Automatically update a web page with dynamic elements

Li Li Lin
October 13, 2009

You may know how to hide and display optional JavaServer Faces (JSF) components by using JavaScript and Cascading Style Sheets (CSS) in standard JSF components. To do this, you would first need to identify all JSF components and write them into JSF pages. But, that is impossible to do when you are developing a Web page that contains dynamic elements that are unknown until run time. With this article, learn how you can clear old UI components while automatically updating the dynamic elements of a Web page, as well as how to use Java™ code to add new elements and put them into their proper spot on a Web page. You'll also learn how to bind different event handlers to different dynamic elements of a Web page, how to register a listener listening to changes of server-side data to invoke a page refresh, and how to use Asynchronous JavaScript and XML (Ajax) techniques to refresh only the dynamic parts of the Web page.

Introduction

Sometimes, you might need to automatically update a Web page with dynamic elements. For example, you would want a poll Web site to update the poll results as soon as its database receives new votes, or you might need a stock Web site that periodically updates real-time trading data of securities. The poll results and the real-time trading data are dynamic elements that are unknown until run time, but elements that should be added or updated when the server side sends a signal. So, how do you do this in a JSF application?

A previous developerWorks article, "Craft Ajax applications using JSF with CSS and JavaScript, Part 2: Dynamic JSF forms" explains how to hide and display optional JSF components without refreshing a Web page. However, you can't use the method described in that article to solve this problem. That method requires that you identify all JSF components and write them into your JSF pages. What if the dynamic elements cannot be identified until run time?

At this time, JSF has no good solution to this problem. Although you may be familiar with Java Swing applications such as "clock," in which data changes prompt GUI updates, or you might have read the implementation details of such applications in some basic Swing development guides, this approach will not work with my scenarios. Swing already provides a mature way to automatically update the GUI based solely on the internal data status, but JSF does not have good support for refreshing the GUI based on a request from the server side. If you check the standard life cycle of
JSF, you will find that the user normally needs to produce an event on the Web page (by clicking a button, for example) to invoke the GUI refresh. This means that even though dynamic elements can be created and added to a Web page during the run time, the Web page will not be refreshed automatically without interaction from the user.

So, how do you automatically update a Web page with dynamic elements? Within this article, I'll describe the following solutions:

- Clearing old UI components and adding new ones in their proper places on a Web page
- Binding different event handlers to different dynamic elements of a Web page
- Registering a listener looking for changes on the server side
- Using Ajax techniques to refresh only the dynamic parts of the Web page instead of the whole page

**Monitoring data changes on the server side**

To better explain my solution, I'll work with a single example throughout the entire article. The application is a Web site for online book sales. The inventory information such as book categories and the number of books in each category is displayed in the Web home page (see Figure 1).

**Figure 1. Home page for online book sale**

To reflect accurate information, you need to synchronize the inventory information of the page with the server-side data in real time. Because actions such as adding or removing a book in the inventory leads the server-side data to change, you must monitor these actions. The method for monitoring changes is to add a listener looking for changes of server-side data, and have the
server-side notify the listener after any change occurs. Listing 1 shows how to register and deregister listeners to a class.

**Listing 1. Add and remove listeners to inventory**

```java
public class Inventory{
    private Map<String, InventoryListener> listeners =
        new HashMap<String, InventoryListener>();
    
    public void register(String id, InventoryListener listener){
        listeners.put(id, listener);
    }
    public void deregister(String id){
        listeners.remove(id);
    }
}
```

Inventory listeners can be added and removed from the `Inventory` class shown in Listing 1 through two Java methods. Assuming any inventory changes are the result of actions such as adding or removing books, you can notify all the listeners registered to the `Inventory` class every time these actions occur. Listing 2 shows how to notify the listeners about the changes.

**Listing 2. Notify changes to listeners**

```java
public class Inventory{
    public void addBookItem(String bookName, String author, String price, String category){
        //codes for adding books
        categoryChanged();
    }
    public void removeBookItem(String bookName, String author, String price, String category){
        //codes for deleting books
        categoryChanged();
    }
    private synchronized void categoryChanged(){
        for (InventoryListener listener : listeners.values()) {
            listener.categoryChanged();
        }
    }
}
```

Next, you can make the managed bean, `InventoryBean`, implement `InventoryListener` and register it to the inventory data so that this bean can get the notification when the inventory data changes. Listing 3 shows how to register the managed bean to the `Inventory` class.

**Listing 3. Register the managed bean to Inventory**

```java
public interface InventoryListener {
    public abstract void categoryChanged();
}

public class InventoryBean implements InventoryListener{
    
}
```
private String m_clientId;
private InventoryNotifier m_notifier;
public InventoryBean(){
    m_notifier = InventoryNotifier.getInstance();
    if(m_clientId == null) {
        m_clientId = "bookstore";
        m_notifier.register(m_clientId, this);
    }
}
public void categoryChanged() {
    refresh();
    //code for refresh dynamic part via ajax
}
……
}

Using the methods outlined in Listing 1 through Listing 3, you establish a framework for a managed bean to monitor changes of the server-side data. The workflow is, when the managed bean gets the notification that the server-side data has been changed, the categoryChanged() method of InventoryBean is invoked and the data model is updated. Figure 2 shows that this framework builds a bridge between the database and the "Bean parts." Any application wanting to monitor data changes on the server side or receive events from the server side can use this framework as a template.

Figure 2. Business process model

![Business process model diagram]

Updating the data model and creating dynamic GUI elements

After building a framework that monitors data changes on the server side, you need to find a way to update the data model and create dynamic GUI elements if the bean is to be notified of any changes. This process takes place inside the managed bean (see the Bean layer of Figure 2) and can be divided into two sub-processes: updating the data model and creating GUI elements.
Exploring the data model

This sub-process is invoked by the `refresh()` method shown previously in Listing 3. Listing 4 shows the method for updating the data model. The `refresh()` method is used to reorganize the inventory to ensure that books have been assigned to the right categories. Therefore, after updating the data model, you can guarantee that any category without books has been removed and any new category has been added.

You'll understand the `refresh()` method better with a brief explanation of the self-defined data structures that I am using. I use the `Category` class to hold inventory information. The `Category` class includes the category name and the book's metadata in the form of the `ArrayList<BookItem>.BookItem` class, which contains the book's name, author, price, and category. Listing 4 shows how to update the data model.

Listing 4. Update data model

```java
public class InventoryBean implements InventoryListener{
    ... 
    private Inventory m_notifier;
    private Category[] m_category;
    public InventoryBean(){
        m_notifier = Inventory.getInstance();
    }
    private void refresh(){
        // reorganize the data model
        ArrayList<Category> categoryList = m_notifier.reorganizeCategory();
        // code for converting data to the type used in this bean,
        // ArrayList<Category> to Category[]
        }
    }
}
```

Creating dynamic GUI elements

Next, I'll talk about the other sub-process, creating dynamic GUI elements. The dynamic GUI elements in this case are the category links (see Figure 1). If the user clicks a specific category on the home page, he will be redirected to a new page that contains detailed information for all the books in this category. Figure 3 shows an example with all the books under the detective category.
Figure 3. Details of Detective Category

To make the category links function, you need to remove old links, insert new links into the proper position of the Web page, and bind different category detail information to different category links.

Inserting and removing links
There are two ways to remove or insert links. One way is to search the parent component of the dynamic elements in the JSF components tree and then remove or insert elements. If the parent component of the dynamic elements is changing, this method should be adopted. The
other way is to bind the dynamic elements directly to the Web page. This method is easier than the first because there is no need to find the parent node in the JSF components tree. However, this method also has a limitation because of its ease: it can only be used when the element to be removed or inserted has a fixed parent that is known before run time. I choose this method (see Listing 5) because the parent of the category links is fixed and predefined in the example.

Listing 5. Create/Update GUI components and bind different components to different action handlers

category.jsp

```java
public class InventoryBean implements InventoryListner {

    private Category[] m_category;
    public HtmlAjaxOutputPanel getCategorygrid() {
        updateGUI();
        return categorygrid;
    }

    public void setCategorygrid(HtmlAjaxOutputPanel categorygrid) {
        this.categorygrid = categorygrid;
    }

    private void updateGUI() {
        categorygrid.getChildren().clear();
        if (m_category != null) {
            int num = m_category.length;
            for (int index = 0; index < num; index++) {
                HtmlPanelGrid categorySubgrid =
                    JSFUtil.getLinkgrid("Bookstore_sublink" + index,
                    "#{InventoryBean.category[index].categoryLabel}",
                    "#{InventoryBean.category[index].onClickAction}");
                categorygrid.getChildren().add(categorySubgrid);
            }
        }
    }
}
```

As you can see, the `updateGUI()` line of the category.jsp file is to bind dynamic elements in the managed bean. It clears all dynamic elements created previously, creates new dynamic elements based on the new data model, and adds them to the predefined parent.

### Bind different behaviors to different links

Now let's talk about how to bind different category detail information to different category links. I want to iterate an Array, transfer each element to a GUI component, and insert it into the JSF components tree. My mechanism is to put all categories into an Array, with each category as an
element. Every element has a method to return the label of its category and a method to bind the clicking action. I can ensure each element has a unique behavior bound to the "onclick" action by making each element keep its own category information used to distinguish it from others.

Inside updateGUI(), "Bookstore_sublink" + index is the ID of the category link. 
"#{InventoryBean.category[" + index+ "].categoryLabel}" is the label of the category link. 
"#{InventoryBean.category[" + index+ "].onClickAction}" is the action bound to the category link. The getCategoryLabel() method is used to return the link label and onClickAction() binds the click action. (See Listing 6.)

**Listing 6. Value and action binding method**

class Category {
    private String category;
    private ArrayList<BookItem> bookitems;

    public String getCategoryLabel(){
        if(bookitems.size() <2){
            return bookitems.size() + " " + category;
        }else{
            return bookitems.size() + " " + category+"(s)";
        }
    }

    public String onClickAction(){
        HttpSession session = (HttpSession)JSFUtil.getFacesContext().
                             getExternalContext().getSession(true);
        session.setAttribute("CATEGORY", this);
        return "success";
    }
}

Redirecting the Web page

This section describes how to redirect a user to a new page based on the link clicked. I use JSF navigation rules to redirect the page. The onClickAction() method returns "success" to begin the action. The content of the new page is given by the data sent into HttpSession. The data will be retrieved from HttpSession by the managed bean, DetailBean, of the new page. Then DetailBean creates its GUI components accordingly.

Listing 7 shows the detailed implementation. "detail.jsp" is the new page to which the user will be redirected. getDetailgrid(), part of DetailBean in detail.jsp is bound to a method that creates dynamic elements in this page. In this method, you first obtain the category data that should be displayed and then create corresponding GUI content using the populate() method. Study populate() to see how to create dynamic GUI elements, and even page layouts, in real time. All page information is passed by category data from HttpSession, so, theoretically, data put into HttpSession determines what the new page will look like.

**Listing 7. redirect user to detailed information page**
detail.jsp

<f:view>
public class DetailBean {
    private HtmlPanelGrid detailgrid = null;
    private Category cat;
    public HtmlPanelGrid getDetailgrid() {
        if (detailgrid == null) {
            detailgrid = new HtmlPanelGrid();
        }
        detailgrid.getChildren().clear();
        HttpSession session = (HttpSession) JSFUtil.getFacesContext().getExternalContext().getSession(true);
        cat = (Category) session.getAttribute("CATEGORY");
        session.removeAttribute("CATEGORY");
        populate(detailgrid);
        return detailgrid;
    }
    public void setDetailgrid(HtmlPanelGrid detailgrid) {
        this.detailgrid = detailgrid;
    }
    private void populate(HtmlPanelGrid parent) {
        if (cat != null) {
            String category = cat.getCategory();
            ArrayList<BookItem> items = cat.getBookitems();
            if (category.equals("News paper")) {
                //create GUI for News paper category.
            } else if (category.equals("Magazine")) {
                //create GUI for Magazine category.
            } else {
                //create GUI for other categories.
            }
        }
    }
}

So far, you have learned how to update the data model and how to create dynamic GUI elements. Three aspects—how to insert elements into or remove elements from the proper place of a Web page, how to bind different behaviors to different elements, and how to redirect a Web page—are discussed in detail. Try to understand the relationships between them, and choose the parts you need for your own development scenarios.

### Refreshing the dynamic elements of the Web page using Ajax

In this section, I build a bridge between the "Bean" and "GUI" layers from Figure 2 in order to refresh the dynamic parts of the Web page. I use Ajax4jsf of RichFaces to do the refresh. RichFaces is an open-source framework that adds Ajax capability into existing JSF applications without resorting to JavaScript. Through Ajax4jsf, I can overcome the current JSF limitation of not supporting any page refresh from the server side, and I can meet the requirement of only refreshing the necessary content.
Register RichFaces

After installing RichFaces, you need to change the `web.xml` file by adding the lines from Listing 8 to register RichFaces.

Listing 8. Register RichFaces

```xml
<!-- Plugging the "Blue Sky" skin into the project -->
<context-param>
    <param-name>org.richfaces.SKIN</param-name>
    <param-value>blueSky</param-value>
</context-param>

<!-- Making the RichFaces skin spread to standard HTML controls -->
<context-param>
    <param-name>org.richfaces.CONTROL_SKINNING</param-name>
    <param-value>enable</param-value>
</context-param>

<!-- Defining and mapping the RichFaces filter -->
<filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>
<filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Changes made on the Web page

After registering RichFaces, you need to add the tags from Listing 9 into the category.jsp file to realize "reverse ajax," which is pushing data from the server side to the client side and using Ajax technology to refresh the page.

Listing 9. Push data to Web page

```xml
<f:view>
    <h:form id="helloForm">
        <a4j:region>
            <a4j:push reRender="book" eventProducer="#{InventoryBean.addListener}"/>
        </a4j:region>
        <h:panelGrid id="title">
            <h:outputText id = "hello_title" value="Inventory"/>
            <a4j:outputPanel  id = "book" binding="#{InventoryBean.categorygrid}"/>
            <h:outputText id = "summary" value="#{InventoryBean.categoryNumber}"/>
        </h:panelGrid>
    </h:form>
</f:view>
```

Look at the `a4j:push` tag. With `eventProducer="#{InventoryBean.addListener}"`, the Web page registers a listener to the managed bean so that the managed bean can refresh the Web page, if needed. `reRender = "book"` means that only a component with the ID "book" should be refreshed after the server-side data has been pushed to the page. `a4j:outputPanel` allows marking of a page area, which is updated on the Ajax response.
Changes made in the managed bean

In the managed bean, you need to register `PushEventListener` so that the server-side data will be pushed to the client side if any pushing event occurs. This method is bound to the Web page by the `eventProducer` attribute. The pushing event is generated by `this.listener.onEvent(new EventObject(this));` in the `categoryChanged()` method, which will be called every time the server-side data changes. I talked earlier about `categoryChanged()`. Listing 10 shows its detailed implementation.

Listing 10. Register eventProducer and push data

```java
public class InventoryBean implements InventoryListener{
    // ...
    public void addListener(EventListener listener) {
        synchronized (listener) {
            if (this.listener != listener) {
                this.listener = (PushEventListener) listener;
            }
        }
    }

    public void categoryChanged() {
        refresh(); //code for refresh dynamic part via ajax
        this.listener.onEvent(new EventObject(this));
    }
}
```

Now you can push Ajax refreshing from the server side. Combining this technique with previous ones, you can connect the "Database," "Bean," and "GUI" layers from Figure 2 together. Like any of the methods I have discussed, this method can be used independently in any suitable situation.

Conclusion

JSF is a convenient Web framework to generate HTML pages, receive user input, and manage navigation flow. To refresh a page in JSF, a user normally needs to execute some actions on the Web page to generate an HTTP request, which will be answered using an HTTP response that leads to a subsequent page refreshing. Any Web page change triggered by the server side is not easy in JSF. This article offers a solution, not only automatically updating Web pages based on requests from the server side, but also synchronizing the server-side data with dynamic elements of the Web page, which are created in the run time and continuously altered.
Related topic

- Read "An introduction to RichFaces" (developerWorks, March 2008) to get an introduction and learn to add desktop-like features to your browser applications.

© Copyright IBM Corporation 2009
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)