https://bytemartdigital.in/

{code}

API

Spring Framework By ByteMart

Copyright and Disclaimer

Spring Framework by ByteMart

Copyright Notice

For permission inquiries, please contact:
Email: contact@bytemartdigital.in
Website: bytemartdigital.in

Disclaimer

This book is provided "as is" without any guarantees or warranties, express or implied. While every effort has been made to ensure accuracy and completeness, ByteMart makes no representations regarding the currentness, performance, or applicability of the information.
The contents are intended for **educational purposes only**. As technologies such as Hibernate are constantly evolving, readers are advised to consult the official documentation for the most up-to-date information and adapt the examples accordingly.

Legal and Ethical Use Guidelines

- Reproduction, republication, or unauthorized distribution of this book, in part or in full, is **strictly prohibited**.
- Sharing this content on unauthorized platforms or using it for commercial purposes without written permission is **illegal**.
- Using code or content from this book for **unethical or unlawful activities** is not permitted.

⚠️ **Warning**

This book is the result of significant expertise and effort. Any act of intellectual property theft—including plagiarism, unauthorized redistribution, or content misuse—will result in **immediate legal action** under applicable copyright and IP laws.
ByteMart assumes no responsibility for any loss, damage, or consequences resulting from the use of the information or code provided in this publication.

By Accessing or Using This Book, You Agree To:

1. Respect the intellectual property rights of ByteMart.
2. Use the content responsibly and ethically.
3. Refrain from any activity that violates the copyright terms outlined in this document.

ByteMart reserves the right to pursue **strict legal action** against any unauthorized use, duplication, or distribution of this material.
Proceed responsibly.

Contact Information

For permissions, collaborations, or support inquiries:

- **Email:** contact@bytemartdigital.in
- **Website:** https://bytemartdigital.in

Spring Framework

Table of Content

Chapter 1: Getting Started with Spring Framework

Section 1.1: Setup (XML Configuration)

Introduction

Spring Framework is one of the most popular frameworks for building Java applications. It provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so developers can focus on application logic. In this chapter, we'll cover the basics of getting started with Spring, including setting up your environment and understanding the core features of the framework.

Setup (XML Configuration)

To start with Spring, we need to set up a basic project structure and configure Spring using XML configuration. Here are the steps:

1. **Create a Maven Project:**
   First, create a Maven project. Maven is a build automation tool used for managing project dependencies and building the project.

```
<project xmlns="<http://maven.apache.org/POM/4.0.0>"
         xmlns:xsi="<http://www.w3.org/2001/XMLSchema-in
stance>"
         xsi:schemaLocation="<http://maven.apache.org/PO
M/4.0.0> <http://maven.apache.org/xsd/maven-4.0.0.xsd>">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>spring-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.9</version>
        </dependency>
    </dependencies>
</project>
```

2. **Create Spring Configuration File:**
   Create an XML configuration file named `applicationContext.xml` in the `src/main/resources` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="<http://www.springframework.org/schema/beans>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
       xsi:schemaLocation="<http://www.springframework.org/schema/beans>
           <http://www.springframework.org/schema/beans/spring-beans.xsd>">

    <!-- Bean Definitions -->
    <bean id="myBean" class="com.example.MyBean">
        <property name="property" value="Hello, Spring!" />
    </bean>

</beans>
```

3. **Create a Simple Bean:**
   Create a Java class named `MyBean` in the `src/main/java/com/example` directory.

```
package com.example;

public class MyBean {
    private String property;

    public void setProperty(String property) {
        this.property = property;
    }

    public void printProperty() {
        System.out.println("Property Value: " + property);
    }
}
```

4. **Initialize the Spring Context:**
   Create a main class to load the Spring context and retrieve the bean.

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlA
pplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApp
licationContext("applicationContext.xml");
        MyBean myBean = (MyBean) context.getBean("myBea
n");
        myBean.printProperty();
    }
}
```


Running the Application

1. **Compile the Project:**
   Run the following Maven command to compile the project:

```
mvn compile
```

2. **Run the Application:**
   Run the main class:

```
mvn exec:java -Dexec.mainClass="com.example.SpringExampl
e"
```

You should see the following output:

```
Property Value: Hello, Spring!
```

This simple setup demonstrates how to configure Spring using XML. In the next sections, we will explore more about Spring features and how to leverage them

in your applications.

Section 1.2: Showcasing Core Spring Features by Example

Introduction

Spring Framework offers a multitude of features that facilitate the development of robust, scalable, and maintainable applications. In this section, we will showcase some of the core features of Spring by example.

Dependency Injection (DI)

Dependency Injection is a fundamental concept in Spring, which allows the creation of dependent objects outside of a class and provides those objects to a class in different ways.

Example:

```
<!-- applicationContext.xml -->
<bean id="dependencyBean" class="com.example.DependencyBea
n">
    <property name="message" value="Injected Dependency!" /
>
</bean>

<bean id="dependentBean" class="com.example.DependentBean">
    <property name="dependency" ref="dependencyBean" />
</bean>
```

```
package com.example;

public class DependencyBean {
    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
```

```
    }
}
```

```
package com.example;

public class DependentBean {
    private DependencyBean dependency;

    public void setDependency(DependencyBean dependency) {
        this.dependency = dependency;
    }

    public void showDependencyMessage() {
        System.out.println(dependency.getMessage());
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        DependentBean dependentBean = (DependentBean) conte
xt.getBean("dependentBean");
        dependentBean.showDependencyMessage();
    }
}
```

Output:

```
Injected Dependency!
```

Aspect-Oriented Programming (AOP)

Spring AOP allows for the modularization of concerns such as transaction management, logging, or security.

Example:

```
<!-- applicationContext.xml -->
<aop:config>
    <aop:aspect ref="myAspect">
        <aop:pointcut id="myPointcut" expression="execution
(* com.example.MyService.*(..))" />
        <aop:before method="beforeAdvice" pointcut-ref="myP
ointcut" />
    </aop:aspect>
</aop:config>

<bean id="myAspect" class="com.example.MyAspect" />
<bean id="myService" class="com.example.MyService" />
```

```
package com.example;

public class MyService {
    public void doSomething() {
        System.out.println("Doing something in MyService");
    }
}
```

```
package com.example;

public class MyAspect {
    public void beforeAdvice() {
        System.out.println("Before Advice: Method is about
to be called.");
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        MyService myService = (MyService) context.getBean
("myService");
        myService.doSomething();
    }
}
```

Output:

```
Before Advice: Method is about to be called.
Doing something in MyService
```

These examples highlight some of the core features of Spring, including Dependency Injection and Aspect-Oriented Programming. As you continue through this book, you'll discover many more capabilities and best practices for using Spring Framework in your projects.

Section 1.3: What is Spring Framework, Why Should We Go for It?

Introduction

Spring Framework is a comprehensive framework for enterprise Java development. It provides a wide range of functionalities that can be used to develop any kind of Java application. Its popularity is due to its simplicity, power, and ease of use.

What is Spring Framework?

Spring Framework is an open-source framework created to address the complexity of enterprise application development. It offers a comprehensive programming and configuration model for modern Java-based enterprise applications.

Key Features:

- **Inversion of Control (IoC):** Manages the dependencies of your application.
- **Aspect-Oriented Programming (AOP):** Modularizes cross-cutting concerns.
- **Data Access Framework:** Simplifies database interaction.
- **Transaction Management:** Manages transactions declaratively.
- **MVC Framework:** Simplifies the development of web applications.
- **Batch Processing:** Supports the creation of robust batch processing applications.
- **Integration:** Integrates with various enterprise services.

Why Should We Go for Spring Framework?

1. **Modularity:**
   Spring promotes modularity by breaking down the application into modules. This separation of concerns makes the code easier to manage and understand.
2. **Dependency Injection:**
   Spring's IoC container manages the components' lifecycle and configuration, promoting loose coupling and better testability.
3. **Aspect-Oriented Programming:**
   AOP helps separate cross-cutting concerns, such as logging and transaction management, from the business logic.
4. **Comprehensive Ecosystem:**
   Spring provides a rich ecosystem with various projects like Spring Boot, Spring Data, Spring Security, and Spring Cloud, which offer ready-to-use solutions for different needs.
5. **Community and Support:**
   Spring has a large and active community, which means a wealth of resources, tutorials, and third-party support is available.

6. **Integration:**
   Spring integrates seamlessly with various technologies, frameworks, and enterprise services, making it a versatile choice for many types of applications.

Conclusion

Spring Framework is a powerful tool for building enterprise applications. Its modularity, flexibility, and comprehensive ecosystem make it a go-to choice for developers. In the upcoming chapters, we will delve deeper into the various features and components of Spring, providing you with a solid foundation

Chapter 2: Spring Core

Section 2.1: Introduction to Spring Core

Introduction

The core of the Spring Framework is its Inversion of Control (IoC) container, which manages the lifecycle and configuration of application objects. This container uses Dependency Injection (DI) to manage components, making the application more modular, testable, and maintainable. In this chapter, we will explore the fundamental concepts of Spring Core and understand how Spring manages dependencies.

Inversion of Control (IoC)

Inversion of Control is a design principle where the control of object creation and management is transferred from the application code to the framework. Spring's IoC container is responsible for instantiating, configuring, and assembling the beans.

Dependency Injection (DI)

Dependency Injection is a pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through constructor injection, setter injection, or field injection.

Example: Constructor Injection

```
<!-- applicationContext.xml -->
<bean id="dependencyBean" class="com.example.DependencyBea
```

```
n">
    <constructor-arg value="Injected Dependency!" />
</bean>

<bean id="dependentBean" class="com.example.DependentBean">
    <constructor-arg ref="dependencyBean" />
</bean>
```

```
package com.example;

public class DependencyBean {
    private String message;

    public DependencyBean(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

```
package com.example;

public class DependentBean {
    private DependencyBean dependency;

    public DependentBean(DependencyBean dependency) {
        this.dependency = dependency;
    }

    public void showDependencyMessage() {
        System.out.println(dependency.getMessage());
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        DependentBean dependentBean = (DependentBean) conte
xt.getBean("dependentBean");
        dependentBean.showDependencyMessage();
    }
}
```

Output:

```
Injected Dependency!
```

Setter Injection

Example:

```
<!-- applicationContext.xml -->
<bean id="dependencyBean" class="com.example.DependencyBea
n">
    <property name="message" value="Injected Dependency via
Setter!" />
</bean>

<bean id="dependentBean" class="com.example.DependentBean">
    <property name="dependency" ref="dependencyBean" />
</bean>
```

```
package com.example;
```

```
public class DependencyBean {
    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```

```
package com.example;

public class DependentBean {
    private DependencyBean dependency;

    public void setDependency(DependencyBean dependency) {
        this.dependency = dependency;
    }

    public void showDependencyMessage() {
        System.out.println(dependency.getMessage());
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        DependentBean dependentBean = (DependentBean) conte
```

```
xt.getBean("dependentBean");
        dependentBean.showDependencyMessage();
    }
}
```

Output:

```
Injected Dependency via Setter!
```

Section 2.2: Understanding How Spring Manages Dependency

Bean Factory

The `BeanFactory` is the root interface for accessing the Spring container. It provides basic functionalities to manage beans.

Example:

```
package com.example;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactor
y;
import org.springframework.core.io.ClassPathResource;

public class BeanFactoryExample {
    public static void main(String[] args) {
        BeanFactory factory = new XmlBeanFactory(new ClassP
athResource("applicationContext.xml"));
        DependentBean dependentBean = (DependentBean) facto
ry.getBean("dependentBean");
        dependentBean.showDependencyMessage();
    }
}
```

ApplicationContext

The `ApplicationContext` is a more advanced container that includes all the functionalities of `BeanFactory` and adds more enterprise-specific functionalities,

such as event propagation, declarative mechanisms to create a bean, and more.

Example:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class ApplicationContextExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        DependentBean dependentBean = (DependentBean) conte
xt.getBean("dependentBean");
        dependentBean.showDependencyMessage();
    }
}
```

Bean Scopes

Spring provides different scopes for managing the lifecycle of beans. The default scope is singleton, but other scopes include prototype, request, session, and global session.

Singleton Scope:

The singleton scope ensures that only one instance of the bean is created for the entire Spring IoC container.

Prototype Scope:

The prototype scope creates a new instance of the bean every time it is requested from the container.

Example:

```
<!-- applicationContext.xml -->
<bean id="singletonBean" class="com.example.MyBean" scope
="singleton" />
```

```
<bean id="prototypeBean" class="com.example.MyBean" scope
="prototype" />
```

Section 2.3: Bean Lifecycle

Spring provides several lifecycle hooks that allow you to customize the behavior of your beans. These hooks include:

1. **InitializingBean and DisposableBean interfaces:**
   Implement these interfaces to define custom initialization and destruction logic.
2. **Custom init and destroy methods:**
   Specify custom init and destroy methods in the bean configuration.

Example:

```
<!-- applicationContext.xml -->
<bean id="lifecycleBean" class="com.example.LifecycleBean"
init-method="customInit" destroy-method="customDestroy" />
```

```
package com.example;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class LifecycleBean implements InitializingBean, Dis
posableBean {

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("InitializingBean: afterProperti
esSet method called");
    }

    @Override
    public void destroy() throws Exception {
        System.out.println("DisposableBean: destroy method
called");
    }
```

```
    public void customInit() {
        System.out.println("Custom init method called");
    }

    public void customDestroy() {
        System.out.println("Custom destroy method called");
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class LifecycleExample {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new ClassP
athXmlApplicationContext("applicationContext.xml");
        LifecycleBean lifecycleBean = (LifecycleBean) conte
xt.getBean("lifecycleBean");
        context.close();
    }
}
```

Output:

```
InitializingBean: afterPropertiesSet method called
Custom init method called
DisposableBean: destroy method called
Custom destroy method called
```

This chapter provides a comprehensive understanding of the Spring Core module, covering IoC, DI, bean scopes, and the bean lifecycle. These concepts form the foundation of the Spring Framework and are essential for building

robust and maintainable applications. In the next chapter, we will explore the Spring Expression Language (SpEL) and its usage in Spring applications.

Chapter 3: Spring Expression Language (SpEL)

Section 3.1: Syntax Reference

Introduction

Spring Expression Language (SpEL) is a powerful expression language that supports querying and manipulating an object graph at runtime. It is used extensively in Spring configuration to wire beans, perform operations, and access properties dynamically. In this chapter, we will explore the syntax and usage of SpEL with various examples.

Basic Syntax

SpEL expressions are usually defined within the `${}` syntax in XML or `@Value` annotations. Here are some of the basic operations supported by SpEL:

1. **Literals:**
   - **String:** `'Hello World'`
   - **Numeric:** `42`, `3.14`
   - **Boolean:** `true`, `false`
   - **Null:** `null`
2. **Properties:**
   Accessing properties of beans and objects.

   ```
   <property name="propertyName" value="#{beanName.property}" />
   ```

3. **Methods:**
   Invoking methods on beans and objects.

   ```
   <property name="propertyName" value="#{beanName.methodName()}" />
   ```

4. **Arithmetic Operators:**
   Basic arithmetic operations like `+`, `-`, `*`, `/`, `%`.

```
<property name="propertyName" value="#{beanName.value1 + beanName.value2}" />
```

5. **Logical Operators:**
   Logical operations like `and`, `or`, `not`.

```
<property name="propertyName" value="#{beanName.booleanValue1 and beanName.booleanValue2}" />
```

6. **Relational Operators:**
   Relational operations like `==`, `!=`, `<`, `>`, `<=`, `>=`.

```
<property name="propertyName" value="#{beanName.value1 > beanName.value2}" />
```

7. **Conditional Operator (ternary):**
   Ternary conditional operator `? :`.

```
<property name="propertyName" value="#{beanName.value1 > 0 ? 'Positive' : 'Negative'}" />
```

8. **Elvis Operator:**
   Elvis operator `?:` for null checks.

```
<property name="propertyName" value="#{beanName.property ?: 'Default'}" />
```

9. **Safe Navigation Operator:**
   To avoid null pointer exceptions, `?.` operator is used.

```
<property name="propertyName" value="#{beanName?.propert
y}" />
```

Accessing Bean Properties

Example:

```
<!-- applicationContext.xml -->
<bean id="exampleBean" class="com.example.ExampleBean">
    <property name="name" value="Spring Framework" />
    <property name="version" value="5.3.9" />
</bean>

<bean id="anotherBean" class="com.example.AnotherBean">
    <property name="description" value="#{exampleBean.name}
- Version: #{exampleBean.version}" />
</bean>
```

```
package com.example;

public class ExampleBean {
    private String name;
    private String version;

    // getters and setters
}

public class AnotherBean {
    private String description;

    public void setDescription(String description) {
        this.description = description;
    }

    public void printDescription() {
        System.out.println(description);
```

```
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpELExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        AnotherBean anotherBean = (AnotherBean) context.get
Bean("anotherBean");
        anotherBean.printDescription();
    }
}
```

Output:

```
Spring Framework - Version: 5.3.9
```


Method Invocation

Example:

```
<!-- applicationContext.xml -->
<bean id="exampleBean" class="com.example.ExampleBean">
    <property name="name" value="Spring Framework" />
</bean>

<bean id="anotherBean" class="com.example.AnotherBean">
    <property name="uppercaseName" value="#{exampleBean.nam
e.toUpperCase()}" />
</bean>
```

```
package com.example;

public class ExampleBean {
    private String name;

    // getters and setters
}

public class AnotherBean {
    private String uppercaseName;

    public void setUppercaseName(String uppercaseName) {
        this.uppercaseName = uppercaseName;
    }

    public void printUppercaseName() {
        System.out.println(uppercaseName);
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpELExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        AnotherBean anotherBean = (AnotherBean) context.get
Bean("anotherBean");
        anotherBean.printUppercaseName();
    }
}
```

Output:

```
SPRING FRAMEWORK
```

Using SpEL in Annotations

SpEL can also be used in annotations, such as `@Value`.

Example:

```
<!-- applicationContext.xml -->
<bean id="exampleBean" class="com.example.ExampleBean">
    <property name="name" value="Spring Framework" />
</bean>
```

```
package com.example;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class AnotherBean {
    @Value("#{exampleBean.name}")
    private String name;

    public void printName() {
        System.out.println(name);
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpELExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
```

```
plicationContext(AppConfig.class);
        AnotherBean anotherBean = context.getBean(AnotherBe
an.class);
        anotherBean.printName();
    }
}
```

```
package com.example;

import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

Output:

```
Spring Framework
```

Collection Selection and Projection

SpEL supports powerful features for working with collections, including selection and projection.

Selection Example:

```
package com.example;

import java.util.List;

public class ExampleBean {
    private List<String> items;

    public void setItems(List<String> items) {
        this.items = items;
```

```
    }

    public List<String> getItems() {
        return items;
    }
}
```

```
<!-- applicationContext.xml -->
<bean id="exampleBean" class="com.example.ExampleBean">
    <property name="items">
        <list>
            <value>Apple</value>
            <value>Banana</value>
            <value>Cherry</value>
            <value>Date</value>
        </list>
    </property>
</bean>

<bean id="selectedItemsBean" class="com.example.SelectedIte
msBean">
    <property name="selectedItems" value="#{exampleBean.ite
ms.?[startsWith('B')]}" />
</bean>
```

```
package com.example;

import java.util.List;

public class SelectedItemsBean {
    private List<String> selectedItems;

    public void setSelectedItems(List<String> selectedItem
s) {
        this.selectedItems = selectedItems;
    }
```

```
    public void printSelectedItems() {
        System.out.println(selectedItems);
    }
}
```

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpELExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        SelectedItemsBean selectedItemsBean = (SelectedItem
sBean) context.getBean("selectedItemsBean");
        selectedItemsBean.printSelectedItems();
    }
}
```

Output:

```
[Banana]
```

Conclusion

Spring Expression Language (SpEL) is a versatile and powerful tool for querying and manipulating object graphs at runtime. It provides a wide range of features, from basic property access and method invocation to more advanced collection manipulation and conditional logic. By leveraging SpEL, you can make your Spring configuration more dynamic and flexible, leading to more maintainable and robust applications.

In the next chapter, we will explore how to obtain a `SqlRowSet` from `SimpleJdbcCall` and delve into database interactions with Spring.

Chapter 4: Obtaining a SqlRowSet from SimpleJdbcCall

Section 4.1: SimpleJdbcCall Creation

Introduction

`SimpleJdbcCall` is a multi-threaded, reusable object provided by Spring's `JdbcTemplate` for executing stored procedures and functions in a relational database. It simplifies the execution of stored procedures and functions, handling input and output parameters, and returning results in various formats, including `SqlRowSet`.

Setting Up `SimpleJdbcCall`

To use `SimpleJdbcCall`, you need to configure your database connection and set up the `JdbcTemplate` and `DataSource` beans in your Spring configuration.

Example Configuration:

```
<!-- applicationContext.xml -->
<bean id="dataSource" class="org.springframework.jdbc.datas
ource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jd
bc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:330
6/mydatabase" />
    <property name="username" value="myusername" />
    <property name="password" value="mypassword" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.cor
e.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Creating a `SimpleJdbcCall`

To create a `SimpleJdbcCall`, you need to inject the `JdbcTemplate` and use it to instantiate the `SimpleJdbcCall` object.

Example:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.stereotype.Component;

@Component
public class MyStoredProcedure {

    private SimpleJdbcCall simpleJdbcCall;

    @Autowired
    public MyStoredProcedure(JdbcTemplate jdbcTemplate) {
        this.simpleJdbcCall = new SimpleJdbcCall(jdbcTempla
te)
            .withProcedureName("my_stored_procedure");
    }

    public SqlRowSet executeStoredProcedure(Map<String, Obj
ect> inParams) {
        return simpleJdbcCall.execute(inParams);
    }
}
```

In this example, `MyStoredProcedure` is a component that defines a `SimpleJdbcCall` to execute a stored procedure named `my_stored_procedure`. The `executeStoredProcedure` method takes a map of input parameters and returns a `SqlRowSet`.

Section 4.2: Oracle Databases

Special Considerations for Oracle

When working with Oracle databases, there are some special considerations to keep in mind, such as handling `REF_CURSOR` types and setting up the Oracle-specific data source.

Oracle DataSource Configuration:

```
<!-- applicationContext.xml -->
<bean id="dataSource" class="oracle.jdbc.pool.OracleDataSou
rce">
    <property name="URL" value="jdbc:oracle:thin:@localhos
t:1521:xe" />
    <property name="user" value="myusername" />
    <property name="password" value="mypassword" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.cor
e.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Handling `REF_CURSOR` in Oracle

Oracle stored procedures often use `REF_CURSOR` types to return result sets. Spring's `SimpleJdbcCall` can handle these types efficiently.

Example:


```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.stereotype.Component;

import java.sql.Types;
import java.util.HashMap;
import java.util.Map;

@Component
public class OracleStoredProcedure {
```

```
    private SimpleJdbcCall simpleJdbcCall;

    @Autowired
    public OracleStoredProcedure(JdbcTemplate jdbcTemplate)
{
        this.simpleJdbcCall = new SimpleJdbcCall(jdbcTempla
te)
            .withProcedureName("my_oracle_stored_procedur
e")
            .declareParameters(
                new SqlParameter("in_param", Types.VARCHA
R),
                new SqlOutParameter("out_cursor", Types.REF
_CURSOR));
    }

    public SqlRowSet executeStoredProcedure(String inParam)
{
        Map<String, Object> inParams = new HashMap<>();
        inParams.put("in_param", inParam);

        Map<String, Object> outParams = simpleJdbcCall.exec
ute(inParams);
        return (SqlRowSet) outParams.get("out_cursor");
    }
}
```

In this example, `OracleStoredProcedure` defines a `SimpleJdbcCall` to execute an Oracle stored procedure named `my_oracle_stored_procedure`. It declares an input parameter `in_param` and an output parameter `out_cursor` of type `REF_CURSOR`. The `executeStoredProcedure` method takes an input parameter and returns the result set as a `SqlRowSet`.

Using `SqlRowSet`

`SqlRowSet` is an interface provided by Spring that extends `javax.sql.RowSet` and allows you to navigate and retrieve results from a result set.

Example Usage:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.jdbc.support.rowset.SqlRowSet;
import org.springframework.stereotype.Component;

@Component
public class StoredProcedureExecutor {

    private final OracleStoredProcedure oracleStoredProcedu
re;

    @Autowired
    public StoredProcedureExecutor(OracleStoredProcedure or
acleStoredProcedure) {
        this.oracleStoredProcedure = oracleStoredProcedure;
    }

    public void printStoredProcedureResults(String param) {
        SqlRowSet rowSet = oracleStoredProcedure.executeSto
redProcedure(param);

        while (rowSet.next()) {
            System.out.println("Column1: " + rowSet.getStri
ng("column1"));
            System.out.println("Column2: " + rowSet.getInt
("column2"));
        }
    }
}
```

In this example, `StoredProcedureExecutor` uses `OracleStoredProcedure` to execute the stored procedure and retrieve results as a `SqlRowSet`. It then iterates over the `SqlRowSet` and prints the values of `column1` and `column2`.

Conclusion

`SimpleJdbcCall` provides a powerful and flexible way to interact with stored procedures and functions in relational databases, including handling Oracle's `REF_CURSOR` types. By using `SimpleJdbcCall` and `SqlRowSet`, you can efficiently execute stored procedures and retrieve results in a manner that integrates seamlessly with Spring's data access support.

In the next chapter, we will explore the creation and use of beans in Spring, focusing on autowiring, annotations, and dynamic bean instantiation using `FactoryBean`.

Chapter 5: Creating and Using Beans

Introduction

Spring Framework revolves around the concept of a "bean," which is an object that is instantiated, assembled, and managed by the Spring IoC container. In this chapter, we'll explore various methods to create and use beans, including autowiring, annotations, and dynamic bean instantiation using `FactoryBean`.

Section 5.1: Autowiring All Beans of a Specific Type

Introduction

Autowiring is a feature of Spring that enables automatic injection of dependencies into a bean. It reduces the need for explicit configuration by allowing Spring to resolve and inject the correct beans automatically. In this section, we'll discuss how to autowire all beans of a specific type.

Example

Beans Configuration:

```
<!-- applicationContext.xml -->
<bean id="service1" class="com.example.MyServiceImpl1"/>
<bean id="service2" class="com.example.MyServiceImpl2"/>
<bean id="service3" class="com.example.MyServiceImpl3"/>
<bean id="aggregator" class="com.example.ServiceAggregato
r"/>
```

Service Interface and Implementations:

```
package com.example;

public interface MyService {
    void performService();
}
```

```
package com.example;

public class MyServiceImpl1 implements MyService {
    @Override
    public void performService() {
        System.out.println("Service 1 performing...");
    }
}
```

```
package com.example;

public class MyServiceImpl2 implements MyService {
    @Override
    public void performService() {
        System.out.println("Service 2 performing...");
    }
}
```

```
package com.example;

public class MyServiceImpl3 implements MyService {
    @Override
    public void performService() {
        System.out.println("Service 3 performing...");
    }
}
```

Service Aggregator:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Component;

import java.util.List;

@Component
public class ServiceAggregator {
    private List<MyService> services;

    @Autowired
    public void setServices(List<MyService> services) {
        this.services = services;
    }

    public void performAllServices() {
        for (MyService service : services) {
            service.performService();
        }
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        ServiceAggregator aggregator = (ServiceAggregator)
context.getBean("aggregator");
```

```
        aggregator.performAllServices();
    }
}
```

Output:

```
Service 1 performing...
Service 2 performing...
Service 3 performing...
```

Section 5.2: Basic Annotation Autowiring

Introduction

Spring supports autowiring using annotations, making configuration easier and more readable. The `@Autowired` annotation is used to automatically wire dependencies.

Example

Beans Configuration:

Using annotations eliminates the need for explicit bean definitions in the XML configuration.

```
package com.example;

import org.springframework.stereotype.Component;

@Component
public class MyService {
    public void performService() {
        System.out.println("Performing service...");
    }
}
```

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
```

```
import org.springframework.stereotype.Component;

@Component
public class MyComponent {
    private MyService myService;

    @Autowired
    public void setMyService(MyService myService) {
        this.myService = myService;
    }

    public void perform() {
        myService.performService();
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        MyComponent component = context.getBean(MyComponen
t.class);
        component.perform();
    }
}
```

Configuration Class:

```
package com.example;
```

```
import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

Output:

```
Performing service...
```

Section 5.3: Using `FactoryBean` for Dynamic Bean Instantiation

Introduction

`FactoryBean` is a special bean in Spring that allows you to customize the instantiation logic of a bean. It provides more control over the creation of beans, enabling dynamic and complex object creation processes.

Example

Custom FactoryBean:

```
package com.example;

import org.springframework.beans.factory.FactoryBean;

public class MyFactoryBean implements FactoryBean<MyBean> {
    @Override
    public MyBean getObject() throws Exception {
        return new MyBean("Dynamic Bean");
    }

    @Override
    public Class<?> getObjectType() {
        return MyBean.class;
```

```
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

Bean Class:

```
package com.example;

public class MyBean {
    private String name;

    public MyBean(String name) {
        this.name = name;
    }

    public void printName() {
        System.out.println("Bean name: " + name);
    }
}
```

Beans Configuration:

```
<!-- applicationContext.xml -->
<bean id="myFactoryBean" class="com.example.MyFactoryBean"/
>
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;
```

```
public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        MyBean myBean = (MyBean) context.getBean("myFactory
Bean");
        myBean.printName();
    }
}
```

Output:

```
Bean name: Dynamic Bean
```

Section 5.4: Declaring Bean

Introduction

Beans can be declared in multiple ways: using XML configuration, Java configuration, or annotations. Each method has its advantages and is suitable for different scenarios.

Declaring Beans Using XML Configuration

Example:

```
<!-- applicationContext.xml -->
<bean id="myBean" class="com.example.MyBean">
    <property name="name" value="XML Configured Bean"/>
</bean>
```

Bean Class:

```
package com.example;

public class MyBean {
    private String name;

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

```
    }

    public void printName() {
        System.out.println("Bean name: " + name);
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        MyBean myBean = (MyBean) context.getBean("myBean");
        myBean.printName();
    }
}
```

Output:

```
Bean name: XML Configured Bean
```

Declaring Beans Using Java Configuration

Example:

```
package com.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
```

```
public class AppConfig {
    @Bean
    public MyBean myBean() {
        MyBean bean = new MyBean();
        bean.setName("Java Configured Bean");
        return bean;
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        MyBean myBean = context.getBean(MyBean.class);
        myBean.printName();
    }
}
```

Output:

```
Bean name: Java Configured Bean
```

Declaring Beans Using Annotations

Example:

```
package com.example;

import org.springframework.stereotype.Component;

@Component
```

```
public class MyBean {
    private String name = "Annotated Bean";

    public void printName() {
        System.out.println("Bean name: " + name);
    }
}
```

Configuration Class:

```
package com.example;

import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        MyBean myBean = context.getBean(MyBean.class);
        myBean.printName();
    }
}
```

Output:

```
Bean name: Annotated Bean
```

Section 5.5: Autowiring Specific Bean Instances with `@Qualifier`

Introduction

When multiple beans of the same type are available, Spring's `@Autowired` annotation might not know which bean to inject. The `@Qualifier` annotation helps resolve this ambiguity by specifying the exact bean to autowire.

Example

Beans Configuration:

```
package com.example;

import org.springframework.stereotype.Component;

@Component("service1")
public class MyService1 implements MyService {
    @Override
    public void performService() {
        System.out.println("Service 1 performing...");
    }
}
```

```
package com.example;

import org.springframework.stereotype.Component;

@Component("service2")
public class MyService2 implements MyService {
    @Override
    public void performService() {
        System.out.println("Service 2 performing...");
```

```
    }
}
```

Service Interface:

```
package com.example;

public interface MyService {
    void performService();
}
```

Service Consumer:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.beans.factory.annotation.Qualifi
er;
import org.springframework.stereotype.Component;

@Component
public class ServiceConsumer {

    private MyService myService;

    @Autowired
    public void setMyService(@Qualifier("service1") MyServi
ce myService) {
        this.myService = myService;
    }

    public void performService() {
        myService.performService();
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        ServiceConsumer consumer = context.getBean(ServiceC
onsumer.class);
        consumer.performService();
    }
}
```

Configuration Class:

```
package com.example;

import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

Output:

```
Service 1 performing...
```

Section 5.6: Autowiring Specific Instances of Classes Using Generic Type Parameters

Introduction

Spring supports autowiring based on generic type parameters. This can be particularly useful when you have multiple beans of the same type but with different generic parameters.

Example

Beans Configuration:

Service Interface:

```
package com.example;

public interface MyService<T> {
```

```
    T process();
}
```

Service Consumer:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Component;

@Component
public class GenericServiceConsumer {

    private MyService<Integer> integerService;
    private MyService<String> stringService;

    @Autowired
    public void setIntegerService(MyService<Integer> intege
rService) {
        this.integerService = integerService;
    }

    @Autowired
    public void setStringService(MyService<String> stringSe
rvice) {
        this.stringService = stringService;
    }

    public void processServices() {
        System.out.println("Integer Service Result: " + int
egerService.process());
        System.out.println("String Service Result: " + stri
ngService.process());
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        GenericServiceConsumer consumer = context.getBean(G
enericServiceConsumer.class);
        consumer.processServices();
    }
}
```

Configuration Class:

```
package com.example;

import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

Output:

```
Integer Service Result: 42
String Service Result: Hello
```

Section 5.7: Inject Prototype-Scoped Beans into Singletons

Introduction

In Spring, beans can have different scopes. A common requirement is to inject prototype-scoped beans into singleton beans. Spring provides ways to achieve this while ensuring that a new instance of the prototype bean is created every time it is requested.

Example

Prototype Bean:

```
package com.example;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class PrototypeBean {
    public void printMessage() {
        System.out.println("Prototype Bean instance: " + th
is);
    }
}
```

Singleton Bean:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.context.ApplicationContext;
import org.springframework.stereotype.Component;

@Component
public class SingletonBean {

    @Autowired
    private ApplicationContext context;
```

```
    public void usePrototypeBean() {
        PrototypeBean prototypeBean = context.getBean(Proto
typeBean.class);
        prototypeBean.printMessage();
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        SingletonBean singletonBean = context.getBean(Singl
etonBean.class);

        singletonBean.usePrototypeBean();
        singletonBean.usePrototypeBean();
    }
}
```

Configuration Class:

```
package com.example;

import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
@ComponentScan(basePackages = "com.example")
```

```
public class AppConfig {
}
```

Output:

```
Prototype Bean instance: com.example.PrototypeBean@12345
Prototype Bean instance: com.example.PrototypeBean@67890
```

Conclusion

In this chapter, we explored various methods of creating and using beans in Spring, including autowiring all beans of a specific type, basic annotation autowiring, using `FactoryBean` for dynamic bean instantiation, declaring beans using different methods, autowiring specific bean instances with `@Qualifier`, autowiring specific instances of classes using generic type parameters, and injecting prototype-scoped beans into singletons.

In the next chapter, we will delve into bean scopes, discussing additional scopes in web-aware contexts, prototype scope, and singleton scope.

Chapter 7: Conditional Bean Registration in Spring

Introduction

In many applications, you may need to create and register beans conditionally based on certain criteria, such as the presence of a property, the operating system, or the existence of a particular class. Spring provides several mechanisms to achieve this, including the `@Conditional` annotation and various condition annotations like `@ConditionalOnProperty`, `@ConditionalOnClass`, and `@ConditionalOnMissingBean`. This chapter will explore these mechanisms and provide examples of how to use them effectively.

Section 7.1: Register Beans Only When a Property or Value Is Specified

Introduction

One common use case for conditional bean registration is to register a bean only if a specific property or value is set. Spring Boot provides the

`@ConditionalOnProperty` annotation to achieve this.

Example: Using `@ConditionalOnProperty`

Bean Definition:

```
package com.example;

import org.springframework.boot.autoconfigure.condition.Con
ditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
public class ConditionalConfig {

    @Bean
    @ConditionalOnProperty(name = "feature.enabled", having
Value = "true")
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Application Properties:

```
# application.properties
feature.enabled=true
```

Service Interface:

```
package com.example;

public interface MyService {
    void performService();
}
```

Service Implementation:

```
package com.example;

public class MyServiceImpl implements MyService {
    @Override
    public void performService() {
        System.out.println("Feature is enabled!");
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run
(SpringExample.class, args);
        MyService myService = context.getBean(MyService.cla
ss);
        myService.performService();
    }
}
```

Output:

```
Feature is enabled!
```

Section 7.2: Condition Annotations

Introduction

Spring Boot provides a variety of condition annotations to register beans based on different conditions, such as the presence or absence of classes, beans, and properties. These annotations help to manage the configuration and initialization of beans in a more flexible and maintainable way.

Example: Using `@ConditionalOnClass`

Bean Definition:

```
package com.example;

import org.springframework.boot.autoconfigure.condition.Con
ditionalOnClass;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
public class ConditionalConfig {

    @Bean
    @ConditionalOnClass(name = "com.example.SomeClass")
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Class to Check:

```
package com.example;

// This class needs to be present for the condition to be t
rue
public class SomeClass {
}
```

Service Implementation:

```
package com.example;

public class MyServiceImpl implements MyService {
    @Override
    public void performService() {
        System.out.println("SomeClass is present!");
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run
(SpringExample.class, args);
        MyService myService = context.getBean(MyService.cla
ss);
        myService.performService();
    }
}
```

Output:

```
SomeClass is present!
```

Example: Using `@ConditionalOnMissingBean`

Bean Definition:

```
package com.example;

import org.springframework.boot.autoconfigure.condition.Con
ditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
public class ConditionalConfig {

    @Bean
    @ConditionalOnMissingBean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Service Implementation:

```
package com.example;

public class MyServiceImpl implements MyService {
    @Override
    public void performService() {
        System.out.println("Default MyService bean create
d!");
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.context.ApplicationContext;
```

```
@SpringBootApplication
public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run
(SpringExample.class, args);
        MyService myService = context.getBean(MyService.cla
ss);
        myService.performService();
    }
}
```

Output:

```
Default MyService bean created!
```

Section 7.3: Custom Conditions

Introduction

In addition to the provided condition annotations, you can create custom conditions by implementing the `Condition` interface and using the `@Conditional` annotation.

Example: Custom Condition

Custom Condition Implementation:

```
package com.example;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionCont
ext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class CustomCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, Annota
tedTypeMetadata metadata) {
        // Add custom logic to determine whether the condit
```

```
ion matches
        String property = context.getEnvironment().getPrope
rty("custom.condition.enabled");
        return "true".equals(property);
    }
}
```

Bean Definition:

```
package com.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
public class ConditionalConfig {

    @Bean
    @Conditional(CustomCondition.class)
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Application Properties:

```
# application.properties
custom.condition.enabled=true
```

Service Implementation:

```
package com.example;

public class MyServiceImpl implements MyService {
    @Override
    public void performService() {
```

```
        System.out.println("Custom condition met!");
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run
(SpringExample.class, args);
        MyService myService = context.getBean(MyService.cla
ss);
        myService.performService();
    }
}
```

Output:

```
Custom condition met!
```

Conclusion

Conditional bean registration in Spring provides a powerful and flexible way to control the creation and configuration of beans based on various conditions. By leveraging annotations like `@ConditionalOnProperty`, `@ConditionalOnClass`, `@ConditionalOnMissingBean`, and custom conditions, you can create more dynamic and adaptable Spring applications.

In the next chapter, we will explore Spring JSR 303 Bean Validation, discussing how to use annotations to validate your data and customize error messages.

Chapter 8: Spring JSR 303 Bean Validation

Introduction

JSR 303, also known as Bean Validation, provides a way to declare constraints on object properties using annotations. Spring integrates JSR 303 with its validation framework, allowing you to easily validate input data in your applications. In this chapter, we will explore how to use JSR 303 Bean Validation in Spring, including basic usage, custom error messages, and validation of nested properties.

Section 8.1: @Valid Usage to Validate Nested POJOs

Introduction

The `@Valid` annotation is used to trigger validation on an object, including its nested properties. This is particularly useful when you have complex objects that contain other objects and you need to validate the entire hierarchy.

Example

Domain Classes:

```
package com.example;

import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.NotNull;

public class Address {

    @NotEmpty(message = "Street cannot be empty")
    private String street;

    @NotEmpty(message = "City cannot be empty")
    private String city;

    // Getters and setters
}
```

```
public class User {

    @NotEmpty(message = "Name cannot be empty")
    private String name;

    @NotNull(message = "Address cannot be null")
    @Valid
    private Address address;

    // Getters and setters
}
```

Controller:

```
package com.example;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestControll
er;

import javax.validation.Valid;

@RestController
public class UserController {

    @PostMapping("/users")
    public String createUser(@RequestBody @Valid User user)
{
        return "User is valid";
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApp
lication;

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

Section 8.2: Spring JSR 303 Validation - Customize Error Messages

Introduction

Customizing error messages allows you to provide user-friendly feedback when validation fails. You can define custom error messages directly in the annotation or in a properties file.

Example

Domain Class:

```
package com.example;

import javax.validation.constraints.NotEmpty;

public class Product {

    @NotEmpty(message = "{product.name.notempty}")
    private String name;

    // Getters and setters
}
```

Validation Messages Properties:

```
# messages.properties
product.name.notempty=Product name cannot be empty
```

Configuration:

```
package com.example;

import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.context.support.ResourceBundleMe
ssageSource;
import org.springframework.validation.beanvalidation.LocalV
alidatorFactoryBean;

@Configuration
public class ValidationConfig {

    @Bean
    public MessageSource messageSource() {
        ResourceBundleMessageSource messageSource = new Res
ourceBundleMessageSource();
        messageSource.setBasename("messages");
        return messageSource;
    }

    @Bean
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean bean = new LocalValidator
FactoryBean();
        bean.setValidationMessageSource(messageSource());
        return bean;
    }
}
```

Controller:

```
package com.example;

import org.springframework.web.bind.annotation.PostMapping;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestControll
er;

import javax.validation.Valid;

@RestController
public class ProductController {

    @PostMapping("/products")
    public String createProduct(@RequestBody @Valid Product
product) {
        return "Product is valid";
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;

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

Section 8.3: JSR 303 Annotation Based Validations in Spring Examples

Example: Validating a Registration Form

Domain Class:

```
package com.example;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotEmpty;
import javax.validation.constraints.Size;

public class RegistrationForm {

    @NotEmpty(message = "Username cannot be empty")
    @Size(min = 4, max = 20, message = "Username must be be
tween 4 and 20 characters")
    private String username;

    @NotEmpty(message = "Password cannot be empty")
    @Size(min = 6, message = "Password must be at least 6 c
haracters long")
    private String password;

    @NotEmpty(message = "Email cannot be empty")
    @Email(message = "Email should be valid")
    private String email;

    // Getters and setters
}
```

Controller:

```
package com.example;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestControll
er;
```

```
import javax.validation.Valid;
import java.util.HashMap;
import java.util.Map;

@RestController
public class RegistrationController {

    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody @Valid R
egistrationForm form, BindingResult result) {
        if (result.hasErrors()) {
            Map<String, String> errors = new HashMap<>();
            for (FieldError error : result.getFieldErrors
()) {
                errors.put(error.getField(), error.getDefau
ltMessage());
            }

            return new ResponseEntity<>(errors, HttpStatus.
BAD_REQUEST);
        }
        return new ResponseEntity<>("Registration successfu
l", HttpStatus.OK);
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;

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

```
    }
}
```

Conclusion

Spring's integration with JSR 303 Bean Validation provides a powerful and flexible way to validate input data. By using annotations, you can easily enforce constraints on your domain models and customize error messages to provide meaningful feedback to users. This chapter covered basic validation usage, customization of error messages, and validation of nested properties.

In the next chapter, we will discuss application context configuration, exploring how to bootstrap and configure the Spring ApplicationContext using both Java and XML configurations.

Chapter 9: ApplicationContext Configuration

Introduction

The `ApplicationContext` is a central interface in the Spring framework. It provides configuration information to the application and manages the lifecycle of beans. This chapter will cover different ways to configure the `ApplicationContext`, including autowiring, bootstrapping, and using Java and XML configurations.

Section 9.1: Autowiring

Introduction

Autowiring in Spring allows you to inject the dependencies automatically. It reduces the need for explicit wiring in configuration files or code. Autowiring can be done using annotations such as `@Autowired`, `@Inject`, and `@Resource`.

Example: Using `@Autowired`

Service Interface:

```
package com.example;

public interface GreetingService {
    String greet();
}
```

Service Implementation:

```
package com.example;

import org.springframework.stereotype.Service;

@Service
public class GreetingServiceImpl implements GreetingService
{
    @Override
    public String greet() {
        return "Hello, World!";
    }
}
```

Controller:

```
package com.example;

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Controller;

@Controller
public class GreetingController {

    private final GreetingService greetingService;

    @Autowired
    public GreetingController(GreetingService greetingServi
ce) {
        this.greetingService = greetingService;
    }

    public void sayHello() {
        System.out.println(greetingService.greet());
    }
}
```

Main Application:

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class SpringExample {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run
(SpringExample.class, args);
        GreetingController controller = context.getBean(Gre
etingController.class);
        controller.sayHello();
    }
}
```

Section 9.2: Bootstrapping the ApplicationContext

Introduction

Bootstrapping the `ApplicationContext` is the process of initializing the Spring container and loading the configuration. This can be done programmatically or using configuration files.

Example: Using `AnnotationConfigApplicationContext`

Configuration Class:

```
package com.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;
```

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

    @Bean
    public GreetingService greetingService() {
        return new GreetingServiceImpl();
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new An
notationConfigApplicationContext(AppConfig.class);
        GreetingController controller = context.getBean(Gre
etingController.class);
        controller.sayHello();
        context.close();
    }
}
```

Section 9.3: Java Configuration

Introduction

Java configuration allows you to configure the Spring container using Java classes instead of XML files. This approach is type-safe and provides better tooling support.

Example: Java-based Configuration

Configuration Class:

```
package com.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

    @Bean
    public GreetingService greetingService() {
        return new GreetingServiceImpl();
    }
}
```

Main Application:


```
package com.example;

import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class SpringExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new An
notationConfigApplicationContext(AppConfig.class);
        GreetingController controller = context.getBean(Gre
etingController.class);
        controller.sayHello();
        context.close();
    }
}
```

Section 9.4: XML Configuration

Introduction

XML configuration is the traditional way of configuring the Spring container. It allows you to declare beans and their dependencies in an XML file.

Example: XML-based Configuration

XML Configuration:

```
<!-- applicationContext.xml -->
<beans xmlns="<http://www.springframework.org/schema/beans
>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instanc
e>"
       xsi:schemaLocation="<http://www.springframework.org/
schema/beans>
                           <http://www.springframework.org/
schema/beans/spring-beans.xsd>">

    <bean id="greetingService" class="com.example.GreetingS
erviceImpl"/>

    <bean id="greetingController" class="com.example.Greeti
ngController">
        <constructor-arg ref="greetingService"/>
    </bean>

</beans>
```

Service Interface:

```
package com.example;

public interface GreetingService {
    String greet();
}
```

Service Implementation:

```
package com.example;

public class GreetingServiceImpl implements GreetingService
{
    @Override
    public String greet() {
        return "Hello, World!";
    }
}
```

Controller:

```
package com.example;

public class GreetingController {

    private final GreetingService greetingService;

    public GreetingController(GreetingService greetingServi
ce) {
        this.greetingService = greetingService;
    }

    public void sayHello() {
        System.out.println(greetingService.greet());
    }
}
```

Main Application:

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;

public class SpringExample {
    public static void main(String[] args) {
```

```
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        GreetingController controller = context.getBean(Gre
etingController.class);
        controller.sayHello();
    }
}
```

Conclusion

In this chapter, we explored various ways to configure the Spring `ApplicationContext`, including autowiring, bootstrapping, and using both Java and XML configurations. These methods allow you to manage the lifecycle and dependencies of your beans effectively.

In the next chapter, we will discuss how to use `RestTemplate` for making RESTful web service calls, including setting headers, handling responses, and configuring authentication.

Chapter 10: RestTemplate

Section 10.1: Downloading a Large File

Explanation

RestTemplate is a synchronous client to perform HTTP requests. It simplifies communication with HTTP servers and enforces RESTful principles. When downloading large files, handling the response efficiently is crucial to avoid memory issues.

Example

Here's an example of how to download a large file using RestTemplate:

```
import org.springframework.core.io.Resource;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RequestCallback;
import org.springframework.web.client.ResponseExtractor;
import org.springframework.web.client.RestTemplate;
```

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

public class FileDownloadExample {

    public static void main(String[] args) {
        String url = "<https://example.com/largefile.zip>";
        RestTemplate restTemplate = new RestTemplate();

        ResponseExtractor<Void> responseExtractor = respons
e -> {
            try (InputStream inputStream = response.getBody
();
                 FileOutputStream outputStream = new FileOu


tputStream(new File("largefile.zip"))) {
                byte[] buffer = new byte[4096];
                int bytesRead;
                while ((bytesRead = inputStream.read(buffe
r)) != -1) {
                    outputStream.write(buffer, 0, bytesRea
d);
                }
            }

            return null;
        };

        restTemplate.execute(url, HttpMethod.GET, null, res
ponseExtractor);
    }
}
```

Section 10.2: Setting Headers on Spring RestTemplate Request

Explanation

Setting headers is essential for many operations, such as authentication, content-type specification, etc. RestTemplate allows you to set headers through the `HttpHeaders` class.

Example

Here's an example of setting headers on a RestTemplate request:

```
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

public class HeaderExample {

    public static void main(String[] args) {
        String url = "<https://example.com/resource>";
        RestTemplate restTemplate = new RestTemplate();

        HttpHeaders headers = new HttpHeaders();
        headers.set("Authorization", "Bearer some-token");

        HttpEntity<String> entity = new HttpEntity<>(header
s);
        ResponseEntity<String> response = restTemplate.exch
ange(url, HttpMethod.GET, entity, String.class);

        System.out.println(response.getBody());
    }
}
```

Section 10.3: Generics Results from Spring RestTemplate

Explanation

Handling generic types with RestTemplate can be tricky due to Java's type erasure. The `ParameterizedTypeReference` class can help manage this.

Example

Here's an example of handling generics results:

```
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

import java.util.List;
import java.util.Map;

public class GenericsExample {

    public static void main(String[] args) {
        String url = "<https://example.com/api/data>";
        RestTemplate restTemplate = new RestTemplate();

        ResponseEntity<List<Map<String, Object>>> response
= restTemplate.exchange(
                url,
                HttpMethod.GET,
                null,
                new ParameterizedTypeReference<List<Map<Str
ing, Object>>>() {
                });

        List<Map<String, Object>> data = response.getBody
();
        data.forEach(System.out::println);
    }
}
```

Section 10.4: Using Preemptive Basic Authentication with RestTemplate and HttpClient

Explanation

Preemptive Basic Authentication sends the authentication header with the initial request, reducing the need for an extra request-response round trip.

Example

Here’s an example using Apache HttpClient with RestTemplate for preemptive authentication:

```
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.impl.client.BasicCredentialsProvide
r;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.impl.client.ProxyAuthenticationStrat
egy;
import org.apache.http.impl.client.TargetAuthenticationStra
tegy;
import org.springframework.http.client.HttpComponentsClient
HttpRequestFactory;
import org.springframework.web.client.RestTemplate;

public class PreemptiveAuthExample {

    public static void main(String[] args) {
        String url = "<https://example.com/secure>";

        CredentialsProvider credsProvider = new BasicCreden
tialsProvider();
        credsProvider.setCredentials(AuthScope.ANY, new Use
rnamePasswordCredentials("user", "password"));

        CloseableHttpClient httpClient = HttpClients.custom
()
                .setDefaultCredentialsProvider(credsProvide
r)
                .setProxyAuthenticationStrategy(new ProxyAu
thenticationStrategy())
                .setTargetAuthenticationStrategy(new Target
AuthenticationStrategy())
                .build();
```

```
        HttpComponentsClientHttpRequestFactory requestFacto
ry = new HttpComponentsClientHttpRequestFactory(httpClien
t);
        RestTemplate restTemplate = new RestTemplate(reques
tFactory);

        String response = restTemplate.getForObject(url, St
ring.class);
        System.out.println(response);
    }
}
```

Section 10.5: Using Basic Authentication with HttpComponent's HttpClient

Explanation

Basic Authentication can also be set up using HttpComponents' HttpClient library, providing more control over the authentication process.

Example

Here's an example of setting up basic authentication using HttpComponents' HttpClient:

```
import org.apache.http.auth.AuthScope;
import org.apache.http.auth.UsernamePasswordCredentials;
import org.apache.http.client.CredentialsProvider;
import org.apache.http.impl.client.BasicCredentialsProvide
r;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.springframework.http.client.HttpComponentsClient
HttpRequestFactory;
import org.springframework.web.client.RestTemplate;

public class BasicAuthExample {

    public static void main(String[] args) {
```

```
        String url = "<https://example.com/secure>";

        CredentialsProvider credsProvider = new BasicCreden
tialsProvider();
        credsProvider.setCredentials(AuthScope.ANY, new Use
rnamePasswordCredentials("user", "password"));

        CloseableHttpClient httpClient = HttpClients.custom
()
                .setDefaultCredentialsProvider(credsProvide
r)
                .build();

        HttpComponentsClientHttpRequestFactory requestFacto
ry = new HttpComponentsClientHttpRequestFactory(httpClien
t);
        RestTemplate restTemplate = new RestTemplate(reques
tFactory);

        String response = restTemplate.getForObject(url, St
ring.class);
        System.out.println(response);
    }
}
```

This structure and content should provide a comprehensive guide for Chapter 10 on RestTemplate. Let me know if you need further details or adjustments!

Chapter 11: Task Execution and Scheduling

Task execution and scheduling are essential features in any application for running tasks asynchronously and periodically. Spring provides comprehensive support for both task execution and scheduling, making it easy to manage background tasks and scheduled operations.

Section 11.1: Enable Scheduling

Explanation

In Spring, task scheduling can be enabled using annotations or XML configuration. The `@EnableScheduling` annotation is used to enable Spring's scheduled task execution capability.

Example

1. **Maven Dependencies**

   Make sure you have the necessary dependencies in your `pom.xml` file:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.8</version>
</dependency>
```

2. **Configuration Class**

   Create a configuration class to enable scheduling:

```
import org.springframework.context.annotation.Configurat
ion;
import org.springframework.scheduling.annotation.EnableS
cheduling;

@Configuration
@EnableScheduling
public class SchedulingConfig {
}
```

3. **Scheduled Task**

   Create a component with a scheduled task:

```
import org.springframework.scheduling.annotation.Schedul
ed;
import org.springframework.stereotype.Component;

@Component
public class ScheduledTasks {
```

```
    @Scheduled(fixedRate = 5000)
    public void performTask() {
        System.out.println("Scheduled task executed at "
+ System.currentTimeMillis());
    }
}
```

Details

- `@Scheduled` annotation can be used with methods to define various scheduling options:
  - `fixedRate`: Executes the annotated method at a fixed interval.
  - `fixedDelay`: Executes the annotated method with a fixed delay between the end of the last execution and the start of the next.
  - `cron`: Executes the annotated method using a cron expression.

Section 11.2: Cron Expression

Explanation

Cron expressions are used to define complex time-based schedules. Spring supports cron expressions for scheduling tasks.

Example

1. **Scheduled Task with Cron Expression**

```
import org.springframework.scheduling.annotation.Schedul
ed;
import org.springframework.stereotype.Component;

@Component
public class CronScheduledTasks {

    @Scheduled(cron = "0 0/1 * * * ?")
    public void performCronTask() {
        System.out.println("Cron task executed at " + Sy
stem.currentTimeMillis());
```

```
    }
}
```

Details

- A cron expression consists of six fields representing seconds, minutes, hours, day of month, month, and day of week.
- Example: `0 0/1 * * * ?` - This expression means "every minute at the 0th second."

Section 11.3: Fixed Delay

Explanation

The `fixedDelay` attribute ensures that a task is executed at a fixed interval after the completion of the last execution.

Example

1. **Scheduled Task with Fixed Delay**

```
import org.springframework.scheduling.annotation.Schedul
ed;
import org.springframework.stereotype.Component;

@Component
public class FixedDelayScheduledTasks {

    @Scheduled(fixedDelay = 3000)
    public void performFixedDelayTask() {
        System.out.println("Fixed delay task executed at
" + System.currentTimeMillis());
    }
}
```

Details

- The task is executed 3000 milliseconds after the last execution has completed.

Section 11.4: Fixed Rate

Explanation

The `fixedRate` attribute ensures that a task is executed at a fixed interval regardless of the duration of the last execution.

Example

1. **Scheduled Task with Fixed Rate**

```
import org.springframework.scheduling.annotation.Schedul
ed;
import org.springframework.stereotype.Component;

@Component
public class FixedRateScheduledTasks {

    @Scheduled(fixedRate = 2000)
    public void performFixedRateTask() {
        System.out.println("Fixed rate task executed at
" + System.currentTimeMillis());
    }
}
```

Details

- The task is executed every 2000 milliseconds, even if the previous execution is still running.

Chapter 12: Spring Lazy Initialization

Section 12.1: Example of Lazy Init in Spring

Explanation

Lazy initialization is a technique where the initialization of a bean is deferred until it is first needed rather than during application startup. This can improve startup performance and resource usage. In Spring, beans are eagerly

initialized by default, but you can specify lazy initialization using annotations or XML configuration.

Example

Here's an example of using lazy initialization in Spring:

Java Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.context.annotation.Lazy;

@Configuration
public class AppConfig {

    @Bean
    @Lazy
    public MyService myService() {
        return new MyService();
    }
}

public class MyService {
    public MyService() {
        System.out.println("MyService initialized");
    }
}
```

In this example, `MyService` will only be instantiated when it is first requested from the application context.

XML Configuration

```
<beans xmlns="<http://www.springframework.org/schema/beans
>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instanc
e>"
       xsi:schemaLocation="<http://www.springframework.org/
```

```
schema/beans>
            <http://www.springframework.org/schema/beans/spr
ing-beans.xsd>">

    <bean id="myService" class="com.example.MyService" lazy
-init="true"/>
</beans>
```

Section 12.2: For Component Scanning and Auto-wiring

Explanation

Lazy initialization can also be applied to beans discovered through component scanning. By default, all beans found through component scanning are eagerly initialized, but this behavior can be modified.

Example

Here's an example of lazy initialization using component scanning and auto-wiring:

Java Configuration with Annotations

```
import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;

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

import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Service;

@Service
@Lazy
public class MyService {
    public MyService() {
```

```
        System.out.println("MyService initialized");
    }
}

import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Component;

@Component
public class MyComponent {

    private final MyService myService;

    @Autowired
    public MyComponent(MyService myService) {
        this.myService = myService;
    }

    public void doSomething() {
        System.out.println("Doing something");
        myService.toString();
    }
}
```

In this example, `MyService` is only instantiated when `doSomething()` is called on `MyComponent`.

Section 12.3: Lazy Initialization in the Configuration Class

Explanation

You can also set the default behavior for lazy initialization at the configuration level. This approach allows all beans within the context to be lazily initialized unless explicitly overridden.

Example

Here's an example of enabling lazy initialization for all beans in a configuration class:

Java Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.context.annotation.Lazy;

@Configuration
@Lazy
public class AppConfig {

    @Bean
    public MyService myService() {
        return new MyService();
    }

    @Bean
    public AnotherService anotherService() {
        return new AnotherService();
    }
}

public class MyService {
    public MyService() {
        System.out.println("MyService initialized");
    }
}

public class AnotherService {
    public AnotherService() {
        System.out.println("AnotherService initialized");
    }
}
```

In this configuration, both `MyService` and `AnotherService` will be lazily initialized.

XML Configuration

```
<beans xmlns="<http://www.springframework.org/schema/beans
>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instanc
e>"
       xsi:schemaLocation="<http://www.springframework.org/
schema/beans>
           <http://www.springframework.org/schema/beans/spr
ing-beans.xsd>"
       default-lazy-init="true">

    <bean id="myService" class="com.example.MyService"/>
    <bean id="anotherService" class="com.example.AnotherSer
vice"/>
</beans>
```

In this XML configuration, all beans will be lazily initialized unless specified otherwise.

This structure and content should provide a comprehensive guide for Chapter 12 on Spring Lazy Initialization. Let me know if you need further details or adjustments!

Chapter 13: Property Source

Property sources are essential in Spring for externalizing configuration. This allows you to manage and change configuration without altering the application code, thus making your application more flexible and easier to maintain.

Section 13.1: Sample XML configuration using PropertyPlaceholderConfigurer

Explanation

`PropertyPlaceholderConfigurer` is a legacy Spring bean for resolving placeholders in bean property values. It replaces placeholders with values from a properties file.

Example

1. **Properties File**

Create a properties file named `application.properties`:

```
app.name=My Spring Application
app.version=1.0.0
```

2. **Spring XML Configuration**

   Configure Spring to use `PropertyPlaceholderConfigurer`:

```
<beans xmlns="<http://www.springframework.org/schema/beans>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
       xsi:schemaLocation="<http://www.springframework.org/schema/beans>
           <http://www.springframework.org/schema/beans/spring-beans.xsd>">

    <bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:application.properties"/>
    </bean>

    <bean id="myBean" class="com.example.MyBean">
        <property name="name" value="${app.name}"/>
        <property name="version" value="${app.version}"/>
    </bean>
</beans>
```

3. **Java Bean**

   Create a Java bean to use the properties:

```
package com.example;

public class MyBean {
    private String name;
```

```
    private String version;

    // getters and setters

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

    public void setVersion(String version) {
        this.version = version;
    }

    @Override
    public String toString() {
        return "MyBean{name='" + name + "', version='" +
version + "'}";
    }
}
```

4. **Main Application**

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlA
pplicationContext;

public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApp
licationContext("applicationContext.xml");
        MyBean myBean = context.getBean(MyBean.class);
        System.out.println(myBean);
    }
}
```


Details

- `PropertyPlaceholderConfigurer` replaces placeholders with values from the specified properties file.
- The properties file is loaded from the classpath.

Section 13.2: Annotation

Explanation

The `@PropertySource` annotation is used in conjunction with Java-based configuration to load properties files into the Spring Environment.

Example

1. **Properties File**

   Create a properties file named `application.properties`:

```
app.name=My Spring Application
app.version=1.0.0
```

2. **Configuration Class**

   Create a configuration class to load the properties file:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configurat
ion;
import org.springframework.context.annotation.PropertySo
urce;
import org.springframework.core.env.Environment;
import org.springframework.beans.factory.annotation.Valu
e;
import org.springframework.beans.factory.config.Property
PlaceholderConfigurer;

@Configuration
@PropertySource("classpath:application.properties")
public class AppConfig {

    @Value("${app.name}")
    private String appName;
```

```
    @Value("${app.version}")
    private String appVersion;

    @Bean
    public MyBean myBean() {
        MyBean myBean = new MyBean();
        myBean.setName(appName);
        myBean.setVersion(appVersion);
        return myBean;
    }

    @Bean
    public static PropertyPlaceholderConfigurer property
PlaceholderConfigurer() {
        return new PropertyPlaceholderConfigurer();
    }
}
```

3. **Java Bean**

   Create a Java bean to use the properties:

```
package com.example;

public class MyBean {
    private String name;
    private String version;

    // getters and setters

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

    public void setVersion(String version) {
        this.version = version;
    }
```

```
    @Override
    public String toString() {
        return "MyBean{name='" + name + "', version='" +
version + "'}";
    }
}
```

4. **Main Application**

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;

public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfi
gApplicationContext(AppConfig.class);
        MyBean myBean = context.getBean(MyBean.class);
        System.out.println(myBean);
    }
}
```

Details

- `@PropertySource` annotation is used to specify the location of the properties file.
- The `@Value` annotation is used to inject property values into Spring beans.

Chapter 14: Dependency Injection (DI) and Inversion of Control (IoC)

Section 14.1: Autowiring a Dependency through Java Configuration

Explanation

Autowiring in Spring allows the Spring container to automatically resolve and inject collaborating beans into our bean. Java configuration provides a type- safe way to configure beans.

Example

Here's an example of autowiring a dependency through Java configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentSca
n;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.context.annotation.Scope;
import org.springframework.beans.factory.annotation.Autowir
ed;

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

    @Bean
    public ServiceA serviceA() {
        return new ServiceA();
    }

    @Bean
    public ServiceB serviceB() {
        return new ServiceB();
    }
}

public class ServiceA {
    public void perform() {
        System.out.println("ServiceA performing");
    }
}

public class ServiceB {
```

```
    private final ServiceA serviceA;

    @Autowired
    public ServiceB(ServiceA serviceA) {
        this.serviceA = serviceA;
    }

    public void execute() {
        serviceA.perform();
    }
}

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        ServiceB serviceB = context.getBean(ServiceB.clas
s);
        serviceB.execute();
    }
}
```

Section 14.2: Autowiring a Dependency through XML Configuration

Explanation

XML configuration allows us to define beans and their dependencies in an XML file. This method is particularly useful for legacy systems or when Java configuration is not preferred.

Example

Here's an example of autowiring a dependency through XML configuration:

```
<beans xmlns="<http://www.springframework.org/schema/beans
>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instanc
e>"
       xsi:schemaLocation="<http://www.springframework.org/
schema/beans>
           <http://www.springframework.org/schema/beans/spr
ing-beans.xsd>">

    <bean id="serviceA" class="com.example.ServiceA"/>
    <bean id="serviceB" class="com.example.ServiceB" autowi
re="byType"/>
</beans>
```

```
public class ServiceA {
    public void perform() {
        System.out.println("ServiceA performing");
    }
}

public class ServiceB {

    private ServiceA serviceA;

    public void setServiceA(ServiceA serviceA) {
        this.serviceA = serviceA;
    }

    public void execute() {
        serviceA.perform();
    }
}

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlAppl
icationContext;
```

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplic
ationContext("applicationContext.xml");
        ServiceB serviceB = context.getBean(ServiceB.clas
s);
        serviceB.execute();
    }
}
```

Section 14.3: Injecting a Dependency Manually through XML Configuration

Explanation

Manual dependency injection involves explicitly defining the dependencies in the configuration, which can provide more control over the dependency resolution process.

Example

Here's an example of manually injecting a dependency through XML configuration:

```
<beans xmlns="<http://www.springframework.org/schema/beans
>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instanc
e>"
       xsi:schemaLocation="<http://www.springframework.org/
schema/beans>
           <http://www.springframework.org/schema/beans/spr
ing-beans.xsd>">

    <bean id="serviceA" class="com.example.ServiceA"/>
    <bean id="serviceB" class="com.example.ServiceB">
        <property name="serviceA" ref="serviceA"/>
    </bean>
</beans>
```

Section 14.4: Injecting a Dependency Manually through Java Configuration

Explanation

Manual injection through Java configuration allows for programmatically controlling bean dependencies within a configuration class.

Example

Here's an example of manually injecting a dependency through Java configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;

@Configuration
public class AppConfig {

    @Bean
    public ServiceA serviceA() {
        return new ServiceA();
    }

    @Bean
    public ServiceB serviceB() {
        ServiceB serviceB = new ServiceB();
        serviceB.setServiceA(serviceA());
        return serviceB;
    }
}

public class ServiceA {
    public void perform() {
        System.out.println("ServiceA performing");
    }
}

public class ServiceB {
```

```
    private ServiceA serviceA;

    public void setServiceA(ServiceA serviceA) {
        this.serviceA = serviceA;
    }

    public void execute() {
        serviceA.perform();
    }
}

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigAp
plicationContext(AppConfig.class);
        ServiceB serviceB = context.getBean(ServiceB.clas
s);
        serviceB.execute();
    }
}
```

This structure and content should provide a comprehensive guide for Chapter 14 on Dependency Injection (DI) and Inversion of Control (IoC). Let me know if you need further details or adjustments!

Chapter 15: JdbcTemplate

JdbcTemplate is a central class in the Spring framework for working with relational databases. It simplifies database operations by reducing boilerplate code. This chapter covers various features and use cases of JdbcTemplate.

Section 15.1: Basic Query Methods

Explanation

JdbcTemplate provides several methods to execute SQL queries, update statements, and call stored procedures. Basic query methods include `queryForObject`, `queryForList`, and `query`.

Example

1. **Maven Dependencies**

   Make sure you have the necessary dependencies in your `pom.xml` file:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.3.8</version>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.200</version>
</dependency>
```

2. **Configuration Class**

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configurat
ion;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.datasource.DriverManager
DataSource;

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverM
anagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
```

```
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public JdbcTemplate jdbcTemplate() {
        return new JdbcTemplate(dataSource());
    }
}
```

3. **DAO Class**

   Create a DAO class to use JdbcTemplate:

```
import org.springframework.beans.factory.annotation.Auto
wired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public class UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public String getUserNameById(int id) {
        String sql = "SELECT name FROM users WHERE id =
?";
        return jdbcTemplate.queryForObject(sql, new Obje
ct[]{id}, String.class);
    }

    public List<String> getAllUserNames() {
        String sql = "SELECT name FROM users";
        return jdbcTemplate.queryForList(sql, String.cla
```

```
ss);
    }
}
```

4. **Main Application**

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;

public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfi
gApplicationContext(JdbcConfig.class);
        UserDao userDao = context.getBean(UserDao.clas
s);

        // Initialize database
        JdbcTemplate jdbcTemplate = context.getBean(Jdbc
Template.class);
        jdbcTemplate.execute("CREATE TABLE users (id IN
T, name VARCHAR(50))");
        jdbcTemplate.execute("INSERT INTO users (id, nam
e) VALUES (1, 'John Doe')");
        jdbcTemplate.execute("INSERT INTO users (id, nam
e) VALUES (2, 'Jane Doe')");

        // Fetch data
        System.out.println(userDao.getUserNameById(1));
// John Doe
        System.out.println(userDao.getAllUserNames());
// [John Doe, Jane Doe]
    }
}
```

Details

- `queryForObject` : Executes a query that returns a single object.

- `queryForList`: Executes a query that returns a list of objects.
- `query`: Executes a query that returns a list of rows, each row mapped to an object.

Section 15.2: Query for List of Maps

Explanation

JdbcTemplate allows you to execute queries that return a list of maps, where each map represents a row and the keys are the column names.

Example

1. **DAO Method**

```
import org.springframework.beans.factory.annotation.Auto
wired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Map;

@Repository
public class UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public List<Map<String, Object>> getAllUsers() {
        String sql = "SELECT * FROM users";
        return jdbcTemplate.queryForList(sql);
    }
}
```

2. **Main Application**

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;
```

```
public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfi
gApplicationContext(JdbcConfig.class);
        UserDao userDao = context.getBean(UserDao.clas
s);

        // Initialize database
        JdbcTemplate jdbcTemplate = context.getBean(Jdbc
Template.class);
        jdbcTemplate.execute("CREATE TABLE users (id IN
T, name VARCHAR(50))");
        jdbcTemplate.execute("INSERT INTO users (id, nam
e) VALUES (1, 'John Doe')");
        jdbcTemplate.execute("INSERT INTO users (id, nam
e) VALUES (2, 'Jane Doe')");

        // Fetch data
        List<Map<String, Object>> users = userDao.getAll
Users();
        for (Map<String, Object> user : users) {
            System.out.println(user);
        }
    }
}
```

Details

- `queryForList`: Executes a query that returns a list of maps, where each map represents a row of the result set.

Section 15.3: SQLRowSet

Explanation

The `SqlRowSet` interface provides a disconnected view of the result set, similar to `ResultSet` but without requiring a connection to remain open.

Example

1. **DAO Method**

```
import org.springframework.beans.factory.annotation.Auto
wired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.support.rowset.SqlRowSe
t;
import org.springframework.stereotype.Repository;

@Repository
public class UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public SqlRowSet getUserById(int id) {
        String sql = "SELECT * FROM users WHERE id = ?";
        return jdbcTemplate.queryForRowSet(sql, new Obje
ct[]{id});
    }
}
```

2. **Main Application**

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;
import org.springframework.jdbc.support.rowset.SqlRowSe
t;

public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfi
gApplicationContext(JdbcConfig.class);
        UserDao userDao = context.getBean(UserDao.clas
s);

        // Initialize database
```

```
        JdbcTemplate jdbcTemplate = context.getBean(Jdbc
Template.class);
        jdbcTemplate.execute("CREATE TABLE users (id IN
T, name VARCHAR(50))");
        jdbcTemplate.execute("INSERT INTO users (id, nam
e) VALUES (1, 'John Doe')");
        jdbcTemplate.execute("INSERT INTO users (id, nam
e) VALUES (2, 'Jane Doe')");

        // Fetch data
        SqlRowSet rowSet = userDao.getUserById(1);
        while (rowSet.next()) {
            System.out.println("ID: " + rowSet.getInt("i
d"));
            System.out.println("Name: " + rowSet.getStri
ng("name"));
        }
    }
}
```


Details

- `queryForRowSet`: Executes a query and returns the result as a `SqlRowSet`.

Section 15.4: Batch Operations

Explanation

JdbcTemplate supports batch operations, which allow you to execute a large number of statements efficiently.

Example

1. **DAO Method**

```
import org.springframework.beans.factory.annotation.Auto
wired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
```

```
@Repository
public class UserDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public void batchUpdateUsers(List<Object[]> users) {
        String sql = "INSERT INTO users (id, name) VALUE
S (?, ?)";
        jdbcTemplate.batchUpdate(sql, users);
    }
}
```

2. **Main Application**

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;

import java.util.Arrays;
import java.util.List;

public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfi
gApplicationContext(JdbcConfig.class);
        UserDao userDao = context.getBean(UserDao.clas
s);

        // Batch insert data
        List<Object[]> users = Arrays.asList(
            new Object[]{1, "John Doe"},
            new Object[]{2, "Jane Doe"},
            new Object[]{3, "Sam Smith"}
        );
        userDao.batchUpdateUsers(users);

        // Fetch data
```

```
        JdbcTemplate jdbcTemplate = context.getBean(Jdbc
Template.class);
        List<Map<String, Object>> result = jdbcTemplate.
queryForList("SELECT * FROM users");
        for (Map<String, Object> row : result) {
            System.out.println(row);
        }
    }
}
```

Details

- `batchUpdate`: Executes a batch of SQL statements.

Section 15.5: NamedParameterJdbcTemplate Extension of JdbcTemplate

Explanation

`NamedParameterJdbcTemplate` is an extension of `JdbcTemplate` that adds support for named parameters, improving code readability.

Example

1. **Configuration Class**

```
import org.springframework.context.annotation.Bean;
import org.springframework.context
```

.annotation.Configuration;
import
org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;

@Configuration
public class JdbcConfig {

```
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManag
erDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:testdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }

    @Bean
    public NamedParameterJdbcTemplate namedParameterJdbcTemp
late() {
        return new NamedParameterJdbcTemplate(dataSource());
    }
```

}

2. **DAO Class**

Create a DAO class to use NamedParameterJdbcTemplate:

```java
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.jdbc.core.namedparam.MapSqlParam
eterSource;
import org.springframework.jdbc.core.namedparam.NamedParame
terJdbcTemplate;
import org.springframework.stereotype.Repository;

import java.util.List;
import java.util.Map;

@Repository
public class UserDao {
```

```
    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTe
mplate;

    public List<Map<String, Object>> getUsersByName(String
name) {
        String sql = "SELECT * FROM users WHERE name = :nam
e";
        MapSqlParameterSource params = new MapSqlParameterS
ource();
        params.addValue("name", name);
        return namedParameterJdbcTemplate.queryForList(sql,
params);
    }
}
```

1. **Main Application**

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;

public class MainApplication {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfi
gApplicationContext(JdbcConfig.class);
        UserDao userDao = context.getBean(UserDao.clas
s);

        // Initialize database
        NamedParameterJdbcTemplate namedParameterJdbcTem
plate = context.getBean(NamedParameterJdbcTemplate.clas
s);
        namedParameterJdbcTemplate.getJdbcTemplate().exe
cute("CREATE TABLE users (id INT, name VARCHAR(50))");
        namedParameterJdbcTemplate.getJdbcTemplate().exe
cute("INSERT INTO users (id, name) VALUES (1, 'John Do
```

```
e')");
        namedParameterJdbcTemplate.getJdbcTemplate().exe
cute("INSERT INTO users (id, name) VALUES (2, 'Jane Do
e')");

        // Fetch data
        List<Map<String, Object>> users = userDao.getUse
rsByName("John Doe");
        for (Map<String, Object> user : users) {
            System.out.println(user);
        }
    }
}
```

Details

- `NamedParameterJdbcTemplate` supports named parameters, improving the readability and maintainability of SQL queries.
- `MapSqlParameterSource` is used to pass named parameters to the query.

Chapter 16: SOAP WS Consumption

Section 16.1: Consuming a SOAP WS with Basic Auth

Explanation

SOAP (Simple Object Access Protocol) is a protocol for exchanging structured information in the implementation of web services. Spring Web Services (Spring-WS) facilitates the creation of SOAP web services and clients. Consuming a SOAP web service with basic authentication involves setting up the necessary authentication headers and making the SOAP request.

Example

Here's an example of consuming a SOAP web service with basic authentication using Spring-WS:

Maven Dependencies

First, include the necessary dependencies in your `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-support</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.ws</groupId>
        <artifactId>spring-ws-test</artifactId>
    </dependency>
</dependencies>
```

Configuration Class

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.ws.client.core.WebServiceTemplat
e;
import org.springframework.ws.transport.http.HttpComponents
MessageSender;

@Configuration
public class SoapClientConfig {

    @Bean
    public WebServiceTemplate webServiceTemplate() {
        WebServiceTemplate webServiceTemplate = new WebServ
iceTemplate();
        webServiceTemplate.setMessageSender(httpComponentsM
```

```
essageSender());
        return webServiceTemplate;
    }

    @Bean
    public HttpComponentsMessageSender httpComponentsMessag
eSender() {
        HttpComponentsMessageSender messageSender = new Htt
pComponentsMessageSender();
        messageSender.setCredentialsProvider(credentialsPro
vider());
        return messageSender;
    }

    @Bean
    public org.apache.http.client.CredentialsProvider crede
ntialsProvider() {
        org.apache.http.impl.client.BasicCredentialsProvide
r provider = new org.apache.http.impl.client.BasicCredentia
lsProvider();
        provider.setCredentials(AuthScope.ANY, new Username
PasswordCredentials("user", "password"));
        return provider;
    }
}
```

SOAP Request and Response Classes

Create the necessary request and response classes based on the SOAP service's WSDL:

```
package com.example.soap;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "GetCountryRequest")
public class GetCountryRequest {
```

```
    private String name;

    @XmlElement(required = true)
    public String getName() {
        return name;
    }

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

@XmlRootElement(name = "GetCountryResponse")
public class GetCountryResponse {

    private Country country;

    @XmlElement(required = true)
    public Country getCountry() {
        return country;
    }

    public void setCountry(Country country) {
        this.country = country;
    }
}

public class Country {

    private String name;
    private String capital;
    private int population;

    // Getters and setters
}
```

SOAP Client

Create a client to consume the SOAP service:

```
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.stereotype.Component;
import org.springframework.ws.client.core.WebServiceTemplat
e;

@Component
public class SoapClient {

    private static final String URI = "<http://example.com/
ws>";

    @Autowired
    private WebServiceTemplate webServiceTemplate;

    public GetCountryResponse getCountry(String countryNam
e) {
        GetCountryRequest request = new GetCountryRequest
();
        request.setName(countryName);
        return (GetCountryResponse) webServiceTemplate.mars
halSendAndReceive(URI, request);
    }
}
```

Main Application

Use the client to make a request to the SOAP service:

```
import org.springframework.context.annotation.AnnotationCon
figApplicationContext;

public class Main {

    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new An
notationConfigApplicationContext(SoapClientConfig.class);
        SoapClient client = context.getBean(SoapClient.clas
```

```
s);

        GetCountryResponse response = client.getCountry("Sp
ain");
        System.out.println("Country: " + response.getCountr
y().getName());
        System.out.println("Capital: " + response.getCountr
y().getCapital());
        System.out.println("Population: " + response.getCou
ntry().getPopulation());

        context.close();
    }
}
```

Chapter 17: Spring Profile

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. This allows you to activate different beans or configurations based on the current environment (e.g., development, testing, production).

Section 17.1: Spring Profiles allows configuring parts available for certain environments

Explanation

Spring Profiles help in managing environment-specific configurations. You can define beans or configurations that should only be loaded in a specific environment. This is particularly useful for applications that need to run in different environments with different settings.

Example

1. **Maven Dependencies**

   Make sure you have the necessary dependencies in your `pom.xml` file:

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

```
    <version>5.3.8</version>
</dependency>
```

2. **Configuration Classes**

   Create configuration classes for different environments.

   Development Configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configurat
ion;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("dev")
public class DevConfig {

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverM
anagerDataSource();
        dataSource.setDriverClassName("org.h2.Driver");
        dataSource.setUrl("jdbc:h2:mem:devdb");
        dataSource.setUsername("sa");
        dataSource.setPassword("");
        return dataSource;
    }
}
```

   Production Configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configurat
ion;
import org.springframework.context.annotation.Profile;

@Configuration
@Profile("prod")
public class ProdConfig {
```

```
    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverM
anagerDataSource();
        dataSource.setDriverClassName("com.mysql.cj.jdb
c.Driver");
        dataSource.setUrl("jdbc:mysql://localhost:3306/p
roddb");
        dataSource.setUsername("root");
        dataSource.setPassword("password");
        return dataSource;
    }
}
```

3. **Main Configuration Class**

```
import org.springframework.context.annotation.ComponentS
can;
import org.springframework.context.annotation.Configurat
ion;
import org.springframework.context.annotation.PropertySo
urce;

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

4. **Java Bean**

   Create a Java bean to use the data source:

```
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Auto
wired;
import org.springframework.stereotype.Component;

@Component
```

```
public class MyService {

    private final DataSource dataSource;

    @Autowired
    public MyService(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void performDatabaseOperation() {
        System.out.println("Using data source: " + dataS
ource);
        // Perform database operations
    }
}
```

5. **Main Application**

```
import org.springframework.context.annotation.Annotation
ConfigApplicationContext;

public class MainApplication {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();
        context.getEnvironment().setActiveProfiles("de
v"); // Activate "dev" profile
        context.register(AppConfig.class);
        context.refresh();

        MyService myService = context.getBean(MyService.
class);
        myService.performDatabaseOperation();

        context.close();
    }
}
```

Details

- **Profiles**: Annotate configuration classes or beans with `@Profile` to specify the environment in which they should be active.

- **Activating Profiles**: Profiles can be activated programmatically (as shown above) or via environment variables or system properties (`-Dspring.profiles.active=dev`).
- **Environment-Specific Beans**: Beans defined in a profile-specific configuration will only be created when that profile is active.

Chapter 18: Understanding the `dispatcher-servlet.xml`

Section 18.1: `dispatcher-servlet.xml`

Explanation

The `dispatcher-servlet.xml` file is a core part of a Spring MVC application. It

defines beans and configurations specific to the Spring DispatcherServlet, which handles all incoming HTTP requests and routes them to appropriate handlers.

When a Spring MVC application is initialized, the `DispatcherServlet` searches for a configuration file named `[servlet-name]-servlet.xml` (in this case, `dispatcher-servlet.xml`). This file typically contains bean definitions for controllers, view resolvers, handler mappings, and other web components.

Example

Here's an example of a basic `dispatcher-servlet.xml` configuration file:

```
<beans xmlns="<http://www.springframework.org/schema/beans
>"
       xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instanc
e>"
       xmlns:context="<http://www.springframework.org/schem
a/context>"
       xmlns:mvc="<http://www.springframework.org/schema/mv
c>"
```

```
       xsi:schemaLocation="<http://www.springframework.org/
schema/beans>
           <http://www.springframework.org/schema/beans/spr
ing-beans.xsd>
           <http://www.springframework.org/schema/context>
           <http://www.springframework.org/schema/context/s
pring-context.xsd>
           <http://www.springframework.org/schema/mvc>
           <http://www.springframework.org/schema/mvc/sprin
g-mvc.xsd>">

    <!-- Enable component scanning -->
    <context:component-scan base-package="com.example.contr
oller"/>

    <!-- Enable Spring MVC configuration -->
    <mvc:annotation-driven/>

    <!-- View resolver to map logical view names to actual
view files -->
    <bean class="org.springframework.web.servlet.view.Inter
nalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

    <!-- Static resource handling -->
    <mvc:resources mapping="/resources/**" location="/resou
rces/"/>

</beans>
```

Breakdown

- `<context:component-scan base-package="com.example.controller"/>` : This enables component scanning, allowing Spring to automatically detect and register beans (like controllers) within the specified package.

- `<mvc:annotation-driven/>`: This activates Spring MVC's annotations, such as `@Controller`, `@RequestMapping`, etc.

- `<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">`: This bean defines a view resolver that maps logical view names to actual JSP files located under `/WEB-INF/views/`.
- `<mvc:resources mapping="/resources/**" location="/resources/"/>`: This configuration serves static resources from the `/resources/` directory.

Section 18.2: Dispatcher Servlet Configuration in `web.xml`

Explanation

The `web.xml` file is the deployment descriptor for a Java web application. It defines the servlets, filters, and other components that make up the application. For a Spring MVC application, you must configure the `DispatcherServlet` in `web.xml`.

Example

Here's an example of configuring the `DispatcherServlet` in `web.xml`:

```
<web-app xmlns="<http://java.sun.com/xml/ns/javaee>"
         xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"
          xsi:schemaLocation="<http://java.sun.com/xml/ns/javaee>
                              <http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd>"
         version="3.0">

    <!-- Spring DispatcherServlet -->
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
```

```
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>

    <!-- Character encoding filter -->
    <filter>
        <filter-name>characterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.Charac
terEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Breakdown

- `<servlet>`: Defines the `DispatcherServlet`, giving it the name `dispatcher` and specifying the class `org.springframework.web.servlet.DispatcherServlet`. The `contextConfigLocation` parameter points to the `dispatcher-servlet.xml` configuration file. The `load-on-startup` element ensures that the servlet is loaded and initialized when the web application starts.
- `<servlet-mapping>`: Maps the `DispatcherServlet` to the root URL pattern (`/`), meaning it will handle all incoming requests.

- `<filter>` and `<filter-mapping>`: These elements define a character encoding filter that ensures all requests and responses use UTF-8 encoding.

This structure and content provide a comprehensive guide for Chapter 18 on understanding the `dispatcher-servlet.xml` and configuring the Dispatcher Servlet in `web.xml`. Let me know if you need further details or adjustments!

Chapter 19: Advanced Spring MVC

Section 19.1: Exception Handling

Explanation

Exception handling in Spring MVC is a crucial part of building robust applications. Spring MVC provides various ways to handle exceptions in a clean and manageable way. Two primary methods include using `@ControllerAdvice` for global exception handling and `@ExceptionHandler` for specific exceptions within controllers.

Example: Global Exception Handling with `@ControllerAdvice`

`@ControllerAdvice` is a specialization of the `@Component` annotation which allows you to handle exceptions across the whole application in one global handling component.


```
import org.springframework.http.HttpStatus;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ControllerAd
vice;
import org.springframework.web.bind.annotation.ExceptionHan
dler;
import org.springframework.web.bind.annotation.ResponseStat
us;

@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(value = Exception.class)
    public String handleGenericException(Exception e, Model
model) {
```

```
        model.addAttribute("errorMessage", e.getMessage());
        return "error";
    }

    @ExceptionHandler(value = NullPointerException.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public String handleNullPointerException(NullPointerExc
eption e, Model model) {
        model.addAttribute("errorMessage", "A null pointer
exception occurred");
        return "error";
    }
}
```

In this example, `handleGenericException` will catch all exceptions, while `handleNullPointerException` will specifically handle `NullPointerException` instances.

Example: Using `@ExceptionHandler` for Specific Exceptions

You can also handle exceptions within specific controllers using the `@ExceptionHandler` annotation.

```
import org.springframework.web.bind.annotation.Controller;
import org.springframework.web.bind.annotation.ExceptionHan
dler;
import org.springframework.ui.Model;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStat
us;

@Controller
public class MyController {

    @ExceptionHandler(MyCustomException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public String handleMyCustomException(MyCustomException
e, Model model) {
        model.addAttribute("errorMessage", e.getMessage());
        return "customError";
```

```
    }
}
```

Here, `handleMyCustomException` will handle `MyCustomException` thrown within `MyController`.

Section 19.2: Interceptors

Explanation

Interceptors in Spring MVC allow you to pre-process and post-process requests. They can be used for various purposes, such as logging, authentication, and modifying request and response objects.

Example: Defining and Configuring Interceptors

First, create an interceptor by implementing the `HandlerInterceptor` interface or extending `HandlerInterceptorAdapter`.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class MyInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, Ht
tpServletResponse response, Object handler) throws Exceptio
n {
        // Pre-processing logic here
        System.out.println("Pre Handle method is Calling");
        return true; // Continue with the next interceptor
or the handler itself
    }

    @Override
    public void postHandle(HttpServletRequest request, Http
ServletResponse response, Object handler, ModelAndView mode
lAndView) throws Exception {
```

```
        // Post-processing logic here
        System.out.println("Post Handle method is Callin
g");
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception exc
eption) throws Exception {
        // After the complete request has finished
        System.out.println("Request and Response is complet
ed");
    }
}
```

Next, register the interceptor in your Spring MVC configuration.

```
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.web.servlet.config.annotation.In
terceptorRegistry;
import org.springframework.web.servlet.config.annotation.We
bMvcConfigurer;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registr
y) {
        registry.addInterceptor(new MyInterceptor()).addPat
hPatterns("/**");
    }
}
```

Section 19.3: Handling File Uploads

Explanation

Handling file uploads in Spring MVC is straightforward. You need to configure a multipart resolver and create a controller method to handle the file upload.

Example: Configuring File Uploads with Spring MVC

First, configure the multipart resolver in your `dispatcher-servlet.xml` or Java configuration class.

XML Configuration:

```
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
    <!-- max upload size in bytes -->
    <property name="maxUploadSize" value="10485760"/> <!-- 10MB -->
</bean>
```

Java Configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.multipart.commons.CommonsMultipartResolver;

@Configuration
public class AppConfig {

    @Bean
    public CommonsMultipartResolver multipartResolver() {
        CommonsMultipartResolver multipartResolver = new CommonsMultipartResolver();
        multipartResolver.setMaxUploadSize(10485760); // 10MB
        return multipartResolver;
    }
}
```

Example: Handling Multipart Requests

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestPara
m;
import org.springframework.web.multipart.MultipartFile;
import org.springframework.ui.Model;

import java.io.File;
import java.io.IOException;

@Controller
public class FileUploadController {

    @PostMapping("/upload")
    public String handleFileUpload(@RequestParam("file") Mu
ltipartFile file, Model model) {
        if (!file.isEmpty()) {
            try {
                // Save the file to a local directory
                String uploadDir = "/uploads/";
                file.transferTo(new File(uploadDir + file.g
etOriginalFilename()));
                model.addAttribute("message", "File uploade
d successfully!");
            } catch (IOException e) {
                model.addAttribute("message", "File upload
failed: " + e.getMessage());
            }
        } else {
            model.addAttribute("message", "File is empty");
        }
        return "uploadResult";
    }
}
```

HTML Form for File Upload:

```
<!DOCTYPE html>
<html>
<head>
    <title>File Upload</title>
</head>
<body>
    <h2>File Upload Form</h2>
    <form method="POST" enctype="multipart/form-data" actio
n="/upload">
        <input type="file" name="file"/>
        <input type="submit" value="Upload"/>
    </form>
    <p th:text="${message}"></p>
</body>
</html>
```

This comprehensive guide provides detailed explanations, examples, and code snippets for advanced Spring MVC features, including exception handling, interceptors, and file uploads.

Chapter 20: Spring Security

Section 20.1: Introduction to Spring Security

Explanation

Spring Security is a powerful and customizable framework for securing Java applications. It provides comprehensive support for authentication, authorization, and protection against common attacks.

- **Authentication** is the process of verifying the identity of a user.
- **Authorization** determines what resources and actions the authenticated user is allowed to access.

Spring Security integrates seamlessly with Spring applications, providing robust security capabilities with minimal configuration.

Basic Concepts of Authentication and Authorization

- **Authentication**: Verifies user identity using credentials (e.g., username and password).
- **Authorization**: Grants or denies access to resources based on user roles or permissions.
- **Principal**: Represents the authenticated user.
- **Granted Authority**: Represents permissions granted to the user.

Overview of Spring Security Architecture

- **SecurityContext**: Holds the authentication and authorization information.
- **Authentication**: Represents the token for an authenticated user.
- **UserDetailsService**: Loads user-specific data for authentication.
- **GrantedAuthority**: Represents an authority granted to the `Authentication` object.
- **Filters**: Intercept requests and perform security checks.

Section 20.2: Configuring Spring Security

Explanation

Spring Security can be configured using XML or Java-based configuration. The framework provides a flexible approach to secure web applications and method invocations.

Setting Up Spring Security in a Spring Application

Java Configuration Example:

```
import org.springframework.context.annotation.Configuratio
n;
import org.springframework.security.config.annotation.authe
ntication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.b
uilders.HttpSecurity;
import org.springframework.security.config.annotation.web.c
onfiguration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.c
onfiguration.WebSecurityConfigurerAdapter;
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAd
apter {

    @Override
    protected void configure(AuthenticationManagerBuilder a
uth) throws Exception {
        auth.inMemoryAuthentication()
            .withUser("user").password("{noop}password").ro
les("USER")
            .and()
            .withUser("admin").password("{noop}admin").role
s("ADMIN");
    }


    @Override
    protected void configure(HttpSecurity http) throws Exce
ption {
        http.authorizeRequests()
            .antMatchers("/admin/**").hasRole("ADMIN")
            .antMatchers("/user/**").hasRole("USER")
            .antMatchers("/", "/home").permitAll()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout()
            .permitAll();
    }
}
```

Explanation:

- `@EnableWebSecurity`: Enables Spring Security's web security support.
- `configure(AuthenticationManagerBuilder auth)`: Sets up in-memory authentication with two users.

- `configure(HttpSecurity http)`: Configures URL-based security and form-based login.

Section 20.3: Implementing Authentication

Explanation

Spring Security supports various authentication methods, including in-memory, JDBC-based, and custom user details service.

Using In-Memory and JDBC-Based Authentication

In-Memory Authentication Example:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user").password("{noop}password").roles
("USER")
        .and()
        .withUser("admin").password("{noop}admin").roles("A
DMIN");
}
```

JDBC-Based Authentication Example:

```
import javax.sql.DataSource;

@Autowired
private DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
throws Exception {
    auth.jdbcAuthentication()
        .dataSource(dataSource)
        .usersByUsernameQuery("select username, password, e
nabled from users where username=?")
        .authoritiesByUsernameQuery("select username, autho
```

```
rity from authorities where username=?");
}
```

Custom User Details Service

Custom UserDetailsService Example:

```
import org.springframework.security.core.userdetails.UserDe
tails;
import org.springframework.security.core.userdetails.UserDe
tailsService;
import org.springframework.security.core.userdetails.Userna
meNotFoundException;
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements UserDetail
sService {

    @Override
    public UserDetails loadUserByUsername(String username)
throws UsernameNotFoundException {
        // Load user from database
        return new CustomUserDetails(user);
    }
}

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAd
apter {

    @Autowired
    private CustomUserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder a
uth) throws Exception {
        auth.userDetailsService(userDetailsService);
```

```
    }
}
```

Section 20.4: Implementing Authorization

Explanation

Authorization in Spring Security can be role-based or permission-based. Annotations can be used to secure methods.

Role-Based and Permission-Based Authorization

Role-Based Authorization Example:

```
@Override
protected void configure(HttpSecurity http) throws Exceptio
n {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasRole("USER")
        .antMatchers("/", "/home").permitAll()
        .and()
        .formLogin()
        .loginPage("/login")
        .permitAll()
        .and()
        .logout()
        .permitAll();
}
```

Permission-Based Authorization Example:

```
@Override
protected void configure(HttpSecurity http) throws Exceptio
n {
    http.authorizeRequests()
        .antMatchers("/admin/**").hasAuthority("ADMIN_PRIVI
LEGE")
        .antMatchers("/user/**").hasAuthority("USER_PRIVILE
GE")
```

```
            .antMatchers("/", "/home").permitAll()
            .and()
            .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
            .logout()
            .permitAll();
    }
```

Using Annotations for Securing Methods

Using `@Secured` Annotation:

```
import org.springframework.security.access.annotation.Secur
ed;
import org.springframework.stereotype.Service;

@Service
public class MyService {

    @Secured("ROLE_ADMIN")
    public void adminMethod() {
        // Method logic here
    }

    @Secured("ROLE_USER")
    public void userMethod() {
        // Method logic here
    }
}
```

Using `@PreAuthorize` and `@PostAuthorize` Annotations:

```
import org.springframework.security.access.prepost.PreAutho
rize;
import org.springframework.stereotype.Service;

@Service
```

```
public class MyService {

    @PreAuthorize("hasRole('ADMIN')")
    public void adminMethod() {
        // Method logic here
    }

    @PreAuthorize("hasRole('USER')")
    public void userMethod() {
        // Method logic here
    }
}
```

This chapter provides a comprehensive guide on Spring Security, covering the basics of authentication and authorization, configuration, implementing different authentication methods, and securing applications using role-based and permission-based authorization.

Chapter 21: Spring Cloud

Section 21.1: Introduction to Spring Cloud

Explanation

Spring Cloud provides a suite of tools to manage the complexity of distributed systems and microservice architectures. It builds on Spring Boot to enable the rapid creation of robust, scalable microservices.

Overview of Microservices Architecture

- **Microservices**: A style of software architecture that involves developing single-function modules with well-defined interfaces and operations. These modules can be independently deployed and scaled.
- **Benefits**: Improved modularity, easier deployment, scalability, and resilience.
- **Challenges**: Complexity in managing inter-service communication, service discovery, load balancing, fault tolerance, and monitoring.

Key Components of Spring Cloud

- **Spring Cloud Netflix**: Provides integrations with Netflix OSS libraries like Eureka, Hystrix, and Zuul.
- **Spring Cloud Config**: Manages external configuration for applications across all environments.
- **Spring Cloud Gateway**: Provides a library for building API gateways.
- **Spring Cloud Sleuth**: Adds distributed tracing capabilities to your application.

Section 21.2: Service Discovery with Eureka

Explanation

Eureka, part of the Netflix OSS suite, is a service registry that helps in service discovery. It allows services to find and communicate with each other without hardcoding hostname and port.

Setting Up a Eureka Server

pom.xml Dependencies:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server
</artifactId>
</dependency>
```

Spring Boot Application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.cloud.netflix.eureka.server.Enab
leEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.clas
```

```
s, args);
    }
}
```

application.yml Configuration:

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
  server:
    wait-time-in-ms-when-sync-empty: 0
```

Registering Services with Eureka

pom.xml Dependencies for Client Service:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client
</artifactId>
</dependency>
```

Spring Boot Application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.cloud.netflix.eureka.EnableEurek
aClient;

@SpringBootApplication
@EnableEurekaClient
public class EurekaClientApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaClientApplication.clas
```

```
s, args);
    }
}
```

application.yml Configuration:

```
server:
  port: 8080

eureka:
  client:
    service-url:
      defaultZone: <http://localhost:8761/eureka/>
```

Section 21.3: Circuit Breaker with Hystrix

Explanation

Hystrix, from the Netflix OSS suite, implements the circuit breaker pattern to handle service failures gracefully. It helps in building resilient applications.

Implementing Circuit Breakers

pom.xml Dependencies:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artif
actId>
</dependency>
```

Spring Boot Application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.cloud.client.circuitbreaker.Enab
leCircuitBreaker;

@SpringBootApplication
```

```
@EnableCircuitBreaker
public class HystrixApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixApplication.class, arg
s);
    }
}
```

Example Service with Circuit Breaker:

```
import com.netflix.hystrix.contrib.javanica.annotation.Hyst
rixCommand;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

@Service
public class MyService {

    private final RestTemplate restTemplate;

    public MyService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    @HystrixCommand(fallbackMethod = "fallbackMethod")
    public String callExternalService() {
        return restTemplate.getForObject("<http://external-
service/api>", String.class);
    }

    public String fallbackMethod() {
        return "Fallback response";
    }
}
```

Monitoring Circuit Breakers with Hystrix Dashboard

pom.xml Dependencies:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashbo
ard</artifactId>
</dependency>
```

Spring Boot Application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.cloud.netflix.hystrix.dashboard.
EnableHystrixDashboard;

@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {
    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardApplication.c
lass, args);
    }
}
```

application.yml Configuration:

```
server:
  port: 8081
```

Accessing Dashboard:
Navigate to
`http://localhost:8081/hystrix` to view the dashboard.

Section 21.4: API Gateway with Zuul

Explanation

Zuul, from the Netflix OSS suite, acts as an API gateway that provides dynamic routing, monitoring, resiliency, and security.

Setting Up Zuul as an API Gateway

pom.xml Dependencies:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifact
Id>
</dependency>
```

Spring Boot Application:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApp
lication;
import org.springframework.cloud.netflix.zuul.EnableZuulPro
xy;

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

application.yml Configuration:

```
server:
  port: 8080

zuul:
  routes:
    users-service:
      path: /users/**
      url: <http://localhost:8081>
    orders-service:
```

```
      path: /orders/**
      url: <http://localhost:8082>
```

Routing and Filtering Requests

Pre-Filter Example:

```
import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import org.springframework.stereotype.Component;

@Component
public class PreFilter extends ZuulFilter {

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 1;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() throws ZuulException {
        RequestContext ctx = RequestContext.getCurrentConte
xt();
        System.out.println("Request Method : " + ctx.getReq
uest().getMethod());
        return null;
```

```
    }
}
```

This chapter provides a comprehensive guide on Spring Cloud, covering the introduction to Spring Cloud, service discovery with Eureka, implementing circuit breakers with Hystrix, and setting up an API gateway with Zuul.

Chapter 22: Testing in Spring

Section 22.1: Unit Testing with Spring

Explanation

Unit testing focuses on testing individual components or units of an application in isolation. In Spring applications, unit tests are crucial for ensuring that each part of the application behaves as expected.

Writing Unit Tests for Spring Components

Unit tests for Spring components typically involve testing service, repository, or controller classes.

Example Service Class:

```
@Service
public class CalculatorService {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Unit Test for Service Class:

```
import static org.junit.jupiter.api.Assertions.assertEqual
s;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
```

```
public class CalculatorServiceTest {

    @InjectMocks
    private CalculatorService calculatorService;

    @Test
    public void testAdd() {
        int result = calculatorService.add(10, 20);
        assertEquals(30, result);
    }
}
```

Explanation:

- `@InjectMocks` injects dependencies into the test class.
- `@ExtendWith(MockitoExtension.class)` integrates Mockito with JUnit 5.
- The `testAdd` method verifies that the `add` method of `CalculatorService` correctly adds two numbers.

Using Mockito for Mocking Dependencies

Mockito is a popular mocking framework that simplifies the creation of mock objects for testing.

Mockito Example:

```
import static org.mockito.Mockito.*;

@SpringBootTest
public class UserControllerTest {

    @Mock
    private UserRepository userRepository;

    @InjectMocks
    private UserController userController;

    @Test
    public void testFindUserById() {
        User mockUser = new User("1", "John Doe");
```

```
        when(userRepository.findById("1")).thenReturn(Optio
nal.of(mockUser));

        ResponseEntity<User> response = userController.getU
ser("1");

        assertEquals(HttpStatus.OK, response.getStatusCode
());
        assertEquals("John Doe", response.getBody().getName
());
    }
}
```

Section 22.2: Integration Testing

Explanation

Integration testing verifies the interaction between different components of an application to ensure they function together correctly.

Setting Up Integration Tests with Spring Test

Spring provides robust support for integration testing using annotations like `@SpringBootTest`.

Example Integration Test:

```
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.boot.test.context.SpringBootTes
t;
import org.springframework.boot.test.web.client.TestRestTem
plate;
import org.springframework.boot.web.server.LocalServerPort;

import static org.junit.jupiter.api.Assertions.assertEqual
s;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironm
```

```
ent.RANDOM_PORT)
public class IntegrationTestExample {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void testHomePage() {
        String url = "<http://localhost>:" + port + "/";
        String response = restTemplate.getForObject(url, St
ring.class);
        assertEquals("Hello, World!", response);
    }
}
```

Explanation:

- `@SpringBootTest` launches the Spring context and loads the application.
- `@LocalServerPort` injects the random port number assigned by Spring Boot.
- `TestRestTemplate` is used to perform HTTP requests against the running application.

Section 22.3: Testing REST APIs

Explanation

Testing REST APIs ensures that the endpoints correctly handle requests and return the expected responses.

Writing Tests for REST Controllers

Spring provides `MockMvc` for testing REST controllers without starting a full HTTP server.

Example REST Controller Test:

```
import static org.springframework.test.web.servlet.request.
MockMvcRequestBuilders.*;
```

```
import static org.springframework.test.web.servlet.result.M
ockMvcResultMatchers.*;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowir
ed;
import org.springframework.boot.test.autoconfigure.web.serv
let.AutoConfigureMockMvc;
import org.springframework.boot.test.context.SpringBootTes
t;
import org.springframework.test.web.servlet.MockMvc;

@SpringBootTest
@AutoConfigureMockMvc
public class UserControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testGetUser() throws Exception {
        mockMvc.perform(get("/users/1"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id").value("1"))
            .andExpect(jsonPath("$.name").value("John Do
e"));
    }
}
```

Explanation:

- `@AutoConfigureMockMvc` sets up the `MockMvc` instance.
- `mockMvc.perform(get("/users/1"))` sends a GET request to `/users/1`.
- `andExpect(status().isOk())` verifies that the response status is OK (200).
- `andExpect(jsonPath("$.id").value("1"))` verifies the response JSON contains the expected values.

This chapter provides a comprehensive guide on testing in Spring, covering unit testing with Spring, integration testing, and testing REST APIs using

MockMvc.

Chapter 23: Spring and JPA/Hibernate

Section 23.1: Introduction to Spring Data JPA

Explanation

Spring Data JPA simplifies the implementation of data access layers by providing a higher-level abstraction over JPA (Java Persistence API) and Hibernate.

Overview of JPA and Hibernate

- **JPA**: Java Persistence API is a specification for managing relational data in Java applications.
- **Hibernate**: A popular JPA implementation that provides object-relational mapping capabilities.

Benefits of Using Spring Data JPA

- **Reduces Boilerplate Code**: Spring Data JPA reduces the amount of boilerplate code required to implement data access layers.
- **Automatic Query Generation**: It can automatically generate queries from method names.
- **Integration with Spring Framework**: Seamlessly integrates with Spring's dependency injection and transaction management.

Section 23.2: Setting up Spring Data JPA

Explanation

Setting up Spring Data JPA involves configuring the data source and defining repositories that interact with the database.

Configuring a Data Source

application.properties Configuration:

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabas
e
```

```
spring.datasource.username=root
spring.datasource.password=password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Drive
r
spring.jpa.properties.hibernate.dialect=org.hibernate.diale
ct.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=update
```

Defining JPA Repositories

Example Repository Interface:

```
import org.springframework.data.jpa.repository.JpaRepositor
y;

public interface UserRepository extends JpaRepository<User,
Long> {
    User findByUsername(String username);
}
```

Section 23.3: Query Methods

Explanation

Spring Data JPA allows defining queries using method names or custom queries using `@Query` annotations.

Using Method Names to Define Queries

```
public interface UserRepository extends JpaRepository<User,
Long> {
    User findByUsername(String username);
    List<User> findByAgeGreaterThan(int age);
}
```

Writing Custom Queries with `@Query`

```
import org.springframework.data.jpa.repository.Query;
import java.util.List;

public interface UserRepository extends JpaRepository<User,
Long> {
    @Query("SELECT u FROM User u WHERE u.age >= :age")
    List<User> findUsersWithAgeGreaterThanOrEqual(@Param("a
ge") int age);
}
```

Section 23.4: Transactions

Explanation

Transactions ensure that database operations are atomic, consistent, isolated, and durable (ACID properties).

Managing Transactions with Spring

Using `@Transactional` Annotation:

```
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transacti
onal;

@Service
public class UserService {

    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    @Transactional
    public void updateUser(User user) {
        userRepository.save(user);
```

```
        }
    }
```

Example of Transactional Behavior:

- **Atomicity**: Ensures that all operations within a transaction complete successfully or none at all.
- **Consistency**: Ensures that the database remains in a consistent state before and after the transaction.
- **Isolation**: Ensures that the intermediate state of a transaction is invisible to other transactions.
- **Durability**: Ensures that committed data is persisted even in the event of a system failure.

This chapter provides a comprehensive guide on using Spring Data JPA with Hibernate, covering its introduction, setup, query methods, and transaction management. Let me know if you need further details or adjustments!