JSP Custom Tag Library for In-Place Editing in Disconnected Architecture – A Case Study

Dr.Poornima G. Naik
Department of Computer Studies
CSIBER
Kolhapur, India
pgnaik@siberindia.edu.in

Dr. K.S.Oza
Department of Computer Science
Shivaji University
Kolhapur, India

Abstract— Even a trivial web application involves some sort of database functionality with few basic operations. However, such applications reveal a lot of code repetition. To avoid such a scenario, various techniques have been proposed in literature for reusability of code. One such technique is implementing custom tag library. Many frameworks rely on their own proprietary tag libraries for compacting the code. For example, struts framework abundantly uses HTML tag library, Beans tag library and Logic Tag library. JSF frame work utilizes core tag library. Connected architecture suffers from severe limitations and has become obsolete in web applications where no. of users and hence no. of database connections is not predetermined. In such a scenario connected architecture becomes predominant and scores well on its connected counterpart. In order to cater this need of a typical web application, in this paper, authors have presented JSP tag libraries for in-place editing in disconnected architecture. The tag currently works with MS-Access, MySQL and Oracle and can easily be extended to incorporate other back ends. Current work reveals encapsulation basics targeting the elimination of boilerplate code where lot of repeated code is hidden behind custom tags. Such a mechanism boils down to the entire database operations to one liner when tag attributes conceal the plethora of information involved in implementation of functionality. This paper emphasizes on development of a generic tag for disparate back ends. The database extensions are properly taken care of. The proper data type casting is performed by pulling out database schema information from the underlying DBMS. The communication between Tag Handler class and JSP page is established through PageContext class. The name of JSP page becomes available to the Tag Handler class through PageContext class which can also be used for retrieving query string parameters in a Tag Handler class.
Keywords-**Disconnected Architecture, JDOM Parser, PageContext Class, Tag Handler Class, TagLib Directive, XML.**

*****

I. INTRODUCTION

A lot of boilerplate code is seen when using JDBC. Often the small bit of code that is specific to the table operations such as, inserting a record, deleting or updating a record is concealed in a heap of other JDBC code. Most of the code is a boilerplate code required towards code management and handle exceptions. Such a boilerplate code makes the application hard to maintain, as the same code is repeated at many places to accomplish common, otherwise simple tasks which makes the application potentially buggy. There are many alternatives to overcome this issue. Spring's JDBC template is one such solution which potentially eliminates such boilerplate code by encapsulating it in various templates.

JSP section incorporates a section entitled Tag Extensions describing a mechanism for creating new actions which can be embedded in a JSP page. There are several design goals for tag construction focusing on the simplicity of tag usage by the non-programmers and the usability of tags in any JSP-Compliant container.

JSP action tags encapsulate a huge amount of code inside a tag library and provide a convenient platform for an end user for rapid development of web applications. One of the authors has demonstrated the flavor of JSP custom tag for displaying table data for disparate back ends, performing various DML operations, displaying master-details relationships and performing table joins, which conceal lot of boilerplate code. JSP custom tags have the strength of reducing complex operations involving huge code to one liners [1-3].

.

There are three main steps involved in writing a custom tag.

- Implementation of a tag handler class for the custom tag extending either javax.servlet.jsp.tagext.TagSupport or javax.servlet.jsp.tagext.BodyTagSupport
- Writing a Tag Library Descriptor (TLD) document in XML format that describes the custom tag such as its name, name of the tag handler class, body content, attributes, etc
.
- Using the custom tag in JSP page using taglib directive.

Depending on the behavior of tag, tag handler class can implement one of the following three interfaces:

- Tag
- IterationTag or
- BodyTag

where the inheritance relationship between the three is shown in Figure 1.

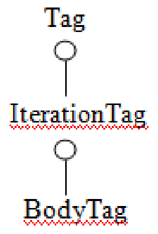


Figure 1. Inheritance Relationship Between Tag Handler Interfaces

In order to render the implementation of custom tag easier, two utility classes have been provided which include default implementation of the methods of these interfaces.

- TagSupport
- BodyTagSupport

TagSupport implements both the Tag and IterationTag interfaces whereas BodyTagSupport class extends TagSupport and implements BodyTag interface as shown in Figure 2.

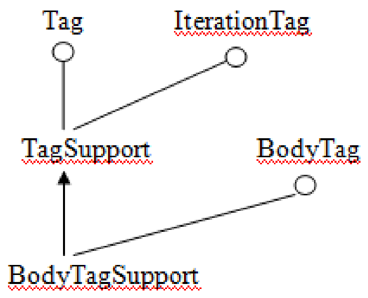


Figure 2. Inheritacne Diagram for bodyTagSupport Class

TagSupport is normally used for empty tags whereas BodyTagSupport is handy for tags with some content.

Runtime expressions render the tag more configurable at runtime by enabling the values to be assigned to the attributes dynamically at runtime depending on the end user interaction with the application.

A. Tag Libray Descriptor Document

The tag library descriptor file is a simple XML file which contains a set of custom tags. Each tag may define its own set of attributes identified by a unique name, among other required information, attributes contain some optional information such as whether the attribute is a required attribute or not, whether can be initialized dynamically at runtime or not etc. The standard format of the tab library descriptor file is shown below:

```
<?xml version="1.0" ?>
<!DOCTYPE taglib
 PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library
1.2//EN"
 "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
 <taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.1</jsp-version>
  <short-name>simpletaglib</short-name>
  <description>My first Tag Library</description>
<tag>
  <name>…</name>
   <tag-class>…</tag-class>
   <body-content>…</body-content>
   <attribute>
    <name>…</name>
     .
     .
   </attribute>
    .
   .
</tag>
.
.
<tag>
</tag>
</taglib>
```

The required child elements of <tag> element are <name>, <tag-class> and <body-content> and optional child element is <attribute>. The <attribute> child element contains the compulsory child element <name> and other optional child elements such as <rtexprvalue>, <required>, etc.

Using runtime expressions in custom tag attributes simply boils down to adding the element

<rtexprvalue>true</rtexprvalue>

to the corresponding tag in the tag library descriptor file. Further, any attribute can be made compulsory by adding the element

<required>true</required>

to the corresponding tag in the tag library descriptor file.

B. Communication Between Tag Handler class and Client JSP Page

The submit button on the JSP page can cause either to the same page or can cause a cross page post back. During the self post the same page is rendered with the suitable query string parameters for displaying the records of the detail table. Since the table is rendered by the corresponding Tag Handler class, the whole crux of the problem is retrieving the name of the client JSP page within Tag Handler class and specifying the type of the operation the user desires to perform at runtime. Hence the main logic of the application is two-fold.

• Extracting the name of the JSP page using the tag in the Tag Handler class at runtime.
• Extracting request parameters from the Tag Handler class.

The code employed for extracting client JSP page name inside a Tag Handler class is

```
String pagename=this.pageContext.getPage().toString();
int to=pagename.indexOf("@");
int from=pagename.lastIndexOf(".");
String page=pagename.substring(from+1, to-4)+".jsp";
```

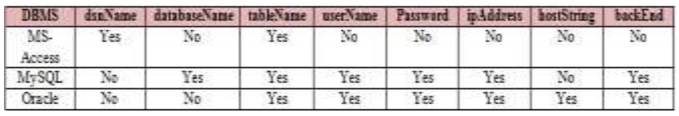
The syntax used for extracting request parameters from within Tag Handler class is

this.pageContext.getRequest().getParameter("ParameterName");

Substitute the actural parameter name in the place holder "ParameterName".

In order to render the tag generic, various mutually exclusive attributes specific to the back end database management system being used are added to the tag definition in tag library descriptor file. The Table I summarizes the attributes meaningful to different database management systems.

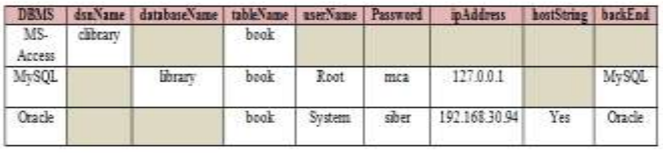
TABLE I ATTRIBUTES APPLICABLE TO DIFFERENT BACKENDS



DBMS	dsnName	databaseName	tableName	userName	Password	ipAddress	hostString	backEnd
MS-Access	Yes	No	Yes	No	No	No	No	No
MySQL	No	Yes	Yes	Yes	Yes	Yes	No	Yes
Oracle	No	No	Yes	Yes	Yes	Yes	Yes	Yes

The Table II summarizes the values of the attributes corresponding to different database management systems

TABLE II. ATTRIBUTE VALUES FOR DIFFERENT BACKENDS



DBMS	dsnName	databaseName	tableName	userName	Password	ipAddress	hostString	backEnd
MS-Access	clibrary		book					
MySQL		library	book	Root	mca	127.0.0.1		MySQL
Oracle			book	System	siber	192.168.30.94	Yes	Oracle

backEnd attribute defaults to MS-Access.

C. Disconnected Architecture

A typical database application operates in two different architectures.

- Connected Architecture
- Disconnected Architecture.

The main drawback associated with connected architecture is that the application remains connected to the database system even when it is not performing any database operations. Hence if the web application uses connected architecture most of the users trying to connect to the database server will be deprived of database connections as the number of connections is limited. The communication between web server and web browser is facilitated using HTTP protocol which is connectionless and stateless. On the same lines database disconnected architecture operates to facilitate resource sharing between multiple users and enabling connection to database server to multiple users.

The languages supporting web application development promote a disconnected architecture in which data can be retrieved from the database even when the connection to the database is closed, providing better application scalability. In such an architecture, table updation information is stored persistently in a dataset which can also be stored on a client machine persistently in an XML format.

In a disconnected architecture database operations are performed in three phases :

Phase 1 : Data Retrieval Phase

In this phase connection to the database server is established only to pull the required data from the server after which the database connection is closed.

Phase 2 : Local Data Updation

In this phase the data is updated and the updation information along with the updated data is stored locally, predominantly in XML format.

Phase 3 : Remote Database Updation Process

In this phase, database server connection is re-established, updation information which is stored locally is read and the remote database is updated accordingly.

In the current work, authors have developed a custom tag library for common JDBC operations. This tag library enables an end user to perform database operations hiding traditional JDBC code behind the set of tags and frees an end user from the intricacies involved in connection to different back ends, sorting, filtering, joining and developing master-detail relationships. The project is developed in Eclipse and tested with several test cases.

II. LITERATURE REVIEW

Custom tags play an important role in web applications. JSP custom tags are written to extract data from database using drop down menu to generate options dynamically [4]. A through investigation for categorization of requirements and design of tag software in web application has been carried out by [5]. Authors have presented a case study of freely available tag software. The development and testing of an accurate mass–time (AMT) tag approach for the LC/MS-based identification of plant natural products in complex extracts has been reported by [6]. Its utility is verified by the detection and annotation of active principles in different medicinal plant species with diverse chemical constituents. Tagging plays a vital role in bioinformatics also . A method to generate poly(A) tags libraries for high-throughput sequencing (PAT-seq) has been reported by [7]. This method has been applied to investigate mRNA polyadenylation in Arabidopsis.

Internet has become a vital source of information. Due to this there is need for powerful internet systems which can help in audiovisual content searching on internet. A new technique of searching and indexing of audio visual contents on the internet has been carried out by [8]. When developers are working on different platforms then code migration is a major issue. Three methods of code migrations from JSP to ASP.NET Entire code transform migration, Reserved migration and Neutral migration has been proposed by [9]. In development of IOT based applications there is need for a way to connect things and services together and processing of data emitted by them using data flow paradigms. Automation of distribution of these data flows using appropriate distribution mechanism has been carried out by [10].

III. DESIGN OF CUSTOM TAG

This section presents structure of tag, the control flow diagram and the proposed algorithm for the implementation of InPlaceEditingTable tag. Figure 3(a) – 3(b) present the control flow diagram for database updation in different architectures in disconnected architecture.

The structure of InPlaceEditingTable tag along with name of the tag handler class and attribute information is shown below:

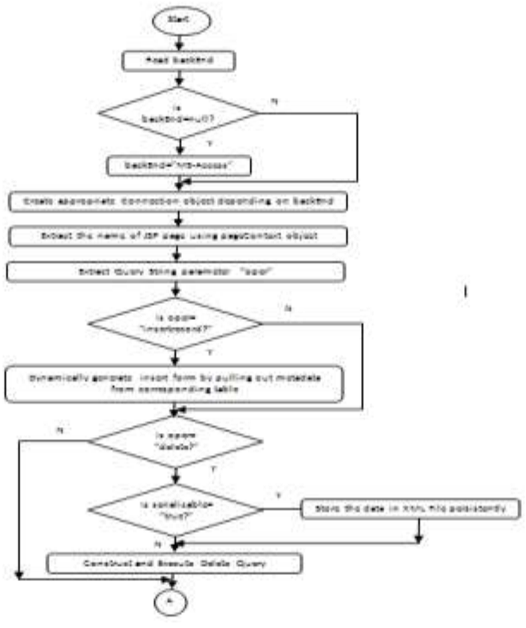
A. Structure of InPlaceEditingTable Tag.

```
<tag>
    <name>InPlaceEditingTable</name>
    <tag-class>csiber.InPlaceEditingTableTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>dsnName</name>
    </attribute>
    <attribute>
      <name>tableName</name>
    </attribute>
    <attribute>
      <name>databaseName</name>
    </attribute>
    <attribute>
      <name>userName</name>
    </attribute>
    <attribute>
      <name>password</name>
    </attribute>
     <attribute>
      <name>ipAddress</name>
    </attribute>
   <attribute>
      <name>hostString</name>
    </attribute>
    <attribute>
      <name>backEnd</name>
    </attribute>


  </tag>

</taglib>
```

B. Control flow Diagram



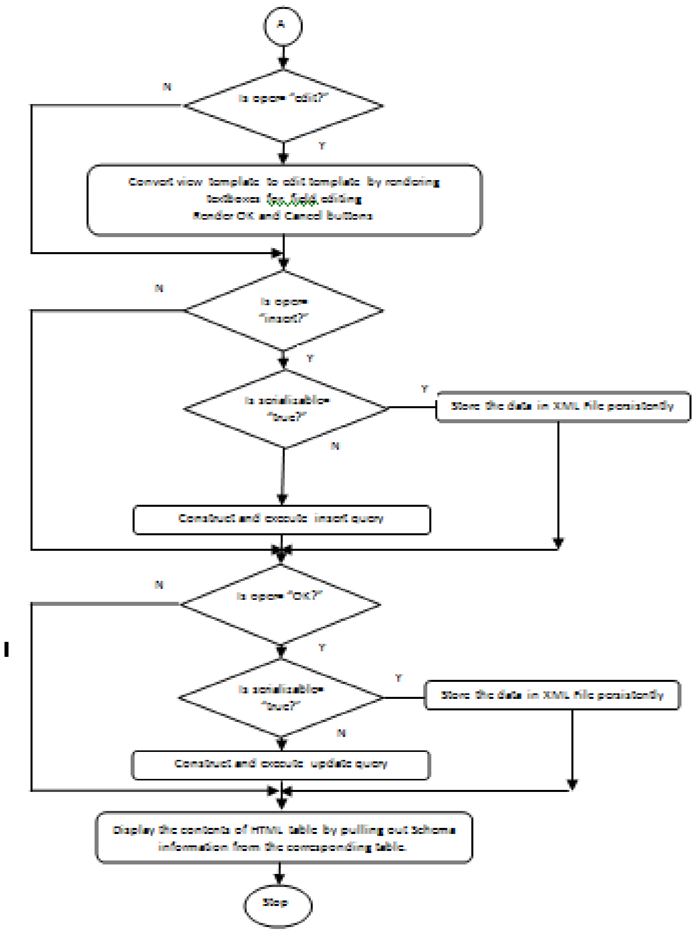

A
Is opcm "edit?"
N
Y
Convert view template to edit template by rendering textboxes for field editing
Render OK and Cancel Buttons
Is opcm "insert?"
Is serializable= "true?"
Store the data in XML file persistently
Construct and execute insert query
Is opcm "OK?"
Is serializable= "true?"
Store the data in XML file persistently
Construct and execute update query
Display the contents of HTML table by pulling out Schema information from the corresponding table.
Stop

Figure 3(a) – 3(b) Control Flow diagram for Database Updation in disconnected Architecture

C. Proposed Algorithm

The corresponding algorithm in C++ style is presented below:

/*Any high level language interfacing with back end database management system provides high level API for primitive database functions such as creating a connection object and generating a page request by sending the necessary input information in a query string. Hence this algorithm assumes some standard functions as shown below:

Standard Functions of language L used in the Algorithm

loadDriver() - is a built-in function in L which accepts the name of DBMS as parameter and loads appropriate driver in memory.

connectTo() - is a built-in function in L which accepts the name of DBMS as parameter and establishes the connection to remote DBMS.

getPageName() - is a built-in function in L which returns the name of requested web page.

getQueryString() - is a built-in function in L which accepts the name of the parameter and returns its value.

setQueryString() - is a built-in function in L which accepts two parameters corresponding to name and value of the parameter.

renderLink() - is a built-in function in L which accepts source and hypertext as parameters.

readPKName() – is a built-in function in L which returns the name of the Primary Key column
readPKValue() - is a built-in function in L which returns the value of the Primary Key column for selected row.
constructXQuery() – where X can be one of Insert, Delete or Update
Constructs an SQL query corresponding to the operation specified.
executeXQuery() – where X can be one of Insert, Delete or Update
Executes the query against backend database management system.

```
*/
/* Invoked wheh start tag is rendered */
function doStartTag()
{
        Read backEnd;
        if (backEnd==null)
        {
                backEnd=”MS-Access”;
        }
        /* Load appropriate database driver and construct
database connection */

        if (backEnd==”MS-Access”)
        {
                loadDriver(“MS-Access”);
                connectTo(“MS-Access”);
        }

        if (backEnd==”MySQL”)
        {
                loadDriver(“MySQL”);
                connectTo(“MySQL”);

        }

        if (backEnd==”Oracle”)
        {
                loadDriver(“Oracle”);
                connectTo(“Oracle”);
        }

        /* Extract the name of the page for self postback */
        String page=getPageName();
         /* Extract Query String Parameter oper */
        String oper=getQueryString(“oper”);

        if (oper != null)
        {
                if (oper==”insert”)
                {
                        displayInsertForm();
                }
                else
                {

        setQueryString(“oper”,”insertrecord”);
                        renderLink(page,”Insert”);
                }
                if (oper==”delete”)
                {
                        constructDeleteQuery();
                        executeDeleteQuery();
                }

                if (oper==”OK”)
                {
                        constructUpdateQuery();
                        executeUpdateQuery();
                }
                if (oper==”insertrecord”)
                {
                        constructInsertQuery();
                        executeInsertQuery();
                }

                if (oper==”edit”)
                {
                        renderButton(“OK”);
                        renderButton(“Cancel”);

                }
                else
                {
                        pColumnName=readPKName();
                        pColumnValue=readPKValue();
                        setQueryString(“oper”,”edit”);

        setQueryString(pColumnName,pColumnValue);
                        render_link(page,”Edit”);
                }


        }
}
```

D. *Using the code*

The partial code for parsing the serialized XML file using JDOM parser is show below:

```
SAXBuilder builder = new SAXBuilder();
File xmlFile = new File(tableName+".xml");
String cvalue=null;
String ctype=null;
String name=null;
try {

    Document document = (Document)
                builder.build(xmlFile);
    Element rootNode = document.getRootElement();
    List list = rootNode.getChildren("insert");
    for (int i = 0; i < list.size(); i++) {
    query="INSERT INTO " + tableName + " VALUES(";
    Element node = (Element) list.get(i);
    List list1 = node.getChildren("column");
    for (int j=0;j<list1.size()-1;j++){
        Element node1 = (Element) list1.get(j);
        ctype=node1.getChildText("type").trim();
```

```
            cvalue=node1.getChildText("value");
            if (ctype.equals("CHAR") ||
ctype.equals("VARCHAR") || ctype.equals("VARCHAR2"))
                cvalue="'"+cvalue+"'";
                query+=cvalue;
                query+=",";
        }
        Element node1 = (Element) list1.get(list1.size()-1);
        ctype=node1.getChildText("type").trim();
        cvalue=node1.getChildText("value");
        if (ctype.equals("CHAR") || ctype.equals("VARCHAR")
|| ctype.equals("VARCHAR2"))
            cvalue="'"+cvalue+"'";
            query+=cvalue;
            query+=")";
            st = con.createStatement();
            st.executeUpdate(query);
}
```

IV. RESULTS AND ANALYSIS

The algorithm presented in Section III (C) is implemented in Java using MS-Access, MySQL, and Oracle 10g back end database management systems.

A. *Pre-requisite components*

Figure 4. shows pre-requisite components required for the bug-free working of custom components.

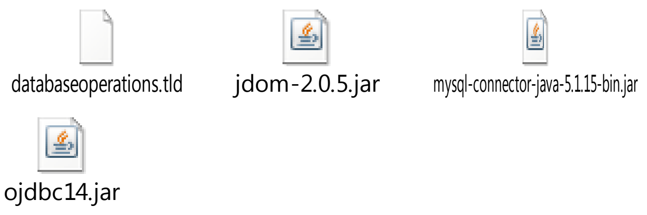


Figure 4. Components Required for Working of a Custom Tag

In order to access and display MS-Access table you just need to create a 32-bit DSN on windows and for displaying MySQL and Oracle tables you can download appropriate JDBC drivers and place them in WEB-INF/lib folder of your context root. For parsing XML file using JDOM parser jdom-2.0.5.jar file is required in WEB-INF/lib folder.

B. *JSP Client for MS-Access*

```
<%@ taglib uri="/WEB-INF/lib/databaseoperations.tld"
prefix="Database" %>
<html>
  <head>
    <title>Custom Tags for Database Operations</title>
  </head>
  <body>
    <h3>In Place Editing</h3>
    <Database:InPlaceEditingTable dsnName="clibrary"
tableName="book"/><br>
    </body>
</html>
```

JSP Client for My-SQL

```
<%@ taglib uri="/WEB-INF/lib/databaseoperations.tld"
prefix="Database" %>
<html>
  <head>
    <title>Custom Tags for Database Operations</title>
  </head>
  <body>
    <h3>In Place Editing</h3>
    <Database:InPlaceEditingTable databaseName="library"
tableName="book" backEnd="MySQL" userName="root"
password="mca" /><br>
    </body>
</html>
```

JSP Client for Oracle

```
<%@ taglib uri="/WEB-INF/lib/databaseoperations.tld"
prefix="Database" %>
<html>
  <head>
    <title>Custom Tags for Database Operations</title>
  </head>
  <body>
    <h3>In Place Editing</h3>
    <Database:InPlaceEditingTable tableName="book"
backEnd="Oracle" userName="system" password="siber"
ipAddress="192.168.30.94" /><br>
    </body>
</html>
```

C. *Structure of XML File Generated*

XML file is generated and stored locally when the serilizable attribute is set to true which contains updation information. The structure of the XML file is shown below:

```
<operations>
  <insert>
    <column>
      <name>bookId</name>
      <type>INT     </type>
      <value>1     </value>
    </column>

    <column>
      <name>bookName</name>
      <type>CHAR     </type>
      <value>vvv     </value>
    </column>

    <column>
      <name>bookPrice</name>
      <type>FLOAT     </type>
      <value>1     </value>
    </column>
  </insert>
  <insert>
```

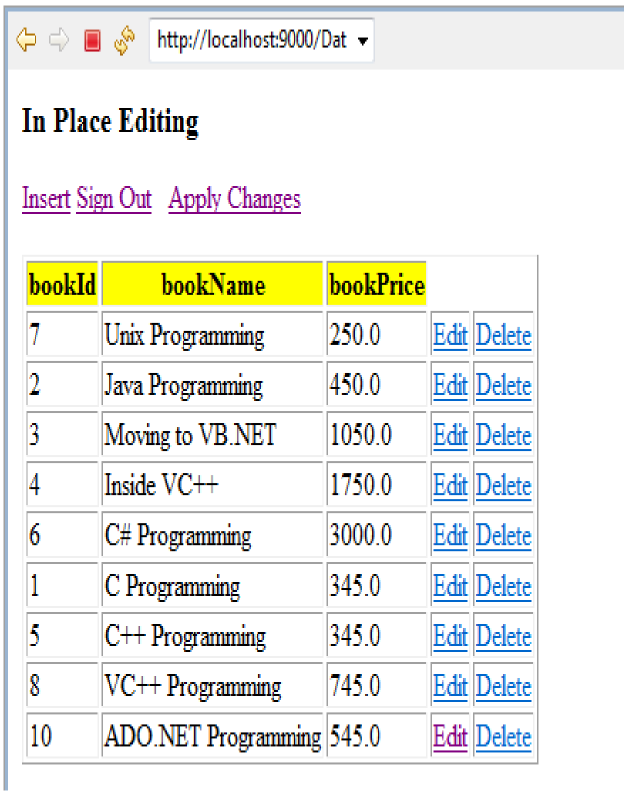
```
    <column>
      <name>bookId</name>
      <type>INT     </type>
      <value>2     </value>
    </column>

    <column>
      <name>bookName</name>
      <type>CHAR     </type>
      <value>ddd     </value>
    </column>

    <column>
      <name>bookPrice</name>
      <type>FLOAT     </type>
      <value>2     </value>
    </column>
</insert>
</operations>
```

On the contrary, if the serializable attribute is set to false, all the database operations are immediately committed on the same database connection instead of storing them persistently.

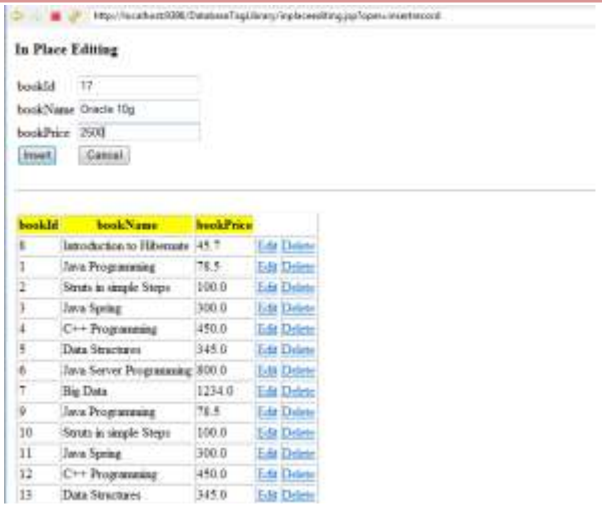
The Graphical User Interface (GUI) dynamically generated by the custom tag is shown in Figure 5(a). 'Apply changes' link is dynamically rendered, if the serializable attribute is set to true. On clicking 'Apply changes' link, the XML file is parsed using JDOM parser and updation information is committed to the remote database. Figure 5(b) and 5(c) show GUI on clicking Insert link and Edit link, respectively.


http://localhost:9000/Dat

In Place Editing

Insert Sign Out Apply Changes

bookId	bookName	bookPrice		
7	Unix Programming	250.0	Edit	Delete
2	Java Programming	450.0	Edit	Delete
3	Moving to VB.NET	1050.0	Edit	Delete
4	Inside VC++	1750.0	Edit	Delete
6	C# Programming	3000.0	Edit	Delete
1	C Programming	345.0	Edit	Delete
5	C++ Programming	345.0	Edit	Delete
8	VC++ Programming	745.0	Edit	Delete
10	ADO.NET Programming	545.0	Edit	Delete

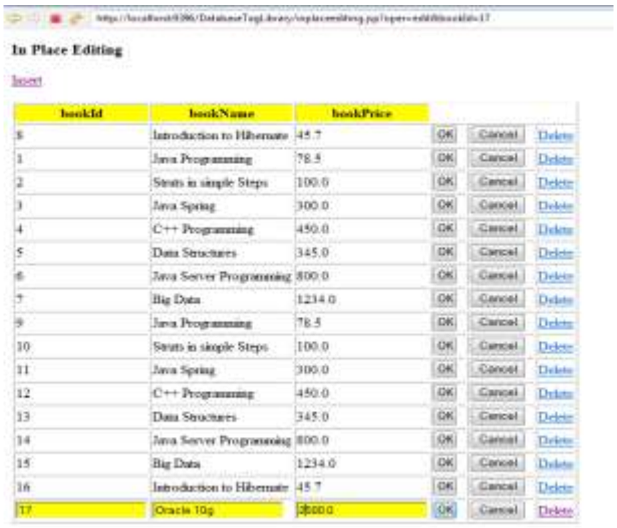

http://localhost:9000/DatabaseTagLibrary/inplaceediting.jsp?oper=insertrecord

In Place Editing

bookId	17
bookName	Oracle 10g
bookPrice	2500

Insert Cancel

bookId	bookName	bookPrice		
8	Introduction to Hibernate	45.7	Edit	Delete
1	Java Programming	78.5	Edit	Delete
2	Struts in simple Steps	100.0	Edit	Delete
3	Java Spring	300.0	Edit	Delete
4	C++ Programming	450.0	Edit	Delete
5	Data Structures	345.0	Edit	Delete
6	Java Server Programming	800.0	Edit	Delete
7	Big Data	1234.0	Edit	Delete
9	Java Programming	78.5	Edit	Delete
10	Struts in simple Steps	100.0	Edit	Delete
11	Java Spring	300.0	Edit	Delete
12	C++ Programming	450.0	Edit	Delete
13	Data Structures	345.0	Edit	Delete


http://localhost:9000/DatabaseTagLibrary/inplaceediting.jsp?oper=edit&bookId=17

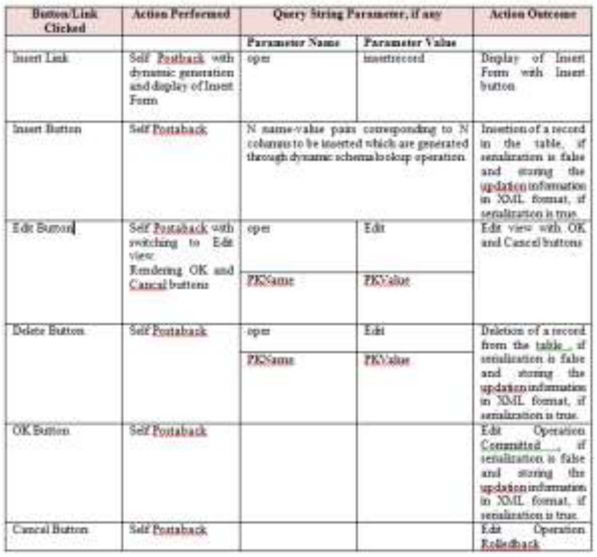
In Place Editing

Insert

bookId	bookName	bookPrice			
8	Introduction to Hibernate	45.7	OK	Cancel	Delete
1	Java Programming	78.5	OK	Cancel	Delete
2	Struts in simple Steps	100.0	OK	Cancel	Delete
3	Java Spring	300.0	OK	Cancel	Delete
4	C++ Programming	450.0	OK	Cancel	Delete
5	Data Structures	345.0	OK	Cancel	Delete
6	Java Server Programming	800.0	OK	Cancel	Delete
7	Big Data	1234.0	OK	Cancel	Delete
9	Java Programming	78.5	OK	Cancel	Delete
10	Struts in simple Steps	100.0	OK	Cancel	Delete
11	Java Spring	300.0	OK	Cancel	Delete
12	C++ Programming	450.0	OK	Cancel	Delete
13	Data Structures	345.0	OK	Cancel	Delete
14	Java Server Programming	800.0	OK	Cancel	Delete
15	Big Data	1234.0	OK	Cancel	Delete
16	Introduction to Hibernate	45.7	OK	Cancel	Delete
17	Oracle 10g	2500.0	OK	Cancel	Delete

Table III depicts the various actions performed on clicking a button/link on user interface, actions performed, query string information, if any, passed to the target page and action outcome in each case.

TABLE III ACTIONS PERFORMED AND ACTION OUTCOMES



Button/Link Clicked	Action Performed	Query String Parameter, if any		Action Outcome
		Parameter Name	Parameter Value	
Insert Link	Self Postback with dynamic generation and display of Insert Form	oper	insertrecord	Display of Insert Form with Insert button
Insert Button	Self Postaback	N name-value pairs corresponding to N columns to be inserted which are generated through dynamic schema lookup operation.		Insertion of a record in the table, if serialization is false and storing the updation information in XML format, if serialization is true.
Edit Button	Self Postaback with switching to Edit view. Rendering OK and Cancal buttons	oper	Edit	Edit view with OK and Cancel buttons
		PKName	PKValue	
Delete Button	Self Postaback	oper	Edit	Deletion of a record from the table , if serialization is false and storing the updation information in XML format, if serialization is true.
		PKName	PKValue	
OK Button	Self Postaback			Edit Operation Committed , if serialization is false and storing the updation information in XML format, if serialization is true.
Cancel Button	Self Postaback			Edit Operation Rolledback

V. CONCLUSION

In this paper, the authors have provided the design and implementation of a custom tag which operates in a disconnected architecture rendering it suitable for web applications where no. of database connections is not known before hand. Two options are presented to an end user, immediately committing the database operations to a remote database server or store all of them locally in an XML file format and commit all of them at some time in future as a single logical transaction when the load on the server is light. For parsing XML file JDOM parser which offers object-oriented interface is adopted.

REFERENCES

[1] Dr. Poornima G. Naik, JSP Custom Tag Library for Implementing JDBC Functionality, http://www.codeproject.com/Articles/1084607/JSP-Custom-Tag-Library-for-Implementing-JDBC-Funct, 11th March 2016.

[2] Dr. Poornima G. Naik, JSP Custom Tag Library (Version 2) for DML Operations, http://www.codeproject.com/Articles/1085185/JSP-Custom-Tag-Library-Version-for-DML-Operations, 14th March, 2016

[3] Dr. Poornima G. Naik, JSP Custom Tag Library for Table Joins and Master Detail Relationships, http://www.codeproject.com/Articles/1086716/JSP-Custom-Tag-Library-for-Table-Joins-and-Master, 19th March, 2016.

[4] Xiong, Yingyidu. "The design of automatically generating drop-down a menu on JSP." *Computer Science and Information Processing (CSIP), 2012 International Conference on*. IEEE, 2012.

[5] Gupta, Karan, and Anita Goel. "Requirement Estimation and Design of Tag software in Web Application." *International Journal of Information Technology and Web Engineering (IJITWE)* 9.2 (2014): 1-19.

[6] Cuthbertson, Daniel J., et al. "Accurate mass–time tag library for LC/MS based metabolite profiling of medicinal plants." *Phytochemistry* 91 (2013): 187-197.

[7] Liu, Man, Xiaohui Wu, and Qingshun Quinn Li. "DNA/RNA Hybrid Primer Mediated Poly (A) Tag Library Construction for Illumina Sequencing."*Polyadenylation in Plants: Methods and Protocols* (2015): 175-184.

[8] Kamal, Arif. "Tag Based Audiovisual Content Indexing.", MASTER'S THESIS, Master of Science, Computer Science and Engineering,Luleå University of Technology, Department of Computer science, Electrical and Space engineering, 2016

[9] Xu, Ming, et al. "Research on the Method of Code Migration from JSP to ASP. NETMing." *Advanced materials* research. Vol. 756. 2013.

[10] Nam Ky Giang, Michael Blackstock, Rodger Lea, Victor C.M. Leung , Developing IoT Applications in the Fog: a Distributed Dataflow Approach. Procs. of the Internet of Things (IOT), 2015 International Conference on the, Seoul, Korea, Oct 26-28, 2015