Journal of Advances in Developmental Research (IJAIDR)

E-ISSN: 0976-4844 ● Website: www.ijaidr.com ● Email: editor@ijaidr.com

Building Reactive GraphQL Services with Spring WebFlux and Netflix DGS

Saurabh Atri

srbwin@gmail.com, satri@ieee.org

Abstract:
This journal describes how to build and operate a GraphQL server using Spring WebFlux and the Netflix Domain Graph Service (DGS) framework. Spring WebFlux provides a non-blocking web runtime based on Reactive Streams and is optimized for high-concurrency, I/O-bound workloads [1][2]. Netflix DGS provides a schema-first, annotation-driven programming model on Spring Boot, and integrates internally with Spring for GraphQL to reuse its transport and execution capabilities [3][4]. The focus is practical: correct reactive boundaries (avoiding event-loop blocking), resolver batching with DataLoader, subscriptions over SSE/WebSocket, testing strategy, and operational concerns such as version compatibility and observability.

Keywords: Spring WebFlux, Project Reactor, GraphQL, Netflix DGS, Spring for GraphQL, reactive systems.

1. INTRODUCTION

WebFlux and GraphQL solve different problems. WebFlux is a non-blocking web runtime; GraphQL is an API query language and execution model. Combining them is useful when your GraphQL resolvers are primarily I/O bound (remote services, reactive databases, event streams) and you need high concurrency without thread-per-request scaling. If your resolvers are mostly blocking (JPA/JDBC/legacy SDKs), WebFlux will not help unless you isolate blocking work on dedicated schedulers.

2. BACKGROUND: SPRING WEBFLUX RUNTIME MODEL

Spring WebFlux is the reactive-stack web framework in Spring. It runs on non-blocking servers such as Reactor Netty and is based on Reactive Streams backpressure [1][2]. Instead of dedicating one thread per request, the server keeps many requests in-flight while awaiting downstream I/O.
Hard rules for correctness:

- Do not call block() on request threads (event loop).
- Avoid blocking clients and drivers inside resolvers; prefer reactive clients (WebClient) and reactive data access.
- If you must call blocking code, offload it using a bounded scheduler and treat it as a temporary containment strategy.

Containment pattern for unavoidable blocking calls:

```
Mono.fromCallable(() -> blockingLegacyCall())
        .subscribeOn(Schedulers.boundedElastic());
```

3. NETFLIX DGS OVERVIEW

The DGS framework is a GraphQL server framework designed for Spring Boot. It provides schema-first development, annotation-based data fetchers, DataLoader support for batching, and testing utilities [3].

Modern DGS releases integrate internally with Spring for GraphQL so that DGS users can benefit from Spring for GraphQL features without the framework reimplementing transports and execution [4].

4. ARCHITECTURE: DGS + SPRING FOR GRAPHQL ON WEBFLUX

At runtime, a typical WebFlux + DGS deployment looks like the following:

```
Client
        -> HTTP POST /graphql (queries, mutations)
        -> WebSocket or SSE (subscriptions)
Spring WebFlux (reactive server)
        -> Spring for GraphQL transport + execution
        -> DGS components (schema, @DgsQuery/@DgsData fetchers, DataLoaders)
        -> Downstream services (reactive DB, HTTP APIs, message brokers)
```

Spring for GraphQL provides server transports over HTTP and WebSocket [6]. Spring for GraphQL also added support for GraphQL subscriptions over SSE in recent releases [7]. DGS leverages this integration layer [4].

5. IMPLEMENTATION PATTERNS

5.1 Project setup

Minimal dependencies (Gradle Kotlin DSL):

```
dependencies {
        implementation("com.netflix.graphql.dgs:dgs-starter")
        implementation("org.springframework.boot:spring-boot-starter-webflux")
}
```

By default, DGS loads schema files from classpath locations under schema/ (configurable) [3].

5.2 Schema-first development

Example schema (schema/product.graphqls):

```
type Query {
        product(id: ID!): Product
}

type Product {
        id: ID!
        name: String!
        price: Float!
        reviews: [Review!]!
}

type Review {
        id: ID!
        productId: ID!
        rating: Int!
        body: String
}
```

5.3 Reactive queries and field resolvers

Fetchers should return non-blocking types and avoid blocking I/O. In WebFlux, use reactive clients/drivers end-to-end.
Example query fetcher:

```
@DgsComponent
public class ProductDataFetcher {
	private final ProductService service;

	public ProductDataFetcher(ProductService service) {
		this.service = service;
	}

	@DgsQuery
	public Mono<Product> product(@InputArgument String id) {
		return service.getProduct(id); // must be non-blocking
	}
}
```

5.4 DataLoader batching to avoid N+1

Nested GraphQL fields often trigger N+1 patterns (one downstream call per parent object). DGS supports DataLoaders to batch and cache within a request [3]. The batch function must be implemented as an efficient downstream query.
Mapped batch loader sketch:

```
@DgsDataLoader(name = "reviewsByProduct")
public class ReviewsByProductLoader implements MappedBatchLoader<String, List<Review>> {
	private final ReviewService reviewService;

	public CompletionStage<Map<String, List<Review>>> load(Set<String> keys) {
		return reviewService.batchReviews(keys).toFuture();
	}
}
```

6. SUBSCRIPTIONS: SSE AND WEBSOCKET

GraphQL subscriptions are the primary case where a GraphQL operation naturally returns a stream. Spring for GraphQL supports WebSocket transports for subscriptions and also supports subscriptions over SSE [6][7].
Operational guidance:

- Use WebSocket when you need bidirectional sessions, many concurrent subscriptions, or client libraries built around graphql-transport-ws.
- Use SSE when you want simple HTTP-based streaming (often easier through proxies), but be aware of connection limits and intermediaries.
- Make subscription publishers hot (shared) only when appropriate; otherwise each client connection may trigger duplicated upstream work.

7. PERFORMANCE AND RELIABILITY

Checklist:

- Event-loop safety: no blocking calls in resolvers.
- Batching: DataLoader for nested fields; aggregate downstream calls.
- Query controls: depth/complexity limits; request size limits.
- Timeouts and retries: enforce timeouts for downstream calls; prefer bounded retries with jitter.
- Backpressure: keep streaming publishers bounded; apply buffering policies intentionally.

8. OBSERVABILITY

In reactive services, correlation and tracing must not rely on thread-local assumptions. Use structured logging and tracing that propagates context across reactive boundaries. Add GraphQL-level timing and error telemetry (operation name, field latencies, downstream call timing) using instrumentation.

9. TESTING STRATEGY

Recommended layers:

- Schema validation tests: ensure the schema loads and basic operations execute.
- Fetcher unit tests: direct tests for resolver logic with mocked services.
- End-to-end tests: WebFlux WebTestClient against /graphql for typical queries and error cases.
- Load tests: concurrency-focused tests to detect event-loop blocking and downstream saturation.

10. VERSION COMPATIBILITY AND UPGRADE GUIDANCE

DGS major versions track Spring Boot major versions. DGS Framework 11.0.0 is built on Spring Boot 4; applications on Spring Boot 3 should remain on DGS 10.x until upgraded [5].

Spring Boot	Recommended DGS	Notes
3.x	10.x	Use DGS 10 baseline; upgrade Boot first.
4.x	11.x	DGS 11 baseline aligns with Boot 4 [5].

11. CONCLUSION

Spring WebFlux and Netflix DGS are a strong fit when you need a GraphQL server that handles high concurrency and streaming subscriptions while keeping I/O non-blocking. The core engineering discipline is enforcing non-blocking boundaries, batching resolver access with DataLoader, and operating the service with strong query controls and observability.

REFERENCES:

1. Spring Framework Reference Documentation, “Web on Reactive Stack (Spring WebFlux).” https://docs.spring.io/spring-framework/reference/web-reactive.html (accessed 2026-01-03).
2. Spring Framework Reference Documentation, “Spring WebFlux.” https://docs.spring.io/spring-framework/reference/web/webflux.html (accessed 2026-01-03).
3. Netflix DGS Framework Documentation, “DGS Framework - Getting Started.” https://netflix.github.io/dgs/ (accessed 2026-01-03).
4. Netflix DGS Framework Documentation, “Spring GraphQL Integration.” https://netflix.github.io/dgs/spring-graphql-integration/ (accessed 2026-01-03).
5. Netflix DGS Framework GitHub Releases, “DGS Framework 11.0.0 (Spring Boot 4 baseline).” https://github.com/Netflix/dgs-framework/releases (accessed 2026-01-03).
6. Spring for GraphQL Reference Documentation, “Server Transports.” https://docs.spring.io/spring-graphql/reference/transports.html (accessed 2026-01-03).

7. Spring for GraphQL Project Wiki, “Spring for GraphQL 1.3 - SSE Transport.” https://github.com/spring-projects/spring-graphql/wiki/Spring-for-GraphQL-1.3 (accessed 2026-01-03).