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

```java
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.

```java
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:

```java
import javax.sql.DataSource;

import org.springframework.stereotype.Component;

import org.springframework.beans.factory.annotation.Autowired;


@Component

public class UserDao {


    @Autowired

    private DataSource dataSource;


}
```

```java
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.

```java
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.