# Spring and Spring Boot

# What is Spring and Spring Boot?

## Spring Framework

The Spring Framework is pretty much a framework like Django (but a little different) that gives you the tools to build a Java applications quickly and conveniently. There are many aspects that the Spring Framework provide and we will go through couple of them.

At it's heart the Spring Framework does dependency injection, along with MVC, RPC, database access, and aspect-oritented programming.

### What is dependency

Dependency is something that your class requires in order to function properly.

Assume that you are currently writing a Java class that let you access a user table in your database. You call these classes DAO (data access object, they let you access to data in database duh). So you write a class called UserDAO class.

```
public class UserDAO {
    public User findById(Integer id) {
        // sql query to find user
    }
}
```

In order to execute the sql query that you have in `findById` method, you would need a database connection. In Java you usually get that database connection from another class, called a DataSource. So now after importing the DataSource class your code to execute the proper sql query looks something like this.

```
import javax.sql.DataSource;


public class UserDao {


    public User findById(Integer id) throws SQLException {
        try (Connection connection = dataSource.getConnection()) { // (1)
```

```
            PreparedStatement selectStatement = connection.prepareStatement("select * from users where id =
?");
            // use the connection etc.
        }
    }
}
```

Now how do we get a valid DataSource object that the code is currently using `dataSource`? The `dataSource` object that UserDAO is currently using is a dependency it requires, it obviously needs a valid DataSource object in order to execute the sql queries.

Natively, we can just construct a new DataSource object using `new` on the spot whenever the method is invoked like such:

```
import com.mysql.cj.jdbc.MysqlDataSource;

public class UserDao {

    public User findById(Integer id) {
        MysqlDataSource dataSource = new MysqlDataSource(); // (1)
        dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");
        dataSource.setUser("root");
        dataSource.setPassword("s3cr3t");

        try (Connection connection = dataSource.getConnection()) { // (2)
            PreparedStatement selectStatement = connection.prepareStatement("select * from users where id =
?");
            // execute the statement..convert the raw jdbc resultset to a user
            return user;
        }
    }
}
```

But constructing a new DataSource everytime the method is invoked is costly, because underneath there is the cost of setting up the sockets and everything. Now also think about if you have more variants of the methods. `findByName, findByFirstName, findByLastName`. Every one of those method will then need to initiate a database connection before it can send the sql queries. It is just not efficient and repetitive coding.

The next step is you would probably set up another method that is dedicated to returning a DataSource, so you modulated the DataSource to a method. Then all the DAO methods that needs to run sql queries will just call this particular function before it runs the query like such:

```
import com.mysql.cj.jdbc.MysqlDataSource;

public class UserDao {

    public User findById(Integer id) {
        try (Connection connection = newDataSource().getConnection()) { // (1)
            PreparedStatement selectStatement = connection.prepareStatement("select * from users where id =
?");
            // TODO execute the select , handle exceptions, return the user
        }
    }

    public User findByFirstName(String firstName) {
        try (Connection connection = newDataSource().getConnection()) { // (2)
            PreparedStatement selectStatement = connection.prepareStatement("select * from users where
first_name =  ?");
            // TODO execute the select ,  handle exceptions, return the user
        }
    }

    public DataSource newDataSource() {
        MysqlDataSource dataSource = new MysqlDataSource(); // (3)
        dataSource.setUser("root");
        dataSource.setPassword("s3cr3t");
        dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");
        return dataSource;
    }
}
```

This is better but now think about what happens if you have another DAO object for other table in your database? The DataSource dependency only exists in UserDAO. Then you would have to again write the same method for say ProductDAO, then repetition comes back again. For every DAO object you have, you would have to write this method that gives you the DataSource dependency. In addition, there is the cost of opening up a socket for every DAO object, for every method inside the DAO, it will be expensive.

## Next approach global dependency class

The next step is to pull the dependency to another dedicated class of it's own. Make it a singleton so that every DAO will only be interacting with a single DataSource instance, without making a new one each time the method is invoked. It just need to interact with one that exist throughout the lifetime of the program.

```
import com.mysql.cj.jdbc.MysqlDataSource;

public enum Application {

    INSTANCE;

    private DataSource dataSource;

    public DataSource dataSource() {
        if (dataSource == null) {
            MysqlDataSource dataSource = new MysqlDataSource();
            dataSource.setUser("root");
            dataSource.setPassword("s3cr3t");
            dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");
            this.dataSource = dataSource;
        }
        return dataSource;
    }
}
```

```
import com.yourpackage.Application;

public class UserDao {
    public User findById(Integer id) {
        try (Connection connection = Application.INSTANCE.dataSource().getConnection()) { // (1)
            PreparedStatement selectStatement = connection.prepareStatement("select * from users where id =
?");
            // TODO execute the select etc.
        }
    }

    public User findByFirstName(String firstName) {
        try (Connection connection = Application.INSTANCE.dataSource().getConnection()) { // (2)
            PreparedStatement selectStatement = connection.prepareStatement("select * from users where
first_name =  ?");
            // TODO execute the select etc.
        }
    }
}
```

Now inside any DAO object you would just simply need to import the global DataSource class and use it.

**We can improve this even further by giving the programmer who are constructing the DAO object the responsibility to provide a valid DataSource object by making it as a constructor parameter. This is called inversion of control.**

> However, these steps are still not mitigating the fact that the programmer is still actively setting up the DataSource dependency manually. That is where the dependency injection comes in, we let the Spring Framework see that our DAO has a DataSource dependency and let it construct it and wire it automatically for us. Resulting in a working DataSource and thus a working UserDAO automatically.

## Dependency Injection via ApplicationContext

**What is an ApplicationContext? It is a class that has control over all your classes and can manage them, i.e. create instances with the necessary dependencies.**

How do we use ApplcationContext to give us a properly configured UserDAO instance? And conversely a proper DataSource object? Here is how:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import javax.sql.DataSource;

public class MyApplication {

    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(MyApplicationContextConfiguration); //
(1)

        UserDao userDao = ctx.getBean(UserDao.class); // (2)
        User user1 = userDao.findById(1);
        User user2 = userDao.findById(2);

        DataSource dataSource = ctx.getBean(DataSource.class); // (3)
        // etc ...
    }
}
```

1. This is where we are constructing the ApplicationContext, again it is a class that has the ability to give you back a properly configured class in this case UserDAO, with it's

dependency set. You have to pass it a ApplicationContextConfiguration class which contains methods to ACTUALLY construct the classes.
2. This is how you get the UserDAO that's configured from ApplicationContext
3. This is how you get the DataSource that's configured from ApplicationContext

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyApplicationContextConfiguration {  // (1)

    @Bean
    public DataSource dataSource() {  // (2)
        MysqlDataSource dataSource = new MysqlDataSource();
        dataSource.setUser("root");
        dataSource.setPassword("s3cr3t");
        dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");
        return dataSource;
    }

    @Bean
    public UserDao userDao() { // (3)
        return new UserDao(dataSource());
    }

}
```

1. This is the ApplicationContextConfiguration class which you pass into ApplicationContext constructor. It is annotated with the @Configuration annotation
2. There is a method that returns the DataSource
3. There is a method that returns a UserDAO that's properly configured with the DataSource set

**You can think of the ApplicationContextConfiguration as a factory that gives you a properly configured (dependency set) object. Which ApplicationContext uses to actually return you the item constructed from the factory.**

## Wait what does @Bean annotation do? What is a Spring bean?

Like mentioned previously, ApplciationContextConfiguration is like a factory and those methods annotated as @Bean are the factory methods. To be more precise, they can be methods or classes that are managed by the Spring container.

**Spring container: It is part of the core of the Spring Framework and is responsible for managing all of the beans (creating and destroying). It is also responsible for performing the dependency injection when it creates the beans.**

In the previous case there are two factory methods, one that creates the DataSource, and one that creates UserDAO.

In addition, we can limit the number of instances that are created from these @Bean (factory method) by adding Spring bean scopes. Here are the three main scopes:

- Singleton: i.e. All DAO share the same DataSource
- Prototype: i.e. All DAO get their own DataSource
- More complex scope: DAO get their own DataSource per HttpRequest, per HttpSession, or even per WebSocket

The singleton scope meaning there will only be one instance is the most often used one, so we will use that for DataSource. The code below by adding the @Scope("singleton") annotation, Spring will only ever construct one instance of DataSource.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyApplicationContextConfiguration {

   @Bean
   @Scope("singleton")
   // @Scope("prototype") etc.
   public DataSource dataSource() {
      MysqlDataSource dataSource = new MysqlDataSource();
      dataSource.setUser("root");
      dataSource.setPassword("s3cr3t");
      dataSource.setURL("jdbc:mysql://localhost:3306/myDatabase");
      return dataSource;
   }
   ...
}
```

## @ComponentScan annotation

In the ApplicationContextConfiguration, we had a @Bean that explicitly construct the UserDAO object, why do we need that? Why can't Spring figure it out? This is where @CompnentScan comes into play.

The @ComponentScan annotation will tell Spring to look at all Java classes in the same package as the current ApplicationContextConfiguration file and find all those methods that are Spring beans. How does it know it is a Spring bean by giving it a marker annotation called @Component.

So we can remove the userDAO method from the ApplicationContextConfiguration and just add @Component to our UserDAO class like such:

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;

+ @Component
public class UserDao {

    private DataSource dataSource;

    public UserDao(@Autowired DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Now Spring is able to tell that this particular class is a Spring bean.

- Spring will create it when you ask it to
- UserDAO has an @AutoWired constructor argument, meaning Spring will automatically inject the DataSource that is configured in ApplicationContextConfiguration class
- If there is no DataSource configured in any Spring configuration then you will receive a NoSuchBeanDefinition exception

## @Autowire needed?

In the newer Spring version, Spring will be smart enough to inject the dependencies without explicit @Autowired annotation but it doesn't hurt to make things more explicit.

## Field and setter injection

Spring doesn't necessarily have to inject the dependencies in the constructor, you can inject directly into fields, or into setters like such:

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;

@Component
public class UserDao {

    @Autowired
    private DataSource dataSource;

}
```

```
import javax.sql.DataSource;
import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;

@Component
public class UserDao {

    private DataSource dataSource;

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

}
```

Both injection style have the same outcome, the dependency of DataSource will be injected correctly when UserDAO is instantiated.

## Aspect-Oriented Programming

Dependency injection isn't the only thing that Spring provides. The ability to add additional features to your methods or you beans right before it is executed or after it has been executed for say house keeping, logging purposes, is referred to as aspect-oriented programming.

Doing this will help keep your core business logic code clean, and any additional code that are not part of the main logic are separated.

## Resource management

How would you get local resources from your applicaiton's classpath? Via HTTP or FTP? You would need to do some low level coding digging if you want to do it yourself, but the Spring resource abstraction got you covered.

```
import org.springframework.core.io.Resource;

public class MyApplication {

    public static void main(String[] args) {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(someConfigClass); // (1)

        Resource aClasspathTemplate = ctx.getResource("classpath:com/myapp/config.xml"); // (2)

        Resource aFileTemplate = ctx.getResource("file:///someDirectory/application.properties"); // (3)

        Resource anHttpTemplate = ctx.getResource("https://marcobehler.com/application.properties"); // (4)

        Resource depends = ctx.getResource("myhost.com/resource/path/myTemplate.txt"); // (5)

        Resource s3Resources = ctx.getResource("s3://myBucket/myFile.txt"); // (6)
    }
}
```

1. You will need ApplicationText to do the resource loading
2. You can then call getResource() on an applicationcontext with a string that starts with classpath:, Spring will look for a resource under your application's classpath.
3. You can look for files on your harddrive
4. You can look for files on the web
5. If you don't specify a prefix then depending on what ApplicationContext you have it will look for that resource accordingly
6. Doesn't work right out of the box but you can use additional library to make this work

The Resource object that you get back have several useful methods such as whether or not it `exists(), getFilename(), getFile(), getInputStream()`.

So that means you can even get a raw binary data stream from it.

## Spring's Environment

An environment consists of one or many `.properties` files.

This is like `dotenv` where you have some important properties like database username, password, email, configurations that you wouldn't hard code it into your code.

You would instead read it in as an environment variable, or in the case of Spring you store them into `.properties` files and then you can access them via `getEnvironment` and `getProperty` method on the ApplicationContext object.

In addition, an environment also have profiles, you can have "dev", "qa", or "production" profiles, that uses different property depending on the environment you are deploying to.

You can also inject properties into your beans via @Value annotation into fields just like how @Autowired inject dependencies.

### Other Spring Modules

There is a module for writing a reactive web application.

There is a testing framework that let you test do integration test.

# Annotation Summary

You use @Configuration to mark an Application Context Configuration class to say that there will be multiple @Bean methods in there.

A @Bean method is just a method that will be managed by Spring framework, they are a factory method that will be responsible for actually initializing the object. However, instead of letting the programmer doing the object initialization, Spring will do the initialization for you. You just have to write the constructor.

A @Component annotated class is one that @ComponentScan will search for and will be managed by Spring framework. They will also be treated as a @Bean method, a factory method. You can do dependency injection here by using @Autowired to have Spring framework do any dependency injection after the object has been created.

# Spring Boot

To understand Spring boot one needs to understand what business applications are. They are basically enterprise (company) developed apps that are used to improve operations of the business. They can increase productivity, give more functionality.

**They can be used by internal employees, suppliers, customers. So basically an applications that is designed to make operating the company easier, either customer facing or internal.**

# Spring boot application.properties

In a `application.properties` file you can use place holders to reference environment variables rather than hard coding the value. For example: Instead of writing your `application.properties` like so:

```
// application.properties
aws.region=east
```

Which is fine but then if you have sensitive data which you wouldn't want to commit to the GitHub repositories you would need to use place holders and store those secrets in your environment variables which Spring Boot can then substitute in when is parsing the .properties files.

```
// application.properties
db.secrets=${DB_SECRETS}
```

If you write this and store your `DB_SECRETS` as an environment variable, it will be substituted in when the Spring boot application is ran, then you can access it using `@Value` annotation in your application.

This is much safer compared to hard coding it into your .properties file.

> This is not a feature of Intellij IDEA, it is a native functionality by Spring Boot.

## Extra note

The placeholders within application.properties can refer to either of the following

1. System property
2. Environment variable
3. Another application.properties variable

It is very flexible.

In addition, if somehow for some reason all three of the same variable are assigned under system property, environment variable, and also another application.properties variable exist and you try to use it in another variable, the precedence that which of the three will be used are listed above.

For example, if you are using @Value("var") in your Spring-boot application and there is

```
-Dvar=system
env of var=environment
application.properties variable of var=appvar
```

Then "system" will be used, and if System property doesn't exist, then "environment" will be used, then finally "appvar" will be used last if the environment variable doesn't exist either.

# Spring boot command line arguments with Maven

## Running application with Maven

To run your application with Maven you just run the following command:

```
mvn spring-boot:run
```

### Specifying profiles

You can specify an active profile for you Spring boot application with the following command for mvn

```
mvn spring-boot:run -Dspring-boot.run.profiles=local,dev
```

### Why using -D with mvn don't do anything

The Spring boot application is executed in a forked process, thus setting the properties with the command-line using -D for example will not affect the application. That is if you're running the application using maven.

### How to set system property with maven

Like previously mentioned since the Spring boot application is executed in a forked process, if you set a system property using -D it will not be used in the actual application. To set a system property for the Spring boot application to use you will have to use

```
mvn spring-boot:run -Dspring-boot.run.jvmArguments="-Dproperty1=overridden"
```

This is also how you set debugging properties for when maven actually executes with Java.

### What if you want to run with java

If instead you opt to build the jar using maven and then run the Spring boot application with `java` command, then you can provide in the -D and it will be used appropriately within the application and not ignored due to forked process.

However, when you are building the jar file you must also provide in the -D system properties. The active profile that you want to build for will need to be provided as well under `-Dspring.profiles.active` maven CLI argument.

# Fun Annotation Notes

## @ComponentScan + @Component

During Spring initialization the class that is marked with @ComponentScan will have it's package and it's subpackage scanned for any class that's marked with @Component. If any @Component class is found then it will add them to the application context (Basically contains the beans that will be managed by Spring).

Spring resolves dependencies between Spring beans and injects beans into each other's fields or constructor in order to satisfy the dependency.

## @Component and @Bean

@Component is applied to an entire class marking it as an bean for the Spring to manage. i.e. it will do dependency injection on all of the dependency, and then where-ever this class marked as @Component is needed it will be injected.

@Bean is applied to a method. The method will return the dependency class object you specified, with the specified constructor configuration. When the corresponding bean is used, by the type or the name of the @Bean method explicitly, it will retrieve the dependency and use it. The calling of the method and storing it as a bean is done at the initialization step.

If you use @Bean, the name of the method will become the name of the bean.

You should use @Bean when you don't have access to 3rd-party library source code, but you would like to autowire component from it.

You can inject @Bean into @Component variables, and you can also inject @Component into @Bean in the parameter. Inter-injection is also allowed (which should be obvious, you have @Component that uses other @Component dependencies).

> If you have multiple bean either from Component or from @Bean methods, you might run into errors where Spring doesn't know which Bean to use, you can distinguish between each of them using the @Qualifier annotation to mark the bean that you would like to name and then use in @Autowired

## @Autowired / @Inject

@Autowired and @Inject can be used interchangeably. Spring created @Autowired, @Inject is part of Java but Spring supports it fully.

This is the annotation that is used to actually inject the dependency into the class. There are many styles of doing the injection, you can do:

- Field injection: Putting @Autowired on the private field of the class will inject that field for you
- Setter injection: Putting @Autowired on the setter for that particular field that you are injecting of the class
- Constructor injection: Putting @Autowired on the class's constructor for dependency injection

> For Spring 4.3+, if a class that is configured as a Spring bean or component, only have one constructor and constructor have the wanted auto injected field in the parameter, then you can skip out on the @Autowired annotation.

**Before Spring 4.3:**

```
@Component
public class ExampleDB {

    private ExampleRecord record;

    @Autowired
    public ExampleDB(ExampleRecord record) {
        this.record = record;
    }
}
```

**After:**

```
@Component
public class ExampleDB {

    private ExampleRecord record;

    public ExampleDB(ExampleRecord record) {
        this.record = record;
    }
}
```

As you can see this is more leegant, BUT, at the cost of context. If you don't specify it explicitly the programmer will not know at immediately that record is an injected dependency.

# See all Spring Managed Beans

```
public void run(String... args) throws Exception {
    String[] beans = ctx.getBeanDefinitionNames();
        Arrays.sort(beans);
        for (String bean : beans) {
            System.out.println(bean);
        }
}
```

This will print out all of the registered beans.

For @Components without name arguments, the bean name will be the class name but camelcased. ExampleDB will be exampleDB for bean name.

For @Beans it is the same case as well, the return type will be the bean name but camelcased. Unless you specify the name argument for the @Bean annotation.

When you @Autowired, if there are multiple @Bean of the same type, you can disambiguate by using the @Qualifier annotation to specify exactly which bean you would like to use according to the name.

If you have multiple bean with the same bean name, it will be an error and you cannot run your application.

# @Autowired, @Resource, @Inject

They all do the same stuff, @Autowired is by Spring. Other two is by Java.

https://www.baeldung.com/spring-annotations-resource-inject-autowire

# @Autowiring an interface

If you have multiple implementation of an interface and are all marked as beans. To specify which implementation to @Autowire you would need to add the @Qualifier to specify exactly which bean it is to use.

Funny enough in the latest Spring framework, I observe that if the variable name matches one of the bean name then it will @Autowire to that bean without the need to specify the bean name.

# Spring Injection Order

## First of all don't mix and match injection for your fields

If you are going to let Spring do your dependency injections for you, just stick to one method. Constructor, setter, or field injections.

However, if you're curious on the order of the evaluation here it is

1. It will call the constructor and inject the necessary dependency, since a constructor is needed before anything happens
2. It will then inject fields annotated with @Autowired, which are the field injections
3. Finally it will call methods annotated with @Autowired, which are the setter injections

So in conclusion, setter injection takes precedence over all the other methods. Again don't mix and match the type of injections that you are using, it is bad practice and is confusing.

# Aspect Oriented Programming

## What the heck is AOP?

Aspect oriented programming is a programming technique just like object oriented programming where you group codes and software designs into objects rather than keeping it as separate functions and logic.

AOP aim to help mitigate and solve the problem of cross-cutting concerns. Cross-cutting concerns are any logic that are not part of your main business logic that are repeated thus you can extract them out into it's own module.

For example: Say you have a class, User that suppose to facilitate the CRUD operation between the databases for a user. You have methods that does the creation, update, read, and delete. Now your manager wants you to add some logging functionalities and/or authentication capability before the operation is carried out. You could hard code them in method by method if it isn't many. But imagine you have hundreds of these classes, hard coding them in those logics are just not going to cut it.

Besides, those logging functionality and authentication are NOT part of the main business logic. Your main business logic is just CRUD operations in this case. So these are cross-cutting concerns they are extra functionality that you are adding to mingle with the main business logic.

AOP aims to solve that problem by extracting those concerns (**either logging or authentication, really any extra logic or functionality**) out from your main business logic into something called an Aspect.

Then you can just use Aspect whenever you need them.

## Terminology

I'm going to now define some of the jargon terminology that's used in AOP :)

### JoinPoint

A joinpoint is just anywhere you can intercept / insert an Aspect (functionality such as logging, authentication) in. Methods being called, an exception beign thrown, or even a field being

modified. Basically where you can insert the extracted concerns into in your application for new behaviors.

Concrete examples would be:

```
class Employee{
    public String getName(int id){....}
    private int getID(String name){...}
}
```

Here getName and getID would be considereed JoinPoint because they are candidate methods for where you can insert aspects, either before execution, after execution, after return of the method, or after an exception is thrown from the method.

## Pointcut

So joinpoint is basically where you can insert the aspects, pointcut defines at which particular joinpoint you would like to invoke the aspect with the associated advice.

Say if you define something like below:

```
@Before("execution(* *.*.get*())")
public void beforeGets() {
  System.out.println("I'm getting");
}
```

First line is a pointcut along with the advice on when the logic of the aspect should be applied. The pointcut being on all get* method invocations, advice being that the logic should be carried out before the execution of the pointcuts.

So whenever you call say `getId`, it will print "I'm getting" before the actual `getId` method is called.

> You can think of pointcuts being a regex expression to match what you would like the aspect to be applied on.
> Advice being when you would like to take the action (logic of the aspect) when you have a pointcut.

## Advice

Used together with pointcuts to define WHAT logic you would like to execute and WHEN you would like to execute on the pointcut (matched places).

There are different advice types:

1. @Before: Runs before the execution of the joinpoint method
2. @After: Runs after the joinpoint method finishes executing, normally or when throwing an exception
3. @AfterReturning: Runs after the method returned
4. @AfterThrowing: Runs after the method throws an exception
5. @Around: Most powerful advice. You can choose whether or not to execute the jointpoint method or not. Write advice code that executes both before and after the execution.

# Big caviar

If your method that's being intercepted by Spring Boot's Aspect, and you call another method within it and expect it to be also be intercepted, it will NOT.

Aspect's method interception only works on method that's invoked ACROSS class. If you invoke method within the same class, it will not trigger the interception. Meaning those are not valid Pointcut!

https://stackoverflow.com/questions/27635310/spring-aspect-not-working-when-called-by-method-of-the-same-class