Spring Framework 5.2: Core Container Revisited

Juergen Hoeller
Spring Framework Lead
Pivotal

Core API Revision

Java 8+ Baseline in Spring Framework 5

- **Entire framework codebase is Java 8 based**
  - internal use of lambda expressions and collection streams
  - efficient introspection of constructor/method parameter signatures
- **Framework APIs can expose Java 8 API types**
  - Executable, CompletableFuture, Instant, Duration, Stream
  - java.util.function interfaces: Supplier, Consumer, Predicate
- **Framework interfaces make use of Java 8 default methods**
  - existing methods with default implementations *for convenience*
  - new methods with default implementations *for backwards compatibility*

spring by Pivotal

Nullability

- **Comprehensive nullability declarations across the codebase**
  - non-null by default + individual `@Nullable` declarations
- **The Java effect: nullability validation in IntelliJ IDEA and Eclipse**
  - allowing applications to validate their own interaction with Spring APIs
- **The Kotlin effect: straightforward assignments to non-null variables**
  - Kotlin compiler only allows such assignments for APIs with clear nullability
- **Currently directly supported + JSR-305 meta-annotations**
  - collaboration on common code analysis annotations with Google & JetBrains

Programmatic Lookup via ObjectProvider

- `@Autowired ObjectProvider<Foo> foo`
- `ObjectProvider<Foo> foo = ctx.getBeanProvider(Foo.class)`

- **ObjectProvider methods with nullability declarations**
  - `@Nullable T getIfAvailable()`
  - `@Nullable T getIfUnique()`

- **Overloaded variants with java.util.function callbacks (new in 5.0)**
  - `T getIfAvailable(Supplier<T> defaultSupplier)`
  - `void ifAvailable(Consumer<T> dependencyConsumer)`
  - `T getIfUnique(Supplier<T> defaultSupplier)`
  - `void ifUnique(Consumer<T> dependencyConsumer)`

spring by Pivotal

Bean Stream Retrieval via ObjectProvider

- `@Autowired ObjectProvider<Foo> foo`
- `ObjectProvider<Foo> foo = ctx.getBeanProvider(Foo.class)`

- **Individual object retrieval (primary/unique)**
  - `T getObject()`
  - `@Nullable T getIfAvailable()`
  - `@Nullable T getIfUnique()`

- **Iteration and stream retrieval (new in 5.1)**
  - `Iterator<T> iterator()`
  - `Stream<T> stream()`
  - `Stream<T> orderedStream()`

Programmatic Bean Registration with Java 8

```
// Starting point may also be AnnotationConfigApplicationContext

GenericApplicationContext ctx = new GenericApplicationContext();
ctx.registerBean(Foo.class);
ctx.registerBean(Bar.class,
        () -> new Bar(ctx.getBean(Foo.class)));


// Or alternatively with some bean definition customizing

GenericApplicationContext ctx = new GenericApplicationContext();
ctx.registerBean(Foo.class, Foo::new);
ctx.registerBean(Bar.class,
        () -> new Bar(ctx.getBeanProvider(Foo.class)),
        bd -> bd.setLazyInit(true));
```

Programmatic Bean Registration with Kotlin

```
// Java-style usage of Spring's Kotlin extensions

val ctx = GenericApplicationContext()
ctx.registerBean(Foo::class)
ctx.registerBean { Bar(it.getBean(Foo::class)) }


// Gradle-style usage of Spring's Kotlin extensions

val ctx = GenericApplicationContext {
    registerBean<Foo>()
    registerBean { Bar(it.getBean<Foo>()) }
}
```

Performance Tuning

spring by Pivotal

Component Scanning

- **Classpath scanning on startup may be slow**
  - `<context:component-scan>` or `@ComponentScan`
  - file system traversal of all packages within the specified base packages
- **The common solution: narrow your base packages**
  - Spring only searches within the specified roots in the classpath
  - alternatively: fully enumerate your component classes (no scanning at all)
- **New variant in 5.0: a build-time annotation processor**
  - spring-context-indexer generates `META-INF/spring.components` per jar
  - automatically used at runtime for compatible component-scan declarations

spring by Pivotal

Annotation Processing

- **New MergedAnnotations API in 5.2**
  - sophisticated introspection of meta-annotation arrangements
  - backing Spring's common AnnotationUtils and AnnotatedElementUtils now
- **Enabling a custom annotation registry**
  - registering presence/absence of certain annotations per component class
  - bean post-processors avoid unnecessary introspection of methods/fields
- **Integration with indexers (e.g. Jandex) ?**
  - adapting index metadata to Spring's annotation lookup facilities on startup
  - however, prototyping efforts did not lead to significant gains yet

Component Model Implications

- **Most efficient: purely programmatic functional registration**
  - no component scanning, no reflective factory methods
  - no annotation-config setup (no annotation post-processors)

- **@Configuration(proxyBeanMethods=false) in 5.2**
  - same effect: @Bean methods on non-@Configuration classes
  - avoiding runtime generation of CGLIB subclasses
  - drawback: no interception of cross-@Bean method calls

- **Prefer interface-based proxies over CGLIB proxies**
  - again: avoiding runtime generation of CGLIB subclasses

Third-Party Libraries

- **Persistence provider bootstrapping**
  - consider specifying an async bootstrap executor for JPA / Hibernate
  - Spring Data+Boot: spring.data.jpa.repositories.bootstrap-mode=deferred
- **Hibernate ORM 5.4.5**
  - internal performance improvements, lower memory consumption
  - optional: bytecode enhancement (also for lazy loading), Jandex integration
- **Jackson 2.9 / 2.10**
  - Spring Framework 5.2 requires Jackson 2.9.7+, supports Jackson 2.10
  - consider Jackson's alternative data formats: Smile, CBOR, JSON Arrays

GraalVM Native Images (experimental)

- **Prepared for GraalVM since Spring Framework 5.1**
  - avoiding unnecessary internal reflection
  - skipping parameter name discovery

- **As of Spring Framework 5.2: custom setup for GraalVM 19 GA**
  - explicit reflection configuration and command line args necessary
  - note: Graal's SubstrateVM is still an early adopter plugin in 19 GA

- **Expected for Spring Framework 5.3: out-of-the-box setup**
  - automatic reflection setup through custom Graal configuration integration
  - https://github.com/spring-projects/spring-framework/wiki/GraalVM-native-image-support

spring by Pivotal

Looking Forward: OpenJDK's Project Loom

- **"Fibers"**
  - lightweight user-mode threads
  - efficient scheduling within the JVM
- **Classic Thread API adapted to fibers**
  - e.g. ThreadLocal effectively "fiber-local"
- **Fibers versus reactive programming**
  - new life for traditional synchronous programming arrangements
  - reactive programming primarily for backpressure handling ?
  - Spring MVC versus Spring WebFlux

spring by Pivotal

Reactive @ All Levels

spring by Pivotal

Reactive Web Results (with MVC or WebFlux)

```
@Controller
public class MyReactiveWebController {

    private final UserRepository repository;

    public MyReactiveWebController(UserRepository repository) {
        this.repository = repository;
    }

    @GetMapping("/users/{id}")
    public Mono<User> getUser(@PathVariable Long id) {
        return this.repository.findById(id);
    }

    @GetMapping("/users")
    public Flux<User> getUsers() {
        return this.repository.findAll();
    }
}
```

Reactive Transactions (e.g. with R2DBC)

```
@Service
public class MyReactiveTransactionalService {

    private final UserRepository repository;

    public MyReactiveTransactionalService(UserRepository repository) {
        this.repository = repository;
    }

    @Transactional
    public Mono<User> getUser(Long id) {
        return this.repository.findById(id);
    }

    @Transactional
    public Flux<User> getUsers() {
        return this.repository.findAll();
    }
}
```

Reactive Transaction Setup

- **ReactiveTransactionManager SPI**
  - as an alternative to PlatformTransactionManager
  - relies on Reactor context instead of ThreadLocals
- **Implementations for R2DBC, MongoDB, Neo4j**
  - available in Spring Data Moore
  - also usable with programmatic TransactionalOperator
- **Common setup through @EnableTransactionManagement**
  - automatic adaptation to each annotated method signature
  - works with any Reactive Streams Publisher as return value

Reactive Messaging (e.g. with RSocket)

```
@Controller
public class MyReactiveMessagingController {

    @MessageMapping("echo-async")
    public Mono<String> echoAsync(String payload) {
        return ...
    }

    @MessageMapping("echo-stream")
    public Flux<String> echoStream(String payload) {
        return ...
    }

    @MessageMapping("echo-channel")
    public Flux<String> echoChannel(Flux<String> payloads) {
        return ...
    }
}
```

Reactive Application Events

```
@Service
public class MyReactiveApplicationEventService {

    @EventListener
    public Mono<Void> processRefresh(ContextRefreshedEvent event) {
        return ...
    }

    @EventListener
    public Mono<MyOtherEvent> processWithResponse(MyEvent event) {
        return ...
    }

    @EventListener
    public CompletableFuture<MyOtherEvent> processAsync(MyEvent event) {
        return ...
    }
}
```

Reactive API Adapters

- **Spring automatically adapts common reactive API types**
  - according to return/parameter declarations in user components
  - org.reactivestreams.Publisher interface or library-specific API types
  - adapted to Reactor Flux/Mono for internal processing purposes

- **Traditionally supported: RxJava 1 & 2, j.u.c.Flow, CompletableFuture**
  - RxJava: Flowable, Observable, Single, Maybe, Completable
  - on JDK 9+: java.util.concurrent.Flow.Publisher interface

- **New in 5.2: support for Kotlin coroutines ("suspend fun")**
  - Flow and Deferred return values, as exposed by Kotlin-based code

Spring Framework 5.2

September 2019

Java 8 API Refinements
Annotation Processing
Reactive Transactions
RSocket Messaging
Kotlin Coroutines

Spring Boot 2.2
October 2019

Building on
Spring Framework 5.2
& Spring Data Moore

Enjoy!

spring by Pivotal