**SPRING-MVC**

1. **Using default annotation based validation:**

Do not forget to add the following while using mvc validation:

1. **@Valid** annotation in handle method

2. **<mvc:annotation-driven />** in your config file

1. **Custom validation**

1. use @Component in validator class

2. use @Autowired for validator instance in controller

1. **Locale based messages**

To enable the internationalization in spring MVC. We need to register below 3 beans in the spring context.

* 1. **MessageSource**

<bean id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<property name="basename" value="classpath:messages" />

<property name="defaultEncoding" value="UTF-8" />

</bean>

ReloadableResourceBundleMessageSource bean enables the internationalization (i18N).

basename property is used to provide the location of resource bundles.

value specifies that resource bundles are located at messages\_{locale}.properties.

defaultEncoding specifies the encoding used for the messages.

* 1. **LocaleResolver**

Resolves the locale based on its implementation.

It has following 3 implementations:

* + 1. **SessionLocaleResolver:**

resolves the locale based on the predefined attribute in the session.

* + 1. **CookieLocaleResolver :**

resolves the locale based on the predefined attribute in the cookie.

* + 1. **AcceptHeaderLocaleResolver :**

default implementation which resolves the locale by checking accept-language header in the HTTP request.

If we don’t define any locale resolvers , spring by default takes AcceptHeaderLocaleResolver.

Lets implement SessionLocaleResolver in our application.

Copy this code

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver">

<property name="defaultLocale" value="en" />

</bean>

* 1. **LocaleChangeInterceptor**

This interceptor intercept the HTTP request and checks for the special parameter in the request.

The name of the parameter that it has to check in the request can be specified by the property called paramName, its value will be searched in the HTTP request by this interceptor.

So here , it searches for the parameter language in the HTTP request.Its value in the request is the locale value which it will add in the place of {locale} in the line messages\_ {locale} and searches for the corresponding messages property file.

<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">

<property name="paramName" value="language" />

</bean>

NOTE: <http://www.springframework.org/schema/beans/spring-beans-3.1.xsd> was needed in config.xml file to work.

1. **Using Localized messages in Java files** 
   1. Use message source in your class (any class )

@Autowired

**private** MessageSource messageSource;

* 1. Get Locale using following method:

//independent from the Servlet API and is based on a thread local in order to provide the current locale in any entity of your architecture. locale=LocaleContextHolder.*getLocale*();

4.3. Get message using following method:

messageSource.getMessage("validation.error.userName", null, "user name missing", locale))

**JDBC-IN-SPRING**

1. **Using simple jdbc in java.**
   1. Download and install the MySQL server(The default port is 3360).
   2. Download the JDBC driver and put in classpath( extract the ZIP file and put the containing JAR file in the classpath).
   3. If you're using an IDE like Eclipse or Netbeans, then you can add it to the classpath by adding the JAR file as Library, If you're doing it "plain vanilla" in the command console, then you need to specify the path to the JAR file in the -cp or -classpath argument.
   4. Create a database in MySQL.
      1. CREATE DATABASE javabase DEFAULT CHARACTER SET utf8 COLLATE utf8\_unicode\_ci;
      2. Create an user for Java and grant it access. Simply because using root is a bad practice.

CREATE USER 'java'@'localhost' IDENTIFIED BY 'password';

GRANT ALL ON javabase.\* TO 'java'@'localhost' IDENTIFIED BY 'password';

Yes, java is the username and password is the password here.

Test the connection to MySQL using Java.

String url = "jdbc:mysql://localhost:3306/javabase";

String username = "java";

String password = "password";

System.out.println("Connecting database...");

try (Connection connection = DriverManager.getConnection(url, username, password)) {

System.out.println("Database connected!");

} catch (SQLException e) {

throw new IllegalStateException("Cannot connect the database!", e);

}

NOTE: Note that closing the Connection is extremely important. If you don't close connections and keep getting a lot of them in a short time, then the database may run out of connections and your application may break. Always acquire the Connection in a try-with-resources statement. Or if you're not on Java 7 yet, explicitly close it in finally of a try-finally block. Closing in finally is just to ensure that it get closed as well in case of an exception. This also applies to Statement, PreparedStatement and ResultSet.

1. **Using jdbc template in spring.**

Spring JdbcTemplate is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

**Problems of JDBC API**

The problems of JDBC API are as follows:

* 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 exception classes defined in the **org.springframework.dao** package.

Let's see the methods of spring JdbcTemplate class.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public int update(String query) | is used to insert, update and delete records. |
| 2) | public int update(String query,Object... args) | is used to insert, update and delete records using PreparedStatement using given arguments. |
| 3) | public void execute(String query) | is used to execute DDL query. |
| 4) | public T execute(String sql, PreparedStatementCallback action) | executes the query by using PreparedStatement callback. |
| 5) | public T query(String sql, ResultSetExtractorrse) | is used to fetch records using ResultSetExtractor. |
| 6) | public List query(String sql, RowMapper rse) | is used to fetch records using RowMapper. |

**Using Jdbc Template in spring:**

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<property name="dataSource" ref="ds"></property>

</bean>

**public** **void** createRepo(Integer id, Object value) {

User user=(User)value;

String SQL = "insert into USER (NAME, EMAIL,PASSWORD,ACCESS\_KEY) values (? , ? , ?, ?)";

jdbcTemplateObject.update(SQL, user.getUserName(), user.getUserEmail(), user.getPassword(), user.getAccessToken());

}

1. Handling SQL exception:

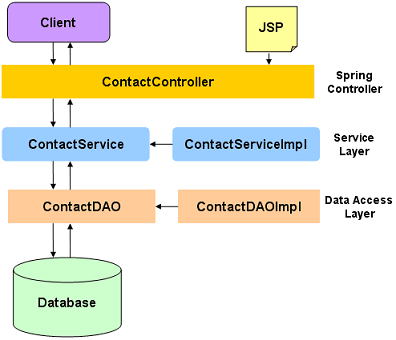
3.1 There are roughly three strategies: HandlerExceptionResolver, @ExceptionHandler and handling exceptions internally within action.

The use cases for these are: common exception handler for whole application, whole controller, specific action accordingly.

Handling exceptions internally

3.2 Spring translates all sql exception to DataAccessException which is of type Runtime. Catch this exception in your service class make it Your own

Business exception and then rethrow it again.



DAO

package com.bilitutor.cct.dao;

import com.bilitutor.cct.bean.User;

import org.hibernate.SessionFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Repository;

@Repository

public class UserDAOImpl implements UserDAO {

@Autowired

private SessionFactory sessionFactory;

public void setSessionFactory(SessionFactory sessionFactory) {

this.sessionFactory = sessionFactory;

}

public void addUser(User user) {

sessionFactory.getCurrentSession().save(user);

}

public void removeUser(String email) {

User user = (User) sessionFactory.getCurrentSession().load(User.class, email);

if (user!=null) {

sessionFactory.getCurrentSession().delete(user);

}

}

}

Service:

package com.bilitutor.cct.service;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.bilitutor.cct.dao.UserDAO;

import com.bilitutor.cct.bean.User;

@Service

public class UserServiceImpl implements UserService {

@Autowired

private UserDAO userDAO;

@Transactional

public void addUser(User user) {

try{

userDAO.addUser(user);

}

Catch(DataAccessException ex){

// Rethrow your business exception

}

}

@Transactional

public void removeUser(String email) {

try{

userDAO.removeUser(email);

}

Catch(DataAccessException ex){

// Rethrow your business exception

}

}

}

Remaining other two methods to be explored later.

**HIBERNATE-IN-SPRING**

1. **Hibernate – Sessions**

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed as needed. The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Instances may exist in one of the following three states at a given point in time:

* **transient:** A new instance of a persistent class which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.
* **persistent:** You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
* **detached:** Once we close the Hibernate Session, the persistent instance will become a detached instance.

A typical transaction should use the following idiom:

Session session = factory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

// do some work

...

tx.commit();

}

catch (Exception e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

If the Session throws an exception, the transaction must be rolled back and the session must be discarded.

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate.

1. **Mapping File**

An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate how to map the defined class or classes to the database tables.

Example:

public class Employee {

private int id;

private String firstName;

private String lastName;

private int salary;

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="Employee" table="EMPLOYEE">

<meta attribute="class-description">

This class contains the employee detail.

</meta>

<id name="id" type="int" column="id">

<generator class="native"/>

</id>

<property name="firstName" column="first\_name" type="string"/>

<property name="lastName" column="last\_name" type="string"/>

<property name="salary" column="salary" type="int"/>

</class>

</hibernate-mapping>

You should save the mapping document in a file with the format <classname>.hbm.xml.

1. **Using Hibernate in Spring**

Natively, the main interface for working with Hibernate is org.hibernate.Session. The Session interface provides basic data-access functionality such as the ability to

save, update, delete, and load objects from the database. Through the Hibernate Session, an application’s repository performs all of its persistence needs.

The standard way to get a reference to a Hibernate Session object is through an implementation of Hibernate’s SessionFactory interface.

In Spring, the way to get a Hibernate SessionFactory is through one of Spring’s Hibernate session-factory beans. As of version 3.1, Spring comes with three sessionfactory beans to choose from:

 org.springframework.orm.hibernate3.LocalSessionFactoryBean

 org.springframework.orm.hibernate3.annotation.AnnotationSession-

FactoryBean

 org.springframework.orm.hibernate4.LocalSessionFactoryBean

These session-factory beans are implementations of Spring’s FactoryBean interface that produce a Hibernate SessionFactory when wired into any property of type

SessionFactory.

LocalSessionFactoryBean is configured here with three properties. The dataSource property is wired with a reference to a DataSource bean. The mappingResources

property lists one or more Hibernate mapping files that define the persistence strategy for the application. Finally, hibernateProperties is where you configure the

minutia of how Hibernate should operate.

Example:

<bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

<property name=*"mappingResources"*>

<list>

<value>device.hbm.xml</value>

</list>

</property>

<property name=*"hibernateProperties"*>

<props>

<prop key=*"hibernate.dialect"*>org.hibernate.dialect.MySQLDialect</prop>

<prop key=*"hibernate.hbm2ddl.auto"*>update</prop>

<prop key=*"hibernate.show\_sql"*>true</prop>

</props>

</property>

</bean>

MySQLDialect is used to construct SQL for MySQL database.

But instead of listing Hibernate mapping files, you can use the packagesToScan property to tell Spring to scan one or more packages, looking for domain classes that are annotated for persistence with Hibernate. This includes classes that are annotated with JPA’s @Entity or @MappedSuperclass and Hibernate’s own @Entity annotation.

In the early days of Spring and Hibernate, writing a repository class would involve working with Spring's HibernateTemplate. HibernateTemplate would ensure that only one Hibernate session would be used per transaction. The downside of this approach is that your repository implementation would be directly coupled to Spring.

The best practice now, however, is to take advantage of Hibernate contextual sessions and not use HibernateTemplate at all.



Here @Repository serves two purposes:

1. It is looked up while component-scanning.
2. In addition to helping to reduce explicit configuration, @Repository serves another purpose. Recall that one of the jobs of a template class is to catch platformspecific exceptions and rethrow them as one of Spring’s unified unchecked exceptions. But if you’re using Hibernate contextual sessions and not a Hibernate template, how can the exception translation take place? To add exception translation to a template-less Hibernate repository, you just need to add a PersistenceExceptionTranslationPostProcessor bean to the Spring application context.

<!-- Initialization for transactionManager -->

<bean id=*"transactionManager"*

class=*"org.springframework.orm.hibernate3.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"* />

</bean>

<!-- Initialization for transactionManager ends-->

<bean id=*"PersistenceExceptionTranslationPostProcessor"* class=*"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"* />



**NOTE:**

1. Transaction is needed to open a session, Else we were getting error while getting current session.
2. While converting unchecked exception to checked exception we need to define rollbackFor=”abcException”, otherwise transaction will not rollback and your unchecked exception

Will be overridden by some other exception.

1. When is the session closed?

It all depends on how you obtain the session.

if you use sessionFactory.getCurrentSession(), you'll obtain a "current session" which is bound to the lifecycle of the transaction and will be automatically flushed and closed when the transaction ends (commit or rollback).

if you decide to use sessionFactory.openSession(), you'll have to manage the session yourself and to flush and close it "manually".

To implement a session-per-request pattern, prefer the first approach (much easier and less verbose). Use the second approach to implement long conversations.

1. xmlns:xsi is needed for AOP and TX
2. **Using Annotation**

Following changes will enable annotation in hibernate

<bean id=*"sessionFactory"* class=*"org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

<property name=*"packagesToScan"*>

<list>

<value>com.spring.hibernate.entities.Device</value>

</list>

</property>

-----------

</bean>

@Entity

@Table(name = "DEVICE")

**public** **class** Device {

@javax.persistence.Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

@Column(name = "ID")

**private** **int** Id;

@Column(name = "DEVICE\_ID")

**private** String deviceId;

//

}