FractalX Framework

Developer Guide

Decompose. Deploy. Scale.

This guide covers everything you need to annotate a Spring Boot modular monolith, run `mvn fractalx:decompose`, and get a production-ready microservice platform. No infrastructure expertise required.

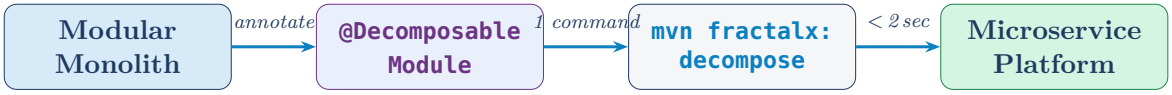
Framework Version	0.2.4
Java	17+
Spring Boot	3.2.x
Build Tool	Apache Maven 3.8+
Date	March 2026
Author	FractalX Engineering

Contents

Introduction

FractalX is a **static decomposition framework** for Spring Boot applications. It analyses the structure of a modular monolith using Abstract Syntax Tree (AST) inspection, detects inter-module dependencies, and generates a complete, independently deployable microservice platform — all from a single Maven command.

What FractalX Does



A single run produces, for every domain module:

- A standalone Spring Boot service with its own `pom.xml` (filtered Maven dependencies, no fat-JAR waste)
- Inter-service communication via **NetScope** (gRPC-based) with auto-generated client interfaces and server annotations
- Resilience4j circuit-breakers, retries, and time limiters per dependency
- OpenTelemetry tracing, health metrics, and structured log shipping
- Environment profiles (`base` / `dev` / `docker`)
- A multi-stage `Dockerfile` and `docker-compose.yml` entry

Plus four shared infrastructure services generated once per project: **API Gateway**, **Service Registry**, **Admin Dashboard**, and **Logger Service**.

Design Principles

Principle	What it means in practice
Annotation-minimal	One `@DecomposableModule` per module — that is the entire developer contract
Zero-intrusion	No changes to business logic, entities, or service classes
Safe-default	Unknown dependencies are always kept; unknown code is always copied
Incremental	Re-running `decompose` replaces only changed output
Observable	Every generated service ships with tracing, metrics, and health checks on day one

Prerequisites & Installation

System Requirements

Component	Minimum	Notes
JDK	17	Tested on OpenJDK 17, 21. Java 24 requires `-Djacoco.skip=true`
Maven	3.8	Maven wrapper (`mvnw`) recommended
Spring Boot	3.2.x	Spring Boot 2.x not supported
PostgreSQL	14+	Or MySQL 8+, MongoDB 6+ (one DB per POC)
Docker	24+	Optional — needed only for container deployment

Adding FractalX to Your Project

Add the annotation library as a `provided` dependency and declare the Maven plugin in your monolith's `pom.xml`:

pom.xml — dependency + plugin

```xml
<properties>
  <fractalx.version>0.2.4</fractalx.version>
</properties>

<!-- 1. Annotation library (compile-time only, no runtime cost) -->
<dependencies>
  <dependency>
    <groupId>org.fractalx</groupId>
    <artifactId>fractalx-annotations</artifactId>
    <version>${fractalx.version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<!-- 2. Maven plugin -->
<build>
  <plugins>
    <plugin>
      <groupId>org.fractalx</groupId>
      <artifactId>fractalx-maven-plugin</artifactId>
      <version>${fractalx.version}</version>
    </plugin>
  </plugins>
</build>
```

Scope matters. Using `scope=provided` ensures that `@DecomposableModule` and friends are *not* included in your production JAR. FractalX annotations are purely a decomposition-time contract.

Verifying the Installation

```
$ mvn fractalx:help

FractalX Maven Plugin 0.2.4
Available goals:
  fractalx:decompose   Decompose monolith into microservices
  fractalx:menu        Interactive decomposition menu
  fractalx:help        Display this help
```

Quick Start — 5 Minutes

This walkthrough converts a two-module monolith into two independently deployable microservices.

Step 1 — Create a module marker class

For each bounded context in your monolith, add a plain class annotated with `@DecomposableModule`:

OrderModule.java

```
package com.myapp.order;

import org.fractalx.annotations.DecomposableModule;

@DecomposableModule(
    serviceName = "order-service",
    port        = 8081
)
public class OrderModule { }
```

PaymentModule.java

```
package com.myapp.payment;

import com.myapp.order.OrderService;
import org.fractalx.annotations.DecomposableModule;

@DecomposableModule(
    serviceName = "payment-service",
    port        = 8082
)
public class PaymentModule {
    // Declare cross-module dependencies as fields.
    // FractalX detects these via AST and wires them via NetScope.
    private OrderService orderService;
}
```

Step 2 — Run decomposition

```
$ mvn fractalx:decompose

[INFO] FractalX 0.2.4 -- starting decomposition
[INFO] Phase 1: Parsing 22 source files ...
[INFO] Phase 2: Detected 2 modules, 1 cross-module dependency
[INFO] Generating order-service   (port 8081) ...
[INFO] Generating payment-service (port 8082) ...
[INFO] Generating fractalx-gateway, fractalx-registry,
       admin-service, logger-service ...
[INFO] BUILD SUCCESS  [610ms]
```

Step 3 — Start everything

```
# Option A: Docker Compose (recommended)
$ docker-compose -f fractalx-output/docker-compose.yml up -d

# Option B: Shell script
$ ./fractalx-output/start-all.sh
```

Admin Dashboard is available at `http://localhost:9090` immediately after startup. It shows live service topology, health status, distributed traces, and alert rules — all auto-configured.

Core Annotations Reference

`@DecomposableModule` Required

The primary annotation. Marks a class as the boundary definition for a microservice. One per package subtree.

Full attribute listing

```
@DecomposableModule(
    serviceName           = "order-service",   // Required. kebab-case
    name
    port                  = 8081,              // Required. 1-65535
    independentDeployment = true,              // Optional. default:
    false
    ownedSchemas          = {"orders",         // Optional. for Flyway
    scope
                             "order_items"}
)
public class OrderModule {
    // Declare cross-module service dependencies as fields.
    // FractalX detects constructor-injected types from the same
    // project, then generates NetScope client interfaces for them.
    private PaymentService paymentService;
}
```

Attribute	Type	Default	Description
`serviceName`	`String`	—	Generated service directory name and Spring `spring.application.name`. Use kebab-case. **Required.**
`port`	`int`	—	HTTP port for the generated service. gRPC port is automatically set to `port + 10000`. **Required.**
`independentDeployment`	`boolean`	`false`	When `true`, marks this service as having no runtime dependency on other generated services. Used by the registry startup order.
`ownedSchemas`	`String[]`	`{}`	Schema names owned by this service. Used to scope Flyway migrations and generate `@EntityScan` / `@EnableJpaRepositories` boundaries.

Table 1: `@DecomposableModule` attribute reference

`@ServiceBoundary` Optional

Applied to individual methods or classes inside a module to explicitly mark which methods are part of the public contract that other services may call. Without this annotation, FractalX infers boundary methods automatically from cross-module dependency analysis.

Usage

```
@Service
public class OrderService {

    // Explicitly mark this method as cross-service callable.
    // FractalX will add @NetworkPublic and expose it via NetScope.
    @ServiceBoundary
    public boolean isConfirmed(Long orderId) {
        return repository.existsByIdAndStatus(orderId, CONFIRMED);
    }

    // This method stays internal -- not exposed to other services.
    public void recalculateTotal(Order order) { ... }
}
```

`@AdminEnabled` Optional

Enables the admin service to include this module in its management endpoints, health summary, and topology map.

Usage

```
@DecomposableModule(serviceName = "order-service", port = 8081)
@AdminEnabled
public class OrderModule { }
```

`@DistributedSaga` Advanced

Marks a method that implements a business operation spanning multiple services. FractalX generates a `fractalx-saga-orchestrator` service with forward steps and compensation logic for each annotated method.

Usage

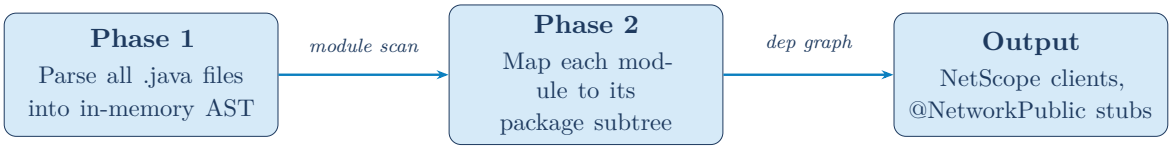
```
@Service
public class CheckoutService {

    @DistributedSaga(
        sagaId       = "checkout",
        compensation = "cancelCheckout"
    )
    public void processCheckout(Long orderId, Long userId) {
        // FractalX detects cross-service calls here and generates
        // a saga orchestrator with rollback support.
        orderService.confirm(orderId);
        paymentService.charge(userId, orderService.getTotal(orderId));
        inventoryService.reserve(orderId);
    }
```

```
    public void cancelCheckout(Long orderId, Long userId) {
        // Compensation: called if any step fails
        paymentService.refund(userId);
        orderService.cancel(orderId);
    }
}
```

Cross-Module Dependencies

How Detection Works

FractalX uses a **two-phase AST analysis** to detect cross-module dependencies without any explicit declaration:



Phase 1 parses every `.java` file in the monolith source tree and stores the result in a `Map<Path, CompilationUnit>`.

Phase 2 finds all classes annotated with `@DecomposableModule`, identifies each module's package prefix, then searches all files whose package falls within that prefix. From those files, it collects every import referencing a type that belongs to a *different* module's package prefix — these are the cross-module dependencies.

Declaring Dependencies in the Module Marker

FractalX also reads field declarations directly in the `@DecomposableModule` class. This is the recommended way to declare dependencies explicitly:

Explicit declaration (recommended)

```
@DecomposableModule(serviceName = "leave-service", port = 8084)
public class LeaveModule {
    // Both of these will generate NetScope client interfaces
    // in leave-service and @NetworkPublic in the targets.
    private EmployeeService employeeService;
    private DepartmentService departmentService;
}
```

Implicit detection (via service class imports)

```
// You do NOT need the module marker fields if the service class
// already has constructor injection -- FractalX finds it automatically.
package com.myapp.leave;

import com.myapp.employee.EmployeeService;    // cross-module import
import com.myapp.department.DepartmentService; // cross-module import

@Service
public class LeaveService {
    private final EmployeeService employeeService;
    private final DepartmentService departmentService;

    public LeaveService(LeaveRepository repo,
                        EmployeeService employeeService,
```

```
                        DepartmentService departmentService) { ... }
}
```

What Gets Generated per Dependency

For each detected cross-module dependency `A` → `B`:

Location	Artefact	Description
`service-A`	`BServiceClient.java`	`@NetScopeClient` interface mirroring the public methods of `BService`. Injected into `AService` as a drop-in replacement.
`service-A`	Resilience4j config	`CircuitBreakerConfig`, `RetryConfig`, `TimeLimiterConfig` for the B dependency.
`service-B`	`@NetworkPublic`	Added to each public method of `BService` that is called across the boundary. Exposes those methods over gRPC.
`service-B`	gRPC port	Set automatically to `B.port + 10000` (e.g., HTTP :8082 ⇒ gRPC :18082).

Table 2: Generated artefacts per cross-module dependency

Bean name convention. The type name is converted to a service name by stripping the `Service` / `Client` suffix, converting to kebab-case, and appending `-service`. For example: `PaymentService` → `payment-service`.

Configuration Reference

fractalx-config.yml

Place `fractalx-config.yml` in the root of your monolith (same level as `pom.xml`). FractalX reads this file at generation time to populate service-specific overrides into the generated `application.yml` files.

fractalx-config.yml — full reference

```yaml
fractalx:
  # Platform-level configuration
  registry-url: http://fractalx-registry:8761
  logger-url:   http://logger-service:9099
  otel-endpoint: http://jaeger:4317
  gateway-port:  9999
  admin-port:    9090

  # Gateway security (all disabled by default)
  gateway:
    security:
      enabled: false
      bearer:
        enabled: false
        secret: "change-me-in-production"
      oauth2:
        enabled: false
        jwks-uri:
    http://keycloak:8080/realms/app/protocol/openid-connect/certs
      basic:
        enabled: false
        username: admin
        password: secret
      api-key:
        enabled: false
        header: X-Api-Key

  # CORS
  cors:
    allowed-origins: "*"
    allowed-methods: "GET,POST,PUT,DELETE,OPTIONS"

  # Per-service overrides (datasource, ports, etc.)
  services:
    order-service:
      datasource:
        url:      jdbc:postgresql://localhost:5432/order_db
        username: postgres
        password: secret
    payment-service:
      datasource:
```

```
        url:      jdbc:postgresql://localhost:5432/payment_db
        username: postgres
        password: secret
```

Fallback chain. If `fractalx-config.yml` is absent, FractalX reads `application.yml` then `application.properties` in the monolith source tree and extracts what it can. All values fall back to safe defaults (e.g., `localhost:8761` for the registry).

Generated application.yml Structure

Every generated service receives three YAML profiles:

File	Active with	Content
`application.yml`	always	Port, app name, JPA dialect, Flyway, NetScope server gRPC port, logging levels
`application-dev.yml`	`-Dspring.profiles.active=dev`	Local datasource URL (`localhost`), debug logging, Swagger UI enabled
`application-docker.yml`	`-Dspring.profiles.active=docker`	Container-friendly datasource URLs using service DNS names, OTLP endpoint, registry URL from env vars

Table 3: Generated YAML profile structure per service

application.yml — excerpt from a generated service

```yaml
spring:
  application:
    name: order-service
  datasource:
    url:      ${ORDER_DB_URL:jdbc:postgresql://localhost:5432/order_db}
    username: ${ORDER_DB_USER:postgres}
    password: ${ORDER_DB_PASS:postgres}

server:
  port: 8081

netscope:
  server:
    enabled: true
    grpc:
      port: 18081           # port + 10000
  client:
```

```
    servers:
      payment-service:
        host: ${PAYMENT_HOST:localhost}
        port: 18082

management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,prometheus
```

Maven Plugin Reference

Goals

Goal	Phase	Description
`fractalx:decompose`	`generate-sources`	Main goal. Runs the full decomposition pipeline and writes all generated services to the output directory.
`fractalx:menu`	none	Launches an interactive terminal menu for configuring and running decomposition. Falls back to numbered input on Windows.
`fractalx:help`	none	Prints available goals and plugin version.

Plugin Configuration Parameters

pom.xml — full plugin configuration

```xml
<plugin>
  <groupId>org.fractalx</groupId>
  <artifactId>fractalx-maven-plugin</artifactId>
  <version>0.2.4</version>
  <configuration>
    <!-- Source root of the monolith (default: src/main/java) -->
    <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>

    <!-- Output directory for generated services -->
    <!-- default: ${project.basedir}/fractalx-output -->
    <outputDirectory>${project.basedir}/fractalx-output</outputDirectory>

    <!-- Path to fractalx-config.yml -->
    <!-- default: ${project.basedir}/fractalx-config.yml -->
    <configFile>${project.basedir}/fractalx-config.yml</configFile>

    <!-- Generate Docker artefacts (default: true) -->
    <generateDocker>true</generateDocker>

    <!-- Generate Admin service (default: true) -->
    <generateAdmin>true</generateAdmin>

    <!-- FractalX framework version used in generated POMs -->
    <frameworkVersion>0.2.4</frameworkVersion>
  </configuration>
</plugin>
```

Command-Line Overrides

All configuration parameters can be overridden on the command line:

```
# Custom output directory
$ mvn fractalx:decompose -DoutputDirectory=./my-services

# Skip Docker generation
$ mvn fractalx:decompose -DgenerateDocker=false

# Skip Admin service
$ mvn fractalx:decompose -DgenerateAdmin=false

# Skip JaCoCo (needed on JDK 24)
$ mvn fractalx:decompose -Djacoco.skip=true

# Full example
$ mvn fractalx:decompose \
    -DoutputDirectory=./output \
    -DgenerateDocker=true \
    -Djacoco.skip=true \
    -q
```

Generated Architecture

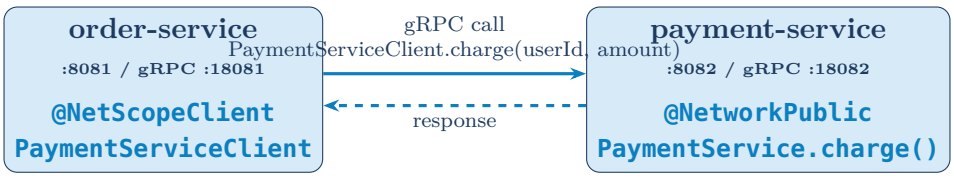
Output Directory Structure

After running `mvn fractalx:decompose`, the output directory contains one folder per service:

```
fractalx-output/
|-- order-service/
|   |-- pom.xml
|   |-- Dockerfile
|   |-- src/
|   |   |-- main/
|   |   |   |-- java/com/fractalx/generated/orderservice/
|   |   |   |   |-- OrderServiceApplication.java
|   |   |   |   |-- OtelConfig.java
|   |   |   |   |-- ServiceHealthConfig.java
|   |   |   |   |-- ResilienceConfig.java
|   |   |   |-- java/com/myapp/order/        <- copied business logic
|   |   |   |   |-- Order.java
|   |   |   |   |-- OrderRepository.java
|   |   |   |   |-- OrderService.java        <- @NetworkPublic added
|   |   |   |   |-- OrderController.java
|   |   |   |-- java/com/myapp/order/client/
|   |   |   |   |-- PaymentServiceClient.java <- NetScope client
|   |   |   |-- resources/
|   |   |       |-- application.yml
|   |   |       |-- application-dev.yml
|   |   |       |-- application-docker.yml
|   |   |       |-- db/migration/V1__init.sql
|-- payment-service/   <- same structure
|-- fractalx-gateway/
|-- fractalx-registry/
|-- admin-service/
|-- logger-service/
|-- docker-compose.yml  <- starts entire platform
|-- start-all.sh
```

Service Communication — NetScope

NetScope is FractalX's gRPC-based inter-service communication layer. It replaces Feign clients with zero-configuration, strongly typed, resilience-wrapped remote calls.



gRPC Port Convention

Rule: gRPC port = HTTP port + 10 000.
Example: `order-service` at HTTP `:8081` listens for gRPC on `:18081`. This convention is applied automatically — no configuration needed.

Generated NetScope Client

PaymentServiceClient.java — auto-generated

```java
package com.myapp.order.client;

import org.fractalx.netscope.annotations.NetScopeClient;

// Injected into OrderService as a drop-in for PaymentService.
// Calls are routed over gRPC with circuit-breaker protection.
@NetScopeClient(
    server   = "payment-service",
    beanName = "paymentService"
)
public interface PaymentServiceClient {
    boolean charge(Long userId, java.math.BigDecimal amount);
    void    refund(Long userId);
}
```

Infrastructure Services

Service	Port	Purpose
`fractalx-registry`	8761	Lightweight service registry. All domain services self-register on startup. The gateway queries it for live route resolution. Starts first — all others depend on it.
`fractalx-gateway`	9999	Spring Cloud Gateway with dynamic routing, JWT/OAuth2/API-Key auth, Resilience4j circuit breaker, token-bucket rate limiter, CORS, and observability filters.
`admin-service`	9090	Full-stack management dashboard. Includes topology visualisation, per-service health, distributed trace viewer, alert engine (SSE, webhook, Slack, email), live config editor, and gRPC inspector.
`logger-service`	9099	Centralised structured log sink. All domain services ship logs to this endpoint via HTTP. Searchable from the Admin Dashboard.

Table 4: Generated infrastructure services (one set per project, regardless of domain size)

Non-Functional Capabilities

Every generated service ships with 14 production-grade non-functional capabilities without any developer action.

Observability

Distributed Tracing — OpenTelemetry

Each service gets an `OtelConfig.java` that configures the OTLP/gRPC exporter pointing at Jaeger. W3C `traceparent` headers are propagated through all HTTP and gRPC calls automatically.

OtelConfig.java — auto-generated excerpt

```
@Configuration
public class OtelConfig {

    @Bean
    public OpenTelemetry openTelemetry() {
        return OpenTelemetrySdk.builder()
            .setTracerProvider(
                SdkTracerProvider.builder()
                    .addSpanProcessor(BatchSpanProcessor.builder(
                        OtlpGrpcSpanExporter.builder()
                            .setEndpoint(otelEndpoint)
                            .build())
                    .build())
                .build())
            .setPropagators(ContextPropagators.create(
                W3CTraceContextPropagator.getInstance()))
            .buildAndRegisterGlobal();
    }
}
```

Jaeger UI is available at `http://localhost:16686` when using the generated `docker-compose.yml`.

Health Metrics — Micrometer

`ServiceHealthConfig.java` registers a TCP health check per dependency and exposes a Micrometer gauge `fractalx.service.dependency.up` for each one. Scraped by Prometheus; visualised on the Admin Dashboard.

Correlation IDs

The gateway injects two headers on every incoming request:

Header	Description
`X-Request-Id`	Unique per HTTP request (UUID).
`X-Correlation-Id`	Spans a logical user operation across multiple service calls (set once at gateway, propagated).

Resilience

Each cross-module dependency gets three Resilience4j components configured in `ResilienceConfig.java`:

Component	Default behaviour	Config key
Circuit Breaker	Opens after 50% failure rate over a 10-call sliding window. Half-open after 30s.	`resilience4j.circuitbreaker`
Retry	3 attempts with 500ms exponential backoff for `IOException` and `TimeoutException`.	`resilience4j.retry`
Time Limiter	2s timeout per call before raising `TimeoutException`.	`resilience4j.timelimiter`

Table 5: Default resilience configuration per dependency

Gateway Capabilities

Capability	Description	Default
Dynamic routing	Routes fetched live from registry; static YAML fallback	On
Rate limiting	Token-bucket per IP+service; no Redis required	On
CORS	`CorsWebFilter`; origins/methods config-driven	On
JWT (HMAC)	Bearer token validation with shared secret	Off
OAuth2	JWK Set URI (Keycloak, Auth0, etc.)	Off
Basic Auth	Username + password	Off
API Key	`X-Api-Key` header or `?api_key=` param	Off
Circuit Breaker	Resilience4j per route + `GatewayFallbackController`	On
Request logging	Method, path, status, duration on every request	On
Tracing filter	Injects `X-Request-Id`, `X-Correlation-Id`, traceparent	On

Table 6: Gateway capabilities and defaults

Data Management

Database Isolation

FractalX generates a separate datasource configuration per service when `ownedSchemas` is specified. Each service gets:

- Scoped `@EntityScan` and `@EnableJpaRepositories` pointing only at that service's package
- Datasource properties read from `fractalx-config.yml` `fractalx.services.{name}.datasource`
- A Flyway migration path at `src/main/resources/db/migration/V1__init.sql`

Cross-service FK references. When two entities that previously shared a foreign key are split across services, FractalX generates a `ReferenceValidator` bean that replaces FK integrity with a NetScope `exists()` call at write time. Review the generated `ReferenceValidatorConfig.java` before production use.

Schema Migration Scaffold

Each service gets a Flyway `V1__init.sql` containing `CREATE TABLE` statements inferred from the service's `@Entity` classes:

application.yml — Flyway per service

```
spring:
  flyway:
    enabled:            true
    locations:          classpath:db/migration
    baseline-on-migrate: true
    schemas:            order_db    # from ownedSchemas
```

Saga Orchestration Advanced

Methods annotated with `@DistributedSaga` trigger the generation of a dedicated `fractalx-saga-orchestrator` service at port 8099.

Endpoint	Description
`POST /saga/{sagaId}/start`	Start a saga instance. Returns `correlationId`.
`GET /saga/status/{correlationId}`	Poll saga progress (STARTED, IN_PROGRESS, DONE, FAILED).
`GET /saga`	List all saga definitions and recent executions.

Table 7: Saga orchestrator REST endpoints

Stub completion required. The generated NetScope client method bodies in the saga orchestrator are stubs — you must fill in the method signatures after decomposition if the saga calls non-trivial overloaded methods.

Deployment

Local Development

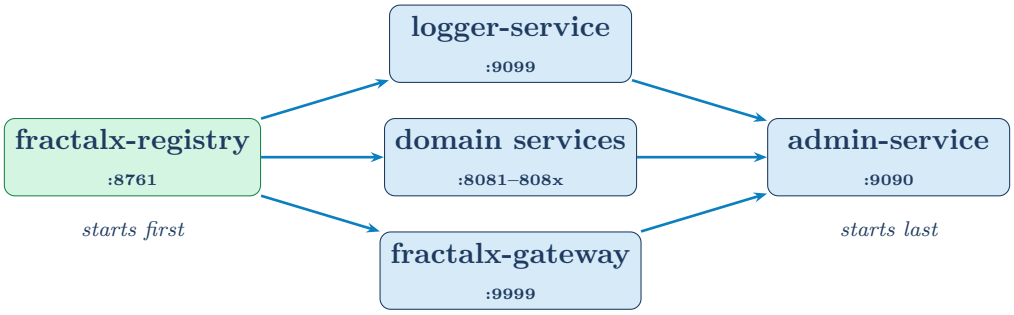
```
# 1. Start the registry first (all others depend on it)
$ cd fractalx-output/fractalx-registry
$ mvn spring-boot:run

# 2. Start domain services (any order after registry)
$ cd fractalx-output/order-service
$ mvn spring-boot:run -Dspring.profiles.active=dev

# Or use the generated start script (starts in dependency order)
$ ./fractalx-output/start-all.sh
```

Docker Compose

The generated `docker-compose.yml` starts the entire platform including Jaeger and the logger service:

```
# Start all services
$ docker-compose -f fractalx-output/docker-compose.yml up -d

# View logs
$ docker-compose -f fractalx-output/docker-compose.yml logs -f
    order-service

# Stop everything
$ docker-compose -f fractalx-output/docker-compose.yml down
```

Service Startup Order



Port Map

Service	HTTP	gRPC	Description
`fractalx-registry`	8761	—	Service registry
`fractalx-gateway`	9999	—	API gateway
`admin-service`	9090	—	Admin dashboard
`logger-service`	9099	—	Log sink
Domain service 1	8081	18081	First domain service
Domain service 2	8082	18082	Second domain service
⋮	⋮	⋮	
Domain service n	$808n$	$1808n$	n-th domain service
`Jaeger (all-in-one)`	16686	4317	Distributed trace UI

Table 8: Default port assignments

Admin Dashboard

The admin service at `http://localhost:9090` is a full-stack management interface generated for every decomposition. It requires no additional setup.

Dashboard Sections

Section	Content
Overview	Service count, health summary, recent alerts
Services	Per-service status, uptime, instance count
Network Map	Interactive topology graph of all services and dependencies
gRPC / NetScope	Active connections, call counts, latency per dependency
API Explorer	Live REST client against any generated service
Observability	Distributed traces (Jaeger embedded), log search
Alerts	Configurable alert rules (SSE, webhook, Slack, email)
Config Editor	View and override `FractalxConfig` at runtime
Data Consistency	Outbox event queue status, saga instance tracker
Settings	Platform configuration, service registration list

Table 9: Admin Dashboard sections

Alert Channels

alerting.yml — generated with 3 default rules

```yaml
fractalx:
  alerting:
    channels:
      sse:
        enabled: true           # browser push via /api/alerts/stream
      webhook:
        enabled: false
        url: https://hooks.example.com/alerts
      slack:
        enabled: false
        webhook-url: https://hooks.slack.com/services/...
      email:
        enabled: false
        smtp-host: smtp.example.com
        to: ops@example.com

    rules:
      - name: ServiceDown
        condition: "health == DOWN"
        severity: CRITICAL
      - name: HighErrorRate
        condition: "error_rate > 0.1"
        severity: WARNING
      - name: SlowResponse
```

```
        condition: "p99_latency_ms > 2000"
        severity: INFO
```

Troubleshooting

Common Issues

No modules detected

Error: `[WARN] No @DecomposableModule classes found. BUILD SUCCESS (no output generated).`

Causes and fixes:

1. `@DecomposableModule` class is not in the scanned source root. Verify `<sourceDirectory>` in plugin config.
2. The annotation is on an inner class. Move it to a top-level class.
3. The `fractalx-annotations` dependency is missing or wrong scope. Check `mvn dependency:list | grep fractalx`.

Cross-module dependency not detected

Symptom: Generated service for `A` has no `BServiceClient.java`, even though `AService` uses `BService`.

Causes and fixes:

1. The field in `AService` uses the *interface* type, not the concrete class. FractalX maps by class name. Ensure the field type is `BService` (the class), not `IBService`.
2. The import for `BService` is missing (IDE auto-imported a different class with the same name). Check the import statements.
3. Declare the dependency explicitly in the module marker: `private BService bService;` in `AModule`.

JaCoCo failure on JDK 24

Error: `Caused by: java.lang.IllegalArgumentException: Unsupported class file major version 68`

JaCoCo 0.8.11 does not support JDK 24 class files. Pass the skip flag:

```
$ mvn fractalx:decompose -Djacoco.skip=true
```

mvn fractalx:menu produces no output on Windows

Symptom: `BUILD SUCCESS` with no menu displayed.

The ANSI raw-mode menu uses `/dev/tty` which does not exist on Windows. FractalX detects this automatically (since v0.2.4) and falls back to a numbered text menu. If you still see no output, ensure your terminal is not swallowing stdout — try running from **Windows Terminal** or **PowerShell** instead of the legacy `cmd.exe`.

Generated service fails to start — datasource not found

Error: `Failed to configure a DataSource: 'url' attribute is not specified`

The generated service requires a running database. Either:

1. Create `fractalx-config.yml` with the correct datasource URL before running `decompose`, so it is embedded in the generated `application.yml`.
2. Set the environment variable before starting: `export ORDER_DB_URL=jdbc:postgresql://localhost`
3. Use the Docker Compose file which starts the database containers automatically alongside the services.

Flyway migration fails — relation already exists

Symptom: `ERROR: relation "orders" already exists`

```
# application-dev.yml -- add this to skip baseline check in dev
spring:
  flyway:
    baseline-on-migrate: true
    baseline-version: 0
```

For a clean slate in development, drop and recreate the database schema.

Diagnostic Commands

```
# Check what FractalX detected (verbose output)
$ mvn fractalx:decompose -Dfractalx.verbose=true

# List all registered services
$ curl http://localhost:8761/api/services | jq .

# Check gateway routes
$ curl http://localhost:9999/actuator/gateway/routes | jq .

# Health summary from admin
$ curl http://localhost:9090/api/health/summary | jq .

# Service topology
$ curl http://localhost:9090/api/topology | jq .
```

Frequently Asked Questions

Can I decompose an existing monolith that I did not write?

Yes. FractalX needs only to be able to *parse* the source files — it does not compile or run them. Add `@DecomposableModule` to one class per bounded context, identify the cross-module field dependencies, and run `decompose`. No modification to existing code is required.

Does FractalX modify my original source files?

No. FractalX is read-only on the monolith source tree. All generated code is written to the output directory (`fractalx-output/` by default).

Can I run decompose multiple times?

Yes. Each run replaces the output directory. It is safe to run iteratively as you refine module boundaries. The monolith source is never modified.

What databases are supported?

PostgreSQL, MySQL, and MongoDB datasource configuration is generated from `fractalx-config.yml`. H2 is supported for testing. The pruning rules recognise JPA, Hibernate, Flyway, and driver dependencies and keep or prune them per service based on import analysis.

What if my service has a non-standard port requirement?

Set `port` in `@DecomposableModule` to any valid port (1–65535). The gRPC port is always `port + 10000`. Ensure no two modules share the same port.

Can I customise the generated pom.xml?

The generated `pom.xml` is a starting point. You can edit it after generation. To persist customisations across re-runs, add a `fractalx-config.yml` `services.{name}` section — future versions will support per-service Maven dependency overrides.

Is there a Spring Boot 2.x version?

No. FractalX targets Spring Boot 3.2.x and Jakarta EE namespaces. Spring Boot 2.x uses `javax.*` namespaces which are incompatible with the generated code.

How do I add custom non-functional requirements?

Implement the `ServiceFileGenerator` interface and register it in `ServiceGenerator.buildPipeline()`. Each step receives a `GenerationContext` containing the module descriptor, output path, and platform configuration. This is the extension point for custom security, custom observability, or organisation-specific templates.

What happens to shared utility classes?

Classes that are imported by multiple modules are *copied* into each service that needs them. FractalX does not deduplicate — shared utilities should be extracted to a common library before decomposition if sharing is intentional.

Quick Reference Card

Task	How
Add annotation library	`<dependency>org.fractalx:fractalx-annotations:0.2.4<scope>provided</sc`
Add Maven plugin	`<plugin>org.fractalx:fractalx-maven-plugin:0.2.4</plugin>`
Mark a module	`@DecomposableModule(serviceName="x", port=808n)` on any class
Declare dep	Field `private BService bService;` in the module marker
Run decomposition	`mvn fractalx:decompose`
Start with Docker	`docker-compose -f fractalx-output/docker-compose.yml up -d`
Start locally	`./fractalx-output/start-all.sh`
Admin dashboard	`http://localhost:9090`
Trace viewer	`http://localhost:16686`
Registry API	`http://localhost:8761/api/services`
Skip JaCoCo (JDK 24)	`-Djacoco.skip=true`
Verbose output	`-Dfractalx.verbose=true`
Custom output dir	`-DoutputDirectory=./my-output`
Skip Docker generation	`-DgenerateDocker=false`

Table 10: FractalX quick reference

`@DecomposableModule` → `mvn fractalx:decompose` → **Production microservices**

Appendix A — Annotation Summary

Annotation	Target	Purpose
`@DecomposableModule`	Class	Primary boundary definition. Declares service name and port.
`@ServiceBoundary`	Method/Class	Explicitly marks a method as cross-service callable. Optional — FractalX infers boundaries automatically.
`@AdminEnabled`	Class	Includes the module in admin topology and management endpoints.
`@DistributedSaga`	Method	Triggers saga orchestrator generation for a multi-service operation.
`@NetworkPublic`	Method	*Generated by FractalX.* Do not use manually. Marks a method as exposed over NetScope gRPC.
`@NetScopeClient`	Interface	*Generated by FractalX.* Do not use manually. Marks an interface as a NetScope remote proxy.
`@EnableNetScopeServer`	Class	*Generated by FractalX.* Added to the main application class of any service that exposes gRPC endpoints.
`@EnableNetScopeClient`	Class	*Generated by FractalX.* Added to the main application class of any service that consumes remote services.

Table 11: Complete FractalX annotation inventory

Appendix B — Generated File Types

File	Description
`pom.xml`	Maven build with filtered + pruned deps
`Dockerfile`	Multi-stage build: builder + slim runtime
`application.yml`	Base Spring Boot configuration
`application-dev.yml`	Local dev overrides
`application-docker.yml`	Container-ready overrides
`*Application.java`	Spring Boot main class
`OtelConfig.java`	OpenTelemetry OTLP exporter configuration
`ServiceHealthConfig.java`	TCP health checks + Micrometer gauges
`ResilienceConfig.java`	Circuit breaker + retry + time limiter
`*ServiceClient.java`	NetScope `@NetScopeClient` interface
`ServiceRegistrationStep.java`	Self-registration with fractalx-registry
`V1__init.sql`	Flyway initial schema scaffold
`docker-compose.yml`	Full platform (all services + Jaeger)
`start-all.sh`	Startup script in dependency order

Table 12: Generated file types reference