SPRING IO

#springio24

SPRING FRAMEWORK 6.2
Core Container Revisited

JUERGEN HOELLER
@springjuergen

MAY · 30-31 · 2024

SPRINGIO.NET

Agenda

- **Autowiring Algorithm**
  - Autowiring by Type+Qualifier
  - Autowiring by Bean Name
  - Primary vs. Fallback Beans

- **Container Initialization**
  - Singleton Locking
  - Background Initialization
  - Lifecycle Management

Autowiring Algorithm

Autowiring by Type+Qualifier

- Spring's general autowiring algorithm has several steps

a. *determine all beans which match the given type*
b. *select the beans which match the given qualifiers*
c. for non-unique matches, *identify primary candidate*
d. if still not unique, *match qualified name against bean name*

Autowiring by Type+Qualifier

```
@Bean
public DataSource commonDataSource() {
    ...
}

@Bean
public MyRepository repository(DataSource ds) {  // type only
    ...
}
```

Autowiring by Type+Qualifier

```
@Bean @MyQualifier
public DataSource commonDataSource() {
    ...
}

@Bean
public MyRepository repository(@MyQualifier DataSource ds) {
    ...
}
```

Autowiring by Bean Name

- As of 6.2, there is a *“fast path” for bean name matches*
- Optimistic assumption: *check for bean name match first*

- Parameter/field name to match bean name convention
  - or explicit `@Qualifier(...)` value
- Otherwise, the standard autowiring algorithm applies
  - e.g. bean name mismatch or qualifier/primary conflict

Autowiring by Bean Name

```
@Bean
public DataSource commonDataSource() {
    ...
}

@Bean
public MyRepository repository(DataSource commonDataSource) {
    ...
}
```

Autowiring by Bean Name

```
@Bean
public DataSource commonDataSource() {
    ...
}

@Bean
public MyRepository repository(
        @Qualifier("commonDataSource") DataSource dataSource) {
    ...
}
```

Primary vs. Fallback Beans

- As of 6.2, there is also the notion of fallback beans
- *Primary beans* vs. *regular beans* vs. *fallback beans*

- `@Primary` beans override regular beans
  - to be selected in case of multiple type matches
- Regular beans override `@Fallback` beans
  - regular bean selected if other matches are fallback beans

Primary vs. Fallback Beans

```
@Bean @Primary
public DataSource commonDataSource() {   // primary among type
    ...
}

@Bean
public DataSource otherDataSource() {
    ...
}
```

Primary vs. Fallback Beans

```
@Bean
public DataSource commonDataSource() {   // only non-fallback
    ...
}

@Bean @Fallback
public DataSource otherDataSource() {
    ...
}
```

Container Initialization

Singleton Locking

- Common singleton initialization lock
- On startup: *initializing all non-lazy singletons*

- Order of registration + bean dependency structure
  - as well as explicit depends-on declarations
- Inverse order on shutdown
  - last created, first destroyed

Singleton Locking

- As of 6.2, lenient locking with `tryLock` fallback
- Bonus: Virtual Threads friendly, using `ReentrantLock`

- Still *consistent singleton locking in main bootstrap thread*
  - just lenient for other threads while main thread holds lock
- Avoiding immediate deadlock risk
  - in particular for unexpected side interactions on startup

Background Initialization Options

- Traditionally, Spring uses a *single startup thread*
- Consistent initialization of dependent singletons

- Lenient locking makes it feasible to use *multiple threads*
  - a common feature request over many years
- Multiple threads lead to less predictable startup behavior
  - still recommended: *stay single-threaded by default*

Background Initialization Options

- Asynchronous initialization *within specific bean classes*
- Custom internal handling of execution/completion state

- E.g. asynchronous JPA bootstrapping since 4.3
  - `*EntityManagerFactoryBean.setBootstrapExecutor`
- Can benefit from domain-specific characteristics
  - lazy interaction with `EntityManagerFactory` proxy

Declarative Background Initialization

- As of 6.2, background initialization of *individual beans*
- Typically in combination with `@Lazy` injection points

- `@Bean(bootstrap=BACKGROUND)`
  - uses common bootstrap executor at BeanFactory level
- Special `@DependsOn` semantics
  - forces dependency initialization in main thread first

Declarative Background Initialization

```
@Bean(bootstrap=BACKGROUND)
public DataSource commonDataSource() {
    ...
}

@Bean
public MyRepository repository(@Lazy DataSource ds) {
    ...
}
```

Lifecycle Management

- Spring-managed *start/stop* lifecycle
- `Lifecycle` and `SmartLifecycle` interfaces

- Auto-startup of specific lifecycle beans
  - after instantiating all non-lazy singletons
- Graceful parallel stopping on shutdown
  - before any destroy methods are invoked

Lifecycle Management

- Also part of the arrangement: *pause/restart*
- E.g. for snapshots (CRaC) or reconfiguration

- Supported by message listener containers etc.
  - temporarily stopping message delivery
- As of 6.1, also supported by executors/schedulers
  - pausing further tasks and triggers

Customizable Lifecycle Management

- A challenge: *order of start/stop callbacks*
- Lifecycle phase and interdependency order

- Phase configurable for every `SmartLifecycle` bean
  - within same phase, depends-on taken into account
- As of 6.2, executors/schedulers use own default phase
  - `Integer.MAX_VALUE/2` with room for earlier stopping

Customizable Lifecycle Management

```
@Bean
public ThreadPoolTaskExecutor taskExecutor() {
    …
    executor.setPhase(...);
    …
}

@Bean
public ThreadPoolTaskScheduler taskScheduler() {
    …
    scheduler.setPhase(...);
    …
}
```

Core Container Revisited

New in Spring Framework 6.2

- Fast shortcut autowiring by name
- Fallback bean definitions
- Declarative background initialization
- Revised lifecycle phases

- *Currently available: 6.2 M3*
- *6.2 GA in November 2024*

THANKS!!!

JUERGEN HOELLER

@springjuergen