TRƯỜNG ĐẠI HỌC
SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
HCMC University of Technology and Education
HCMUTE

KHOA CÔNG NGHỆ THÔNG TIN
BỘ MÔN CÔNG NGHỆ PHẦN MỀM

LẬP TRÌNH WEB (WEBPR330479)

Spring IOC Container

THS. NGUYỄN HỮU TRUNG

PR33
WEB
0479

Thông tin Giảng viên

- Ths. Nguyễn Hữu Trung
- Khoa Công Nghệ Thông Tin
- Trường Đại học Sư Phạm Kỹ Thuật TP.HCM
- 090.861.7108
- trungnh@hcmute.edu.vn
- https://www.youtube.com/@baigiai

Nội dung

- Spring IOC Container
- Khởi tạo Spring Container
- Lấy beans từ Spring Container bằng ApplicationContext
- Lấy beans từ Spring Container bằng BeanFactory
- Cấu hình IOC bằng Java
- Phạm vi hoạt động của Bean
- Các Annotation trong Spring

Spring IOC Container

- Spring IOC Container là trái tim của Spring. Nó có nhiệm vụ quản lý vòng đời của bean (các đối tượng trong dự án spring), khởi tạo, cấu hình, và tương tác giữa các bean trong ứng dụng Spring. Mình có thể cấu hình Spring IOC bằng XML, Java code hoặc Java annotation.
- Spring framework hỗ trợ 2 loại container là BeanFactory container và ApplicationContext container. Giúp chúng ta có thể khởi tạo và quản lý các beans (đối tượng) trong Spring.
- BeanFactory là interface trên cùng của Spring IOC container, còn ApplicationContext là lớp con của BeanFactory. Sự khác nhau chính của BeanFactory và ApplicationContext là
  - BeanFactory: Các bean được tạo ra khi chúng ta gọi phương thức **getBean()**
  - ApplicationContext: chúng ta không cần phải chờ phương thức getBean được gọi mới tạo Bean. Mà khi container được start (khởi động) thì bean cũng đã được tạo ra do vậy không phải chờ gọi phương thức getBean.

Khởi tạo Spring Container

- Có 3 cách khởi tạo Spring Container như sau :
  - **AnnotationConfigApplicationContext**: Sử dụng khi chúng ta viết 1 chương trình Java độc lập. Không phải là Java web mà chỉ là ứng dụng java thông thường.
  - **ClassPathXmlApplicationContext**: Nếu như ta sử dụng XML để cấu hình cho Spring thì ta dùng ClassPathXmlApplicationContext để nạp các cấu hình đó thông qua file XML.
  - **FileSystemXmlApplicationContext**: Cũng tương tự như ClassPathXmlApplicationContext nhưng file cấu hình chúng ta không phải là XML và chúng ta chỉ định đường dẫn để nạp file đó là ở đâu.
- Ví dụ như ta sử dụng ApplicationContext để tạo Spring Container cho ứng dụng Java độc lập như sau. Giả sử ta đã có file applicationContext.xml

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
```

Lấy beans từ Spring Container bằng ApplicationContext

- Spring Container là nơi chứa đựng tất cả các beans (đối tượng) của ứng dụng trong 1 chương trình. Mỗi bean sẽ có 1 cái tên riêng. Dựa vào cái tên đó ta có thể lấy được đối tượng tương ứng với tên.
- Bước 1: ta khởi tạo Spring Container bằng cách đọc file applicationContext.xml **ClassPathXmlApplicationContext("applicationContext.xml").** Như vậy trong file applicationContext.xml sẽ có tất cả các bean trong đó giả sử ta có 1 bean tên là **helloWorld**. Ta sẽ lấy ra ở bước 2
- Bước 2: ta sử dụng phương thức **context.getBean("bean name")** lúc này container sẽ trả lại cho mình 1 đối tượng như mình muốn **context.getBean("helloWorld");**

```
ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
```

Lấy beans từ Spring Container bằng BeanFactory


```
XmlBeanFactory factory = new XmlBeanFactory (new
ClassPathResource("beans.xml"));

HelloWorld obj = (HelloWorld) factory.getBean("helloWorld");
```

Cấu hình IOC bằng Java

- Bước 1: Tạo Maven project
- Bước 2: Cấu hình thư viện file pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.0.RELEASE</version>
</dependency>
```

- Bước 3: Cấu hình HelloWorld Spring Bean bằng Java

```
public class HelloWorld {
   private String message;
   public void setMessage(String message) {
      this.message = message;
   }
   public void getMessage() {
      System.out.println("My Message : " + message);
   }
}
```

Cấu hình IOC bằng Java

- Bước 4: Cấu hình Metadata Java

Sử dụng @Congiguration và @Bean. Khi container load lên nó sẽ nhận biết các annotation @ này để tạo bean (đối tượng).

```
@Configuration
public class AppConfig {
      @Bean
     public HelloWorld helloWorld() {
          HelloWorld helloWorld = new HelloWorld();
          helloWorld.setMessage("Hello World!");
          return helloWorld;
     }
}
```

Cấu hình IOC bằng Java

- Bước 5: Tạo Spring Container

```
public class Application {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext  context = new
AnnotationConfigApplicationContext(AppConfig.class);
        context.close();
    }
}
```

- Bước 6: Lấy đối tượng bean HelloWorld và gọi phương thức

```
public class Application {
  public static void main(String[] args) {
AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
    HelloWorld obj = (HelloWorld) context.getBean("helloWorld");
    obj.getMessage();
    context.close();
  }
}
```

Kết quả ta nhận được là text : Hello World

Phạm vi hoạt động của Bean

- Singleton Scope
  - Khi một Bean được khai báo là Singleton thì Bean đó là duy nhất trong Spring IoC và được share cho tất cả các Beans khác nếu cần sử dụng nó. Như vậy ta chỉ cần tạo một Bean duy nhất và sử dụng cho toàn hệ thống. Ví dụ mình có 1 Bean về connect database thì mình chỉ tạo một lần duy nhất. Các Bean khác muốn dùng thì nhúng vào chứ không phải mình có 10 Beans khác nhau dùng Bean Connect Database thì mình tạo 10 Bean Database trong Spring IoC.
  - Scope mặc định khi một Bean được tạo ra là Singleton.
  - Định nghĩa Scope Singletion bằng XML.

```
<bean id="accountService"
class="vn.iotstar.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton
scope is the default) -->

<bean id="accountService"
class="vn.iotstar.DefaultAccountService" scope="singleton"/>
```

Phạm vi hoạt động của Bean

- **Singleton Scope:**
  - Định nghĩa Scope Singletion bằng Java

```
@Configuration
public class AppConfiguration {
 @Bean
 @Scope("singleton") // default scope
 public UserService userService(){
  return new UserService();
 }
}
```

Phạm vi hoạt động của Bean

- **Prototype Scope**:
  - Khác với Singleton Scope, Bean (đối tượng) sẽ được tạo ra mới mỗi khi có một yêu cầu tạo Bean. Như vậy mỗi lần gọi tới Bean mà có Scope là Prototype thì nó sẽ tạo ra một đối tượng (Bean) trong Spring IoC container.
  - Định nghĩa Scope Prototype bằng XML

```
<bean id="accountService"
class="vn.iotstar.DefaultAccountService" scope="prototype"/>
```

  - Định nghĩa Scope Prototype bằng JAVA

```
@Configuration
public class AppConfiguration {
 @Bean
 @Scope("prototype")
 public UserService userService(){
  return new UserService();
 }
}
```

Phạm vi hoạt động của Bean

- Request, Session, Application và WebSocket Scope
  - Những Scope như Request, Session, Application và Websocket thì chỉ có tồn tại ở những ứng dụng là Web Application. Nếu ta sử dụng ở những ứng dụng Spring độc lập thì sẽ nhận được thông báo lỗi IllegalExection unknow bean scope vì ứng dụng này không phải là ứng dụng web nên không có Scope nêu trên.
  - Để sử dụng được các Scope trên thì chúng ta phải Configure (cấu hình) dự án web thêm một vài thông số như thêm SpringDispatchServlet vào file cấu hình trong file web.xml trước khi sử dụng các Scope trên.

```
<web-app>
   ...
   <filter>
      <filter-name>requestContextFilter</filter-name>
      <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>requestContextFilter</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
   ...
</web-app>
```

Phạm vi hoạt động của Bean

- Request Scope
  - Cấu hình Request Scope cho XML.

```
<bean id="loginAction" class="vn.iotstar.LoginAction"
scope="request"/>
```

  - Cấu hình Request Scope cho Java.

```
@RequestScope
@Component
public class LoginAction {
    // ...
}
```

Spring Container sẽ tạo bean (đối tượng) LoginAction mới khi có một request (yêu cầu) từ người dùng. Sau khi Request (yêu cầu) xử lý xong thì Bean sẽ bị xoá đi

Phạm vi hoạt động của Bean

- Session Scope:
  - Scope Session sẽ tồn tại chừng nào Sesion ở HTTP bị xoá hoặc hết hiệu lực.
  - Cấu hình Session Scope cho XML.

```
<bean id="loginAction" class=“vn.iotstar.LoginAction"
scope=“session"/>
```

  - Cấu hình Session Scope cho Java.

```
@SessionScope
@Component
public class UserInfo {
    // ...
}
```

Phạm vi hoạt động của Bean

- Application Scope:
  - Application Scope được tạo một lần cho toàn bộ ứng dụng Web Application. Application Scope được chứa đựng như một ServletContext, nó cũng gần tương tự như Singleton Scope nhưng nó là Singleton cho từng ServeletContext.
  - Cấu hình Application Scope cho XML.

```
<bean id="appiotstar" class="vn.iotstar.LoginAction"
scope="application "/>
```

  - Cấu hình Application Scope cho Java.

```
@ApplicationScope
@Component
public class UserInfo {
    // ...
}
```

Annotation @Bean trong Spring

- Tạo bean bằng Java

```
@Configuration
public class Application {
 @Bean
 public CustomerService customerService() {
  return new CustomerService();
 }
 @Bean
 public OrderService orderService() {
  return new OrderService();
 }
}
```

- Tạo Bean bằng XML

```
<beans>
   <bean id="customerService" class="vn.iotstar.projectname.CustomerService"/>
   <bean id="orderService" class="vn.iotstar.projectname.OrderService"/>
</beans>
```

Annotation @Bean trong Spring

- Vòng đời @Bean

```
public class Foo {
    public void init() {
        // initialization logic via xml config
    } }
public class Bar {
    public void cleanup() {
        // destruction logic via xml config
    }
}
@Configuration
public class AppConfig {
    @Bean(initMethod = "init")
    public Foo foo() {
        return new Foo();
    }
    @Bean(destroyMethod = "cleanup")
    public Bar bar() {
        return new Bar();
    }
}
```

Annotation @Bean trong Spring

- Thay đổi tên của bean

```
@Configuration
public class Application {

 @Bean(name = "cService")
 public CustomerService customerService() {
  return new CustomerService();
 }

 @Bean(name = "oService")
 public OrderService orderService() {
  return new OrderService();
 }
}
```

Annotation @Configuration trong Spring

- Spring @Configuration là một thành phần trong Spring Core Framework. @Configuration Annotation chỉ ra rằng trong Class đó có @Bean. Vì vậy khi Spring IoC quét qua các Class mà có Annotation là @Configuration nó sẽ hiểu trong Class đó có khai báo một số bean và vào đó tạo các bean.

- Thường những cấu hình file cấu hình dự án mình sẽ dùng Annotation @Configuration để đánh dấu cho Spring IoC biết.

```
@Configuration
public class Application {

    @Bean
    public CustomerService customerService() {
        return new CustomerService();
    }

    @Bean
    public OrderService orderService() {
        return new OrderService();
    }
}
```

Annotation @PropertySource trong Spring

- Annotation @PropertySource dùng để đọc các giá trị từ file và gán vào các thuộc tính trong Bean. Ví dụ như trong dự án chúng ta thường có 1 file cấu hình database trong file này chúng ta điền tham số của database như username và password. Sau đó trong Class Java bean chúng ta chỉ cần lấy giá trị lưu đó từ file ra thông qua Annotation @Value hoặc @Eviroment.
- Trong thực tế chúng ta không điền thẳng giá trị user name và password trực tiếp trong code mà phải tạo 1 file riêng chứa các thông tin này. Sau đó dùng @PropertySource để load thông tin ra.
- Giả sử ta có 1 file tên config.properties

```
jdbc.driver = com.mysql.driver
jdbc.url    = http://localhost:3306/btap01
jdbc.user   = trung
jdbc.password = trung
```

Annotation @PropertySource trong Spring

- Chúng ta sẽ dùng @PropertySource để load file config.properties. Sau đó sử dụng lấy các giá trị.
- Chúng ta khai báo 1 đối tượng Environment. Nếu muốn lấy giá trị jdbc driver nào ta chỉ cần dùng phương thức env.getProperty("jdbc.driver")

```
@Configuration
@PropertySource("classpath:config.properties")
public class ProperySourceDemo implements InitializingBean {
 @Autowired
 Environment env;
 @Override
 public void afterPropertiesSet() throws Exception {
  setDatabaseConfig();
 }
 private void setDatabaseConfig() {
  DataSourceConfig config = new DataSourceConfig();
  config.setDriver(env.getProperty("jdbc.driver"));
  config.setUrl(env.getProperty("jdbc.url"));
  config.setUsername(env.getProperty("jdbc.username"));
  config.setPassword(env.getProperty("jdbc.password"));
  System.out.println(config.toString());
 }
}
```

Annotation @PropertySource trong Spring

- **Load nhiều file configure**: Chúng ta có thể load nhiều file properties cùng một lúc nếu dự án có nhiều file properties

```
@Configuration
 @PropertySources({
  @PropertySource("classpath:config.properties"),
  @PropertySource("classpath:db.properties")
 })
 public class AppConfig {
  //...
 }
```

Annotation @PropertySource trong Spring

- Sử dụng @Value: Ngoài cách sử dụng đối tượng Environment, ta có thể sử dụng @Value để lấy giá trị trong file properties.

```
@PropertySource("classpath:config.properties")
public class ProperySourceDemo implements InitializingBean {
 private static final Logger LOGGER = LoggerFactory.getLogger(ProperySourceDemo.class);
 @Value("${jdbc.driver}")
 private String driver;
 @Value("${jdbc.url}")
 private String url;
 @Value("${jdbc.username}")
 private String username;
 @Value("${jdbc.password}")
 private String password;
 @Autowired
 Environment env;
 @Override
 public void afterPropertiesSet() throws Exception {
  LOGGER.info(driver);
  LOGGER.info(url);
  LOGGER.info(password);
  LOGGER.info(username);
  setDatabaseConfig();
 }
 private void setDatabaseConfig() {
  DataSourceConfig config = new DataSourceConfig();
  config.setDriver(env.getProperty("jdbc.driver"));
  config.setUrl(env.getProperty("jdbc.url"));
  config.setUsername(env.getProperty("jdbc.username"));
  config.setPassword(env.getProperty("jdbc.password"));
  System.out.println(config.toString());
 }}
```

Annotation @Autowire

- Sử dụng để nhúng các Bean vào Bean cần dùng. Có thể nhúng qua Constructor, Setter và Biến

```
@RestController
public class CustomerController {
   private CustomerService customerService;
   @Autowired
   public CustomerController(CustomerService customerService) {
      this.customerService = customerService;
   }
}
```

```
@RestController
public class CustomerController {
   private CustomerService customerService;
   @Autowired
   public void setCustomerService(CustomerService customerService) {
      this.customerService = customerService;
   }
}
```

```
@Autowired
   private CustomerService customerService;
```

Annotation @Qualifier

- Khi có nhiều Bean có cùng kiểu dữ liệu thì @Qualifier giúp chúng ta xác định nó là kiểu gì. Ví dụ như ta có 2 loại gửi tin nhắn là Email và SMS cùng implements 1 interface là Message Service. Chúng ta sử dụng @Qualifier để xác định đó là loại Email hay SMS vì chúng cùng 1 kiểu Message Service.

```
public interface MessageService {
    public void sendMsg(String message);
}
```

```
public class EmailService implements MessageService{
    public void sendMsg(String message) {
        System.out.println(message);
    }
}
```

```
public class SMSService implements MessageService{
    public void sendMsg(String message) {
        System.out.println(message);
    }
}
```

Annotation @Qualifier


```
public interface MessageProcessor {
    public void processMsg(String message);
}
public class MessageProcessorImpl implements MessageProcessor {
    private MessageService messageService;
    // setter based DI
    @Autowired
    @Qualifier("emailService")
    public void setMessageService(MessageService messageService) {
        this.messageService = messageService;
    }
    // constructor based DI
    @Autowired
    public MessageProcessorImpl(@Qualifier("emailService")
MessageService messageService) {
        this.messageService = messageService;
    }
    public void processMsg(String message) {
        messageService.sendMsg(message);
    }
}
```