**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:

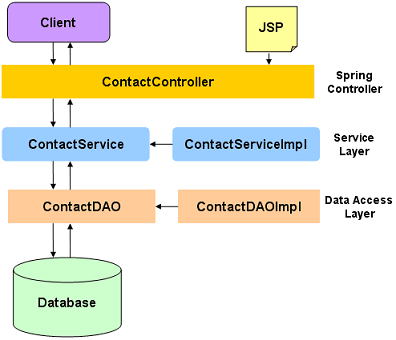
3.1 There are roughly three strategies: HandlerExceptionResolver, @ExceptionHandler and handling exceptions internally within action.

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.