**Spring features**

Types of IOC

* 1. Dependency lookup
  2. Dependency injection

Injection is preferred over lookup as

* 1. Easier to test
  2. Less code
  3. Lesser error prone

Java configuration is advantageous over annotation in case of 3rd party libraries whose source code cannot be modified

XML configuration is used to decouple configuration from actual code

Framework for faster application development

* 1. Non-intrusive application development using POJO's

No need to extend specialized classes or implement interfaces

* 1. Loose coupling using DI

Adv of DI-easy to test my mocking out dependencies

Annotations

* 1. @Component - declare a bean
     1. @Repository - data access component (DB operations) / convert vendor specific exceptions to Spring's DataAccessException
     2. @Service (providing service operations)
     3. @Controller - web component
     4. @Configuration - used to create Java configuration classes (Classes that contain bean declaration)
  2. @Autowired
  3. @Qualifier - specify bean name to be injected
  4. @Required - Mandatory dependency
  5. @Lazy (lazy-init)- Injected first time when it is used (not to be used) as IoC container by default eagerly loads the bean definitions
  6. @ImportResource - used to import bean s from xml files into java configuration class
  7. @Primary(primary) - in case of multiple dependencies of same type which dependency should be used
  8. @DependsOn (depends-on) - Used to tell Spring to instantiate a bean after its dependencies have been created
  9. @Profile - used to specify which beans should be instantiated when the profile is set

|  |  |
| --- | --- |
| **Spring** | **JSR** |
| Component | Named |
| Autowired | Inject |
| Autowired + Qualifier | Resource |

Creating the application context

* 1. AnnotationConfigApplicationContext - Java based configuration
  2. ClassPathXmlAplicationContext - cml based configuration

@PropertySource - used to specify the location of property file from which property name specified in ${} will be read

PropertySourcesPlaceHolderconfigurer needs to initialized in order to resolve properties in ${} and should be specified in static method in configuration classes as all required properties can be read before bean is created

Order precedence

System properties

Environment properties

Application property (can be overridden to be used first)

@Autowired

* 1. First search by type
  2. Then by @Qualifier
  3. By name

@Value is used to inject scalar value

Constructor and setter injection use proxy pattern whereas field injection uses Reflection

@Scope

Used to define the scope of the bean, singleton or prototype

By default all beans are singleton

For specifying a bean as prototype we need to define the proxyMode as well

* 1. DEFAULT/TARGET\_CLASS - Uses CGLIB based class proxy - Created at runtime by sub classing the target class
  2. INTERFACES -Uses jdk dynamic proxy - Created at runtime by implemented the interface of the target class

When a bean with scope prototype is defined singleton instance of the proxy( of that bean) is created and when get() method is called on the proxy it creates new instance of the bean and returns this instance, the logic for creating new bean is defined in proxy

Bean lifecycle

* 1. Bean definitions are read, based on which the beans are instantiated, if constructor injection is used then dependant beans are instantiated and initialization also happens
  2. Bean post processors are invoked before initialization
  3. Beans are initialized (setter injection)
  4. Bean post processors are invoked after initialization (life cycle methods - init() and destroy())

Invoking before bean initialization

* 1. @PostConstruct

Should be non-argument method, called by Spring after instantiating the bean (can have any access types but private is recommended)

* 1. @Bean(init-method)
  2. Implement InitializingBean (Spring specific)
  3. init-method in .xml file

Order of execution

* 1. PostConstruct
  2. afterPropertiesSet
  3. Init-method

Invoking after bean initialization

* 1. @PreDestroy

Should be non-argument method, called by Spring after initializing the bean (can have any access types but private is recommended)

* 1. @Bean(destroyMethod)
  2. Implement DisposableBean (Spring specific)
  3. destroy-method in .xml file

Order of execution

* 1. PreDestroy
  2. afterPropertiesSet
  3. destroy-method

Implement BeanNameAware if you have multiple beans of same type and you need to log the name of each bean, Spring calls setName method after bean is created and before lifecycle callback methods

Implement ApplicationContextAware if the bean needs to access the application context (Eg to register shutdownhook)

In order to convert non-primitive values to values which are injected custom converter beans need to be defined

convert() method needs to implemented for converting

This converter needs to be added in conversionService bean which is created from ConversionServiceFactoryBean (factory method)

Injecting collections

Use <map>, <list>,<set> and <prop>, prop only allows String while the remaining allows any types

Implment FactoryBean if you want to create beans which cannot be created using new operator but through factory methods, Spring calls the getInstance() method to retrieve instances from the factory class

If factory class is provided by 3rd party using factory-bean and factory-method attributes

PropertyEditors used to convert to and from String values to respective Types

**Spring Boot**

Typical spring applications require the need to specify the dependencies in build file and check if the different versions are compatible, then configuration classes need to defined using Spring classes

@SpringBootApplication - @SpringBootConfiguration (specialized Configuration) + @ComponentScan + @EnableAutoConfiguration

AOP enabled through spring-boot-aop-starter

**AOP**

AOP is used to separate out cross cutting concerns from the business logic

**@EnableAspectJAutoProxy**

Used to enable aspects

By default Spring AOP uses JDK Dynamic proxies (provided the target object implements any interface) ,if not then CGLIB proxies are used

In order to use CGLIB proxy explicitly use **@EnableAspectJAutoProxy(proxyTargetClass = true)**

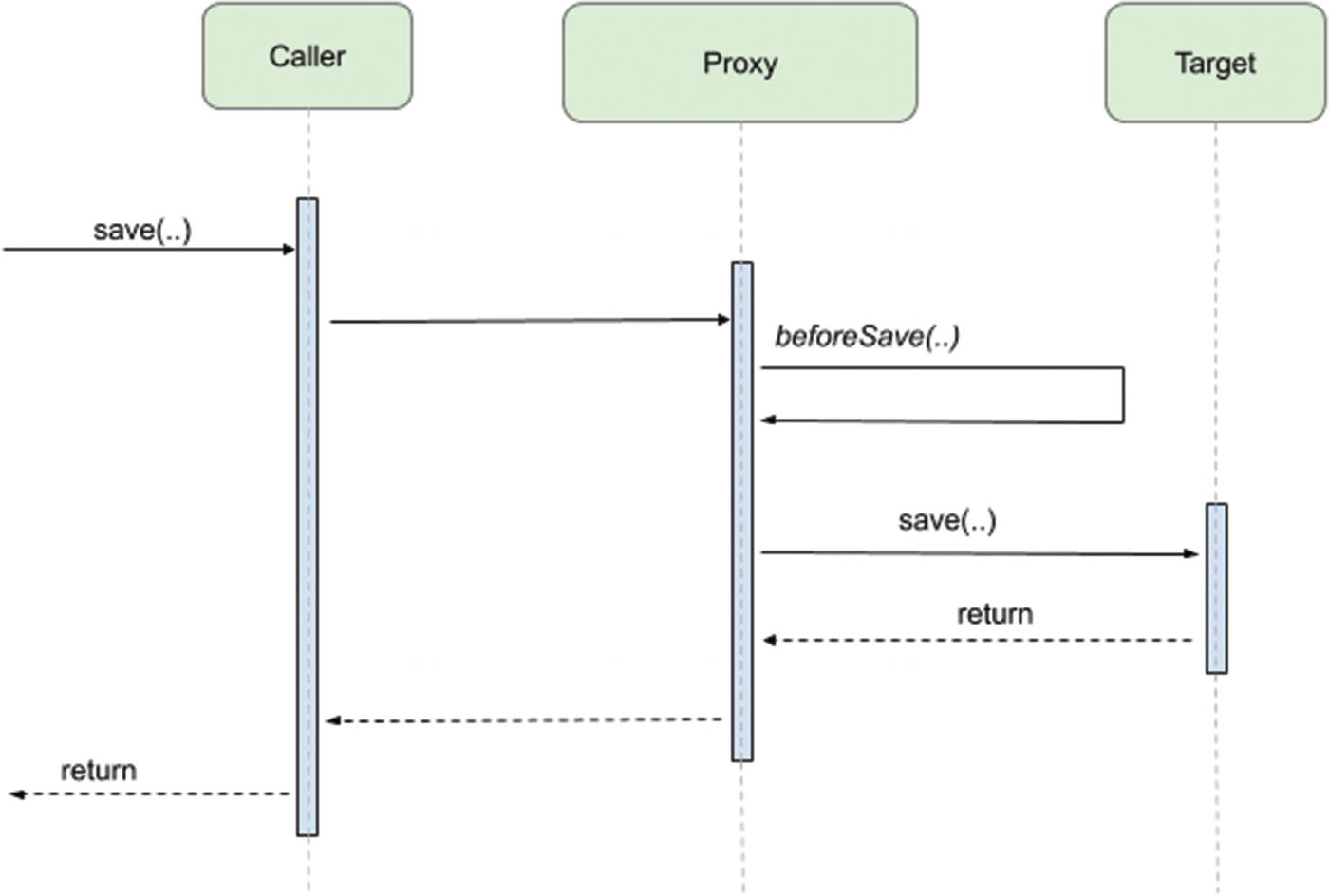
* 1. Spring AOP does have its limitations, and they are listed below.
  2. Only public Join Points can be advised (you probably suspected that).
  3. Aspects can be applied only to Spring Beans.
  4. Even if Spring AOP is not set to use CGLIB proxies, if a Join Point is in a class that does not implement an interface Spring AOP will try to create a CGLIB proxy
  5. If a method in the proxy calls another method in the proxy, and both match the pointcut expression of an advice, the advice will be executed only for the first method. This is the proxy’s nature; it executes the extra behaviour only when the caller calls the target method.

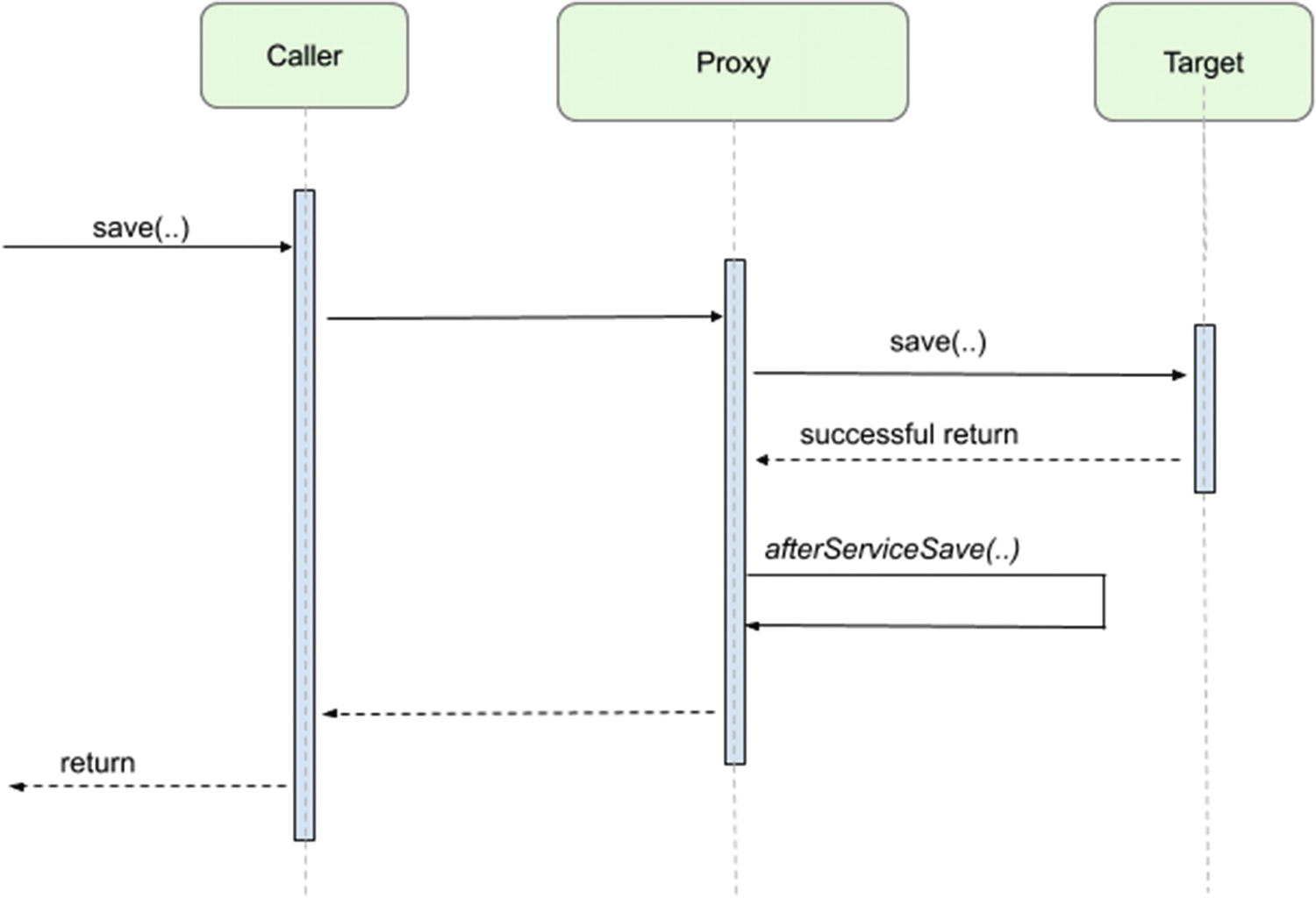
Around advice uses the ProceedingJoinPoint class which extends JoinPoint, other advices use JoinPoint interface

After Returning has additional argument which returns the object return by the target method

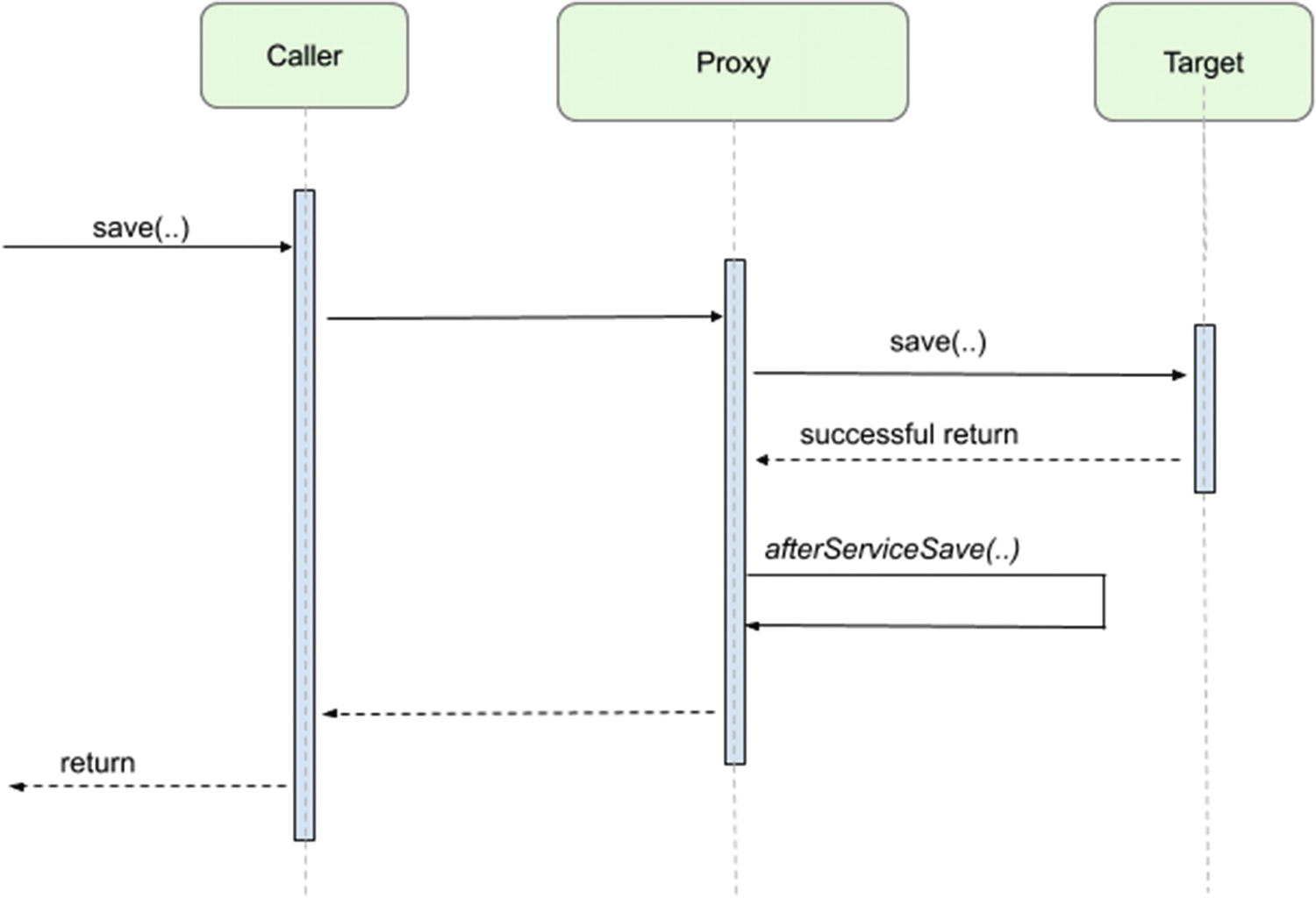
After Throwing has additional argument which returns the exception thrown from target method

Application of Advices

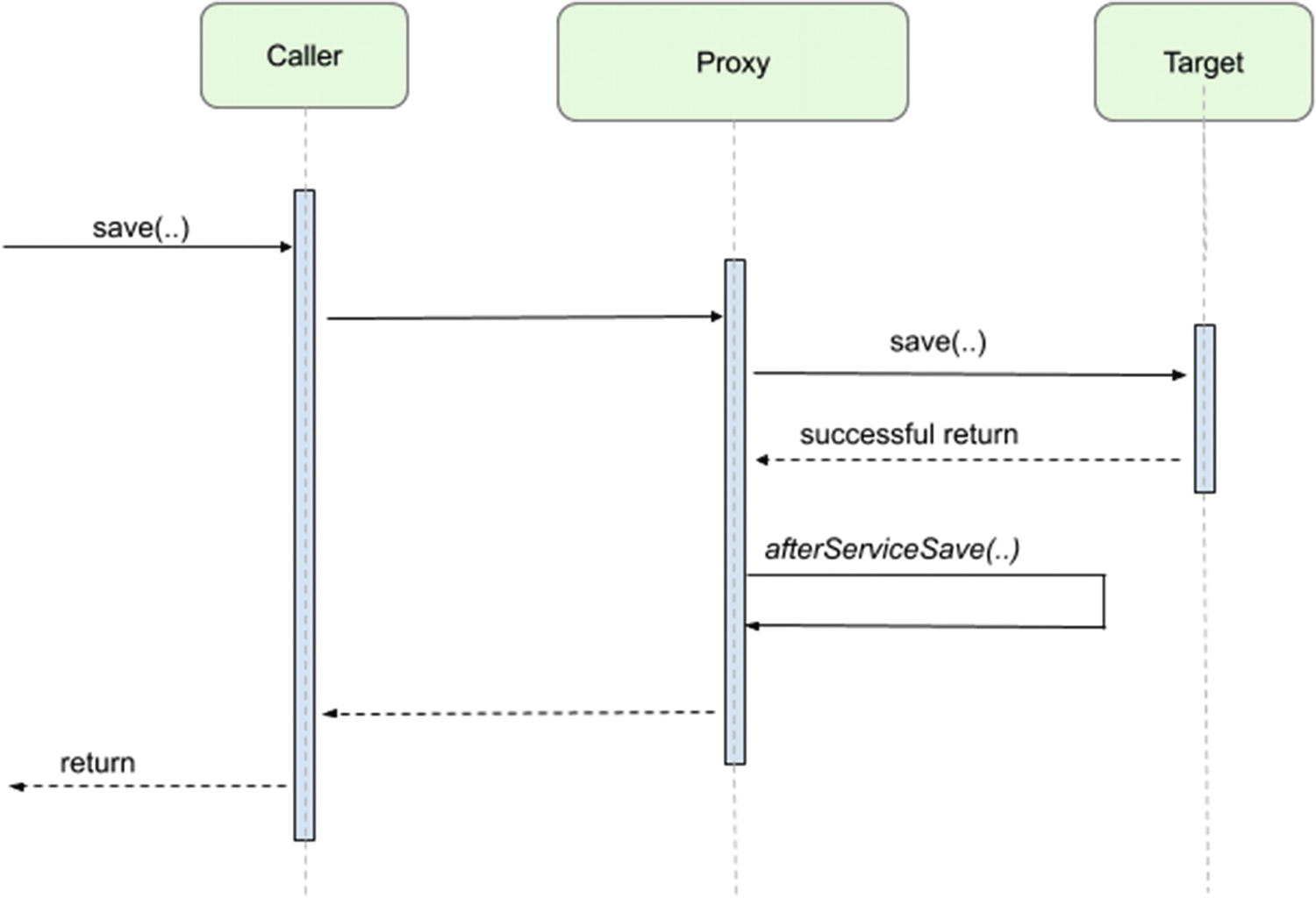
* 1. Before - Validations (Before execution of method, target method not executed if exception is thrown in proxy method)
  2. 
     1. After Returning - Caching (After successful execution of method, not called if exception is throw from the target method)



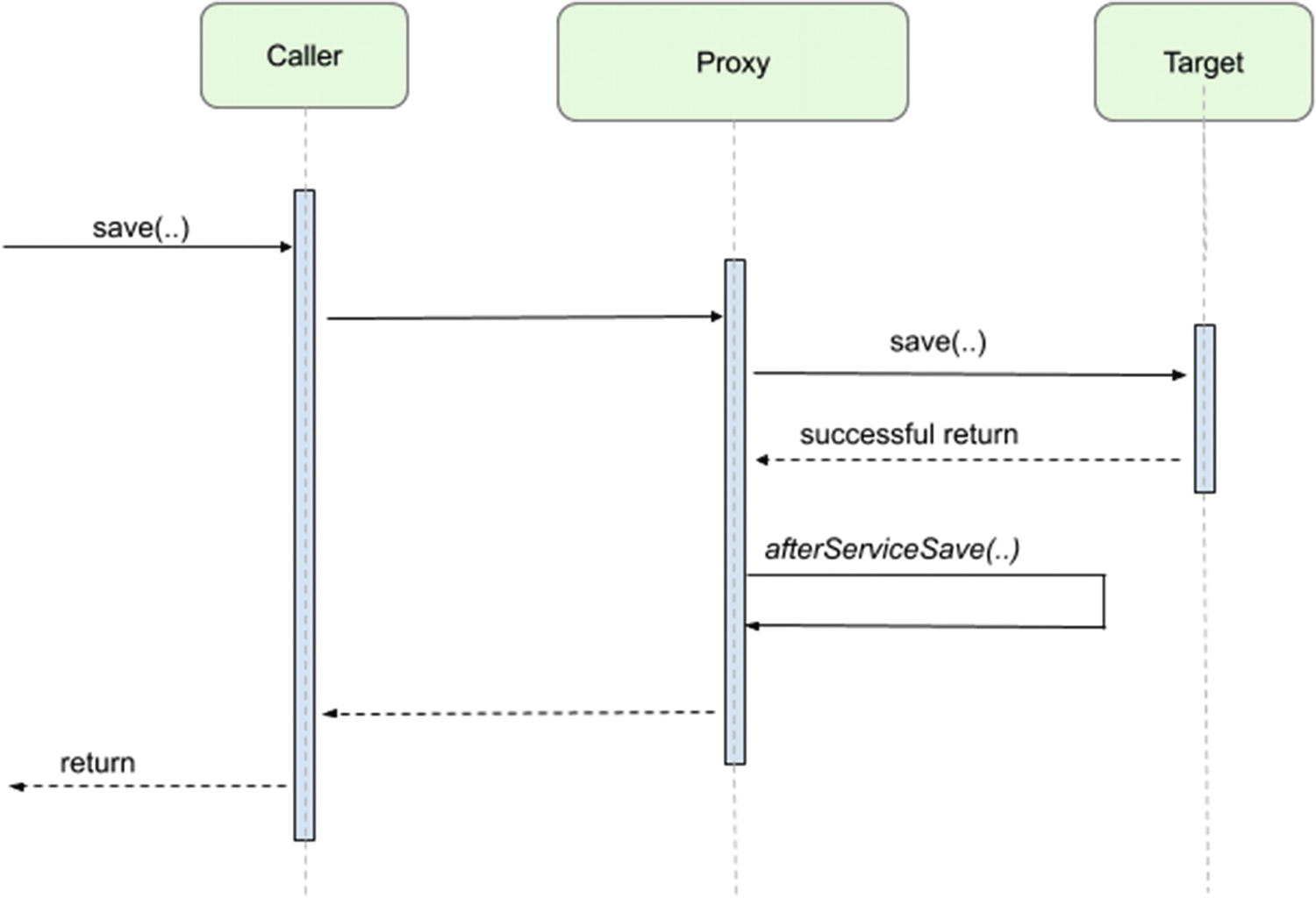
* 1. After Throwing - Exception management (After exception thrown during execution of method)



* 1. After - Logging (Every time after method executes successfully or not)



* 1. Around (Controls the execution of method as well as the return type and exception)



Need to explicitly add aspectjweaver and aspectjrt (runtime) dependencies in Spring to enable AOP

**Spring Data**

Every DB provides its own database driver, which is used to communicate with DB and from which DB connection is obtained

DataSource provides and manages the connection

Every DB driver has its own implementation of DataSource, which is used to communicate with the DB

By default JDBC considers each statement as transaction, and the committed after every transaction

JDBC always throws checked exception SQLException

**JdbcTemplate**

queryForObject - single row

RowMapper - Used to transform database record to domain object (applicable when mapping each row to single entity)

RowCallBackHandler - When no value should be returned

ResultSetExtractor - Transform record from joined tables in to single object (for nested entities, mapping row from multiple tables)

**NamedParameterJdbcTemplate** internally uses JdbcTemplate, named parameters along with their value is passed as Map

update() - Used to INSERT, DELETE, UPDATE DB records, returns no of rows affected

execute() - Used for DDL statements

In Spring data access exceptions are unchecked (extend RuntimeException) which are implementations of DataAccessExceptions

Exceptions are categorized under 3 types

* 1. Non-transient / Non recoverable Retry will not succeed, No record present
  2. Transient - Retry will succeed , Query time out
  3. Recoverable - Can be recovered after corrective step , DB connection timeout

@EnableTransactionManagement - to enable @Transactional behaviour,

in xml <tx:annotation-driven transaction-manager="transactionManager"/>

@Primary - used to identify the default dependency in case of multiple autowired dependencies

Transactions are rolled back in case of RunTimeExceptions

@Transactional can be declared at class level and this can be overridden by declaring @Transactional at method level

* 1. Transactional behaviour is a cross-cutting concern, and the declarative transactional model is supported in Spring using AOP. As expected, interface-based proxies are created, unless specified otherwise by customizing @EnableTransactionManagement annotation. This annotation declares a boolean property named proxyTargetClass with the default value of false. The default behaviour is not to create class-based proxies. This can be changed by using @EnableTransactionManagement(proxyTargetClass = true) to annotate the configuration class.
  2. Thus, you should be very careful where you place the @Transactional annotation. The recommendation is to apply the @Transactional annotation only to methods with public visibility, this way, regardless of the type of proxy created, you’ll always get the transactional behaviour where you expect it. Also, keep in mind local calls within the same class cannot be intercepted.
  3. The @Transactional annotation can be used at the class level. In this case, all the public methods inherit the transactional behaviour defined by the annotation on the class, but @Transactional annotations used at the method level, can override any transactional settings inherited from the class. Therefore, the most derived location takes precedence when evaluating the transactional settings for a method.
  4. It is recommended and practical to annotate only concrete classes (and methods of concrete classes) with the @Transactional annotation.
  5. The @Transactional annotation can be used on an interface (or an interface method), this requires you to use interface-based proxies; otherwise, if class-based proxies are used, the annotations are not inherited from the interface.
  6. @Transactional can also be used on abstract classes as well, and in this case, whether the transactional behaviour is applied depends on the type of proxy created.

Spring Cloud

Eureka - Service registry which coordinates communication with the microservices, each microservice registers with the Eureka server

@EnableEurekaServer - To activate eureka server configuration

By default spring boot searches for "application" file in resources, to change the name of resource use spring.config.name

@EnableEurekaClient - To enable the client to connect to Eureka server

Eureka server attributes

* 1. Port
  2. Hostname
  3. Wait time - Time for registry to fill up no answers to clients

Eureka client attributes

* 1. Heartbeat interval - default 30s
  2. Register with eureka - default true, this should be explicitly set as false for eureka server to prevent it from self-registering
  3. Eureka server URL
  4. Fetch registry - required if service needs to communicate with other services as well

Centralized configuration

Spring cloud config provides client and server support for externalized configurations

Spring cloud config client request for configuration from cloud config server, the cloud config server in turn fetches the required configuration file from the config repository

@EnableConfigServer

Configurations are loaded on starting the application , in order to refresh configurations without app restart actuator dependencies needs to be added

@RefreshScope is defined on the bean whose configuration value will be changed at runtime, on receiving RefreshScopeRefreshedEvent Spring will recreate the bean with new configuration

**Isolation levels**

Say you have a table T with a column C with one row in it, say it has the value '1'. And consider you have a simple task like the following:

BEGIN TRANSACTION;  
SELECT \* FROM T;  
WAITFOR DELAY '00:01:00'  
SELECT \* FROM T;  
COMMIT;

That is a simple task that issue two reads from table T, with a delay of 1 minute between them.

under READ COMMITTED, the second SELECT may return any data. A concurrent transaction may update the record, delete it, insert new records. The second select will always see the new data.

under REPEATABLE READ the second SELECT is guaranteed to display at least the rows that were returned from the first SELECT unchanged. New rows may be added by a concurrent transaction in that one minute, but the existing rows cannot be deleted nor changed.

under SERIALIZABLE reads the second select is guaranteed to see exactly the same rows as the first. No row can change, nor deleted, nor new rows could be inserted by a concurrent transaction.

If you follow the logic above you can quickly realize that SERIALIZABLE transactions, while they may make life easy for you, are always completely blocking every possible concurrent operation, since they require that nobody can modify, delete nor insert any row.

And finally, there is also the SNAPSHOT isolation level. SNAPSHOT isolation level makes the same guarantees as serializable, but not by requiring that no concurrent transaction can modify the data. Instead, it forces every reader to see its own version of the world (it's own 'snapshot'). This makes it very easy to program against as well as very scalable as it does not block concurrent updates. However, that benefit comes with a price: extra server resource consumption

**Propagation levels**

* 1. NEVER - Will throw exception if transaction exists
  2. REQUIRED - Will use existing transaction but creates new one if no existing transaction
  3. REQUIRES\_NEW - Suspends existing transaction and create new transaction
  4. NOT\_SUPPORTED - Suspends existing transaction and executes non transactionally
  5. SUPPORTS - Use existing transaction but executes non transactionally if no existing transaction
  6. NESTED - Creates a nested transaction (Savepoints)
  7. MANDATORY - Requires an active transaction

Transaction management

Declarative

* 1. Using @Transactional
  2. Using AOP

Programmatic

* 1. TransactionTemplate
  2. Defining TransactionManager

Global transactions - Transactions that span multiple resources (distributed)

MappingSqlQuery<T> specialization of JdbcTemplate.queryForObject() with mapRow() method

SQLUpdate specialization of JdbcTemplate.update()

**Hibernate**

Uses HibernateTransactionManger for transactional support

Creates SessionFactory from datasource, hibernate properties and entity packages to be scanned

@Version - Used for optimistic locking

While doing an update hibernate checks the version of the record in DB with version it is updating if same then the record is update and version incremented

If no then it raises HibernateOptimisticLockingFailureException

Spring data JPA

Provides repository classes to simplify data access logic

CrudRepository and JpaRepository

@EnableJpaRepository to enable repository support

**AOP terms**

**AOP :** Tool for managing cross cutting concerns

**Crosscutting concerns :** Part of application code that cannot be separated from application code and which may lead to duplicate code and tight coupling like logging and security

**Joinpoints** - Points in the application code where additional logic can be inserted using AOP

**Advice** - Code that is executed before or after join point

**Pointcuts** - Expression representing join points

**Aspect** - Combination of pointcuts and advice encapsulated in a class, defines what logic should be included in app and where it should be executed, containing the advice

**Weaving** - Process of inserting aspects in the app at appropriate point

**Target** - Object whose flow is modified by AOP process

**Introduction** - Modify the structure of object without having to define additional fields and methods

Spring uses proxies for AOP while AspectJ uses weaving to advice target objects

Based on when weaving happens AOP can be static or dynamic

Static AOP - AspectJ compile time , aspects are compiled to bytecode, not flexible as requires recompilation

Dynamic AOP - Spring uses proxy, performance is less compared to static

MethodInterceptor is example of around advice

**Final classes cannot be advised**

**Spring supports only one type of joinpoint method invocation**

**When target object implements interface Spring uses JDK dynamic proxy for proxy creation and CGLIB proxy if target object is a concrete class**

**In spring Aspect is represented by Advisor**

Advisor

|

|-----------------------|

PointcutAdvisor IntroductionAdvisor

|

|------------------------|

DefaultPointcutAdvisor

ProxyFactory -> DefaultAopProxyFactory -> Cglib2AopProxy/JdkDynamicAopProxy

addAdvice() - applies to all the methods in a class

Types of advices

MethodBeforeAdvice: Before execution of the method, no control over execution of method, has access to arguments of the method, if exception occurs further exception is aborted

Example is security

AfterReturningAdvice: After execution of the method, has access to arguments and return value, no control on the execution of the method, if exception is thrown during method execution advice execution is aborted

Example is additional error checking if method return invalid value

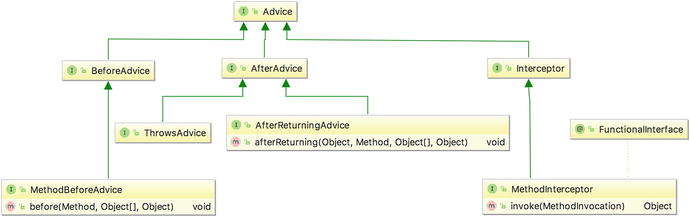
AfterAdvice: After execution of method but guaranteed to be run

ThrowsAdvice: If exception is thrown from the method

Example Centralized error logging

MethodInterceptor: Before and after method execution, can control method execution

Example is profiling



*From <*[*https://learning.oreilly.com/library/view/pro-spring-5/9781484228081/A315511\_5\_En\_5\_Chapter.html*](https://learning.oreilly.com/library/view/pro-spring-5/9781484228081/A315511_5_En_5_Chapter.html)*>*

Method before advice - can change value of the arguments and prevent method from executing but throwing exception

After returning advice - not possible to change return values or prevent method from executing but can throw exceptions

Around advice - possible to modify return values and prevent method from executing

Throws advice - similar to after returning advice, cannot prevent exception but can change the exception thrown

ThrowsAdvice is marker interface

Pointcuts

Advisor is spring implementation name for aspect, which is combination of advice and pointcut

DefaultPointcutAdvisor

Default advisor when adding advice

StaticMethodMatcherPointcut - extend this class for creating static pointcut

DynamicMethodMatcherPointcut - extend this class for creating dynamic pointcut

For optimal performance perform

* Class checking in getClassFilter()
* Method checking in matches(Method, Class<?>)
* Argument checking in matches(Method, Class<?>, Object[])

Additional overhead is involved in dynamic pointcut

NameMatchMethodPointcut is subclass of StaticMethodMatcherPointcut

JdkRegexMethodPointcut - matching of method name based on regular expression

AspectJExpressionPointcut

AnnotationmathcingPointcut - advice applied on method annotated

**JDK dynamic proxy and CGLIB proxy**

Jdk proxy implements the interface implemented by the target bean

All methods calls are intercepted by JVM and routed to invoke() method which does the determination if method is to be advised

Advised and non-advised method are called by reflection

CGLIB proxy extends the target bean

When CGLIB proxy is created how the methods should be handled are specified by spring once instead of calling invoke() method in JDK proxy

It generates bytecode for new class per proxy

CGLIB in frozen mode(advice chain is not altered) has best performance

**CGLIB can be used to proxy both classes and interfaces**

For proxying interface setOptimize() to true in ProxyFactory

ControlFlowPointcut

Specifies the calling method and class from which all called methods will be advised

ComposablePointCut

Compose 1 or more pointcuts

Pointcuts - compose 2 pointcuts for more than 2 use ComposablePointcut

Introduction

Special type of around advice applicable @ class level so cannot be used in pointcut

Used to check object modification in case of database persists