Spring Framework

1

Why should we use Frameworks?

DEV SANJEEV

Uses best readily available best frameworks like Spring, Angular etc. to build a web app

- Leverage Security, Logging etc. from frameworks
- Can easily scale his application
- App will work in an predictable manner
- Focus more on the business logic
- Less efforts and more results/revenue

DEV VICKY

Build his own code by himself to build a web app

- Need to build code for Security, Logging etc.
- Scaling his is not an option till he test everything
- App may not work in an predictable manner
- Focus more on the supporting components
- More efforts and less results/revenue

2

What is Spring?

Well, It's a Bit Confusing …

"Spring" could mean …

- The Spring Framework
- Spring Boot
- Spring Data
- Spring Cloud
- Spring Batch
- And more!

3

What is Spring?

"Most often, when people say **'Spring'**, they mean the entire family of projects."

Spring Framework Reference Documentation

4

What is Spring?

- Spring Framework is an application container for Java that supplies many useful features, such as **Inversion of Control, Dependency Injection**, abstract data access, transaction management, and more.
- Compliments JEE by making them easier to use.
  - It was conceived in 2002 in response to industry complaints that the Java EE specification was sorely lacking and very difficult to use.
  - BUT, it is by no means strictly restricted to an EE environment!
- Spring Framework is not a requirement to develop rich Java web applications.
  - We can create full-featured web applications without Spring.
  - But, Spring makes developing applications of all types rapidly with modular concepts and testable code easier.
    - Build with best practices (Singletons, Factories, etc…)
  - When used properly, Spring Framework is the most powerful tool in your Java development toolbox

5

Spring - Business Focus

- Spring helps us focus on the **business logic**.
- *Example without /with Spring:*

```
public Car getById(String id) {
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;

    try {
        String sql = "select * from CAR where ID = ?";
        con = DriverManager.getConnection("localhost:3306/
cars");
        stmt = con.prepareStatement(sql);
        stmt.setString(1, id);
        rs = stmt.executeQuery();
        if(rs.next()) {
            Car car = new Car();
            car.setMake(rs.getString(1));
            return car;
        }
        else {
            return null;
        }
    } catch (SQLException e) { e.printStackTrace();}
    finally {
        try {
            if(rs != null && !rs.isClosed()) {
                rs.close();
            }
        } catch (Exception e) {}
    }
    return null;
}
```

```
public Car findCar(String id) {
    return getEntityManager().find(Car.class, id);
}
```

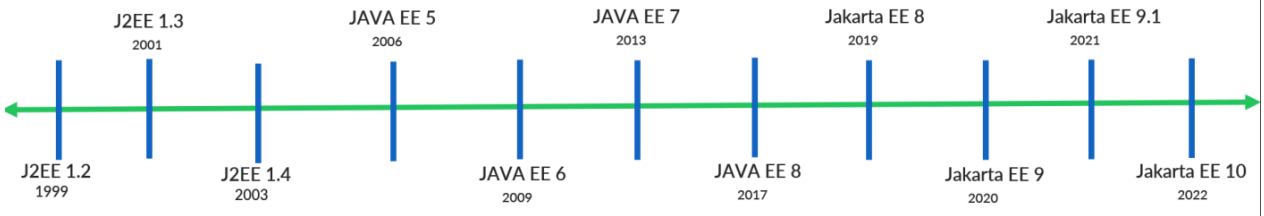
6

Spring popularity

- Spring is so popular that its main competitor quit the race when Oracle stopped the evolution of Java EE 8, and the community took over its maintenance via Jakarta EE
  - Java brand name is the property of Oracle!
- The main reason for Spring Framework's success is that it regularly introduces features/projects based on the latest market trends
  - E.g. Spring Boot
- Spring is open source. It has a large and active community of developers that continuously provide feedback.
- **Spring can't exist without JavaEE.**

7

Java EE Releases



- *Java/Jakarta Enterprise Edition (EE) contains Servlets, JSPs, EJB, JMS, RMI, JPA, JSF, JAXB, JAX-WS, Web Sockets etc.*
- *Components of Java/Jakarta Enterprise Edition (EE) like EJB, Servlets are complex in nature due to which everyone adapted Spring framework for web applications development.*
- *Java EE quit the race against the Spring framework, when Oracle stopped the evolution of Java EE 8, and the community took over its maintenance via Jakarta EE.*
- *Since Oracle owns the trademark for the name "Java", Java EE renamed to Jakarta EE. All the packages are updated with javax.* to jakarta.* namespace change.*

8

Spring Releases

- *The first version of Spring was written by Rod Johnson, who released the framework with the publication of his book Expert One-on-One J2EE Design and Development in October 2002*
- *Spring came into being in 2003 as a response to the complexity of the early J2EE specifications. While some consider Java EE and Spring to be in competition, Spring is, in fact, complementary to Java EE. The Spring programming model does not embrace the Java EE platform specification; rather, it integrates with carefully selected individual specifications from the EE umbrella*
- *Spring continues to innovate and to evolve. Beyond the Spring Framework, there are other projects, such as Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, among others.*

9

Dependency Inversion Principle

10

Topics covered

- **What is Dependency Inversion Principle**
- **Writing code that respects this principle**
- **Dependency Injection (DI)**
- **Inversion of Control (IoC)**

11

Dependency Inversion Principle

1. High level modules should not depend on low level modules; both should depend on abstractions.

2. Abstractions should not depend on details. Details should depend upon abstraction.

12

High Level Modules

Modules written to solve real problems and use cases

They are more abstract and map to the business domain

What the software should do

13

Examples of Low Level Modules

14

Low Level Modules

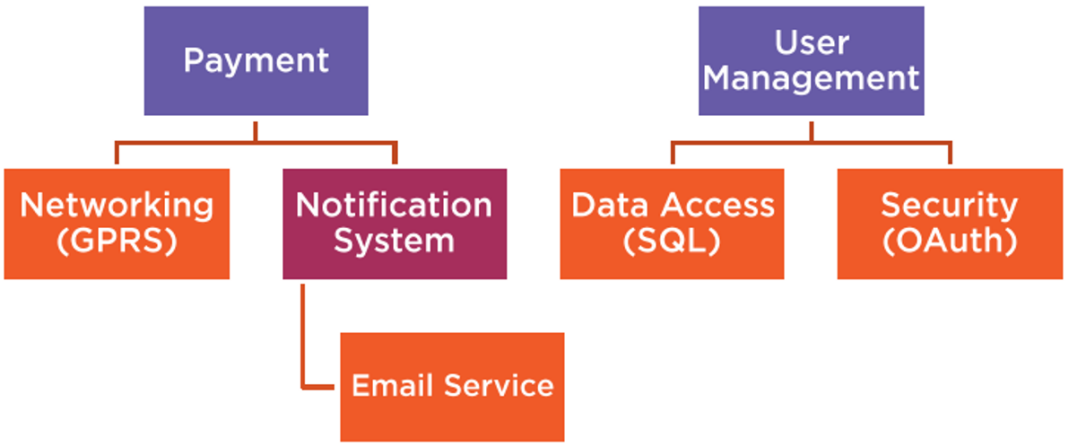
Contain implementation details that are required to execute the business policies

They are considered the "plumbing" or "internals" of an application

How the software should do various tasks

15

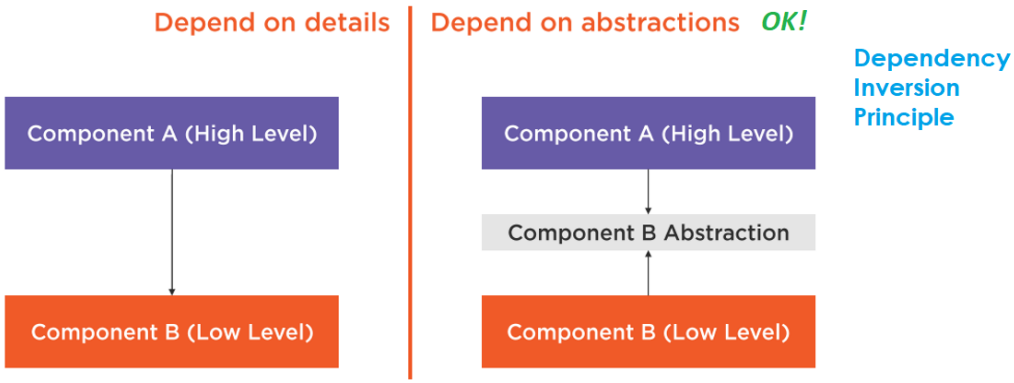
High Level Modules Work Together with Low Level Modules



16

Abstraction

- Something that is not concrete
- Something that you can not "new" up.
  - In Java we use Interfaces and Abstract classes.



17

Example

Low Level Class

```
class SqlProductRepo{
    public Product getById(String productId){
        // Grab product from SQL database
    }
}
```

High Level Class

```
class PaymentProcessor{
    public void pay(String productId){
        SqlProductRepo repo = new SqlProductRepo();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
```

The PaymentProcessor has a direct dependency with the SqlProductRepo because it instantiates this repo.
This clearly violates the dependency inversion principle!

18

Example…

Product Repository Abstraction

```
interface ProductRepo {
    Product getById(String productId);
}
```

Low Level Class Depends on Abstraction

```
class SqlProductRepo implements ProductRepo{
  @Override
  public Product getById(String productId){
     // Concrete details for fetching a product
  }
```

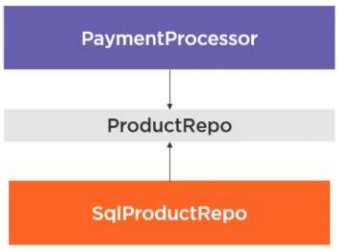
Product Repository Factory

```
class ProductRepoFactory{
    public static ProductRepo create(){
        return new SqlProductRepo();
    }
}
```

High Level Class Depends on Abstraction

```
class PaymentProcessor{
    public void pay(String productId){
        ProductRepo repo = ProductRepoFactory.create();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}
```

After Applying the DIP



19

Example (Factory Method)

Product Repository Factory

```
class ProductRepoFactory{
    public static ProductRepo create(String type){
        if(type.equals("mongo")){
            return new MongoProductRepo();
        }
        return new SqlProductRepo();
    }
}
```

We can modify this factory and make it more smart.
For example, we can modify the create method by passing in a type argument.
Now we can return varies instances of ProductRepo based on that type.
For example, if the type equals Mongo, we can return a MongoProductRepo.
Else, we can return a SqlProductRepo. The consuming code (PayementProcessor) will not care what particular instance is returned.

20

Example (Still Some Coupling)

```
class PaymentProcessor{
    public void pay(String productId){
        ProductRepo repo = ProductRepoFactory.create();
        Product product = repo.getById(productId);
        this.processPayment(product);
    }
}
```

Although we have eliminated the coupling with the concrete SqlProductRepo class, we still have a small coupling with the ProductRepoFactory.

21

Dependency Injection

- A technique that allows the creation of dependent objects outside of a class and provides those objects to a class!
- **Dependency Injection** is used in conjunction with the **Dependency Inversion** principle!
  - BUT, they are not the same thing

22

Example…

Declaring Dependencies in Constructor

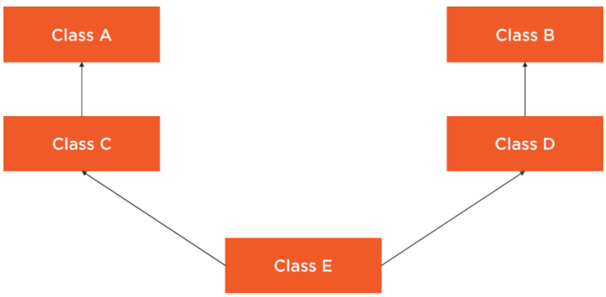
```
class PaymentProcessor{
    public PaymentProcessor(ProductRepo repo){
        this.repo = repo;
    }
    public void pay(String productId){
        Product product = this.repo.getById(productId);
        this.processPayment(product);
    }
}
```

Passing Dependencies

E.g. Main method

```
ProductRepo repo = ProductRepoFactory.create();
PaymentProcessor paymentProc = new PaymentProcessor(repo);
paymentProc.pay("123");
```


More Complex Dependencies



Handling Dependencies Becomes Harder

```
A a = new A();
B b = new B();
C c = new C(a);
D d = new D(b);
E e = new E(c,d);

e.doSomething();
```

Class C depends on class A. Class D has a dependency on class B. And class E has a dependency on both classes C and D.
If you want to call a method on class E, we also need to create all the dependencies that it needs.

- *We need to take care of all of that dependency handling.*
- *We need to create concrete instances of those classes in a particular order.*
- *We have to create classes A and B first because they have no dependencies, then classes C and D.*
- *And at the end, you can create class E and call a method on it.*

This is a pretty simple example with just five classes.

But, real projects have hundreds if not thousands of classes that can be instantiated!!!

Inversion of Control

- Is a **design principle** in which the control of object creation, configuration, and lifecycle is passed to a container or framework.

You don't "new" up objects

- They are created by someone else (IoC container)

Control of object creation is inverted

- It's not the programmer, but someone else who controls the objects

It makes sense to use it for some objects in an application (services, data access, controllers)

- For others (entities, value objects) it doesn't

25

IoC Container Benefits

- Makes it easy to switch between different implementations at runtime
- Increased program modularity
- Manages the lifecycle of objects and their configuration

- E.g. The Spring Framework has the IoC container
  - Objects used by the application are managed by the **Spring IoC container**.
  - These objects are called **Spring Beans**.
  - They are created with the configuration that you supply to the container!

26

Spring Bean Definition Example

```
@Configuration
public class DependencyConfig {
    @Bean public A a(){return new A();}

    @Bean public B b(){return new B();}

    @Bean public C c(A a, B b){return new C(a(),b());}
}
```

Class C has dependencies on class A and class B. We are using the @Bean annotation, and we are passing the A and B arguments to the constructor of type C.
Notice that we are not creating new types for A and B. We let the IoC container handle the creation of A and B for us!

```
public class C {
    private A a;
    private B b;
    public C(A a, B b){
        this.a = a;
        this.b = b;
    }
}
```

When we need an instance of this class, the Spring IoC container will look at the constructor.
It then extracts the dependencies that class needs, in this case A and B.
Because we declared A and B as beans in the configuration class, the Spring IoC container can iniect them!

27

Summary

Classes should depend on abstractions, not implementation details

DIP, DI and IoC work hand in hand to eliminate coupling and make applications less brittle

Testability can greatly be improved by using the DIP and the DI technique

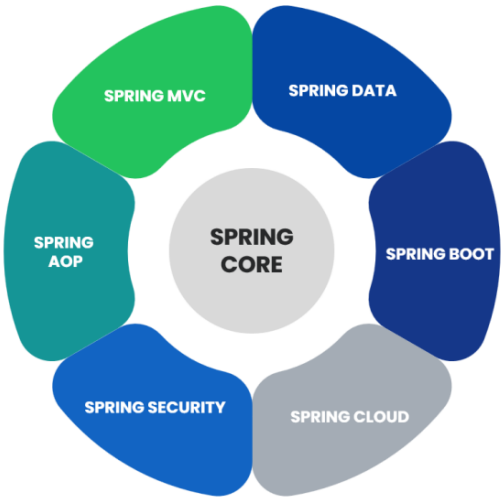
Take advantage of the powerful capabilities of the Spring IoC container

28

Spring

29

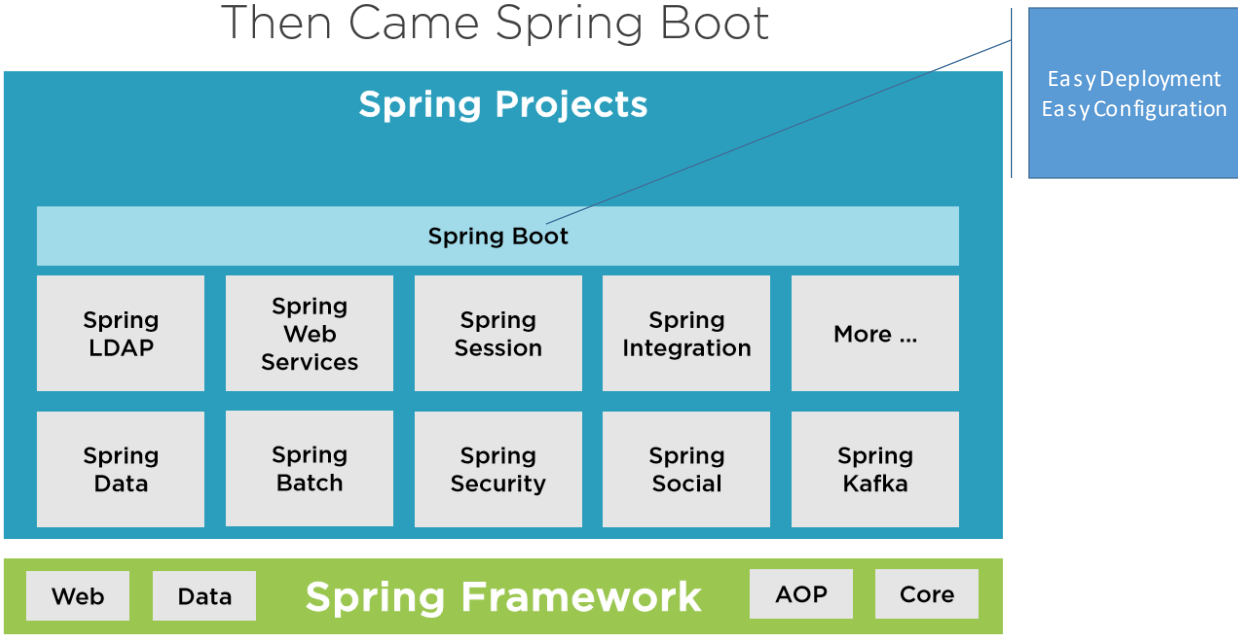
Spring Core



- Spring Core is the heart of entire Spring. It contains some base framework classes, principles and mechanisms.
- The entire Spring Framework and other projects of Spring are developed on top of the Spring Core.
- Spring Core contains following important components,
  - ✓ IoC (Inversion of Control)
  - ✓ DI (Dependency Injection)
  - ✓ Beans
  - ✓ Context
  - ✓ SpEL (Spring Expression Language)
  - ✓ IoC Container

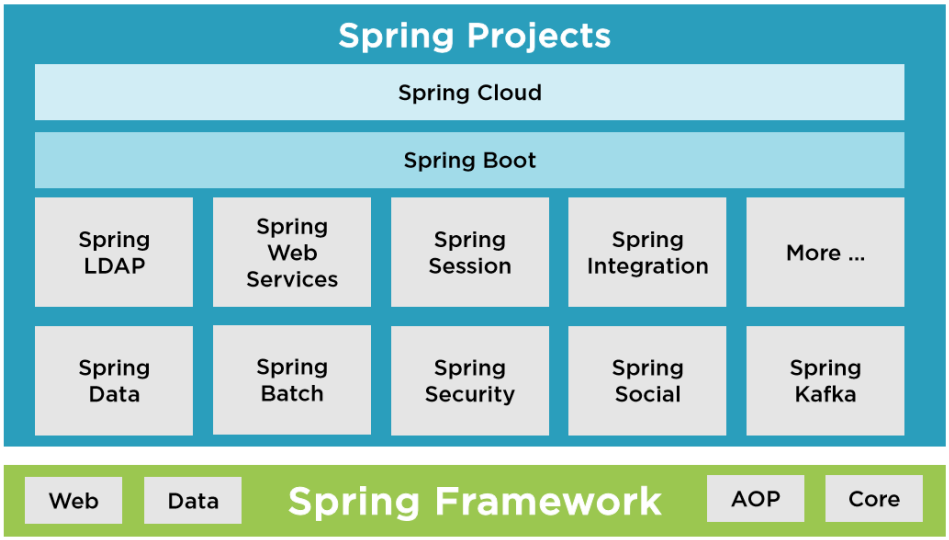
30

Then Came Spring Boot



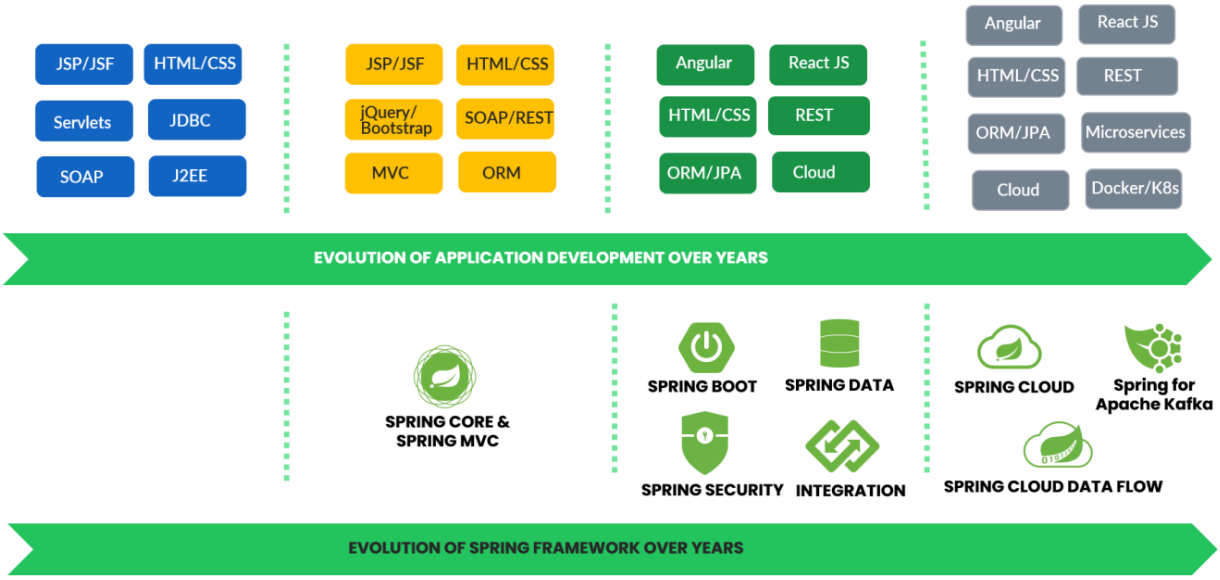
31

Then Came Spring Cloud



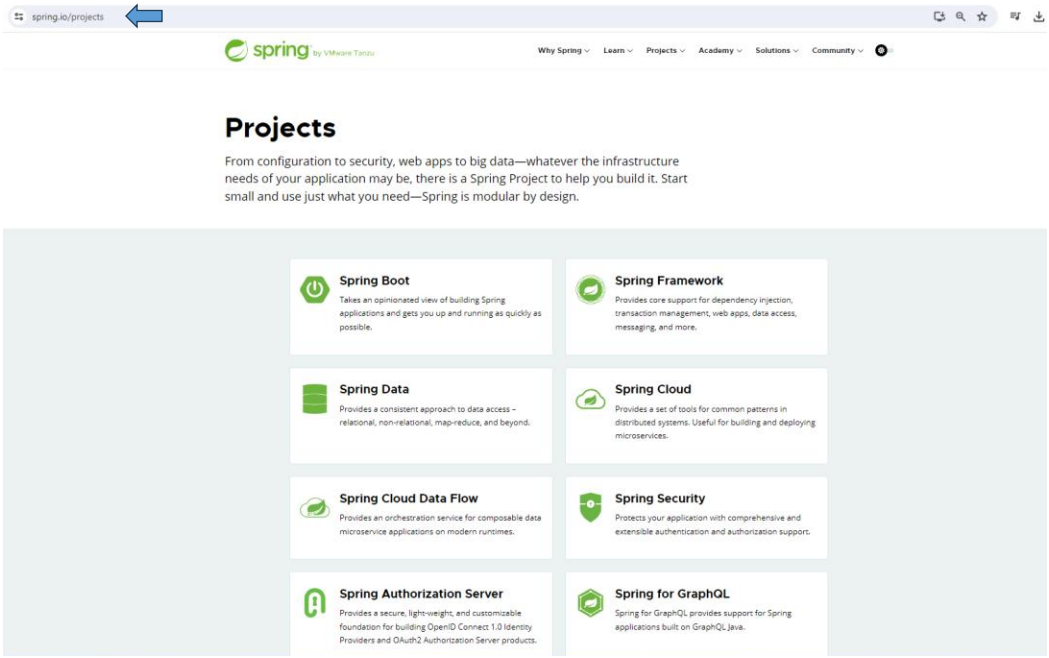
32

Spring Evolution



33

spring.io official Spring webpage



34

Spring

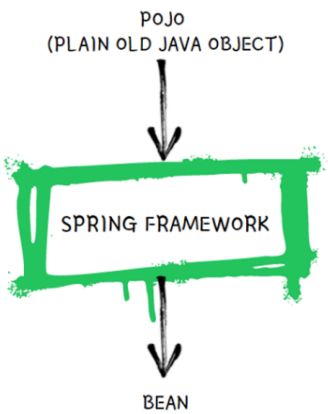
Flexible, modular, and backwards compatible

Large and active community

Continually innovates and evolves

35

Spring Beans, Context, SpEL



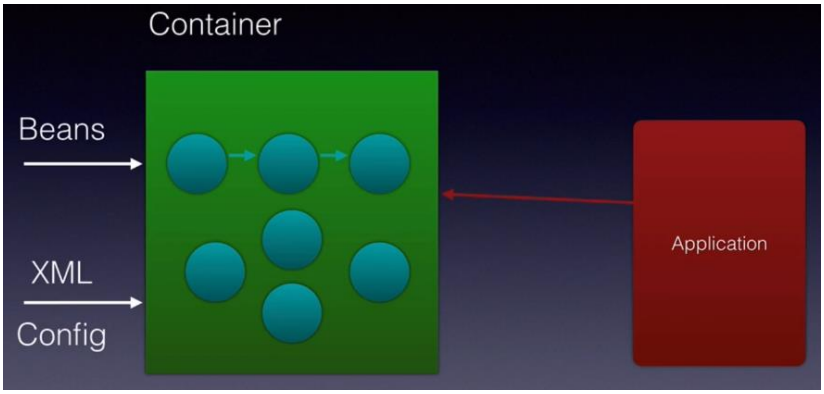
- Any normal Java class that is instantiated, assembled, and otherwise managed by a Spring IoC container is called Spring Bean.

  Not all Java classes inside our App should be managed by the Spring IoC container... e.g. the utility classes, etc.
- These beans are created with the configuration metadata that you supply to the container either in the form of XML configs and Annotations.
- Spring IoC Container manages the lifecycle of Spring Bean scope and injecting any required dependencies in the bean.
- Context is like a memory location of your app in which we add all the object instances that we want the framework to manage. By default, Spring doesn't know any of the objects you define in your application. To enable Spring to see your objects, you need to add them to the context.
- The SpEL provides a powerful expression language for querying and manipulating an object graph at runtime like setting and getting property values, property assignment, method invocation etc.

36

Spring IoC Container

- Container: is a component and is responsible for creating the objects, injecting them into other objects, and destroying them.
- It needs:
  - The Java Beans
  - XML Configuration/Annotations to tell the container what the dependencies are.



37

Spring IoC Container

Spring IoC Container

- The IoC container is responsible
  - ✓ to instantiate the application class
  - ✓ to configure the object
  - ✓ to assemble the dependencies between the objects

- There are two types of IoC containers. They are:
  - ✓ org.springframework.beans.factory.BeanFactory
  - ✓ org.springframework.context.ApplicationContext

BeanFactory: is a basic IoC container... it only handles the bean creation, bean auto-wiring and bean injection.
ApplicationContext: it is a more advanced IoC container (implements BeanFactory). It provides extra features... e.g. after creating a bean to execute some code etc.

- The Spring container uses dependency injection (DI) to manage the components/objects that make up an application.

38

ApplicationContext

- ApplicationContext is an interface that represents the Spring container. Spring provides different types of ApplicationContext containers suitable for different requirements.
- Some implementations of the ApplicationContext interface:
  - **AnnotationConfigApplicationContext** (no XML… the beans are configured using annotations like @Configuration, @Component)
  - **FileSystemXMLApplicationContext** (search for the XML config file on the file system)
  - **ClassPathXmlApplicationContext** (search for the XML config file on the Java classPath)
  - …
- A Spring application always has at least **one** application context.
  - It can also have a hierarchy of multiple application contexts.
    - This is useful for defining a set of shared application components while isolating other application components from each other.

39

Spring Configuration

- Using Java
- Using XML.
  - Why use XML?
    - First method available in Spring.
      - Not very popular today.
    - Some things are simpler with XML.
    - Clear separation of concerns.

40

ApplicationContext - examples

XML configuration

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {

    public static void main(String args[]) {
        ApplicationContext appContext = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");

        SpeakerService service = appContext.getBean( s: "speakerService", SpeakerService.class);

        System.out.println(service.findAll().get(0).getFirstName());
    }
}
```

Java configuration

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Application {

    public static void main(String args[]) {
        ApplicationContext appContext = new AnnotationConfigApplicationContext(AppConfig.class);

        SpeakerService service = appContext.getBean( s: "speakerService", SpeakerService.class);

        System.out.println(service.findAll().get(0).getFirstName());
    }
}
```

41

Spring Configuration – Using Java

- Create a Java class (with name e.g. **`AppConfig`**) to add all your configurations.
  E.g.:

```
@Configuration
public class AppConfig {

@Bean(name = "customerRepository")
  public CustomerRepository getCustomerRepository() {
    return new HibernateCustomerRepositoryImpl();
  }
}
```

What this method returns is an instance of a Bean.

@Configuration

applicationContext.xml replaced by @Configuration

@Configuration at class level — This annotation will make the IoC container scan all this class (during initialization) because there is some configuration here.

Spring Beans defined by @Bean

@Bean at method level

The name getCustomerRepository is not relevant -- any name can be used!

42

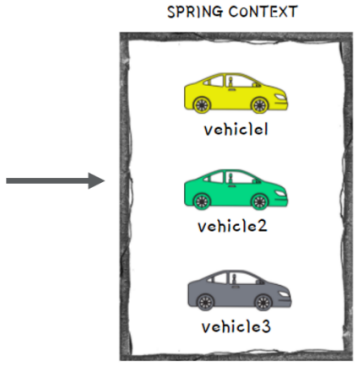
NoUniqueBeanDefinitionException

When we create multiple objects of same type and try to fetch the bean from context by type, then Spring cannot guess which instance you've declared you refer to. This will lead to NoUniqueBeanDefinitionException like shown below,

```
@Bean
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```




```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle veh = context.getBean(Vehicle.class);
```

Output on Console

```
Exception in thread "main" org.springframework.beans.factory.NoUniqueBeanDefinitionException Create breakpoint : No qualifying bean of type
'com.example.beans.Vehicle' available: expected single matching bean but found 3: vehicle1,vehicle2,vehicle3
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveNamedBean(DefaultListableBeanFactory.java:1262)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveBean(DefaultListableBeanFactory.java:494)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:349)
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.getBean(DefaultListableBeanFactory.java:342)
    at org.springframework.context.support.AbstractApplicationContext.getBean(AbstractApplicationContext.java:1172)
    at com.example.main.Example2.main(Example2.java:18)
```

43

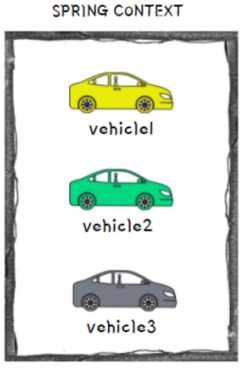
NoUniqueBeanDefinitionException - Solution

To avoid NoUniqueBeanDefinitionException in these kind of scenarios, we can fetch the bean from the context by mentioning it's name like shown below,

```
@Bean
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Bean
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```




```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle veh = context.getBean( name: "vehicle1",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh.getName());
```

Output on Console

```
Vehicle name from Spring Context is: Audi
```

44

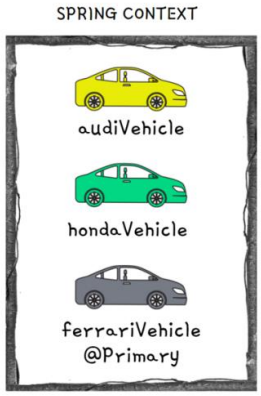
NoUniqueBeanDefinitionException – Solution (2)

When you have multiple beans of the same kind inside the Spring context, you can make one of them primary by using @Primary annotation. Primary bean is the one which Spring will choose if it has multiple options and you don't specify a name. In other words, it is the default bean that Spring Context will consider in case of confusion due to multiple beans present of same type.

```
@Bean(name="audiVehicle")
Vehicle vehicle1() {
    var veh = new Vehicle();
    veh.setName("Audi");
    return veh;
}

@Bean(value="hondaVehicle")
Vehicle vehicle2() {
    var veh = new Vehicle();
    veh.setName("Honda");
    return veh;
}

@Primary
@Bean("ferrariVehicle")
Vehicle vehicle3() {
    var veh = new Vehicle();
    veh.setName("Ferrari");
    return veh;
}
```




```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Primary Vehicle name from Spring Context is: " + vehicle.getName());
```

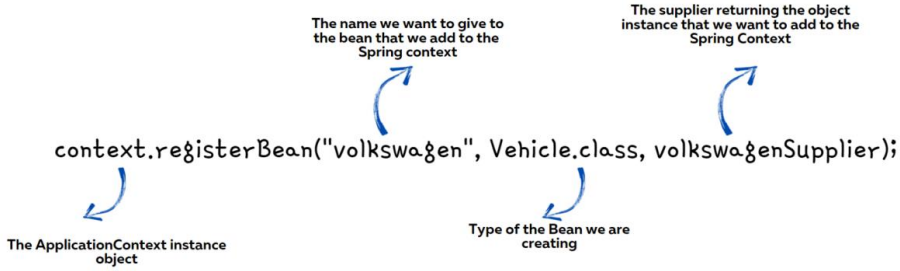
Output on Console

```
Primary Vehicle name from Spring Context is: Ferrari
```

45

Adding new Beans Programmatically

Sometimes we want to create new instances of an object and add them into the Spring context based on a programming condition. For the same, from Spring 5 version, a new approach is provided to create the beans programmatically by invoking the registerBean() method present inside the context object.



Example:

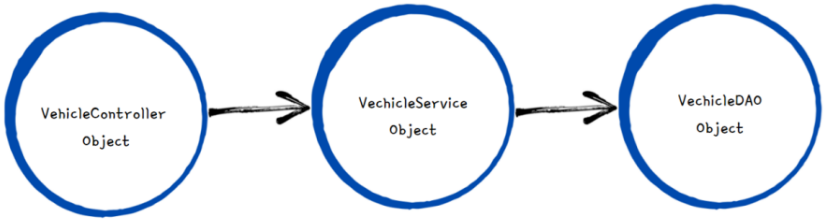
```
if((randomNumber% 2) == 0){
    context.registerBean( beanName: "volkswagen",
            Vehicle.class,volkswagenSupplier);
}else{
    context.registerBean( beanName: "audi",
            Vehicle.class,audiSupplier);
}
```

46

Introduction to Bean Wiring

Inside Java web applications, usually the objects delegate certain responsibilities to other objects. So in this scenarios, objects will have dependency on others.

In very similar lines when we create various beans using Spring, it our responsibility to understand the dependencies that beans have and wire them. This concept inside is called Wiring/Autowiring.



We don't write all the code inside a single component. Components depend on each other.
We need to handle the dependencies. *E.g. Controller -> Service -> DAO*

Spring will not scan all the POJO classes that are in the application!

47

Dependency Injection Types

- The two major flavors of the Dependency Injection (DI) are:
  - **Setter Injection**: This is accomplished by the container calling the setter methods on the beans after invoking a no-argument constructor or a no-argument static factory method to instantiate the bean.
    - (-) Objects are not immutable.
  - **Constructor Injection**: It is accomplished when the container invokes a class constructor with several arguments where each representing a dependency on the other class.
    - (-) Need to have different constructors
- They can also be used together.

48

Spring Configuration – Using Java – **Setter Injection**

```
@Bean(name="customerService")
public CustomerService getCustomerService() {
   CustomerServiceImpl customerService = new CustomerServiceImpl();
   customerService.setCustomerRepository(getCustomerRepository());

   return customerService;
}

@Bean(name = "customerRepository")
public CustomerRepository getCustomerRepository() {
   return new HibernateCustomerRepositoryImpl();
}
```

Setter Injection

Simple as a method call

"Mystery" of injection goes away

Setter Injection simply calling a setter

We have to call getCustomerRepository and not create the instance of CustomerRepository in getCustomerService

Because in Spring, the beans are by default a singleton!

49

Spring Configuration – Using Java – **Constructor Injection**

```
@Bean(name="customerService")
public CustomerService getCustomerService() {
   CustomerServiceImpl customerService =
      new CustomerServiceImpl(getCustomerRepository());

   return customerService;
}

@Bean(name = "customerRepository")
public CustomerRepository getCustomerRepository() {
   return new HibernateCustomerRepositoryImpl();
}
```

Constructor Injection

Just like setter injection

We need to add a constructor to CustomerServiceImpl.

50

- Spring Configuration using XML

51

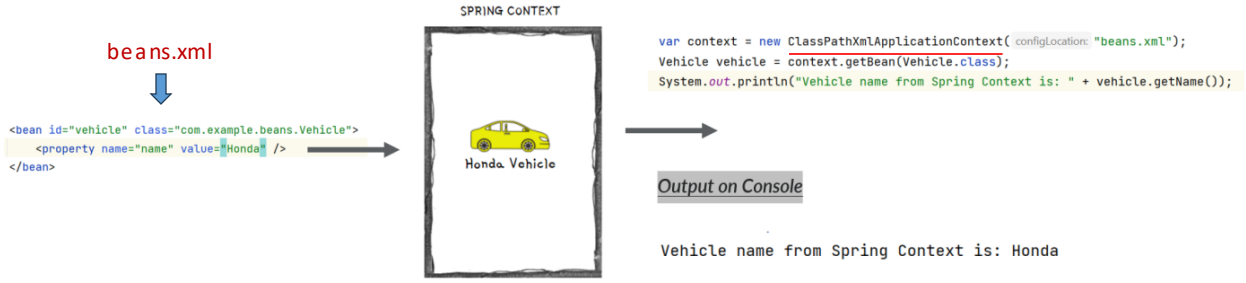
applicationContext.xml

- Same thing as ApplicationContext.java
  - Name doesn't matter
- It is a sort of a HashMap
  - Key-value pairs.
- This file should be added to the resources folder.

52

Adding new Beans using XML

- *In the initial versions of Spring, the bean and other configurations used to be done using XML. But over the time, Spring team brings annotation based configurations to make developers life easy. Today we can see XML configurations only in the older applications built based on initial versions of Spring.*
- *It is good to understand on how to create a bean inside Spring context using XML style configurations. So that, it will be useful if ever there is a scenario where you need to work in a project based on initial versions of Spring.*



53

Bean definition in XML

```
<bean
    name="customerService"
    class="com.example.service.CustomerServiceImpl"
    autowire="byName">

    <constructor-arg index="0"
                     ref="customerRepository" />   Constructor Injection

</bean>
```

```
<bean
    name="customerService"
    class="com.example.service.CustomerServiceImpl"
    autowire="byName">

    <property name="customerRepository"
              ref="customerRepository" />   Setter Injection

</bean>
```

The setter should be named setCustomerRepository

54

- Autowiring

55

Autowired

- It is a technique used to reduce the wiring up and the configuration code.
- To autowire, we have to add a @ComponentScan to the configuration file:
  - @ComponentScan ({“com.example”})
    - The text inside the {}, tells where to look for Autowire annotations.
- We can mark any @Bean as Autowired
- We can choose:
  - By Name
  - Instance Type

56

Inject Beans using @Autowired on class fields

The @Autowired annotation marks on a field, setter method, constructor is used to auto-wire the beans that is 'injecting beans'(Objects) at runtime by Spring Dependency Injection mechanism.

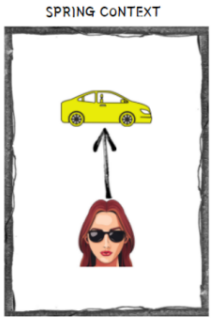
With the below code, Spring injects/auto-wire the vehicle bean to the person bean through a class field and dependency injection.

The below style is not recommended for production usage as we can't mark the fields as final.

```
@Component
public class Person {

    private String name="Lucy";

    /*...*/
    @Autowired
    private Vehicle vehicle;
```




```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Person person = context.getBean(Person.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Person name from Spring Context is: " + person.getName());
System.out.println("Vehicle name from Spring Context is: " + vehicle.getName());
System.out.println("Vehicle that Person own is: " + person.getVehicle());
```

Output on Console

```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

@Autowired(required = false) will help to avoid the NoSuchBeanDefinitionException if the bean is not available during Autowiring process.

57

@Autowired on class fields

- It should not be used in production code.
  - Even though a lot of developers use it!
- If we want to make the injected field final… we will get a compilation error:

```
public class Person {

    private String name="Lucy";

    /*
    The @Autowired annotation marks on a field, constructor, Setter method
    is used to auto-wire the beans that is 'injecting beans'(Objects) at runtime
    by Spring Dependency Injection mechanism
    * */
    @Autowired
    private final Vehicle vehicle;

    public String g
        return name
    }

    public void set
        this.name = name;
    }
```

Variable 'vehicle' might not have been initialized
Add constructor parameter Alt+Shift+Enter More actions... Alt+Enter

```
com.example.beans
@Component
public class Vehicle
extends Object
example_11
```

58

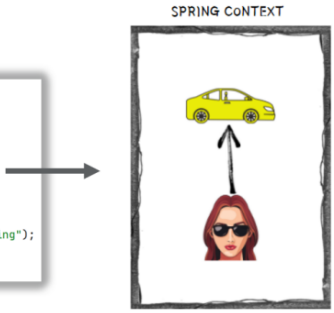
Inject Beans using @Autowired with constructor

- *The @Autowired annotation marks on a field, setter method, constructor is used to auto-wire the beans that is 'injecting beans'(Objects) at runtime by Spring Dependency Injection mechanism.*
- *With the below code, Spring injects/auto-wire the vehicle bean to the person bean through a constructor and dependency injection.*
- *From Spring version 4.3, when we only have one constructor in the class, writing the @Autowired annotation is optional*

```
@Component
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(Vehicle vehicle){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle;
    }
```




```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Person person = context.getBean(Person.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Person name from Spring Context is: " + person.getName());
System.out.println("Vehicle name from Spring Context is: " + vehicle.getName());
System.out.println("Vehicle that Person own is: " + person.getVehicle());
```

Output on Console

```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

This is the recommended method to be used in production code!

59

Inject Beans using @Autowired on setter method

The @Autowired annotation marks on a field, setter method, constructor is used to auto-wire the beans that is 'injecting beans'(Objects) at runtime by Spring Dependency Injection mechanism.

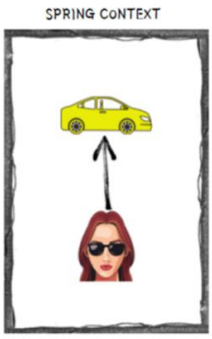
With the below code, Spring injects/auto-wire the vehicle bean to the person bean through a setter method and dependency injection.

The below style is not recommended for production usage as we can't mark the fields as final and not readable friendly.

```
@Component
public class Person {

    private String name="Lucy";
    private Vehicle vehicle;

    @Autowired
    public void setVehicle(Vehicle vehicle) {
        this.vehicle = vehicle;
    }
```




```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Person person = context.getBean(Person.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Person name from Spring Context is: " + person.getName());
System.out.println("Vehicle name from Spring Context is: " + vehicle.getName());
System.out.println("Vehicle that Person own is: " + person.getVehicle());
```

Output on Console

```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

Even in this case, if we want to make the Vehicle field final, this approach will not work.

60

Autowire in Java - Example

SpeakerServiceImpl.java

```
@Autowired
public void setRepository(SpeakerRepository repository) {
    this.repository = repository;
}
```

```
@Configuration
public class AppConfig {

    @Bean(name = "speakerService")
    @Scope(value= BeanDefinition.SCOPE_SINGLETON)
    public SpeakerService getSpeakerService() {
        SpeakerServiceImpl service = new SpeakerServiceImpl();
        //service.setRepository(getSpeakerRepository());
        return service;
    }
```

No need to call the setter, since we added @Autowire

61

Stereotypes

- To fully Autowire, we need to use Stereotypes.
  - Not use the hybrid way (like in the previous slide)
- A stereotype is a special kind of annotation that provides metadata about the role of a class in the application.
  - It is very useful when there are hundreds of beans.
- There are 4 stereotypes:

@Component	@Repository	@Service
(The same as @Bean)	It is used to denote a class that acts as a repository (we can use instead @Bean or @Component), but it is clearer with @Repository	It is used to denote a class where we put the business logic.

There is also the @Controller stereotype... used in web applications.

62

@Component Annotation

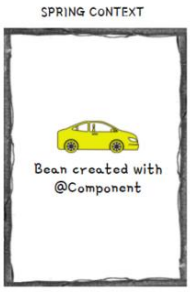
```java
@Component
public class Vehicle {

    private String name;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public void printHello(){
        System.out.println(
            "Printing Hello from Component Vehicle Bean");
    }
}
```




```java
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
Vehicle vehicle = context.getBean(Vehicle.class);
System.out.println("Component Vehicle name from Spring Context is: " + vehicle.getName());
vehicle.printHello();
```

Output on Console

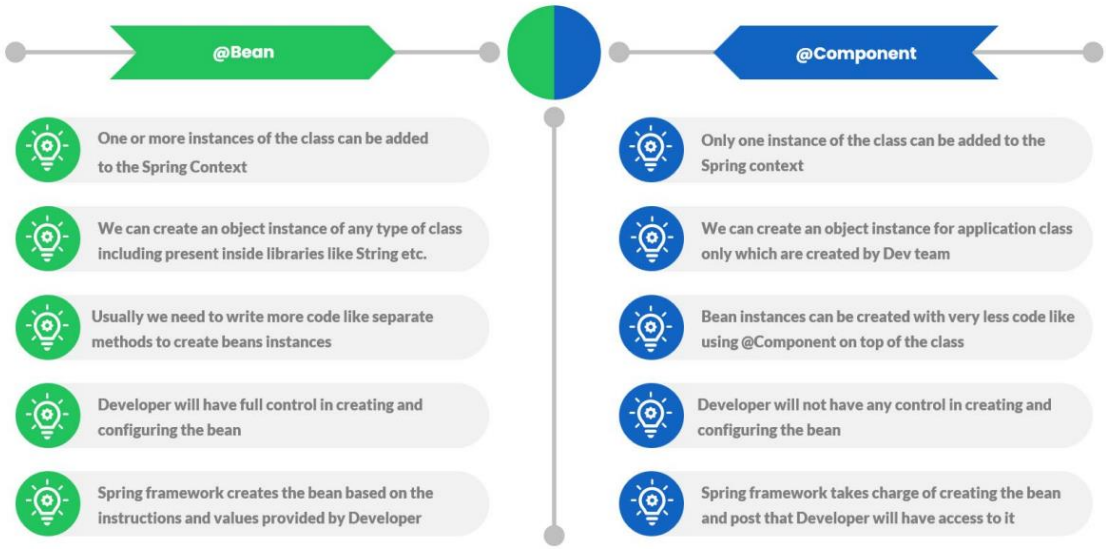
```
Component Vehicle name from Spring Context is: null
Printing Hello from Component Vehicle Bean
```

```java
@Configuration
@ComponentScan(basePackages = "com.example.beans")
public class ProjectConfig {

}
```

63

@Bean Vs @Component



64

Fully Autowire – Example

```
@Configuration
@ComponentScan({"com.example"})
public class AppConfig {

    /*
    @Bean(name = "speakerService")
    @Scope(value= BeanDefinition.SCOPE_SINGLETON)
    public SpeakerService getSpeakerService() {
        //SpeakerServiceImpl service = new SpeakerServiceImpl(getSpeakerRepository());
        SpeakerServiceImpl service = new SpeakerServiceImpl();
        //service.setRepository(getSpeakerRepository());
        return service;
    }
```

Don't need these bean definitions anymore!

```
    @Bean(name = "speakerRepository")
    public SpeakerRepository getSpeakerRepository() {
        return new HibernateSpeakerRepositoryImpl();
    }*/

}
```

```
@Service("speakerService")
public class SpeakerServiceImpl implements SpeakerService {

    private SpeakerRepository repository;

    public SpeakerServiceImpl() {
        System.out.println("SpeakServiceImpl no args constructor");
    }

    public List<Speaker> findAll() {
        return repository.findAll();
    }

    @Autowired
    public void setRepository(SpeakerRepository repository) {
        this.repository = repository;
    }
```

```
@Repository("speakerRepository")
public class HibernateSpeakerRepositoryImpl implements SpeakerRepository {

    public List<Speaker> findAll() {
```

Both SpeakerServiceImpl and HibernateSpeakerRepositoryImpl are on package "com.example"

65

Autowire in XML - Example

```
<bean name="speakerRepository"
      class="com.example.repository.HibernateSpeakerRepositoryImpl" />

<bean name="speakerService" class="com.example.service.SpeakerServiceImpl" autowire="constructor">
    <!--<constructor-arg index="0" ref="speakerRepository" />-->
</bean>
```

```
<bean name="speakerRepository"
      class="com.example.repository.HibernateSpeakerRepositoryImpl" />

<bean name="speakerService" class="com.example.service.SpeakerServiceImpl" autowire="byType"> use the setter
    <!--<constructor-arg index="0" ref="speakerRepository" />-->
</bean>
```

Or we can fully autowire:

```
<!-- Enable component scanning for the specified base package(s) -->
<context:component-scan base-package="com.example"/>
```

(All classes annotated with Spring stereotypes such as @Component, @Service, @Repository, @Controller, within the specified base package(s) will be automatically detected and registered as Spring beans)

66

How Autowiring works with multiple Beans of the same type (1)

By default Spring tries autowiring with class type. But this approach will fail if the same class type has multiple beans.

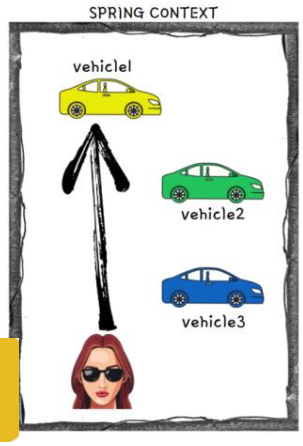
If the Spring context has multiple beans of same class type like below, then Spring will try to auto-wire based on the parameter name/field name that we use while configuring autowiring annotation.

In the below scenario, we used 'vehicle1' as constructor parameter. Spring will try to auto-wire with the bean which has same name like shown in the image below.

```
@Component
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(Vehicle vehicle1){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle1;
    }
```



STEP 1

From the Spring 6.1.0 version, the support for the STEP 1 is removed. We no more can autowire the beans using their names. Only STEP 2 and STEP 3 are going to work.

67

How Autowiring works with multiple Beans of the same type (2)

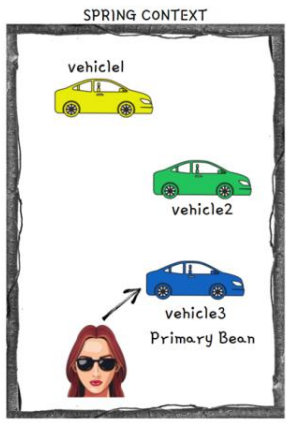
If the parameter name/field name that we use while configuring autowiring annotation is not matching with any of the bean names, then Spring will look for the bean which has @Primary configured.

In the below scenario, we used 'vehicle' as constructor parameter. Spring will try to auto-wire with the bean which has same name and since it can't find a bean with the same name, it will look for the bean with @Primary configured like shown in the image below.

```
@Component
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(Vehicle vehicle){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle;
    }
```



STEP 2

68

How Autowiring works with multiple Beans of the same type (3)

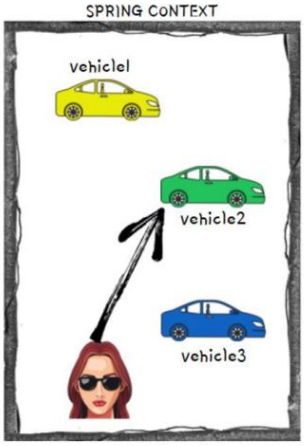
If the parameter name/field name that we use while configuring autowiring annotation is not matching with any of the bean names and even Primary bean is not configured, then Spring will look if @Qualifier annotation is used with the bean name matching with Spring context bean names.

In the below scenario, we used 'vehicle2' with @Qualifier annotation. Spring will try to auto-wire with the bean which has same name like shown in the image below.

```java
@Component
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(@Qualifier("vehicle2") Vehicle vehicle){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle;
    }
```



69

Understanding & Avoiding Circular Dependencies

A Circular dependency will happen if 2 beans are waiting for each to create inside the Spring context in order to do auto-wiring.

Consider the below scenario, where Person has a dependency on Vehicle and Vehicle has a dependency on Person. In such scenarios, Spring will throw UnsatisfiedDependencyException due to circular reference.

As a developer, it is our responsibility to make sure we are defining the configurations/dependencies that will result in circular dependencies.

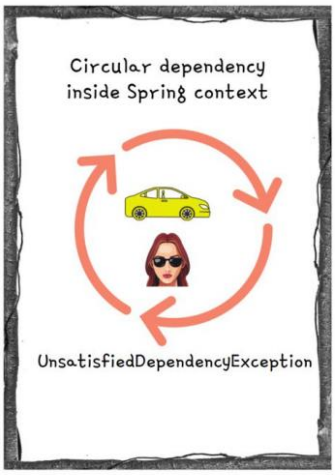
```java
@Component
public class Person {

    private String name="Lucy";
    private Vehicle vehicle;

    @Autowired
    public void setVehicle(Vehicle vehicle) {
        this.vehicle = vehicle;
    }
```

```java
public class Vehicle {

    private String name;
    @Autowired
    private Person person;
```



70

- Bean Scopes

71

Scopes

5 scopes

Valid in any configuration

- Singleton
- Prototype

Valid only in web-aware Spring projects

- Request
- Session
- Global

72

Singleton

One instantiation
(only one object)

Default bean scope

Single instance per Spring container
(one instance per application context)

73

Singleton - Example

```
@Service("customerService")
@Scope("singleton")
public class CustomerServiceImpl implements
CustomerService {
```

Singleton - Java Config

@Scope

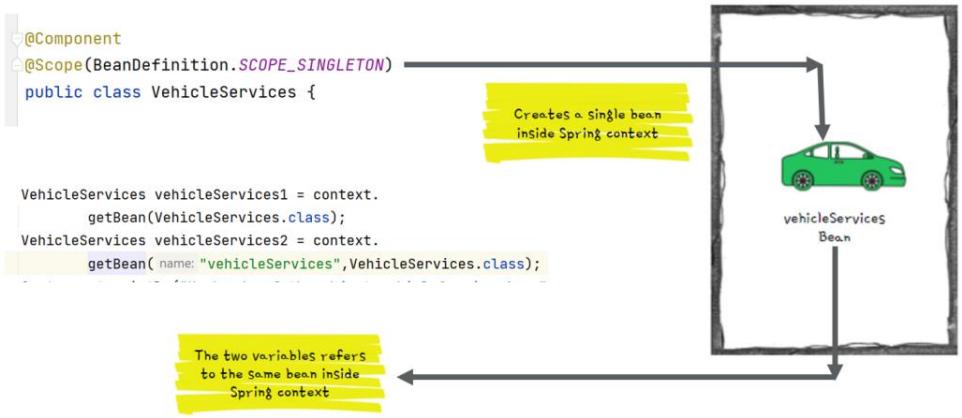
Requires AOP jar (if we use Maven, this is a transitive dependency)

74

Singleton Bean Scope

Singleton is the default scope of a bean in Spring. In this scope, for a single bean we always get a same instance when you refer or autowire inside your application.

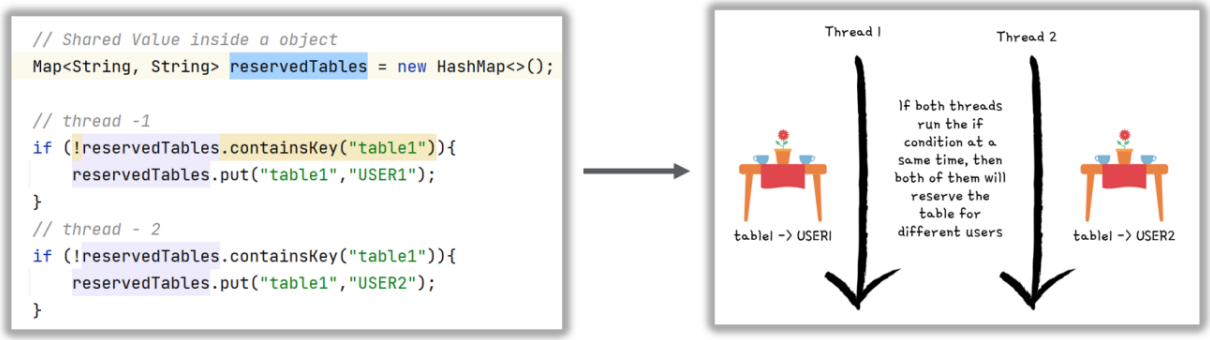
Unlike Singleton design pattern where we have only 1 instance in entire app, inside Singleton scope Spring will make sure to have only 1 instance per unique bean. For example, if you have multiple beans of same type, then Spring Singleton scope will maintain 1 instance per each bean declared of same type.



75

Race Condition

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.



76

Use cases of singleton beans

Singleton Beans use cases

- ✓ Since the same instance of singleton bean will be used by multiple threads inside your application, it is very important that these beans are immutable.
- ✓ This scope is more suitable for beans which handles service layer, repository layer business logics.

1. Building mutable singleton beans, will result in the race conditions inside multi thread environments.
2. There are ways to avoid race conditions due to mutable singleton beans with the help of synchronization.
3. But it is not recommended, since it brings lot of complexity and performance issues inside your app. So please don't try to build mutable singleton beans.

Singleton Beans SHOULD be immutable.

(Immutable: There are no field variables or these variables are declared as FINAL.)

Simple Java POJO classes (without business logic) should not be created as beans.

77

Eager & Lazy Implementation

By default Spring will create all the singleton beans eagerly during the startup of the application itself. This is called Eager instantiation.
We can change the default behavior to initialize the singleton beans lazily only when the application is trying to refer to the bean. This approach is called Lazy instantiation.

```
@Component(value="personBean")
@Lazy
public class Person {

    private String name="Lucy";
    private final Vehicle vehicle;

    @Autowired
    public Person(Vehicle vehicle){
        System.out.println("Person bean created by Spring");
        this.vehicle = vehicle;
    }

    public String getName() {
        return name;
    }
```

```
public static void main(String[] args) {
    var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
    System.out.println("Before retrieving the Person bean from the Spring Context");
    Person person = context.getBean(Person.class);
    System.out.println("After retrieving the Person bean from the Spring Context");
}
```

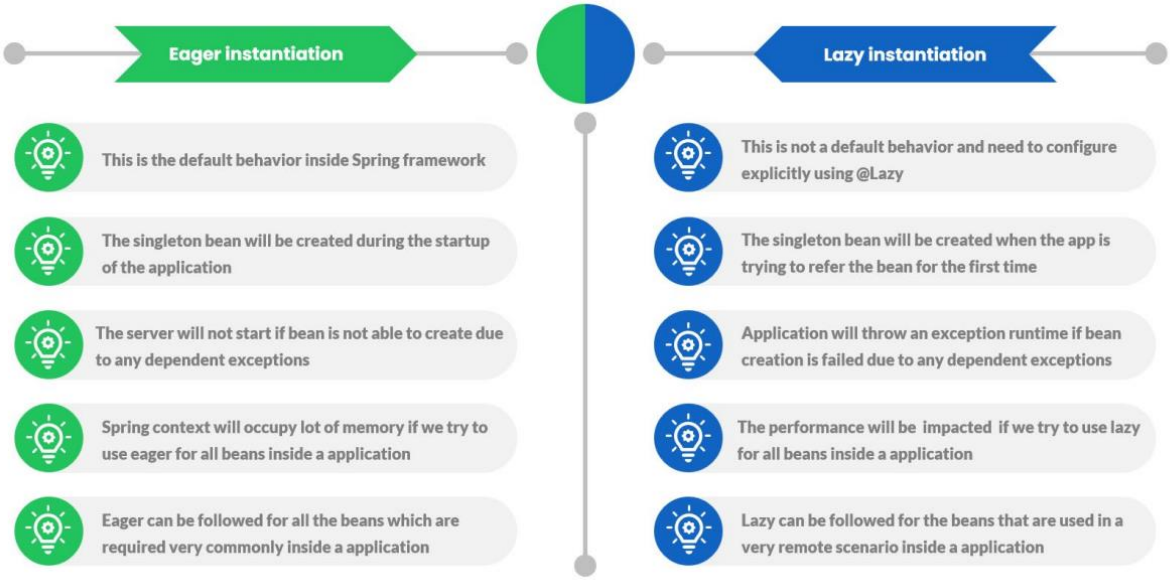
Output:

```
Before retrieving the Person bean from the Spring Context
Person bean created by Spring
After retrieving the Person bean from the Spring Context
```

We should usually use the 'Eager' implementation. If there is an error during the bean creation, the application start will fail. If we use the 'Lazy' implementation, and there is a problem, then an exception will be thrown.

78

Eager Vs Lazy Instantiation



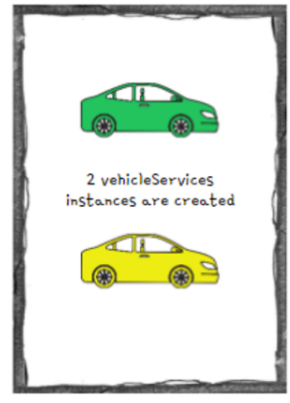
79

Prototype Bean Scope

With prototype scope, every time we request a reference of a bean, Spring will create a new object instance and provide the same.
Prototype scope is rarely used inside the applications and we can use this scope only in the scenarios where your bean will frequently change the state of the data which will result race conditions inside multi thread environment. Using prototype scope will not create any race conditions.

```
@Component
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class VehicleServices {
```

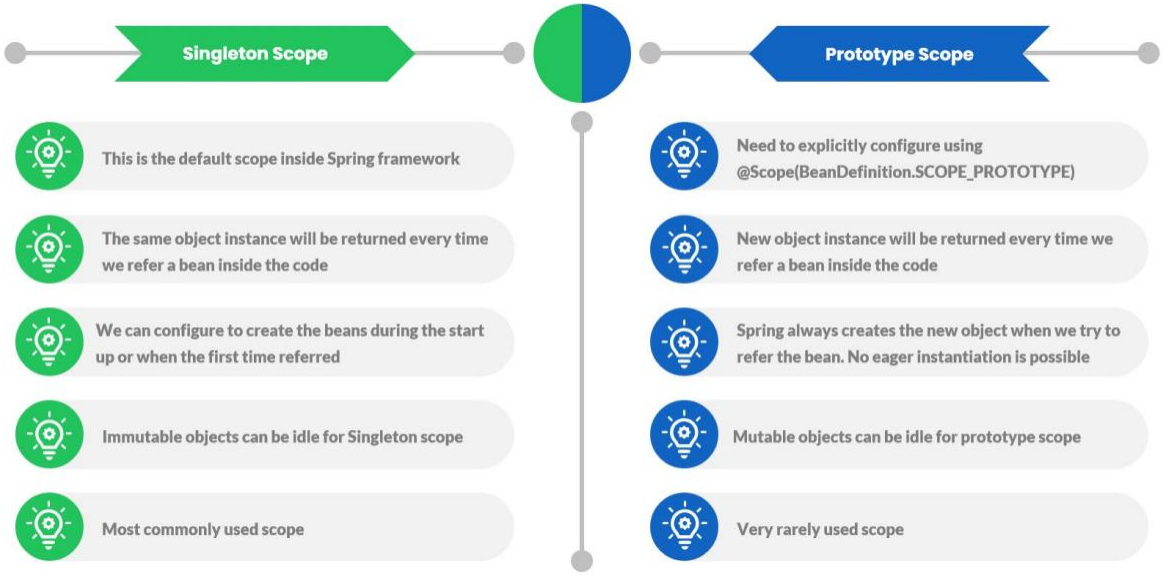
```
VehicleServices vehicleServices1 = context.
        getBean(VehicleServices.class);
VehicleServices vehicleServices2 = context.
        getBean( name: "vehicleServices",VehicleServices.class);
```



Unique instance Per request	Each time we request a bean, we are Guaranteed unique instance	Opposite of Singleton

80

Singleton Vs Prototype



81