

# Spring, SpringBoot, JPA, Hibernate

## Zero To Master

# Agenda of the course

eazy  
bytes

## Intro to Spring framework

- *What is Spring?*
- *Spring Vs Java EE*
- *Evolution of Spring*
- *Spring release timeline*
- *Different projects inside Spring*

## Spring Core

- *Inversion of Control (IoC)*
- *Dependency Injection (DI)*
- *Different approaches for Beans creation*
- *Autowiring, Bean Scopes*
- *Aspect-Oriented Programming (AOP)*

## Spring MVC

- *Introduction to MVC pattern*
- *Overview of Web Apps*
- *How Web Apps works ?*
- *Spring MVC internal flow*
- *Create a web App using Spring MVC*
- *Spring MVC validations*

# Agenda of the course

eazy  
bytes

## Thymeleaf

- *How to build dynamic web apps using Thymeleaf & Spring*
- *Thymeleaf integration with Spring, Spring MVC, Spring Security*

## Spring Boot

- *Why Spring Boot?*
- *Auto-configuration*
- *Build Web Apps using SpringBoot, Thymeleaf, Spring MVC*
- *Spring Boot Dev Tools*
- *Spring Boot H2 Database*
- *Connecting to AWS MYSQL DB*

## Spring Security

- *Authentication & Authorization*
- *Securing web Apps/REST APIs using Spring Security*
- *Role based access*
- *Cross-Site Request Forgery (CSRF)*
- *Cross-Origin Resource Sharing (CORS)*

# Agenda of the course

eazy  
bytes

## Spring JDBC

- *Problems with Core Java JDBC*
- *Advantages with Spring JDBC*
- *Performing CRUD operations with Spring JDBC*
- *Details about JdbcTemplate*
- *RowMapper*

## Spring Data

- *Why do we need ORM frameworks?*
- *Introduction to JPA*
- *Derived Query methods in JPA*
- *OneToOne, OneToMany, ManyToOne, ManyToMany mappings inside JPA/Hibernate*
- *Sorting, Pagination, JPQL*

## Spring REST

- *Building Rest Services*
- *Consuming Rest Services using OpenFeign, Web Client, RestTemplate*
- *Securing Rest Services*
- *Spring Data Rest*
- *HAL Explorer*

# Agenda of the course

eazy  
bytes

## Logging

- *Logging severities*
- *How to make logging configurations inside Spring Boot*
- *Writing logs into a file*

## Properties & Profiles

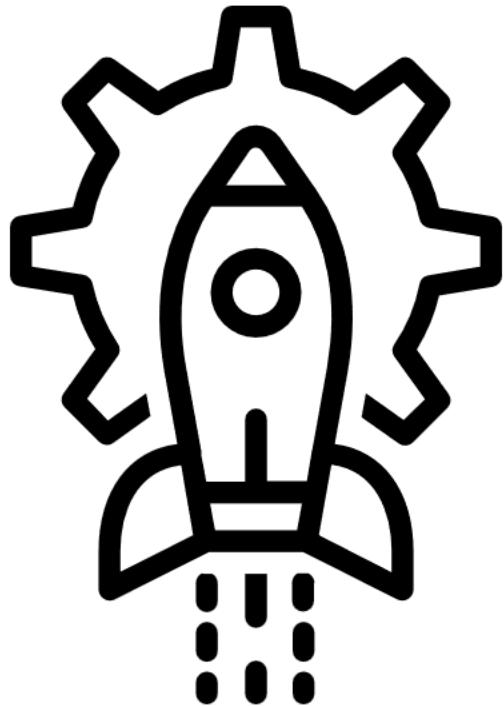
- *How to define custom properties*
- *How to read properties in Java code*
- *Deep dive on Spring Profiles*
- *Activating a Profile*
- *Conditional Bean creation using Profile*

## Actuator

- *Introduction to SpringBoot Actuator*
- *Exploring the APIs of Actuator*
- *Securing Actuator*
- *Viewing Actuator Data in Spring Boot Admin*

# Agenda of the course

---



**DEPLOYING  
SPRINGBOOT  
WEBAPP INTO  
CLOUD (AWS)**



# WHAT IS SPRING?



The Spring Framework (shortly, Spring) is a mature, powerful and highly flexible framework focused on building web applications in Java.

Spring makes programming Java quicker, easier, and safer for everybody. Its focus on speed, simplicity, and productivity has made it the world's most popular Java framework.

Whether you're building secure, reactive, cloud-based microservices for the web, or complex streaming data flows for the enterprise, Spring has the tools to help.

Born as an alternative to EJBs in the early 2000s, the Spring framework quickly overtook its opponent with its simplicity, variety of features, and its third-party library integrations.

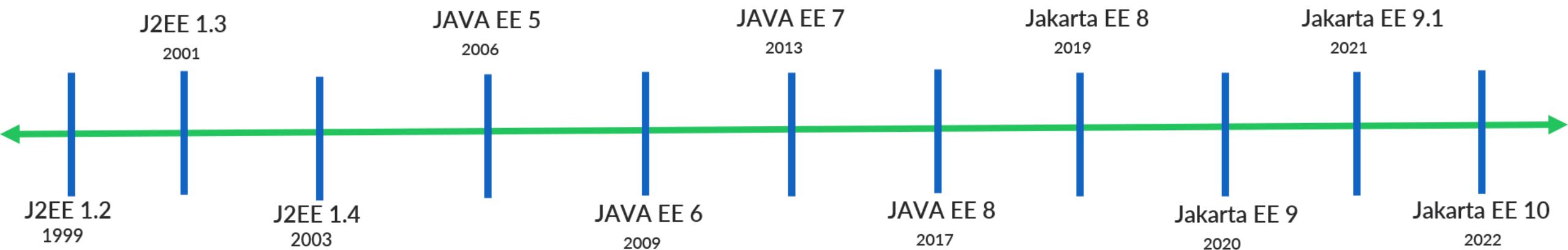
It is so popular, that its main competitor quit the race when Oracle stopped the evolution of Java EE 8, and the community took over its maintenance via Jakarta EE.

The main reason of Spring framework success is, it regularly introduces features/projects based on the latest market trends, needs of the Dev community. For ex: SpringBoot

Spring is open source. It has a large and active community that provides continuous feedback based on a diverse range of real-world use cases.

# JAVA EE RELEASE TIMELINE

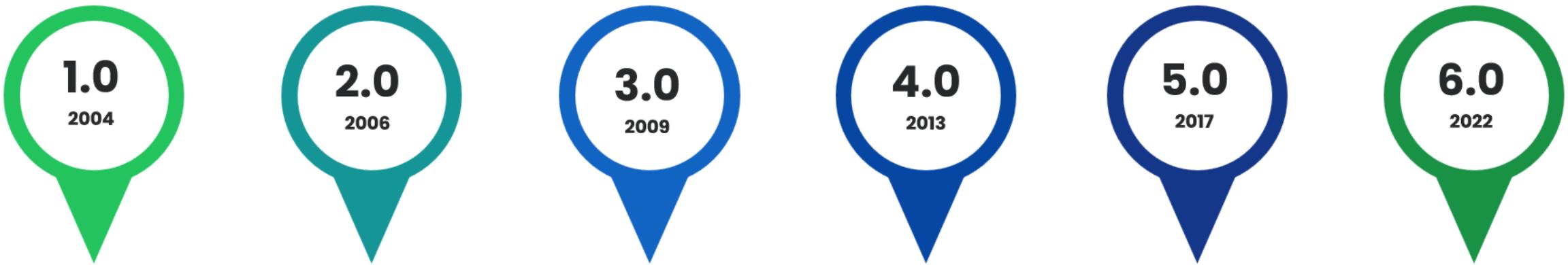
eazy  
bytes



- 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.

# SPRING RELEASE TIMELINE

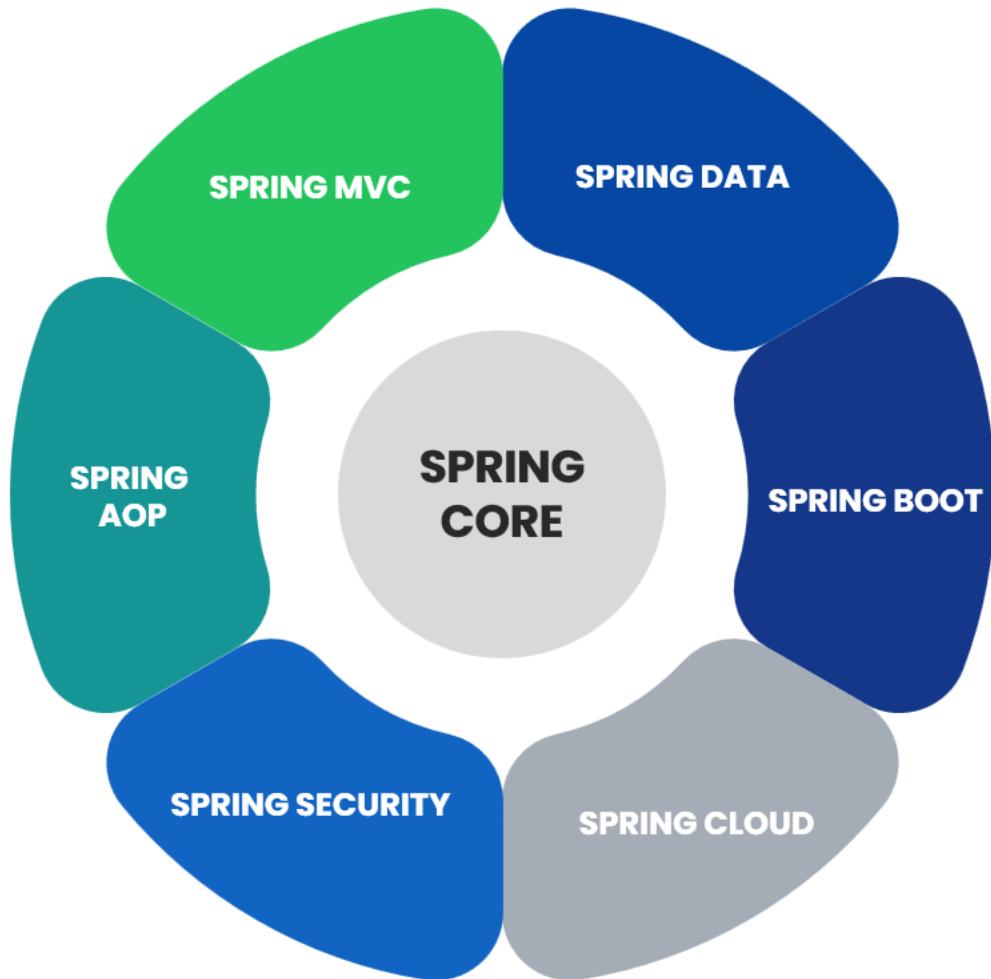
eazy  
bytes



- 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.

# SPRING CORE

eazy  
bytes



- 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

# INVERSION OF CONTROL & DEPENDENCY INJECTION

eazy  
bytes



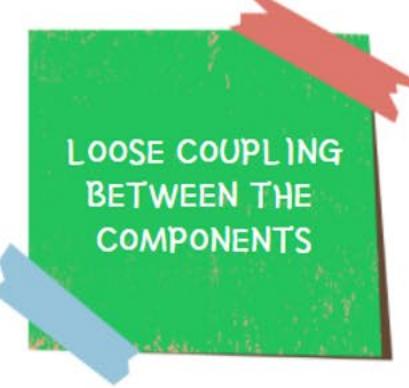
CORE PRINCIPLES  
OF SPRING

- Inversion of Control (IoC) is a Software Design Principle, independent of language, which does not actually create the objects but describes the way in which object is being created.
- IoC is the principle, where the control flow of a program is inverted: instead of the programmer controlling the flow of a program, the framework or service takes control of the program flow.
- Dependency Injection is the pattern through which Inversion of Control achieved.
- Through Dependency Injection, the responsibility of creating objects is shifted from the application to the Spring IoC container. It reduces coupling between multiple objects as it is dynamically injected by the framework.

# ADVANTAGES OF IoC & DI

---

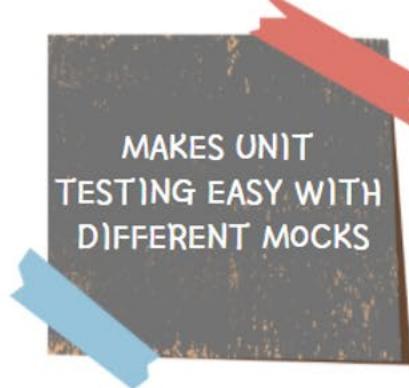
eazy  
bytes



LOOSE COUPLING  
BETWEEN THE  
COMPONENTS



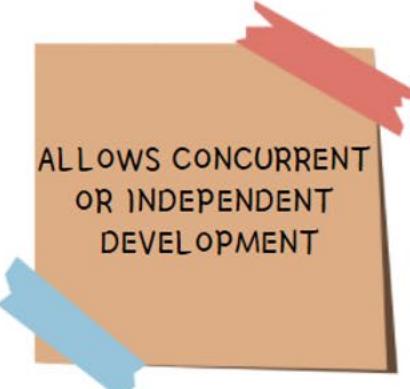
MINIMIZES THE  
AMOUNT  
OF CODE IN YOUR  
APPLICATION



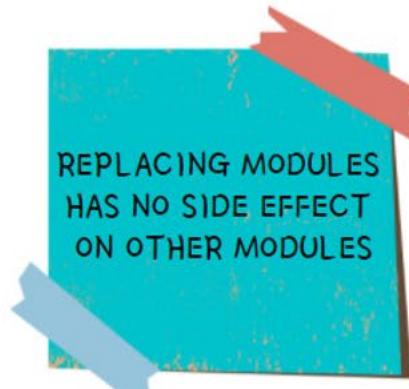
MAKES UNIT  
TESTING EASY WITH  
DIFFERENT MOCKS



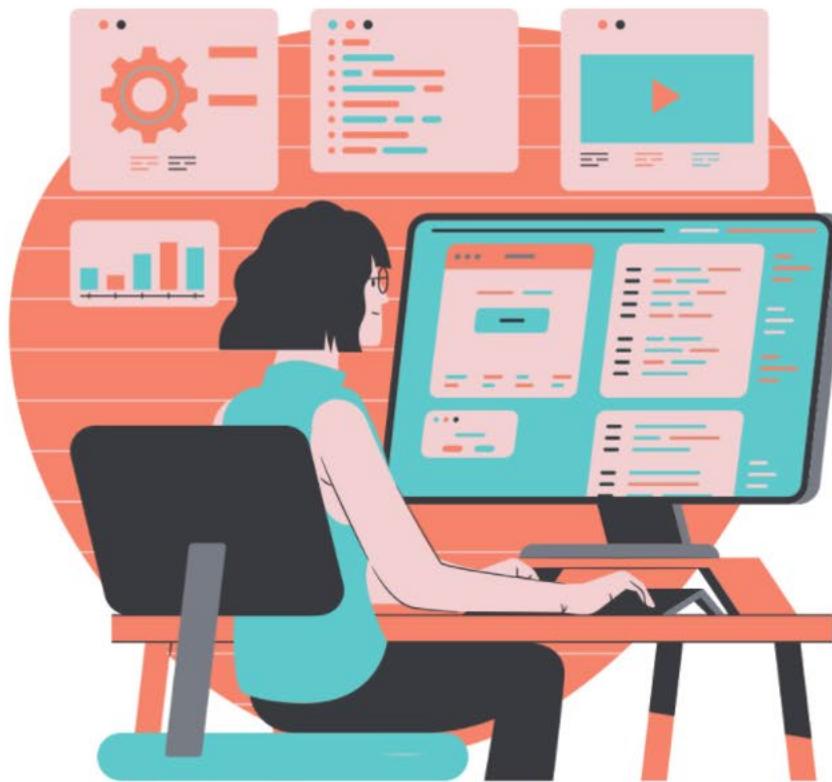
INCREASED SYSTEM  
MAINTAINABILITY &  
MODULE REUSABILITY



ALLOWS CONCURRENT  
OR INDEPENDENT  
DEVELOPMENT



REPLACING MODULES  
HAS NO SIDE EFFECT  
ON OTHER MODULES



TIGHT COUPLING  
SCENARIO

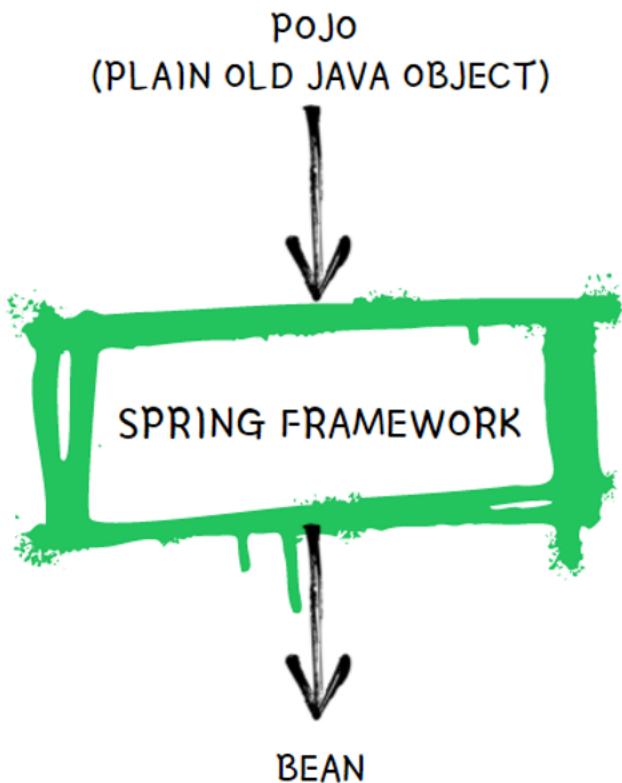


LOOSE COUPLING  
SCENARIO

# REAL LIFE LOOSE COUPLING EXAMPLE

# SPRING BEANS, CONTEXT, SpEL

eazy  
bytes



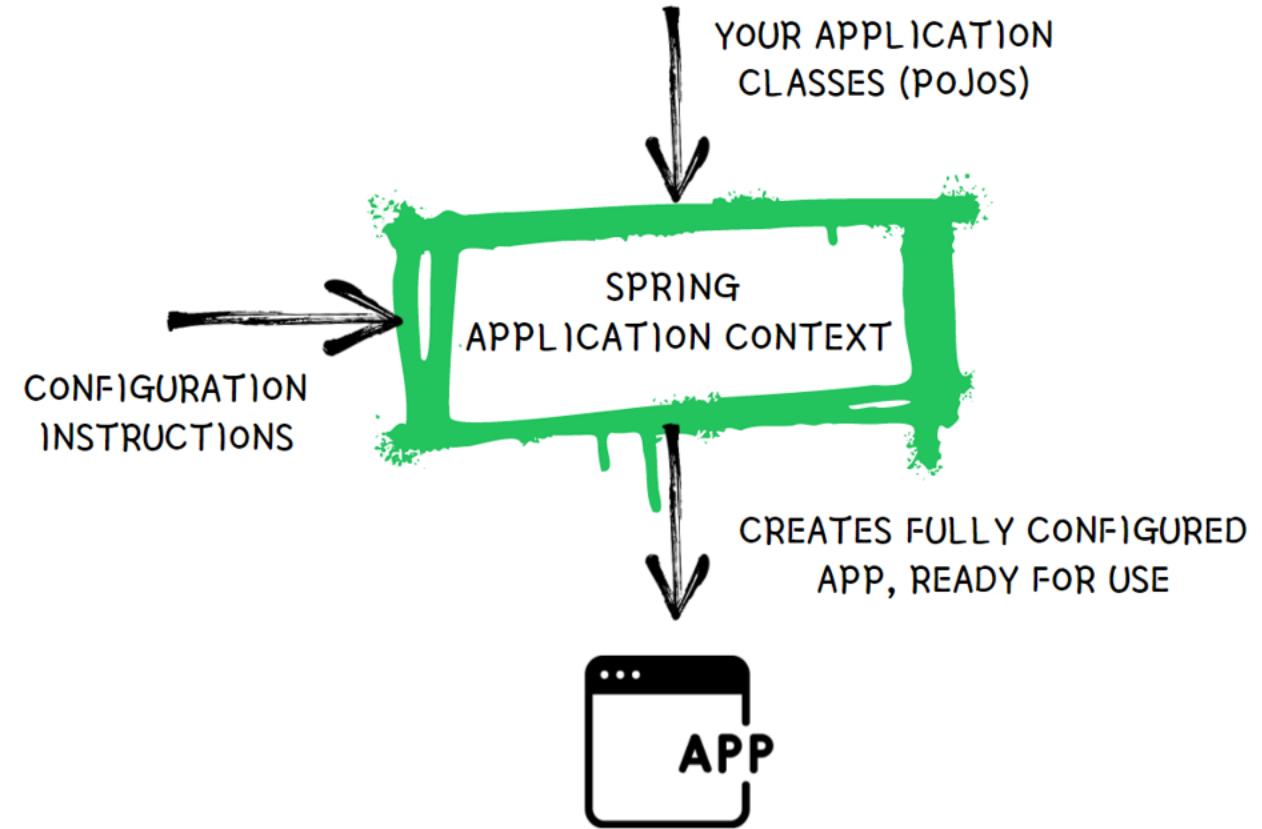
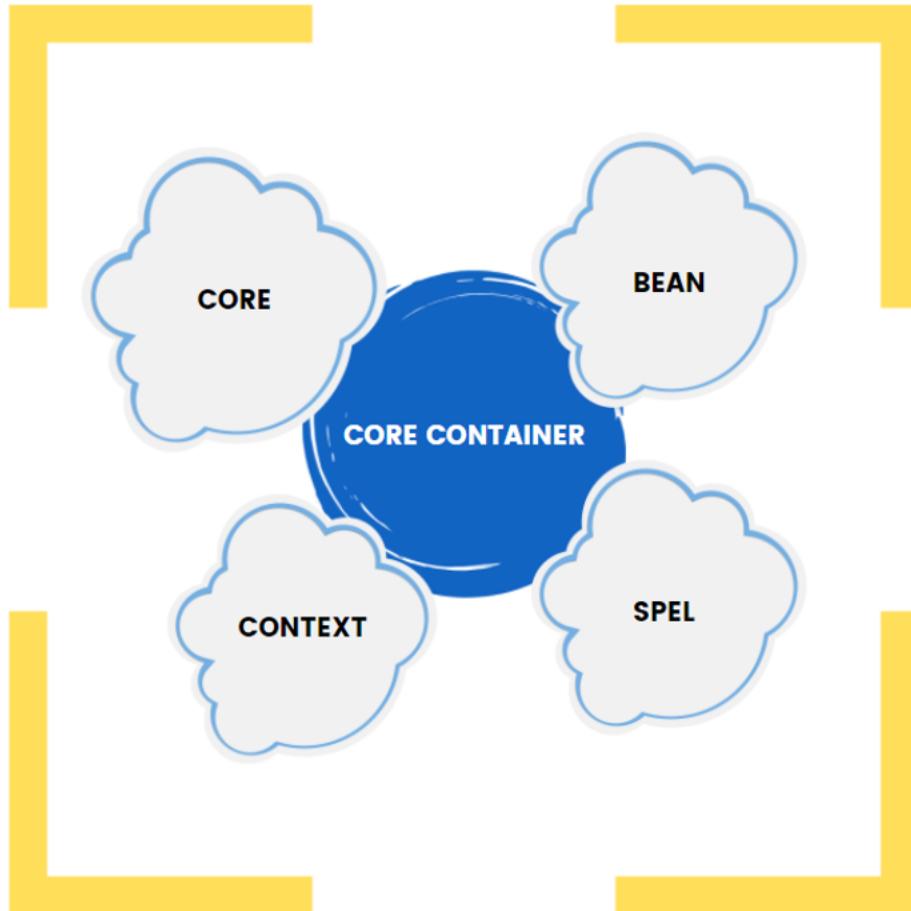
- Any normal Java class that is instantiated, assembled, and otherwise managed by a Spring IoC container is called Spring Bean.
- These beans are created with the configuration metadata that you supply to the container either in the form of XML configs and Annotations.
- Spring IoC Container manages 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.

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

# SPRING IoC CONTAINER

eazy  
bytes



# MAVEN

# ADDING NEW BEANS TO SPRING CONTEXT

eazy  
bytes

When we create an java object with new () operator directly as shown below, then your Spring Context/Spring IoC Container will not have any clue of the object.

SPRING CONTEXT



```
Vehicle vehicle = new Vehicle();
```

@Bean annotation lets Spring know that it needs to call this method when it initializes its context and adds the returned object/value to the Spring context/Spring IoC Container.

SPRING CONTEXT



```
@Bean  
Vehicle vehicle() {  
    var veh = new Vehicle();  
    veh.setName("Audi 8");  
    return veh;  
}
```

# NoUniqueBeanDefinitionException

eazy  
bytes

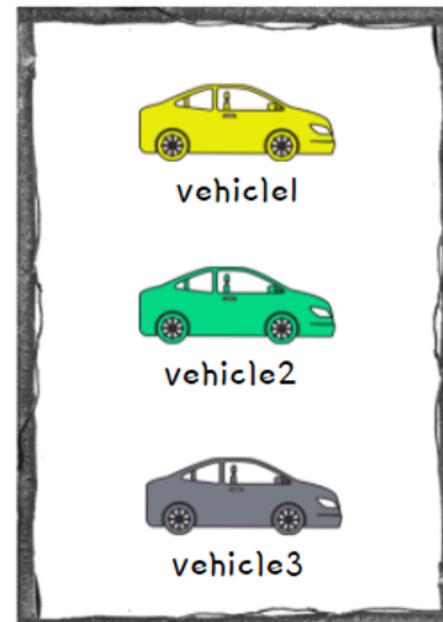
When we create multiple objects of same type and try to fetch the bean from context by type, then Spring cannot guess which instance you've declared you refer to. This will lead to NoUniqueBeanDefinitionException like shown below,

```
@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;
}
```

SPRING CONTEXT



```
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)
```

# NoUniqueBeanDefinitionException

eazy  
bytes

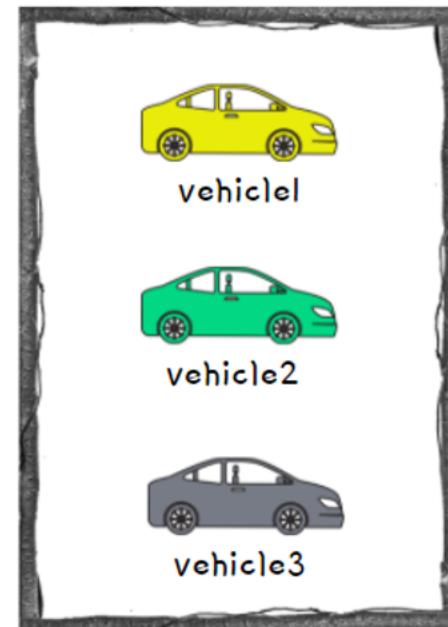
To avoid `NoUniqueBeanDefinitionException` in these kind of scenarios, we can fetch the bean from the context by mentioning its name like shown below,

```
@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;
}
```

SPRING CONTEXT



```
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

# DIFFERENT WAYS TO NAME A BEAN

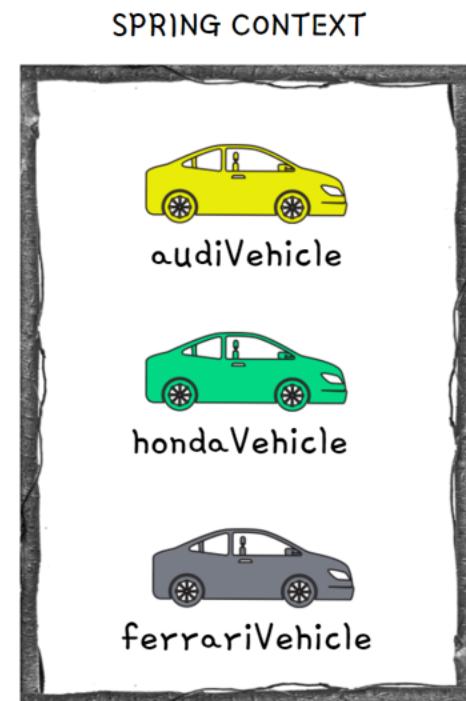
eazy  
bytes

By default, Spring will consider the method name as the bean name. But if we have a custom requirement to define a separate bean name, then we can use any of the below approach with the help of @Bean annotation,

```
@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;
}

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



```
Vehicle veh1 = context.getBean( name: "audiVehicle",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh1.getName());

Vehicle veh2 = context.getBean( name: "hondaVehicle",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh2.getName());

Vehicle veh3 = context.getBean( name: "ferrariVehicle",Vehicle.class);
System.out.println("Vehicle name from Spring Context is: " + veh3.getName());
```

## Output on Console

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

# @Primary Annotation

eazy  
bytes

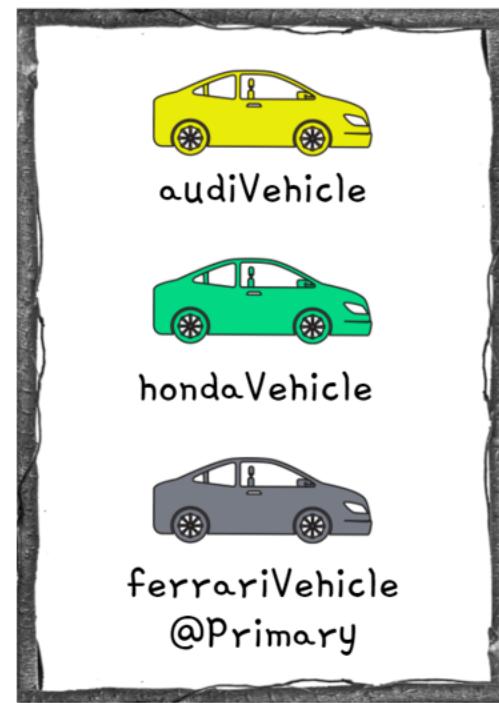
When you have multiple beans of the same kind inside the Spring context, you can make one of them primary by using @Primary annotation. Primary bean is the one which Spring will choose if it has multiple options and you don't specify a name. In other words, it is the default bean that Spring Context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

SPRING CONTEXT



```
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

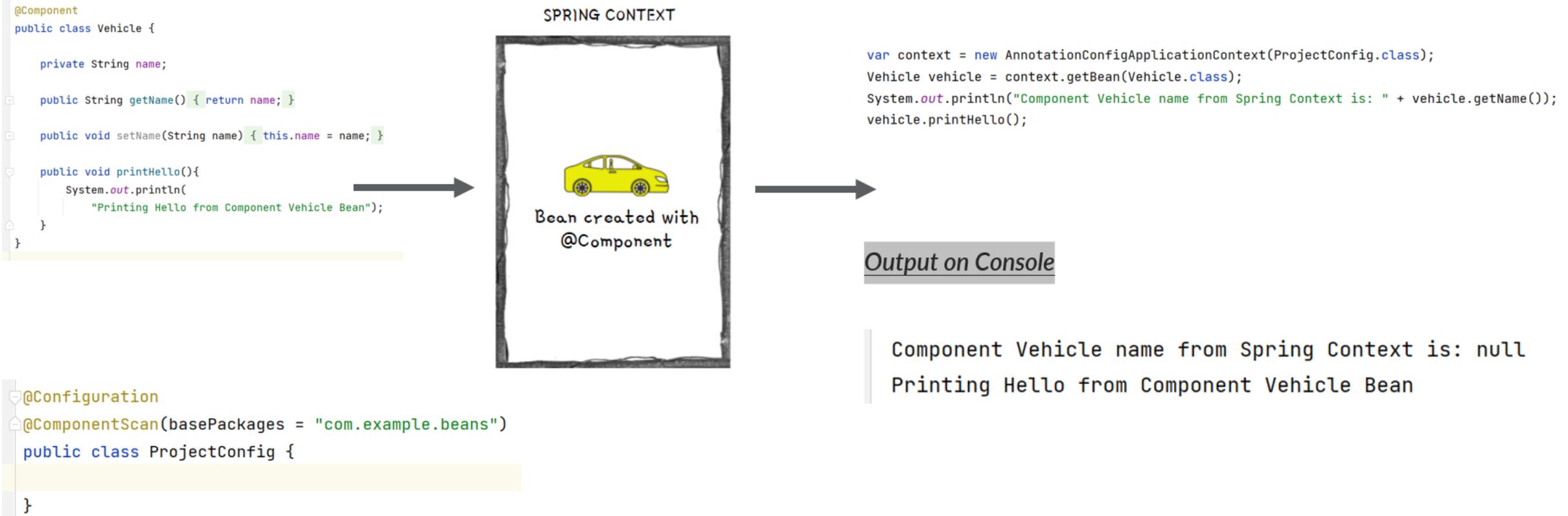
Primary Vehicle name from Spring Context is: Ferrari

# @Component Annotation

eazy  
bytes

`@Component` is one of the most commonly used stereotype annotation by developers. Using this we can easily create and add a bean to the Spring context by writing less code compared to the `@Bean` option. With stereotype annotations, we need to add the annotation above the class for which we need to have an instance in the Spring context.

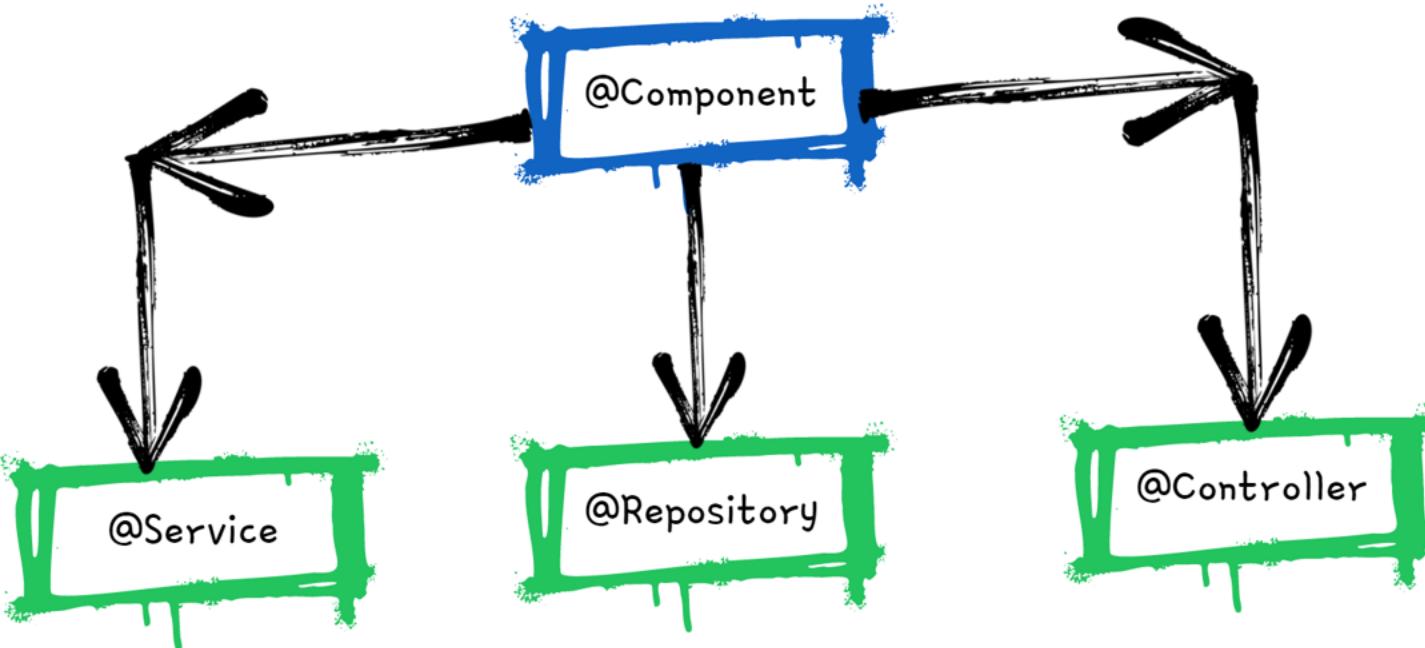
Using `@ComponentScan` annotation over the configuration class, instruct Spring on where to find the classes you marked with stereotype annotations.



# Spring Stereotype Annotations

eazy  
bytes

- Spring provides special annotations called Stereotype annotations which will help to create the Spring beans automatically in the application context.
- The stereotype annotations in spring are @Component, @Service, @Repository and @Controller



@Component is used as general on top of any Java class. It is the base for other annotations.

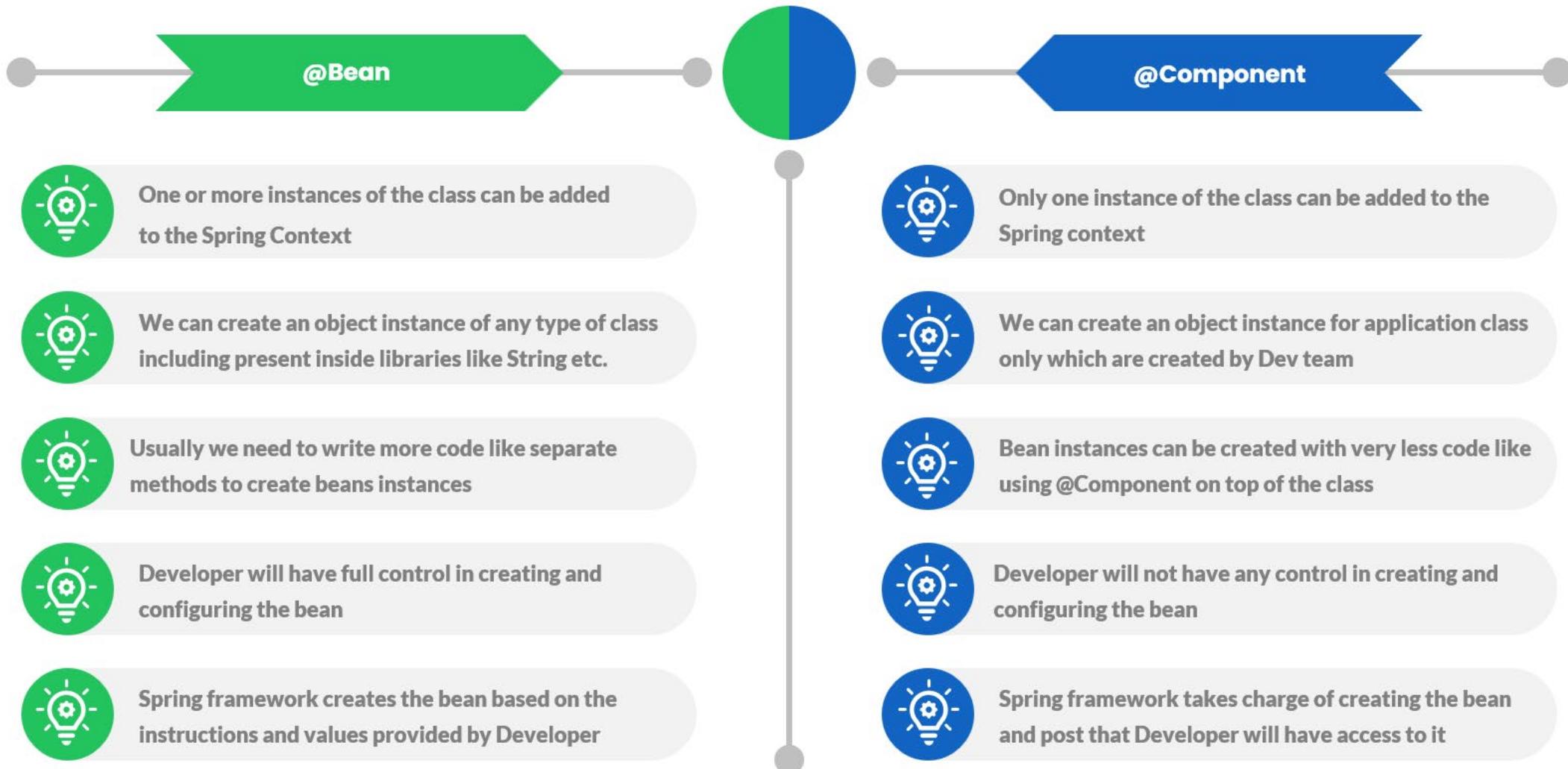
@Service can be used on top of the classes inside the service layer especially where we write business logic and make external API calls.

@Repository can be used on top of the classes which handles the code related to Database access related operations like Insert, Update, Delete etc.

@Controller can be used on top of the classes inside the Controller layer of MVC applications.

# @Bean Vs @Component

eazy  
bytes



# @PostConstruct Annotation

eazy  
bytes

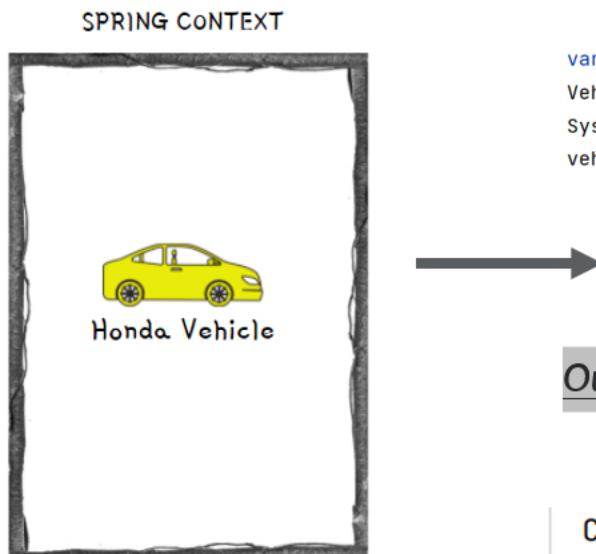
- We have seen that when we are using stereotype annotations, we don't have control while creating a bean. But what if we want to execute some instructions post Spring creates the bean. For the same, we can use @PostConstruct annotation.
- We can define a method in the component class and annotate that method with @PostConstruct, which instructs Spring to execute that method after it finishes creating the bean.
- Spring borrows the @PostConstruct annotation from Java EE.

```
@Component
public class Vehicle {
    private String name;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @PostConstruct
    public void initialize() {
        this.name = "Honda";
    }

    public void printHello(){...}
}

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



```
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: Honda  
Printing Hello from Component Vehicle Bean

# @PreDestroy Annotation

eazy  
bytes

- *@PreDestory annotation can be used on top of the methods and Spring will make sure to call this method just before clearing and destroying the context.*
- *This can be used in the scenarios where we want to close any IO resources, Database connections etc.*
- *Spring borrows the @PreDestory annotation also from Java EE.*

```
@Component
public class Vehicle {

    private String name;

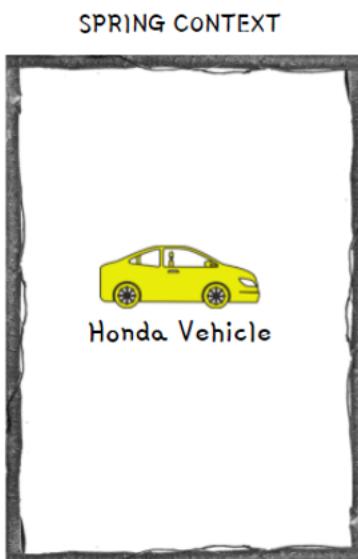
    public String getName() { return name; }

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

    @PreDestroy
    public void destroy() {
        System.out.println(
            "Destroying Vehicle Bean");
    }
}

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

}
```



```
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();
context.close();
```

## Output on Console

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

# ADDING NEW BEANS PROGRAMMATICALLY

eazy  
bytes

- 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.

```
context.registerBean("volkswagen", Vehicle.class, volkswagenSupplier);
```

The name we want to give to the bean that we add to the Spring context

The supplier returning the object instance that we want to add to the Spring Context

The ApplicationContext instance object

Type of the Bean we are creating

This diagram illustrates the parameters of the `registerBean` method. It shows the code `context.registerBean("volkswagen", Vehicle.class, volkswagenSupplier);`. Four blue arrows point from text labels to specific parts of the code:

- An arrow points from "The name we want to give to the bean that we add to the Spring context" to the string "volkswagen".
- An arrow points from "The supplier returning the object instance that we want to add to the Spring Context" to the variable `volkswagenSupplier`.
- An arrow points from "Type of the Bean we are creating" to the class `Vehicle.class`.
- An arrow points from "The ApplicationContext instance object" to the variable `context`.

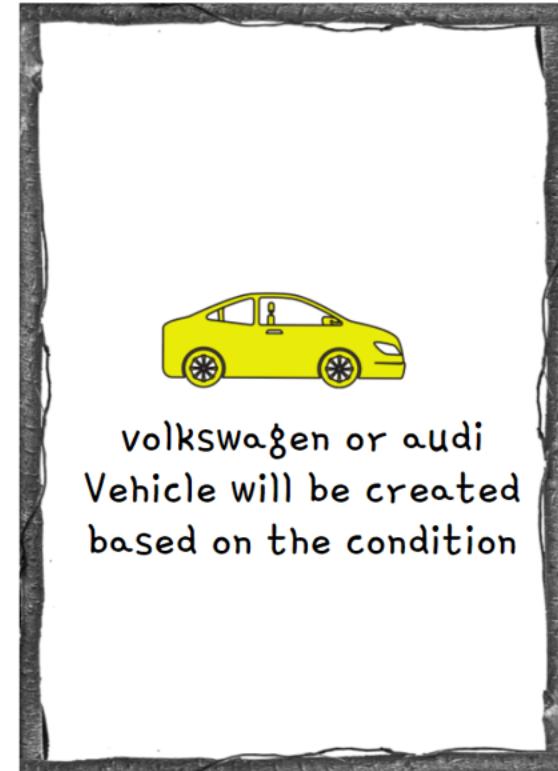
# ADDING NEW BEANS PROGRAMMATICALLY

eazy  
bytes

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



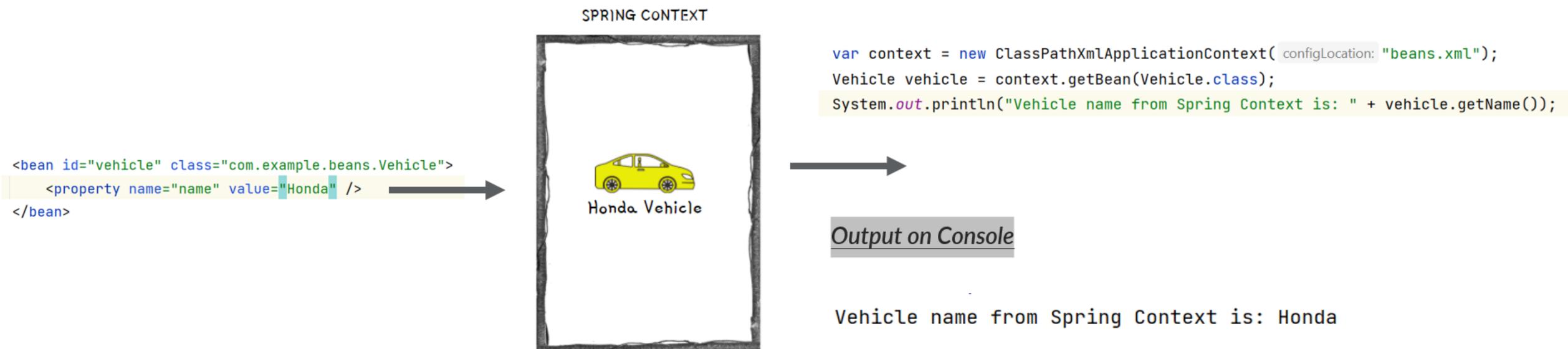
SPRING CONTEXT



# ADDING NEW BEANS USING XML CONFIGS

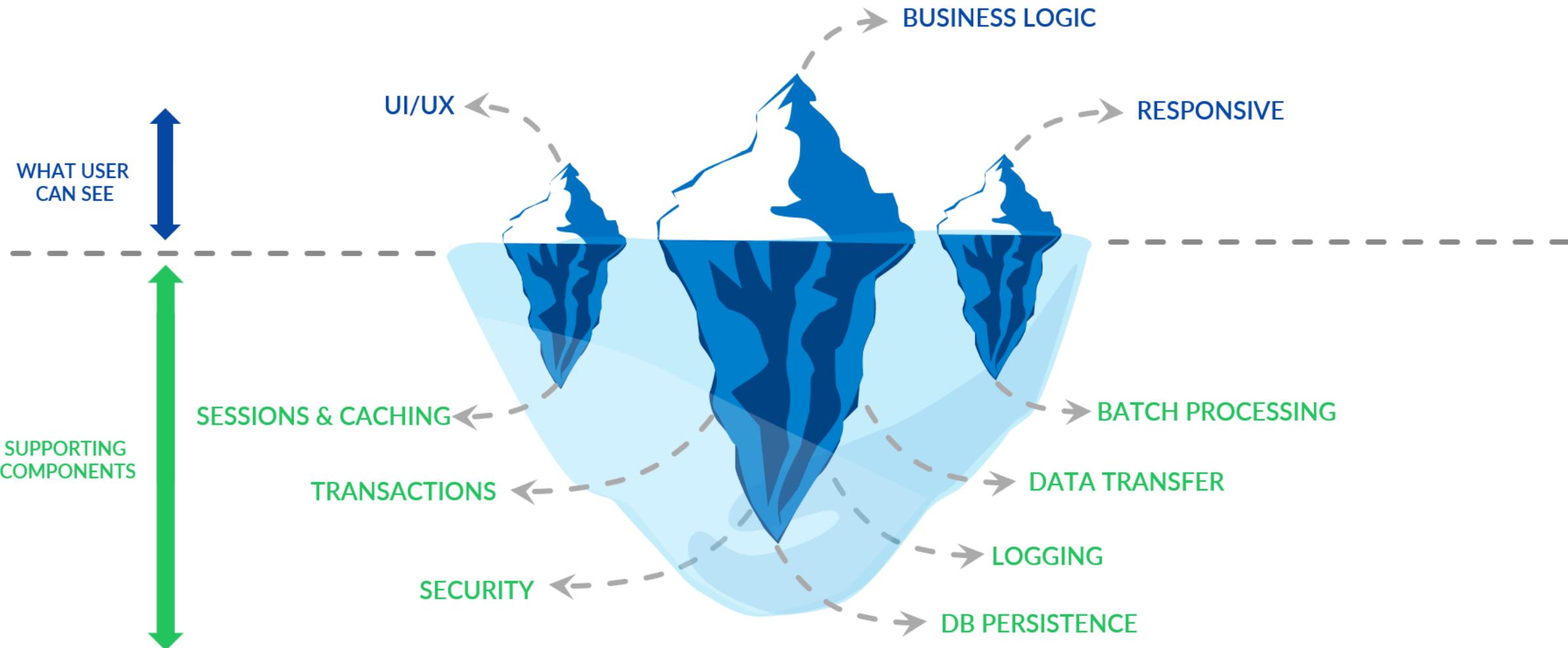
eazy  
bytes

- 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.



# BEHIND THE SCENES OF A WEB APP

eazy  
bytes



# WHY SHOULD WE USE FRAMEWORKS?

eazy  
bytes



## CHEF VICKY

Uses best readily available best ingredients like Cheese, Pizza Dough etc. to prepare Pizza



Pizza preparation time is less



Can easily scale his restaurant pizza orders



Gets consistent taste for his pizzas



Focus more on the pizza preparation



Less efforts and more results/revenue



## CHEF SANJEEV

Prepare all the ingredients like Cheese, Pizza Dough etc. by himself to prepare Pizza



Pizza preparation time is more



Scaling his restaurant pizza orders is not an option



May not get a consistent taste for his pizzas



Focus more on the raw material & ingredients



More efforts and less results/revenue

# WHY SHOULD WE USE FRAMEWORKS?

eazy  
bytes



## 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

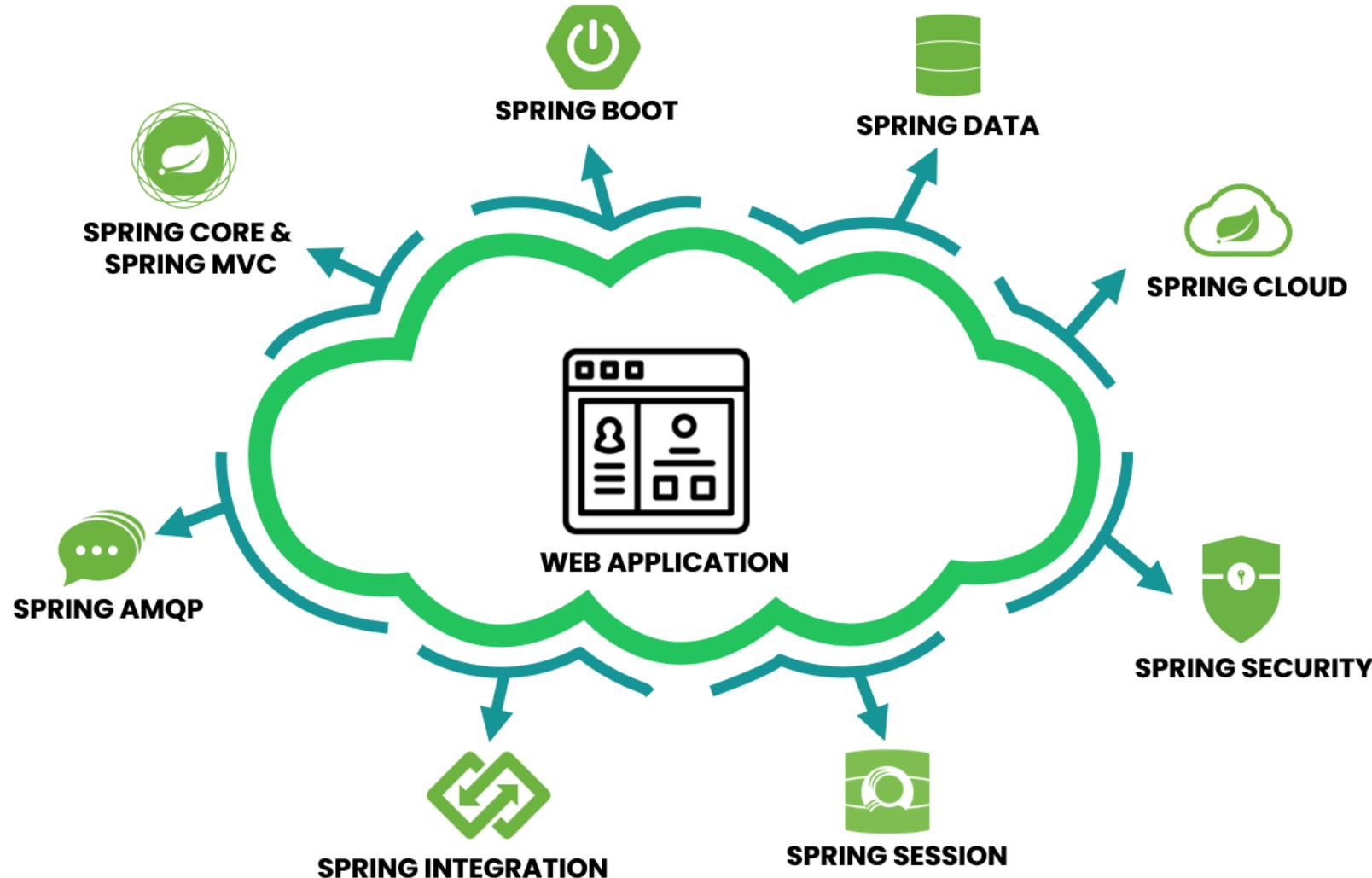


Focus more on the supporting components



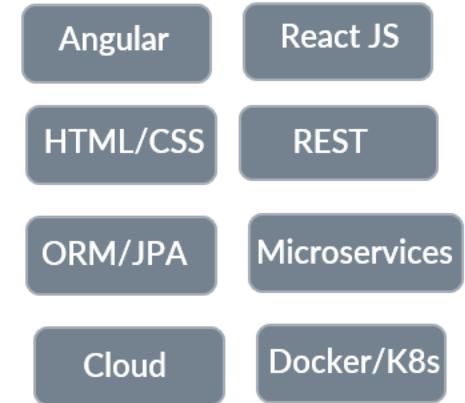
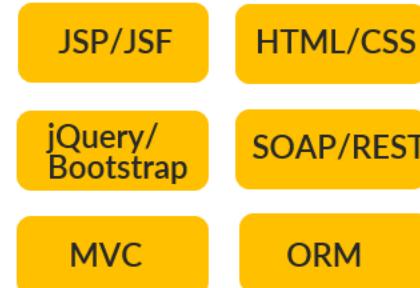
More efforts and less results/revenue

# SPRING PROJECTS



\* These projects are few important projects from Spring but not a complete list.

# SPRING PROJECTS



EVOLUTION OF APPLICATION DEVELOPMENT OVER YEARS



SPRING CORE &  
SPRING MVC



SPRING BOOT



SPRING DATA



SPRING SECURITY



INTEGRATION



SPRING CLOUD



Spring for  
Apache Kafka



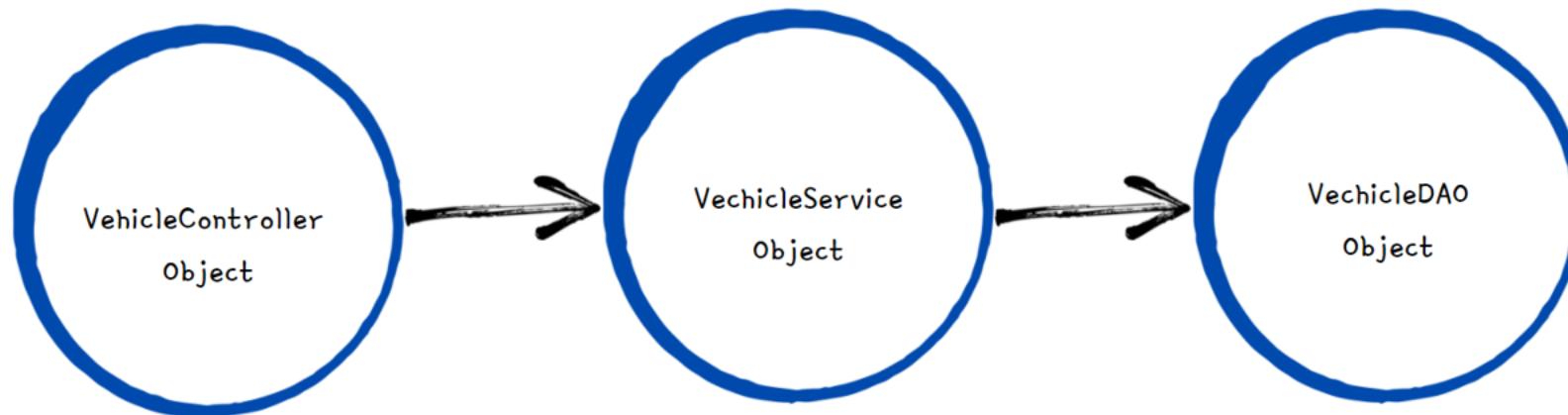
SPRING CLOUD DATA FLOW

EVOLUTION OF SPRING FRAMEWORK OVER YEARS

# INTRODUCTION TO BEANS WIRING INSIDE SPRING

eazy  
bytes

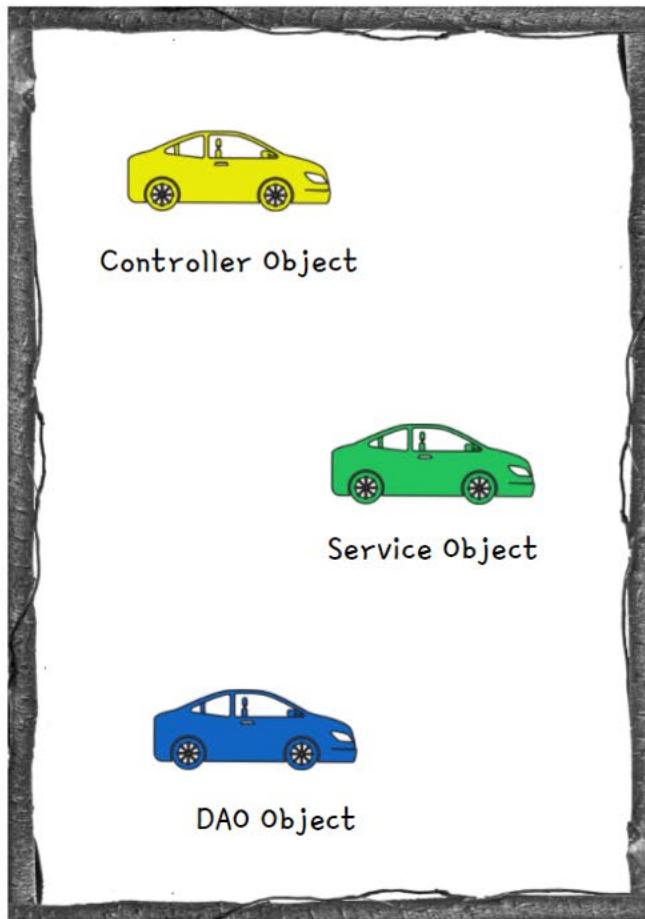
- Inside Java web applications, usually the objects delegate certain responsibilities to other objects. So in this scenarios, objects will have dependency on others.
- In very similar lines when we create various beans using Spring, it our responsibility to understand the dependencies that beans have and wire them. This concept inside is called **Wiring/Autowiring**.



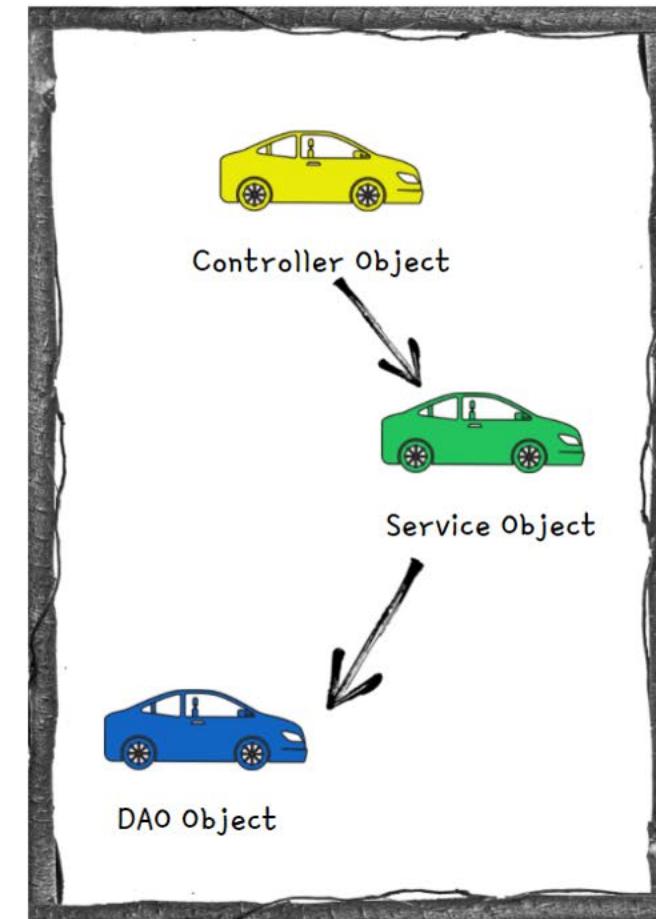
# INTRODUCTION TO BEANS WIRING INSIDE SPRING

eazy  
bytes

SPRING CONTEXT WITH OUT  
WIRING



SPRING CONTEXT WITH  
WIRING & DI



# NO WIRING SCENARIO INSIDE SPRING

eazy  
bytes

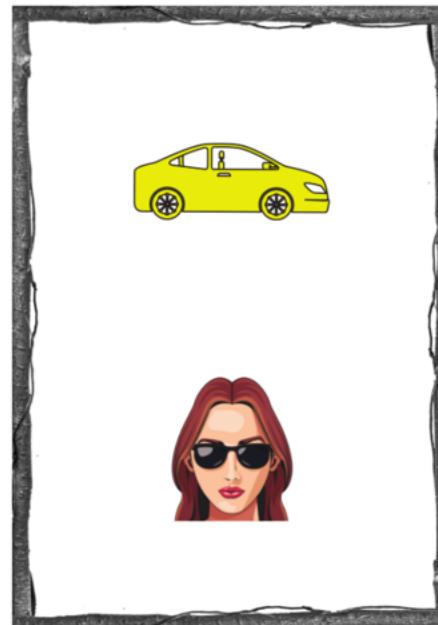
Consider a scenario where we have two java classes Person and Vehicle. The Person class has a dependency on the Vehicle. Based on the below code, we are only creating the beans inside the Spring Context and no wiring will be done. Due to this both this beans present inside the Spring context with out knowing about each other.

```
public class Vehicle {  
  
    private String name;
```

```
public class Person {  
  
    private String name;  
    private Vehicle vehicle;
```

```
@Bean  
public Vehicle vehicle() {  
    Vehicle vehicle = new Vehicle();  
    vehicle.setName("Toyota");  
    return vehicle;  
}  
  
@Bean  
public Person person() {  
    Person person = new Person();  
    person.setName("Lucy");  
    return person;  
}
```

SPRING CONTEXT



Vehicle doesn't belong to any Person.  
The Person and Vehicle beans are present in context but no relation established.

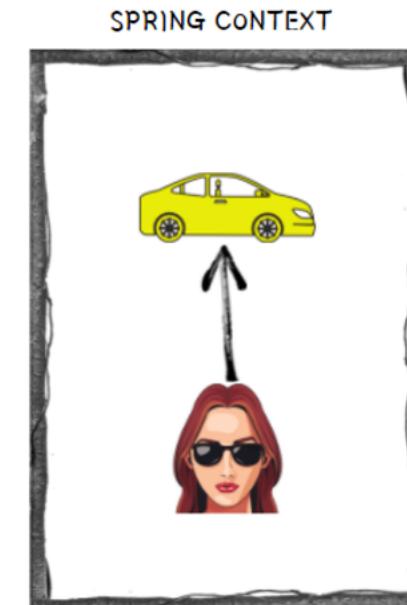
# WIRING BEANS USING METHOD CALL

eazy  
bytes

- Here in the below code, we are trying to wire or establish a relationship between Person and Vehicle, by invoking the vehicle() bean method from person() bean method. Now inside Sprint Context, person owns the vehicle.
- Spring will make sure to have only 1 vehicle bean is created and also vehicle bean will be created first always as person bean has dependency on it.

```
@Bean
public Vehicle vehicle() {
    Vehicle vehicle = new Vehicle();
    vehicle.setName("Toyota");
    return vehicle;
}

@Bean
public Person person() {
    Person person = new Person();
    person.setName("Lucy");
    person.setVehicle(vehicle());
    return person;
}
```



```
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

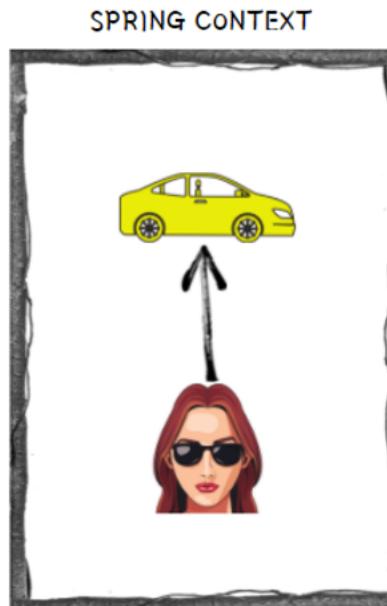
```
Vehicle bean created by Spring
Person bean created by Spring
Person name from Spring Context is: Lucy
Vehicle name from Spring Context is: Toyota
Vehicle that Person own is: Vehicle name is - Toyota
```

# WIRING BEANS USING METHOD PARAMETERS

- Here in the below code, we are trying to wire or establish a relationship between Person and Vehicle, by passing the vehicle as a method parameter to the person() bean method. Now inside Sprint Context, person owns the vehicle.
- Spring injects the vehicle bean to the person bean using Dependency Injection
- Spring will make sure to have only 1 vehicle bean is created and also vehicle bean will be created first always as person bean has dependency on it.

```
@Bean
public Vehicle vehicle() {
    Vehicle vehicle = new Vehicle();
    vehicle.setName("Toyota");
    return vehicle;
}

/*
@Bean
public Person person(Vehicle vehicle) {
    Person person = new Person();
    person.setName("Lucy");
    person.setVehicle(vehicle);
    return person;
}
```



```
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());
```

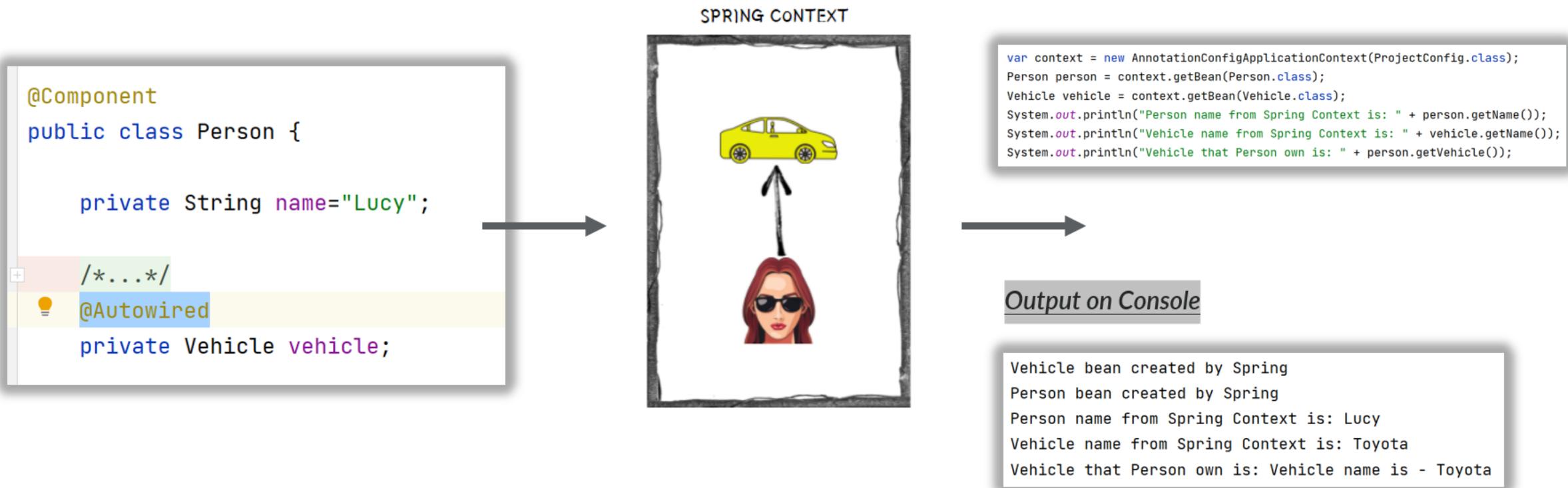


## Output on Console

```
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
```

# Inject Beans using @Autowired on class fields

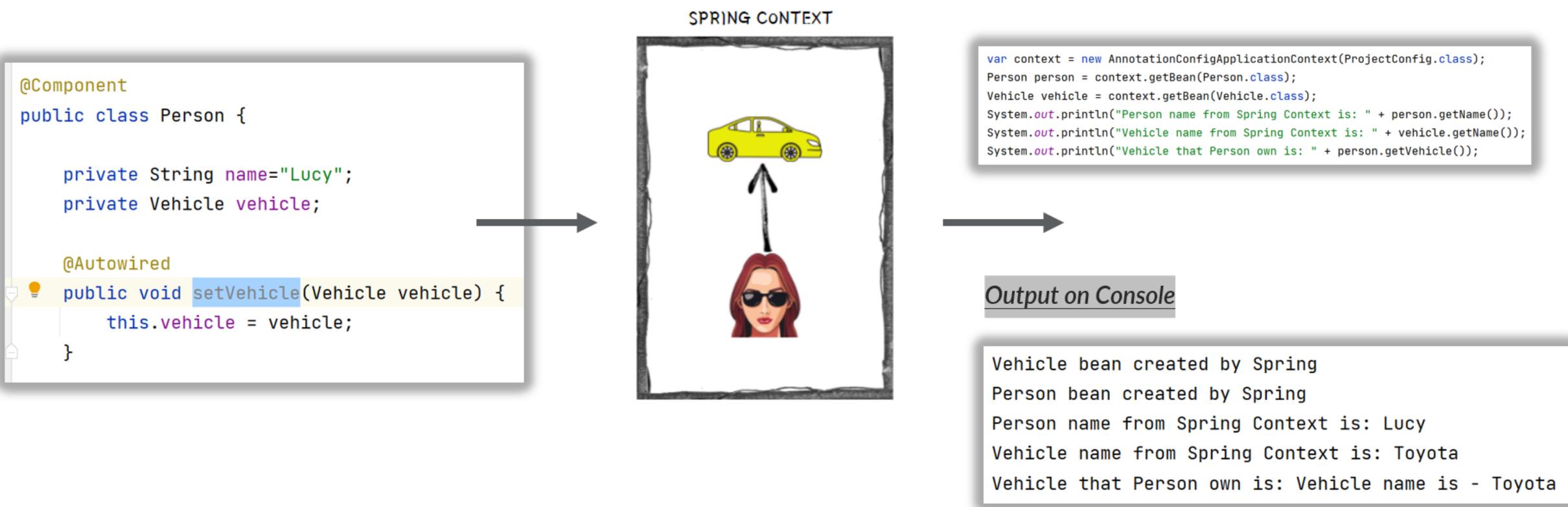
- The `@Autowired` annotation marks on a field, setter method, constructor is used to auto-wire the beans that is 'injecting beans'(Objects) at runtime by Spring Dependency Injection mechanism.
- With the below code, Spring injects/auto-wire the vehicle bean to the person bean through a class field and dependency injection.
- The below style is not recommended for production usage as we can't mark the fields as final.



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

# 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.



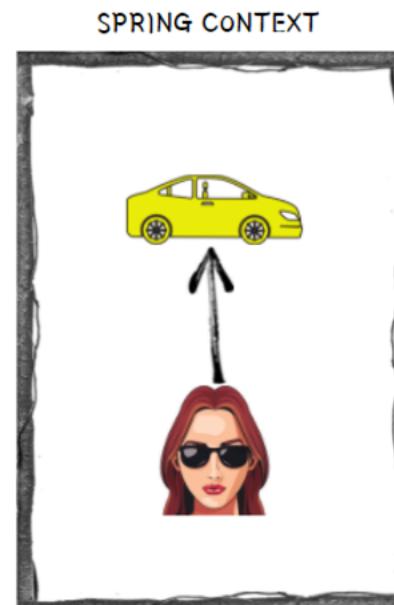
# Inject Beans using `@Autowired` with constructor

- 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
```

# How Autowiring works with multiple Beans of same type

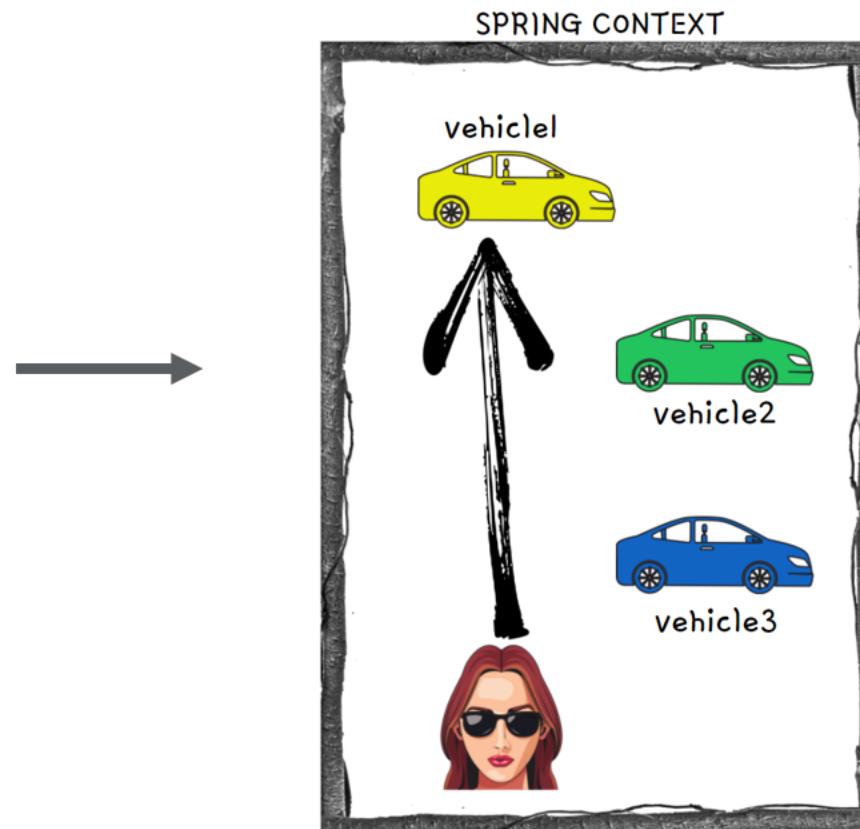
- By default Spring tries autowiring with class type. But this approach will fail if the same class type has multiple beans.
- If the Spring context has multiple beans of same class type like below, then Spring will try to auto-wire based on the parameter name/field name that we use while configuring 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



# How Autowiring works with multiple Beans of same type

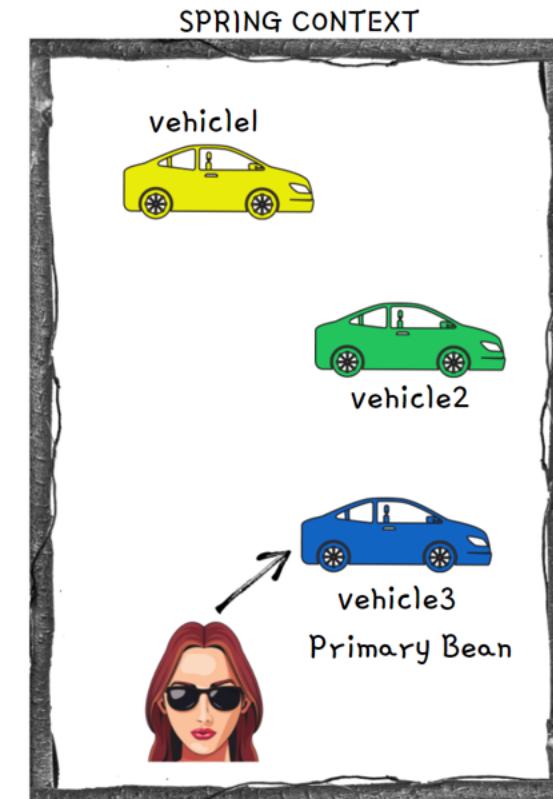
- If the parameter name/field name that we use while configuring autowiring annotation is not matching with any of the bean names, then Spring will look for the bean which has @Primary configured.
- In the below scenario, we used 'vehicle' as constructor parameter. Spring will try to auto-wire with the 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

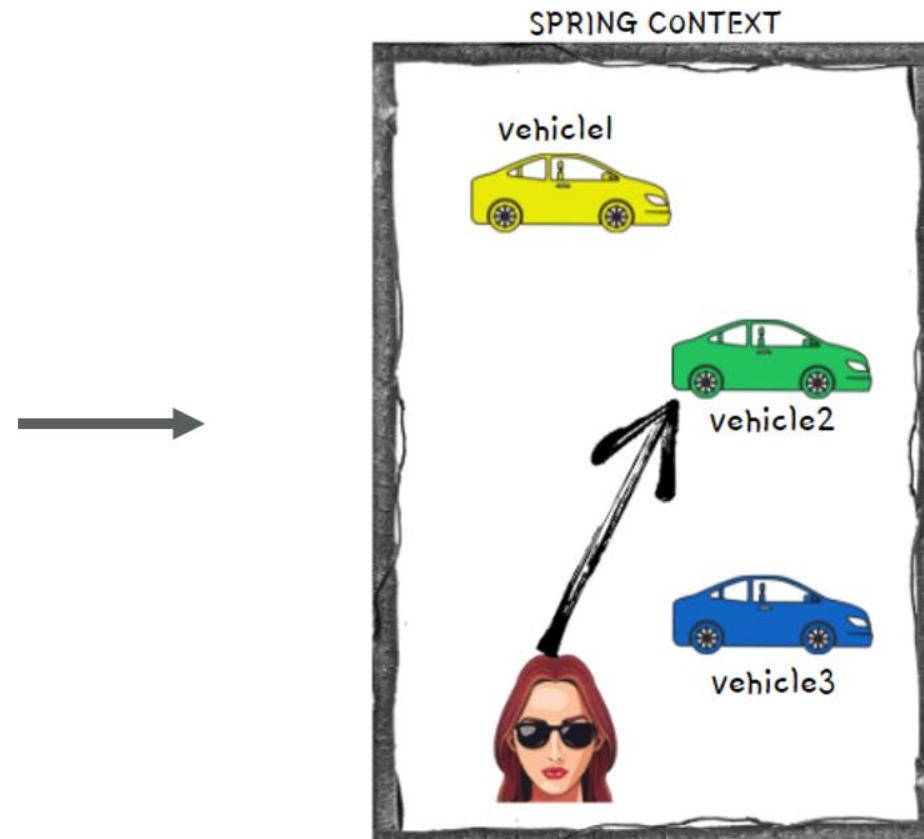


# How Autowiring works with multiple Beans of same type

- 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.

```
@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;  
    }  
}
```

STEP 3



# 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.

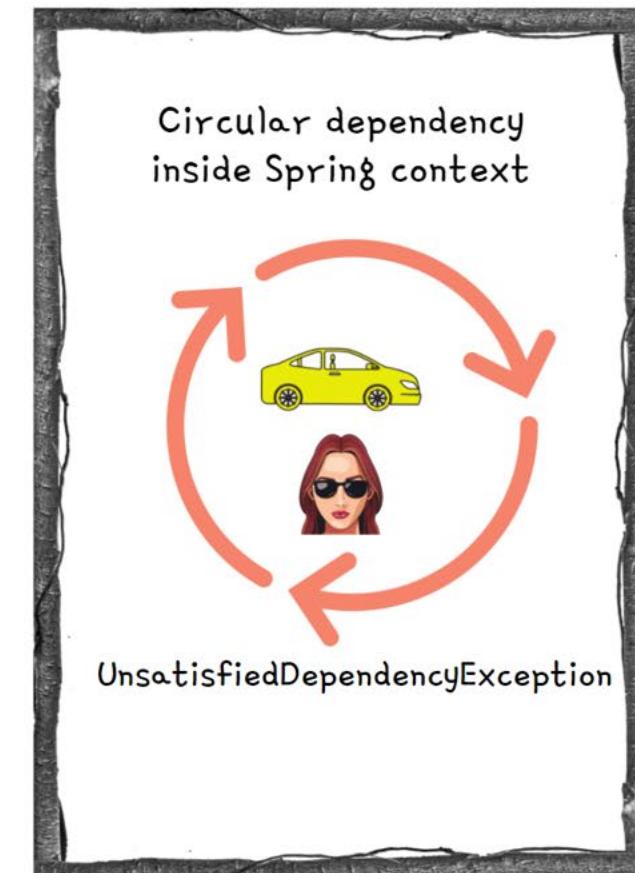
```
@Component
public class Person {

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

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

```
public class Vehicle {

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



# ASSIGNMENT RELATED TO BEANS, AUTOWIRING, DI



Person Bean has a dependency on Vehicle Bean



Your application should play music from one of the Speakers implementations & move using one of the Tyres implementation. It should also give flexibility to switch between the implementations easily.

Vehicle Bean has a dependency on VehicleServices Bean, to play music and move the vehicle

Speakers interface with makeSound() method



SonySpeakers Bean  
implementation of Speakers

BoseSpeakers Bean  
implementation of Speakers

VehicleServices bean depend on the implementations of Speakers and Tyres to serve vehicle bean requests.



BridgeStoneTyres Bean  
implementation of Tyres

MichelinTyres Bean  
implementation of Tyres

Tyres interface with rotate() method

# Bean Scopes inside Spring

1 Singleton

2 Prototype

3 Request

4 Session

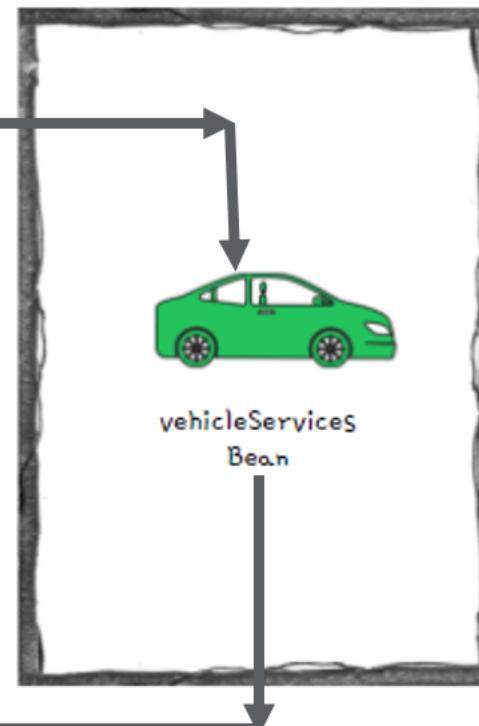
5 Application

# 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.

```
@Component  
@Scope(BeanDefinition.SCOPE_SINGLETON)  
public class VehicleServices {  
  
    VehicleServices vehicleServices1 = context.  
        getBean(VehicleServices.class);  
    VehicleServices vehicleServices2 = context.  
        getBean(name: "vehicleServices", VehicleServices.class);
```

Creates a single bean  
inside Spring context



The two variables refers  
to the same bean inside  
Spring context

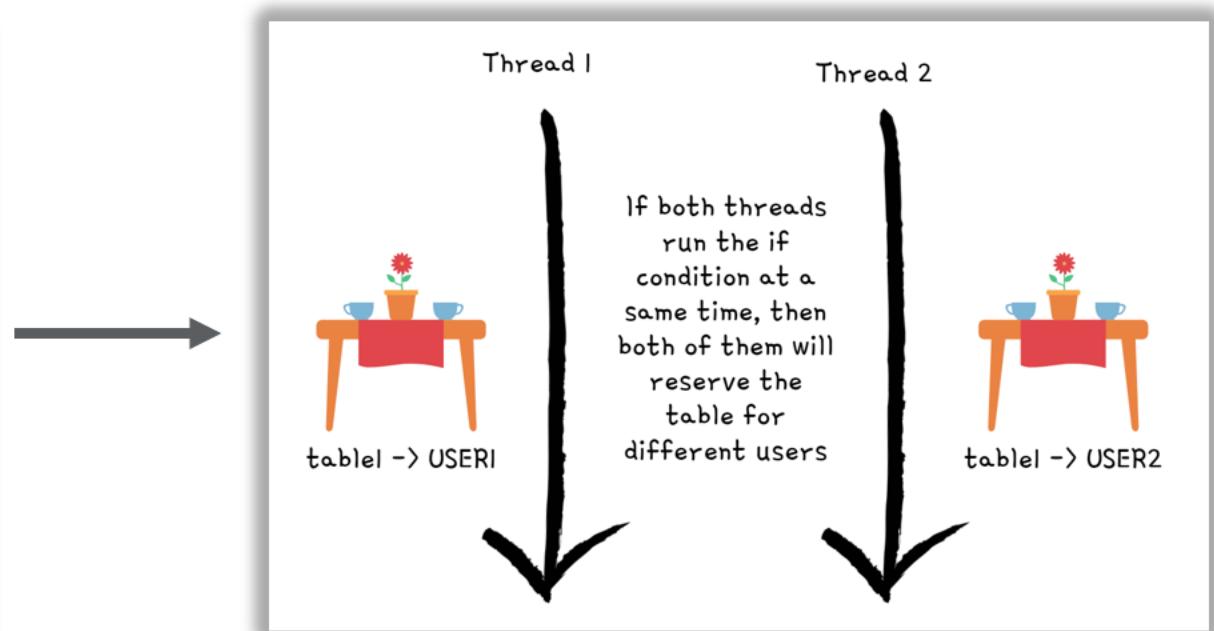
# RACE CONDITION

- 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.

```
// Shared Value inside a object
Map<String, String> reservedTables = new HashMap<>();

// thread -1
if (!reservedTables.containsKey("table1")){
    reservedTables.put("table1", "USER1");
}

// thread - 2
if (!reservedTables.containsKey("table1")){
    reservedTables.put("table1", "USER2");
}
```



# 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

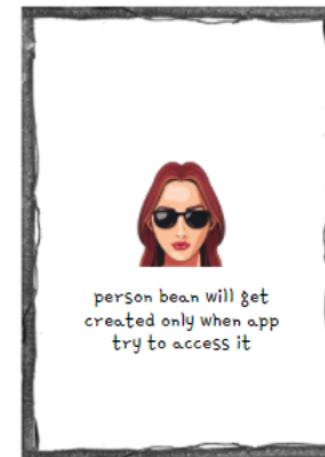
But it is not recommended, since it brings lot of complexity and performance issues inside your app. So please don't try to build mutable singleton beans.

# EAGER & LAZY INSTANTIATION

- 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.

## Sample demo of Lazy instantiation

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



## Output on Console

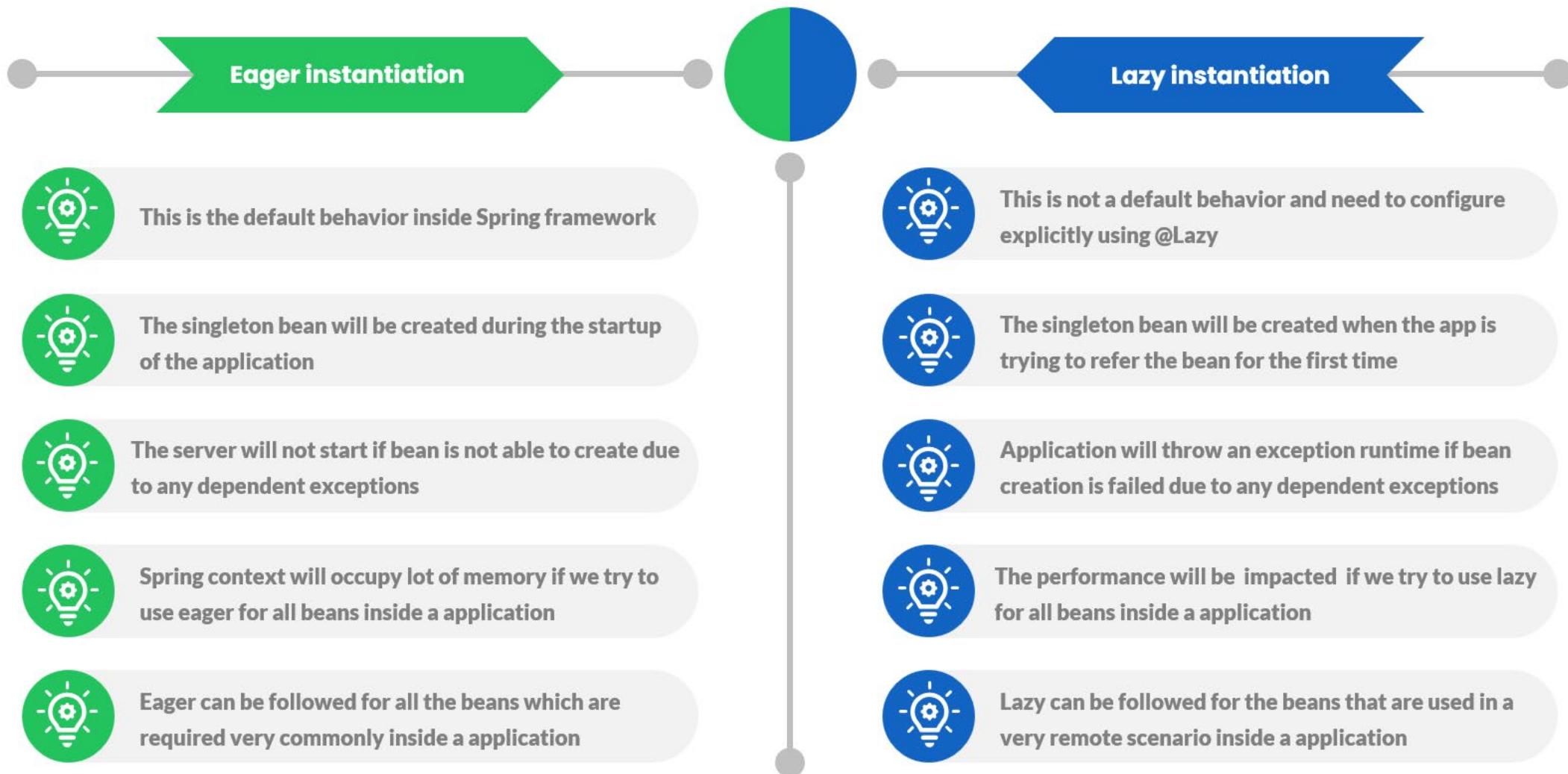
```
var context = new AnnotationConfigApplicationContext(ProjectConfig.class);
System.out.println("Before retrieving the Person bean from the Spring Context");
Person person = context.getBean(Person.class);
System.out.println("After retrieving the Person bean from the Spring Context");
```



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

# Eager Vs Lazy

eazy  
bytes

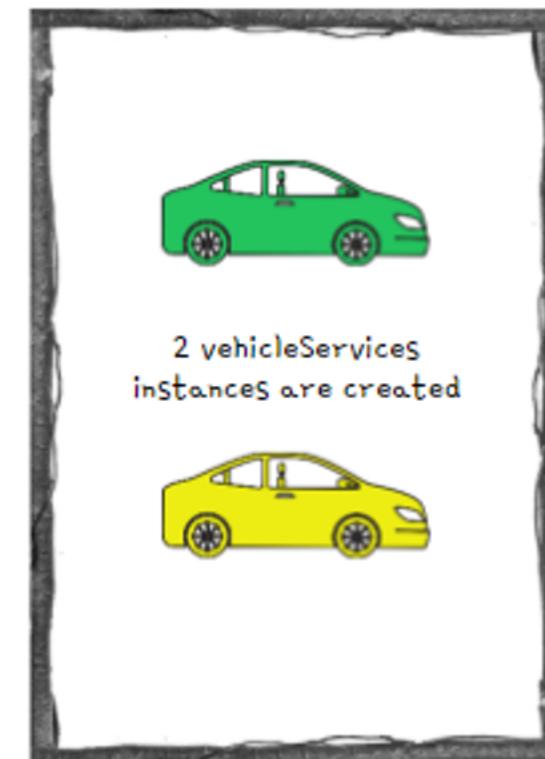


## PROTOTYPE BEAN SCOPE

- 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.

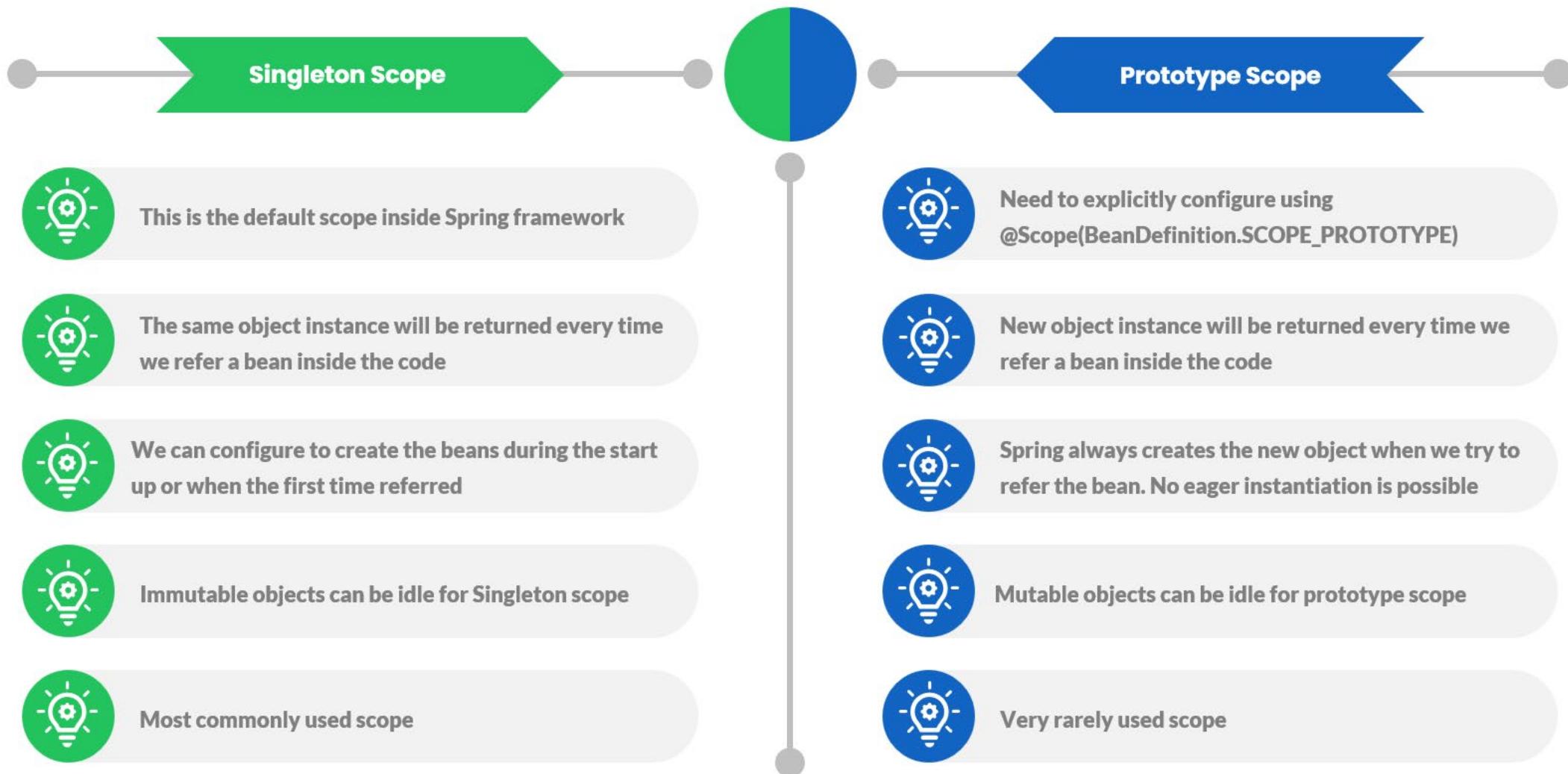
```
@Component
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class VehicleServices {

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



# Singleton Vs Prototype

eazy  
bytes



# ASPECT-ORIENTED PROGRAMMING (AOP)

## Aspect-Oriented Programming

- ✓ An aspect is simply a piece of code the Spring framework executes when you call specific methods inside your app.
- ✓ Spring AOP enables Aspect-Oriented Programming in spring applications. In AOP, aspects enable the modularization of concerns such as transaction management, logging or security that cut across multiple types and objects (often termed crosscutting concerns).

1

AOP provides the way to dynamically add the cross-cutting concern before, after or around the actual logic using simple pluggable configurations.

2

AOP helps in separating and maintaining many non-business logic related code like logging, auditing, security, transaction management.

3

AOP is a programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does this by adding additional behavior to existing code without modifying the code itself.

More details : <https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop-ataspectj>

# ASPECT-ORIENTED PROGRAMMING (AOP)

```
public String moveVehicle(boolean started){  
    Instant start = Instant.now();  
    logger.info( msg: "method execution start");  
    String status = null;  
    if(started){  
        status = tyres.rotate();  
    }else{  
        logger.log(Level.SEVERE, msg: "Vehicle not started to perform the" +  
            " operation");  
    }  
    logger.info( msg: "method execution end");  
    Instant finish = Instant.now();  
    long timeElapsed = Duration.between(start, finish).toMillis();  
    logger.info( msg: "Time took to execute the method : "+timeElapsed);  
    return status;  
}
```



```
public String moveVehicle(boolean started){  
    return tyres.rotate();  
}
```

There is so much of non business logic code along with the main business logic

With AOP magic, all the non business logic is moved to different location which will make method clean & clear

## AOP Jargons

- ✓ When we define an Aspect or doing configurations, we need to follow WWW (3 Ws)

- WHAT → Aspect
- WHEN → Advice
- WHICH → Pointcut

1

WHAT code or logic we want the Sprint to execute when you call a specific method. This is called as Aspect.

2

WHEN the Spring need to execute the given Aspect. For example is it before or after the method call. This is called as Advice.

3

WHICH method inside App that framework needs to intercept and execute the given Aspect. This is called as a Pointcut.

- Join point which defines the event that triggers the execution of an aspect. Inside Spring, this event is always a method call.
- Target object is the bean that declares the method/pointcut which is intercepted by an aspect.

## Typical Scenario of AOP implementation

Aspect



Advice



Joinpoint



Developer want some logic to be executed before each execution of method `playMusic()` present inside the bean `VehicleServices`.

Point cut

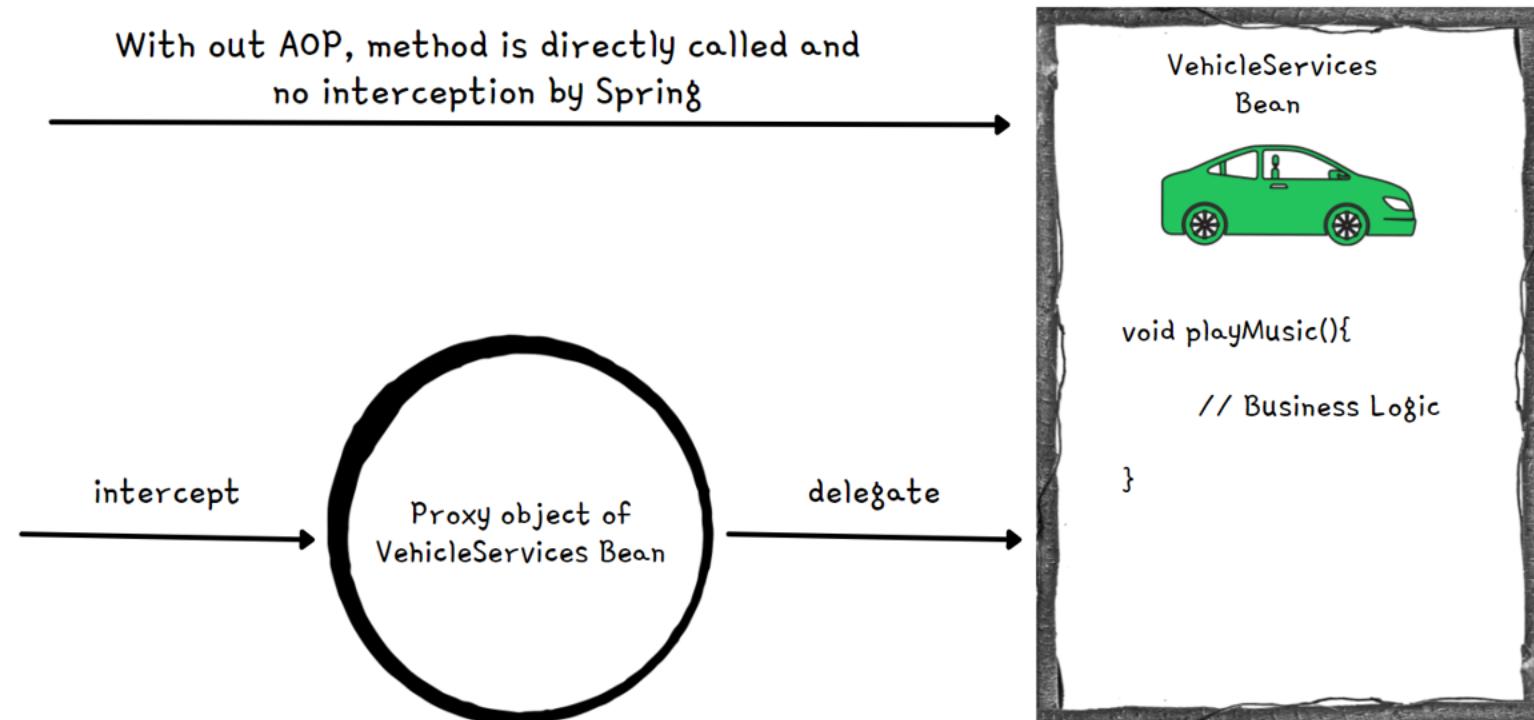


Target Object



## WEAVING INSIDE AOP

- When we are implementing AOP inside our App using Spring framework, it will intercept each method call and apply the logic defined in the Aspect.
- But how does this works ? Spring does this with the help of proxy object. So we try to invoke a method inside a bean, Spring instead of directly giving reference of the bean instead it will give a proxy object that will manage the each call to a method and apply the aspect logic. This process is called Weaving.



With AOP, method executions will be intercepted by proxy object and aspect will be executed. Post that actual method invocation will happen

## ADVICE TYPES INSIDE AOP

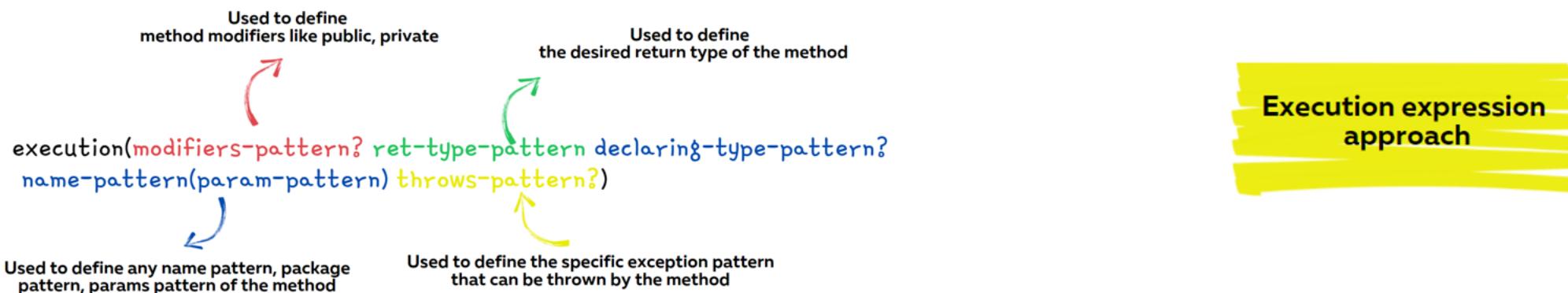
### Type of Advices in Spring AOP

- ✓ **@Before**
- ✓ **@AfterReturning**
- ✓ **@AfterThrowing**
- ✓ **@After**
- ✓ **@Around**

- 1 Before advice runs before a matched method execution.
- 2 After returning advice runs when a matched method execution completes normally.
- 3 After throwing advice runs when a matched method execution exits by throwing an exception
- 4 After (finally) advice runs no matter how a matched method execution exits.
- 5 Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method runs and to determine when, how, and even if the method actually gets to run at all.

# CONFIGURING ADVICES INSIDE AOP

- We can use the AspectJ pointcut expression to provide details to Spring about what kind of methods it needs to intercept by mentioning details around modifier, return type, name pattern, package name pattern, params pattern, exceptions pattern etc.
- Below is the format of the same,



- Like shown below, we can mention the pointcut expressions as an input after advise annotations that we use,

```
@Configuration
@ComponentScan(basePackages = {"com.example.implementation",
    "com.example.services", "com.example.aspects"})
@EnableAspectJAutoProxy
public class ProjectConfig { }
```

```
@Aspect
@Component
public class LoggerAspect {

    @Around("execution(* com.example.services.*.*(..))")
    public void log(ProceedingJoinPoint joinPoint) throws Throwable {
        // Aspect Logic
    }
}
```

# CONFIGURING ADVICES INSIDE AOP

- Alternatively, we can use Annotation style of configuring Advices inside AOP. Below are the three steps that we follow for the same,

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface LogAspect {  
}
```

Step 1: Create an annotation type

Annotations  
approach

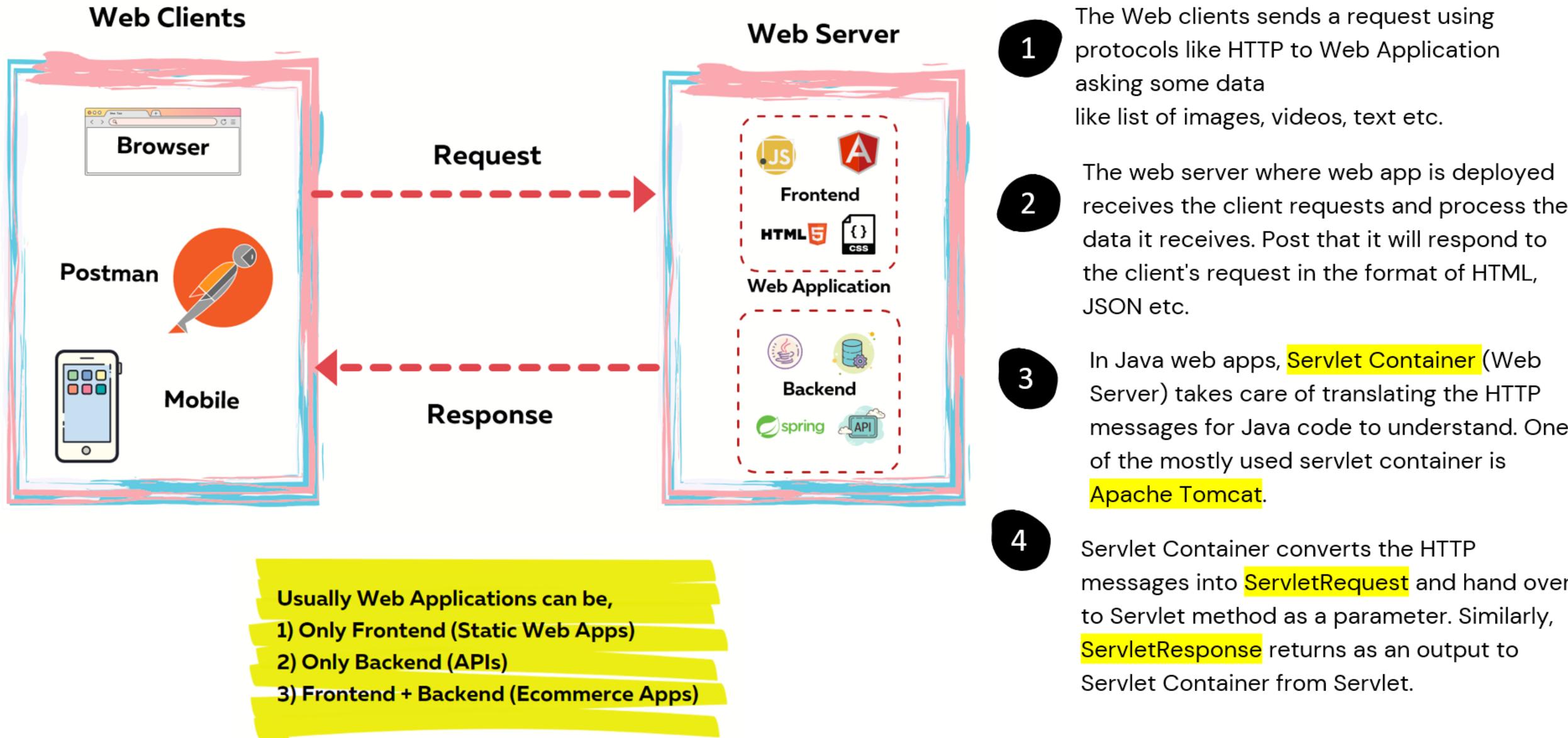
```
@LogAspect  
public String playMusic(boolean started, Song song){  
    // Business Logic  
}
```

Step 2: Mention the same annotation  
on top of the method which we want to  
intercept using AOP

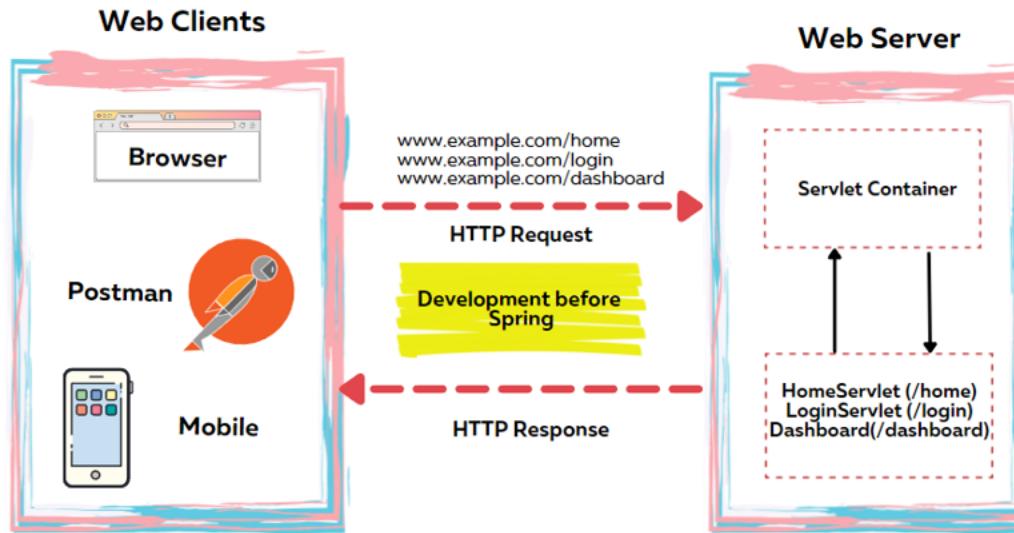
```
@Around("@annotation(com.example.interfaces.LogAspect)")  
public void logWithAnnotation(ProceedingJoinPoint joinPoint) throws Throwable {  
    // Aspect Logic  
}
```

Step 3: Use the annotation details to configure  
on top of the aspect method to advice

# OVERVIEW OF A WEB APP



# ROLE OF SERVLETS INSIDE WEB APPS

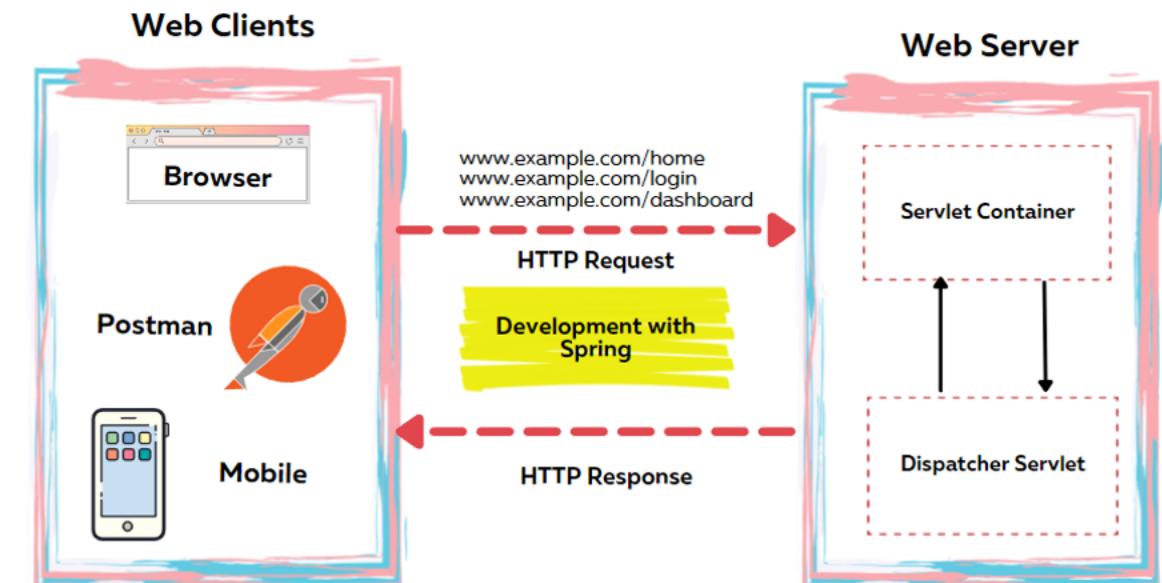


## With Spring

- 1** With Spring, it defines a servlet called **Dispatcher Servlet** which maintains all the URL mapping inside a web application.
- 2** The servlet container calls this Dispatcher Servlet for any client request, allowing the servlet to manage the request and the response. This way Spring internally does all the magic for Developers without the need of defining the servlets inside a

## Before Spring

- 1** Before Spring, developer has to create a new servlet instance, configure it in the servlet container, and assign it to a specific URL path
- 2** When the client sends a request, Tomcat calls a method of the servlet associated with the path the client requested. The servlet gets the values on the request and builds the response that Tomcat sends back to the client.



# EVOLUTION OF WEB APPS INSIDE JAVA ECO SYSTEM

Somewhere in 2000



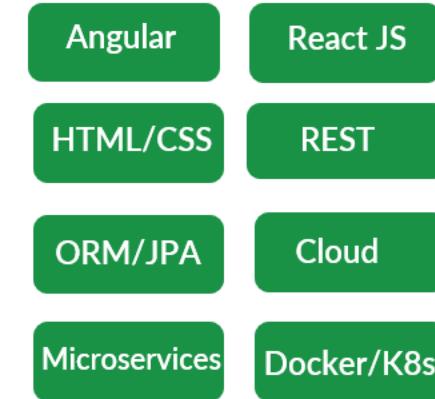
No Web Design patterns and frameworks support present in 2000s. So all the web application code is written in such a way all the layers like Presentation, Business, Data layers are tightly coupled.

Somewhere in 2010



With the help & invention of design patterns like MVC and frameworks like Spring, Struts, Hibernate, developers started building web applications separating the layers of Presentation, Business, Data layers. But all the code deployed into a single jumbo server as monolithic application.

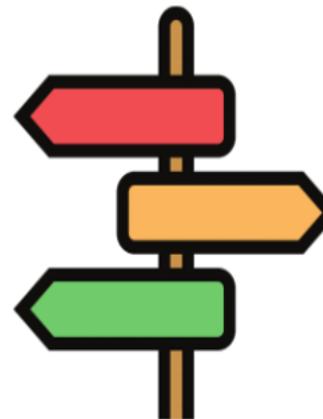
Somewhere in 2020



With the invention of UI frameworks like Angular, React and new trends like Microservices, Containers, developers started building web application by separating UI and backend layers. The code also deployed into multiple servers using containers and cloud.

## APPROACH 1

- Web Apps which holds UI elements like HTML, CSS, JS and backend logic
- Here the App is responsible to fully prepare the view along with data in response to a client request
- Spring Core, Spring MVC, SpringBoot, Spring Data, Spring Rest, Spring Security will be used



*Approaches to build web applications using Spring framework*

## APPROACH 2

- Web Apps which holds only backend logic. These Apps send data like JSON to separate UI Apps built based on Angular, React etc.
- Here the App is responsible to only process the request and respond with only data ignoring view.
- Spring Core, SpringBoot, Spring Data, Spring Rest, Spring Security will be used

*Spring MVC is the key differentiator between these two approaches.*

# SPRINGBOOT : THE HERO OF SPRING FRAMEWORK



- 1** Spring Boot was introduced in April 2014 to reduce some of the burdens while developing a Java web application.
- 2** Before SpringBoot, Developer need to configure a servlet container, establish link b/w Tomcat and Dispatcher servlet, deploy into a server, define lot of dependencies.....
- 3** But with SpringBoot, we can create Web Apps skeletons with in seconds or at least in 1-2 mins 😊. It helps eliminating all the configurations we need to do.
- 4** Spring Boot is now one of the most appreciated projects in the Spring ecosystem. It helps us to create Spring apps more efficiently and focus on the business code.
- 5** SpringBoot is a mandatory skill now due to the latest trends like Full Stack Development, Microservices, Serverless, Containers, Docker etc.

## BEFORE & AFTER SPRINGBOOT



Configure a Maven/Gradle project with all the dependencies needed



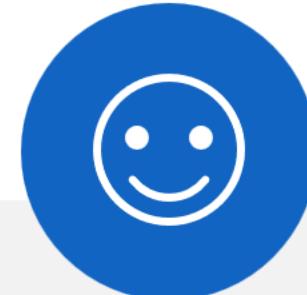
Understand how servlets work & configure the DispatcherServlet inside web.xml



Package the web application into a WAR file. Deploy the same into a server



Deal with complicated class loading strategies, application monitoring and management



Spring boot automatically configures the bare minimum components of a Spring application.



Spring Boot applications embed a web server so that we do not require an external application server.



Spring Boot provides several useful production-ready features out of the box to monitor and manage the application

## Quick Recap

- 1 <https://start.spring.io/> is the website which can be used to generate web projects skeleton based on the dependencies required for an application.
- 2 We can identify the Spring Boot main class by looking for an annotation `@SpringBootApplication`.
- 3 A single `@SpringBootApplication` annotation can be used to enable those three features, that is:
  - `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism
  - `@ComponentScan`: enable `@Component` scan on the package where the application is located
  - `@SpringBootConfiguration`: enable registration of extra beans in the context or the import of additional configuration classes. An alternative to Spring's standard `@Configuration` annotation



## Quick Recap

4

The `@RequestMapping` annotation provides “routing” information. It tells Spring that any HTTP request with the given path should be mapped to the corresponding method. It is a Spring MVC annotation and not specific to Spring Boot.

5

`server.port` and `server.servlet.context-path` properties can be mentioned inside the `application.properties` to change the default port number and context path of a web application.

6

Mentioning `server.port=0` will start the web application at a random port number every time.

7

Mentioning `debug=true` will print the Autoconfiguration report on the console. We can mention the exclusion list as well for SpringBoot auto-configuration by using the below config,

```
@SpringBootApplication(exclude = { DataSourceAutoConfiguration.class })
```



## QUICK TIP

Do you know, we can configure multiple paths against a single method using Spring MVC annotations ?

```
@Controller
public class HomeController {

    @RequestMapping(value={"/", "home"})
    public String displayHomePage(Model model) {
        // Business Logic
    }
}
```



# INTRODUCTION TO THYMELEAF

- Thymeleaf is a modern server-side Java template engine for both web and standalone environments. This allow developers to build dynamic content inside the web applications.
- Thymeleaf has great integration with Spring especially with Spring MVC, Spring Security etc.
- The Thymeleaf + Spring integration packages offer a `SpringResourceTemplateResolver` implementation which uses all the Spring infrastructure for accessing and reading resources in applications, and which is the recommended implementation in Spring-enabled applications.

```
<table>
    <thead>
        <tr>
            <th th:text="#{msgs.headers.name}">Name</th>
            <th th:text="#{msgs.headers.price}">Price</th>
        </tr>
    </thead>
    <tbody>
        <tr th:each="prod: ${allProducts}">
            <td th:text="${prod.name}">Oranges</td>
            <td th:text="#{numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
        </tr>
    </tbody>
</table>
```



For more details, pls refer <https://www.thymeleaf.org/>

# SPRINGBOOT DEVTOOLS

- The Spring Boot DevTools provides features like Automatic restart & LiveReload that make the application development experience a little more pleasant for developers.
- It can be added into any of the SpringBoot project by adding the below maven dependency,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

DevTools maintains 2 class loaders. one with classes that doesn't change and other one with classes that change. When restart needed it only reload the second class loader which makes restarts faster as well.

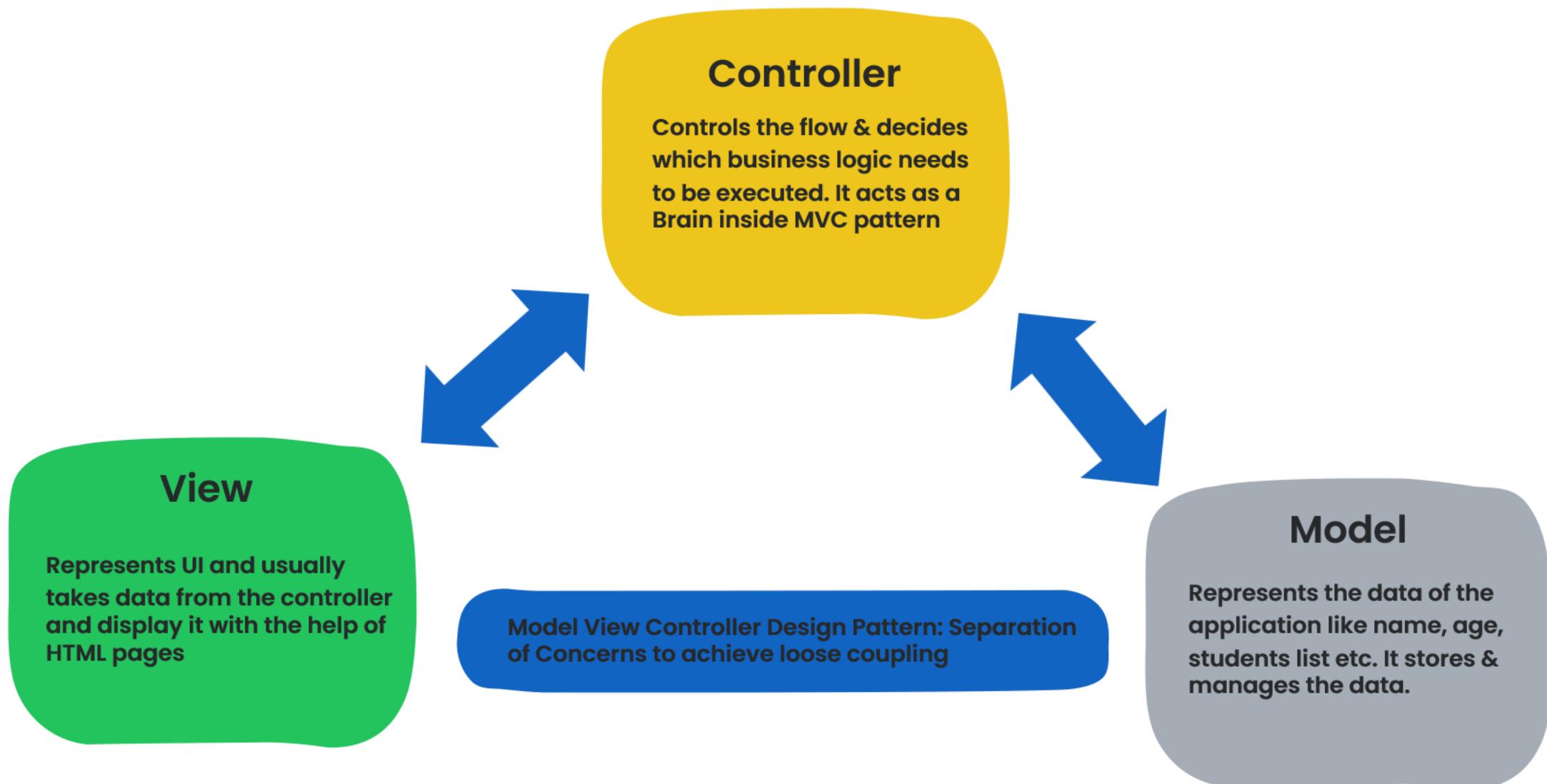
DevTools includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed. LiveReload related browser extensions are freely available for Chrome, Firefox

DevTools triggers a restart when ever a build is triggered through IDE or by maven commands etc.

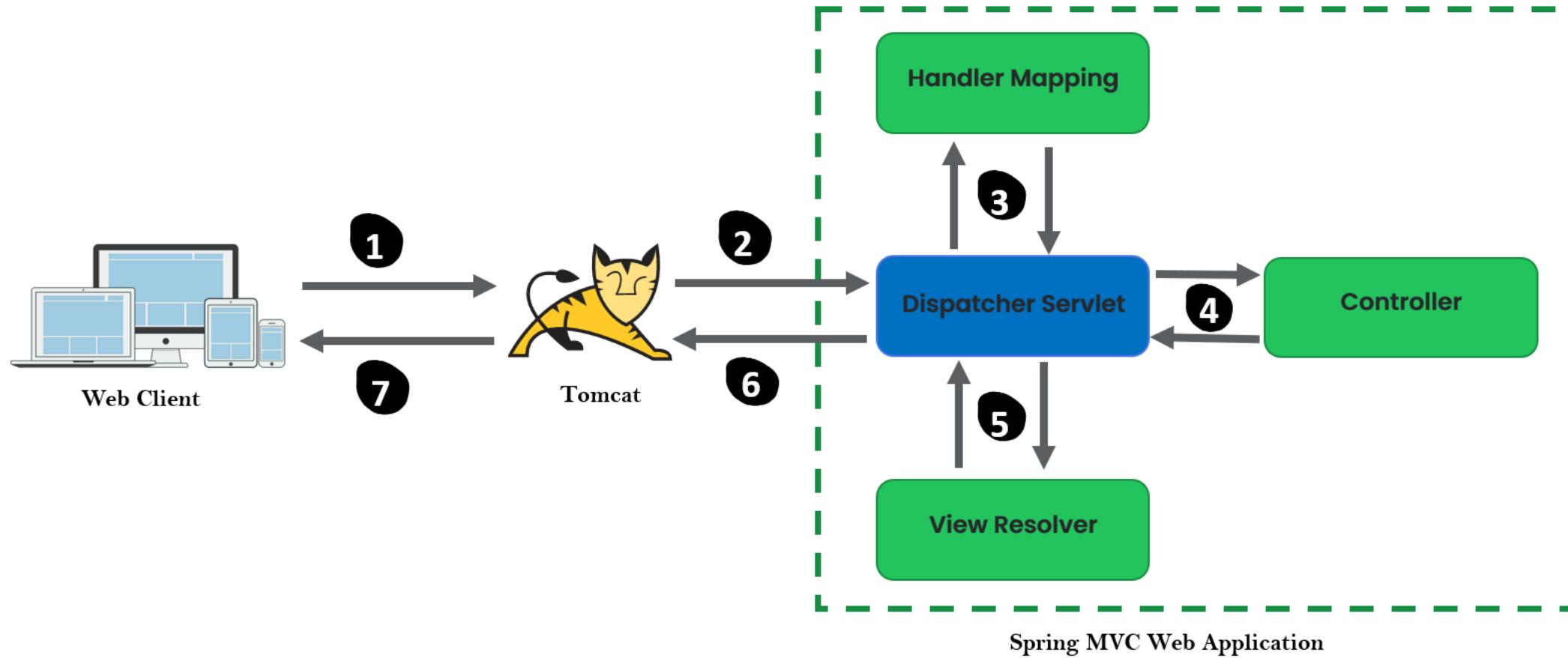
DevTools disables the caching options by default during development.

Repackaged archives do not contain DevTools by default.

# INTRODUCTION TO MVC PATTERN



# SPRING MVC ARCHITECTURE & INTERNAL FLOW



- 1 Web Client makes HTTP request
- 2 Servlet Containers like Tomcat accepts the HTTP requests and handovers the Servlet Request to **Dispatcher Servlet** inside Spring Web App.
- 3 The Dispatcher Servlet will check with the **Handler Mapping** to identify the controller and method names to invoke based on the HTTP method, path etc.
- 4 The Dispatcher Servlet will invoke the corresponding controller & method. After execution, the controller will provide a view name and data that needs to be rendered in the view.
- 5 The Dispatcher Servlet with the help of a component called **View Resolver** finds the view and render it with the data provided by the controller.
- 6 The Servlet Container or Tomcat accepts the Servlet Response from the Dispatcher servlet and convert the same to HTTP response before returning to the client.
- 7 The browser or client intercepts the HTTP response and display the view, data etc.

## QUICK TIP

Do you know, we can register view controllers that create a direct mapping between the URL and the view name using the `ViewControllerRegistry`. This way, there's no need for any `Controller` between the two.

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController(urlPathOrPattern: "/courses").setViewName("courses");
        registry.addViewController(urlPathOrPattern: "/about").setViewName("about");
    }

}
```



# REDUCING BOILERPLATE CODE WITH LOMBOK

- Java expects lot of boilerplate code inside POJO classes like getters and setters.
- Lombok, which is a Java library provides you with several annotations aimed at avoiding writing Java code known to be repetitive and/or boilerplate.
- It can be added into any of the Java project by adding the below maven dependency,

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

Project Lombok works by plugging into your build process. Then, it will auto-generate the Java bytecode into your .class files required to implement the desired behavior, based on the annotations you used.

Most commonly used Lombok annotations,

`@Getter, @Setter  
@NoArgsConstructor  
@RequiredArgsConstructor  
@AllArgsConstructor  
@ToString, @EqualsAndHashCode  
@Data`

`@Data` is a shortcut annotation that combines the features of below annotations together,

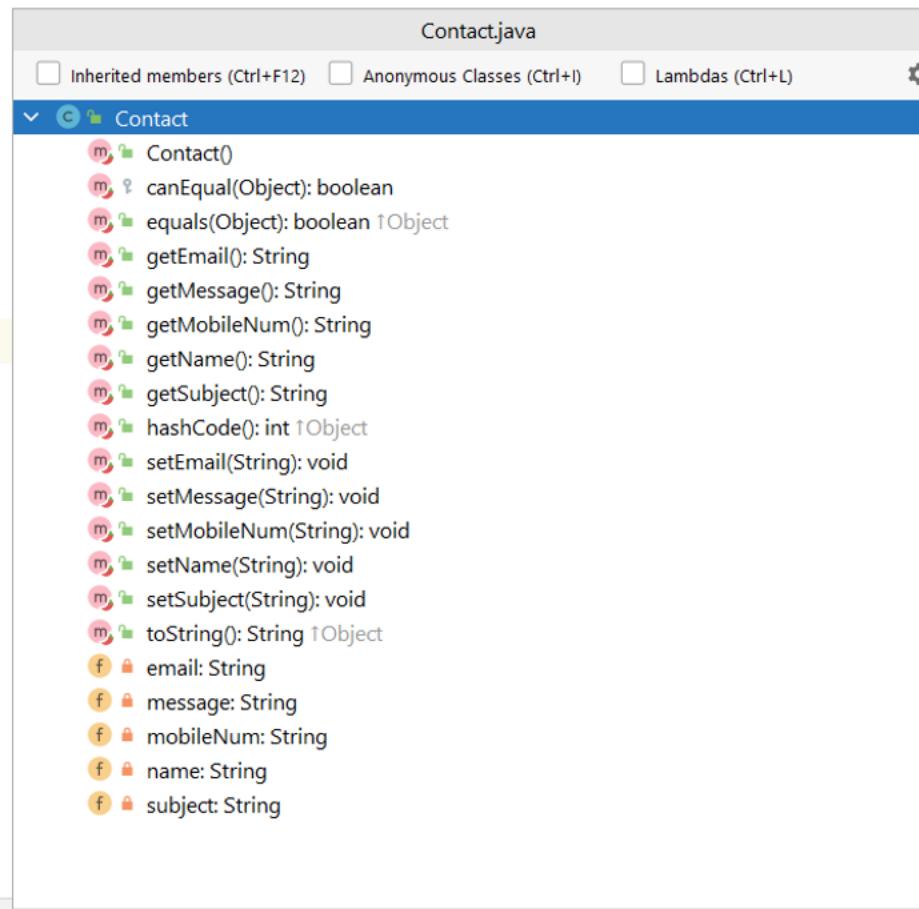
`@ToString  
@EqualsAndHashCode  
@Getter, @Setter  
@RequiredArgsConstructor`

# REDUCING BOILERPLATE CODE WITH LOMBOK

```
/*
@Data annotation is provided by Lombok library which generates getter, setter,
equals(), hashCode(), toString() methods & Constructor at compile time.
This makes our code short and clean.

*/
@Data
public class Contact {

    private String name;
    private String mobileNum;
    private String email;
    private String subject;
    private String message;
}
```



A sample outline of a Java POJO class with @Data annotation used. The source code will not have boilerplate code but the compiled byte code will have.

# @RequestParam Annotation

- In Spring @RequestParam annotation is used to map either query parameters or form data.
- For example, if we want to get parameters value from a HTTP GET requested URL then we can use @RequestParam annotation like in below example.

http://localhost:8080/holidays?festival=true&federal=true

```
@GetMapping("/holidays")
public String displayHolidays(@RequestParam(required = false) boolean festival,
                               @RequestParam(required = false) boolean federal) {
    // Business Logic
    return "holidays.html";
}
```

✓ The @RequestParam annotation supports attributes like **name**, **required**, **value**, **defaultValue**. We can use them in our application based on the requirements.

✓ The **name** attribute indicates the name of the request parameter to bind to.  
✓ The **required** attribute is used to make a field either optional or mandatory. If it is mandatory, an exception will throw in case of missing fields.

✓ The **value** attribute is similar to **name** elements and can be used as an alias.  
✓ **defaultValue** for the parameter is to handle missing values or null values. If the parameter does not contain any value then this default value will be considered.

## @PathVariable Annotation

- The `@PathVariable` annotation is used to extract the value from the URL. It is most suitable for the RESTful web service where the URL contains some value. Spring MVC allows us to use multiple `@PathVariable` annotations in the same method.
- For example, if we want to get the value from a requested URI path, then we can use `@PathVariable` annotation like in below example.

http://localhost:8080/holidays/all  
http://localhost:8080/holidays/federal  
http://localhost:8080/holidays/festival

```
@GetMapping("/holidays/{display}")
public String displayHolidays(@PathVariable String display) {
    //Business Logic
    return "holidays.html";
}
```

✓ The `@PathVariable` annotation supports attributes like `name`, `required`, `value` similar to `@RequestParam`. We can use them in our application based on the requirements.

# VALIDATION WITH SPRING BOOT

- Bean Validation (<https://beanvalidation.org/>) is the standard for implementing validations in the Java ecosystem. It's well integrated with Spring and Spring Boot.
- Below is the maven dependency that we can add to implement Bean validations in any Spring/SpringBoot project,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Bean Validation works by defining constraints to the fields of a class by annotating them with certain annotations.

We can put the `@Valid` annotation on method parameters and fields to tell Spring that we want a method parameter or field to be validated.

Below are the important packages where validations related annotations can be identified,

- ✓ `jakarta.validation.constraints.*`
- ✓ `org.hibernate.validator.constraints.*`

# Important Validation Annotations

`jakarta.validation.constraints.*`

- ✓ `@Digits`
- ✓ `@Email`
- ✓ `@Max`
- ✓ `@Min`
- ✓ `@NotBlank`
- ✓ `@NotEmpty`
- ✓ `@NotNull`
- ✓ `@Pattern`
- ✓ `@Size`

`org.hibernate.validator.constraints.*`

- ✓ `@CreditCardNumber`
- ✓ `@Length`
- ✓ `@Currency`
- ✓ `@Range`
- ✓ `@URL`
- ✓ `@UniqueElements`
- ✓ `@EAN`
- ✓ `@ISBN`

# VALIDATION WITH SPRING BOOT

- Sample validations declaration inside a java POJO class,

```
@Data
public class Contact {

    @NotNull(message="Email must not be blank")
    @Email(message = "Please provide a valid email address" )
    private String email;

    @NotNull(message="Subject must not be blank")
    @Size(min=5, message="Subject must be at least 5 characters long")
    private String subject;

    @NotNull(message="Message must not be blank")
    @Size(min=10, message="Message must be at least 10 characters long")
    private String message;
}
```

# VALIDATION WITH SPRING BOOT

- We can put the `@Valid` annotation on method parameters to tell Spring framework that we want a particular POJO object needs to be validated based on the validation annotation configurations. For any issues, framework populates the error details inside the `Errors` object. The errors can be used to display on the UI to the user. Sample example code is below,

```
@RequestMapping(value = "/saveMsg",method = POST)
public String saveMessage(@Valid @ModelAttribute("contact") Contact contact, Errors errors) {

    if(errors.hasErrors()){
        log.error("Contact form validation failed due to : " + errors.toString());
        return "contact.html";
    }
    contactService.saveMessageDetails(contact);
    return "redirect:/contact";
}
```

```
<ul>
    <li class="alert alert-danger" role="alert" th:each="error : ${#fields.errors('contact.*')}"
        th:text="${error}" />
</ul>
```

# Bean Scopes inside Spring

1 Singleton

2 Prototype

3 Request

4 Session

5 Application

## Web Scopes inside Spring

- ✓ Request (@RequestScope)
- ✓ Session (@SessionScope)
- ✓ Application (@ApplicationScope)

1

**Request Scope** – Spring creates an instance of the bean class for every HTTP request. The instance exists only for that specific HTTP request.

2

**Session Scope** – Spring creates an instance and keeps the instance in the server's memory for the full HTTP session. Spring links the instance in the context with the client's session.

3

**Application Scope** – The instance is unique in the app's context, and it's available while the app is running.

# Key points of Spring Web scopes

### Request Scope

- ✓ Spring creates a lot of instances of this bean in the app's memory for each HTTP request. So these type of beans are short lived.
- ✓ Since Spring creates a lot of instances, please make sure to avoid time consuming logic while creating the instance.
- ✓ Can be considered for the scenarios where the data needs to be reset after new request or page refresh etc..

### Session Scope

- ✓ Session scoped beans have longer life & they are less frequently garbage collected.
- ✓ Avoid keeping too much information inside session data as it impacts performance. Never store sensitive information as well.
- ✓ Can be considered for the scenarios where the same data needs to be accessed across multiple pages like user information.

### Application Scope

- ✓ In the application scope, Spring creates a bean instance per web application runtime.
- ✓ It is similar to singleton scope, with one major difference. Singleton scoped bean is singleton per ApplicationContext where application scoped bean is singleton per ServletContext.
- ✓ Can be considered for the scenarios where we want to store Drop Down values, Reference table values which won't change for all the users.

- Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications.
- Below is the maven dependency that we can add to implement security using Spring Security project in any of the SpringBoot projects,

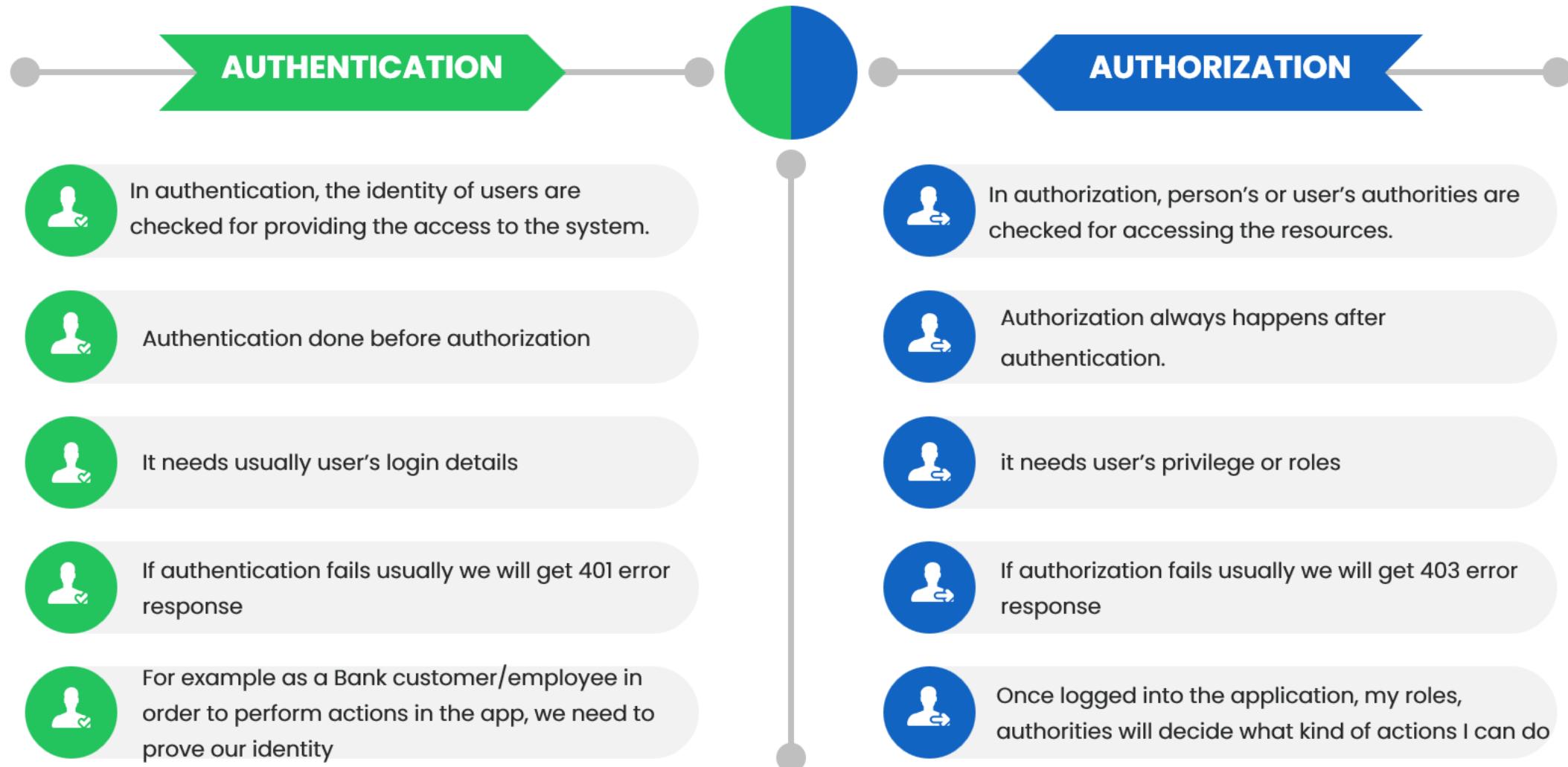
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Spring Security is a framework that provides authentication, authorization, and protection against common attacks.

Spring Security helps developers with easier configurations to secure a web application by using standard username/password authentication mechanism.

Spring Security provides out of the box features to handle common security attacks like CSRF, CORs. It also has good integration with security standards like JWT, OAUTH2 etc.

# AUTHENTICATION & AUTHORIZATION



## QUICK TIP

Do you know,

- ✓ As soon as we add spring security dependency to a web application, by default it protects all the pages/APIs inside it. It will redirect to the inbuilt login page to enter credentials.
- ✓ The default credentials are **user** and password is randomly generated & printed on the console.
- ✓ We can configure custom credentials using the below properties to get started for POCs etc. But for PROD applications, Spring Security supports user credentials configuration inside DB, LDAP, OAUTH2 Server etc.

```
spring.security.user.name = eazybytes  
spring.security.user.password = 12345
```



# DEFAULT SECURITY CONFIGURATIONS IN SPRING SECURITY

By default, Spring Security framework protects all the paths present inside the web application. This behaviour is due to the code present inside the method `defaultSecurityFilterChain(HttpSecurity http)` of class `SpringBootWebSecurityConfiguration`

```
@Bean
@Order(SecurityProperties.BASIC_AUTH_ORDER)
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.authorizeHttpRequests().anyRequest().authenticated();
    http.formLogin();
    http.httpBasic();
    return http.build();
}
```

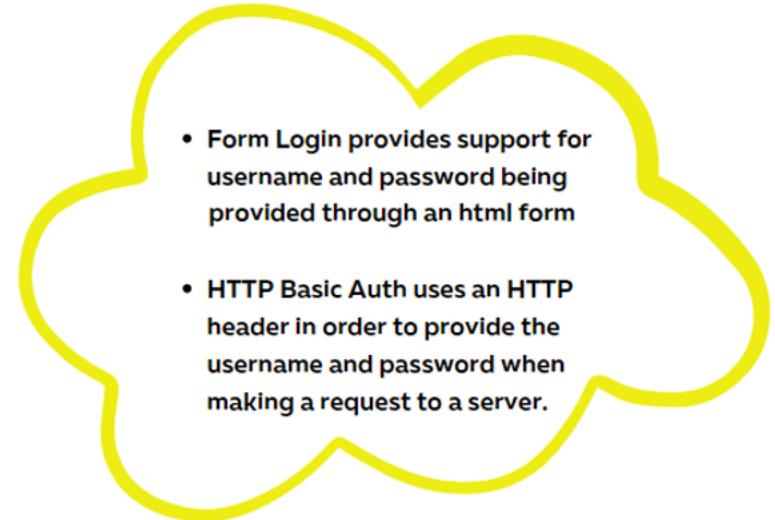
# CONFIGURE `permitAll()` WITH SPRING SECURITY

- Using `permitAll()` configurations we can allow full/public access to a specific resource/path or all the resources/paths inside a web application.
- Below is the sample configuration that we can do in order to allow any requests in a Web application with out security,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .anyRequest().permitAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```

- 
- Form Login provides support for username and password being provided through an html form
  - HTTP Basic Auth uses an HTTP header in order to provide the username and password when making a request to a server.

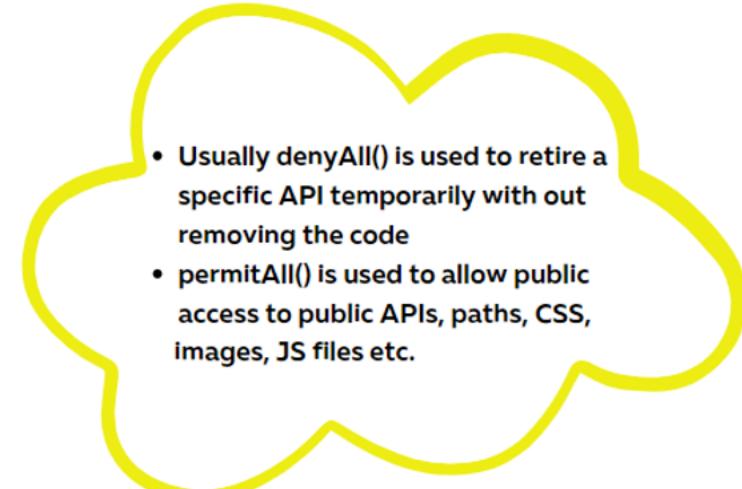
# CONFIGURE denyAll() WITH SPRING SECURITY

- Using `denyAll()` configurations we can deny access to a specific resource/path or all the resources/paths inside a web application regardless of user authentication.
- Below is the sample configuration that we can do in order to deny any requests that is coming into a web application,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
            .anyRequest().denyAll()
            .and().formLogin()
            .and().httpBasic();

        return http.build();
    }
}
```



## CONFIGURE CUSTOM SECURITY CONFIGS & CSRF DISABLE

- We can apply custom security configurations based on our requirements for each API/URL like below.
- `permitAll()` can be used to allow access w/o security and `authenticated()` can be used to protect a web page/API.
- By default any requests with HTTP methods that can update data like POST, PUT will be stopped with 403 error due to CSRF protection. We can disable the same for now and enable it in the coming sections when we started generating CSRF tokens.
- Below is the sample configuration that we can do to implement custom security configs and disable CSRF,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("/dashboard").authenticated()
            .requestMatchers("", "/", "/home").permitAll()
            .requestMatchers("/holidays/**").permitAll()
            .requestMatchers("/contact").permitAll()
            .requestMatchers("/saveMsg").permitAll()
            .requestMatchers("/courses").permitAll()
            .requestMatchers("/about").permitAll()
            .requestMatchers("/assets/**").permitAll()
            .and().formLogin()
            .and().httpBasic();
        return http.build();
    }
}
```

# IN-MEMORY AUTHENTICATION IN SPRING SECURITY

- Spring Security provide support for username/password based authentication based on the users stored in application memory.
- Like mentioned below, we can configure any number of users & their roles, passwords using in-memory authentication,

```
@Configuration
public class ProjectSecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {...}

    @Bean
    public InMemoryUserDetailsManager userDetailsService() {

        UserDetails admin = User.withDefaultPasswordEncoder()
            .username("user")
            .password("12345")
            .roles("USER")
            .build();
        UserDetails user = User.withDefaultPasswordEncoder()
            .username("admin")
            .password("54321")
            .roles("USER", "ADMIN")
            .build();

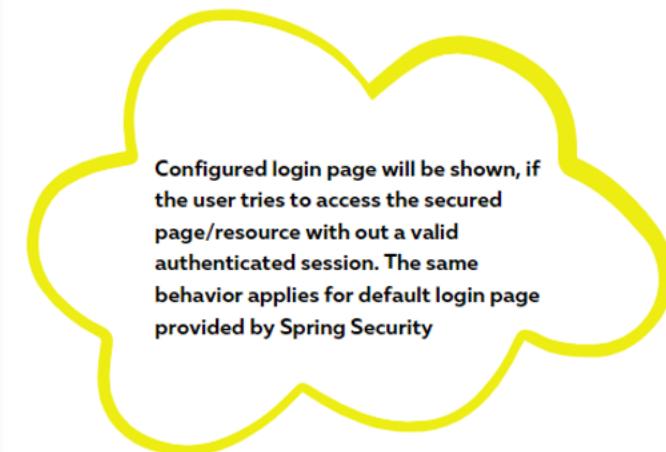
        return new InMemoryUserDetailsManager(user, admin);
    }
}
```



## CONFIGURING LOGIN & LOGOUT PAGE

- Spring Security allows us to configure a custom login page to our web application instead of using the Spring Security default provided login page.
- Similarly we can configure logout page as well.
- Below is the sample configuration that we can follow,

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        .authorizeHttpRequests()
        .requestMatchers("/dashboard").authenticated()
        .requestMatchers("", "/", "/home").permitAll()
        .requestMatchers("/holidays/**").permitAll()
        .requestMatchers("/contact").permitAll()
        .requestMatchers("/saveMsg").permitAll()
        .requestMatchers("/courses").permitAll()
        .requestMatchers("/about").permitAll()
        .requestMatchers("/assets/**").permitAll()
        .requestMatchers("/login").permitAll()
        .and().formLogin().LoginPage("/login")
        .defaultSuccessUrl("/dashboard").failureUrl("/login?error=true").permitAll()
        .and().logout().logoutSuccessUrl("/login?logout=true").invalidateHttpSession(true).permitAll()
        .and().httpBasic();
    return http.build();
}
```



Configured login page will be shown, if the user tries to access the secured page/resource with out a valid authenticated session. The same behavior applies for default login page provided by Spring Security

## QUICK TIP

Do you know, Thymeleaf has a great integration with Spring Security. More details can be found at <https://www.thymeleaf.org/doc/articles/springsecurity.html>

*Step 1 : Add the below dependency in the pom.xml*

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>
</dependency>
```

*Step 2 : Add the below XML name space which enable us to use Thymeleaf Security related tags*

```
<html lang="en" xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity6">
```

*Step 3 : Use any of the Thymeleaf Security tags inside HTML code based on Authentication details,*

```
sec:authorize="isAnonymous()", sec:authorize="isAuthenticated()", sec:authorize="hasRole('ROLE_ADMIN')"
sec:authentication="name", sec:authentication="principal.authorities"
```



# @ControllerAdvice & @ExceptionHandler

eazy  
bytes

- **@ControllerAdvice** is a specialization of the **@Component** annotation which allows to handle exceptions across the whole application in one global handling component. You can think of it as an interceptor of exceptions thrown by methods annotated with **@RequestMapping** or one of the shortcuts like **@GetMapping** etc.
- We can define the exception handle logic inside a method and annotate it with **@ExceptionHandler**
- Below is the sample configuration that we can follow,

```
@ControllerAdvice
public class GlobalExceptionController {

    @ExceptionHandler(Exception.class)
    public ModelAndView exceptionHandler(Exception exception){
        ModelAndView errorPage = new ModelAndView();
        errorPage.setViewName("error");
        errorPage.addObject("errormsg", exception.getMessage());
        return errorPage;
    }

}
```



Combination of **@ControllerAdvice** & **@ExceptionHandler** can handle the exceptions across all the controllers inside a web application globally.

## QUICK TIP

Do you know,

- ✓ If a method annotated with `@ExceptionHandler` present inside a `@Controller` class, then the exception handling logic will be applicable for any exceptions occurred in that specific controller class.
- ✓ If the same `@ExceptionHandler` annotated method present inside a `@ControllerAdvice` class, then the exception handling logic will be applicable for any exceptions occurred across all the controller classes.
- ✓ Using `@ExceptionHandler` annotation, we can handle any number of exceptions like below sample code,

```
@ExceptionHandler({NullPointerException.class,  
                    ArrayIndexOutOfBoundsException.class,  
                    IOException.class})  
public ModelAndView handleException(RuntimeException ex)  
{  
    //Exception handling logic  
}
```



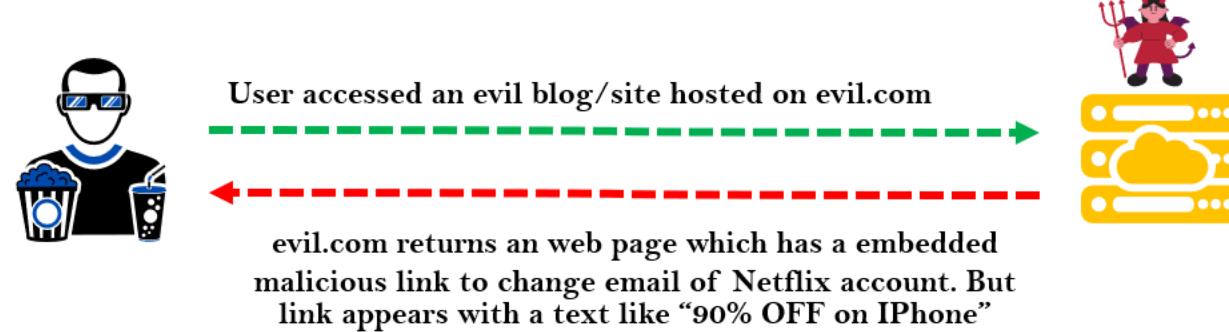
# CROSS-SITE REQUEST FORGERY (CSRF)

- A typical Cross-Site Request Forgery (CSRF or XSRF) attack aims to perform an operation in a web application on behalf of a user without their explicit consent. In general, it doesn't directly steal the user's identity, but it exploits the user to carry out an action without their will.
- Consider you are using a website `netflix.com` and the attacker's website `evil.com`.

**Step 1 : The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com**

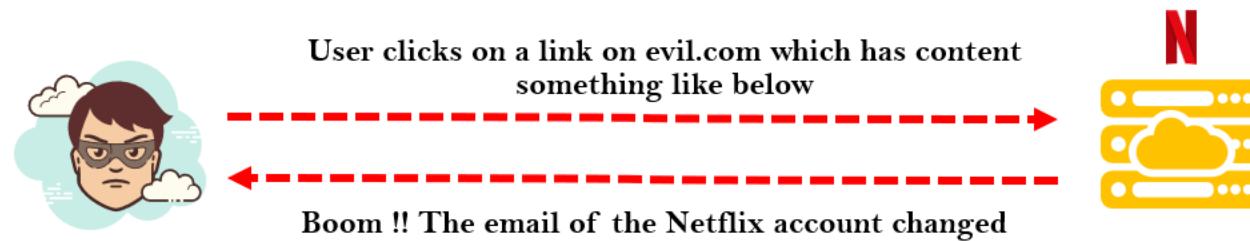


**Step 2 : The same Netflix user opens an evil.com website in another tab of the browser.**



# CROSS-SITE REQUEST FORGERY (CSRF)

Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com, the backend server of Netflix.com can't differentiate from where the request came. So here the evil.com forged the request as if it is coming from a Netflix.com UI page.



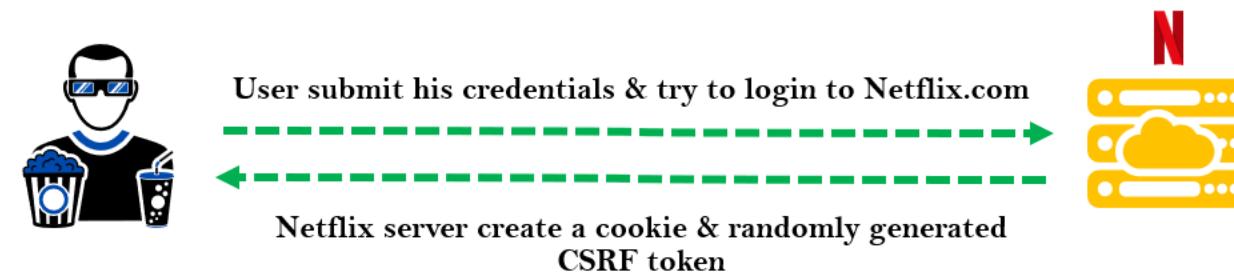
```
<form action="https://netflix.com/changeEmail"
      method="POST" id="form">
    <input type="hidden" name="email" value="user@evil.com">
</form>

<script>
  document.getElementById('form').submit()
</script>
```

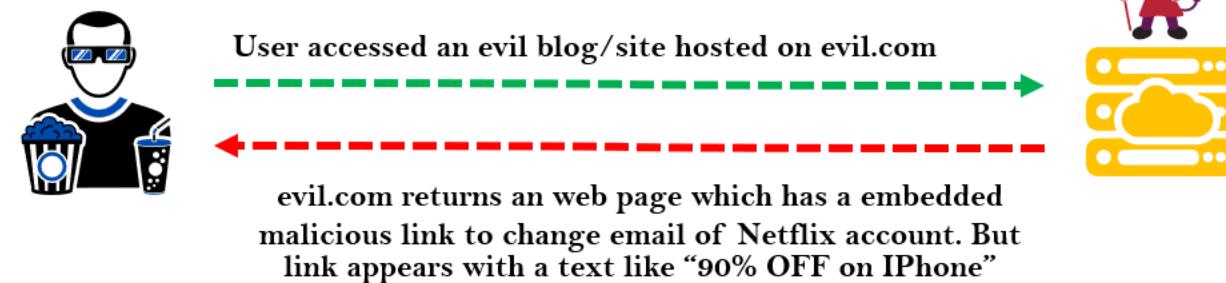
## SOLUTION TO CSRF

- To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application's user interface. The best way to achieve this is through a **CSRF token**. A CSRF token is a secure random token that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.
- Let's see how this solve CSRF attack by taking the previous Netflix example again,

**Step 1 :** The Netflix user login to Netflix.com and the backend server of Netflix will provide a cookie which will store in the browser against the domain name Netflix.com along with a randomly generated unique CSRF token for this particular user session. CSRF token is inserted within hidden parameters of HTML forms to avoid exposure to session cookies.



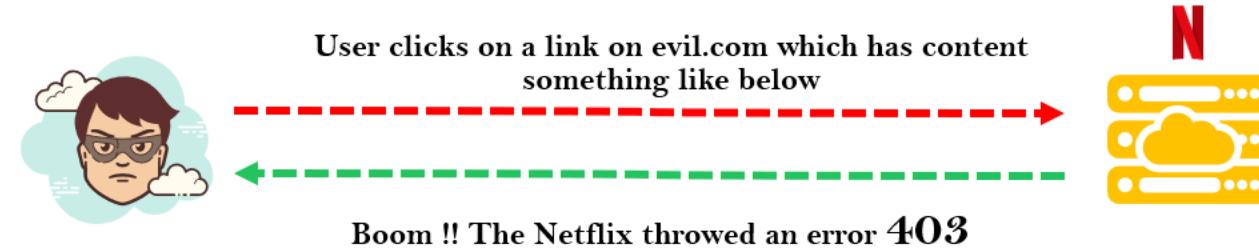
**Step 2 :** The same Netflix user opens an evil.com website in another tab of the browser.



## SOLUTION TO CSRF

eazy  
bytes

*Step 3 : User tempted and clicked on the malicious link which makes a request to Netflix.com. And since the login cookie already present in the same browser and the request to change email is being made to the same domain Netflix.com. This time the Netflix.com backend server expects CSRF token along with the cookie. The CSRF token must be same as initial value generated during login operation*



**The CSRF token will be used by the application server to verify the legitimacy of the end-user request if it is coming from the same App UI or not. The application server rejects the request if the CSRF token fails to match the test.**

## QUICK TIP

Do you know,

- ✓ By default, Spring Security enables CSRF fix for all the HTTP methods which results in data change like POST, DELETE etc. But not for GET.
- ✓ Using Spring Security configurations we can disable the CSRF protection for complete application or for only few paths based on our requirements like below.
  - `http.csrf().disable()`
  - `http.csrf().ignoringRequestMatchers("/saveMsg")`
- ✓ Thymeleaf has great integration & support with Spring Security to generate a CSRF token. We just need to add the below code in login html form code and Thymeleaf will automatically appends the CSRF token for the remaining pages/forms inside the web application,

```
<input type="hidden" th:name="${_csrf.parameterName}" th:value="${_csrf.token}" />
```



# SPRING BOOT & H2 DATABASE

- H2 is an embedded, open-source, and in-memory database. SpringBoot supports integration with H2 DB which can be used for POC applications and excellent for examples/testing.
- Below is the maven dependency that we can add to any SpringBoot projects in order to use internal memory H2 Database,

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Since it is a internal memory DB, we need to create the schema and data that is needed during startup of the App. Any updates to the data will be lost after restarting the server.

To create schema & data for the H2 DB, we can add schema.sql & data.sql inside the maven project's resources folder. Any table creation scripts and DB records scripts can be present inside schema.sql and data.sql respectively.

By default, the H2 web console is available at /h2-console. You can customize the console's path by using the spring.h2.console.path property.

The default credentials of H2 DB are username is sa and password is "" (Empty)

# Key points of JDBC

Intro to JDBC	Steps in JDBC to access DB	Problem with JDBC
✓ JDBC or Java Database Connectivity is a specification from Core Java that provides a standard abstraction for java apps to communicate with various databases.	We need to follow the below steps to access DB using JDBC,  <ol style="list-style-type: none"><li>1) Load Driver Class</li><li>2) Obtain a DB connection</li><li>3) Obtain a statement using connection object</li><li>4) Execute the query</li><li>5) Process the result set</li><li>6) Close the connection</li></ol>	✓ Developers are forced to follow all the steps mentioned to perform any kind of operation with DB which results in lot of duplicate code at many places.
✓ JDBC API along with the database driver is capable of accessing databases		✓ Developers needs to handle the checked exceptions that will throw from the API.
✓ JDBC is a base framework or standard for frameworks like Hibernate, Spring Data JPA, MyBatis etc.		✓ JDBC is database dependent

# INTRO TO JDBC & PROBLEMS WITH IT

```
public Optional<Contact> findById(int id) {  
    Connection connection = null;  
    PreparedStatement statement = null;  
    ResultSet resultSet = null;  
    try {  
        Class.forName("com.mysql.jdbc.Driver");  
        connection = DriverManager.getConnection("url", "username", "pwd");  
        statement = connection.prepareStatement(  
            "select id, name, message from contact");  
        statement.setInt(1, id);  
        resultSet = statement.executeQuery();  
        Contact contact = null;  
        if(resultSet.next()) {  
            contact = new Contact(  
                resultSet.getInt("id"),  
                resultSet.getString("name"),  
                resultSet.getString("message"));  
        }  
        return Optional.of(contact);  
    } catch (SQLException e) {  
        // ??? What should be done here ???  
    } finally {  
        if (resultSet != null) {  
            try {  
                resultSet.close();  
            } catch (SQLException e) {}  
        }  
        if (statement != null) {  
            try {  
                statement.close();  
            } catch (SQLException e) {}  
        }  
        if (connection != null) {  
            try {  
                connection.close();  
            } catch (SQLException e) {}  
        }  
    }  
    return null;  
}
```

Sample program using JDBC to fetch a record from the database table. You can see there is lot of boilerplate code present



# INTRO TO SPRING JDBC

- Spring JDBC simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. It does this magic by providing JDBC templates which developers can use inside their applications.
- Below is the maven dependency that we need to add to any Spring/SpringBoot projects in order to use Spring JDBC provided templates,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
```

Spring provides many templates for JDBC related activities. Among them, the famous ones are JdbcTemplate , NamedParameterJdbcTemplate.

JdbcTemplate is the classic and most popular Spring JDBC approach. This provides “lowest-level” approach and all others templates uses JdbcTemplate under the covers.

NamedParameterJdbcTemplate wraps a JdbcTemplate to provide named parameters instead of the traditional JDBC ? placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.

## Spring JDBC - who does what?

Action	Spring JDBC	Developer
Define connection parameters		✗
Open the connection	✗	
Specify the SQL statement		✗
Declare parameters and provide parameter values		✗
Prepare and run the statement	✗	
Set up the loop to iterate through the results (if any).	✗	
Do the work for each iteration		✗
Process any exception	✗	
Handle transactions	✗	
Close the connection, the statement, and the resultset	✗	

# USING JdbcTemplate

- *JdbcTemplate is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors, such as forgetting to close the connection. It performs the basic tasks of the core JDBC workflow (such as statement creation and execution), leaving application code to provide SQL and extract results.*
- *You can use JdbcTemplate within a DAO implementation through direct instantiation with a DataSource reference, or you can configure it in a Spring IoC container and give it to DAOs as a bean reference.*
- *We need to follow the below steps in order to configure JdbcTemplate inside a Spring Web application (With out Spring Boot)*

**Step 1 :** First we need to create a Data Source Bean inside Web application with the DB credentials like mentioned below.

```
@Bean
public DataSource myDataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/eazyschool");
    dataSource.setUsername("user");
    dataSource.setPassword("password");
    return dataSource;
}
```

# USING JdbcTemplate

Step 2 : Inside any Repository/DAO classes where we want to execute queries, we need to create a bean/object of JdbcTemplate by injecting data source bean

```
@Repository
public class PersonDAOImpl implements PersonDAO {
    JdbcTemplate jdbcTemplate;

    @Autowired
    public PersonDAOImpl(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }

}
```

- Instances of the JdbcTemplate class are thread-safe, once configured. This is important because if needed we can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs (or repositories).

## USING JdbcTemplate

Sample usage of JdbcTemplate for SELECT queries,

```
// The following query gets the number of rows in a table
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from person", Integer.class);
```

```
// The following query uses a bind variable
int countOfPersonsNamedJoe = this.jdbcTemplate.queryForObject(
    "select count(*) from person where first_name = ?", Integer.class, "Joe");
```

```
// The following query looks for a string column based on a condition:
String lastName = this.jdbcTemplate.queryForObject("select last_name from person where id = ?",
    String.class, 1212L);
```

## USING JdbcTemplate

*Sample usage of JdbcTemplate for Updating (INSERT, UPDATE, and DELETE)*

```
// The following example inserts a new entry
this.jdbcTemplate.update("insert into person (first_name, last_name) values (?, ?)",
    "Jon", "Doe");
```

```
// The following example updates an existing entry
this.jdbcTemplate.update("update person set last_name = ? where id = ?",
    "Maria", 5276L);
```

```
// The following example deletes an entry
this.jdbcTemplate.update("delete from person where id = ?", 5276L);
```

## USING JdbcTemplate

### Other JdbcTemplate Operations

```
// You can use the execute(..) method to run any arbitrary SQL. Consequently, the method is often
// used for DDL statements.
this.jdbcTemplate.execute("create table person (id integer, name varchar(100));")
```

```
// The following example invokes a stored procedure
this.jdbcTemplate.update("call SUPPORT.REFRESH_PERSON_SUMMARY(?)", 5276L);
```

## USING ROWMAPPER

- RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.
- RowMapper saves a lot of code because it internally adds the data of ResultSet into the collection.
- It defines only one method mapRow that accepts ResultSet instance and int as parameters. Below is the sample usage of it,

```
private final RowMapper<Person> personRowMapper = (resultSet, rowNum) -> {
    Person person = new Person();
    person.setFirstName(resultSet.getString("first_name"));
    person.setLastName(resultSet.getString("last_name"));
    return person;
};
```

```
public List<Person> findAllPersons() {
    return this.jdbcTemplate.query("select first_name, last_name from person", personRowMapper);
}
```

## QUICK TIP

Do you know, if the column names in a table and field names inside a POJO/Bean are matching, then we can use **BeanPropertyRowMapper** which is provided by Spring framework.

Spring BeanPropertyRowMapper, class saves you a lot of time since we don't have to define the mappings like we do inside a RowMapper implementation.

```
public List<Holiday> findAllHolidays() {  
    String sql = "SELECT * FROM HOLIDAYS";  
    var rowMapper = BeanPropertyRowMapper.newInstance(Holiday.class);  
    return jdbcTemplate.query(sql, rowMapper);  
}
```



# USING NamedParameterJdbcTemplate

- The `NamedParameterJdbcTemplate` class adds support for programming JDBC statements by using named parameters, as opposed to programming JDBC statements using only classic placeholder ('?') arguments. The `NamedParameterJdbcTemplate` class wraps a `JdbcTemplate` and delegates to the wrapped `JdbcTemplate` to do much of its work.
- The following example shows how to use `NamedParameterJdbcTemplate`

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfPersonsByFirstName(String firstName) {
    String sql = "select count(*) from Person where first_name = :first_name";
    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);
    return this.namedParameterJdbcTemplate.queryForObject(sql, namedParameters, Integer.class);
}
```

## QUICK TIP

Do you know, with Spring Boot working with JdbcTemplate is very easy. Spring Boot auto configures DataSource, JdbcTemplate and NamedParameterJdbcTemplate classes based on the DB connection details mentioned in the property file and you can @Autowire them directly into your own repository classes, as shown in the following example.

You can customize some properties of the template by using the `spring.jdbc.template.*` properties, like mentioned below,

```
spring.jdbc.template.max-rows=500
```

```
@Repository
public class HolidaysRepository {

    private final JdbcTemplate jdbcTemplate;

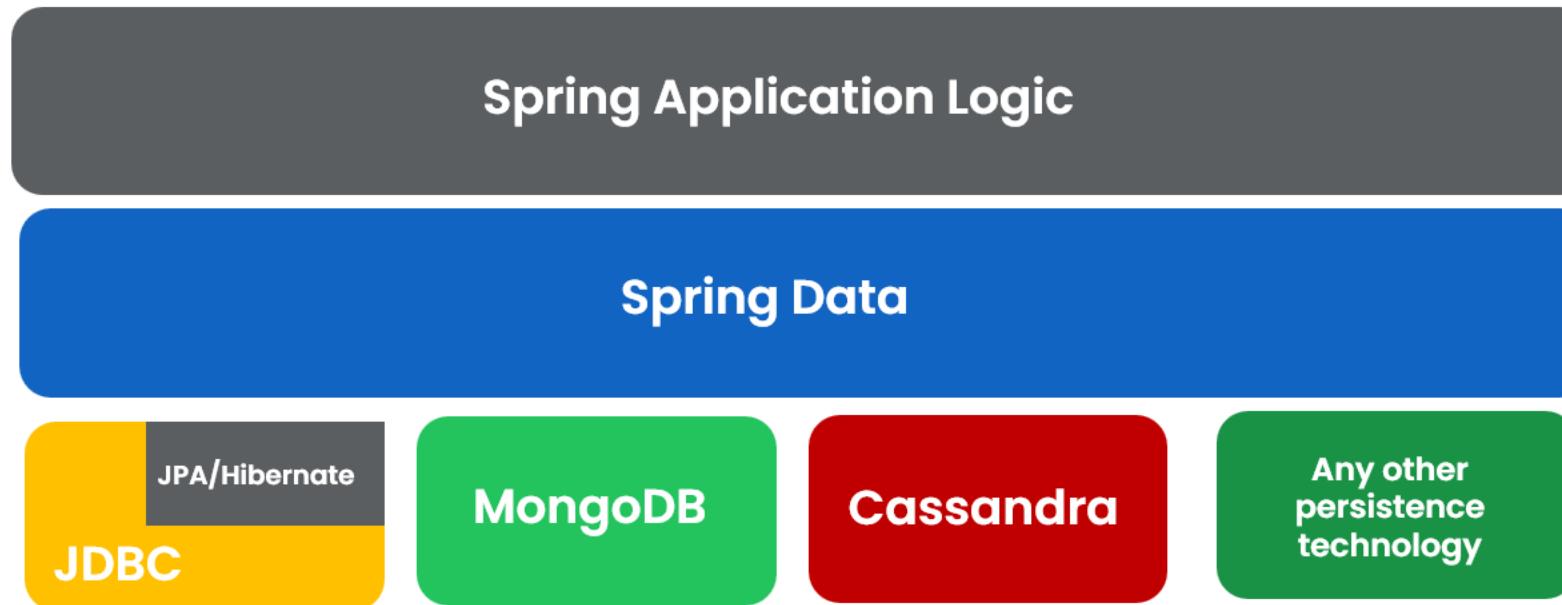
    @Autowired
    public HolidaysRepository(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```



# INTRO TO SPRING DATA

eazy  
bytes

*Spring Data is a Spring ecosystem project that simplifies the persistence layer's development by providing implementations according to the persistence technology we use. This way, we only need to write a few lines of code to define the repositories of our Spring app.*



**Spring Data is a high-level layer that simplifies the persistence implementation by unifying the various technologies under the same abstractions.**

# INTRO TO SPRING DATA

- Whichever persistence technology your app uses, Spring Data provides a common set of interfaces (contracts) you extend to define the app's persistence capabilities.
- The central interface in the Spring Data repository abstraction is **Repository**

Repository is the most abstract contract. If you extend this contract, your app recognizes the interface you write as a particular Spring Data repository. Still, you won't inherit any predefined operations (such as adding a new record, retrieving all the records, or getting a record by its primary key). The Repository interface doesn't declare any method (it is a marker interface).

CrudRepository is the simplest Spring Data contract that also provides some persistence capabilities. If you extend this contract to define your app's persistence capabilities, you get the simplest operations for creating, retrieving, updating, and deleting records. ListCrudRepository is an extension to CrudRepository returning List instead of Iterable where ever applicable.

PagingAndSortingRepository provide methods to retrieve entities using the pagination and sorting abstraction. ListPagingAndSortingRepository an extension to PagingAndSortingRepository returning List instead of Iterable where ever applicable.

# INTRO TO SPRING DATA

eazy  
bytes

To implement your app's repositories using Spring Data, you extend specific interfaces. The main interfaces that represent Spring Data contracts are `Repository`, `CrudRepository`, `ListCrudRepository`, `PagingAndSortingRepository` and `ListPagingAndSortingRepository`. You extend one of these contracts to implement your app's persistence capabilities.



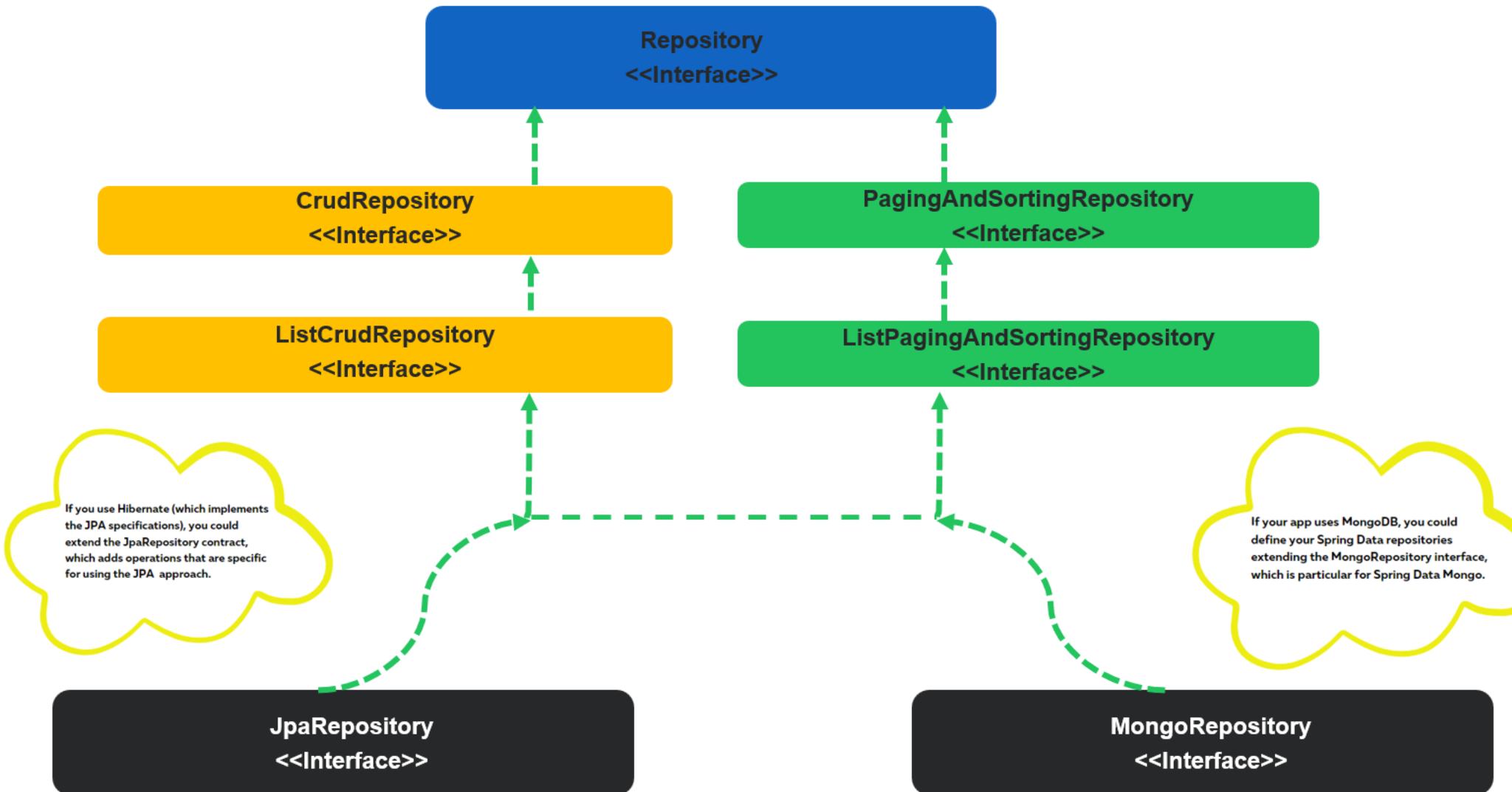
## QUICK TIP

Do you know,

- ✓ We should not confuse between `@Repository` annotation and Spring Data Repository interface.
- ✓ Spring Data provides multiple interfaces that extend one another by following the principle called interface segregation. This helps apps to extend what they want instead of always following fat implementation.
- ✓ Some Spring Data modules might provide specific contracts to the technology they represent. For example, using Spring Data JPA, you also can extend the `JpaRepository` interface directly and similarly using Spring Data Mongo module to your app provides a particular contract named `MongoRepository`



# INTRO TO SPRING DATA



# INTRO TO SPRING DATA JPA

eazy  
bytes

- Spring Data JPA is available to Spring Boot applications with the JPA starter. This starter dependency not only brings in Spring Data JPA, but also transitively includes Hibernate as the JPA implementation.
- Below is the maven dependency that we need to add to any SpringBoot projects in order to use Spring Data JPA,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```



- We need to follow the below steps in order to query a DB using Spring Data JPA inside a Spring Boot applications,

Step 1 : We need to indicate a java POJO class as an entity class by using annotations like @Entity, @Table, @Column

```
@Entity
@Table(name="contact_msg")
public class Contact extends BaseEntity{

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO,generator="native")
    @GenericGenerator(name = "native",strategy = "native")
    @Column(name = "contact_id")
    private int contactId;
```

# INTRO TO SPRING DATA JPA

eazy  
bytes

Step 2 : We need to create interfaces for a given table entity by extending framework provided Repository interfaces. This helps us to run the basic CRUD operations on the table w/o writing method implementations.

```
@Repository
public interface ContactRepository extends CrudRepository<Contact, Integer> {
}
```

Step 3 : Enable JPA functionality and scanning by using the annotations `@EnableJpaRepositories` and `@EntityScan`

```
@SpringBootApplication
@EnableJpaRepositories("com.eazybytes.eazyschool.repository")
@EntityScan("com.eazybytes.eazyschool.model")
public class EazyschoolApplication {
```

# INTRO TO SPRING DATA JPA

eazy  
bytes

Step 4 : We can inject repository beans into any controller/service classes and execute the required DB operations.

```
  @Service
  public class ContactService {

    @Autowired
    private ContactRepository contactRepository;

    public boolean saveMessageDetails(Contact contact){
        boolean isSaved = false;
        Contact savedContact = contactRepository.save(contact);
        if(null != savedContact && savedContact.getContactId()>0) {
            isSaved = true;
        }
        return isSaved;
    }
}
```

# Derived Query Methods in Spring Data JPA

eazy  
bytes

- With Spring Data JPA, we can use the method names to derive a query and fetch the results w/o writing code manually like with traditional JDBC
- As a developer we Just need to define the query methods in a repository interface that extends one of the Spring Data's repositories such as CrudRepository. Spring Data JPA will create queries and implementation at runtime automatically by parsing these method names.
- Below are few examples,

```
// find persons by last name
List<Person> findByLastName(String lastName);
```

```
// find person by email
Person findByEmail(String email);
```

```
// find person by email and last name
Person findByEmailAndLastname(String email, String lastname);
```

## QUICK TIP

Do you know a derived query method name has two main components separated by the first `By` keyword.

1. The **introducer** clause like `find`, `read`, `query`, `count`, or `get` which tells Spring Data JPA what you want to do with the method. This clause can contain further expressions, such as `Distinct` to set a distinct flag on the query to be created.
2. The **criteria** clause that starts after the first `By` keyword. The first `By` acts as a delimiter to indicate the start of the actual query criteria. The criteria clause is where you define conditions on entity properties and concatenate them with `And` and `Or` keywords.

*Using `readBy`, `getBy`, and `queryBy` in place of `findBy` will behave the same. For example, `readByEmail(String email)` is same as `findByEmail(String email)`.*



## Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
Distinct	findDistinctByLastnameAndFirstname	select distinct ... where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
Greater Than	findByAgeGreaterThan	... where x.age > ?1
GreaterThanOrEqual	findByAgeGreaterThanOrEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1

## Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
IsNull, Null	findByAge(Is)Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	... where x.age not in ?1

## Supported keywords inside method names

Keyword	Sample method name	Sample Query that form by JPA
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

Spring Data JPA is a powerful tool that provides an extra layer of abstraction on top of an existing JPA providers like Hibernate. The derived query feature is one of the most loved features of Spring Data JPA.

# Auditing Support By Spring Data JPA

- Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and when the change happened. To benefit from that functionality, you have to equip your entity classes with auditing metadata that can be defined either using annotations or by implementing an interface.
- Additionally, auditing has to be enabled either through Annotation configuration or XML configuration to register the required infrastructure components.
- Below are the steps that needs to be followed,

Step 1 : We need to use the annotations to indicate the audit related columns inside DB tables. Spring Data JPA ships with an entity listener that can be used to trigger the capturing of auditing information. We must register the AuditingEntityListener to be used for all the required entities.

```
@Data  
@MappedSuperclass  
@EntityListeners(AuditingEntityListener.class)  
public class BaseEntity {  
  
    @CreatedDate  
    @Column(updatable = false)  
    private LocalDateTime createdAt;  
    @CreatedBy  
    @Column(updatable = false)  
    private String createdBy;  
    @LastModifiedDate  
    @Column(insertable = false)  
    private LocalDateTime updatedAt;  
    @LastModifiedBy  
    @Column(insertable = false)  
    private String updatedBy;  
}
```



@CreatedDate, @CreatedBy, @LastModifiedDate, @LastModifiedBy are the key annotations that support JPA auditing

# Auditing Support By Spring Data JPA

eazy  
bytes

Step 2 : Date related info will be fetched from the server by JPA but for `CreatedBy` & `UpdatedBy` we need to let JPA know how to fetch that info by implementing `AuditorAware` interface like shown below,

```
@Component("auditAwareImpl")
public class AuditAwareImpl implements AuditorAware<String> {

    @Override
    public Optional<String> getCurrentAuditor() {
        return Optional.ofNullable(SecurityContextHolder.getContext()
            .getAuthentication().getName());
    }
}
```

Step 3 : Enable JPA auditing by annotating a configuration class with the `@EnableJpaAuditing` annotation.

```
@SpringBootApplication
@EnableJpaRepositories("com.eazybytes.eazyschool.repository")
@EntityScan("com.eazybytes.eazyschool.model")
@EnableJpaAuditing(auditorAwareRef = "auditAwareImpl")
public class EazyschoolApplication {
```

## QUICK TIP

Do you know we can print the queries that are being formed and executed by Spring Data JPA by enabling the below properties,

`spring:jpa.show-sql=true`

`spring:jpa.properties.hibernate.format_sql=true`

- ✓ show-sql property will print the query on the console/logs whereas format\_sql property will print the queries in a readable friendly style.
- ✓ But please make sure to leverage them in non-prod environments only as they impact the performance of the web application.



# Spring MVC Custom Validations

We have seen before using Bean validations like Max, Min, Size etc. we can do validations on the input received. Now let's try to define custom validations that are specific to our business requirements. For the same we need to follow the below steps,

Step 1 : Suppose if we have a requirement to not allow some weak passwords inside our user registration form, we first need to create a custom annotation like below. Here we need to provide the class name where the actual validation logic present.

```
@Documented
@Constraint(validatedBy = PasswordStrengthValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
public @interface PasswordValidator {
    String message() default "Please choose a strong password";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

# Spring MVC Custom Validations

eazy  
bytes

*Step 2 : We need to create a class that implements ConstraintValidator interface and overriding the isValid() method like shown below,*

```
public class PasswordStrengthValidator implements
    ConstraintValidator<PasswordValidator, String> {

    List<String> weakPasswords;

    @Override
    public void initialize(PasswordValidator passwordValidator) {
        weakPasswords = Arrays.asList("12345", "password", "qwerty");
    }

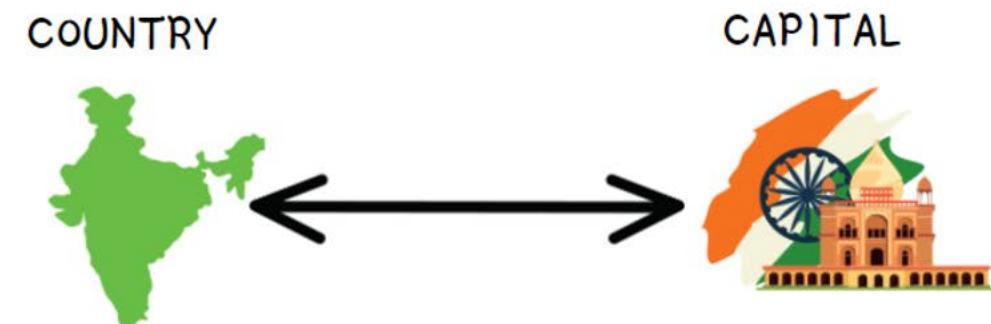
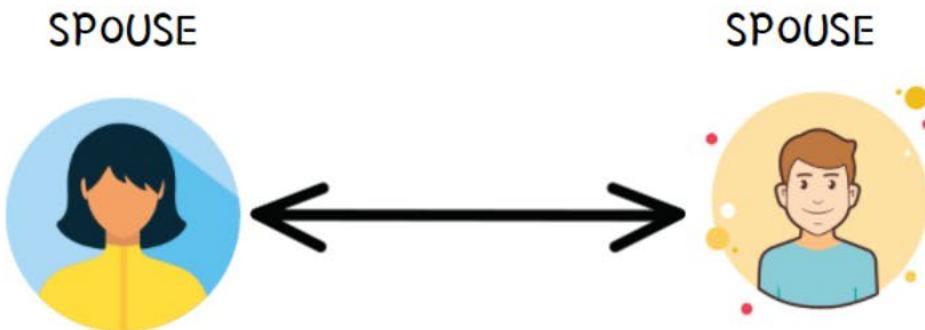
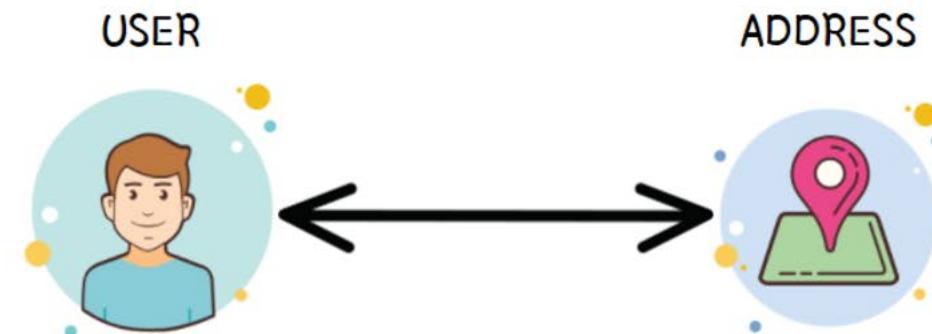
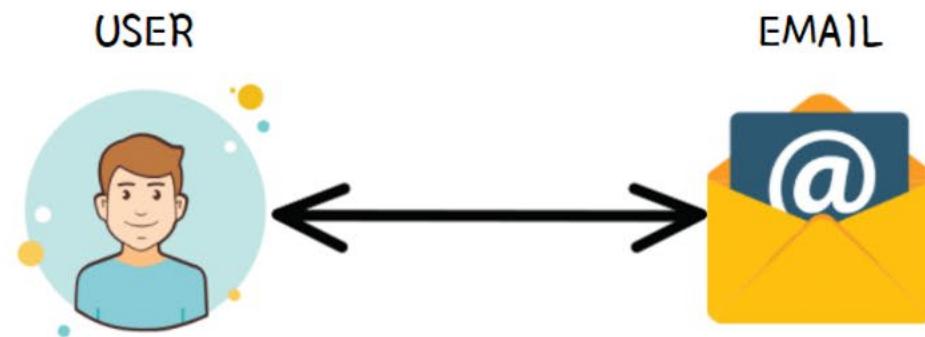
    @Override
    public boolean isValid(String passwordField,
                          ConstraintValidatorContext ctx) {
        return passwordField != null && (!weakPasswords.contains(passwordField));
    }
}
```

*Step 3 : Finally we can mention the annotation that we created on top of the field inside a POJO class,*

```
@NotBlank(message="Password must not be blank")
@Size(min=5, message="Password must be at least 5 characters long")
@PasswordValidator
private String pwd;
```

# One to One Relationship inside JPA

- First, what is a one-to-one relationship? It's a relationship where a record in one entity (table) is associated with exactly one record in another entity (table).
- Below are few real-life examples of one-to-one relationships,



# One to One Relationship inside JPA

eazy  
bytes

- Spring Data JPA allow developers to build one-to-one relationship between the entities with simple configurations. For example if we want to build a one-to-one between Person and Address entities, then we can configure like mentioned below.

```
@Data  
@Entity  
public class Person {  
  
    @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL, targetEntity = Address.class)  
    @JoinColumn(name = "address_id", referencedColumnName = "addressId", nullable = true)  
    private Address address;  
}
```

- In Spring Data JPA, a one-to-one relationship between two entities is declared by using the `@OneToOne` annotation. Using it we can configure `FetchType`, `cascade` effects, `targetEntity`.
- The `@JoinColumn` annotation is used to specify the foreign key column relationship details between the 2 entities. `"name"` defines the name of the foreign key column, `referencedColumnName` indicates the field name inside the target entity class, `nullable` defines whether foreign key column can be nullable or not.

- *Based on the fetch configurations that developer has done, JPA allows entities to load the objects with which they have a relationship.*

We can declare the fetch value in the `@OneToOne`, `@OneToMany`, `@ManyToOne` and `@ManyToMany` annotations. These annotations have an attribute called `fetch` that serves to indicate the type of fetch we want to perform. It have two valid values: `FetchType.EAGER` and `FetchType.LAZY`

With `LAZY` configuration, we are telling JPA that we want to lazily load the relation entities, so when retrieving an entity, its relations will not be loaded until unless we try refer the related entity using getter method. On the contrary, with `EAGER` it will load it's relation entities as well.

By default all `ToMany` relationships are `LAZY`, while `ToOne` relationships are `EAGER`.

# Key points of Cascade Types

Intro to Cascade	Cascade Types	Best Practices
<ul style="list-style-type: none"><li>✓ JPA allows us to propagate entity state changes from Parents to Child entities. This concept is calling Cascading in JPA.</li><li>✓ The cascade configuration option accepts an array of CascadeType.</li></ul>	<p>The cascade types supported by JPA are as below:</p> <ol style="list-style-type: none"><li>1) CascadeType.PERSIST</li><li>2) CascadeType.MERGE</li><li>3) CascadeType.REFRESH</li><li>4) CascadeType.REMOVE</li><li>5) CascadeType.DETACH</li><li>6) CascadeType.ALL</li></ol>	<ul style="list-style-type: none"><li>✓ Cascading makes sense only for Parent – Child associations (where the Parent entity state transition being cascaded to its Child entities). Cascading from Child to Parent is not very useful and not recommended.</li><li>✓ There is no default cascade type in JPA. By default, no operation is cascaded.</li></ul>

# CASCADE TYPES

eazy  
bytes

**CascadeType.PERSIST** : means that `save()` or `persist()` operations cascade to related entities.

**CascadeType.MERGE** : means that related entities are merged when the owning entity is merged.

**CascadeType.REFRESH** : means the child entity also gets reloaded from the database whenever the parent entity is refreshed.

**CascadeType.REMOVE** : means propagates the remove operation from parent to child entity.

**CascadeType.DETACH** : means detach all child entities if a "manual detach" occurs for parent.

**CascadeType.ALL** : is shorthand for all of the above cascade operations.

# Spring Security AuthenticationProvider

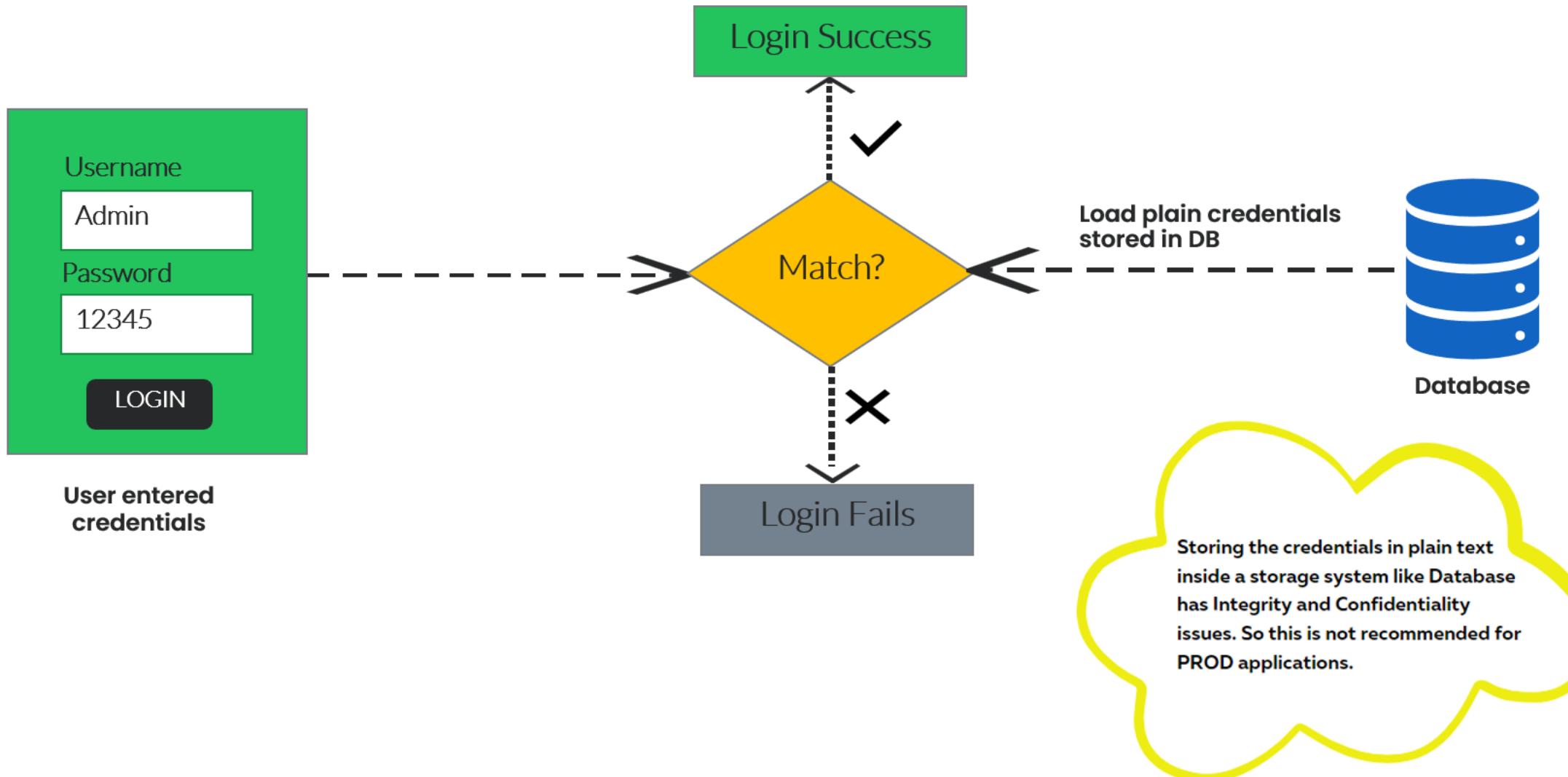
- As of now we are performing the login operation using the `inMemoryAuthentication`. But the idle way is to perform login check against a DB table or any other storage system which is more secure.
- For the same, Spring Security allow us to write our own custom logic to authenticate a user based on our requirements by implementing `AuthenticationProvider` interface. Below is the sample implementation of the same,

```
@Component
public class EazySchoolUsernamePwdAuthenticationProvider implements AuthenticationProvider {
    @Override
    public Authentication authenticate(Authentication authentication)
        throws AuthenticationException {
        String name = authentication.getName();
        String password = authentication.getCredentials().toString();
        if (authenticateUserBasedonYourRequirement()) {
            return new UsernamePasswordAuthenticationToken(
                name, password, new ArrayList<>());
        } else {
            return null;
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return authentication.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

# How Passwords Validated w/oPasswordEncoder

eazy  
bytes



# Different ways of Pwd management

### Encoding

- ✓ Encoding is defined as the process of converting data from one form to another and has nothing to do with cryptography.
- ✓ It involves no secret and completely reversible.
- ✓ Encoding can't be used for securing data. Below are the various publicly available algorithms used for encoding.

Ex: ASCII, BASE64, UNICODE

### Encryption

- ✓ Encryption is defined as the process of transforming data in such a way that guarantees confidentiality.
- ✓ To achieve confidentiality, encryption requires the use of a secret which, in cryptographic terms, we call a "key".
- ✓ Encryption can be reversible by using decryption with the help of the "key". As long as the "key" is confidential, encryption can be considered as secured.

### Hashing

- ✓ In hashing, data is converted to the hash value using some hashing function.
- ✓ Data once hashed is non-reversible. One cannot determine the original data from a hash value generated.
- ✓ Given some arbitrary data along with the output of a hashing algorithm, one can verify whether this data matches the original input data without needing to see the original data

## QUICK TIP

Do you know Spring Security provides various PasswordEncoders to help developers with hashing of the secured data like password.

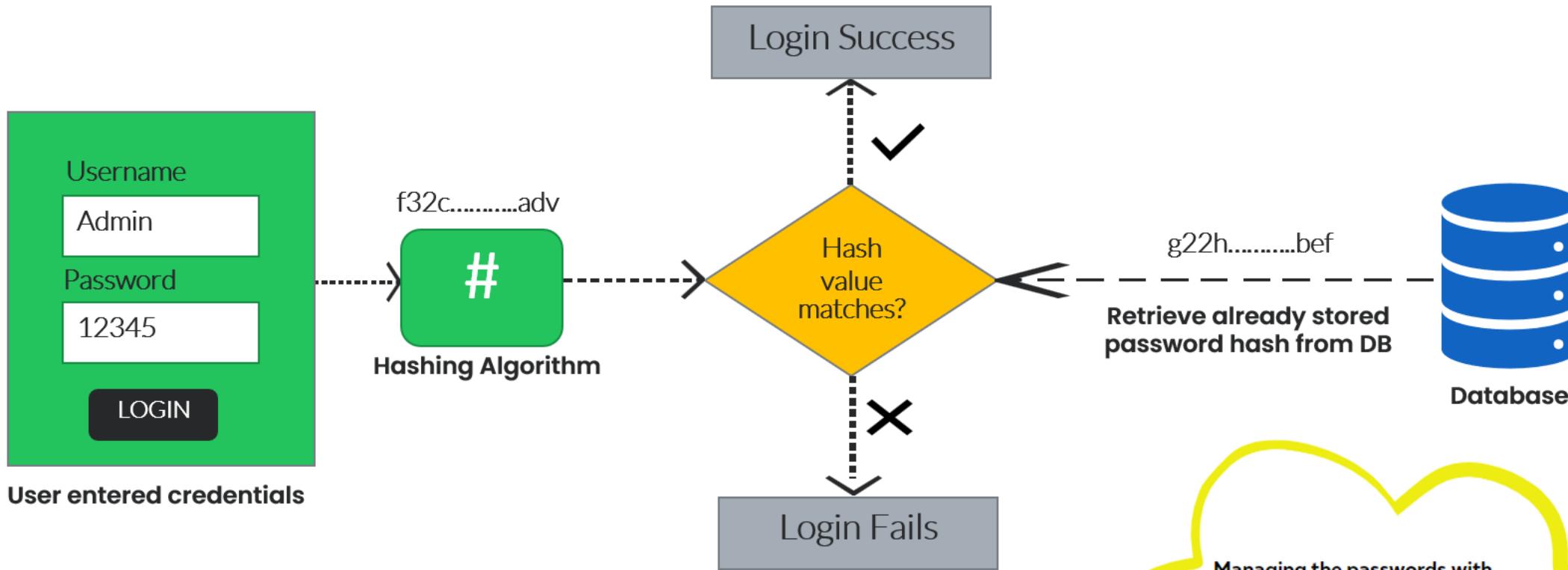
### Different Implementations of PasswordEncoders provided by Spring Security

- ✓ **NoOpPasswordEncoder** (No hashing stores in plain text)
- ✓ **StandardPasswordEncoder**
- ✓ **Pbkdf2PasswordEncoder**
- ✓ **BCryptPasswordEncoder** (Most Commonly used)
- ✓ **SCryptPasswordEncoder**



# How Passwords Validated With Hashing &PasswordEncoder

eazy  
bytes



Managing the passwords with Hashing is the recommended approach for PROD web application. With PasswordEncoders like BCryptPasswordEncoder, Spring Security makes our life easy.

## QUICK TIP

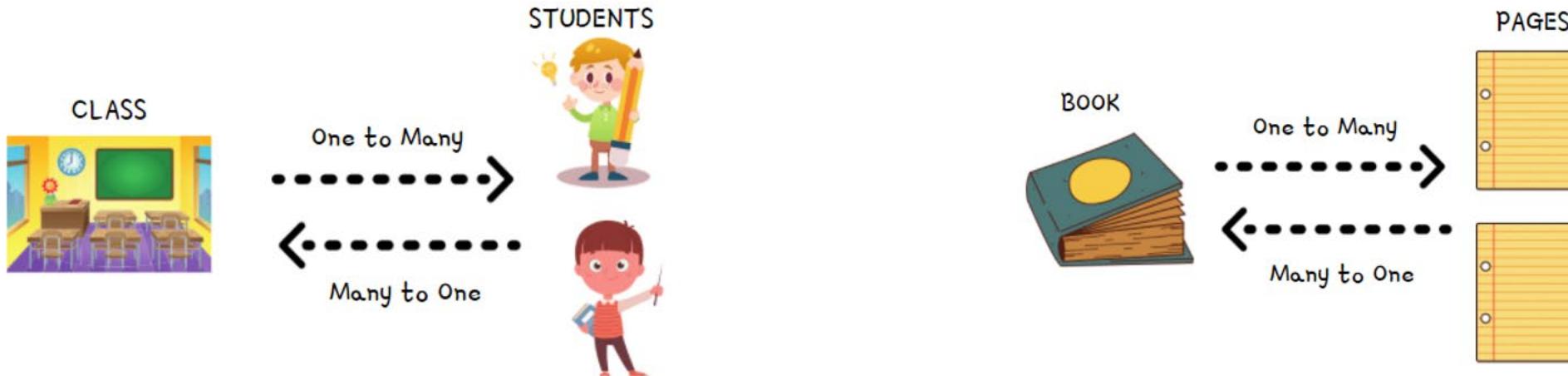
Do you know we can disable the javax bean validations by the Spring Data JPA using the below property,

*spring:jpa:properties:javax:persistence:validation:mode=none*



# One to Many & Many to One Relationship inside JPA

- A one-to-many relationship refers to the relationship between two entities/tables A and B in which one element/row of A may only be linked to many elements/rows of B, but a member of B is linked to only one element/row of A.
- The opposite of one-to-many is many-to-one relationship.
- Below are few real-life examples of one-to-many and many-to-one relationships,



# One to Many & Many to One Relationship inside JPA

- Spring Data JPA allow developers to build one-to-many & many-to-one relationships between the entities with simple configurations. Below are the sample configurations between Class and Persons,

```
@Entity
public class Person extends BaseEntity{

    @ManyToOne(fetch = FetchType.LAZY, optional = true)
    @JoinColumn(name = "class_id", referencedColumnName = "classId", nullable = true)
    private EazyClass eazyClass;
}
```

- The `@ManyToOne` annotation is used to define a many-to-one relationship between two entities. The child entity, that has the join column, is called the owner of the relationship.
- The `@JoinColumn` annotation is used to specify the foreign key column details.

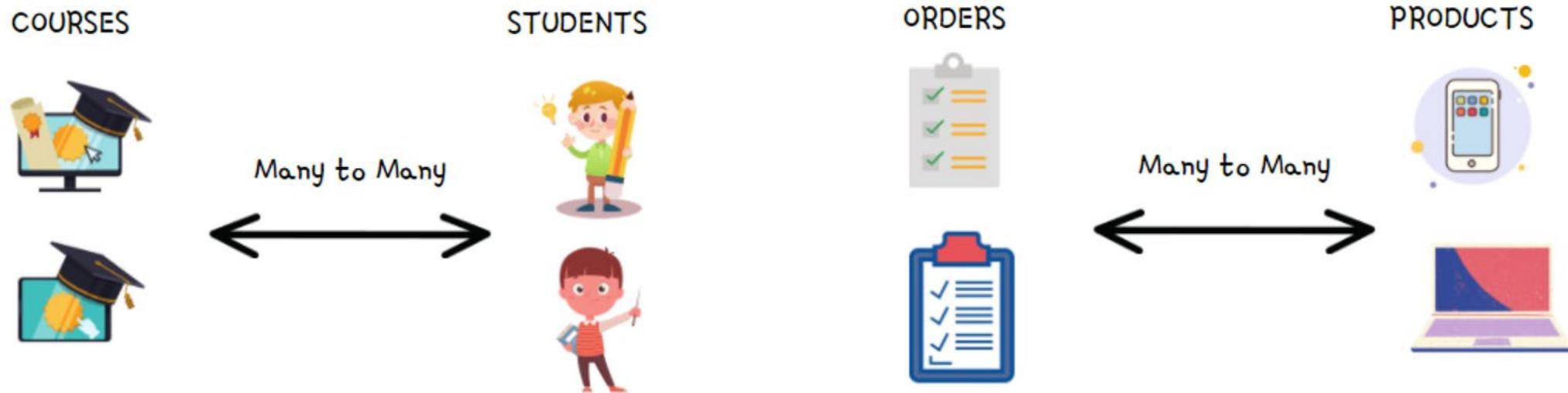
- A one-to-many relationship between two entities is defined by using the `@OneToMany` annotation. It also declares the `mappedBy` element to indicate the entity that owns the bidirectional relationship. Usually, the child entity is one that owns the relationship and the parent entity contains the `@OneToMany` annotation.

```
@Entity
@Table(name = "class")
public class EazyClass extends BaseEntity{

    @OneToMany(mappedBy = "eazyClass", fetch = FetchType.LAZY,
               cascade = CascadeType.PERSIST, targetEntity = Person.class)
    private Set<Person> persons;
}
```

# Many to Many inside JPA

- A many-to-many relationship refers to the relationship between two entities/tables A and B in which one element/row of A are associated with many elements/rows of B and vice versa.
- Below are few real-life examples of many-to-many relationship,



# Many to Many inside JPA

eazy  
bytes

- Spring Data JPA allow developers to build many-to-many relationship between the entities with simple configurations. Below are the sample configurations between Courses and Persons,

```
@Entity
public class Person extends BaseEntity{

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
    @JoinTable(name = "person_courses",
        joinColumns = {
            @JoinColumn(name = "person_id", referencedColumnName = "personId")},
        inverseJoinColumns = {
            @JoinColumn(name = "course_id", referencedColumnName = "courseId")})
    private Set<Courses> courses = new HashSet<>();
}
```

- A many-to-many relationship between two entities is defined by using the `@ManyToMany` annotation.
- The `@JoinTable` annotation defines the join table between two entities

- In a bidirectional relationship of `@ManyToMany`, only one entity can own the relationship. Here we choose Courses as the owning entity. We usually mention `mappedBy` parameter on the owning entity.

```
@Entity
public class Courses extends BaseEntity{

    @ManyToMany(mappedBy = "courses", fetch = FetchType.EAGER
                ,cascade = CascadeType.PERSIST)
    private Set<Person> persons = new HashSet<>();
}
```

# Many to Many inside JPA

- Why do we really need a third table in Many to Many relationship? Lets assume we have below tables and records which has many-many relationship between them.
- If we need to represent multiple courses that a same student enrolled, the best way is to maintain a middle third table like shown below. Otherwise we need to maintain lot of duplicate rows inside Person table.

COURSE ID	NAME	FEES
I23	Music	\$ 1200
I24	Yoga	\$ 800
I25	Football	\$ 1500

COURSES Table

PERSON ID	COURSE ID
456	I23
456	I24
456	I25

PERSON\_COURSES Table

PERSON ID	NAME	EMAIL
456	John	j@gmail.com
457	Peter	p@gmail.com
458	Anna	a@gmail.com

PERSONS Table

# Sorting

## Introduction

- ✓ Spring Data JPA supports Sorting to Query results with easier configurations.
- ✓ Spring Data JPA provides default implementations of Sorting with the help of PagingAndSortingRepository interface
- ✓ There are two ways to achieve Sorting in Spring Data JPA:
  - 1) Static Sorting
  - 2) Dynamic Sorting

## Static Sorting

✓ Static sorting refers to the mechanism where the retrieved data is always sorted by specified columns and directions. The columns and sort directions are defined at the development time and cannot be changed at runtime.

✓ Below is an examples of Static Sorting,

```
List<Person> findByOrderByNameDesc()
```

## Dynamic Sorting

✓ By using dynamic sorting, you can choose the sorting column and direction at runtime to sort the query results.  
✓ Dynamic sorting provides more flexibility in choosing sort columns and directions with the help of Sort parameter to your query method. The Sort class is just a specification that provides sorting options for database queries. Below is an example,

```
Sort sort=Sort.by("name").descending()  
.and(Sort.by("age"));
```

# Pagination

## Introduction

- ✓ Spring Data JPA supports Pagination which helps easy to manage & display large amount of data in various pages inside web applications
- ✓ Just like the special Sort parameter, we have for the dynamic sorting, Spring Data JPA supports another special parameter called **Pageable** for paginating the query results.
- ✓ We can combine both pagination and dynamic sorting with the help of **Pageable**

## Dynamic Sorting

- ✓ Whenever we want to apply pagination to query results, all we need to do is just add **Pageable** parameter to the query method definition and set the return by **Page<T>** like below,

```
Pageable pageable = PageRequest.of(0, 5, Sort.by("name").  
descending());
```

```
Page<Person> findByName(String name, Pageable pageable);
```

# Custom Queries with JPA

## Introduction

- ✓ Derived queries are good as long as they are not complex. As the number of query parameters goes beyond 3 to 4, you need a more flexible strategy.
- ✓ For such complicated or custom scenarios, Spring Data JPA allow developers to write their own queries with the help of below annotations,

`@Query`

`@NamedQuery`

`@NamedNativeQuery`

## `@Query Annotation`

- ✓ The `@Query` annotation defines queries directly on repository methods. This gives you full flexibility to run any query without following the method naming conventions.
- ✓ With the help of `@Query` annotation we can write queries in the form of JPQL or Native SQL query.
- ✓ When ever we are writing a native SQL query then we need to mention `nativeQuery = true` inside `@Query` annotation

## Named Queries

- ✓ For bigger applications where they may have 1000s of queries scattered across the application, it would make sense for them to maintain all these queries in a single place logically by using annotations, properties and XML files.
- ✓ We can create named queries easily with the below annotations on top of an entity class,

`@NamedQuery` - Used to define a JPQL named query.

`@NamedNativeQuery` - Used to define a native SQL named query.

# JPQL

## Introduction

- ✓ The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.
- ✓ JPQL is used to make queries against entities stored in a relational database. It is heavily inspired by SQL, and its queries resemble SQL queries in syntax, but operate against JPA entity objects rather than directly with database tables.
- ✓ The only drawback of using JPQL is that it supports a subset of the SQL standard. So, it may not be a great choice for complex queries.

## JPQL Example

- ✓ Below is an example of JPQL. You can observe we are using the entity names and fields present inside it instead of using table and column names,

```
@Query("SELECT c FROM Contact c WHERE c.contactId = ?1  
ORDER BY c.createdAt DESC")  
List<Contact> findByIdOrderByCreatedDesc(long id);
```

## Sorting Examples

*Static Sorting using derived query from method name*

```
List<Person> findByNameOrderByAgeDesc(String name);
```

*Static Sorting using @Query annotation with JPQL*

```
@Query("SELECT p FROM person p WHERE p.age > ?1 ORDER BY p.name DESC")
List<Person> findByAgeGreaterThanJPQL(int age);
```

*Static Sorting using @Query annotation with native query*

```
@Query(value = "SELECT * FROM person p WHERE p.name = :givenName ORDER BY p.age ASC",
       nativeQuery = true)
List<Person> findByGivenNameNativeSQL(@Param("givenName") String givenName);
```

# Sorting Examples

## Static Sorting using `NamedQuery` & `NamedNativeQuery` annotations

```
@NamedQuery(name = "Person.findByAgeGreaterThanNamedJPQL",
    query = "SELECT p FROM Person p WHERE p.age > :age ORDER BY p.name ASC")
@NamedNativeQuery(name = "Person.findAllNamedNativeSQL",
    query = "SELECT * FROM person p ORDER BY p.age DESC")
@Entity
public class Person extends BaseEntity{
```

Dynamic Sorting using `Sort` parameter. The `Sort` parameter can be passed with `@Query`, `@NamedQuery` annotations. Sort fields can be add dynamically as well based on the request params from the UI/API.

```
Sort sort = Sort.by("name").descending().and(Sort.by("age"));
List<Person> persons = personRepository.findByName("John", sort);
```

Spring Data JPA does not support dynamic sorting for native SQL queries, as it would require the updating of the actual SQL query defined, which cannot be done correctly by the Spring Data JPA.

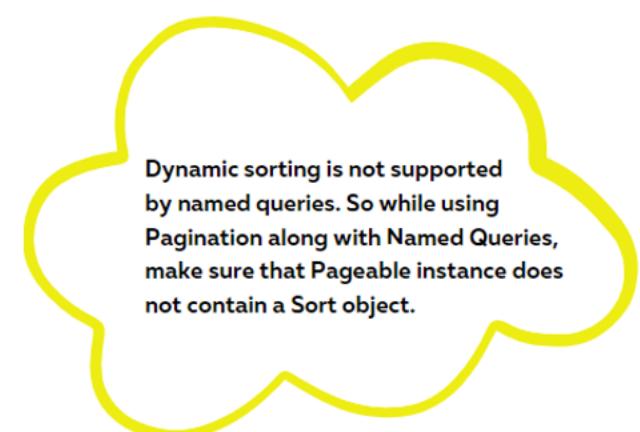
# Pagination Examples

*Below is an example where we are telling to JPA to fetch the first page by considering the total page size as 5*

```
Pageable pageable = PageRequest.of(0, 5);
Page<Contact> msgPage = contactRepository.findByStatus("Open", pageable);
```

*Below is an example where we are applying both pagination & sorting dynamically based on the input received.*

```
public Page<Contact> findMsgsWithOpenStatus(int pageNum, String sortField,
                                              String sortDir){
    int pageSize = 5;
    Pageable pageable = PageRequest.of(pageNum - 1, pageSize,
                                         sortDir.equals("asc") ? Sort.by(sortField).ascending()
                                         : Sort.by(sortField).descending());
    Page<Contact> msgPage = contactRepository.findByStatus("Open", pageable);
    return msgPage;
}
```



Dynamic sorting is not supported by named queries. So while using Pagination along with Named Queries, make sure that Pageable instance does not contain a Sort object.

## **@Query example with positional parameters**

```
// Positional parameters  
@Query("SELECT c FROM Contact c WHERE c.status = ?1 AND c.name = ?2 ORDER BY c.createdAt DESC")  
List<Contact> findByGivenQueryOrderByCreatedDesc(String status, String name);
```

## **@Query example with named parameters**

Using @Query, we can also run UPDATE, DELETE, INSERT as well

```
@Transactional  
@Modifying  
@Query("UPDATE Contact c SET c.status = ?1 WHERE c.contactId = ?2")  
int updateStatusById(String status, int id);
```

Whenever we are using queries that change the state of the database, they should be treated differently. We need to explicitly tell Spring Data JPA that our custom query changes the data by annotating the repository method with an additional @Modifying annotation. It will then execute the custom query as an update operation.

On whichever method/class we declare @Transactional the boundary of transaction starts and boundary ends when method execution completes. Lets say you are updating entity1 and entity2. Now while saving entity2 an exception occur, then as entity1 comes in same transaction so entity1 will be rollback with entity2. Any exception will result in rollback of all JPA transactions with DB.

## @NamedQuery & @ NamedNativeQuery Examples

eazy  
bytes

@NamedQuery example declared on top of the entity class

```
@Entity
@NamedQuery(name="Contact.findOpenMsgs",query = "SELECT c FROM Contact c WHERE c.status = :status")
public class Contact extends BaseEntity{
```

@NamedNativeQuery example declared on top of the entity class

```
@Entity
@NamedNativeQuery(name = "Contact.findOpenMsgsNative",
    query = "SELECT * FROM contact_msg c WHERE c.status = :status",resultClass = Contact.class)
public class Contact extends BaseEntity{
```

For @NamedQuery as long as the method name inside Repository class matches with the name of the query we should be good. Where as for @NamedNativeQuery apart from query name & method name match, we should also mention @Query(nativeQuery = true) on top of the Repository method.

## QUICK TIP

Do you know we can create multiple named queries and named native queries using the annotations `@NamedQueries` and `@NamedNativeQueries`. Below is the syntax of the same,

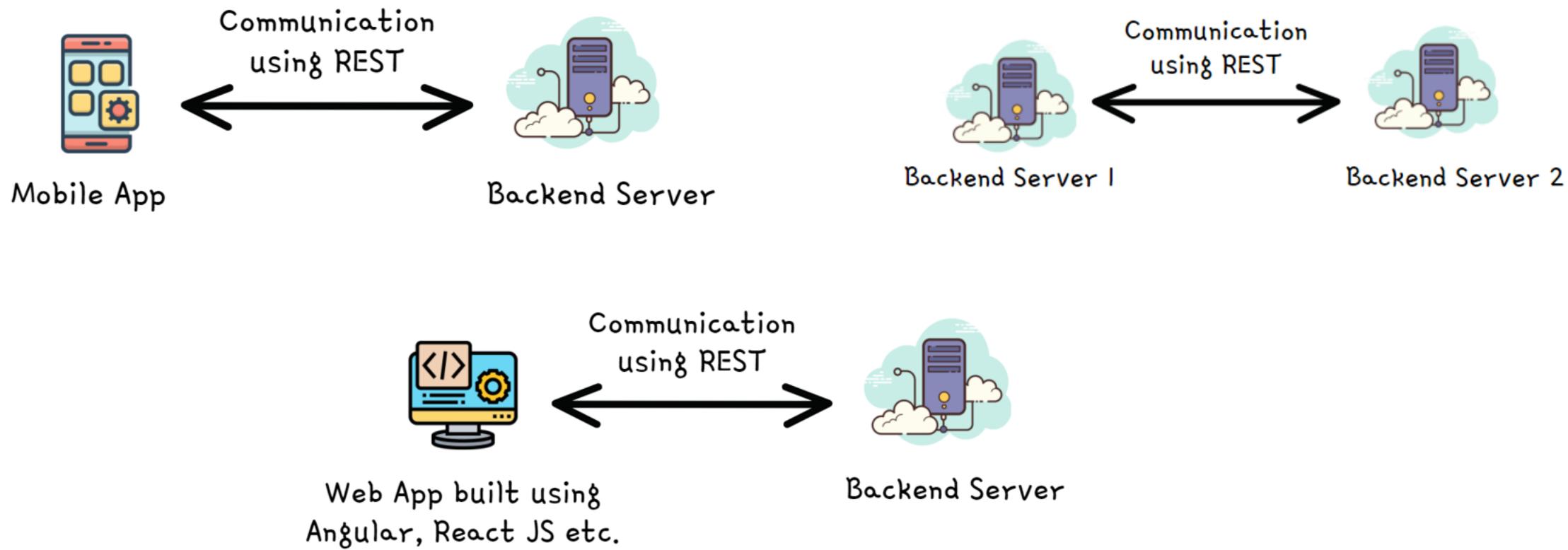
```
@NamedQueries({
    @NamedQuery(name = "one", query = "?"),
    @NamedQuery(name = "two", query = "?")
})
```

```
@NamedNativeQueries({
    @NamedNativeQuery(name = "one", query = "?", resultClass = ?),
    @NamedNativeQuery(name = "two", query = "?", , resultClass = ?)
})
```



# Implementing REST Services

- REST (Representational state transfer) services are one of the most often encountered ways to implement communication between two web apps. REST offers access to functionality the server exposes through endpoints a client can call.
- Below are the different use cases where REST services are being used most frequently these days,



# Implementing REST Services

eazy  
bytes

Below is the sample code implementing Rest Service using Spring MVC style but only with the addition of @ResponseBody annotation

```
@Controller
public class ContactRestController {

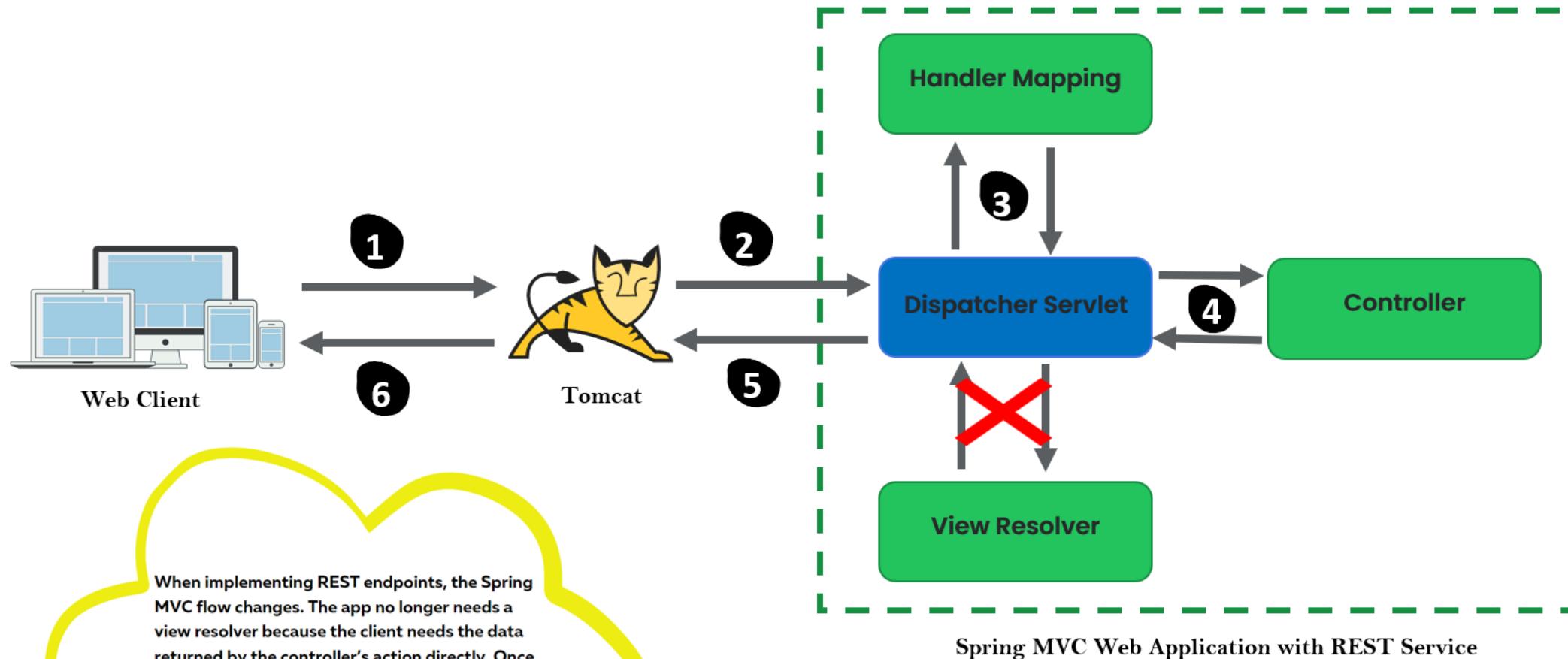
    @Autowired
    ContactRepository contactRepository;

    @GetMapping("/getMessagesByStatus")
    @ResponseBody
    public List<Contact> getMessagesByStatus(@RequestParam(name = "status") String status) {
        return contactRepository.findByStatus(status);
    }

}
```

The @ResponseBody annotation tells the dispatcher servlet that the controller's action will not return a view name but the data sent directly in the HTTP response.

# SPRING MVC ARCHITECTURE WITH REST SERVICE



When implementing REST endpoints, the Spring MVC flow changes. The app no longer needs a view resolver because the client needs the data returned by the controller's action directly. Once the controller's action completes, the dispatcher servlet returns the HTTP response without rendering any view.

# Implementing REST Services

eazy  
bytes

Instead of mentioning `@ResponseBody` on each method inside a `Controller` class which is a duplicate code, Spring offers the `@RestController` annotation, a combination of `@Controller` and `@ResponseBody`.

```
@RestController
public class ContactRestController {
    @Autowired
    ContactRepository contactRepository;

    @GetMapping("/getMessagesByStatus")
    public List<Contact> getMessagesByStatus(@RequestParam(name = "status") String status) {
        return contactRepository.findByStatus(status);
    }
}
```

**`@RestController` = `@Controller + @ResponseBody`**

# Different Annotation & Classes in REST

eazy  
bytes

`@RestController` – can be used to put on top of a call. This will save developers from mentioning `@ResponseBody` on each methods

`@ResponseBody` – can be used on top of a method to build a Rest API when we are using `@Controller` on top of a Java class

`ResponseEntity<T>` – Allow developers to send response body, status, and headers on the HTTP response.

`@RestControllerAdvice` – is used to mark the class as a REST controller advice. Along with `@ExceptionHandler`, this can be used to handle exceptions globally inside app.

`RequestEntity<T>` – Allow developers to receive the request body, header in a HTTP request.

`@RequestHeader & @RequestBody` – is used to receive the request body and header individually.

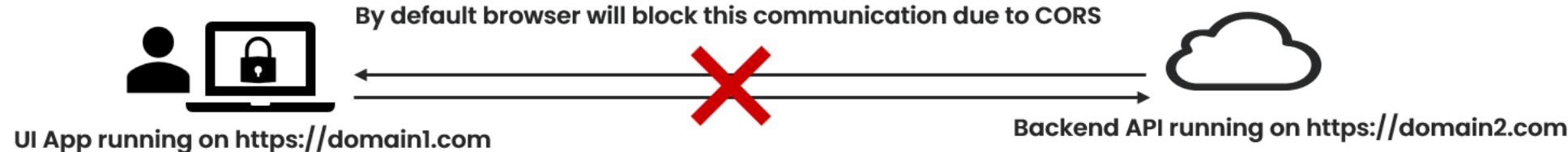
# CROSS-ORIGIN RESOURCE SHARING (CORS)

eazy  
bytes

CORS is a protocol that enables scripts running on a browser client to interact with resources from a different origin. For example, if a UI app wishes to make an API call running on a different domain, it would be blocked from doing so by default due to CORS. So CORS is not a security issue/attack but the default protection provided by browsers to stop sharing the data/communication between different origins.

"other origins" means the URL being accessed differs from the location that the JavaScript is running from, by having:

- a different scheme (HTTP or HTTPS)
- a different domain
- a different port



# CROSS-ORIGIN RESOURCE SHARING (CORS)

eazy  
bytes

If we have a valid scenario, where a Web APP UI deployed on a server is trying to communicate with a REST service deployed on another server, then these kind of communications we can allow with the help of `@CrossOrigin` annotation. `@CrossOrigin` allows clients from any domain to consume the API.

`@CrossOrigin` annotation can be mentioned on top of a class or method like mentioned below,

```
@CrossOrigin(origins = "http://localhost:8080") // Will allow on specified domain
```

```
@CrossOrigin(origins = "*") // Will allow any domain
```



## QUICK TIP

eazy  
bytes

Do you know Jackson library provide annotations to control the way we send data in REST API JSON response. Below are the few annotations,

**@JsonProperty** – This annotation can be mentioned on top of any field of a POJO class. While sending the JSON response, instead of sending the field name Jackson will send the property name that we mentioned.

```
@JsonProperty("person_name")
private String name;
```

**@JsonIgnore** – This annotation will make sure to not send the information present inside the given field. This is useful to filter the data which is sensitive or unnecessary in the response.

```
@JsonIgnore
private LocalDateTime createdAt;
```

We can also use **@JsonIgnoreProperties(value = { "createdAt" })** on top of a POJO class if we want to mention multiple fields.



# Consuming REST Services

eazy  
bytes

- Apart from building the Rest service, often we may need to consume the Rest Services exposed by other third party vendors. So knowing how to consume Rest services is equally important.
- Below are the most commonly used approaches that are provided by Spring framework,

**OpenFeign** – A tool offered by the Spring Cloud project. Using this very similar to like how build Repositories with Spring Data JPA. In similar way we just need to write interfaces but not implementation code.

**RestTemplate** – A well-known tool developers have been using since Spring 3 to call REST endpoints. RestTemplate is often used today in Spring apps. But this is deprecated in the favor of WebClient.

**WebClient** – Created as part of the Spring Web Reactive module, and will be replacing the classic RestTemplate. This is introduced to support all modes of invocation like Sync and Async (non-blocking)

# Consuming REST Services using OpenFeign

eazy  
bytes

In order to consume the REST services using OpenFeign, we need to follow the below steps.

Step 1 : After adding all the required dependencies inside the pom.xml, we need to create a proxy interface with all the details around API that we are going to consume. Inside the interface we need to create method name matching the details of the destination API method we are going to consume,

```
@FeignClient(name = "contact", url = "http://localhost:8080/api/contact",
    configuration = ProjectConfiguration.class)
public interface ContactProxy {

    @RequestMapping(method = RequestMethod.GET, value = "/getMessagesByStatus")
    @Headers(value = "Content-Type: application/json")
    public List<Contact> getMessagesByStatus(@RequestParam("status") String status);

}
```

# Consuming REST Services using OpenFeign

Step 2 : If we need to send the authentication details, then create a bean with the required details,

```
@Bean
public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("admin@eazyschool.com", "admin");
}
```

Step 3 : Finally we are good to use the proxy object to make a Rest call like shown below,

```
@Autowired
ContactProxy contactProxy;

@GetMapping("/getMessages")
public List<Contact> getMessages(@RequestParam("status") String status) {
    return contactProxy.getMessagesByStatus(status);
}
```

# Consuming REST Services using RestTemplate

eazy  
bytes

In order to consume the REST services using RestTemplate, we need to follow the below steps.

Step 1 : After adding all the required dependencies inside the pom.xml, we need to create a RestTemplate bean along with the authentication details if any like below,

```
@Bean
public RestTemplate restTemplate() {
    RestTemplateBuilder restTemplateBuilder =
        new RestTemplateBuilder();
    return restTemplateBuilder.basicAuthentication
        ("admin@eazyschool.com", "admin").build();
}
```

# Consuming REST Services using RestTemplate

eazy  
bytes

Step 2 : Using RestTemplate methods like exchange(), we can consume a Rest Service like mentioned below,

```
@PostMapping("/saveMsg")
public ResponseEntity<Response> saveMsg(@RequestBody Contact contact){
    String uri = "http://localhost:8080/api/contact/saveMsg";
    HttpHeaders headers = new HttpHeaders();
    headers.add("invocationFrom", "RestTemplate");
    HttpEntity<Contact> httpEntity = new HttpEntity<>(contact, headers);
    ResponseEntity<Response> responseEntity = restTemplate.exchange(uri, HttpMethod.POST,
        httpEntity, Response.class);
    return responseEntity;
}
```

# Consuming REST Services using WebClient

eazy  
bytes

In order to consume the REST services using WebClient, we need to follow the below steps.

Step 1 : After adding all the required dependencies inside the pom.xml, we need to create a WebClient bean along with the authentication details if any like below,

```
@Bean
public WebClient webClient() {
    return WebClient.builder()
        .filter(ExchangeFilterFunctions.
            basicAuthentication("admin@eazyschool.com", "admin"))
        .build();
}
```

# Consuming REST Services using WebClient

eazy  
bytes

Step 2 : Using WebClient methods like post(), we can consume a Rest Service like mentioned below,

```
@Autowired
WebClient webClient;

@PostMapping("/saveMessage")
public Mono<Response> saveMessage(@RequestBody Contact contact){
    String uri = "http://localhost:8080/api/contact/saveMsg";
    return webClient.post().uri(uri)
        .header("invocationFrom", "WebClient")
        .body(Mono.just(contact), Contact.class)
        .retrieve()
        .bodyToMono(Response.class);
}
```

# Spring Data REST

eazy  
bytes

- Apart from doing a magic of automatically creating repository implementations based on interfaces you define in your code, Spring Data has another feature that can help you define REST APIs for repositories created by Spring Data.
- For the same, we have Spring Data REST which is another member of the Spring Data family.
- To start using Spring Data REST in our project, we just need to add the following dependency to our pom.xml,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```



We just need to add the Spring Data Rest starter project and believe it or not, that's the only change is expected from Developer. Post that the application gets auto-configuration that enables automatic creation of a REST API for any repositories that were created by Spring Data

The REST APIs will be created for any kind of implementations like Spring Data JPA, Spring Data Mongo etc. The REST endpoints that Spring Data REST generates are at least as good as (and possibly even better than) the ones Developers creates ☺.

By adding the HAL Explorer along with Spring Data REST, we can look all the APIs exposed by Spring Data REST by opening the <http://localhost:8080/> URL in the browser assuming that your App started at 8080 port itself.

# HAL Explorer

eazy  
bytes

- HAL (Hypertext Application Language) is a simple format that gives a consistent and easy way to hyperlink between resources in your API.
- Adopting HAL will make your API explorable, and its documentation easily discoverable from within the API itself. In short, it will make your API easier to work with and therefore more attractive to client developers.
- To start using HAL explorer along with Spring Data REST in our project, we just need to add the following dependency to our pom.xml,

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-hal-explorer</artifactId>
</dependency>
```

The screenshot shows the HAL Explorer interface. At the top, there's a navigation bar with 'HAL Explorer', 'Theme', 'Layout', and 'About'. Below that is a search bar with 'Edit Headers' and 'Go!' button, and a URL field with '/data-api/'. The main area has several sections: 'Links' (a table with columns: Relation, Name, Title, HTTP Request, Doc), 'Response Status' (200 (OK)), 'Response Headers' (cache-control: no-cache, no-store, max-age=0, must-revalidate; connection: keep-alive; content-type: application/hal+json; date: Fri, 21 Jan 2022 02:16:21 GMT; expires: 0; keep-alive: timeout=60; pragma: no-cache; transfer-encoding: chunked; vary: Origin, Access-Control-Request-Method, Access-Control-Request-Headers; x-content-type-options: nosniff; x-frame-options: DENY; x-xss-protection: 1; mode=block), and 'Response Body' (a JSON snippet showing the structure of a HAL resource).



## QUICK TIP

---

eazy  
bytes

Do you know we can change the default path exposed by Spring Data REST using the below configurations. This will change the default path as per your configurations.

```
spring.data.rest.basePath=/data-api
```

Using `@RepositoryRestResource` annotation, we can also control the way the paths of the repositories are being exposed.

```
@RepositoryRestResource(path = "courses")
```

For some reason, if you don't want to expose a Repository, then we can do the below configuration on top of the Repository class,

```
@RepositoryRestResource(exported = false)
```



# Logging

## Introduction

- ✓ By default we don't have to worry about logging if we are using Spring Boot. Most of the configurations and logging done by the Spring Boot itself.
- ✓ Usually we have the following types of logging,

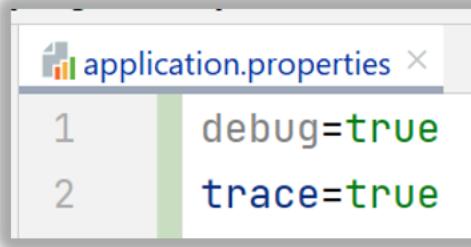
- FATAL (Logback doesn't have)
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

## Logging Types

- ✓ In Java we have many logging frameworks like Java Util logging, Log4J2, SLF4J, Logback. By default, if you use the “Starters”, Logback is used for logging.
- ✓ Appropriate Logback routing is also included in Spring Boot to ensure that dependent libraries that use Java Util Logging, Commons Logging, Log4J, or SLF4J all work correctly.
- ✓ By default, ERROR-level, WARN-level, and INFO-level messages are logged. But we can change them based on our requirements and environments.

# Logging inside SpringBoot

- We can enable debug logging or trace logging by mentioning the below properties inside application.properties file,

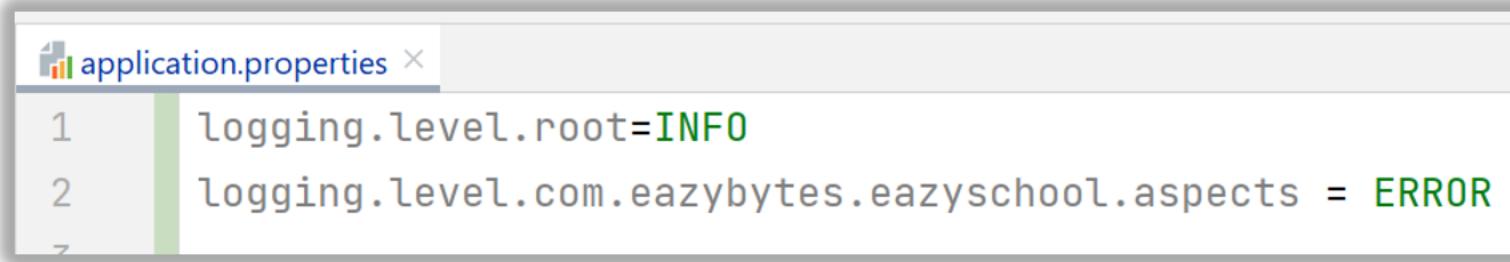


```
application.properties
1 debug=true
2 trace=true
```

Alternatively we can provide the flags while starting the application like mentioned below,

```
$ java -jar myapp.jar --debug
```

- The way logging works is if we enable trace then all above severities like Debug, Info, Warn & Error are also printed. Suppose if we enable error logging only, then all lower severities like Warn, Info, Debug, Trace will not be logged.
- If needed we can control the logging at the package level by mentioning properties like below inside application.properties,

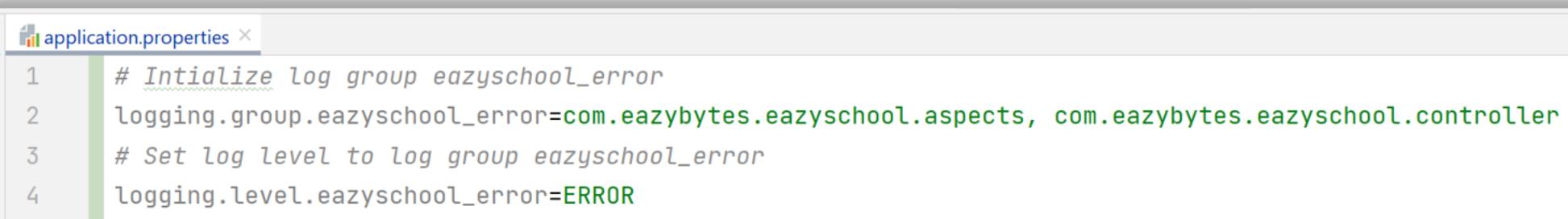


```
application.properties
1 logging.level.root=INFO
2 logging.level.com.eazybytes.eazyschool.aspects = ERROR
```

# Logging inside SpringBoot

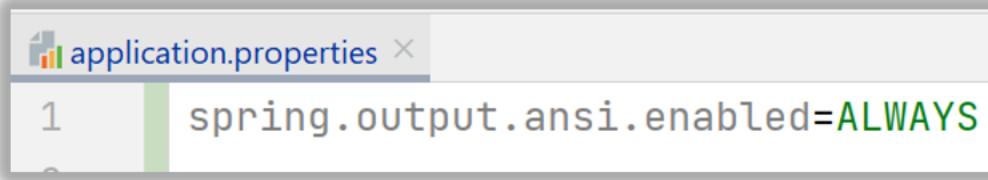
eazy  
bytes

- It is often useful to be able to group related loggers together so that they can all be configured at the same time. For example, I may have an requirement to change the logging levels for all my project related packages very frequently. To help with this, Spring Boot allows you to define logging groups. Below is the sample configuration,



```
# Initialize log group eazyschool_error
logging.group.eazyschool_error=com.eazybytes.eazyschool.aspects, com.eazybytes.eazyschool.controller
# Set log level to log group eazyschool_error
logging.level.eazyschool_error=ERROR
```

- If your terminal supports ANSI, color output is used to aid readability of your logs. You can set the below property to enable the same.



```
spring.output.ansi.enabled=ALWAYS
```

# Logging inside SpringBoot

eazy  
bytes

- Below is the default logging format in Spring Boot. Below is a sample logger message and format details of it,

```
2025-01-21 08:02:46.035 INFO 15084 [restartedMain] c.e.eazyschool.EazyschoolApplication : Started  
EazyschoolApplication in 5.189 seconds (JVM running for 2123.116)
```

- ✓ Date and Time: Millisecond precision and easily sortable.
- ✓ Log Level: ERROR, WARN, INFO, DEBUG, or TRACE.
- ✓ Process ID.
- ✓ A --- separator to distinguish the start of actual log messages.
- ✓ Thread name: Enclosed in square brackets (may be truncated for console output).
- ✓ Logger name: This is usually the source class name (often abbreviated).

The log message.

- By default, Spring Boot logs only to the console and does not write log files. If you want to write log files in addition to the console output, we can create a file with the name `logback.xml` inside class path and define all our logging requirements in it.

Do you know Lombok has various annotations to help developers with logging based on the logging framework being used,

`@Slf4j` will generate the below code and we can use the `log` variable directly,

```
@Slf4j
public class LogExample {

}
```

will generate:

```
public class LogExample {
    private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(LogExample.class);
}
```

Similarly using `@CommonsLog`, `@Log4j2` will generate log variable from the respective library class.



# Configurations

### Introduction

✓ Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources, include Java properties files, YAML files, environment variables, and command-line arguments.

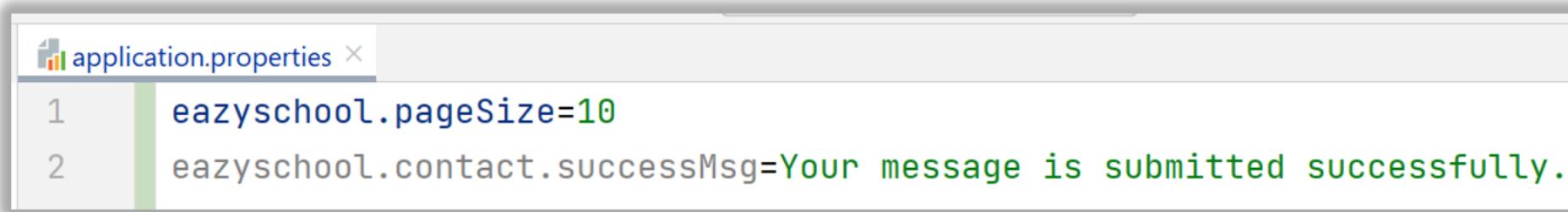
✓ By default, Spring Boot look for the configurations or properties inside application.properties/yaml present in the classpath location. But we can have other property files as well and make SpringBoot to read from them.

### Config/Properties Preferences

- ✓ Spring Boot uses a very particular order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones):
  - Properties present inside files like application.properties
  - OS Environmental variables
  - Java System properties (System.getProperties())
  - JNDI attributes from java:comp/env.
  - ServletContext init parameters.
  - ServletConfig init parameters.
  - Command line arguments.

## Reading properties with @Value

- We can read the properties/configurations, defined inside a properties file with the help of @Value annotation like shown below,



@Value annotation leverages SpEL expressions to read the configurations present inside the properties file

```
@Controller
public class DashboardController {

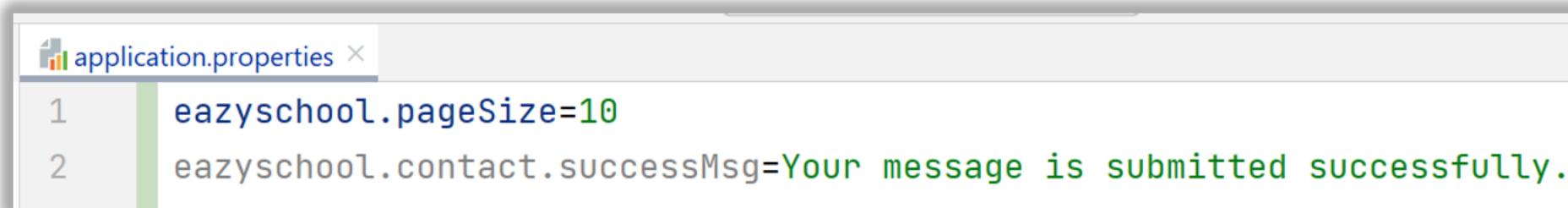
    @Value("${eazyschool.pageSize}")
    private int defaultPageSize;

    @Value("${eazyschool.contact.successMsg}")
    private String message;
```

# Reading properties using Environment

eazy  
bytes

- Along with @Value, we can read the properties/configurations loaded with the help of Environment bean as well which is created by Spring framework. Apart from user defined properties, using Environment we can read any environment specific properties as well.



```
application.properties
1 easyschool.pageSize=10
2 easyschool.contact.successMsg=Your message is submitted successfully.
```

```
@Controller
public class DashboardController {

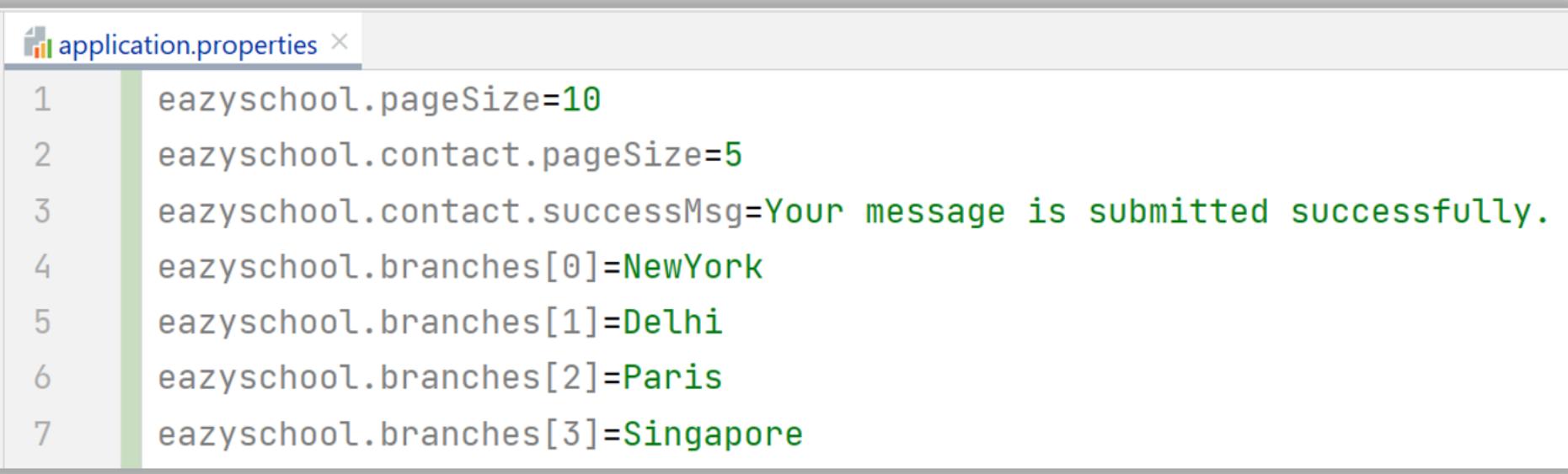
    @Autowired
    private Environment environment;

    private void logProperties() {
        log.info(environment.getProperty("easyschool.pageSize"));
        log.info(environment.getProperty("easyschool.contact.successMsg"));
        log.info(environment.getProperty("JAVA_HOME"));
    }
}
```

# Reading properties with @ConfigurationProperties

eazy  
bytes

- *SpringBoot allow to load all the properties which are logically together into a java bean. For the same we can use @ConfigurationProperties annotation on top of a java bean by providing the prefix value. We need to make sure to use the names inside bean and properties file.*
- *Please follow below steps to read the properties using @ConfigurationProperties ,*
- *Step 1 : We need to maintain the properties like below which have same prefix like 'eazyschool'*



```
application.properties ×
1  eazyschool.pageSize=10
2  eazyschool.contact.pageSize=5
3  eazyschool.contact.successMsg=Your message is submitted successfully.
4  eazyschool.branches[0]=NewYork
5  eazyschool.branches[1]=Delhi
6  eazyschool.branches[2]=Paris
7  eazyschool.branches[3]=Singapore
```

# Reading properties with `@ConfigurationProperties`

eazy  
bytes

- Step 2 : Create a bean like below with all the required details,

```
@Component("eazySchoolProps")
@Data
@PropertySource("classpath:some.properties")
@ConfigurationProperties(prefix = "eazyschool")
@Validated
public class EazySchoolProps {

    @Min(value=5, message="must be between 5 and 25")
    @Max(value=25, message="must be between 5 and 25")
    private int pageSize;
    private Map<String, String> contact;
    private List<String> branches;
}
```

`@PropertySource` – can be used to mention the property file name if we are using something other than `application.properties`

`@ConfigurationProperties` – can be used to mention the prefix value that needs to be considered while loading the properties into a given bean

`@Validated` – can be used if we want to perform validations on the properties based on the validation mentioned on the field

# Reading properties with `@ConfigurationProperties`

eazy  
bytes

- Step 3 : Finally we can inject the bean which we created in the previous step and start reading the properties from it using java style like shown below,

```
❶ @Service
public class ContactService {

    ❷     @Autowired
    ❸     EazySchoolProps eazySchoolProps;

    ❹     public Page<Contact> findMsgsWithOpenStatus(int pageNum, String sortField, String sortDir){
        ❺     int pageSize = eazySchoolProps.getPageSize();
        ❻     if(null!=eazySchoolProps.getContact() && null!=eazySchoolProps.getContact().get("pageSize")){
            ❼     pageSize = Integer.parseInt(eazySchoolProps.getContact().get("pageSize").trim());
        }
    }
}
```

# Reading properties with @PropertySource

eazy  
bytes

- Sometimes we maintain properties inside files which has name is not equal to application.properties. In those scenarios, if we try to use the @Value, it will not work.
- First we need to communicate to SpringBoot about the property files with the below steps,

We need to create a class with @Configuration and @PropertySource annotation like mentioned below. Here we need to mention the property file name. Post these changes, we can refer the properties present inside the config.properties using @Value annotation or Environment bean. ignoreResourceNotFound = true will not throw an exception in case the file is missing.

```
@Configuration
@PropertySource(value = "classpath:config.properties", ignoreResourceNotFound = true)
public class AppConfig {  
}
```

We can configure multiple property files as well using @PropertySources annotation like mentioned here,

```
@Configuration
@PropertySources({
    @PropertySource(value = "classpath:config.properties", ignoreResourceNotFound = true),
    @PropertySource("classpath:server.properties")
})
public class AppConfig {  
}
```

# Profiles

## Introduction

- ✓ Spring provides a great tool for grouping configuration properties into so-called profiles(dev, uat, prod) allowing us to activate a bunch of configurations based on the active profile.
- ✓ Profiles are perfect for setting up our application for different environments, but they're also being used in another use cases like Bean creation based on a profile etc.
- ✓ So basically a profile can influence the application properties loaded and beans which are loaded into the Spring context.

## Configuring Profiles

- ✓ The default profile is always active. Spring Boot loads all properties in application.properties into the default profile.
- ✓ We can create another profiles by creating property files like below,
  - application\_prod.properties -----> for prod profile
  - application\_uat.properties -----> for uat profile
- ✓ We can activate a specific profile using spring.profiles.active property like below,

```
spring.profiles.active=prod
```

Do you know there are many ways to activate a profile. Below are the most commonly used.

- ✓ By mentioning `spring.profiles.active=prod` inside the properties files.
- ✓ Using environment variables like below,

```
export SPRING_PROFILES_ACTIVE=prod  
java -jar myApp-0.0.1-SNAPSHOT.jar
```

- ✓ Using Java System property,

```
java "-Dspring-boot.run.profiles=prod" -jar myApp-0.0.1-SNAPSHOT.jar  
mvn spring-boot:run "-Dspring-boot.run.profiles=prod"
```

- ✓ Activating a profile programmatically invoking the method `setAdditionalProfiles("prod")` inside `SpringApplication` class.
- ✓ Using `@ActiveProfiles` while doing testing,

```
@SpringBootTest  
@ActiveProfiles({"uat"})
```



# Conditional Bean creation using Profiles

eazy  
bytes

- With the help of Profiles we can create the Bean conditionally. Below is an example where we can create different beans based on the active profile,

```
@Component
@Profile("!prod")
public class EazySchoolNonProdUsernamePwdAuthenticationProvider
    implements AuthenticationProvider
{
```

```
@Component
@Profile("prod")
public class EazySchoolUsernamePwdAuthenticationProvider
    implements AuthenticationProvider
{
```

# Spring Boot Actuator

eazy  
bytes

- Actuator offers production-ready features such as monitoring and metrics to Spring Boot applications. Actuator's features are provided by way of several endpoints, which are made available over HTTP.
- To enable Actuator, we can mention the below dependency inside pom.xml,

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

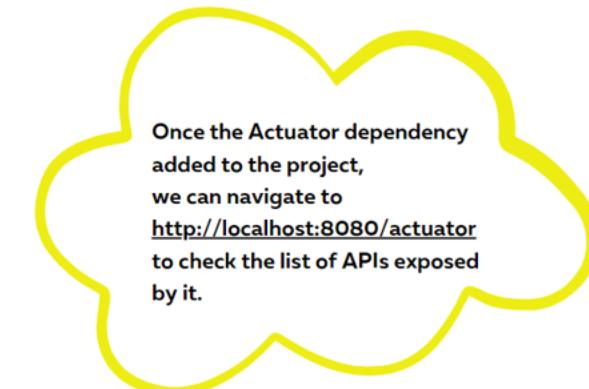
In a machine, an actuator is a component that's responsible for controlling and moving a mechanism. In a Spring Boot application, the Spring Boot Actuator plays that same role, enabling us to see inside of a running application and, to some degree, control how the application behaves.

Using endpoints exposed by Actuator, we can know the following type of info,

- ✓ Health
- ✓ Configuration properties
- ✓ Logging levels
- ✓ Memory consumption
- ✓ Beans information
- ✓ Metrics etc.

By default, Actuator doesn't expose many of the endpoints since they have sensitive information. We can expose them using the below property,

```
management.endpoints.web.exposure.include=*
```



Once the Actuator dependency added to the project, we can navigate to <http://localhost:8080/actuator> to check the list of APIs exposed by it.

## API paths provided by Actuator

HTTP method	Path	Description
GET	/auditevents	Produces a report of any audit events that have been fired.
GET	/conditions	Produces a report of autoconfiguration conditions that either passed or failed, leading to the beans created in the application context.
GET	/configprops	Describes all configuration properties along with the current values.
GET	/beans	Describes all the beans in the Spring application context.
GET, POST, DELETE	/env	Produces a report of all property sources and their properties available to the Spring application.
GET	/env/{toMatch}	Describes the value of a single environment property.
GET	/heapdump	Downloads the heap dump.
GET	/health	Returns the aggregate health of the application and (possibly) the health of external dependent applications.
GET	/httptrace	Produces a trace of the most recent 100 requests.
GET	/info	Returns any developer-defined information about the application.
GET	/loggers	Produces a list of packages in the application along with their configured and effective logging levels.

## API paths provided by Actuator

HTTP method	Path	Description
GET, POST	/loggers/{name}	Returns the configured and effective logging level of a given logger. The effective logging level can be set with a POST request.
GET	/mappings	Produces a report of all HTTP mappings and their corresponding handler methods.
GET	/scheduledtasks	Lists all scheduled tasks.
GET	/threaddump	Returns a report of all application threads.
GET	/metrics	Returns a list of all metrics categories.
GET	/metrics/{name}	Returns a multidimensional set of values for a given metric.



# Deploying Spring Boot Applications



Spring Boot's flexible packaging options provide a great deal of choice when it comes to deploying your application.

You can deploy Spring Boot applications to a variety of cloud platforms, to virtual/real machines

The SpringBoot Apps can be deployed to

- ✓ Public Clouds like AWS, Azure, GCP
- ✓ Kubernetes
- ✓ Heroku
- ✓ OpenShift etc.

Since **AWS** is a famous cloud provider used by majority of the Organizations, let's explore on how to deploy a SpringBoot application using AWS

The common approaches to deploy a Spring Boot inside AWS is using either **EC2** or **Elastic Beanstalk**

# AWS EC2 vs AWS Elastic Beanstalk

**Amazon Elastic Compute Cloud (EC2)** is a virtual cloud infrastructure service offered by AWS that provides users on-demand computing resources through which users can create powerful servers in the cloud. Additionally, EC2 enables users to get a virtual machine up and running in just a few clicks.

With EC2, we can deploy our Spring Boot application into AWS Cloud. But it is a traditional approach and has many drawbacks.

When used EC2, the Developer or Organization, **need to take care of installing required software, libraries like Java, Tomcat and handle Auto scaling, load balancing etc. manually.**

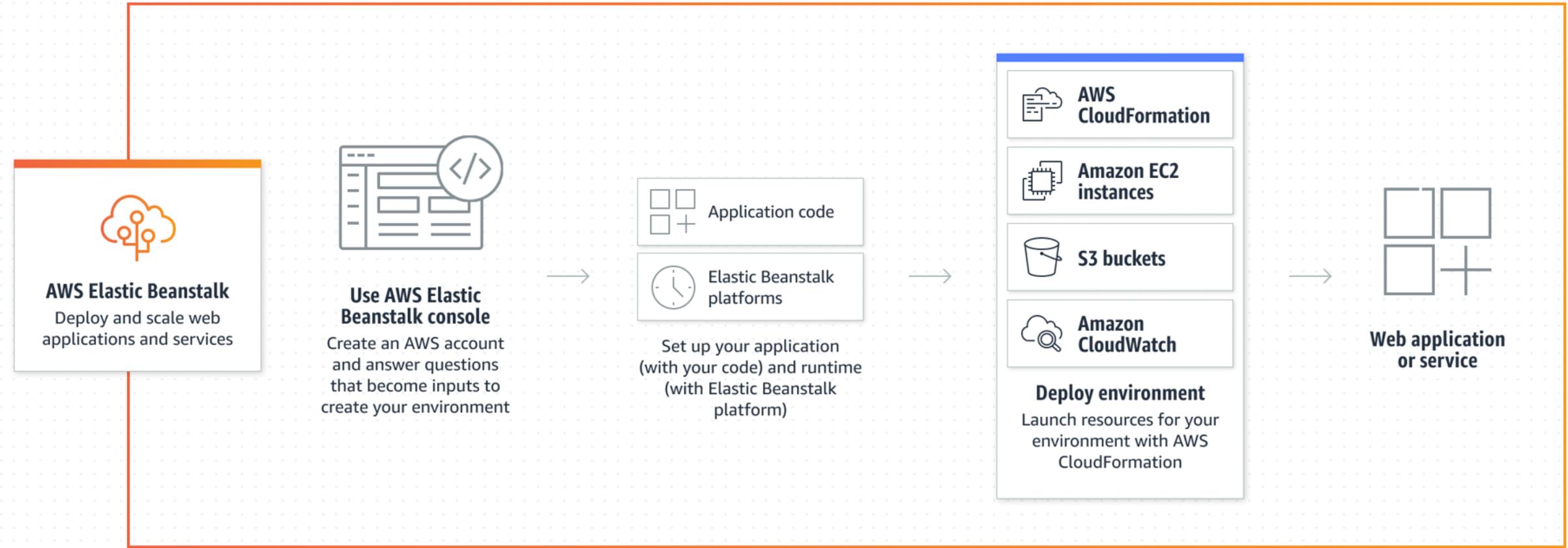
**Elastic Beanstalk** is a service for deploying and scaling web applications and services. Upload your code and Elastic Beanstalk automatically handles the deployment—from capacity provisioning, load balancing, and auto scaling to application health monitoring.

With Beanstalk, Developers can **focus on writing code instead of provisioning and managing infrastructure.**

Elastic Beanstalk supports applications developed in **Go, Java, .NET, Node.js, PHP, Python, and Ruby**. When you deploy your application, Elastic Beanstalk builds the selected supported platform version and provisions one or more AWS resources, such as Amazon EC2 instances, to run your application.

*Recommended Approach*

# AWS Elastic Beanstalk - How it works



**SOURCE:** <https://aws.amazon.com/elasticbeanstalk/>

# THANK YOU

See you next time!

eazy  
bytes

