HIBERNATE AND SPRING MVC INTEGRATION

Instructor:

Table Content

Learning Goals

- **After the course, attendees will be able to:**

Understand Spring Web MVC Framework and Hibernate ORM Integration.
Know how to write a Web application.

Section 1

- @MODELATTRIBUTE ANNOTATION
- MODEL, MODELMAP AND MODELANDVIEW CLASSES

@ModelAttribute Annotation

- The *@ModelAttribute* is an annotation that binds a **method parameter or method return value** to a named model attribute and then exposes it to a web view.
- **Method Argument:**
  - ✓ When used as a method argument, it indicates the argument should be retrieved from the model: **the arguments fields should be populated from all request parameters that have matching names**.
  - ✓ **Example:** In the code snippet that follows the *employee* model attribute is populated with data from a form submitted to the *saveEmployee* endpoint.

```
@RequestMapping(value = "/saveEmployee", method = RequestMethod.POST)
    public String submit(@ModelAttribute("employee") Employees employee) {
        // Code that uses the employee object

        return "employeeView";
    }
```

@ModelAttribute Annotation

❖ Method Level:

- ✓ When the annotation is used at the method level it indicates the **purpose of that method is to add one or more model attributes**.

```
@ModelAttribute
    public void addAttributes(Model model) {
        model.addAttribute("msg",
                           "Welcome to the Netherlands!");
    }
```

- ✓ Spring-MVC will always make a *call first to that method, before it calls any request handler methods*.
- ✓ That is, **@ModelAttribute** methods are invoked before the controller methods annotated with **@RequestMapping** are invoked.

Model class

- The model works a container that contains the data of the application. A data can be in any form such as ***objects***, ***strings***, ***information from the database***, etc.
- The model can supply attributes used for rendering views.
- To provide a view with usable data, we simply add this data to its *Model* object.
- Maps with attributes can be merged with *Model* instances:

```
@GetMapping("/showViewPage")
public String passParametersWithModel(Model model) {
   Map<String, String> map = new HashMap<>();
   map.put("spring", "mvc");
   model.addAttribute("message", “Welcome to Spring framework");
   model.mergeAttributes(map);
   return "viewPage";
}
```

ModelMap class

- Just like the *Model* interface above, *ModelMap* is also used to pass values to render a view.
- The advantage of *ModelMap* is it gives us the ability to pass a collection of values and treat these values as if they were within a *Map*.
- ModelMap class subclasses **LinkedHashMap**. It add some methods for convenience.
- **ModelMap** uses as generics and checks for null values.

```
@RequestMapping("/helloworld")
public String hello(ModelMap map) {
  String helloWorldMessage = "Hello world from FA!";
  String welcomeMessage = "Welcome to  FA!";
  map.addAttribute("helloMessage", helloWorldMessage);
  map.addAttribute("welcomeMessage", welcomeMessage);

  return "hello";
}
```

ModelAndView

- This interface allows us to pass all the information required by Spring MVC in one return.

```
@GetMapping("/goToViewPage")
public ModelAndView passParametersWithModelAndView() {
    ModelAndView modelAndView = new ModelAndView("viewPage");
    modelAndView.addObject("message", "Welcome to FA");
    return modelAndView;
}
```

Section 2

@COMPONENT VS @REPOSITORY AND @SERVICE IN SPRING

Introduction

- In most typical applications, we have distinct layers like data access, *presentation*, *service*, *business*, etc.
- In each layer, we have various beans. Simply put, **to detect them automatically.** Spring uses classpath scanning annotations. Then, it registers each bean in the **ApplicationContext**.
  - ✓ `@Component:` is a generic stereotype for any Spring-managed component.
  - ✓ `@Service:` annotates classes at the service layer
  - ✓ `@Repository:` annotates classes at the persistence layer, which will act as a *database repository*.

Examples:

@Service:

```
public interface UserService {
    User login(User user) throws Exception;
}
@Service("userService")
public class UserServiceImpl implements UserService {
     // TODO Auto-generated method stub
}
```

@Repository

```
public interface UserDao {
    User login(User user) throws Exception;
}

@Repository("userDao")
@Transactional
public class UserDaoImpl implements UserDao {

  @Autowired
  private SessionFactory sessionFactory;
}
```

Section 3

SPRING JDBCTEMPLATE

Spring JdbcTemplate

- Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries.

Problems of **JDBC API:**	We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
	We need to perform exception handling code on the database logic.
	We need to handle transaction.
	Repetition of all these codes from one to another database logic is a time consuming task.

Spring Jdbc Approaches

- Spring framework provides following approaches for JDBC database access:
  - ✓ **JdbcTemplate**
  - ✓ **NamedParameterJdbcTemplate**
  - ✓ **SimpleJdbcTemplate**
  - ✓ **SimpleJdbcInsert** and **SimpleJdbcCall**

JdbcTemplate class

- It is the central class in the Spring JDBC support classes.
- It takes care of **creation** and **release of resources** such as creating and closing of connection object etc. *So it will not lead to any problem if you forget to close the connection.*
- It handles the exception and provides the informative exception messages by the help of excepion classes defined in the **org.springframework.dao** package.
- We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.
- Let's see the methods of spring **JdbcTemplate** class.

JdbcTemplate class

Method	Description
public int **update**(String query)	is used to insert, update and delete records.
public int **update**(String query, Object... args)	is used to insert, update and delete records using PreparedStatement using given arguments.
public void **execute**(String query)	is used to execute DDL query.
public T **execute**(String sql, PreparedStatementCallback action)	executes the query by using PreparedStatement callback.
public T **query**(String sql, ResultSetExtractorrse)	is used to fetch records using ResultSetExtractor.
public List **query**(String sql, RowMapper rse)	is used to fetch records using RowMapper.

DriverManagerDataSource class

- The **DriverManagerDataSource** is used to contain the information about the database such as driver class name, connnection URL, username and password.
- Need to provide the reference of DriverManagerDataSource object in the JdbcTemplate class for the datasource property.
- **Declare datasource bean:**

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

<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName“
                        value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
    <property name="url"
                    value="jdbc:sqlserver://1WDDIEUNT1-LT:1433;databaseName=FAMS" />
    <property name="username" value="sa" />
    <property name="password" value="12345678" />
</bean>
```

Section 4

HIBERNATE AND SPRING MVC INTEGRATION

Add dependencies

```
<!-- Spring ORM -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.2.10.RELEASE</version>
</dependency>
<!-- Hibernate Core -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.17.Final</version>
</dependency>
```

Create a Data Source Bean

- Spring provides **many ways** to establish connection **to a database** and **perform operations** such as *retrieval of records*, *insertion of new records* and *updating / deletion* of existing records.
- The most basic of them is using **DriverManagerDataSource**.

```
<!-- DataSourse -->
<bean id="dataSource“
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
                value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
    <property name="url"
        value="jdbc:sqlserver://localhost:1433;
                                    databaseName=HumanResourceDB" />
    <property name="username" value="sa" />
    <property name="password" value="12345678" />
</bean>
```

Create a Session Factory Bean

- For using Hibernate 5 with Spring, we have to use *org.springframework.orm.hibernate5.LocalSessionFactoryBean*.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource"
                ref="dataSource" /> <!-- Dependency Injection -->
    <property name="packagesToScan" value="fa.training.entities" />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.SQLServerDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
</bean>
```

Transaction Management

❖ Enable the transaction support:

```
<tx:annotation-driven
            transaction-manager="transactionManager" />

<bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

@Transactional

- With Spring **@Transactional**, the above code gets reduced to simply this:

```
@Transactional
public void businessLogic() {
      ... use entity manager inside a transaction ...
}
```

- By using **@Transactional**, many important aspects such as ***transaction propagation are handled automatically***.
- In this case if another transactional method is called by **businessLogic**(), that method will have the option of joining the ongoing transaction.

Entity class

❖ Create User class:

```
@Entity
@Table(name = "USERS")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "USER_ID")
    private long userId;

    @Pattern(regexp="^[A-Za-z0-9_]+$", message="{user.name.rex}")
    @Size(max = 30, min = 8, message = "{user.name.invalid}")
    @NotEmpty(message = "Please enter username")
    @Column(name = "USERNAME", nullable = false, unique = true)
    private String username;

    @Column(name = "PASSWORD")
    private String password;

    @OneToOne(mappedBy = "user", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    // @PrimaryKeyJoinColumn
    private UserDetail userDetail;

    // constructors and getter/setter methods

}
```

Service classes

❖ Create service class:

```
public interface UserService {
    void save(User user);

    List<User> list();
}

@Service
public class UserServiceImp implements UserService {
    @Autowired
    private UserDao userDao;

    public void save(User user) {
        userDao.save(user);
    }

    public List<User> list() {
        return userDao.list();
    }
}
```

DAO classes

- **Create DAO class:**

```
public interface UserDao {
    void save(User user);
    List<User> list();
 }

 @Repository
 @Transactional
 public class UserDaoImp implements UserDao {

    @Autowired
    private SessionFactory sessionFactory;

    @Override
    public void save(User user) {
       sessionFactory.getCurrentSession().save(user);
    }

    @Override
    public List<User> list() {
       TypedQuery<User> query = sessionFactory.getCurrentSession()
                                .createQuery("from User");
       return query.getResultList();
    }
 }
```

Build a simple Web App

- Demo!

Section 5

SPRING @PATHVARIABLE ANNOTATION

A Simple Mapping

- **The *@PathVariable* annotation can be used to handle template variables in the request URI mapping**, and use them as method parameters.
- A simple use case of the *@PathVariable* annotation would be an endpoint that identifies an entity with a primary key:

```
@GetMapping("/api/employees/{id}")
@ResponseBody
public String getEmployeesById(@PathVariable String id) {
    return "ID: " + id;
}
```

- A simple *GET request* to */api/employees/{id}* will invoke ***getEmployeesById*** with the extracted id value:

```
http://localhost:8080/api/employees/111
---- ID: 111
```

Specifying the Path Variable Name

- If the path variable name is different, we can specify it in the argument of the *@PathVariable* annotation:

```
@GetMapping("/api/employeeswithvariable/{id}")
@ResponseBody
public String getEmployeesByIdWithVariableName(@PathVariable("id")
                                                String employeeId) {
        return "ID: " + employeeId;
}
```

- We can also define the path variable name as *@PathVariable(value="id")* instead of *PathVariable("id")* for clarity.

Multiple Path Variables in a Single Reques

- Depending on the use case, **we can have more than one path variable in our request URI for a controller method, which also has multiple method parameters**:

```
@GetMapping("/api/employees/{id}/{name}")
@ResponseBody
public String getEmployeesByIdAndName(@PathVariable String id,
                                       @PathVariable String name) {
        return "ID: " + id + ", name: " + name;
}
```

Multiple Path Variables in a Single Reque

- **We can also handle more than one *@PathVariable* parameters using a method parameter of type *java.util.Map<String, String>:***

```
@GetMapping("/api/employeeswithmapvariable/{id}/{name}")
@ResponseBody
public String getEmployeesByIdAndNameWithMapVariable(
                    @PathVariable Map<String, String> pathVarsMap) {
        String id = pathVarsMap.get("id");
        String name = pathVarsMap.get("name");
        if (id != null && name != null) {
            return "ID: " + id + ", name: " + name;
        } else {
            return "Missing Parameters";
        }
}
```

- **Example:**

```
http://localhost:8080/api/employees/1/bar
---- ID: 1, name: bar
```

@PathVariable as Not Required

- Since method parameters annotated by *@PathVariables* are mandatory by default, it doesn't handle the requests sent to */api/employeeswithrequired* path:
- You can set required = false:

```
@GetMapping(value = { "/api/employeeswithrequiredfalse",
                      "/api/employeeswithrequiredfalse/{id}" })
@ResponseBody
public String getEmployeesByIdWithRequiredFalse(
                           @PathVariable(required = false) String id) {
        if (id != null) {
            return "ID: " + id;
        } else {
            return "ID missing";
        }
}
```

```
http://localhost:8080/api/employeeswithrequiredfalse
---- ID missing
```

SUMMAY

1. • @ModelAttribute annotation, Model, ModelMap class
2. • Spring JdbcTemplate
3. • Spring Web MVC and Hibernate ORM
4. • Spring @Pathvariable Annotation
5. • Build a simple Web App (CRUD)

Thank you