Getting Started with FDD Framework

Overview

Function Driven Development (FDD) revolutionizes serverless development by providing type-safe function composition using pure `java.util.Function` with configuration-driven metadata.

Prerequisites

- Java 17 or later
- Maven 3.6+
- Basic understanding of Spring Framework

Quick Setup

1. Add FDD Starter Dependency

xml

```xml
<dependency>
    <groupId>com.fdd</groupId>
    <artifactId>fdd-starter</artifactId>
    <version>1.0.0-SNAPSHOT</version>
</dependency>
```

2. Create Your First Function

java

```java
@Component("myFunction")
public class MyFunction implements Function<String, String> {
    @Override
    public String apply(String input) {
        return "Hello, " + input + "!";
    }
}
```

3. Configure Function Metadata

Create `src/main/resources/serverless.yml`:

yaml

```yaml
serverless:
  functions:
    myFunction:
      name: "com.example.my.function"
      input: "java.lang.String"
      output: "java.lang.String"
      security:
        group: "public"
      deployment:
        cloud: "aws"
        memory: "256MB"
```

4. Enable FDD in Your Application

java

```java
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

5. Test Function Discovery

Start your application and visit:

- `GET /functions` - See all registered functions
- `GET /metrics` - View function call metrics

Key Concepts

Function Composition

```java
@Component
public class OrderService {

    @Autowired @Qualifier("userValidator")
    private Function<UserData, ValidationResult> userValidator;

    @Autowired @Qualifier("paymentProcessor")
    private Function<PaymentRequest, PaymentResult> paymentProcessor;

    public OrderResult processOrder(OrderRequest request) {
        ValidationResult validation = userValidator.apply(request.getUser());
        if (!validation.isValid()) {
            return OrderResult.failed("Invalid user");
        }

        PaymentResult payment = paymentProcessor.apply(request.getPayment());
        return payment.isSuccess() ?
            OrderResult.success() :
            OrderResult.failed("Payment failed");
    }
}
```

Configuration-Driven Security

```yaml
security:
  jwt:
    issuer: "https://auth.example.com"
  groups:
    user-management:
      description: "User operations"
      allowed-callers: ["order-service"]
    financial-operations:
      description: "Payment processing"
      roles: ["PAYMENT_PROCESSOR"]
```

Best Practices

1. **Keep Functions Pure**: No side effects in function logic
2. **Use Type-Safe Interfaces**: Always specify input/output types

3. **Configure Security**: Define appropriate security groups and roles
4. **Test Composition**: Write integration tests for function workflows
5. **Monitor Performance**: Use `/metrics` endpoint to track function performance

Next Steps

- Read Function Composition Guide
- Learn about Security Model
- Explore Cloud Deployment

Function Composition Patterns

Overview

FDD enables powerful function composition patterns using familiar Spring dependency injection, providing type safety and zero learning curve.

Basic Composition

Sequential Composition

Execute functions in sequence, passing output of one as input to the next:

```java
@Component
public class SequentialProcessor {

    @Autowired @Qualifier("stepOne")
    private Function<InputA, OutputA> stepOne;

    @Autowired @Qualifier("stepTwo")
    private Function<OutputA, OutputB> stepTwo;

    public OutputB process(InputA input) {
        OutputA intermediate = stepOne.apply(input);
        return stepTwo.apply(intermediate);
    }
}
```

Conditional Composition

Execute different functions based on conditions:

```java
@Component
public class ConditionalProcessor {

    @Autowired @Qualifier("validationFunction")
    private Function<UserData, ValidationResult> validator;

    @Autowired @Qualifier("premiumProcessor")
    private Function<OrderData, OrderResult> premiumProcessor;

    @Autowired @Qualifier("standardProcessor")
    private Function<OrderData, OrderResult> standardProcessor;

    public OrderResult processOrder(OrderRequest request) {
        ValidationResult validation = validator.apply(request.getUser());
        if (!validation.isValid()) {
            return OrderResult.failed("Invalid user");
        }

        return request.getUser().isPremium() ?
            premiumProcessor.apply(request.getOrderData()) :
            standardProcessor.apply(request.getOrderData());
    }
}
```

Parallel Composition

Execute multiple functions concurrently and combine results:

java

```java
@Component
public class ParallelProcessor {

    @Autowired @Qualifier("inventoryChecker")
    private Function<ProductRequest, InventoryResult> inventoryChecker;

    @Autowired @Qualifier("priceCalculator")
    private Function<ProductRequest, PriceResult> priceCalculator;

    @Autowired @Qualifier("shippingCalculator")
    private Function<ProductRequest, ShippingResult> shippingCalculator;

    @Autowired
    private TaskExecutor taskExecutor;

    public ProductAvailability checkProductAvailability(ProductRequest request) {
        CompletableFuture<InventoryResult> inventoryFuture =
            CompletableFuture.supplyAsync(() -> inventoryChecker.apply(request), taskExecutor);

        CompletableFuture<PriceResult> priceFuture =
            CompletableFuture.supplyAsync(() -> priceCalculator.apply(request), taskExecutor);

        CompletableFuture<ShippingResult> shippingFuture =
            CompletableFuture.supplyAsync(() -> shippingCalculator.apply(request), taskExecutor

        try {
            InventoryResult inventory = inventoryFuture.get();
            PriceResult price = priceFuture.get();
            ShippingResult shipping = shippingFuture.get();

            return new ProductAvailability(inventory, price, shipping);
        } catch (Exception e) {
            throw new RuntimeException("Failed to check product availability", e);
        }
    }
}
```

Error Handling Patterns

java

```java
@Component
public class RobustProcessor {

    @Autowired @Qualifier("primaryProcessor")
    private Function<Request, Result> primaryProcessor;

    @Autowired @Qualifier("fallbackProcessor")
    private Function<Request, Result> fallbackProcessor;

    public Result processWithFallback(Request request) {
        try {
            return primaryProcessor.apply(request);
        } catch (Exception e) {
            logger.warn("Primary processor failed, using fallback: {}", e.getMessage());
            return fallbackProcessor.apply(request);
        }
    }

    public Optional<Result> processWithOptionalResult(Request request) {
        try {
            Result result = primaryProcessor.apply(request);
            return Optional.of(result);
        } catch (Exception e) {
            logger.error("Processing failed: {}", e.getMessage());
            return Optional.empty();
        }
    }
}
```

Advanced Patterns

Function Factories

Create functions dynamically based on configuration:

java

```java
@Component
public class ProcessorFactory {

    private final Map<String, Function<Request, Result>> processors;

    public ProcessorFactory(
            @Qualifier("typeAProcessor") Function<Request, Result> typeAProcessor,
            @Qualifier("typeBProcessor") Function<Request, Result> typeBProcessor) {
        this.processors = Map.of(
            "TYPE_A", typeAProcessor,
            "TYPE_B", typeBProcessor
        );
    }

    public Result process(String type, Request request) {
        Function<Request, Result> processor = processors.get(type);
        if (processor == null) {
            throw new IllegalArgumentException("Unknown processor type: " + type);
        }
        return processor.apply(request);
    }
}
```

Function Chains

Create reusable processing chains:

```java
@Component
public class ProcessingChain {

    private final List<Function<ProcessingContext, ProcessingContext>> steps;

    public ProcessingChain(
            @Qualifier("authenticationStep") Function<ProcessingContext, ProcessingContext> aut
            @Qualifier("validationStep") Function<ProcessingContext, ProcessingContext> validat
            @Qualifier("enrichmentStep") Function<ProcessingContext, ProcessingContext> enrichm
        this.steps = List.of(auth, validation, enrichment);
    }

    public ProcessingResult process(InitialRequest request) {
        ProcessingContext context = new ProcessingContext(request);

        for (Function<ProcessingContext, ProcessingContext> step : steps) {
            context = step.apply(context);
            if (context.hasErrors()) {
                return ProcessingResult.failed(context.getErrors());
            }
        }

        return ProcessingResult.success(context.getResult());
    }
}
```

Security Model

Overview

FDD provides a comprehensive security model with automatic context propagation, role-based access control, and function-level permissions.

Security Configuration

Basic Security Setup

Configure security in `serverless.yml`:

yaml

```yaml
serverless:
  security:
    jwt:
      issuer: "https://auth.company.com"
      audience: "my-application"
    groups:
      user-management:
        description: "User operations"
        allowed-callers: ["web-ui", "mobile-app"]
      financial-operations:
        description: "Payment processing"
        roles: ["PAYMENT_PROCESSOR", "ADMIN"]
        elevated: true

  functions:
    userValidator:
      security:
        group: "user-management"
        roles: ["USER_VALIDATOR"]
        authentication: "JWT"

    paymentProcessor:
      security:
        group: "financial-operations"
        roles: ["PAYMENT_PROCESSOR"]
        authentication: "JWT"
        elevated: true
```

Security Context

Automatic Context Propagation

Security context flows automatically between function calls:

```java
@Component
public class SecureOrderProcessor {

    @Autowired @Qualifier("userValidator")
    private Function<UserData, ValidationResult> userValidator;

    public OrderResult processOrder(OrderRequest request) {
        // Security context is automatically available
        FunctionSecurityContext context = SecurityContextHolder.getContext();
        logger.info("Processing order for user: {}", context.getUserId());

        // Security validation happens automatically via AOP
        ValidationResult validation = userValidator.apply(request.getUser());

        return validation.isValid() ?
            OrderResult.success() :
            OrderResult.failed("Validation failed");
    }
}
```

Manual Security Checks

Perform additional security validations:

```java
@Component
public class AdminProcessor {

    public AdminResult performAdminOperation(AdminRequest request) {
        FunctionSecurityContext context = SecurityContextHolder.getContext();

        // Check if user has admin role
        if (!context.hasRole("ADMIN")) {
            throw new SecurityException("Admin role required");
        }

        // Check if operation is allowed for user's group
        if (!context.canAccessGroup("admin-operations")) {
            throw new SecurityException("Access denied to admin operations");
        }

        // Proceed with admin operation
        return performOperation(request);
    }
}
```

Security Groups and Roles

Group-Based Access Control

Functions are organized into security groups:

- **user-management**: User validation, profile operations
- **financial-operations**: Payment processing, billing
- **inventory-management**: Stock checking, warehouse operations
- **admin-operations**: System administration, configuration

Role-Based Permissions

Users have roles that grant access to specific functions:

- **USER**: Basic user operations
- **USER_VALIDATOR**: Can validate user data
- **PAYMENT_PROCESSOR**: Can process payments
- **INVENTORY_MANAGER**: Can check and update inventory

- **ADMIN**: Full system access

Security Interceptors

Custom Security Validation

Implement custom security logic:

```java
@Component
@Aspect
public class CustomSecurityInterceptor {

    @Around("@annotation(com.fdd.core.security.RequiresElevation)")
    public Object validateElevatedAccess(ProceedingJoinPoint joinPoint) throws Throwable {
        FunctionSecurityContext context = SecurityContextHolder.getContext();

        // Check if user session is elevated (e.g., recent re-authentication)
        if (!isSessionElevated(context)) {
            throw new SecurityException("Elevated access required");
        }

        return joinPoint.proceed();
    }

    private boolean isSessionElevated(FunctionSecurityContext context) {
        // Check if user has recently re-authenticated
        long lastAuth = (Long) context.getClaims().getOrDefault("last_auth", 0L);
        long currentTime = System.currentTimeMillis();
        return (currentTime - lastAuth) < 300000; // 5 minutes
    }
}
```

Cloud Deployment Guide

Overview

FDD leverages Spring Cloud Function for seamless deployment to all major cloud providers, with enhanced configuration and security features.

AWS Lambda Deployment

Setup

Add AWS dependencies to your `pom.xml`:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-adapter-aws</artifactId>
</dependency>
```

Configuration

Configure AWS-specific settings in `serverless.yml`:

```yaml
serverless:
  provider:
    name: aws
    runtime: java17
    region: us-east-1

  functions:
    userValidator:
      deployment:
        cloud: "aws"
        memory: "512MB"
        timeout: "30s"
        environment:
          SPRING_PROFILES_ACTIVE: "aws"
```

Deployment Commands

```bash
# Build the application
mvn clean package

# Deploy using AWS CLI
aws lambda create-function \
  --function-name user-validator \
  --runtime java17 \
  --role arn:aws:iam::123456789012:role/lambda-execution-role \
  --handler org.springframework.cloud.function.adapter.aws.FunctionInvoker::handleRequest \
  --zip-file fileb://target/fdd-demo.jar

# Update existing function
aws lambda update-function-code \
  --function-name user-validator \
  --zip-file fileb://target/fdd-demo.jar
```

Azure Functions Deployment

Setup

Add Azure dependencies:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-adapter-azure</artifactId>
</dependency>
```

Deployment

```bash
# Install Azure Functions Core Tools
npm install -g azure-functions-core-tools@4

# Deploy to Azure
mvn azure-functions:deploy
```

Google Cloud Functions

Setup

Add GCP dependencies:

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-function-adapter-gcp</artifactId>
</dependency>
```

Deployment

```bash
# Deploy to Google Cloud
gcloud functions deploy user-validator \
  --trigger-http \
  --runtime java17 \
  --source target/ \
  --entry-point org.springframework.cloud.function.adapter.gcp.GcfJarLauncher
```

Best Practices

Environment Configuration

Use profiles for different environments:

```yaml
# application-dev.yml
fdd:
  function:
    discovery:
      enabled: true
  security:
    context-propagation:
      enabled: false

# application-prod.yml
fdd:
  function:
    discovery:
      enabled: false
  security:
    context-propagation:
      enabled: true
      mode: jwt
```

Monitoring and Observability

Configure monitoring for cloud deployments:

```yaml
management:
  endpoints:
    web:
      exposure:
        include: health,info,metrics,functions
  metrics:
    export:
      cloudwatch:
        enabled: true
      prometheus:
        enabled: true
```

Security in Cloud

Ensure proper security configuration:

yaml

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${JWT_ISSUER_URI}

fdd:
  security:
    context-propagation:
      enabled: true
      mode: jwt
```