ADF Code Corner

042. How-to dynamically change the progress bar color according to its current value

twitter.com/adfcodecorner

Abstract:

A while back, a question on OTN was of how to change the color of an af:progressBar component such that it is red in the beginning, yellow in the middle and green at the end of a process. This question has a lot of grounds to cover and the solution - at least the one I could come up with - involves building a custom progress bar model, start and stop the progress bar update by showing/hiding an af:poll component and skinning. This blog article covers it all and provides you with an example to download that gives you a head start when implementing a progress bar into your custom application development

Author: Frank Nimphius, Oracle Corporation
twitter.com/fnimphiu
03-May-2010

Oracle ADF Code Corner is a loose blog-style series of how-to documents that provide solutions to real world coding problems.

Disclaimer: All samples are provided as is with no guarantee for future upgrades or error correction. No support can be given through Oracle customer support.

Please post questions or report problems related to the samples in this series on the OTN forum for Oracle JDeveloper: http://forums.oracle.com/forums/forum.jspa?forumID=83

Introduction

The af:progressBar component in the ADF Faces component set is an important widget in the category of "end user entertainments" as it helps developers to visualize a longer running background process to becalm users. The progress bar uses a managed bean model that extends the Apache Trinidad BoundedRangeModel class to determine the current state of a process based on a maximum value - like estimated row count - and a current value - for example the current row index. Based on the state of the two values, the progress is shown as a progressing bar image, or a clock which is used when the maximum value of a process cannot be determined.

For more information about the af:progressBar, please refer to the tag documentation. In this blog article, a progress bar is implemented that shows a bar item in the color red, yellow and green dependent on its current value state. The use case for this sample was a question on the OTN forum (http://forums.oracle.com/forums/forum.jspa?forumID=83) and has a lot of information to share, which makes it a good example for you to complete reading even if a progress bar is not what you are currently looking for.

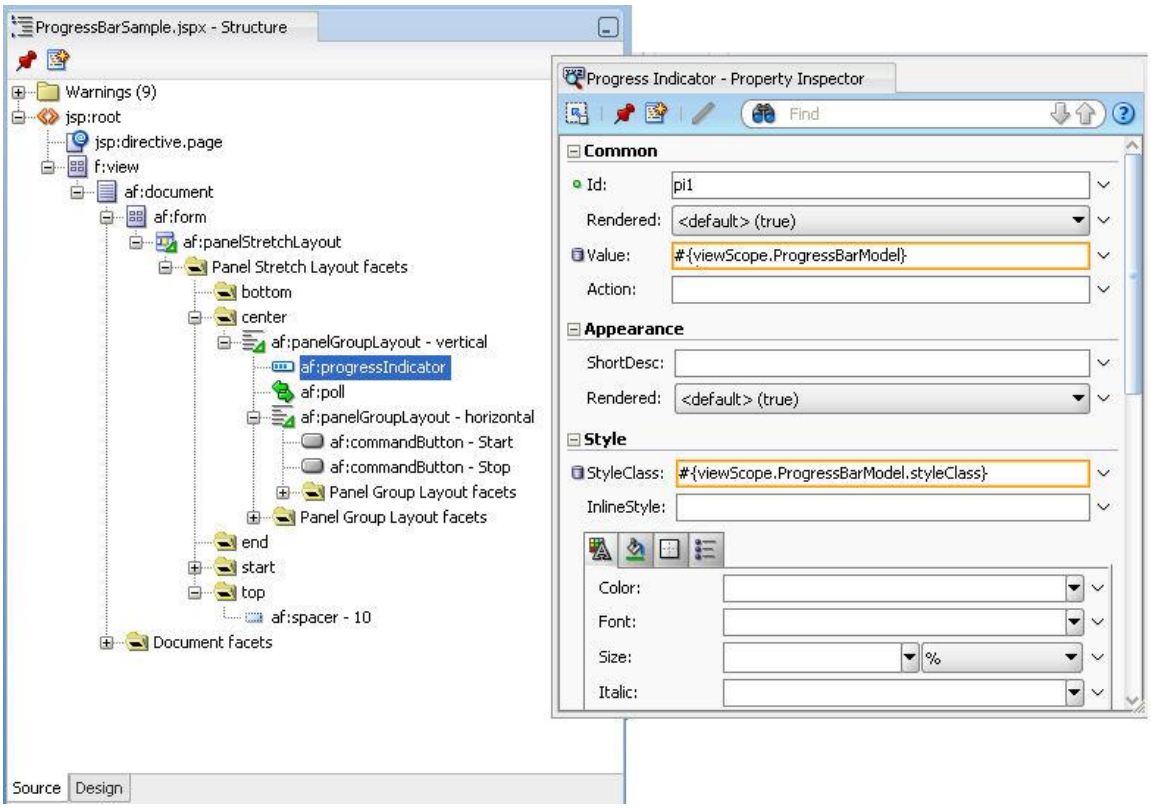
About the Sample

The sample to download at the end of this article shows a progress bar with a start and a stop button. The progress bar reads its value and state from a managed bean that uses a Java Thread to loop through a counter until a maximum value is reached.

The same managed bean exposes a getter method for the progress bar's styleClass attribute to read its value using Expression Language. Dependent on the current value of the progress bar, the returned values for the styleClass property are red, yellow and green. All three values are used as a style class , ".red", ".yellow" and ".green". in the skinning document that determines the color coding of the af:progressBar component. The progress bar is a image that gets stretched as a background image, which means that to switch colors, three images need to be provided, one in red, one in yellow and one in green. The approach I took for creating the images is to save the default image and to use my favorite picture editing software to change the color.

The ProgressBar model

The progressBar references an implementation of the Trinidad BoundedRangeModel from its value property using ExpressionLanguage. In the example, I extended BoundedRangeModel in a JavaBean and configured this bean as a JavaServer Faces managed bean in the ADF Controller configuration file. I chose the ADF Controller so that the managed bean could be stored in viewScope, which is a memory scope that is active for as long as the viewId of the current view does not change. In other words, for as long as the page or page fragment that holds the progress bar is not navigated away from, the managed bean is available. I prefer the view scope over session scope as it ensures the progress bar model is cleaned from memory when it is no longer needed, whereas using the session scope, it would be my task as a developer - and as we all know, to err' is human.



As shown in the image above, the managed bean is referenced two times, from the *value* property and the *StyleClass* property.

```
import javax.faces.event.ActionEvent;
import org.apache.myfaces.trinidad.model.BoundedRangeModel;


/**
 * The progress bar model in this example allows components
 * to use style classes .green, .yellow, .red in combination with
 * skinning to color the progress bar according to its value state.
 *
 * This bean must be in a scope larger than request to keep the state
 * across requests. The scope in this sample is set to viewScope, which
 * means that this progress bar stops when the user navigates off the
 * page.
*/
public class ProgressBarModel extends BoundedRangeModel {

  //progress bar status
  final String GREEN  = "green";
  final String YELLOW = "yellow";
  final String RED    = "red";

  //first style class color is red as progress has just started
  String styleClass = RED;
  Thread newProgress = null;
  boolean stopFlag = false;
```

```
long maximum = 10;
long value = 0;

long greenBoundary = maximum *  4/5;
long yellowBoundary = maximum * 3/5;

  public ProgressBarModel() {
      super();
  }

  public long getMaximum() {
      return maximum;
  }

  public long getValue() {
      return value;
  }

  public void setMaximum(long maximum) {
      this.maximum = maximum;
  }

  public void setValue(long value) {
      this.value = value;
  }

  //start progress count
  public void start (ActionEvent ae){
    value = 0;
    stopFlag = false;
    styleClass = RED;
    //make sure thread is properly cleared before starting a new
    //thread
    if (newProgress!=null){
      newProgress.interrupt();
    }

    ProgressUpdater progressSimulator = new ProgressUpdater();
    newProgress = new Thread(progressSimulator);
    newProgress.start();

  }

public void stop (ActionEvent ae){
  value = 0;
  styleClass = RED;
  stopFlag = true;
}

//Simulate a business service progress
class ProgressUpdater implements Runnable
{


    public void run(){
```

```
    try
     {
      //stop fag is true if it is set to true or if value
      //is equals or greater than maximum
      stopFlag = stopFlag == true?
                               true : (value < maximum? false:true);

      //run in loop until stop condition is met. Make sure system
      //doesn't fail if values are initially set to the same value
     while (!stopFlag && value != maximum) {
              Thread.sleep(1000);
              value = value +1;
              //set the color boundaries
              if (value >= greenBoundary){
               styleClass = GREEN;
              }
              else if(value >= yellowBoundary){
               styleClass = YELLOW;
              }
              else{
               styleClass = RED;
              }

              if (value == maximum){
                stopFlag = true;
              }
            }
            //stop thread
             newProgress.interrupt();
             newProgress = null;
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
      }
    };

    /**
     * method to be referenced from the StyleClass property of the
     * progress bar component. The return string in combination with
     * the skin definition sets the color of the progress bar item
     * @return
     */
    public String getStyleClass() {
        return styleClass;
    }
}
```

The maximum value can be set from an application and only by default is set to 10. Calling the start method from an action listener - like available on af:commandButton - sets all internal state to the beginning and starts the counter thread. The counter thread is what you need to change to show progress of a real business task. For example, you could set the maximum value to the outcome of a getEstimatedRowCount to then - in the Thread - access the rows in range to perform actions on. The

Thread is stopped when the maximum value is reached. In this case, the progress bar component is reset to the default - empty - bar graph.

To update the progress bar UI, an af:poll component is used, as shown in the image above. The af:poll component is referenced from the progress bar PartialTriggers property and is set to 500 ms. In a production environment you don't want a poll frequency like this and instead use something in the range of 5 seconds to avid unnecessary network traffic. So 500 ms really is for this sample so you don't have to wait 50 seconds for what could be demonstrated in 10 seconds.

The af:poll component

As mentioned, the af:poll component is used to refresh the progress bar component so the color for the current state could be shown. The af:poll component frequently and endlessly polls the server in a pre-defined frequency. To start and stop polling, you use the *rendered* property as whenever the af:poll component is not rendered, it does not poll.

```
<af:poll id="p1" interval="500"
         binding="#{viewScope.ProgressBarSampleBean.pollComponent}"
         rendered="false"/>
```

The poll component by default is set to rendered="false" so that it only starts pinging the server when the "start" button shown in the image on top is pressed. The command button for start and stop reference a second managed bean, which also is in view scope.

The responsibility of the second managed bean is to enabled and disable the rendering of the af:poll component, to refresh the af:poll component parent container - af:panelGroupLayout in this sample - and to start and stop the timer thread.

```
<af:commandButton text="Start" id="cb1" partialSubmit="true"
                  actionListener="#{viewScope.ProgressBarSampleBean
                                    .onProgressBarStart}"/>
<af:commandButton text="Stop" id="cb2" partialSubmit="true"
                  actionListener="#{viewScope.ProgressBarSampleBean
                                    .onProgressBarStop}"/>
```

The managed bean that is referenced from the af:poll component and the command buttons is shown below

```
import javax.el.ELContext;
import javax.el.ExpressionFactory;
import javax.el.MethodExpression;

import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import javax.faces.event.ActionEvent;

import oracle.adf.view.rich.component.rich.RichPoll;
import oracle.adf.view.rich.component.rich.layout.RichPanelGroupLayout;
import oracle.adf.view.rich.context.AdfFacesContext;


public class ProgressBarSampleBean {
    private RichPoll pollComponent;
    private RichPanelGroupLayout pollComponentParent;
```

```
public ProgressBarSampleBean() {
    super();
}

//methods referenced from the af:poll component "binding" property
public void setPollComponent(RichPoll pollComponent) {
    this.pollComponent = pollComponent;
}

public RichPoll getPollComponent() {
    return pollComponent;
}

//methods referenced from the af:panelGroupLayout
//component "binding" property
public void setPollComponentParent(
                  RichPanelGroupLayout pollComponentParent) {
    this.pollComponentParent = pollComponentParent;
}

public RichPanelGroupLayout getPollComponentParent() {
    return pollComponentParent;
}


/*
 * Start and stop Polling
 */

 //method called to start af:pollComponent
  public void onProgressBarStart(ActionEvent actionEvent) {
   pollComponent.setRendered(true);
   this.executeMethodExpression(
         "#{viewScope.ProgressBarModel.start}", actionEvent);
    AdfFacesContext.getCurrentInstance().addPartialTarget(
                                          pollComponentParent);}

//method called to stop af:pollComponent
public void onProgressBarStop(ActionEvent actionEvent) {
  pollComponent.setRendered(false);
  this.executeMethodExpression(
              "#{viewScope.ProgressBarModel.stop}", actionEvent);
  AdfFacesContext.getCurrentInstance().addPartialTarget(
                 pollComponentParent);
}

/**
 * helper method to invoke start/stop on the progressbar model
 * @param expression The EL to reference the BoundedRangeModel
 * implementation bean's start and stop method
 * @param actionEvent
 */
  public void executeMethodExpression(
              String expression, ActionEvent actionEvent){
     FacesContext fctx = FacesContext.getCurrentInstance();
```

```
        Application application = fctx.getApplication();
        ELContext elctx = fctx.getELContext();
        ExpressionFactory elFactory =
                              application.getExpressionFactory();
        MethodExpression me =
                 elFactory.createMethodExpression(
                           elctx,expression, null,
                           new Class[]{ActionEvent.class});
        me.invoke(elctx,new Object[]{actionEvent});
    }
}
```

Skinning

The progress ar color coding is determine by a custom skin. To implement skinning, you need to do the following:

1. Create a trinidad-skins.xml file in the WEB-INF directory of the ViewLayer project. This file defines the custom skin and extends one of the default skins - "fusion" in the example - so we don't have to start skinning all components. The content of the trinidad-skins.xml file is shown below

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<skins xmlns="http://myfaces.apache.org/trinidad/skin">
  <skin>
    <id>sample.desktop</id>
    <family>sample</family>
    <render-kit-id>org.apache.myfaces.trinidad.desktop</render-kit-id>
    <extends>fusion.desktop</extends>
    <style-sheet-name>css/sample.css</style-sheet-name>
  </skin>
</skins>
```

2. Edit the trinidad-config.xml file in the WEB-INF directory of the ViewLayer project and set the skin family name to the custom skin definition

3.
```
<?xml version="1.0" encoding="windows-1252"?>
<trinidad-config xmlns="http://myfaces.apache.org/trinidad/config">
 <skin-family>sample</skin-family>
</trinidad-config>
```

4. Create a CSS file in the directory specified in the trinidad-skins.xml file. In the example the directory is "css" and must be created under the public_html directory in the ViewLayer project. You find the kin selector to use from the skin selector documentation on OTN. Note that the three style classes .red, .yellow and .green have their own skin definition defined. The CSS content is shown below

```
/*
 sample.css file in public_html/css
 CSS that sets the progress bar image based on the current styleClass value
```

```
*/
.red af|progressIndicator::determinate-filled-icon-style{background-image:url("../images/redbar.png");}
.yellow af|progressIndicator::determinate-filled-icon-style{background-
image:url("../images/yellowbar.png");}
.green af|progressIndicator::determinate-filled-icon-style{background-
image:url("../images/greenbar.png");}
```

Download Sample

As usual, samples on ADF Code Corner go with the latest fashion, which is Oracle JDeveloper 11g R1 PS2 that is available for download since April 2010. However, this example doesn't use any specific feature of this release and therefore should work with an of the Oracle JDeveloper 11g releases so far. The workspace does no require a database connection and runs off the ProgressBarSample.jspx page. Note that randomly I experienced refresh issues on Firefox, which are a tribute to the 500 ms af:poll frequency. Pressing F5 helps in such cases. Better though is to reduce the poll frequency**. Download the sample from ADF Code Corner**

http://www.oracle.com/technetwork/developer-tools/adf/learnmore/index-101235.html

RELATED DOCOMENTATION

⊠	Progress Indicator tag doc - http://download.oracle.com/docs/cd/E15523_01/apirefs.1111/e12419/tagdoc/af_progressIndicator.html
⊠	Skinning keys - http://download.oracle.com/docs/cd/E15523_01/apirefs.1111/e15862/toc.htm
⊠	