Assert workflow failed

105.1.3. Inspection Methods

- `taskInput(taskId)` - Get task input
- `taskOutput(taskId)` - Get task output
- `getTask(taskId)` - Get task instance
- `getExecutedTasks()` - Get list of executed task IDs

105.1.4. Timing Methods

- `getTaskExecutionTime(taskId)` - Get task duration (ms)
- `getTotalExecutionTime()` - Get workflow duration (ms)

105.1.5. Assertion Helpers

- `assertResultHas(result, field, value)` - Assert result field equals value
- `assertResultHasField(result, field)` - Assert result has field
- `assertTaskSucceeded(taskId)` - Assert task completed
- `assertTaskFailed(taskId)` - Assert task failed

105.1.6. Mock Methods

- `mockTask(taskId, result)` - Mock task success
- `mockTaskFailure(taskId, exception)` - Mock task failure

105.1.7. Test Data Methods

- `input()` - Create fluent input builder

105.2. TaskGraphExecution Methods

- `with(input)` - Set workflow input
- `timeout(ms)` - Set timeout in milliseconds
- `timeout(value, unit)` - Set timeout with TimeUnit
- `expectFailure()` - Expect workflow to fail
- `await()` - Execute and wait for result
- `start()` - Execute without waiting (returns Promise)

Chapter 106. Summary

TaskGraphSpec provides a powerful, expressive testing framework for TaskGraph workflows:

- **Fluent API** - Chain configuration for readable tests
- **Automatic tracking** - No manual instrumentation
- **Clean async** - No boilerplate promise handling
- **Comprehensive verification** - Order, state, timing, I/O
- **Mock support** - Isolate dependencies
- **Failure testing** - Test error paths easily
- **Spock-inspired** - Familiar given-when-then style
- **JUnit compatible** - Works with standard tooling

With TaskGraphSpec, you can write tests that are:

- **Readable** - Clear intent and structure
- **Maintainable** - Minimal boilerplate
- **Reliable** - Comprehensive verification
- **Fast** - No external dependencies required

Chapter 107. Next Steps

- **Chapter 7** - Study workflow examples to test
- **Chapter 8** - Compare with full test harness capabilities
- **Chapter 12** - Test gateway routing logic
- **Chapter 13** - Test different task types

Chapter 108. Part III: Operations

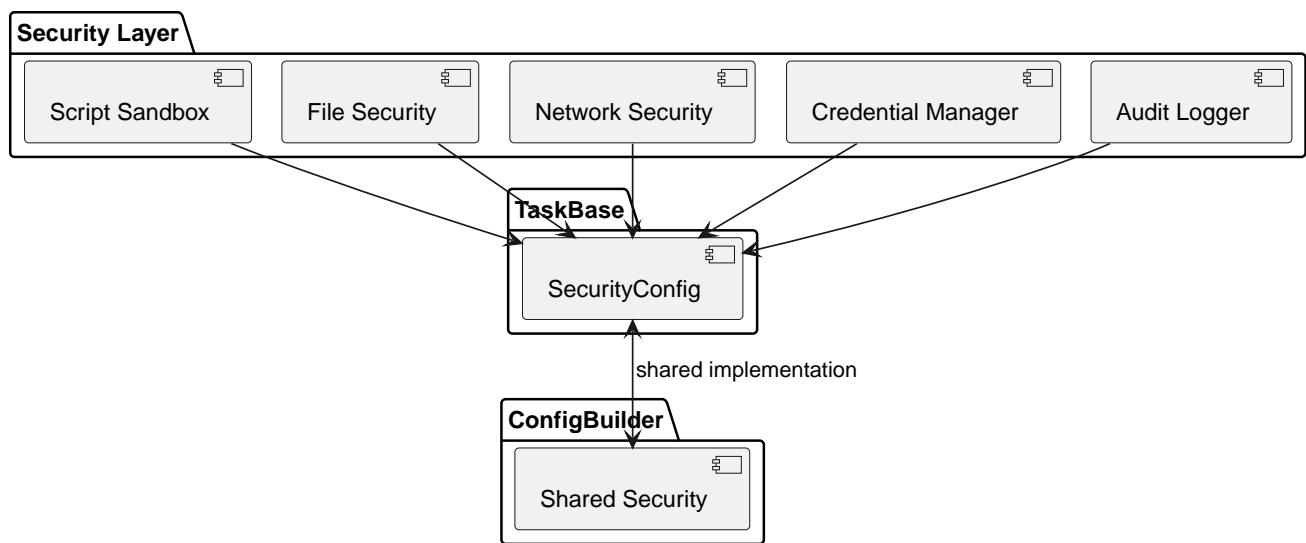
Security Architecture

This chapter covers TaskGraph's comprehensive security features including script sandboxing, credential management, file system access controls, SSRF prevention, and the shared security infrastructure with ConfigBuilder. Security is enabled by default with opt-out granularity.

Chapter 109. Overview

TaskGraph implements a "secure by default" architecture with multiple layers of protection:

- **Script sandboxing** - Restrict dangerous operations in scripts
- **File system security** - Control file access with allowlists
- **Network security** - Prevent SSRF and validate URLs
- **Credential management** - Secure credential storage and access
- **Custom script base classes** - Organization-specific security models
- **Audit logging** - Track security-relevant events
- **Resource limits** - Prevent resource exhaustion



Chapter 110. Script Sandboxing

TaskGraph sandboxes Groovy scripts by default to prevent malicious code execution.

110.1. Default Sandbox Behavior

```
scriptTask("safe-script") {
    // Sandboxed by default
    script '''
        // [] Allowed operations
        def result = [1, 2, 3].collect { it * 2 }
        println "Processing data"
        return result

        // [] Blocked operations (throw SecurityException)
        // System.exit(0)                                // JVM termination blocked
        // "rm -rf /.execute()                          // Process execution blocked
        // new File("/etc/passwd").text      // Unrestricted file access blocked
        // new URL("file:///etc/passwd")    // Local file URL blocked
    '''
}
```

110.2. Sandbox Configuration

```
scriptTask("custom-sandbox") {
    sandboxed true // Default

    // Configure allowed operations
    sandboxConfig {
        // Allow specific packages
        allowPackages([
            "java.util",
            "java.time",
            "org.mycompany.domain"
        ])

        // Allow specific classes
        allowClasses([
            "java.math.BigDecimal",
            "org.mycompany.Utils"
        ])

        // Deny specific methods
        denyMethods([
            "java.lang.System.exit",
            "java.lang.Runtime.exec"
        ])
}
```

```

// Allow specific methods despite package restrictions
allowMethods([
    "java.io.File.exists",
    "java.io.File.isDirectory"
])
}

script '''
import java.time.LocalDate
import org.mycompany.domain.Customer

def today = LocalDate.now()
def customer = new Customer(name: "Alice")

// Allowed because configured
return processCustomer(customer)
...
}

```

110.3. Disabling Sandbox (Not Recommended)

```

scriptTask("unsandboxed-script") {
    sandboxed false // ☠ Use with extreme caution

    script '''
        // Full Groovy access - no restrictions
        // Only use for trusted scripts
        def result = unsafeOperation()
        return result
    ...
}

```

110.4. Custom Script Base Classes

Provide custom security models for your organization:

```

// Define custom base class with allowed operations
abstract class CompanyScriptBase extends Script {

    // Approved utility methods available to all scripts
    def callApprovedAPI(String endpoint) {
        // Internal API with authentication handled
        def client = InternalAPIClient.withCredentials()
        return client.get(endpoint)
    }
}

```

```

def queryDataWarehouse(String sql) {
    // Pre-approved database access
    def ds = DataWarehouseAccess.getDataSource()
    return executeSafely(ds, sql)
}

// Block dangerous operations
@Override
Object invokeMethod(String name, Object args) {
    if (name in ['execute', 'exec', 'system']) {
        throw new SecurityException("Process execution not allowed")
    }
    return super.invokeMethod(name, args)
}

@Override
void setProperty(String name, Object value) {
    if (name.startsWith("sys")) {
        throw new SecurityException("System property modification not allowed")
    }
    super.setProperty(name, value)
}
}

// Use in scripts
scriptTask("company-script") {
    customScriptBaseClass CompanyScriptBase

    script '''
        // Company-approved methods available
        def data = queryDataWarehouse("SELECT * FROM sales")
        def enriched = callApprovedAPI("/enrich")

        return processData(data, enriched)
    '''
}

```

Chapter 111. File System Security

Control file access with granular permissions.

111.1. Allowed Directories

```
fileTask("read-data") {
    operation READ
    sources(["/data/input/file.txt"])

    securityConfig {
        // Only these directories allowed
        allowedDirectories([
            "/data/input",
            "/data/output",
            "/tmp/workspace"
        ])
    }

    // Explicitly deny directories (takes precedence)
    deniedDirectories([
        "/data/input/sensitive",
        "/etc",
        "/root"
    ])
}

// ☐ This would throw SecurityException
fileTask("blocked-read") {
    operation READ
    sources(["/etc/passwd"]) // Not in allowed directories

    securityConfig {
        allowedDirectories(["/data"])
    }
}
```

111.2. Path Traversal Prevention

```
// TaskGraph automatically validates paths
fileTask("safe-read") {
    operation READ

    // ☐ These paths are rejected
    // sources(["../../../etc/passwd"])          // Path traversal
    // sources(["/data ../../../etc/passwd"])     // Path traversal
    // sources(["/data/input/file.txt\u0000"])      // Null byte injection
```

```

securityConfig {
    allowedDirectories(["/data"])
    validatePaths true // Default: true
    rejectPathTraversal true // Default: true
    rejectSymlinks false // Default: false, set true for max security
}
}

```

111.3. File Size Limits

```

fileTask("size-limited") {
    operation READ
    sources(["/data/large-file.dat"])

    securityConfig {
        maxFileSize 100 * 1024 * 1024 // 100 MB max
        maxTotalSize 500 * 1024 * 1024 // 500 MB total for all files
    }
}

```

111.4. File Type Validation

```

fileTask("type-validated") {
    operation READ
    sources(["/data/document.pdf"])

    securityConfig {
        allowedExtensions([".pdf", ".txt", ".csv"])
        allowedMimeTypes([
            "application/pdf",
            "text/plain",
            "text/csv"
        ])

        validateMimeType true // Check actual content, not just extension
    }
}

```

Chapter 112. Network Security

Prevent SSRF (Server-Side Request Forgery) and other network attacks.

112.1. URL Validation

```
httpTask("safe-api-call") {
    url "https://api.example.com/data"

    securityConfig {
        // Allowed URL patterns
        allowedHosts([
            "api.example.com",
            "*.approved-domain.com",
            "internal-api.company.local"
        ])

        // Denied patterns (takes precedence)
        deniedHosts([
            "localhost",
            "127.0.0.1",
            "169.254.169.254", // AWS metadata service
            "*.internal",
            "10.*",           // Private networks
            "172.16.*",
            "192.168.*"
        ])

        // Protocol restrictions
        allowedProtocols(["https"]) // Only HTTPS
        denyInsecureProtocols true // Block HTTP

        // Port restrictions
        allowedPorts([443, 8443])
        denyCommonAttackPorts true // Block 22, 23, 3389, etc.
    }
}
```

112.2. SSRF Prevention

```
// TaskGraph blocks common SSRF attack vectors
httpTask("ssrf-protected") {
    securityConfig {
        preventSSRF true // Default: true

        // Automatically blocks:
        // - Private IP ranges (10.*, 192.168.*, 172.16-31.*)
```

```

    // - Loopback (127.* , localhost, ::1)
    // - Link-local (169.254.*)
    // - Cloud metadata endpoints
    // - DNS rebinding attacks
}

// □ These would be blocked
// url "http://localhost:8080/admin"
// url "http://169.254.169.254/latest/meta-data/"
// url "http://192.168.1.1/router-admin"
}

```

112.3. DNS Resolution Control

```

httpTask("dns-safe") {
    url "https://api.example.com/data"

    securityConfig {
        // Resolve and validate DNS before request
        validateDNS true

        // Reject if resolves to private IP
        rejectPrivateIPs true

        // Cache DNS results
        dnsCacheTTL 300 // 5 minutes
    }
}

```

112.4. Request Header Control

```

httpTask("header-controlled") {
    url "https://api.example.com/data"

    securityConfig {
        // Deny sensitive headers
        deniedHeaders([
            "X-Forwarded-For",
            "X-Real-IP",
            "Host",
            "Authorization" // Use credentialId instead
        ])

        // Require specific headers
        requiredHeaders([
            "User-Agent",
            "Content-Type"
        ])
}

```

```
])  
  
    // Validate header values  
    headerValidation {  
        "User-Agent" { value -> value.length() > 0 }  
        "Content-Type" { value -> value in ["application/json", "text/plain"] }  
    }  
}  
}
```

Chapter 113. Credential Management

Secure credential storage and access.

113.1. Credential Providers

```
// Multiple credential sources supported
def workflow = TaskGraph.build {
    task("use-credentials") {
        action { ctx ->
            // Environment variables
            def apiKey = ctx.credential("API_KEY")

            // AWS Secrets Manager
            def dbPassword = ctx.credential("aws:secretsmanager:prod/db/password")

            // HashiCorp Vault
            def sshKey = ctx.credential("vault:secret/ssh/prod")

            // Azure Key Vault
            def certificate = ctx.credential("azure:keyvault:ssl-cert")

            // Use credentials
            connectToDatabase(dbPassword)
        }
    }
}
```

113.2. Credential Injection

```
// Credentials automatically injected
sqlTask("database-query") {
    credentialId "database-credentials" // Injected automatically

    query "SELECT * FROM users WHERE id = ?"
    parameters([userId])
}

httpTask("api-call") {
    credentialId "api-credentials" // Used for Bearer token auth

    url "https://api.example.com/data"
}
```

113.3. Credential Masking

```
// Credentials masked in logs
task("process") {
    action { ctx ->
        def password = ctx.credential("db-password")

        // Logs show: "Password: ****" instead of actual value
        log.info("Password: $password")

        // Audit log shows: "Credential accessed: db-password"
        // Not the actual credential value
    }
}
```

113.4. Credential Rotation

```
def workflow = TaskGraph.build {
    // Configure credential rotation
    credentialConfig {
        provider "vault"
        rotationInterval 3600 // Rotate every hour
        cacheInvalidation true
    }

    task("use-rotating-credential") {
        action { ctx ->
            // Always gets current credential
            def token = ctx.credential("rotating-api-token")

            callAPI(token)
        }
    }
}
```

Chapter 114. Shared Security with ConfigBuilder

TaskGraph shares security infrastructure with ConfigBuilder.

114.1. Consistent Security Model

```
// Same security config in both TaskGraph and ConfigBuilder
def securityConfig = new SecurityConfig()
securityConfig.allowedDirectories(["/data", "/config"])
securityConfig.customScriptBaseClass(CompanyScriptBase)

// Use in ConfigBuilder
def config = ConfigBuilder.build {
    securityConfig(securityConfig)

    script '''
        // Same security model as TaskGraph
        def data = loadConfig()
        return data
    '''
}

// Use in TaskGraph
def workflow = TaskGraph.build {
    securityConfig(securityConfig) // Reuse same config

    scriptTask("process") {
        script '''
            // Same security restrictions
            def result = processData()
            return result
        '''
    }
}
```

114.2. Cross-Cutting Security

```
// Define organization-wide security
class OrganizationSecurity {
    static SecurityConfig standard() {
        new SecurityConfig(
            sandboxed: true,
            customScriptBaseClass: CompanyScriptBase,
            allowedDirectories: ["/data", "/config"],
            preventSSRF: true,
```

```

        credentialProvider: "vault"
    )
}

static SecurityConfig strict() {
    new SecurityConfig(
        sandboxed: true,
        customScriptBaseClass: StrictScriptBase,
        allowedDirectories: ["/data/approved"],
        deniedDirectories: ["/data/sensitive"],
        preventSSRF: true,
        rejectSymlinks: true,
        validateMimeTypes: true,
        credentialProvider: "vault",
        auditAll: true
    )
}
}

// Apply organization security
def workflow = TaskGraph.build {
    securityConfig(OrganizationSecurity.standard())

    // All tasks inherit security config
    scriptTask("task1") { script '''...''' }
    fileTask("task2") { operation READ }
    httpTask("task3") { url "..." }
}

// Apply strict security for sensitive workflows
def sensitiveWorkflow = TaskGraph.build {
    securityConfig(OrganizationSecurity.strict())

    // Stricter security applied
    scriptTask("sensitive-task") { script '''...''' }
}

```

Chapter 115. Audit Logging

Track security-relevant events.

115.1. Audit Configuration

```
def workflow = TaskGraph.build {
    auditConfig {
        enabled true
        level "INFO" // TRACE, DEBUG, INFO, WARN, ERROR

        // What to audit
        auditCredentialAccess true
        auditFileAccess true
        auditNetworkCalls true
        auditScriptExecution true
        auditSecurityViolations true

        // Where to log
        auditLogger "org.mycompany.security.AuditLogger"
        auditDestination "/var/log/taskgraph/audit.log"

        // Include in audit logs
        includeContext true
        includeStackTrace true
        includeUserInfo true
        includeTimestamp true
    }
}
```

115.2. Audit Events

```
// Audit events automatically logged
scriptTask("audited-task") {
    script '''
        // Logs: AUDIT [SCRIPT_EXECUTION] Task: audited-task, User: alice, Time: 2026-01-20T10:30:00
        def result = processData()

        // Logs: AUDIT [CREDENTIAL_ACCESS] Credential: api-key, Task: audited-task
        def key = bindings.ctx.credential("api-key")

        return result
    ...
}

// Custom audit events
```

```

task("custom-audit") {
    action { ctx ->
        ctx.audit("CUSTOM_EVENT", [
            action: "data-export",
            recordCount: 1000,
            destination: "external-system"
        ])
    }

    exportData()
}

```

115.3. Audit Log Format

```
{
    "timestamp": "2026-01-20T10:30:00.123Z",
    "level": "INFO",
    "eventType": "CREDENTIAL_ACCESS",
    "task": {
        "id": "fetch-data",
        "workflow": "daily-etl"
    },
    "user": {
        "id": "alice",
        "roles": ["data-engineer"]
    },
    "credential": {
        "id": "database-credentials",
        "type": "vault"
    },
    "context": {
        "environment": "production",
        "region": "us-east-1"
    },
    "success": true
}
```

Chapter 116. Resource Limits

Prevent resource exhaustion attacks.

116.1. Memory Limits

```
task("memory-limited") {
    resourceLimits {
        maxMemory 512 * 1024 * 1024 // 512 MB
        monitorMemory true
    }

    action {
        // TaskGraph monitors memory usage
        // Throws OutOfMemoryException if limit exceeded
        def largeData = processLargeDataset()
        return largeData
    }
}
```

116.2. Execution Time Limits

```
task("time-limited") {
    timeout 60000 // 60 seconds max
    hardTimeout true // Kill thread if exceeds (use cautiously)

    action {
        longRunningOperation()
    }
}
```

116.3. Concurrency Limits

```
TaskGraph.build {
    // Global concurrency limit
    maxConcurrency 50

    task("api-call") {
        // Task-level limit
        concurrencyLimit 10

        // Resource semaphore
        resource "external-api", max: 5

        action { callExternalAPI() }
    }
}
```

```
    }  
}
```

116.4. Rate Limiting

```
task("rate-limited") {  
    rateLimit {  
        maxRequests 100  
        perDuration 60 // 100 requests per minute  
        unit TimeUnit.SECONDS  
    }  
  
    action {  
        callAPI()  
    }  
}
```

Chapter 117. Security Best Practices

117.1. Principle of Least Privilege

```
// ☐ Good - minimal permissions
fileTask("read-config") {
    operation READ
    sources(["/config/app.yml"])

    securityConfig {
        allowedDirectories(["/config"])
        // Only what's needed
    }
}

// ☐ Bad - excessive permissions
fileTask("read-config") {
    operation READ
    sources(["/config/app.yml"])

    securityConfig {
        allowedDirectories(["/", "/etc", "/var", "/home"])
        // Too broad!
    }
}
```

117.2. Defense in Depth

```
// Multiple security layers
scriptTask("secure-task") {
    // Layer 1: Sandboxing
    sandboxed true
    customScriptBaseClass CompanySecureScriptBase

    // Layer 2: File restrictions
    securityConfig {
        allowedDirectories(["/data/approved"])
    }

    // Layer 3: Network restrictions
    networkSecurity {
        allowedHosts(["approved-api.company.com"])
        preventSSRF true
    }

    // Layer 4: Resource limits
    resourceLimits {
```

```

    maxMemory 256 * 1024 * 1024
    timeout 30000
}

// Layer 5: Audit logging
auditExecution true

script """
}

```

117.3. Secure Defaults

```

// TaskGraph uses secure defaults
scriptTask("default-secure") {
    // These are all defaults (explicit here for clarity):
    // sandboxed true
    // preventSSRF true
    // validatePaths true
    // rejectPathTraversal true
    // auditSecurityViolations true

    script """
}

```

117.4. Regular Security Audits

```

// Enable comprehensive auditing
def workflow = TaskGraph.build {
    auditConfig {
        enabled true
        level "INFO"
        auditAll true
    }

    // Periodically review audit logs
    task("security-review") {
        action {
            def violations = AuditLogAnalyzer.findViolations(
                since: new Date() - 7 // Last 7 days
            )

            if (violations) {
                notifySecurityTeam(violations)
            }
        }
    }
}

```

}

Chapter 118. Summary

TaskGraph provides comprehensive security:

- **Script sandboxing** - Prevent malicious code execution
- **File system security** - Control file access
- **Network security** - Prevent SSRF and validate URLs
- **Credential management** - Secure credential access
- **Custom base classes** - Organization-specific security
- **Audit logging** - Track security events
- **Resource limits** - Prevent resource exhaustion
- **Shared with ConfigBuilder** - Consistent security model
- **Secure by default** - Opt-out, not opt-in

Security is a first-class concern in TaskGraph, enabled by default and configurable for your organization's specific needs.

Chapter 119. Next Steps

- **Chapter 6** - TaskContext security features
- **Chapter 8** - Test security configurations
- **Appendix** - Complete security reference

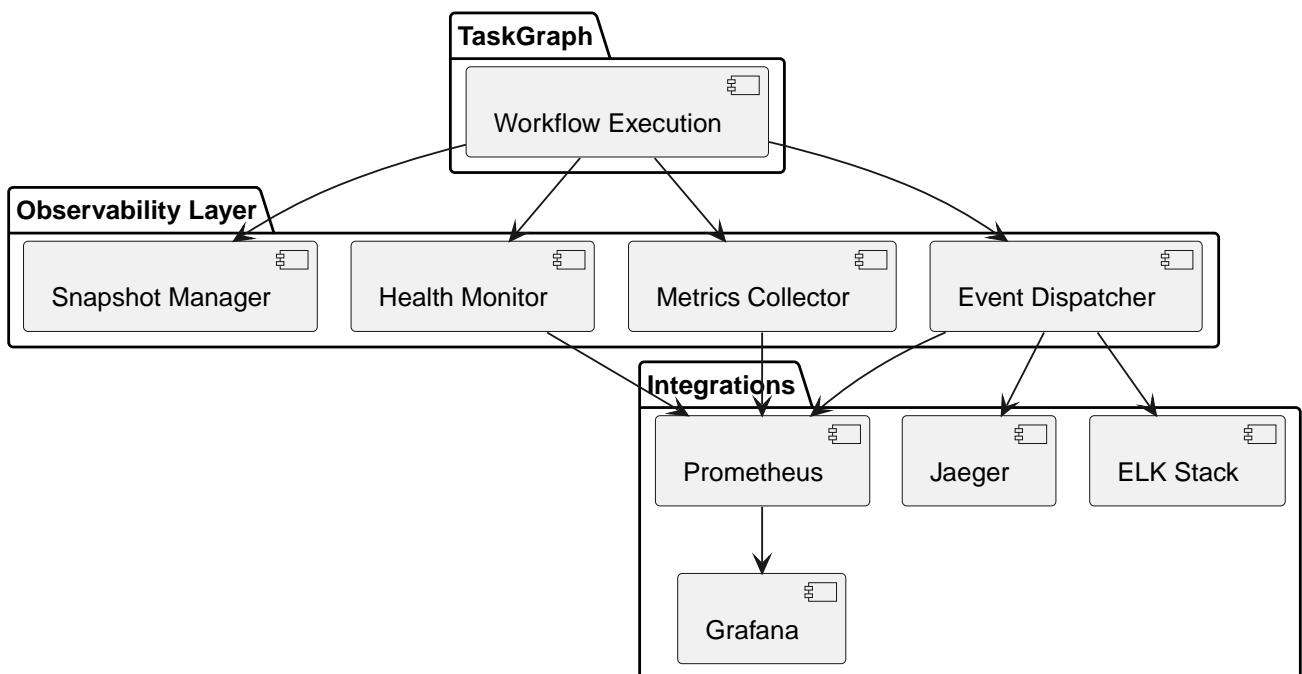
Observability and Monitoring

This chapter covers TaskGraph's observability features including event-driven monitoring, metrics collection, execution snapshots, health checks, and integration with monitoring systems. Learn how to watch and observe TaskGraph execution from your code.

Chapter 120. Overview

TaskGraph provides comprehensive observability through:

- **Event system** - Real-time workflow events
- **Execution snapshots** - Point-in-time workflow state
- **Health checks** - Workflow and task health monitoring
- **Metrics collection** - Performance and resource metrics
- **Distributed tracing** - Track execution across systems
- **Logging integration** - Structured logging support
- **Monitoring integrations** - Prometheus, Grafana, etc.



Chapter 121. Event System

TaskGraph emits events at every lifecycle stage.

121.1. Event Types

```
// Graph-level events
GRAPH_STARTED          // Workflow execution started
GRAPH_COMPLETED        // Workflow completed successfully
GRAPH_FAILED           // Workflow failed
GRAPH_PAUSED           // Workflow paused
GRAPH_RESUMED          // Workflow resumed
GRAPH_CANCELLED        // Workflow cancelled

// Task-level events
TASK_SCHEDULED         // Task scheduled for execution
TASK_STARTED           // Task execution started
TASK_COMPLETED          // Task completed successfully
TASK_FAILED             // Task failed
TASK_RETRYING           // Task retrying after failure
TASK_SKIPPED            // Task skipped (condition false)
TASK_TIMEOUT            // Task timed out

// Resilience events
CIRCUIT_OPENED          // Circuit breaker opened
CIRCUIT_HALF_OPEN       // Circuit breaker half-open
CIRCUIT_CLOSED           // Circuit breaker closed
RATE_LIMITED             // Task rate limited
RESOURCE_WARNING          // Resource usage warning

// Custom events
CUSTOM_EVENT             // User-defined events
```

121.2. Event Listeners

```
import org.softwood.dag.events.*

def workflow = TaskGraph.build {
    task("my-task") {
        action { "result" }
    }
}

// Add event listener
workflow.addListener(new GraphEventListener() {
    @Override
    void onEvent(GraphEvent event) {
```

```

switch (event.type) {
    case GRAPH_STARTED:
        println "Workflow started: ${event.graphId}"
        metricsCollector.workflowStarted(event.graphId)
        break

    case TASK_STARTED:
        def task = event.taskEvent
        println "Task started: ${task.taskName}"
        logToElastic(task)
        break

    case TASK_COMPLETED:
        def task = event.taskEvent
        println "Task completed: ${task.taskName} in ${task.durationMs}ms"
        metricsCollector.taskCompleted(task)
        break

    case TASK_FAILED:
        def task = event.taskEvent
        println "Task failed: ${task.taskName} - ${task.error}"
        alertOps(task)
        break

    case GRAPH_COMPLETED:
        println "Workflow completed in ${event.durationMs}ms"
        metricsCollector.workflowCompleted(event)
        break
    }
}
}

// Execute workflow
workflow.start()

```

121.3. Typed Event Handlers

```

// Register handlers for specific event types
workflow.on(TaskStartedEvent.class) { event ->
    println "Task ${event.taskName} started"
    startTimer(event.taskName)
}

workflow.on(TaskCompletedEvent.class) { event ->
    println "Task ${event.taskName} completed in ${event.durationMs}ms"
    stopTimer(event.taskName)
    recordMetric(event.taskName, event.durationMs)
}

```

```

workflow.on(TaskFailedEvent.class) { event ->
    log.error("Task ${event.taskName} failed", event.error)
    incrementFailureCounter(event.taskName)
    if (event.critical) {
        alertOps(event)
    }
}

workflow.on(CircuitBreakerEvent.class) { event ->
    log.warn("Circuit breaker ${event.state} for task ${event.taskName}")
    updateCircuitBreakerMetric(event.taskName, event.state)
}

```

121.4. Async Event Processing

```

// Process events asynchronously to avoid blocking workflow
def eventQueue = new LinkedBlockingQueue<GraphEvent>()
def eventProcessor = ExecutorPoolFactory.newSingleThreadExecutor()

workflow.addListener { event ->
    // Queue event for async processing
    eventQueue.offer(event)
}

// Process events in background
eventProcessor.submit {
    while (true) {
        def event = eventQueue.take()
        processEvent(event)
    }
}

```

121.5. Event Filtering

```

// Filter events before processing
workflow.addListener(new FilteredEventListener() {
    @Override
    boolean accept(GraphEvent event) {
        // Only process errors and warnings
        return event.severity in [SEVERITY.ERROR, SEVERITY.WARN]
    }

    @Override
    void onEvent(GraphEvent event) {
        handleCriticalEvent(event)
    }
})

```

```
// Task-specific filtering
workflow.addListener(new TaskEventFilter("critical-task")) {
    @Override
    void onEvent(TaskEvent event) {
        // Only events from "critical-task"
        monitorCriticalTask(event)
    }
}
```

Chapter 122. Execution Snapshots

Capture workflow state at any point in time.

122.1. Creating Snapshots

```
def workflow = TaskGraph.build {
    task("task-a") { action { "A" } }
    task("task-b") {
        dependsOn "task-a"
        action { "B" }
    }
    task("task-c") {
        dependsOn "task-b"
        action { "C" }
    }
}

// Start workflow async
def promise = workflow.start()

// Capture snapshot while running
Thread.sleep(100)
def snapshot = workflow.snapshot()

println "Snapshot at ${snapshot.timestamp}:"
println "  Status: ${snapshot.status}"
println "  Completed tasks: ${snapshot.completedTasks}"
println "  Running tasks: ${snapshot.runningTasks}"
println "  Pending tasks: ${snapshot.pendingTasks}"
println "  Progress: ${snapshot.progressPercentage}%"

// Wait for completion
promise.get()
```

122.2. ExecutionSnapshot API

```
interface ExecutionSnapshot {
    // Timestamp
    Instant getTimestamp()

    // Overall status
    ExecutionStatus getStatus() // RUNNING, COMPLETED, FAILED, PAUSED

    // Task counts
    int getTotalTaskCount()
    int getCompletedTaskCount()
```

```

int getRunningTaskCount()
int getPendingTaskCount()
int getFailedTaskCount()
int getSkippedTaskCount()

// Progress
double getProgressPercentage()
Duration getElapsedTime()
Duration getEstimatedRemainingTime()

// Task details
List<TaskSnapshot> getTaskSnapshots()
TaskSnapshot getTaskSnapshot(String taskId)

// Resource usage
ResourceUsage getResourceUsage()

// Export
String toJson()
Map<String, Object> toMap()
}

```

122.3. Periodic Snapshots

```

def workflow = buildComplexWorkflow()

// Capture snapshots periodically
def snapshotScheduler = Executors.newScheduledThreadPool(1)
def snapshots = []

snapshotScheduler.scheduleAtFixedRate(
{
    def snapshot = workflow.snapshot()
    snapshots.add(snapshot)

    println "Progress: ${snapshot.progressPercentage}%"
    println "Running: ${snapshot.runningTasks.join(',')}"

    // Send to monitoring system
    metricsCollector.recordSnapshot(snapshot)
},
0,
1, // Every second
TimeUnit.SECONDS
)

// Execute workflow
workflow.start().get()

```

```
// Stop snapshot collection
snapshotScheduler.shutdown()

// Analyze execution history
analyzeSnapshots(snapshots)
```

122.4. Snapshot Comparison

```
def snapshot1 = workflow.snapshot()
Thread.sleep(5000)
def snapshot2 = workflow.snapshot()

// Compare snapshots
def delta = snapshot2.compareTo(snapshot1)

println "Changes in last 5 seconds:"
println "  Tasks completed: ${delta.completedTasks}"
println "  Tasks started: ${delta.startedTasks}"
println "  Progress: +${delta.progressPercentage}%""
println "  Memory change: +${delta.memoryUsageMB} MB"
```

Chapter 123. Health Monitoring

Monitor workflow and task health.

123.1. HealthStatus API

```
def workflow = buildWorkflow()

// Check overall health
def health = workflow.health()

println "Workflow health: ${health.status}" // HEALTHY, DEGRADED, UNHEALTHY

if (!health.healthy) {
    println "Issues:"
    health.issues.each { issue ->
        println " - ${issue.severity}: ${issue.message}"
    }
}

// Task-level health
def taskHealth = workflow.getTask("my-task").health()
if (taskHealth.circuitBreakerOpen) {
    println "Circuit breaker open for my-task"
}

if (taskHealth.errorRate > 0.1) {
    println "High error rate: ${taskHealth.errorRate * 100}%"}
```

123.2. Health Checks

```
// Define custom health checks
workflow.addHealthCheck(new HealthCheck() {
    String getName() { "database-connectivity" }

    HealthCheckResult check() {
        try {
            def conn = dataSource.connection
            conn.close()
            return HealthCheckResult.healthy()
        } catch (Exception e) {
            return HealthCheckResult.unhealthy("Database unavailable: ${e.message}")
        }
    }
})
```

```

workflow.addHealthCheck(new HealthCheck() {
    String getName() { "external-api-reachability" }

    HealthCheckResult check() {
        try {
            def response = httpClient.get("https://api.example.com/health")
            if (response.statusCode == 200) {
                return HealthCheckResult.healthy()
            } else {
                return HealthCheckResult.degraded("API returned
${response.statusCode}")
            }
        } catch (Exception e) {
            return HealthCheckResult.unhealthy("API unreachable: ${e.message}")
        }
    }
}

// Execute health checks
def overallHealth = workflow.runHealthChecks()
println "Overall health: ${overallHealth.status}"
overallHealth.results.each { name, result ->
    println " ${name}: ${result.status}"
}

```

123.3. Health Endpoints

```

// Expose health endpoint (for load balancers, monitoring)
def healthEndpoint = new HealthEndpoint(workflow)

// HTTP endpoint
server.get("/health") { req, res ->
    def health = healthEndpoint.check()

    res.status(health.healthy ? 200 : 503)
    res.json([
        status: health.status,
        timestamp: new Date(),
        checks: health.results
    ])
}

// Liveness probe (is it running?)
server.get("/health/live") { req, res ->
    res.status(workflow.isRunning() ? 200 : 503)
    res.json([alive: workflow.isRunning()])
}

// Readiness probe (ready to accept work?)

```

```
server.get("/health/ready") { req, res ->
    def ready = workflow.isReady() &&
        !workflow.isPaused() &&
        workflow.health().healthy

    res.status(ready ? 200 : 503)
    res.json([ready: ready])
}
```

Chapter 124. Metrics Collection

Collect and export metrics about workflow execution.

124.1. Built-in Metrics

```
def workflow = buildWorkflow()

// Enable metrics collection
workflow.enableMetrics()

// Execute workflow
workflow.start().get()

// Access metrics
def metrics = workflow.getMetrics()

println "Execution Metrics:"
println "  Total duration: ${metrics.totalDurationMs}ms"
println "  Task count: ${metrics.totalTaskCount}"
println "  Success rate: ${metrics.successRate * 100}%"
println "  Average task duration: ${metrics.avgTaskDurationMs}ms"
println "  Max task duration: ${metrics.maxTaskDurationMs}ms"
println "  Parallel execution factor: ${metrics.parallelExecutionFactor}"

// Task-specific metrics
metrics.taskMetrics.each { taskId, taskMetrics ->
    println "\nTask: ${taskId}"
    println "  Executions: ${taskMetrics.executionCount}"
    println "  Failures: ${taskMetrics.failureCount}"
    println "  Avg duration: ${taskMetrics.avgDurationMs}ms"
    println "  Error rate: ${taskMetrics.errorRate * 100}%"
}
```

124.2. Custom Metrics

```
// Record custom metrics
task("process-data") {
    action { ctx ->
        def startTime = System.currentTimeMillis()
        def data = fetchData()

        // Record custom metrics
        ctx.recordMetric("data.fetch.count", data.size())
        ctx.recordMetric("data.fetch.duration", System.currentTimeMillis() -
        startTime)
    }
}
```

```

    def processed = processData(data)

    ctx.recordMetric("data.process.count", processed.size())

    return processed
}

}

// Access custom metrics
def customMetrics = workflow.getCustomMetrics()
println "Data fetched: ${customMetrics['data.fetch.count'].sum()}"
println "Avg fetch time: ${customMetrics['data.fetch.duration'].average()}ms"

```

124.3. Prometheus Integration

```

import io.prometheus.client.*

// Define Prometheus metrics
def workflowDuration = Histogram.build()
    .name("taskgraph_workflow_duration_seconds")
    .help("Workflow execution duration")
    .labelNames("workflow_id", "status")
    .register()

def taskDuration = Histogram.build()
    .name("taskgraph_task_duration_seconds")
    .help("Task execution duration")
    .labelNames("workflow_id", "task_id", "status")
    .register()

def taskFailures = Counter.build()
    .name("taskgraph_task_failures_total")
    .help("Total task failures")
    .labelNames("workflow_id", "task_id")
    .register()

// Export metrics via event listener
workflow.addListener(new PrometheusExporter() {
    @Override
    void onEvent(GraphEvent event) {
        switch (event.type) {
            case GRAPH_COMPLETED:
            case GRAPH_FAILED:
                workflowDuration
                    .labels(event.graphId, event.status.toString())
                    .observe(event.durationMs / 1000.0)
                break

            case TASK_COMPLETED:

```

```
case TASK_FAILED:
    def task = event.taskEvent
    taskDuration
        .labels(event.graphId, task.taskName, task.status.toString())
        .observe(task.durationMs / 1000.0)

    if (event.type == TASK_FAILED) {
        taskFailures
            .labels(event.graphId, task.taskName)
            .inc()
    }
    break
}
}

// Expose metrics endpoint
server.get("/metrics") { req, res ->
    res.contentType("text/plain; version=0.0.4")
    res.send(io.prometheus.client.exporter.common.TextFormat.write004(
        CollectorRegistry.defaultRegistry.metricFamilySamples()
    ))
}
```

Chapter 125. Distributed Tracing

Integrate with distributed tracing systems.

125.1. OpenTelemetry Integration

```
import io.opentelemetry.api.trace.*

def workflow = TaskGraph.build {
    // Enable tracing
    tracingConfig {
        enabled true
        serviceName "taskgraph-workflow"
        tracerProvider openTelemetryTracer
    }

    task("fetch-data") {
        action { ctx ->
            // Automatically creates span
            fetchData()
        }
    }

    task("process-data") {
        dependsOn "fetch-data"

        action { ctx ->
            // Span includes dependency info
            processData(ctx.prev)
        }
    }
}

// Traces show:
// Workflow Execution Trace
//   └─ fetch-data (50ms)
//     └─ process-data (100ms)
//       └─ dependency on fetch-data
```

125.2. Jaeger Integration

```
// Configure Jaeger tracer
def jaegerTracer = Configuration.fromEnv("taskgraph-workflow")
    .getTracer()

def workflow = TaskGraph.build {
    tracingConfig {
```

```
    tracer jaegerTracer
    sampleRate 1.0 // Sample all traces
}

// Workflow execution automatically traced
task("api-call") {
    action { callAPI() }
}
}

// View traces in Jaeger UI
// http://localhost:16686
```

125.3. Custom Span Attributes

```
task("process") {
    action { ctx ->
        // Add custom attributes to span
        ctx.currentSpan().setAttribute("user.id", ctx.global("userId"))
        ctx.currentSpan().setAttribute("batch.size", data.size())

        processData(data)
    }
}
```

Chapter 126. Logging Integration

Integrate with logging frameworks.

126.1. Structured Logging

```
import org.slf4j.LoggerFactory
import net.logstash.logback.argument.StructuredArguments.*

def logger = LoggerFactory.getLogger("TaskGraph")

workflow.addListener { event ->
    logger.info("workflow.event",
        keyValue("event_type", event.type),
        keyValue("workflow_id", event.graphId),
        keyValue("timestamp", event.timestamp),
        keyValue("duration_ms", event.durationMs)
    )
}
```

126.2. MDC (Mapped Diagnostic Context)

```
import org.slf4j.MDC

workflow.addListener { event ->
    // Set MDC for all log messages
    MDC.put("workflow_id", event.graphId)
    MDC.put("task_id", event.taskEvent?.taskName)

    try {
        processEvent(event)
    } finally {
        MDC.clear()
    }
}

// All logs now include workflow_id and task_id
task("process") {
    action {
        log.info("Processing data") // Includes workflow_id in log
    }
}
```

126.3. ELK Stack Integration

```
// Ship logs to Elasticsearch
def esClient = new RestHighLevelClient(/*...*/)

workflow.addListener { event ->
    def doc = [
        "@timestamp": event.timestamp,
        event_type: event.type,
        workflow_id: event.graphId,
        duration_ms: event.durationMs,
        status: event.status
    ]

    if (event.taskEvent) {
        doc.task_id = event.taskEvent.taskName
        doc.task_status = event.taskEvent.status
    }

    // Index in Elasticsearch
    esClient.index(
        new IndexRequest("taskgraph-events")
            .source(doc, XContentType.JSON)
    )
}

// Query with Kibana
// event_type:TASK_FAILED AND workflow_id:"daily-etl"
```

Chapter 127. Monitoring Dashboards

Create monitoring dashboards for workflows.

127.1. Grafana Dashboard

```
{
  "dashboard": {
    "title": "TaskGraph Monitoring",
    "panels": [
      {
        "title": "Workflow Execution Rate",
        "targets": [
          {
            "expr": "rate(taskgraph_workflow_duration_seconds_count[5m])"
          }
        ],
        "title": "Average Workflow Duration",
        "targets": [
          {
            "expr": "avg(taskgraph_workflow_duration_seconds_sum / taskgraph_workflow_duration_seconds_count)"
          }
        ],
        "title": "Task Failure Rate",
        "targets": [
          {
            "expr": "rate(taskgraph_task_failures_total[5m])"
          }
        ],
        "title": "Active Workflows",
        "targets": [
          {
            "expr": "taskgraph_active_workflows"
          }
        ]
      }
    ]
  }
}
```

127.2. Custom Dashboard

```
// Build custom dashboard
class WorkflowDashboard {
  def workflow

  Map<String, Object> getStatus() {
    def snapshot = workflow.snapshot()
```

```

def health = workflow.health()
def metrics = workflow.getMetrics()

return [
    status: snapshot.status,
    health: health.status,
    progress: snapshot.progressPercentage,
    tasks: [
        total: snapshot.totalTaskCount,
        completed: snapshot.completedTaskCount,
        running: snapshot.runningTaskCount,
        failed: snapshot.failedTaskCount
    ],
    performance: [
        duration: metrics.totalDurationMs,
        avgTaskDuration: metrics.avgTaskDurationMs,
        successRate: metrics.successRate
    ],
    runningTasks: snapshot.runningTasks.collect { task ->
        [
            id: task.id,
            duration: task.elapsedTimeMs,
            status: task.status
        ]
    }
]
}

// Serve dashboard
def dashboard = new WorkflowDashboard(workflow)

server.get("/dashboard") { req, res ->
    res.json(dashboard.getStatus())
}

// Auto-refresh dashboard
server.get("/dashboard/sse") { req, res ->
    res.contentType("text/event-stream")

    while (workflow.isRunning()) {
        def status = dashboard.getStatus()
        res.write("data: ${JsonOutput.toJson(status)}\n\n")
        res.flush()
        Thread.sleep(1000)
    }
}

```

Chapter 128. Alerting

Configure alerts for critical events.

128.1. Alert Rules

```
// Define alert rules
workflow.alerting {
    rule("high-failure-rate") {
        condition { metrics ->
            metrics.errorRate > 0.1 // > 10% error rate
        }
        severity CRITICAL
        notify(["ops@example.com", "slack:alerts"])
        message { "Workflow failure rate: ${it.errorRate * 100}%" }
    }

    rule("long-running-workflow") {
        condition { snapshot ->
            snapshot.elapsedTime.toMinutes() > 60 // > 1 hour
        }
        severity WARNING
        notify(["ops@example.com"])
        message { "Workflow running for ${it.elapsedTime.toMinutes()} minutes" }
    }

    rule("circuit-breaker-open") {
        condition { event ->
            event instanceof CircuitBreakerEvent &&
            event.state == OPEN
        }
        severity ERROR
        notify(["ops@example.com", "pagerduty:oncall"])
        message { "Circuit breaker open for task: ${it.taskName}" }
    }
}
```

128.2. Alert Channels

```
// Configure alert channels
AlertingConfig.configure {
    // Email
    email {
        smtp {
            host "smtp.example.com"
            port 587
            username "alerts@example.com"
```

```
        password credential("smtp-password")
    }
}

// Slack
slack {
    webhookUrl credential("slack-webhook-url")
    channel "#alerts"
    username "TaskGraph Bot"
}

// PagerDuty
pagerduty {
    apiKey credential("pagerduty-api-key")
    serviceKey "taskgraph-service"
}

// Custom webhook
webhook {
    url "https://monitoring.example.com/webhook"
    headers {
        "Authorization": "Bearer ${credential('webhook-token')}"
    }
}
}
```

Chapter 129. Summary

TaskGraph provides comprehensive observability:

- **Event system** - Real-time workflow events
- **Execution snapshots** - Point-in-time state capture
- **Health monitoring** - Workflow and task health checks
- **Metrics collection** - Performance and resource metrics
- **Distributed tracing** - OpenTelemetry/Jaeger integration
- **Logging integration** - Structured logging, ELK stack
- **Monitoring dashboards** - Prometheus, Grafana
- **Alerting** - Configurable alert rules and channels

With these observability features, you have complete visibility into your workflow execution.

Chapter 130. Next Steps

- **Chapter 8** - Test observability features
- **Chapter 7** - Observe example workflows
- **Appendix** - Complete observability reference

Chapter 131. Part IV: Reference

TaskBase Reference

Complete reference for TaskBase, the abstract base class that provides cross-cutting concerns for all tasks. This chapter documents every feature, method, and configuration option available to task implementations.

Chapter 132. Overview

TaskBase is the foundation of all tasks in TaskGraph, providing:

- **Lifecycle management** - Start, pause, cancel, resume
- **Retry logic** - Configurable retry policies
- **Circuit breaker** - Fault tolerance
- **Timeout handling** - Execution time limits
- **Idempotency** - Prevent duplicate executions
- **Event emission** - Lifecycle events
- **Error handling** - Comprehensive error management
- **Result management** - Async result handling

Chapter 133. Class Hierarchy

```
abstract class TaskBase<T> implements Task<T> {
    // Core identity
    String id
    String name
    String description

    // Execution context
    TaskContext context

    // Dependencies
    List<TaskBase> predecessors = []
    List<TaskBase> successors = []

    // State
    TaskState state = NOT_STARTED
    DataflowVariable<T> result = new DataflowVariable<T>()

    // Resilience
    RetryPolicy retryPolicy
    CircuitBreaker circuitBreaker
    TimeoutPolicy timeoutPolicy
    IdempotencyPolicy idempotencyPolicy

    // Concurrency
    ExecutorPool executorPool
    Semaphore resourceSemaphore

    // Metrics
    long startTime
    long endTime
    int attemptCount
    List<Throwable> errors = []

    // Template method - subclasses implement
    protected abstract T performTask(Object previousValue)
}
```

Chapter 134. Core Properties

134.1. Identity

```
// Unique identifier (required)
task.id = "my-task"

// Human-readable name
task.name = "Process User Data"

// Optional description
task.description = "Fetches and processes user data from API"

// Tags for grouping/filtering
task.tags = ["etl", "users", "production"]

// Custom metadata
task.metadata = [
    owner: "data-team",
    sla: "5min",
    criticality: "high"
]
```

134.2. State Management

```
// Task states
enum TaskState {
    NOT_STARTED,          // Initial state
    SCHEDULED,            // Scheduled for execution
    WAITING_DEPENDENCIES, // Waiting for predecessors
    READY,                // Ready to execute
    RUNNING,               // Currently executing
    COMPLETED,             // Successfully completed
    FAILED,                // Failed
    SKIPPED,               // Skipped (condition false)
    CANCELLED,              // Cancelled
    RETRYING,               // Retrying after failure
    PAUSED                 // Paused
}

// Check state
if (task.state == RUNNING) {
    println "Task is running"
}

// State transitions emit events
task.on(TaskStateChanged.class) { event ->
```

```
    println "${event.taskId} changed from ${event.oldState} to ${event.newState}"  
}
```

Chapter 135. Lifecycle Methods

135.1. Starting Execution

```
// Start execution (returns Promise)
Promise<T> start()

// Example
def promise = task.start()

promise.then { result ->
    println "Task completed: $result"
}.onError { error ->
    println "Task failed: $error"
}

// Or block for result
def result = task.start().get()
```

135.2. Pausing and Resuming

```
// Pause execution (if pausable)
task.pause()

// Resume execution
task.resume()

// Check if paused
if (task.isPaused()) {
    task.resume()
}

// Configure pausability
task.pausable = true // Default: false
```

135.3. Cancellation

```
// Cancel execution
task.cancel()

// Check if cancelled
if (task.isCancelled()) {
    println "Task was cancelled"
}
```

```
// Handle cancellation
task.onCancelled {
    cleanup()
    releaseResources()
}
```

Chapter 136. Retry Policy

Configure automatic retry on failure.

136.1. Basic Retry Configuration

```
task("retry-task") {
    retryPolicy {
        maxAttempts 3           // Max retry attempts
        delay 1000              // Initial delay (ms)
        exponentialBackoff true // Exponential backoff
        backoffMultiplier 2.0   // Backoff multiplier
        maxDelay 60000           // Max delay (ms)
    }
}
```

136.2. Selective Retry

```
task("selective-retry") {
    retryPolicy {
        maxAttempts 3

        // Retry on specific exceptions
        retryOn(IOException, TimeoutException)

        // Don't retry on specific exceptions
        abortOn(IllegalArgumentException, AuthenticationException)

        // Custom retry condition
        retryIf { error, attempt ->
            error instanceof HttpException &&
            error.statusCode == 503 && // Service unavailable
            attempt < 5
        }
    }
}
```

136.3. Retry Listeners

```
task("monitored-retry") {
    retryPolicy {
        maxAttempts 3

        onRetry { attempt, error ->
            log.warn("Retry attempt ${attempt} after error: ${error.message}")
        }
    }
}
```

```
    metricsCollector.incrementRetryCount(task.id)
}

onExhausted { error ->
    log.error("All retry attempts exhausted", error)
    alertOps(task.id, error)
}
}

}
```

Chapter 137. Circuit Breaker

Implement fault tolerance patterns.

137.1. Basic Circuit Breaker

```
task("protected-task") {
    circuitBreaker {
        failureThreshold 5          // Open after 5 failures
        successThreshold 2          // Close after 2 successes
        timeout 3000               // Consider failure after 30s
        halfOpenAfter 60000         // Try again after 60s
        resetTimeout 300000         // Full reset after 5min
    }
}
```

137.2. Circuit Breaker States

```
// Circuit breaker states
enum CircuitState {
    CLOSED,           // Normal operation
    OPEN,            // Blocking calls
    HALF_OPEN        // Testing recovery
}

// Check circuit state
def state = task.circuitBreaker.getState()

if (state == OPEN) {
    println "Circuit breaker is open - calls blocked"
}

// Force open/close
task.circuitBreaker.forceOpen()
task.circuitBreaker.forceClose()
task.circuitBreaker.reset()

// Listen to state changes
task.circuitBreaker.onStateChange { oldState, newState ->
    log.info("Circuit breaker ${oldState} -> ${newState}")
    metricsCollector.recordCircuitState(task.id, newState)
}
```

137.3. Fallback on Circuit Open

```
task("circuit-with-fallback") {
    circuitBreaker {
        failureThreshold 3
        timeout 30000

        onOpen {
            log.warn("Circuit opened for ${task.id}")
        }

        fallback { error ->
            // Return cached or default value
            log.info("Using fallback value")
            return cachedValue ?: defaultValue
        }
    }
}
```

Chapter 138. Timeout Policy

Control execution time limits.

138.1. Basic Timeout

```
task("time-limited") {
    timeout 30000 // 30 seconds

    action {
        // Must complete within 30 seconds
        longRunningOperation()
    }
}
```

138.2. Advanced Timeout Configuration

```
task("timeout-config") {
    timeoutPolicy {
        timeout 3000          // Timeout duration
        hardTimeout false      // Don't kill thread (default)
        interruptOnTimeout true // Interrupt thread
        timeoutAction FAIL      // Action: FAIL, RETRY, FALLBACK

        onTimeout { duration ->
            log.warn("Task timed out after ${duration}ms")
            metricsCollector.recordTimeout(task.id)
        }

        fallback {
            // Return fallback value on timeout
            return fallbackValue
        }
    }
}
```

Chapter 139. Idempotency Policy

Prevent duplicate executions.

139.1. Basic Idempotency

```
task("idempotent-task") {
    idempotencyPolicy {
        enabled true
        ttl 3600 // Cache for 1 hour

        // Generate idempotency key
        keyExtractor { ctx ->
            "task:${task.id}:user:${ctx.global('userId')}"
        }
    }

    action { ctx ->
        // Only executes once per key within TTL
        expensiveOperation()
    }
}
```

139.2. Custom Idempotency Store

```
task("redis-idempotent") {
    idempotencyPolicy {
        enabled true
        ttl 3600

        // Custom storage (e.g., Redis)
        store new RedisIdempotencyStore(redisClient)

        keyExtractor { ctx ->
            "workflow:${ctx.global('workflowId')}:task:${task.id}"
        }

        onDuplicate { cachedResult ->
            log.info("Returning cached result for ${task.id}")
            metricsCollector.incrementIdempotentHits(task.id)
        }
    }
}
```

Chapter 140. Concurrency Control

Manage parallel execution.

140.1. Concurrency Limits

```
task("limited-concurrency") {
    concurrencyLimit 5 // Max 5 instances concurrently

    action {
        // Only 5 instances of this task can run at once
        performWork()
    }
}
```

140.2. Resource Semaphores

```
task("shared-resource") {
    resource "database-pool", max: 10

    action {
        // Acquires semaphore permit from "database-pool"
        // Max 10 permits available across all tasks
        queryDatabase()
    }
}

task("another-db-task") {
    resource "database-pool", max: 10

    action {
        // Shares same semaphore
        queryDatabase()
    }
}
```

140.3. Custom Executor Pool

```
// Use custom thread pool
def ioPool = ExecutorPoolFactory.newCachedThreadPool()

task("io-bound") {
    executorPool ioPool

    action {
```

```
// Executes in custom pool  
performIOOperation()  
}  
}
```

Chapter 141. Conditional Execution

Control whether task executes.

141.1. Simple Condition

```
task("conditional") {
    condition { ctx ->
        ctx.global("environment") == "production"
    }

    action {
        // Only runs in production
        sendToProductionAPI()
    }
}
```

141.2. Complex Conditions

```
task("complex-condition") {
    condition { ctx ->
        def enabled = ctx.config("feature.enabled")
        def userType = ctx.global("userType")
        def time = LocalTime.now()

        return enabled &&
            userType == "premium" &&
            time.hour >= 9 && time.hour < 17
    }

    onSkipped {
        log.info("Task skipped: condition false")
        metricsCollector.incrementSkippedTasks(task.id)
    }

    action {
        // Only executes if condition true
        performWork()
    }
}
```

Chapter 142. Event Handling

React to task lifecycle events.

142.1. Lifecycle Hooks

```
task("lifecycle-hooks") {
    // Before execution
    beforeExecute {
        log.info("Starting task ${task.id}")
        startTimer()
    }

    // After successful execution
    afterExecute { result ->
        log.info("Task completed with result: $result")
        stopTimer()
        recordMetric(result)
    }

    // On any error
    onError { error ->
        log.error("Task failed", error)
        alertOps(error)
    }

    // On specific error types
    onError(IOException) { error ->
        log.warn("IO error occurred", error)
    }

    onError(TimeoutException) { error ->
        log.warn("Task timed out", error)
    }

    // On retry
    onRetry { attempt, error ->
        log.warn("Retry attempt ${attempt}", error)
    }

    // On completion (success or failure)
    onComplete {
        cleanup()
        releaseResources()
    }

    // On cancellation
    onCancelled {
        log.info("Task cancelled")
    }
}
```

```
    rollback()
}

action {
    performWork()
}
}
```

142.2. Custom Events

```
task("event-emitter") {
    action { ctx ->
        // Emit custom events
        ctx.emit(new CustomEvent(
            type: "DATA_PROCESSED",
            taskId: task.id,
            data: processedData
        ))

        // Emit progress events
        (1..10).each { i ->
            ctx.emit(new ProgressEvent(
                taskId: task.id,
                progress: i * 10,
                message: "Processing batch ${i}/10"
            ))
        }

        processBatch(i)
    }

    return result
}
}
```

Chapter 143. Validation

Validate task configuration.

143.1. Built-in Validation

```
task("validated") {
    // Validate configuration
    validate {
        require(id != null, "Task ID required")
        require(name != null, "Task name required")
        require(predecessors.every { it != null }, "Invalid predecessor")

        if (retryPolicy) {
            require(retryPolicy.maxAttempts > 0, "Invalid max attempts")
            require(retryPolicy.delay >= 0, "Invalid delay")
        }

        if (circuitBreaker) {
            require(circuitBreaker.failureThreshold > 0, "Invalid failure threshold")
        }
    }
}
```

143.2. Custom Validation

```
class CustomTask extends TaskBase<Result> {
    String apiUrl
    String apiKey

    @Override
    void validate() {
        super.validate()

        if (!apiUrl) {
            throw new ValidationException("API URL required")
        }

        if (!apiUrl.startsWith("https://")) {
            throw new ValidationException("HTTPS required")
        }

        if (!apiKey) {
            throw new ValidationException("API key required")
        }
    }
}
```

```
@Override  
protected Result performTask(Object prev) {  
    // Execute task  
}  
}
```

Chapter 144. Error Handling

Comprehensive error management.

144.1. Error Recovery

```
task("error-recovery") {
    action {
        try {
            return riskyOperation()
        } catch (RecoverableException e) {
            log.warn("Recoverable error", e)
            return fallbackOperation()
        }
    }

    // Automatic recovery for specific errors
    recover(IOException) { error ->
        log.warn("IO error, using cache", error)
        return cachedValue
    }

    recover(TimeoutException) { error ->
        log.warn("Timeout, using default", error)
        return defaultValue
    }
}
```

144.2. Error Propagation

```
task("error-propagation") {
    // Wrap errors
    wrapErrors true // Wrap in TaskExecutionException

    // Transform errors
    errorTransformer { error ->
        if (error instanceof HttpException) {
            return new APIException("API call failed", error)
        }
        return error
    }

    // Suppress specific errors
    suppressErrors(NotFoundException)

    action {
        performWork()
    }
}
```

```
    }  
}
```

Chapter 145. Result Management

Handle task results.

145.1. Result Transformation

```
task("transform-result") {
    action {
        return fetchData() // Returns List<User>
    }

    // Transform result
    transformResult { users ->
        return users.collect { user ->
            [id: user.id, name: user.name]
        }
    }
}
```

145.2. Result Caching

```
task("cached-result") {
    resultCache {
        enabled true
        ttl 3600 // Cache for 1 hour
        key { ctx -> "users:${ctx.global('tenantId')}" }

        // Custom cache implementation
        cache new RedisCacheProvider(redisClient)

        onCacheHit { cachedResult ->
            log.info("Using cached result")
            metricsCollector.incrementCacheHits(task.id)
        }

        onCacheMiss {
            log.info("Cache miss, executing task")
            metricsCollector.incrementCacheMisses(task.id)
        }
    }
}
```

145.3. Result Cleanup

```
task("large-result") {
```

```
cleanupResults true // Clear after successors consume

action {
    return new byte[100 * 1024 * 1024] // 100 MB
}

task("consumer") {
    dependsOn "large-result"

    action { ctx ->
        def data = ctx.prev
        processData(data)

        // "large-result" cleared from memory here
    }
}
```

Chapter 146. Metrics and Monitoring

Built-in metrics collection.

146.1. Task Metrics

```
// Access task metrics
def metrics = task.getMetrics()

println "Execution count: ${metrics.executionCount}"
println "Failure count: ${metrics.failureCount}"
println "Success rate: ${metrics.successRate * 100}%"
println "Avg duration: ${metrics.avgDurationMs}ms"
println "Min duration: ${metrics.minDurationMs}ms"
println "Max duration: ${metrics.maxDurationMs}ms"
println "Total duration: ${metrics.totalDurationMs}ms"
println "Retry count: ${metrics.retryCount}"
println "Circuit breaker opens: ${metrics.circuitBreakerOpens}"
println "Timeout count: ${metrics.timeoutCount}"
```

146.2. Custom Metrics

```
task("custom-metrics") {
    action { ctx ->
        // Record custom metrics
        ctx.recordMetric("${task.id}.records.processed", data.size())
        ctx.recordMetric("${task.id}.processing.duration", duration)
        ctx.recordMetric("${task.id}.memory.used", memoryUsed)

        return result
    }
}

// Export to monitoring systems
task.metrics.export(prometheusRegistry)
```

Chapter 147. Best Practices

147.1. Use Retry for Transient Failures

```
// ☺ Good - retry transient failures
httpTask("api-call") {
    retryPolicy {
        maxAttempts 3
        retryOn(IOException, TimeoutException)
        exponentialBackoff true
    }
}

// ☹ Bad - retry permanent failures
task("bad-retry") {
    retryPolicy {
        maxAttempts 3
        // Will retry even for permanent errors like 404
    }
}
```

147.2. Use Circuit Breaker for External Services

```
// ☺ Good - protect external service calls
httpTask("external-api") {
    circuitBreaker {
        failureThreshold 5
        timeout 30000
    }
}
```

147.3. Set Appropriate Timeouts

```
// ☺ Good - reasonable timeout
task("api-call") {
    timeout 30000 // 30 seconds
}

// ☹ Bad - too long
task("api-call") {
    timeout 600000 // 10 minutes is too long for API call
}
```

147.4. Clean Up Large Results

```
// ☐ Good - clean up large data
task("large-data") {
    cleanupResults true
}

// ☐ Bad - keep large data in memory
task("large-data") {
    cleanupResults false // Keeps 100MB in memory
    action { new byte[100 * 1024 * 1024] }
}
```

Chapter 148. Summary

TaskBase provides comprehensive task infrastructure:

- **Lifecycle** - Start, pause, cancel, resume
- **Retry** - Automatic retry with backoff
- **Circuit breaker** - Fault tolerance
- **Timeout** - Execution time limits
- **Idempotency** - Prevent duplicates
- **Concurrency** - Control parallel execution
- **Events** - Lifecycle hooks and custom events
- **Validation** - Configuration validation
- **Error handling** - Recovery and propagation
- **Metrics** - Built-in monitoring

All these features are available to every task through inheritance from TaskBase.

Chapter 149. Next Steps

- **Chapter 13** - Concrete task implementations
- **Chapter 5** - Extend TaskBase for custom tasks
- **Chapter 8** - Test TaskBase features

Gateway Types Reference

Complete reference for all gateway types in TaskGraph. Gateways control workflow routing, enabling conditional logic, parallel execution, and complex orchestration patterns.

Chapter 150. Overview

Gateways are decision points in workflows that control execution paths. TaskGraph provides several gateway types:

- **Exclusive Gateway** - XOR logic, one path taken
- **Parallel Gateway** - AND logic, all paths taken
- **Inclusive Gateway** - OR logic, multiple paths conditionally
- **Event Gateway** - Event-based routing
- **Complex Gateway** - Custom routing logic

Chapter 151. Exclusive Gateway

Takes exactly one path based on conditions (XOR).

151.1. Basic Usage

```
def workflow = TaskGraph.build {
    task("check-value") {
        action { fetchValue() }
    }

    exclusiveGateway("route") {
        dependsOn "check-value"

        condition { ctx ->
            ctx.prev > 100
        }

        whenTrue {
            task("high-value") {
                action { processHighValue() }
            }
        }

        whenFalse {
            task("low-value") {
                action { processLowValue() }
            }
        }
    }
}
```

151.2. Multiple Conditions

```
exclusiveGateway("classify") {
    dependsOn "check-status"

    // Evaluated in order, first match taken
    when({ ctx -> ctx.prev == "urgent" }) {
        task("urgent-path") { action { handleUrgent() } }
    }

    when({ ctx -> ctx.prev == "high" }) {
        task("high-priority") { action { handleHigh() } }
    }

    when({ ctx -> ctx.prev == "normal" }) {
```

```

        task("normal-priority") { action { handleNormal() } }

    }

otherwise {
    // Default path if no conditions match
    task("low-priority") { action { handleLow() } }
}
}

```

151.3. Switch-Style Gateway

```

exclusiveGateway("switch") {
    dependsOn "get-type"

    switchOn { ctx -> ctx.prev }

    case("type-a") {
        task("process-a") { action { processTypeA() } }
    }

    case("type-b") {
        task("process-b") { action { processTypeB() } }
    }

    case("type-c") {
        task("process-c") { action { processTypeC() } }
    }

    default {
        task("unknown-type") { action { handleUnknown() } }
    }
}

```

151.4. Path Validation

```

exclusiveGateway("validated") {
    // Ensure exactly one path taken
    strictMode true // Throws exception if no path matches

    condition { ctx -> ctx.prev > 0 }
    whenTrue { task("positive") { action { processPositive() } } }
    whenFalse { task("negative") { action { processNegative() } } }
}

```

Chapter 152. Parallel Gateway

Executes all paths in parallel (AND).

152.1. Fork and Join

```
def workflow = TaskGraph.build {
    task("start") {
        action { fetchData() }
    }

    parallelGateway("split") {
        dependsOn "start"

        // Fork: All paths execute in parallel
        fork {
            task("path-a") { action { processA() } }
            task("path-b") { action { processB() } }
            task("path-c") { action { processC() } }
        }

        // Join: Wait for all paths to complete
        join {
            task("merge") {
                dependsOn "path-a", "path-b", "path-c"
                action { ctx ->
                    def results = [
                        ctx.taskResult("path-a"),
                        ctx.taskResult("path-b"),
                        ctx.taskResult("path-c")
                    ]
                    return mergeResults(results)
                }
            }
        }
    }
}
```

152.2. Dynamic Parallel Paths

```
parallelGateway("dynamic-split") {
    dependsOn "get-regions"

    fork { ctx ->
        def regions = ctx.prev

        // Create task for each region
    }
}
```

```

regions.each { region ->
    task("process-${region}") {
        action { processRegion(region) }
    }
}
}

join { ctx ->
    def regions = ctx.taskResult("get-regions")

    task("aggregate") {
        dependsOn regions.collect { "process-${it}" }

        action { ctx ->
            def results = regions.collect { region ->
                ctx.taskResult("process-${region}")
            }
            return aggregateResults(results)
        }
    }
}
}

```

152.3. Concurrent Execution Control

```

parallelGateway("controlled-parallel") {
    // Limit concurrent paths
    maxConcurrency 5

    fork {
        // 10 paths, but only 5 run concurrently
        (1..10).each { i ->
            task("path-${i}") {
                action { processPath(i) }
            }
        }
    }

    join {
        task("join") {
            dependsOn (1..10).collect { "path-${it}" }
            action { ctx -> aggregateAll() }
        }
    }
}

```

152.4. Timeout for Join

```
parallelGateway("timeout-join") {
    fork {
        task("fast") { action { Thread.sleep(1000); "fast" } }
        task("slow") { action { Thread.sleep(60000); "slow" } }
    }

    join {
        // Wait max 5 seconds for all paths
        timeout 5000

        onTimeout { incompleteTasks ->
            log.warn("Timeout waiting for: ${incompleteTasks.join(', ')}")
            // Continue with completed tasks only
        }

        task("merge") {
            action { ctx -> mergeCompleted() }
        }
    }
}
```

Chapter 153. Inclusive Gateway

Executes multiple paths conditionally (OR).

153.1. Basic Inclusive Gateway

```
inclusiveGateway("multi-path") {
    dependsOn "check-flags"

    // Multiple paths can execute
    when({ ctx -> ctx.global("sendEmail") }) {
        task("send-email") { action { sendEmail() } }
    }

    when({ ctx -> ctx.global("sendSMS") }) {
        task("send-sms") { action { sendSMS() } }
    }

    when({ ctx -> ctx.global("sendPush") }) {
        task("send-push") { action { sendPush() } }
    }

    // All matching paths execute in parallel
    join {
        task("confirm") {
            dependsOn "send-email", "send-sms", "send-push"
            action { confirmNotifications() }
        }
    }
}
```

153.2. Minimum Paths

```
inclusiveGateway("at-least-one") {
    minPaths 1 // At least one path must be taken

    when({ ctx -> condition1(ctx) }) {
        task("option-1") { action { processOption1() } }
    }

    when({ ctx -> condition2(ctx) }) {
        task("option-2") { action { processOption2() } }
    }

    when({ ctx -> condition3(ctx) }) {
        task("option-3") { action { processOption3() } }
    }
}
```

```
// If no conditions match, throws exception
}
```

153.3. Maximum Paths

```
inclusiveGateway("max-two") {
    maxPaths 2 // Max two paths can be taken

    when({ ctx -> condition1(ctx) }) {
        task("path-1") { action { process1() } }
    }

    when({ ctx -> condition2(ctx) }) {
        task("path-2") { action { process2() } }
    }

    when({ ctx -> condition3(ctx) }) {
        task("path-3") { action { process3() } }
    }

    // Evaluates in order, stops after 2 matches
}
```

Chapter 154. Event Gateway

Routes based on events (event-driven routing).

154.1. Basic Event Gateway

```
eventGateway("wait-for-event") {  
    dependsOn "initiate"  
  
    // Wait for one of these events  
    on("order.completed") {  
        task("process-completion") {  
            action { processCompletion() }  
        }  
    }  
  
    on("order.cancelled") {  
        task("process-cancellation") {  
            action { processCancellation() }  
        }  
    }  
  
    on("order.timeout") {  
        task("handle-timeout") {  
            action { handleTimeout() }  
        }  
    }  
  
    // First event to arrive determines path  
    timeout 60000 // Max wait time  
}
```

154.2. Message-Based Routing

```
eventGateway("message-router") {  
    dependsOn "send-request"  
  
    // Route based on message type  
    onMessage("response") { message ->  
        task("handle-response") {  
            action { ctx ->  
                processResponse(message)  
            }  
        }  
    }  
  
    onMessage("error") { message ->
```

```

        task("handle-error") {
            action { ctx ->
                handleError(message)
            }
        }
    }

onMessage("timeout") { message ->
    task("handle-timeout") {
        action { ctx ->
            handleTimeout(message)
        }
    }
}

defaultHandler {
    task("unknown-message") {
        action { handleUnknown() }
    }
}
}

```

154.3. Timer-Based Events

```

eventGateway("timer-events") {
    // Wait for event or timer
    onTimer(5000) {
        task("timeout-path") {
            action { handleTimeout() }
        }
    }

    on("data.ready") {
        task("data-path") {
            action { processData() }
        }
    }

    // Whichever happens first
}

```

Chapter 155. Complex Gateway

Custom routing logic for complex scenarios.

155.1. Custom Routing

```
complexGateway("custom-logic") {
    dependsOn "evaluate"

    // Custom routing logic
    route { ctx ->
        def value = ctx.prev
        def paths = []

        if (value > 100) {
            paths << "high-value-processing"
        }

        if (value % 2 == 0) {
            paths << "even-number-handling"
        }

        if (isPrime(value)) {
            paths << "prime-number-analysis"
        }

        return paths
    }

    // Define available paths
    path("high-value-processing") {
        task("high-value") { action { processHigh() } }
    }

    path("even-number-handling") {
        task("even") { action { processEven() } }
    }

    path("prime-number-analysis") {
        task("prime") { action { analyzePrime() } }
    }

    // Multiple paths may be selected
}
```

155.2. State Machine Gateway

```
complexGateway("state-machine") {
    dependsOn "get-state"

    stateMachine {
        state("PENDING") {
            on("approve") { transition("APPROVED") }
            on("reject") { transition("REJECTED") }
        }

        state("APPROVED") {
            on("ship") { transition("SHIPPED") }
            on("cancel") { transition("CANCELLED") }
        }

        state("SHIPPED") {
            on("deliver") { transition("DELIVERED") }
            on("return") { transition("RETURNED") }
        }

        state("DELIVERED") {
            // Final state
        }
    }

    route { ctx ->
        def currentState = ctx.prev.state
        def event = ctx.global("event")

        return getTransition(currentState, event)
    }
}
```

155.3. Weighted Routing

```
complexGateway("weighted-route") {
    // Weighted random selection (e.g., A/B testing)
    weightedRoute {
        path("variant-a", weight: 70) { // 70% traffic
            task("a") { action { processVariantA() } }
        }

        path("variant-b", weight: 20) { // 20% traffic
            task("b") { action { processVariantB() } }
        }

        path("variant-c", weight: 10) { // 10% traffic
    }}
```

```
    task("c") { action { processVariantC() } }
```

```
}
```

```
}
```

```
route { ctx ->
```

```
    def random = new Random().nextInt(100)
```

```
    if (random < 70) return "variant-a"
```

```
    if (random < 90) return "variant-b"
```

```
    return "variant-c"
```

```
}
```

```
}
```

Chapter 156. Gateway Patterns

156.1. Discriminator Pattern

First path to complete wins, others cancelled.

```
discriminatorGateway("race") {  
    dependsOn "start"  
  
    fork {  
        task("source-a") { action { fetchFromA() } }  
        task("source-b") { action { fetchFromB() } }  
        task("source-c") { action { fetchFromC() } }  
    }  
  
    // First to complete wins  
    discriminate FIRST_COMPLETE  
  
    onWinner { winningTask ->  
        log.info("Winner: ${winningTask.id}")  
  
        // Cancel other tasks  
        cancelOthers()  
    }  
  
    join {  
        task("continue") {  
            action { ctx ->  
                // Use result from winning task  
                processWinner(ctx.prev)  
            }  
        }  
    }  
}
```

156.2. N-out-of-M Join

Continue when N out of M paths complete.

```
parallelGateway("n-out-of-m") {  
    fork {  
        task("task-1") { action { fetch1() } }  
        task("task-2") { action { fetch2() } }  
        task("task-3") { action { fetch3() } }  
        task("task-4") { action { fetch4() } }  
        task("task-5") { action { fetch5() } }  
    }
```

```

join {
    // Continue when 3 out of 5 complete
    threshold 3

    onThresholdMet { completedTasks ->
        log.info("3 tasks completed: ${completedTasks.join(', ')}")

        // Cancel remaining tasks
        cancelRemaining()
    }

    task("continue") {
        action { ctx ->
            // Process results from completed tasks
            processCompleted(completedTasks)
        }
    }
}

```

156.3. Synchronizing Merge

Wait for all instances of a repeating task.

```

synchronizingMergeGateway("sync") {
    // Wait for all parallel instances
    waitFor {
        (1..10).collect { "process-batch-${it}" }
    }

    // All must complete before continuing
    merge {
        task("aggregate") {
            action { ctx ->
                def results = (1..10).collect { i ->
                    ctx.taskResult("process-batch-${i}")
                }
                return aggregateResults(results)
            }
        }
    }
}

```

Chapter 157. Gateway Error Handling

157.1. Error in Path

```
exclusiveGateway("error-handled") {
    condition { ctx -> ctx.prev > 0 }

    whenTrue {
        task("positive") {
            action { processPositive() }

            onError { error ->
                // Handle error in this path
                log.error("Positive path failed", error)
                ctx.emit(new PathErrorEvent("positive", error))
            }
        }
    }

    whenFalse {
        task("negative") {
            action { processNegative() }
        }
    }

    // Gateway continues even if one path fails
    continueOnError true
}
```

157.2. Gateway Timeout

```
parallelGateway("timeout-handling") {
    fork {
        task("fast") { action { fastOperation() } }
        task("slow") { action { slowOperation() } }
    }

    join {
        timeout 10000 // 10 seconds

        onTimeout { incompleteTasks ->
            log.warn("Timeout: ${incompleteTasks.join(', ')}")

            // Handle timeout
            incompleteTasks.each { task ->
                task.cancel()
            }
        }
}
```

```
// Continue with completed tasks
return CONTINUE
}

task("merge") {
    action { mergeAvailable() }
}
}
```

Chapter 158. Gateway Observability

158.1. Gateway Events

```
exclusiveGateway("monitored") {
    onEvaluate { condition, result ->
        log.info("Condition evaluated: ${result}")
        metricsCollector.recordGatewayDecision(gateway.id, result)
    }

    onPathSelected { path ->
        log.info("Path selected: ${path.id}")
        metricsCollector.recordPathSelection(gateway.id, path.id)
    }

    condition { ctx -> ctx.prev > 100 }
    whenTrue { task("high") { action { processHigh() } } }
    whenFalse { task("low") { action { processLow() } } }
}
```

158.2. Gateway Metrics

```
def gateway = workflow.getGateway("my-gateway")

// Access gateway metrics
def metrics = gateway.getMetrics()

println "Evaluations: ${metrics.evaluationCount}"
println "Path selections: ${metrics.pathSelections}"
println "Avg decision time: ${metrics.avgDecisionTimeMs}ms"

metrics.pathSelections.each { path, count ->
    println " ${path}: ${count} times (${count/metrics.evaluationCount * 100}%)"
}
```

Chapter 159. Best Practices

159.1. Use Exclusive for Mutually Exclusive Paths

```
// ☐ Good - use exclusive gateway
exclusiveGateway("xor") {
    condition { ctx -> ctx.prev > 0 }
    whenTrue { task("positive") { action { processPositive() } } }
    whenFalse { task("negative") { action { processNegative() } } }
}

// ☐ Bad - don't use inclusive for XOR logic
inclusiveGateway("bad-xor") {
    // Only one should execute, but inclusive allows both
}
```

159.2. Always Provide Default Path

```
// ☐ Good - has default path
exclusiveGateway("safe") {
    when({ ctx -> condition1(ctx) }) { task("path-1") { action { process1() } } }
    when({ ctx -> condition2(ctx) }) { task("path-2") { action { process2() } } }
    otherwise { task("default") { action { processDefault() } } }
}

// ☐ Bad - no default, could fail
exclusiveGateway("unsafe") {
    when({ ctx -> condition1(ctx) }) { task("path-1") { action { process1() } } }
    // What if condition1 is false?
}
```

159.3. Use Parallel for Independent Paths

```
// ☐ Good - parallel for independent work
parallelGateway("parallel") {
    fork {
        task("fetch-users") { action { fetchUsers() } }
        task("fetch-orders") { action { fetchOrders() } }
    }
}

// ☐ Bad - sequential for independent work
task("fetch-users") { action { fetchUsers() } }
task("fetch-orders") {
    dependsOn "fetch-users" // Unnecessary dependency!
```

```
    action { fetchOrders() }  
}
```

159.4. Set Timeouts for Joins

```
// ☺ Good - has timeout  
parallelGateway("safe-join") {  
    fork { /* ... */ }  
    join {  
        timeout 30000  
        task("merge") { action { merge() } }  
    }  
}  
  
// ☹ Bad - no timeout, could wait forever  
parallelGateway("unsafe-join") {  
    fork { /* ... */ }  
    join {  
        // No timeout!  
        task("merge") { action { merge() } }  
    }  
}
```

Chapter 160. Summary

TaskGraph provides comprehensive gateway types:

- **Exclusive** - XOR logic, one path taken
- **Parallel** - AND logic, all paths taken concurrently
- **Inclusive** - OR logic, multiple conditional paths
- **Event** - Event-driven routing
- **Complex** - Custom routing logic
- **Patterns** - Discriminator, N-out-of-M, synchronizing
- **Error handling** - Path errors and timeouts
- **Observability** - Events and metrics

Gateways enable sophisticated workflow routing and orchestration patterns.

Chapter 161. Next Steps

- **Chapter 7** - Gateway usage examples
- **Chapter 4** - Gateways in workflow architecture
- **Chapter 13** - Combine gateways with task types

Concrete Task Types Reference

Complete reference for all built-in task types provided by TaskGraph. This chapter documents every concrete task implementation, its DSL features, configuration options, and usage examples.

Chapter 162. Overview

TaskGraph provides 15+ concrete task types organized by category:

- **Core Tasks** - ServiceTask, ScriptTask
- **HTTP/REST** - HttpTask, RestTask, GraphQLTask
- **Database** - SqlTask, NoSqlTask, BatchSqlTask
- **File Operations** - FileTask, CsvTask, JsonTask, XmlTask
- **Messaging** - MessagingTask, KafkaTask, RabbitMQTask
- **Cloud Services** - S3Task, BlobStorageTask, LambdaTask
- **Utilities** - DelayTask, LogTask, TransformTask

Chapter 163. Core Tasks

163.1. ServiceTask

Generic task for executing arbitrary code.

```
task("service-task") {
    action { ctx ->
        // Any custom logic
        def input = ctx.prev
        def result = processData(input)
        return result
    }

    // With method reference
    action this::myMethod

    // With closure delegate
    action { ctx ->
        def userId = ctx.global("userId")
        return userService.getUser(userId)
    }
}

// Configuration
task("configured-service") {
    description "Process user data"
    timeout 30000
    retryPolicy {
        maxAttempts 3
        delay 1000
    }

    action { ctx ->
        performWork()
    }
}
```

163.2. ScriptTask

Execute Groovy scripts with sandboxing.

```
scriptTask("groovy-script") {
    language "groovy" // Default
    sandboxed true // Default

    script '''
```

```

def data = bindings.data
def result = data.collect { it * 2 }
return result
"""

bindings([
    data: [1, 2, 3, 4, 5]
])

// Dynamic bindings
bindings { ctx ->
    [
        userId: ctx.global("userId"),
        data: ctx.prev
    ]
}
}

// Custom script base class
scriptTask("custom-base") {
    customScriptBaseClass CompanyScriptBase

    script """
        // Company-specific methods available
        def data = queryDataWarehouse("SELECT * FROM sales")
        return processData(data)
    """
}

// JavaScript (via GraalVM)
scriptTask("javascript") {
    language "javascript"

    script """
        const data = bindings.data;
        const result = data.map(x => x * 2);
        return result;
    """
}

```

Chapter 164. HTTP/REST Tasks

164.1. HttpTask

Make HTTP requests.

```
httpTask("api-call") {
    url "https://api.example.com/users"
    method GET

    // Headers
    header "Authorization", "Bearer ${credential('api-token')}"
    header "Content-Type", "application/json"
    header "User-Agent", "TaskGraph/2.0"

    headers {
        "X-Request-ID": UUID.randomUUID().toString()
        "X-Correlation-ID": ctx.global("correlationId")
    }

    // Query parameters
    queryParams "page", "1"
    queryParams "limit", "100"

    queryParams([
        page: 1,
        limit: 100,
        sort: "name"
    ])

    // Timeout
    timeout 30000
    connectTimeout 5000
    readTimeout 10000

    // Retry
    retryPolicy {
        maxAttempts 3
        retryOn(IOException, SocketTimeoutException)
    }

    // SSL
    validateSSL true
    trustStore "/path/to/truststore.jks"

    // Response handling
    responseHandler { response ->
        if (response.statusCode == 200) {
            return response.body
        }
    }
}
```

```

        } else {
            throw new ApiException("API returned ${response.statusCode}")
        }
    }
}

// POST with JSON body
httpTask("post-json") {
    url "https://api.example.com/users"
    method POST

    json {
        name: "Alice"
        email: "alice@example.com"
        age: 30
    }

    // Or from variable
    json { ctx -> ctx.prev }
}

// Form data
httpTask("form-post") {
    url "https://api.example.com/submit"
    method POST
    contentType "application/x-www-form-urlencoded"

    formData {
        username: "alice"
        password: credential("user-password")
    }
}

// Multipart upload
httpTask("file-upload") {
    url "https://api.example.com/upload"
    method POST
    contentType "multipart/form-data"

    multipart {
        file "document", new File("/path/to/file.pdf")
        field "description", "Important document"
    }
}

```

164.2. RestTask

Higher-level REST API client.

```

restTask("rest-api") {
    baseUrl "https://api.example.com"
    version "v1" // Added to path: /v1/

    // Authentication
    auth {
        bearer credential("api-token")
        // or: basic username: "user", password: credential("password")
        // or: oauth2 clientId: "...", clientSecret: credential("...")
    }

    // GET request
    get("/users/${userId}")

    // POST with auto-serialization
    post("/users") {
        body([
            name: "Alice",
            email: "alice@example.com"
        ])
    }

    // PUT
    put("/users/${userId}") {
        body(updatedUser)
    }

    // DELETE
    delete("/users/${userId}")

    // Response deserialization
    responseType User.class

    // Error handling
    errorHandler {
        on(404) { response ->
            log.warn("User not found")
            return null
        }

        on(500..599) { response ->
            throw new ServerException("Server error: ${response.body}")
        }
    }
}

```

164.3. GraphQLTask

Execute GraphQL queries.

```

graphqlTask("query-users") {
    endpoint "https://api.example.com/graphql"

    query """
        query GetUsers($limit: Int!) {
            users(limit: $limit) {
                id
                name
                email
                orders {
                    id
                    total
                }
            }
        }
    """

    variables {
        limit: 10
    }

    // Authentication
    header "Authorization", "Bearer ${credential('graphql-token')}"
}

// Response extraction
extractPath "data.users"

// Error handling
onGraphQLError { errors ->
    errors.each { error ->
        log.error("GraphQL error: ${error.message}")
    }
    throw new GraphQLEException("Query failed")
}

// Mutation
graphqlTask("create-user") {
    endpoint "https://api.example.com/graphql"

    mutation """
        mutation CreateUser($input: CreateUserInput!) {
            createUser(input: $input) {
                id
                name
                email
            }
        }
    """
}

```

```
variables { ctx ->
    def user = ctx.prev
    [input: [
        name: user.name,
        email: user.email
    ]]
}
```

Chapter 165. Database Tasks

165.1. SqlTask

Execute SQL queries.

```
sqlTask("query-users") {
    datasource myDataSource

    query """
        SELECT id, name, email, created
        FROM users
        WHERE status = ?
        AND created > ?
        ORDER BY created DESC
        LIMIT ?
    """

    parameters(["active", yesterday, 100])

    // Dynamic parameters
    parameters { ctx ->
        [
            ctx.global("status"),
            ctx.global("since"),
            ctx.global("limit")
        ]
    }

    // Row mapping
    rowMapper { rs ->
        [
            id: rs.getLong("id"),
            name: rs.getString("name"),
            email: rs.getString("email"),
            created: rs.getTimestamp("created")
        ]
    }

    // Or map to class
    rowMapper User.class

    // Result processing
    resultProcessor { rows ->
        return rows.collect { enrichUser(it) }
    }
}

// INSERT/UPDATE/DELETE
```

```

sqlTask("update-user") {
    datasource myDataSource

    query """
        UPDATE users
        SET name = ?, email = ?, updated = NOW()
        WHERE id = ?
    """

    parameters { ctx ->
        def user = ctx.prev
        [user.name, user.email, user.id]
    }

    // Check affected rows
    validateRowCount { rowCount ->
        if (rowCount == 0) {
            throw new NotFoundException("User not found")
        }
    }
}

// Transactions
sqlTask("transactional") {
    datasource myDataSource
    transactional true
    isolationLevel SERIALIZABLE

    action { ctx ->
        // Multiple queries in transaction
        executeUpdate("UPDATE accounts SET balance = balance - ? WHERE id = ?", [100,
fromId])
        executeUpdate("UPDATE accounts SET balance = balance + ? WHERE id = ?", [100,
toId])

        return "Transfer complete"
    }

    onError { error ->
        // Transaction automatically rolled back
        log.error("Transaction failed, rolled back", error)
    }
}

```

165.2. BatchSqlTask

Batch database operations for performance.

```
batchSqlTask("batch-insert") {
```

```

datasource myDataSource

query """
    INSERT INTO users (name, email, created)
    VALUES (?, ?, NOW())
"""

// Batch data
batchParameters { ctx ->
    def users = ctx.prev

    users.collect { user ->
        [user.name, user.email]
    }
}

batchSize 1000 // Insert 1000 at a time

onBatchComplete { batchNum, rowsAffected ->
    log.info("Batch ${batchNum} completed: ${rowsAffected} rows")
}
}

```

165.3. NoSqlTask

Access NoSQL databases.

```

// MongoDB
noSqlTask("mongo-query") {
    provider "mongodb"
    database "mydb"
    collection "users"

    // Find documents
    find {
        status: "active"
        age: [$gt: 18]
    }

    projection {
        _id: 1
        name: 1
        email: 1
    }

    sort { created: -1 }
    limit 100

    // Result mapping
}

```

```

    resultType User.class
}

// MongoDB insert
noSqlTask("mongo-insert") {
    provider "mongodb"
    database "mydb"
    collection "users"

    insert { ctx ->
        ctx.prev.collect { user ->
            [
                name: user.name,
                email: user.email,
                created: new Date()
            ]
        }
    }
}

// Redis
noSqlTask("redis-cache") {
    provider "redis"

    operation SET
    key { ctx -> "user:${ctx.global('userId')}" }
    value { ctx -> JsonOutput.toJson(ctx.prev) }
    ttl 3600 // 1 hour

    // Or GET
    // operation GET
    // key { ctx -> "user:${ctx.global('userId')}" }
}

// Elasticsearch
noSqlTask("elasticsearch") {
    provider "elasticsearch"
    index "users"

    query {
        bool: {
            must: [
                [match: [status: "active"]],
                [range: [age: [gte: 18]]]
            ]
        }
    }

    size 100
    sort [[created: "desc"]]
```

}

Chapter 166. File Operation Tasks

166.1. FileTask

Read, write, copy, move files.

```
// Read file
fileTask("read-file") {
    operation READ
    sources(["/data/input/file.txt"])

    securityConfig {
        allowedDirectories(["/data/input"])
        maxFileSize 10 * 1024 * 1024 // 10 MB
    }

    encoding "UTF-8"

    // Process content
    contentProcessor { content ->
        return content.toUpperCase()
    }
}

// Write file
fileTask("write-file") {
    operation WRITE
    destination "/data/output/result.txt"

    content { ctx ->
        JsonOutput.prettyPrint(JsonOutput.toJson(ctx.prev))
    }

    securityConfig {
        allowedDirectories(["/data/output"])
    }

    overwrite true
    createDirectories true
}

// Copy files
fileTask("copy-files") {
    operation COPY
    sources(["/data/input/*.txt"])
    destination "/data/archive/"

    preserveTimestamp true
    overwriteExisting false
}
```

```

}

// Move files
fileTask("move-files") {
    operation MOVE
    sources(["/data/temp/*.dat"])
    destination "/data/processed/"

    atomicMove true
}

// Delete files
fileTask("cleanup") {
    operation DELETE
    sources(["/data/temp/*"])

    securityConfig {
        allowedDirectories(["/data/temp"])
    }
}

```

166.2. CsvTask

Read and write CSV files.

```

// Read CSV
csvTask("read-csv") {
    operation READ
    source "/data/customers.csv"

    headers true // First row is headers
    delimiter ","
    quote '"'

    // Map to objects
    rowMapper { row ->
        [
            id: row[0] as Long,
            name: row[1],
            email: row[2]
        ]
    }

    // Or with named columns
    rowMapper { columns ->
        [
            id: columns['id'] as Long,
            name: columns['name'],
            email: columns['email']
        ]
    }
}

```

```

        ]
    }
}

// Write CSV
csvTask("write-csv") {
    operation WRITE
    destination "/data/output/results.csv"

    headers(["ID", "Name", "Email"])

    data { ctx ->
        ctx.prev.collect { user ->
            [user.id, user.name, user.email]
        }
    }

    delimiter ","
    includeHeaders true
}

```

166.3. JsonTask

Read and write JSON files.

```

// Read JSON
jsonTask("read-json") {
    operation READ
    source "/data/config.json"

    // Deserialize to class
    targetType Config.class

    // Or process as map
    processor { json ->
        json.users.collect { user ->
            processUser(user)
        }
    }
}

// Write JSON
jsonTask("write-json") {
    operation WRITE
    destination "/data/output/result.json"

    content { ctx -> ctx.prev }

    prettyPrint true
}

```

```
    serializeNulls false  
}
```

166.4. XmlTask

Read and write XML files.

```
// Read XML  
xmlTask("read-xml") {  
    operation READ  
    source "/data/data.xml"  
  
    // XPath query  
    xpath "//user[@status='active']"  
  
    // Map to objects  
    nodeMapper { node ->  
        [  
            id: node.@id.text(),  
            name: node.name.text(),  
            email: node.email.text()  
        ]  
    }  
}  
  
// Write XML  
xmlTask("write-xml") {  
    operation WRITE  
    destination "/data/output/result.xml"  
  
    root "users"  
  
    content { ctx ->  
        ctx.prev.collect { user ->  
            [  
                '@id': user.id,  
                name: user.name,  
                email: user.email  
            ]  
        }  
    }  
  
    prettyPrint true  
}
```

Chapter 167. Messaging Tasks

167.1. MessagingTask

Send and receive messages.

```
// Send message
messagingTask("send-message") {
    provider "rabbitmq"
    operation SEND

    destination "orders.queue"

    message { ctx ->
        [
            orderId: ctx.global("orderId"),
            customer: ctx.prev,
            timestamp: new Date()
        ]
    }

    headers {
        "Content-Type": "application/json"
        "X-Message-ID": UUID.randomUUID().toString()
    }

    persistent true
    priority 5
}

// Receive message
messagingTask("receive-message") {
    provider "rabbitmq"
    operation RECEIVE

    source "orders.queue"

    timeout 30000 // Wait max 30s

    messageHandler { message ->
        processOrder(message.body)
    }
}
```

167.2. KafkaTask

Apache Kafka operations.

```

// Produce to Kafka
kafkaTask("produce") {
    operation PRODUCE

    topic "user-events"
    key { ctx -> ctx.global("userId") }

    message { ctx ->
        [
            event: "USER_CREATED",
            userId: ctx.global("userId"),
            data: ctx.prev,
            timestamp: System.currentTimeMillis()
        ]
    }
}

partition 0 // Optional
headers {
    "event-type": "user.created"
}

// Delivery semantics
acks "all"
retries 3
}

// Consume from Kafka
kafkaTask("consume") {
    operation CONSUME

    topics(["user-events", "order-events"])
    groupId "taskgraph-consumer"

    pollDuration 1000 // 1 second
    maxRecords 100

    recordHandler { record ->
        log.info("Consumed: ${record.topic()}-${record.partition()}-
        ${record.offset()}")
        processRecord(record.value())
    }

    commitSync true
}

```

Chapter 168. Cloud Service Tasks

168.1. S3Task

AWS S3 operations.

```
// Upload to S3
s3Task("upload") {
    operation PUT

    bucket "my-bucket"
    key { ctx -> "uploads/${ctx.global('userId')}/file.txt" }

    content { ctx -> ctx.prev }

    // Metadata
    metadata {
        "Content-Type": "text/plain"
        "user-id": ctx.global("userId")
    }

    // Server-side encryption
    encryption "AES256"

    // Storage class
    storageClass "STANDARD_IA"
}

// Download from S3
s3Task("download") {
    operation GET

    bucket "my-bucket"
    key "data/file.txt"

    // Process content
    contentHandler { inputStream ->
        return processStream(inputStream)
    }
}

// List S3 objects
s3Task("list") {
    operation LIST

    bucket "my-bucket"
    prefix "uploads/"

    maxKeys 1000
```

```
// Filter results
filter { summary ->
    summary.size > 0 &&
    summary.lastModified.after(yesterday)
}
}
```

168.2. LambdaTask

Invoke AWS Lambda functions.

```
lambdaTask("invoke-lambda") {
    functionName "my-function"
    functionVersion "\$LATEST" // Or specific version

    payload { ctx ->
        [
            userId: ctx.global("userId"),
            data: ctx.prev
        ]
    }

    invocationType "RequestResponse" // Synchronous

    // Async invocation
    // invocationType "Event"

    // Result handling
    resultHandler { response ->
        def result = JsonSlurper().parseText(response.payload)
        return result.data
    }

    timeout 60000 // 1 minute
}
```

Chapter 169. Utility Tasks

169.1. DelayTask

Introduce delays in workflow.

```
delayTask("wait") {  
    delay 5000 // 5 seconds  
  
    // Or dynamic  
    delay { ctx ->  
        ctx.global("delayMs")  
    }  
  
    // With jitter  
    jitter 1000 // +/- 1 second  
  
    onDelay { duration ->  
        log.info("Waiting ${duration}ms")  
    }  
}
```

169.2. LogTask

Log messages and data.

```
logTask("log-result") {  
    level INFO  
  
    message { ctx ->  
        "Processed ${ctx.prev.size()} records"  
    }  
  
    // Structured logging  
    fields {  
        recordCount: ctx.prev.size()  
        userId: ctx.global("userId")  
        duration: ctx.prev.durationMs  
    }  
  
    logger "org.mycompany.workflows"  
}
```

169.3. TransformTask

Transform data between tasks.

```
transformTask("convert") {
    transformer { ctx ->
        def input = ctx.prev

        // Transform data
        return input.collect { item ->
            [
                id: item.id,
                name: item.name.toUpperCase(),
                processed: true,
                timestamp: new Date()
            ]
        }
    }

    // Or use predefined transformers
    transformer JsonToXml
    // transformer CsvToJson
    // transformer XmlToJson
}
```

Chapter 170. Task Configuration Patterns

170.1. Combining Features

```
httpTask("comprehensive") {
    // Identity
    name "Fetch User Data"
    description "Fetches user data from API"

    // HTTP config
    url "https://api.example.com/users/${userId}"
    method GET
    header "Authorization", "Bearer ${credential('api-token')}"

    // Resilience
    timeout 3000
    retryPolicy {
        maxAttempts 3
        delay 1000
        exponentialBackoff true
        retryOn(IOException, TimeoutException)
    }
    circuitBreaker {
        failureThreshold 5
        timeout 3000
    }

    // Concurrency
    concurrencyLimit 10
    resource "api-rate-limit", max: 100

    // Idempotency
    idempotencyPolicy {
        enabled true
        ttl 3600
        keyExtractor { ctx -> "fetch-user-${userId}" }
    }

    // Events
    beforeExecute {
        log.info("Fetching user ${userId}")
        startTimer()
    }

    afterExecute { result ->
        log.info("Fetched user: ${result.name}")
        stopTimer()
    }
}
```

```
onError { error ->
    log.error("Failed to fetch user", error)
    alertOps(error)
}

// Validation
responseHandler { response ->
    if (response.statusCode == 200) {
        return JsonSlurper().parseText(response.body)
    } else if (response.statusCode == 404) {
        return null // User not found
    } else {
        throw new ApiException("API error: ${response.statusCode}")
    }
}
```

Chapter 171. Summary

TaskGraph provides 15+ task types:

- ☰ **Core** - ServiceTask, ScriptTask
- ☰ **HTTP/REST** - HttpTask, RestTask, GraphQLTask
- ☰ **Database** - SqlTask, NoSqlTask, BatchSqlTask
- ☰ **File** - FileTask, CsvTask, JsonTask, XmlTask
- ☰ **Messaging** - MessagingTask, KafkaTask, RabbitMQTask
- ☰ **Cloud** - S3Task, BlobStorageTask, LambdaTask
- ☰ **Utilities** - DelayTask, LogTask, TransformTask

All tasks inherit features from TaskBase: * Retry logic * Circuit breaker * Timeout handling * Idempotency * Event emission * Error handling

Chapter 172. Next Steps

- **Chapter 7** - Task usage examples
- **Chapter 11** - TaskBase reference
- **Chapter 5** - Create custom task types

Appendix A: Appendix A: Migration Guide

Details on migrating from earlier versions.

Appendix B: Appendix B: Performance Tuning

Guidelines for optimizing TaskGraph performance.

Appendix C: Appendix C: Troubleshooting

Common issues and solutions.

Appendix D: Appendix D: API Reference

Complete API documentation (generated from source).

Glossary

Task

A unit of work in a workflow graph

TaskGraph

A directed acyclic graph (DAG) of tasks

Promise

A placeholder for a future value

DSL

Domain-Specific Language for workflow definition

Gateway

A decision point or fork/join in the workflow

Circuit Breaker

A resilience pattern to prevent cascading failures