JAVA WEB

Unit 5: Giới thiệu Spring Framework và Spring Boot



Mục lục

Giới thiệu

Spring FW là một hệ sinh thái gồm nhiều dự án giúp lập trình ứng dụng Java nhanh hơn, dễ dàng hơn và an toàn hơn cho mọi người. Spring FW tập trung vào tốc độ, sự đơn giản và năng suất.

Spring FW giúp phát triển các hệ thống liên kết mềm dẻo và có tính gắn kết cao:

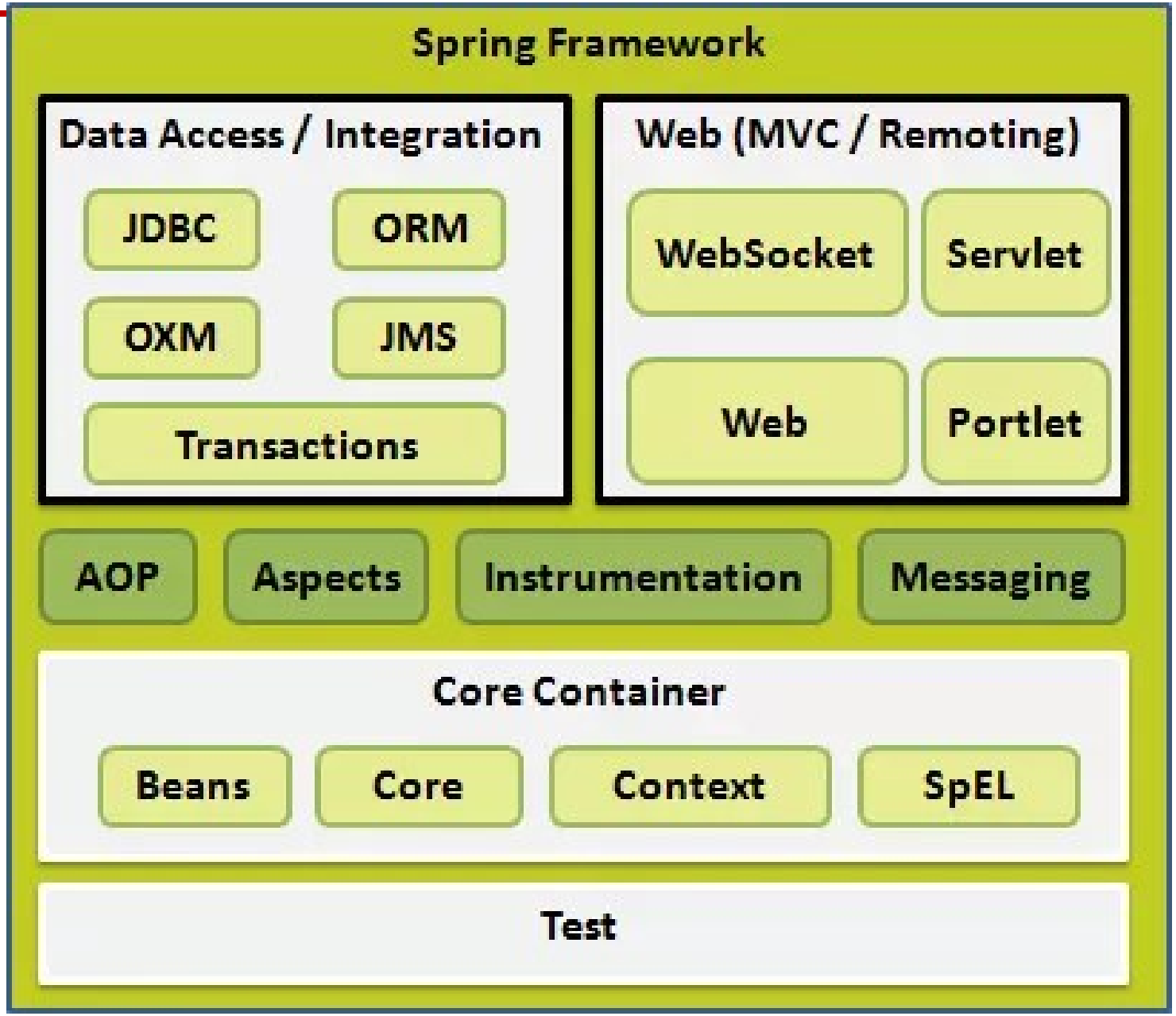
- Loose coupling (liên kết mềm dẻo) nhờ kỹ thuật đảo ngược điều khiển - Inversion of Control (IoC).
- High cohesion (kết dính cao) nhờ kỹ thuật lập trình hướng khía cạnh - Aspect Oriented programming (AOP).

Spring Boot là một framework đơn giản hóa việc phát triển các ứng dụng Java dựa trên Spring framework. **Spring Boot** cho phép bạn tạo các ứng dụng web **chạy độc lập (Standalone)**. Spring Boot sử dụng các kỹ thuật khởi động phụ thuộc và tự động cấu hình để giảm thời gian phát triển và tăng năng suất cho các nhà phát triển.

Kiến trúc Spring framework

1) Spring MVC
2) Spring Core
3) Spring REST
4) Spring Boot
5) Spring JDBCTemplates
6) Spring Batch
7) Spring Security
8) Spring Data JPA (Hibernate)
9) Spring Transaction
10) Spring AOP
11) Spring WebFlow
12) Spring Remote Services
13) Messaging in Spring

.....



Tight-coupling (TC) và loosely coupled (LC)

Tight-coupling (Khớp kết nối chặt) là một khái niệm trong Java chỉ mối quan hệ giữa các Class quá chặt chẽ. Khi yêu cầu thay đổi logic hay một class bị lỗi sẽ dẫn tới ảnh hưởng tới toàn bộ các Class khác.

loosely-coupled (khớp kết nối mềm dẻo) là cách ám chỉ việc làm giảm bớt sự phụ thuộc giữa các Class với nhau.

Khớp kết nối - Cách 1

```
public class BubbleSortAlgorithm{
    public void sort(int[] array) {
    System.out.println
    ("Đã sắp xếp bằng
    thuật toán sx nổi bọt");
    }
    }
```

```
public class VeryComplexService {
private BubbleSortAlgorithm
    bubbleSortAlgorithm = new BubbleSortAlgorithm();
public VeryComplexService(){ }
public void complexBusiness(int array[]) {
bubbleSortAlgorithm.sort(array);
 } }
```


Khớp kết nối chặt:

Với cách làm ở trên, **VeryComplexService** đã hoàn thiện được nhiệm vụ, tuy nhiên, khi có yêu cầu sử dụng thuật toán sắp xếp khác (QuickSort) thì chúng ta sẽ phải sửa lại cả 2 Class trên. Ngoài ra **VeryComplexService** sẽ chỉ tồn tại nếu BubbleSortAlgorithm tồn tại, vì VeryComplexService tạo đối tượng BubbleSortAlgorithm bên trong nó (hay nói cách khác là sự sống chết của VeryComplexService sẽ do BubbleSortAlgorithm quyết định).

Khớp kết nối- Cách 2

```
public interface SortAlgorithm {
        public void sort(int array[]); }

public class BubbleSortAlgorithm implements
SortAlgorithm { @Override
public void sort(int[] array) {
System.out.println("Đã sắp xếp bằng thuật toán sx nổi
bọt"); } }
public class QuicksortAlgorithm implements
SortAlgorithm { @Override
public void sort(int[] array) {
 System.out.println("Đã sắp xếp bằng thuật sx nhanh");
} }
//
```

```
public class VeryComplexService {
private SortAlgorithm sortAlgorithm;
public     VeryComplexService(SortAlgorithm     sortAlgorithm)     {
this.sortAlgorithm = sortAlgorithm; }
public void complexBusiness(int array[]){
sortAlgorithm.sort(array);
}
}}
//Client
class Client {
public static void main(String[] args) {
SortAlgorithm bubbleSortAlgorithm = new BubbleSortAlgorithm();
SortAlgorithm quickSortAlgorithm = new QuicksortAlgorithm();
VeryComplexService business1
= new VeryComplexService(bubbleSortAlgorithm);
VeryComplexService business2
= new VeryComplexService(quickSortAlgorithm); //Dl
}
```

Đây là cách làm phổ biến nhất. Khớp kết nối giữa 2 Class đã "lỏng" hơn trước rất nhiều. VeryComplexService sẽ không quan tâm tới việc thuật toán sắp xếp là gì nữa, mà chỉ cần tập trung vào nghiệp vụ. Còn SortAlgorithm sẽ được đưa vào từ bên ngoài tùy theo nhu cầu sử dụng.

Inversion of Control (IoC); Dependency Injection (DI)

SOLID: 5 nguyên lý của thiết kế lớp HĐT

Single responsibility principle (**Nguyên tắc trách nhiệm đơn lẻ**)

Open/closed principle (**Nguyên tắc đóng mở**)

Liskov substitution principle (**Nguyên tắc phân vùng Liskov**)

Interface segregation principle (**Nguyên tắc phân tách giao diện)**

Dependency inversion principle (**Nguyên tắc đảo ngược phụ thuộc):**

- **Inversion of Control (IoC)**
- **Dependency Injection (DI)**

Dependency Injection (DI)

Từ 2 khái niệm **tight-coupling** và **loosely-coupled,** chúng ta có thể dễ dàng hiểu khái niệm **Dependency Injection**.

Dependency Injection là một **design pattern** tuyệt vời cho phép chúng ta loại bỏ sự phụ thuộc cứng nhắc giữa các phần tử (lớp) và làm cho ứng dụng trở nên linh hoạt mềm dẻo hơn, dễ mở rộng, dễ bảo trì.

Về cơ bản, Dependency Injection phải thỏa mãn những tiêu chí sau:

✓Các phương thức của service phải được viết trong một **interface**.

✓Những class sử dụng service thì dùng interface khi khai báo.

✓Phải có nhóm các class injector phụ trách việc khởi tạo service và sau đó là các class sử dụng service ấy.

Dependency Injection (DI) - Ví dụ

```
public interface MessageService {
        public void sendMessage(String msg, String receiver);
}
```

```
public class EmailServiceImpl
                implements MessageService  {
public void sendMessage(String msg, String receiver){
   // Code logic gửi mail
   System.out.println("Email sent to "+ receiver
      +"with Message="+msg);
  }
}
```

```
public class SMSServiceImpl
                implements MessageService {
public void sendMessage(String msg, String receiver){
   // Code logic gửi sms
    System.out.println("SMS sent to "+receiver
     +"with Message="+msg);
}
}
```

Dependency Injection (DI)

```
public interface SendMessages {
    public void processMessages(String msg, String receiver);
}

public class MyDIApplication implements SendMessages {
        private MessageService service;
        public MyDIApplication() {}
       //setter dependency injection
      public void setService(MessageService service) { this.service = service; }
       public void processMessages(String msg, String receiver){
            this.service.sendMessage(msg, receiver); // Thực hiện validate msg
      }
}

//Client
public class Main   {
public SendMessages getSender() {
    MyDIApplication app = new MyDIApplication();
    app.setService(new EmailServiceImpl());
   return app; }
 }
```

Làm nhiệm vụ tiêm dependency

Inversion of Control (IoC)

Dependency Injection giúp chúng ta dễ dàng mở rộng code và giảm sự phụ thuộc giữa các dependency với nhau.

Tuy nhiên, điều này lại dẫn đến mã code phải kiêm thêm nhiệm vụ Inject dependency (tiêm sự phụ thuộc). Nếu một Class có hàng chục dependency thì sẽ dẫn tới khó khăn trong việc code, quản lý code và dependency.

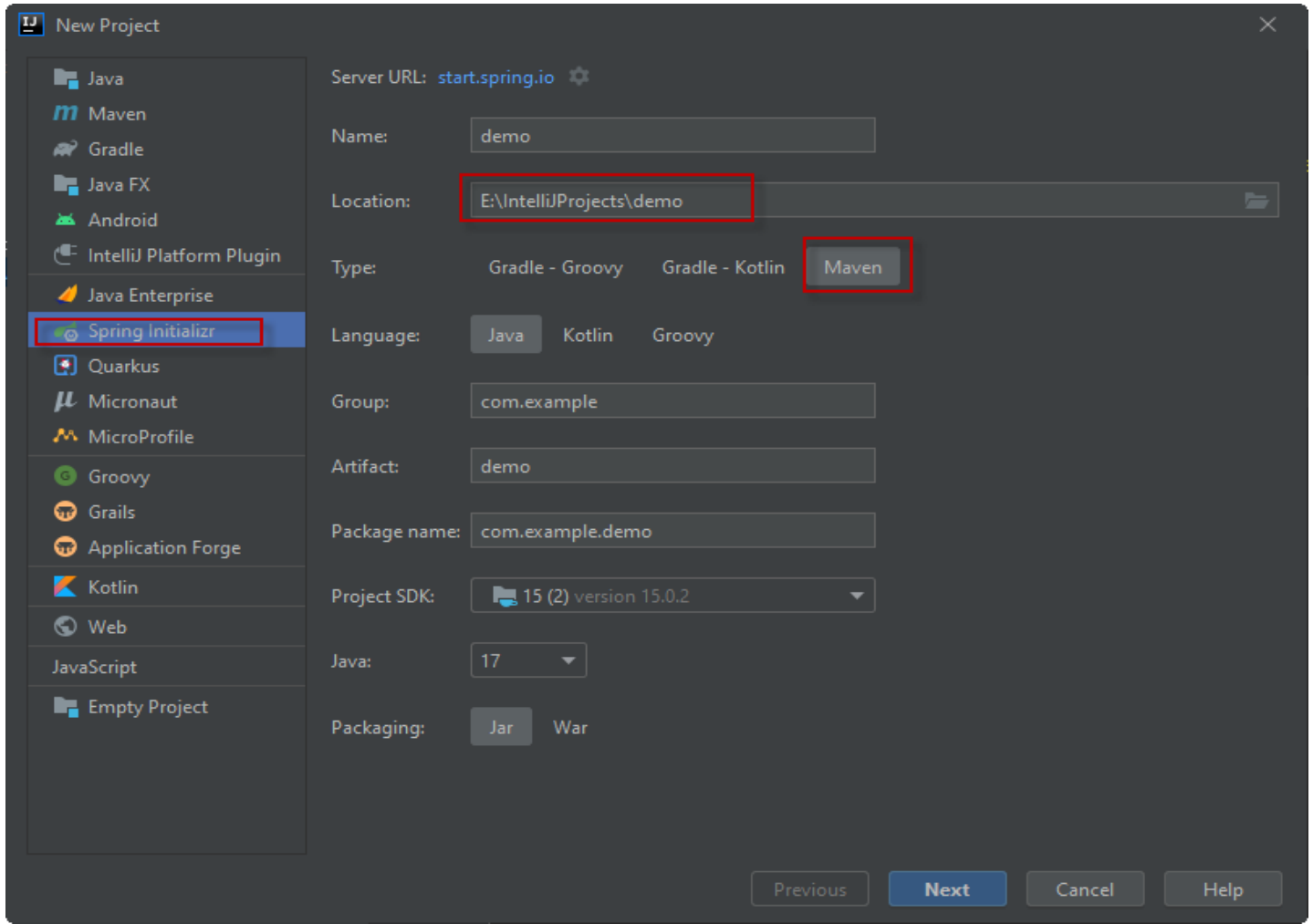
Nếu chúng ta định nghĩa trước toàn bộ các dependency có trong Project và đẩy nó vào 1 cái thùng chứa (**container**) và có một **Chương trình quản lý**. Bất kỳ các Class nào khi khởi tạo, nó cần dependency gì, thì cái **Chương trình quản lý** này sẽ tự động tìm trong kho (container) rồi inject vào thì sẽ rất tiện? **VÀ Spring Container (IoC Container) đảm nhiệm việc này: các** dependencies ở đây **chính là các Spring Bean.**

Tạo ứng dụng Spring Boot - IntelliJ

Tạo ứng dụng Spring boot từ https://start.spring.io

IDEs:

NetBeans, Eclipse, IntelliJ



Cách chạy ứng dụng Spring Boot

Trong Java truyền thống, khi chạy một project, chúng ta sẽ phải định nghĩa một hàm main() và để nó khởi chạy đầu tiên.
Trong Spring Boot, chúng ta phải chỉ cho Spring Boot biết nơi nó khởi chạy để nó cấu hình mọi thứ. Cách thực hiện là thêm annotation **@SpringBootApplication** trên class chính và gọi SpringApplication.run(App.class, args); để chạy project.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class App {
public static void main(String[] args) {
     SpringApplication.run(App.class, args);
 }

}
```

Cách chạy ứng dụng Spring Boot

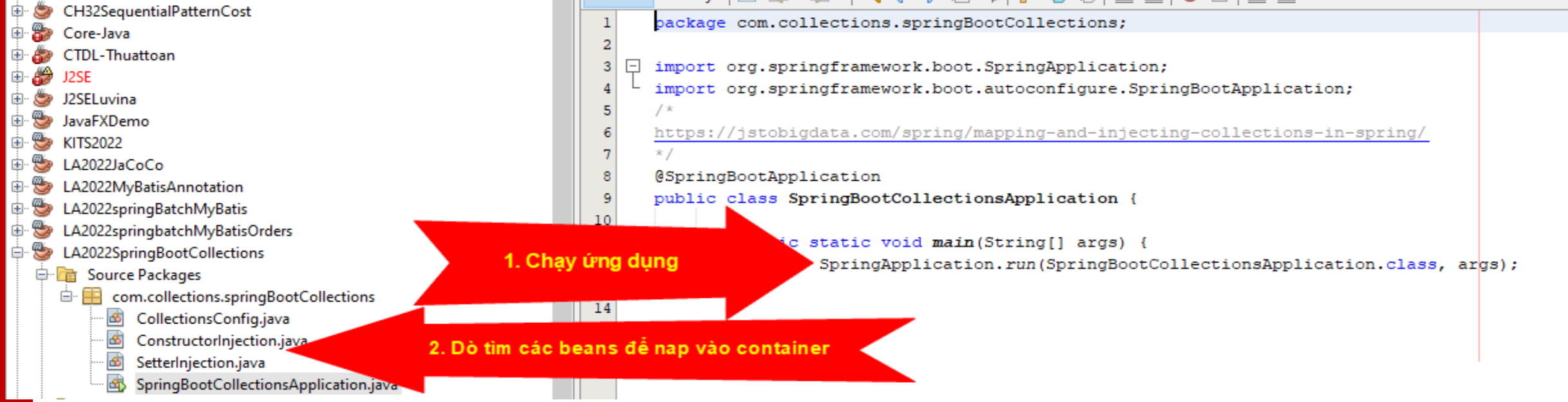
- SpringApplication.run(App.class, args) thực hiện tạo ra ***container***, Sau đó nó tìm toàn bộ các ***dependency (beans)*** trong project của bạn và đưa vào ***container***.
- Spring đặt tên cho ***container*** là ApplicationContext và đặt tên cho các ***dependency*** là Bean

```
@SpringBootApplication
public class App {
public static void main(String[] args) {
    // ApplicationContext chứa toàn bộ dependency trong project.
   ApplicationContext context = SpringApplication.run(App.class, args);
 }
}
```

@Component

Vậy Spring biết đâu là dependency? @Component là một Annotation đánh dấu trên các Class để giúp Spring nhận biết nó là một **Bean**.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.stereotype.Component;
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext = SpringApplication.run(DemoApplication.class, args);

        ComponentDemo componentDemo = (ComponentDemo) applicationContext.getBean("componentDemo");

        System.out.println(componentDemo.getValue());
    }
}
@Component
class ComponentDemo {
    public String getValue() {
        return "Hello World";
    }
```




```java
@Component
public class SetterInjection {
  private List<String> names;
  private Set<Long> phones;
  private Map<Long, String> phoneNameMap;
  @Autowired
  public void setNames(List<String> names)
    this.names = names;
  }
  @Autowired
  public void setPhones(Set<Long> phones) {
    this.phones = phones;
  }
  @Autowired
  public void setPhoneNameMap(Map<Long, String> phoneNameMap) {
    this.phoneNameMap = phoneNameMap;
  }
  @Override
  public String toString(){
      return names.toString();
  }
}
```

1. Phát hiện component

2. Tạo new SetterInjection và

Application Context

SetterInjection instance

@Component

Singleton: Mặc định các Bean được quản lý bên trong ApplicationContext đều là singleton. Trong trường hợp bạn muốn mỗi lần sử dụng là một instance hoàn toàn mới. Thì hãy đánh dấu @Component đó bằng @Scope("prototype")

```
@Component
@Scope("prototype")
```

@Autowired

Chúng ta có thể sử dụng @Autowired để đánh dấu một phụ thuộc mà Spring sẽ giải quyết và tiêm vào. @Autowired có thể sử dụng với phương thức khởi tạo, phương thức set hoặc thuộc tính của lớp.

@Autowired **trực tiếp trên thuộc tính**

@Autowired trên setter method

@Autowired constructor

https://shareprogramming.net/cac-cach-su-dung-autowired-annotation-trong-spring/#Cach_su_dung_Autowired

@Autowired trực tiếp trên thuộc tính

```
@Component
public class FieldBean {
    public String getName() {
        return "Deft Blog";
    }
}
```

```
public interface FieldExampleService {
    String getName();
}

@Service
public class FieldExampleServiceImpl implements FieldExampleService {
    @Autowired
    private FieldBean fieldBean;
    @Override
    public String getName() {
        return fieldBean.getName();
    }
}
```

@Autowired trên setter method

```
@Component
public class SetterBean {
    public int getValue() {
        return 10;
    }
}
```

```
public interface SetterExampleService {
    int getValue();
}

@Service
public class SetterExampleServiceImpl implements SetterExampleService {
    private SetterBean setterBean;

    @Autowired
    private void setBean(SetterBean setterBean) {
        this.setterBean = setterBean;
    }

    @Override
    public int getValue() {
        return setterBean.getValue();
    }
}
```

@Autowired constructor

```
@Service
public class ContructorExampleServiceImpl implements ConstructorExampleService {
    private final FieldBean fieldBean;
    private final SetterBean setterBean;

    @Autowired
    public ContructorExampleServiceImpl( FieldBean fieldBean, SetterBean setterBean) {
        this.fieldBean = fieldBean;
        this.setterBean = setterBean;
    }
    @Override
    public String print() {
        return fieldBean.getName() + " - " + setterBean.getValue();
    }
}
```

@PostConstruct

@PostConstruct được đánh dấu trên một method duy nhất bên trong Bean. ApplicationContext sẽ gọi hàm này sau khi một Bean được tạo ra và quản lý.

```
@Component
public class Person {

   @PostConstruct
    public void postConstruct(){

       System.out.println”Đối tượng Person sau khi khởi tạo xong sẽ chạy hàm này");
    }
}
```

@PreDestroy

@PreDestroy được đánh dấu trên một method duy nhất bên trong Bean. ApplicationContext sẽ gọi hàm này trước khi một Bean bị xóa hoặc không được quản lý nữa.

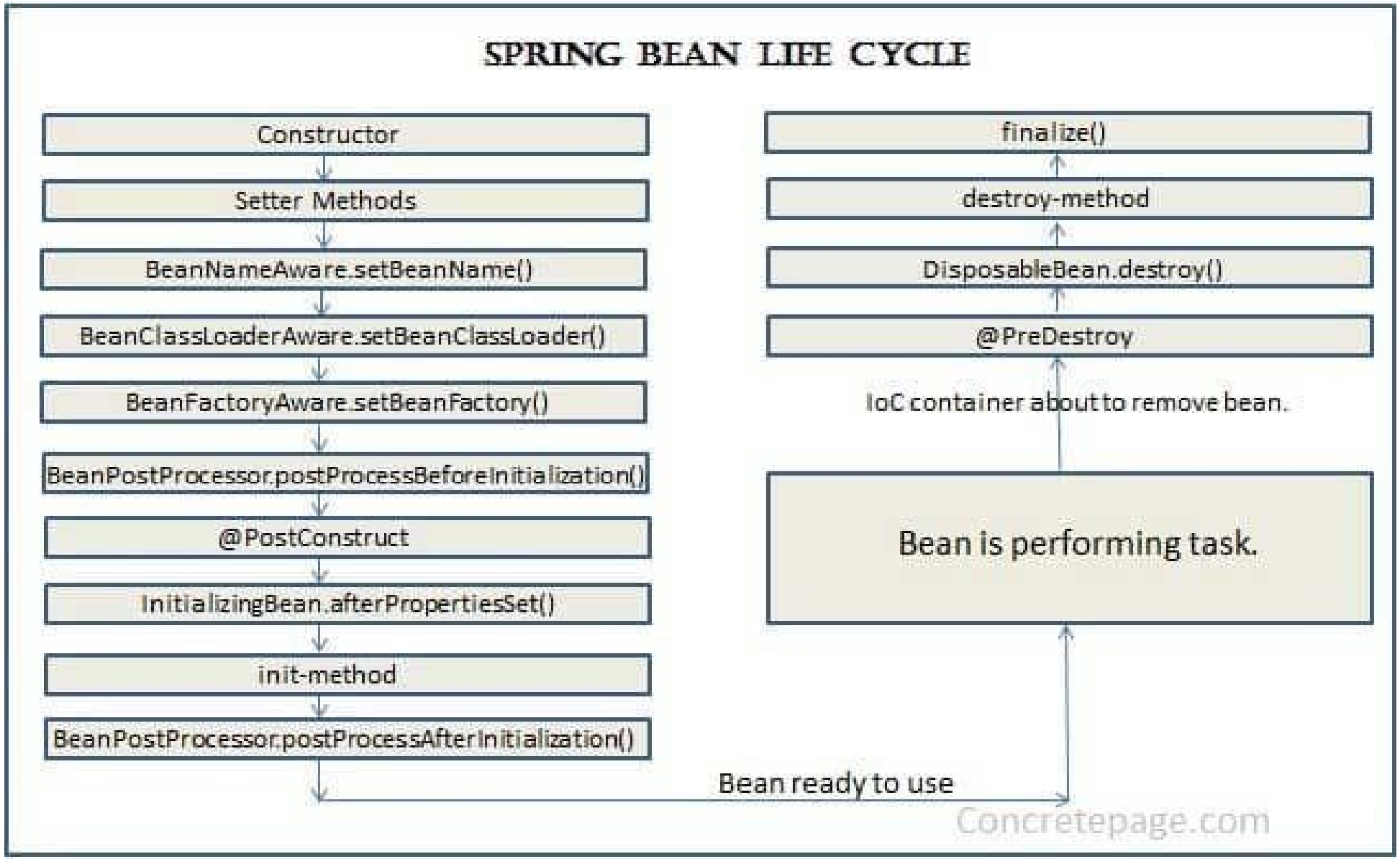
```
@Component
public class Person {
    @PreDestroy
    public void preDestroy(){
        System.out.println("Đối tượng Person trước khi bị destroy thì chạy hàm này");
    }
}
```

@PostConstruct; @PreDestroy

```
@Component
 public class Person {
@PostConstruct public void postConstruct(){
 System.out.println("\t>> Đối tượng Girl sau khi khởi tạo xong sẽ chạy hàm này");
 }
@PreDestroy public void preDestroy(){
System.out.println("\t>> Đối tượng Girl trước khi bị destroy thì chạy hàm này");
 } }
```

```
@SpringBootApplication
public class App {
public static void main(String[] args) {
// ApplicationContext chính là container, chứa toàn bộ các Bean
  System.out.println("> Trước khi IoC Container được khởi tạo");
  ApplicationContext context = SpringApplication.run(App.class, args);
  System.out.println(">Sau khi IoC  Container được khởi tạo");
 // Khi chạy xong, lúc này context sẽ chứa các Bean có đánh  // dấu @Component.
 Person per = context.getBean(Person.class);
  System.out.println("> Trước khi IoC Container destroy Person ");
 ((ConfigurableApplicationContext) context)
                                    .getBeanFactory().destroyBean(per);
  System.out.println("> Sau khi IoC Container destroy Person ");
} }
```

Vòng đời của Bean



@Configuration và @Bean

- **@Configuration** đánh dấu trên một class cho phép Spring Boot biết được đây là nơi định nghĩa ra các Bean.
- **@Bean** đánh dấu trên các method cho phép Spring Boot biết được đây là Bean và sẽ thực hiện đưa Bean này vào container.

=> @Bean sẽ nằm trong các class có đánh dấu @Configuration.

@Configuration và @Bean

```
/** * Một class cơ bản, không sử dụng `@Component` */
 public class SimpleBean { private String username;
  public SimpleBean(String username) { setUsername(username);
  }
 @Override
  public String toString() {
     return "This is a simple bean, name: " + username; }
  public String getUsername() { return username; }
  public void setUsername(String username) {
       this.username = username;
 }
}
```

```
@Configuration
 public class AppConfig {
@Bean
SimpleBean simpleBeanConfigure(){
// Khởi tạo một instance của SimpleBean và trả ra ngoài
return new SimpleBean("loda");
}

}
```


```
@SpringBootApplication
public class App { public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(App.class, args);
     // Lấy ra bean SimpleBean trong Context
    SimpleBean simpleBean = context.getBean(SimpleBean.class);
    // In ra màn hình
    System.out.println("Simple Bean Example: " + simpleBean.toString());
 }
}
```

@Configuration và @Bean

Spring Boot lần đầu khởi chạy, ngoài việc đi tìm các @Component thì nó còn làm một nhiệm vụ nữa là tìm các class @Configuration, cụ thể:

- Đi tìm class có đánh dấu @Configuration; Tạo ra đối tượng từ class có đánh dấu @Configuration; Tìm các method có đánh dấu @Bean trong đối tượng vừa tạo; Thực hiện gọi các method có đánh dấu @Bean để lấy ra các Bean và đưa vào Context.

Như vậy, về bản chất, @Configuration cũng là @Component. Nó chỉ khác ở ý nghĩa sử dụng. (Giống với việc class được đánh dấu **@Service** chỉ nên phục vụ logic vậy).

Component Scan

Trong trường hợp bạn muốn tuỳ chỉnh cấu hình cho **Spring Boot** chỉ tìm kiếm các bean trong một package nhất định thì có các cách sau đây:

- Sử dụng @ComponentScan; Sử dụng scanBasePackages trong @SpringBootApplication.

Cách 1: @ComponentScan

```
@ComponentScan("com.spring.componentscan.others")
@SpringBootApplication
public class App { public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(App.class, args);
    try {  Girl girl = context.getBean(Girl.class);
           System.out.println("Bean: " + girl.toString()); }
    catch (Exception e) {
           System.out.println("Bean Girl không tồn tại"); }
          try { OtherGirl otherGirl = context.getBean(OtherGirl.class);
               if (otherGirl != null) { System.out.println("Bean: " + otherGirl.toString());
              }
            }
           catch (Exception e) { System.out.println("Bean Girl không tồn tại"); }
     }
}
```

Component Scan

Trong trường hợp bạn muốn tuỳ chỉnh cấu hình cho **Spring Boot** chỉ tìm kiếm các bean trong một package nhất định thì có các cách sau đây:

- Sử dụng @ComponentScan
- Sử dụng scanBasePackages trong @SpringBootApplication

Cách 2: scanBasePackages

```
@SpringBootApplication( scanBasePackages = "me.loda.spring.componentscan.others")
public class App {
   public static void main(String[] args) {
      ApplicationContext context = SpringApplication.run(App.class, args);
      try { Girl girl = context.getBean(Girl.class);
           System.out.println("Bean: " + girl.toString());
           }
      catch (Exception e) { System.out.println("Bean Girl không tồn tại"); }
        try { OtherGirl otherGirl = context.getBean(OtherGirl.class);
            if (otherGirl != null) { System.out.println("Bean: " + otherGirl.toString()); }
         }
          catch (Exception e) { System.out.println("Bean Girl không tồn tại"); }
      }
}
```

Multiple package scan

```
@ComponentScan
                ( {"spring.componentscan.others2","spring.componentscan.others"})

@SpringBootApplication(scanBasePackages =
                {"spring.componentscan.others", "spring.componentscan.others2"})
```

@Primary Annotation

Sử dụng @Primary khi ta có nhiều beans cùng kiểu dữ liệu và ta muốn ưu tiên một kiểu nào đó. Trong ví dụ dưới đây ta viết 1 chương trình gửi thư. Có 2 dịch vụ gửi thư là **Facebook** và **Email**.

Cả 2 services FacebookMessageService và EmailMessageService cùng cài đặt chung 1 interface MessageService. Như vậy ta có 2 bean FacebookMessageService và EmailMessageService có cùng kiểu dữ liệu là MessageService.

Vậy cái nào sẽ được ưu tiên đầu tiên?

@Primary Annotation

@Primary Annotation chỉ ra bean được ưu tiên

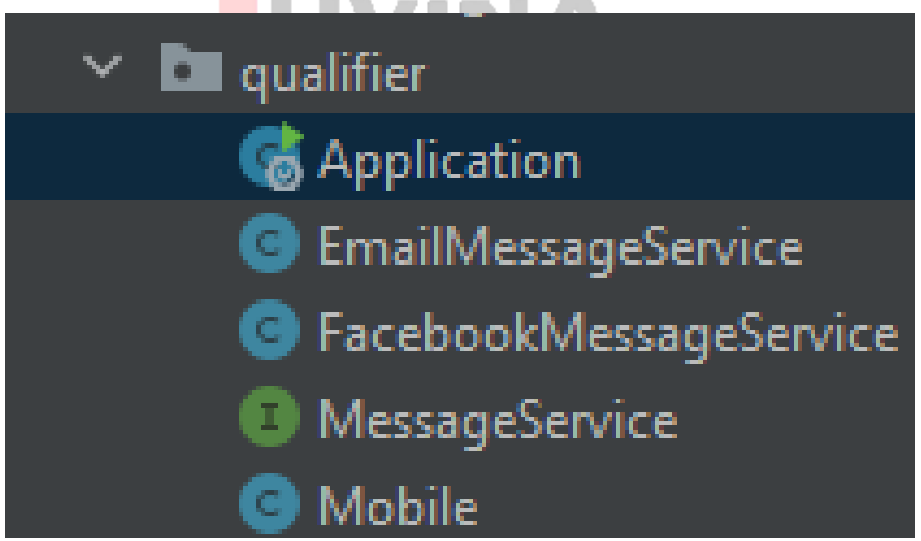

```java
package mta.otherAnnotations.primary;

public interface MessageService {

    void sendMsg();
}
```

```java
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        ApplicationContext cx= SpringApplication.run(Application.class, args);

        MessageService messageService
                = cx.getBean(MessageService.class);
        messageService.sendMsg();// Facebook

    }
}
```

```java
//EmailMessageService.java
package mta.otherAnnotations.primary;
import org.springframework.stereotype.Component;

@Component
public class EmailMessageService implements MessageService {
    @Override
    public void sendMsg() {
        System.out.println("Send message by email");
    }
}
//FacebookMessageService.java
package mta.otherAnnotations.primary;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Component;

@Primary
@Component
public class FacebookMessageService implements MessageService {

    @Override
    public void sendMsg() {
        System.out.println("Send message by Facebook");
    }
}
```

@Qualifier

Chúng ta sử dụng @Qualifier để xác định chính xác loại dữ liệu nào được đưa vào. Ví dụ chúng ta có EmailMessageService và FacebookMessageService có cùng 1 kiểu dữ liệu là MessageService. Vì chúng cùng kiểu nên khi nhúng bean vào thì Spring IoC không biết nên nhúng cái nào là đúng. Nên để giúp Spring IoC nhúng đúng bean thì ta cần dùng **@Qualifier** để chỉ ra bean nhúng vào chính là EmailMessageService hay FacebookMessageService



@Qualifier

```java
package mta.otherAnnotations.qualifier;
public interface MessageService {
    void sendMsg();
}
```

```java
package mta.otherAnnotations.qualifier;

import org.springframework.stereotype.Component;
@Component
public class EmailMessageService implements MessageService {
    @Override
    public void sendMsg() {
        System.out.println("Send message by email");
    }
}
```

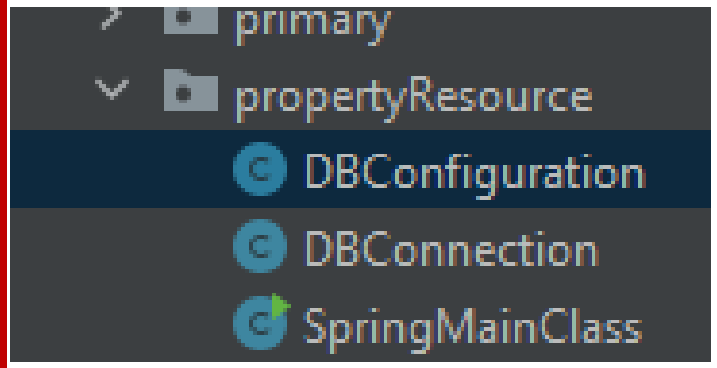
```java
package mta.otherAnnotations.qualifier;

import org.springframework.stereotype.Component;

@Component
public class FacebookMessageService implements MessageService {

    @Override
    public void sendMsg() {
        System.out.println("Send message by Facebook");
    }
```

```java
package mta.otherAnnotations.qualifier;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;
@Component
public class Mobile {

    @Autowired
    @Qualifier("emailMessageService")
    private MessageService messageService;

    public MessageService getMessageService() {
        return messageService;
    }

    public void setMessageService(MessageService messageService) {

        this.messageService = messageService;
    }
}

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        ApplicationContext cx= SpringApplication.run(Application.class, args);

        Mobile mobile = cx.getBean(Mobile.class);
        mobile.getMessageService().sendMsg(); //Send message by email
    }
}
```

@PropertySource Annotation

Sử dụng @PropertySource để đọc các giá trị từ file cấu hình và gán vào các thuộc tính trong Bean.

Lấy giá trị từ file cấu hình ra thông qua Annotation @Value hoặc Enviroment.




```
DB_DRIVER_CLASS=com.mysql.cj.jdbc.Driver
DB_URL=jdbc:mysql://localhost:3306/Test
DB_USERNAME=root
DB_PASSWORD=root
APP_NAME=PropertySource Example1000
```

```java
@Configuration
@PropertySource("classpath:dbTestProperResource.properties")
@PropertySource(value = "classpath:rootTestProperResource.properties"
, ignoreResourceNotFound = true)
public class DBConfiguration {
    @Autowired
    Environment env;
    // Tu dong chay phuong thuc ngay khi khoi dong chuong
    @Bean
    public DBConnection getDBConnection() {
        System.out.println("Getting DBConnection Bean for App: " + env.getProperty("APP_NAME"));

        DBConnection dbConnection = new DBConnection();
        return dbConnection;
    }
}
```

@PropertySource Annotation

```
// dbTestProperResource.properties
DB_DRIVER_CLASS=com.mysql.jdbc.Driver
DB_URL=jdbc:mysql://localhost:3306/Test
DB_USERNAME=root
DB_PASSWORD=root
APP_NAME=PropertySource Example1000
// rootTestProperResource.properties
APP_NAME1=PropertySource Example
//
@Configuration
@PropertySource("classpath:dbTestProperResource.properties")
@PropertySource(value = "classpath:rootTestProperResource.properties"
, ignoreResourceNotFound = true)
public class DBConfiguration {
    @Autowired
    Environment env;
    // Tu dong chay phuong thuc ngay khi khoi dong chuong
    @Bean
    public DBConnection getDBConnection() {
        DBConnection dbConnection = new DBConnection();
        return dbConnection;
    }
}
```

```
public class DBConnection {
    @Value("${APP_NAME}")          private String appName;
    @Value("${APP_NAME1}")         private String appName1;
    @Value("${DB_DRIVER_CLASS}")   private String driverClass;
    @Value("${DB_URL}")            private String dbURL;
    @Value("${DB_USERNAME}")       private String userName;
    @Value("${DB_PASSWORD}")       private char[] password;
    private Connection con;
    public Connection getConnection() {
        if (this.con != null)  return con;
        Connection con = null;
        try {
            Class.forName(driverClass);
            con = DriverManager.getConnection(dbURL, userName, String.valueOf(password));
        } catch (ClassNotFoundException | SQLException e) {
        }
        this.con = con;
        return con;
    }
}
public class SpringMainClass {
    public static void main(String[] args) throws SQLException {
        AnnotationConfigApplicationContext context
        = new AnnotationConfigApplicationContext();//(DBConnection.class);
        context.scan("mta.otherAnnotations.propertyResource");
        context.refresh();
        DBConnection dbConnection = context.getBean(DBConnection.class);
        System.out.println(dbConnection.getAppName() +":"+ dbConnection.getAppName1());
        Connection con = dbConnection.getConnection();
        context.close();
    }
}
```

1. Constructor Injection for Collections

```
@Component
public class ConstructorInjection {
  private List<String> names;
  private Set<Long> phones;
  private Map<Long, String> phoneNameMap;

@Autowired
public ConstructorInjection(  List<String>
names,   Set<Long> phones,
Map<Long, String> phoneNameMap )  {
   this.names = names;
   this.phones = phones;
   this.phoneNameMap = phoneNameMap;
} //Getter, Setter, toString() omitted for bravity
}
```

```
@Configuration
@ComponentScan("basic.ioc.bean.collections")
public class CollectionsConfig {

@Bean
public List<String> namesList() {
List names = new ArrayList<String>();
names.add("Salman Khan");
names.add("Hrithik Roshan");
return names;
}
@Bean
public Set<Long> numbersBean() {
Set<Long> numbers = new HashSet<>();
numbers.add(2222222222L);
numbers.add(1212121212L);
return numbers;
}
@Bean
public Map<Long, String> phoneNameMapBean() {
Map<Long, String> phoneNames = new HashMap<>();
phoneNames.put(888888888888L, "Ram");
phoneNames.put(777777777777L, "Bhim");
return phoneNames;
}
}
```

2. Setter Injection for Collections


```java
@Component
public class SetterInjection {
private List<String> names; private Set<Long>
phones;
private Map<Long, String> phoneNameMap;
@Autowired
public void setNames(List<String> names) {
this.names = names;
}
@Autowired
public void setPhones(Set<Long> phones) {
this.phones = phones;
}
@Autowired
public void setPhoneNameMap(Map<Long, String>
phoneNameMap) {
this.phoneNameMap = phoneNameMap;
} //toString() omitted for bravity.
}
```

```java
@Configuration
@ComponentScan("basic.ioc.bean.collections")
public class CollectionsConfig {
@Bean
public List<String> namesList() {
List names = new ArrayList<String>();
names.add("Salman Khan");
names.add("Hrithik Roshan");
return names;
}
@Bean
public Set<Long> numbersBean() {
Set<Long> numbers = new HashSet<>();
numbers.add(2222222222L);
numbers.add(1212121212L);
return numbers;
}
@Bean
public Map<Long, String> phoneNameMapBean() {
Map<Long, String> phoneNames = new HashMap<>();
phoneNames.put(888888888888L, "Ram");
phoneNames.put(777777777777L, "Bhim");
return phoneNames;
}
}
```

3. Field Injection for Collections

```java
@Component
public class FieldInjection {
@Autowired //Avoid using property injection
private List<String> names;

@Autowired //Avoid using property injection
private Set<Long> phones;

@Autowired //Avoid using property injection
private Map<Long, String> phoneNameMap;
}
```

```java
@Configuration
@ComponentScan("basic.ioc.bean.collections")
public class CollectionsConfig {
@Bean
public List<String> namesList() {
List names = new ArrayList<String>();
names.add("Salman Khan");
names.add("Hrithik Roshan");
return names;
}
@Bean
public Set<Long> numbersBean() {
Set<Long> numbers = new HashSet<>();
numbers.add(2222222222L);
numbers.add(1212121212L);
return numbers;
}
@Bean
public Map<Long, String> phoneNameMapBean() {
Map<Long, String> phoneNames = new HashMap<>();
phoneNames.put(888888888888L, "Ram");
phoneNames.put(777777777777L, "Bhim");
return phoneNames;
}
}
```