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.*

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.

Part 1: Tasks Utility

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

Core Execution Patterns

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

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.

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.toInteger()  // 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

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

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

Single Task Execution

For simple one-off task execution:

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"
```

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
```

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.

Complete Examples

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"
```

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}"
```

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"
```

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()}"
}
```

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.

Part 2: TasksCollection

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)

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]
```

Task Registration Methods

Service Tasks

Standard executable tasks:

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

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()
```

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)
        )
    }
}
```

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"]
    }
}
```

Generic Task Registration

Register any task type:

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

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

Register Existing Tasks

Add already-created tasks:

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

tasks.register(myTask)
```

Task Discovery

Find by ID

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

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-") }
```

Get All Task IDs

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

Check if Task Exists

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

Get by Type

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

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

Lifecycle Management

Start Collection

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

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

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

Stop Collection

Stops all running tasks:

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

// Timers stopped, business rules deactivated
```

Clear Collection

Stop and remove all tasks:

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

// All tasks removed, context cleared
```

Promise Chaining API

Create sequential execution chains with fluent API:

Basic Chain

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

Chain with Initial Value

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

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

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)

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()
```

Metrics and Monitoring

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}"
```

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"
```

Active Tasks

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

Summary Statistics

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

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

Complete Examples

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"
```

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}"
```

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()
```

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)

Choosing the Right Abstraction

Use Case	Tasks	TasksCollection	TaskGraph
Quick script/one-off	✅ Perfect	Overkill	Overkill
Parallel data fetch	✅ Great	✅ Good	Overkill
Sequential pipeline	✅ Great	✅ Great (chains)	✅ Good
Event-driven system	❌ No	✅ Perfect	✅ Possible
Complex dependencies	❌ No	❌ No	✅ Required
Conditional routing	❌ No	⚠ Manual	✅ Built-in

Use Case	Tasks	TasksCollection	TaskGraph
Long-running services	❌ No	✅ Perfect	⚠️ Good
Workflow orchestration	❌ No	⚠️ Limited	✅ Perfect

Migration Path

As your needs grow, you can migrate between abstractions:

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()
```

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()
```

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.

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