Adavanced Java

Agenda

- Spring Expression Language
- Auto-wiring
- Spring Mail
- Spring Bean scopes
- Spring-JDBC Integration

Spring Expression Language

- SpEL is a powerful expression language that supports querying and manipulating an object graph at runtime.
- Syntactically it is similar to JSP EL ${expression}.
- SpEL can be used in all spring framework components/products.
- SpEL supports Literal expressions, Regular expressions, Class expressions, Accessing properties, Collections, Method invocation, Relational operators, Assignment, Bean references, Inline lists/maps, Ternary operator, etc.
- SpEL expressions are internally evaluated using SpELExpressionParser.

```
ExpressionParser parser = new SpELExpressionParser();
value = parser.parseExpression("'Hello World'.concat('!')");
value = parser.parseExpression("new String('Hello World').toUpperCase()");
value = parser.parseExpression("bean.list[0]");
```

- SpEL expressions are slower in execution due parsing. Spring 4.1 added SpEL compiler to speed-up execution by creating a class for expression behaviour at runtime.
- https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#expressions

Using XML config

- Used in bean configuration XML file.
  - value="#{bean.field}"
  - value="#{bean.method()}"
  - value="${property-key}"
    - In beans.xml, `<context:property-placeholder location="classpath:app.properties" />`

Using Java config

- Used with @Value annotation on field or setter.
  - Auto-wire dependency beans: @Value("#{bean}")
  - Values using SPEL expressions: @Value("#{bean.field}") or @Value("#{bean.method()}")
  - Values using SPEL expressions from properties files: @Value("${property-key}")
    - Spring Boot by default gets property values from application.properties.
    - In @Configuration class, use @PropertySource("classpath:app.properties") or @PropertySource("file:///path/to/app.properties") to get values from other files.

```
# application.properties
db.url = jdbc:mysql://localhost:3306/test
db.driver = com.mysql.cj.jdbc.Driver
db.user = root
db.password = root
```

```
@Configuration
class DbConfig {
    @Value("${db.url}")
    private String dbUrl;
    @Value("${db.driver}")
    private String dbDriver;
    @Value("${db.user}")
    private String dbUser;
    @Value("${db.password}")
```

```
    private String dbPassword;
    // other methods
}
```

@Value to initialize beans (sb04_stereoann continued)

- step 14. Create file loginfo.properties in your disk e.g. D:/loginfo.properties (not under the project).
  - logFile = bank.log
- step 15. Use @PropertySource("file:///D:/loginfo.properties") in @Configuration class and @Value("${logFile}") in FileLoggerImpl to initialize logFilePath field.
- step 16. Attach file logger into AccountImpl class using @Value("#{fileLoggerImpl}") -- instead of setting it from main class acc.setLogger().
- step 17. Execute the application and check if it producing desired output.
- step 18. Create @Component TestSpEL with fields accId, accBalance and acc. Initialize them using @Value. Add printInfo() method to display the field values.
  - @Value("#{acc.id}") private int accId;
  - @Value("#{acc.getBalance()}") private double accBalance;
  - @Value("#{acc}") private Account acc;
- step 19. In main(), get the object of TestSpEL bean and invoke printInfo() method.

application.properties

- Spring Boot will automatically find and load application.properties and application.yaml (https://youtu.be/pt0ovzQhqFc) files.
  - From the classpath
    - The classpath root
    - The classpath /config package
  - From the current directory
    - The current directory
    - The /config subdirectory in the current directory
    - Immediate child directories of the /config subdirectory
- Properties file name can be switched from application.properties to other using spring.config.name.
  - java -jar myproject.jar --spring.config.name=app

- The default locations of application.properties can be changed using spring.config.location.
    - java -jar myproject.jar --spring.config.location=optional:classpath:/default.properties,optional:classpath:/override.properties
- A properties file can import another property file.

```
spring.config.import=file:/etc/config/myconfig.properties
```

- A property file may have property-placeholder.

```
s1.name=${random.value}
s1.marks=${random.int}
p1.name=Nilesh
a1.accHolder.name=${p1.name}
```

```
# below are environment variables
spring.datasource.url = ${MYSQL_DB_URL}
spring.datasource.username = ${MYSQL_DB_USERNAME}
spring.datasource.password = ${MYSQL_DB_PASSWORD}
```

Auto-wiring

- Auto-detection of dependencies and injecting them into dependent objects is also referred as "Auto-wiring".
- Auto-wiring can be done in XML config using `<bean id="..." class="..." autowire="default|no|byType|byName|constructor" .../>`.
    - default/no = By default auto-wiring is disabled.
    - byType = auto wire(connect) the bean with matching type (when single bean of matching type is available).
        - e.g. field: ConsoleLoggerImpl logger; --> auto connected to bean of type ConsoleLoggerImpl.
    - byName = auto wire(connect) the bean with matching name.
        - e.g. field: Logger logger; --> auto connected to bean with name "logger" i.e. `<bean id="logger" class="pkg.DbLoggerImpl">`

- constructor = auto wire(connect) the bean into constructor of dependent bean.

```
class AccountImpl {
    // ...
    private Logger logger;
    public AccountImpl(ConsoleLoggerImpl cLogger) {
        this.logger = cLogger;
    }
}
```

- Auto-wiring is done into Java config using spring annotation @Autowired.
- @Autowired can be used at
  - Setter level: setter based DI
  - Constructor level: constructor based DI
  - Field level: field based DI
- Constructor based @Autowired DI is preferred over Setter based DI & Field based DI.
- From Spring 5, single argument constructors have @Autowired implicitly.

@Autowired resolution process

- @Autowired finds bean of corresponding field "type" and assign it.
- If no bean is found of given type, it throws exception.
  - @Autowired(required=false): no exception is thrown, auto-wiring skipped.
- If multiple beans are found of given type, it try to attach bean of same name. If no such bean is found, then throw exception.
- If multiple beans are found of given type, programmer can use @Qualifier to choose expected bean. @Qualifier can only be used to resolve conflict in case of @Autowired.
- If multiple beans are found of given type, one of the bean can be declared as @Primary. If no @Qualifier is mentioned, then @Primary bean will be attached (and no exception is produced).
- @Autowired --> byType, byQualifier (@Qualifier), byName.

@Resource (JSR 250)

- @Resource resolution: DI byName, byType, byQualifier

@Inject (JSR 330) – same as @Autowired

- @Inject: DI byType, byQualifier (@Named), byName

Auto-wiring (sb05_autowiring)

- step 1: Spring Starter Project
    - Fill group id, artifact id, Spring Boot=3.x.y, Language=Java, Java version=17, packaging=Jar -- click Finish
    - Modify pom.xml and update Maven project.
        - `<version>2.7.18</version>`
        - `<java.version>11</java.version>`
- step 2. Create NotificationServiceClient class with notifyUser() method. Display "Hello, User" message in it.
- step 3. In Main class, declare a field of NotificationServiceClient. How to access NotificationServiceClient bean in Main class?
    - @Autowired on field
    - @Autowired on setter
    - @Autowired on constructor (not suitable for Main class)
- step 4. Make main class as CommandLineRunner and call NotificationServiceClient's notifyUser() method.
- step 5. Create interface NotificationService with abstract method `void sendNotification(String to, String message);`
- step 6. How to attach NotificationService as field in NotificationServiceClient?
    - @Autowired on field
    - @Autowired on setter
    - @Autowired on constructor
- step 6 (contd). @Autowired NotificationService service field in NotificationServiceClient class. From notifyUser() method, call service.sendNotification(...); if service is non-null.

- step 7. Execute the program and observe the error.
- step 8. Make NotificationService dependency optional @Autowired(required=false). Execute the program and test the output.
- step 9. Implement NotificationService interface into two @Component i.e. MockSmsServiceImpl and MockEmailServiceImpl. Both prints message on console.
- step 10. Execute the program and observe the error.
- step 11. Make one of the implementation as @Primary. Execute the program and test the output. Comment @Primary on implementation.
- step 12. Name one of the implementation with same name as NotificationService field in NotificationServiceClient using @Component(value="name") on NotificationService implementation class. Execute the program and test the output. Comment naming of the bean.
- step 13. Attach one of the NotificationService implementation using @Qualifier with @Autowired. Execute the program and test the output. Comment @Qualifier on implementation.
- step 14. Add mail starter in pom.xml file.

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

- step 15. Implement NotificationService interface into SmtpEmailServiceImpl. @Autowired javaMailSender in it. Provide necessary SMTP config into application.properties.

```
spring.mail.host = smtp.gmail.com
spring.mail.username = youremail@gmail.com
spring.mail.password = your_application_password

spring.mail.properties.mail.smtp.auth = true
spring.mail.properties.mail.smtp.socketFactory.port = 465
spring.mail.properties.mail.smtp.socketFactory.class = javax.net.ssl.SSLSocketFactory
```

```
spring.mail.properties.mail.smtp.socketFactory.fallback = false
spring.mail.properties.mail.smtp.ssl.enable = true
```

```
MimeMessage message = javaMailSender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setSubject(subject);
helper.setTo(to);
helper.setText(body, true);
javaMailSender.send(message);
```

- step 17. Make SmtpEmailServiceImpl as @Primary. Execute the program and test the output. Comment @Primary on SmtpEmailServiceImpl.
- resources:
  - In Google account settings, Under "How you sign in to Google," select 2-Step Verification.
  - https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-email
  - https://www.google.com/landing/2step/
  - https://support.google.com/accounts/answer/185833
  - https://www.developer.com/java/other/working-with-javamail-and-the-spring-mail-apis.html
- Information on Spring email classes/interfaces
  - MailSender: This is an interface that defines methods for sending simple e-mails, such as simple text e-mails. To incorporate a MIME message, there is an another interface, called JavaMailSender, which is an extension of this interface.
  - MailMessage: This is a common interface for all e-mail messages and designates irrespective of the types of message such as a simple test message or a more complex MIME message.
  - MailException: Base exception class for all exception thrown by mail.
  - SimpleMailMessage: This refers to a class that represents a simple message and includes data such as from, to, cc, subject, and text message contents.
  - JavaMailSenderImpl: This is the core class to send simple mail as well as MIME messages. It is an implementation of the MailSender interface.
  - MimeMailMessage: This is an implementation of the MailMessage interface for JavaMail MIME messages.

- MimeMessageHelper: This is a helper class for populating MIME messages. For example, it supports HTML text content, inline content such as images, typical mail attachments, and so forth.

Spring Bean Scopes

- Using a bean definition (@Bean or Stereo-type annotations + Other config), one or more bean objects can be created.
- The bean scope can be set in XML or annotation.
  - `<bean id="___" class="___" scope="singleton|prototype|request|session" />`
  - @Scope("singleton|prototype|request|session") on @Bean / @Component ...
- Spring 5 supports six different bean scopes. Four scopes are valid only if you use a web-aware ApplicationContext.

Scope	Description
singleton	(Default) A single object instance for each Spring IoC container.
prototype	Any number of object instances (new instance for each access).
request	A single object for current HTTP request.
session	A single object for current HTTP session.
application	A single object for current application i.e. ServletContext.
websocket	A single object for current websocket.

Singleton

- Single bean object is created and accessed throughout the application.
- BeanFactory creates object when getBean() is called for first time for that bean.
- All singleton bean objects are created when ApplicationContext is created.
- For each sub-sequent call to getBean() returns same object reference.
- Reference of all singleton beans is managed by spring container.
- During shutdown, all singleton beans are destroyed (@PreDestroy will be called).

Singleton class vs Singleton bean

- Design patterns are solutions to the well-known problems.
- Singleton is a design pattern. It enable access to a single object throughout the application.
- In OOP languages (like Java), Singleton class is created so that only one object of the class is available for the use in the whole application.

```
public class Singleton {
    // since ctor is private, object of class cannot be created outside this class.
    private Singleton() {
    }

    private static Singleton obj;
    static {
        // create obj of class within class itself
        obj = new Singleton();
        // further obj initialization
    }

    // return object outside the class.
    public static Singleton getInstance() {
        return obj;
    }

    // fields and methods
}
```

```
// client code
Singleton obj1 = Singleton.getInstance();
Singleton obj2 = Singleton.getInstance(); // same obj will return
```

- In Spring, singleton is scope of spring bean. Same bean object (identified by the same id) will be accessible in the whole application. However spring bean class is a simple Java class and hence multiple objects of that bean class possible.

```
@Configuration
public class AppConfig {
    @Scope("singleton")
    @Bean
    public Box b1() {
        return new BoxImpl(...);
    }

    @Scope("singleton")
    @Bean
    public Box b2() {
        return new BoxImpl(...);
    }
}
```

- BoxImpl is not singleton class. There are two objects created in b1() and b2() method.
- However b1 and b2 bean definations are singleton. In the whole application each time `ctx.getBean("b1")` is called, it will return same object. Similar for "b2" as well.

Prototype

- No bean is created during startup.
- Reference of bean is not maintained by ApplicationContext.
- Beans are not destroyed automatically during shutdown.
- Bean object is created each time ctx.getBean() is called.

Dependency with different scopes

Singleton bean inside prototype bean

- Single singleton bean object is created.
- Each call to getBean() create new prototype bean. But same singleton bean is autowired in them.

- e.g. OrderImpl <>--- RestaurantImpl

Prototype bean inside singleton bean

- Single singleton bean object is created.
- While auto-wiring singleton bean, prototype bean is created and is injected in singleton bean.
- Since there is single singleton bean, there is a single prototype bean.
- e.g. Outer1 <>--- Inner1

Need multiple prototype beans from singleton bean?

1. Using ApplicationContext

- The singleton bean class can be inherited from ApplicationContextAware interface or @Autowired ApplicationContext.
- This ApplicationContext can be used to create new prototype bean each time (as per requirement).
- e.g. Outer2 <>--- Inner2

2. Using @Lookup method

- The singleton bean class contains method returning prototype bean.
- If method is annotated with @Lookup, each call to the method will internally call ctx.getBean(). Hence for inner prototype beans, it returns new bean each time.
- e.g. Outer3 <>--- Inner3

Bean Scopes (sb06_beanscopes)

- step 1: Spring Starter Project
    - Fill group id, artifact id, Spring Boot=3.x.y, Language=Java, Java version=17, packaging=Jar -- click Finish
    - Modify pom.xml and update Maven project.
        - `<version>2.7.18</version>`
        - `<java.version>11</java.version>`

- step 2. Implement Box interface and BoxImpl class. In BoxImpl, also implement BeanNameAware interface and a @PostConstruct method.
- step 3. In AppConfig create @Bean b1 as Singleton and @Bean b2 as Prototype.
- step 4. In main() try creating multiple objects of the beans (using ctx.getBean()). Execute the program and observe the output.
    - "b1" bean is created at the start of application. The getBean() returns same reference.
    - "b2" bean is created each time getBean() is called. Obviously returns different reference for each call.
- step 5. Mark both the beans as @Lazy. Execute the program and observe the output.
    - "b1" bean is created when first getBean() is called. The getBean() returns same reference for subsequent calls.
    - "b2" bean is created each time getBean() is called. Obviously returns different reference for each call.
- step 6: Implement Inner1 as Singleton bean and Outer1 as Prototype bean. Autowire Inner1 bean inside Outer1.
- step 7: Create multiple Outer1 beans in main() using (ctx.getBean()) and try calling getInner() on each bean. Execute the program and observe the output.
    - Inner1 bean created at the start of appplication.
    - Outer1 bean created on each call to ctx.getBean(). In each Outer1 bean, same Inner1 bean is injected.
    - Multiple calls to getInner() returns same Inner1 bean reference.

Spring JDBC

Spring Boot JDBC Integration

- Spring DI simplifies JDBC programming.
- Using JDBC we can avoid overheads of ORM tools. This is helpful in small applications, report generation tools or running ad-hoc SQL queries.
- In Spring project (NOT Boot), we need to create DataSource bean for JDBC connections and JdbcTemplate for JDBC operations.
- Spring Boot auto-create JdbcTemplate bean. It can also auto-create DataSource bean (if spring.datasource.xxx properties are given).
- Important JdbcTemplate methods:
    - query() -- SELECT statement
    - update() -- DML statement
    - execute() -- DDL statement

Integration steps

- In pom.xml, add spring-boot-starter-jdbc and JDBC driver(s).
- Create dataSource bean in a @Configuration class. The data-source will auto-created in Spring Boot (if spring.datasource.xxx properties are given).
- Implement RowMapper interface in a class (for dealing with SELECT queries)
  - mapRow() convert resultset row to Java object.
- Create Some DAO (@Repository) and @Autowired JdbcTemplate in it.
- Invoke JdbcTemplate query() and/or update() for appropriate operations.

Spring Jdbc (sb07_jdbcbookshop)

- Spring Initilizr: New Maven Project
  - Fill group id, artifact id, Spring Boot=3.x.y, Language=Java, Java version=17, packaging=Jar. Add dependencies "JDBC API", and "MySQL Driver".
  - Modify pom.xml and update Maven project.
    - `<version>2.7.18</version>`
    - `<java.version>11</java.version>`
- In application.properties add database properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/dbname
spring.datasource.username=dbuser
spring.datasource.password=dbpassword
```

- Create entity POJO class "Book" with fields id, name, author, subject, and price.
- Create BookRowMapper class that converts current ResultSet row to the Book object.

```
@Component
public class BookRowMapper implements RowMapper<Book> {
    @Override
    public Book mapRow(ResultSet rs, int rowNum) throws SQLException {
        int id = rs.getInt("id");
        String name = rs.getString("name");
```

```
        String author = rs.getString("author");
        String subject = rs.getString("subject");
        double price = rs.getDouble("price");
        Book b = new Book(id, name, author, subject, price);
        return b;
    }
}
```

- Create BookDao interface and implement BookDaoImpl class to perform JDBC operations. @Autowired BookRowMapper and JdbcTemplate in it.

```
@Repository
public class BookDaoImpl implements BookDao {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    @Autowired
    private BookRowMapper bookRowMapper;

    public int save(Book b) {
        String sql = "INSERT INTO books(name, author, subject, price) VALUES(?,?,?,?)";
        int count = jdbcTemplate.update(sql, c.getName(), c.getAuthor(), c.getSubject(), c.getPrice());
        return count;
    }

    public List<Category> findAll() {
        String sql = "SELECT * FROM books";
        List<Category> list = jdbcTemplate.query(sql, bookRowMapper);
        return list;
    }

    // ...
}
```

- In main class, @Autowired BookDao.

- Inherit main class from CommandLineRunner and in run() method, test the DAO methods.

Spring Web MVC -- Video Tutorial

- https://youtu.be/NzhEXRDojzA

Assignments

1. In earlier Logger classwork, use @Autowired on Logger field in AccountImpl. If you have set logger manually in main code, comment those lines. Comment @Primary on bean class (if written). Observe the error and solve it.
2. Implement JDBC Daos for BookShop application using Spring.