JSF 2 fu: HTML5 composite components, Part 1

Start implementing an HTML5 component library with JSF 2

Skill Level: Intermediate

David Geary
President
Clarity Training, Inc.

12 Oct 2010

HTML5 gives browser-based applications rich features rivaling those of desktop software. In this *JSF 2 fu* installment, you'll see how you can get the best of the Java™ and HTML5 worlds by implementing an HTML5 composite component with JavaServer Faces (JSF) 2.

HTML5 is ostensibly the next big thing in software development. Originally known as Web Applications, HTML5 finally brings the power of desktop applications — complete with amenities such as drag and drop, canvases, video, and audio — to the browser. HTML5 is a collection of technologies (specifications, specifically) that form a powerful API encompassing HTML, JavaScript, and Cascading Stylesheets (CSS). Here are the HTML5 highlights:

About this series

The *JSF 2 fu* series, a follow-on to David Geary's three-article introduction of the same name, will help you develop and hone your JSF 2 framework skills like a kung fu master. The current series dives deeper into the framework and its surrounding ecosystem. And it takes a peek outside the box by showing how some Java EE technologies, such as Contexts and Dependency Injection, integrate with JSF.

- Canvas
- Drag and drop

- Geolocation*
- Inline editing
- Web workers*
- Web storage*
- Messaging
- Offline applications
- Video and audio*
- Web sockets*

Notice the forward-looking features such as geolocation and offline applications. (Features I've marked with an asterisk are not technically part of the HTML5 specification, but the term HTML5 is used colloquially to encompass all the ones I've listed. See Resources for more information.)

In some respects, HTML5 is the next Java. In the late 1990s, the Java language became immensely popular, in no small part because its *Write once, run anywhere* capability freed developers from having to choose among (or port to) Windows®, Mac, or Linux®. HTML5 lets you *Write once, run in any (modern) browser*, so you don't have to choose among iOS, Android, and Chrome.

Write once, debug everywhere?

Java technology lets you write one application for multiple operating systems, but it doesn't do so perfectly. Neither does HTML5. HTML5 doesn't offer some of the amenities that native OSs provide, such as interacting with an accelerometer. (although there are toolkits — such as PhoneGap [see Resources] — that bridge the gap). Those imperfections will always cause some developers to eschew HTML5 in favor of native apps. But for many apps, HTML5 provides a better return on investment.

Enter Java technology

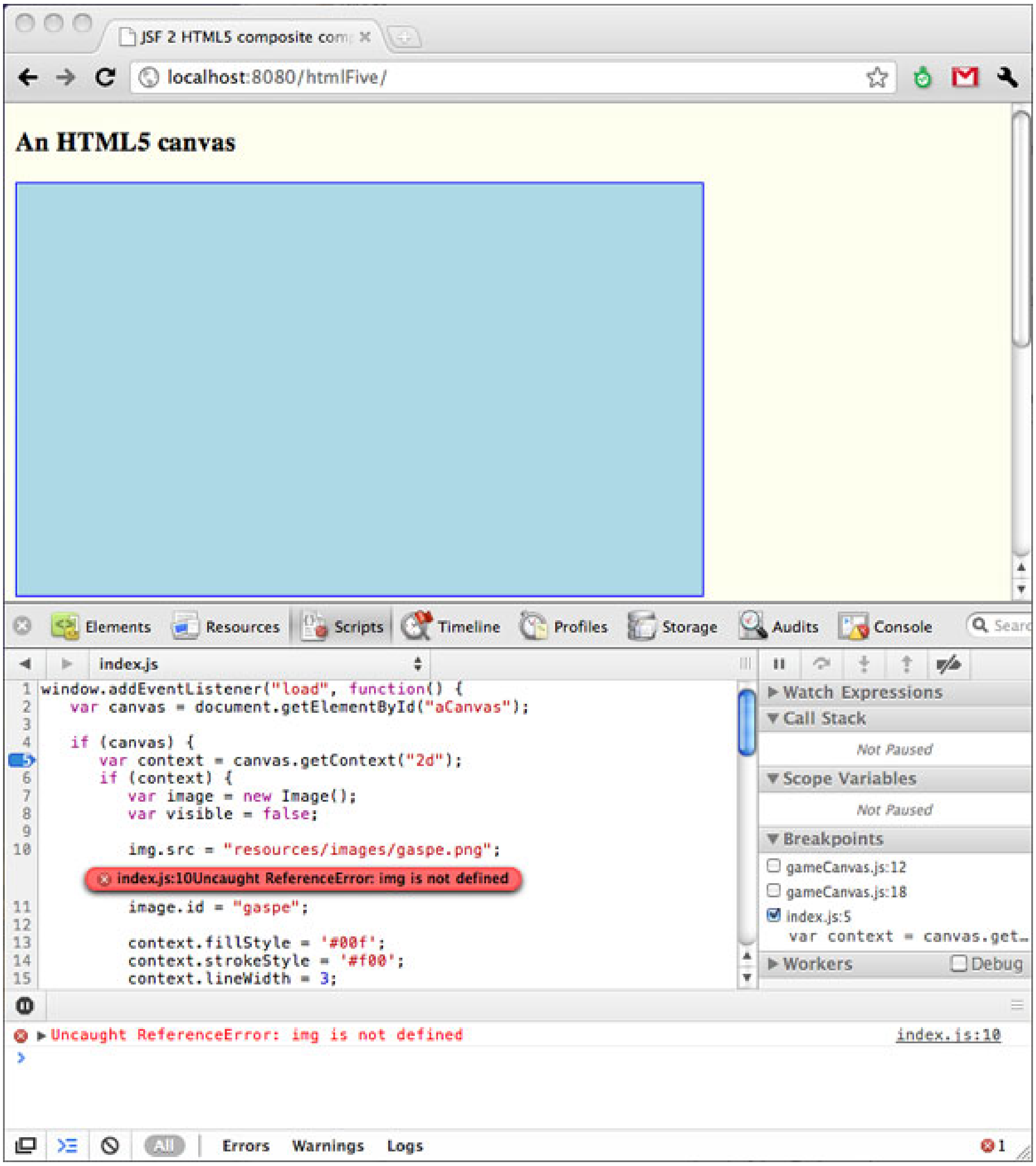
HTML5 may be the next Java, but it won't replace it. Java technology provides a rich ecosystem for server-side programming. And JSF, which is based on HTML to begin with, lets you use HTML5 as easily as you've been using HTML4 with JSF up till now. You get all the powerful JSF features, such as facelets templates, composite components, and built-in Ajax, in addition to HTML5.

In this article, you'll learn how to create HTML5 composite components with JSF2. In the next *JSF 2 fu* article, I'll show you how to create a library of HTML5 components.

Getting started with HTML5

Using HTML5 actually involves a lot more JavaScript than HTML. That means you need a good JavaScript debugger. I recommend the one that comes with Google Chrome's built-in Developer Tools (see Resources), shown in Figure 1:

Figure 1. Debugging JavaScript with Chrome Developer Tools



```
window.addEventListener("load", function() {
   var canvas = document.getElementById("aCanvas");

   if (canvas) {
      var context = canvas.getContext("2d");
      if (context) {
         var image = new Image();
         var visible = false;

         img.src = "resources/images/gaspe.png";
  index.js:10Uncaught ReferenceError: img is not defined
         image.id = "gaspe";

         context.fillStyle = '#00f';
         context.strokeStyle = '#f00';
         context.lineWidth = 3;
```

```
gameCanvas.js:12
gameCanvas.js:18
index.js:5
  var context = canvas.get…
```

```
Uncaught ReferenceError: img is not defined                    index.js:10
```

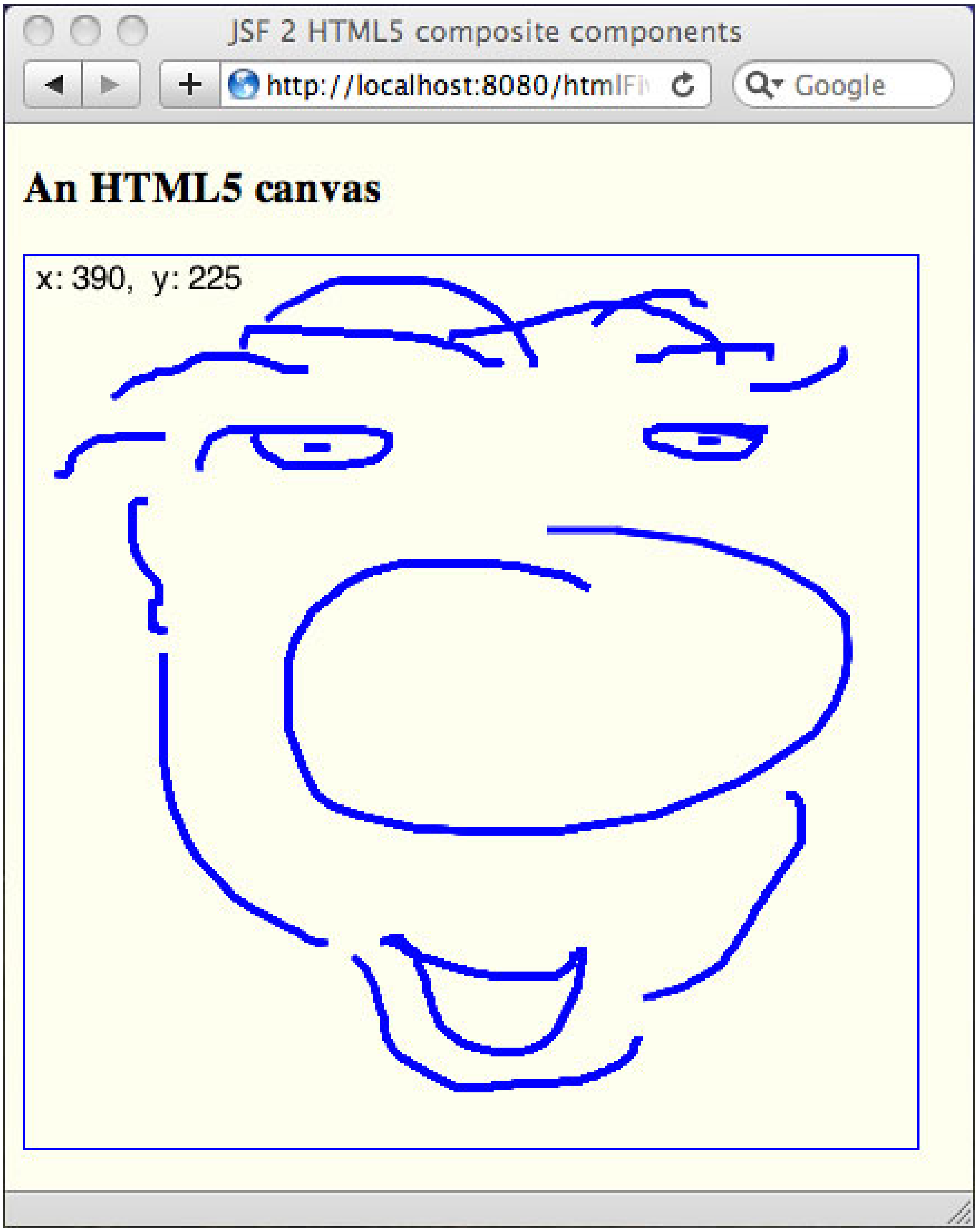
In the Chrome debugger shown in Figure 1, a panel containing the JavaScript code appears below the displayed canvas component.

Now that you have a good JavaScript debugger, you just need a browser that is HTML5-capable. Most of the latest versions of the more popular browsers do a good job of supporting HTML5. (Microsoft appears to have good HTML5 support in the soon-to-be-released Internet Explorer 9.)

Using the HTML5 canvas

The HTML5 canvas is a full-fledged 2D drawing surface that is rich enough to support games such as *Plants vs. Zombies* and *Quake II*. My use of the HTML5 canvas, shown in Figure 2, is probably not as compelling, but it will suffice for instructive purposes:

Figure 2. A simple HTML5 canvas example

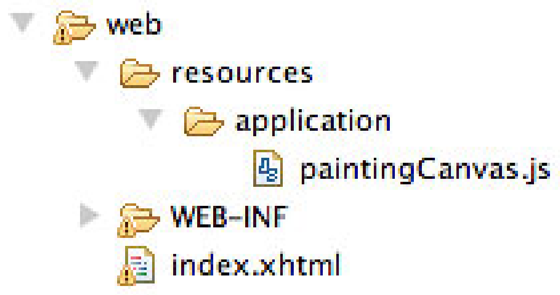

JSF 2 HTML5 composite components
http://localhost:8080/htmlFi
Google
x: 390, y: 225

An HTML5 canvas

I've added some JavaScript to an HTML5 canvas to implement the simple paint application shown in Figure 2. As you move the mouse, the readout in the upper left-hand corner of the canvas shows the mouse coordinates. When you drag the mouse in the canvas, you paint with a blue brush.

The application shown in Figure 2 is a JSF application. Figure 3 shows its directory structure:

Figure 3. Directory structure for the canvas example



The application's lone facelet is defined in web/WEB-INF/index.xhtml, and the application's JavaScript is in web/resources/application/paintingCanvas.js. Listing 1 shows index.xhtml:

Listing 1. Using the <canvas> tag (WEB-INF/index.xhtml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html">

   <h:head>
      <title>#{msgs.windowTitle}</title>
   </h:head>

   <h:body style="background: #fefeef">
      <h:outputScript library="application" name="paintingCanvas.js"
                       target="head" />

      <h3>#{msgs.heading}</h3>


      <canvas width="400px" height="400px" id="paintingCanvas">

         Canvas not supported.

      </canvas>
   </h:body>

</html>
```

The index.xhtml file in Listing 1 is known as an HTML5 *polyglot document* (see Resources), because it has an HTML5 doctype/namespace and well-formed XHTML syntax — exactly what my brew of facelets and HTML5 requires.

I import the corresponding JavaScript with the `<h:outputScript>` tag. Finally, I dive in and use the HTML5 canvas element. If the browser doesn't understand the `<canvas>` tag, it will show a *Canvas not supported* message. The `<canvas>` tag is unassuming; all of the interesting code is in the corresponding JavaScript, shown in Listing 2:

Listing 2. Painting canvas JavaScript (resources/application/paintingCanvas.js)

```
window.addEventListener("load", function() {
   var canvas, context, painting;

   function init() {
      canvas = document.getElementById("paintingCanvas");
      if (canvas == null) return;

      context = canvas.getContext("2d");
      if (context == null) return;

      painting = false;

      context.strokeStyle = "#00f";
      context.lineWidth = 3;
      context.font = "15px Helvetica";
   }

   init();

   canvas.addEventListener("mousedown", function(ev) {
      painting = true;
      context.beginPath();
      context.moveTo(ev.offsetX, ev.offsetY);
   }, false);

   canvas.addEventListener("mousemove", function(ev) {
      updateReadout(ev.offsetX, ev.offsetY);

      if (painting) {
        paint(ev.offsetX, ev.offsetY);
      }
      function updateReadout(x, y) {
         context.clearRect(0, 0, 100, 20);
         context.fillText("x: " + x + ",  y: " + y, 5, 15);
      }
      function paint(x, y) {
         context.lineTo(ev.offsetX, ev.offsetY);
         context.stroke();
      }

   }, false);

   canvas.addEventListener("mouseup", function() {
      painting = false;
      context.closePath();
   }, false);


  }, false);
```

Running the sample code

The code for this series is based on JSF 2 running in an enterprise container, such as GlassFish or Resin. See the first series installment, "*JSF 2 fu*: Ajax components" for a step-by-step tutorial on installing and running the code for this series with GlassFish. See Download to get the sample code for this article.

Listing 2 implements simple painting with a mouse cursor readout, as shown in Figure 2. When the page loads, I get a reference to the canvas with `document.getElementById()`. From the canvas, I get a reference to the canvas's context. I use that context in subsequent event handlers, which I implement using JavaScript closures, or what Java developers would refer to as anonymous inner classes.

If you've used the Abstract Window Toolkit (AWT), the canvas's context will be immediately reminiscent of the AWT's graphics context. After all, there are only so many ways to draw shapes, images, and text in two dimensions. In Listing 2, I initialize the context with a *stroke style* of blue, and set the line width and font. From then on, it's just a matter of move, stroke, repeat, when the mouse goes down, drags, and goes up, respectively.

Now that you've got the HTML5 canvas basics down, I'll show you how to create a JSF 2 HTML5 composite component.

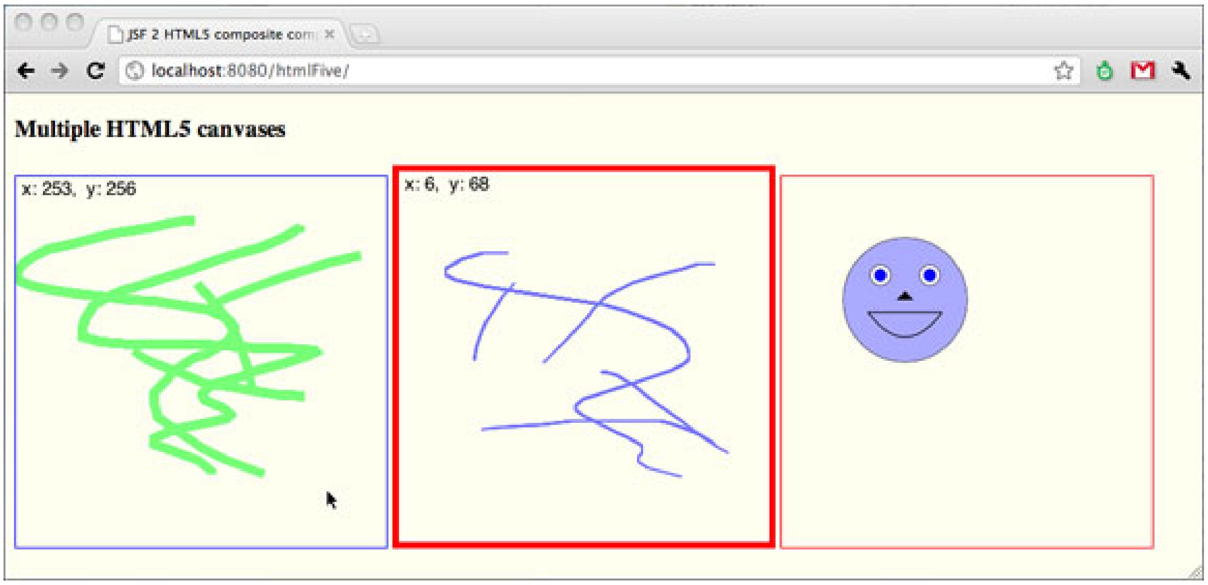
A JSF 2 HTML5 canvas component

Next, I'll implement a JSF 2 composite component that uses an HTML5 canvas. I need the composite component to meet these requirements:

- Have configurable (via tag attributes) width, height, pen color, line width, and CSS style
- Use the body of the component tag as the *canvas not supported* message
- Automatically include the canvas's JavaScript
- Support multiple canvas components in a single page

An application that uses the canvas composite component is shown in Figure 4:

Figure 4. The canvas composite component in action



The application shown in Figure 4 has three canvas components, each configured differently. Starting from the left, the first two are painting canvases, similar to the one shown in Figure 2. The right-most canvas simply paints a smiley face.

Listing 3 shows the markup for the page shown in Figure 4:

Listing 3. Using the canvas composite component (WEB-INF/index.xhtml)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:h="http://java.sun.com/jsf/html"
   xmlns:h5="http://java.sun.com/jsf/composite/html5">

   <h:head>
      <meta charset="UTF-8" />
      <title>#{msgs.windowTitle}</title>
   </h:head>

   <h:body style="background: #fefeef">

      <h3>#{msgs.heading}</h3>

      <h5:canvas id="paintingCanvas" width="300px" height="300px"
                 penColor="#7F7" lineWidth="7">

         #{msgs.canvasUnsupported}

      </h5:canvas>

      <h5:canvas id="secondPaintingCanvas" width="300px" height="300px"
                 style="border: thick solid red">

         #{msgs.canvasUnsupported}

      </h5:canvas>

      <h5:canvas id="smileyCanvas" library="application" script="smiley.js"
                 width="300px" height="300px"
                 style="border: thin solid red">
         #{msgs.canvasUnsupported}

      </h5:canvas>
   </h:body>

</html>
```

In Listing 3, the canvas component imports the appropriate JavaScript — unlike Listing 1, where I use HTML5 by hand and must import the associated JavaScript explicitly. Page authors can use the canvas component's optional `library` and `script` attributes to specify a canvas's JavaScript, or they can rely on the default JavaScript. In Listing 3, I use the `script` attribute for the smiley canvas. I use the default JavaScript (resources/html5/canvasDefault.js, which implements the painting canvas) for the two left-most canvases in the example.

Listing 4 shows the implementation of the canvas composite component:

Listing 4. The canvas composite component (resources/html5/canvas.xhtml)

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:composite="http://java.sun.com/jsf/composite">

  <composite:interface>
    <composite:attribute name="id"/>
    <composite:attribute name="width"/>
    <composite:attribute name="height"/>
    <composite:attribute name="library"  default="html5"/>
    <composite:attribute name="script"   default="canvasDefault.js"/>
    <composite:attribute name="style"    default="border: thin solid blue"/>
    <composite:attribute name="penColor" default="#7777FF"/>
    <composite:attribute name="lineWidth" default="2"/>
  </composite:interface>

  <composite:implementation>
      <canvas id="#{cc.id}"
           width="#{cc.attrs.width}"
          height="#{cc.attrs.height}"
           style="#{cc.attrs.style}">

      <composite:insertChildren/>

     </canvas>

    <h:outputScript library="#{cc.attrs.library}"
                       name="#{cc.attrs.script}"/>

    <script>
      #{cc.attrs.script}.init('#{cc.id}',
                              '#{cc.attrs.penColor}',
                              '#{cc.attrs.lineWidth}')
    </script>
  </composite:implementation>
</html>
```

In Listing 4, I declare eight attributes for the canvas composite component, five of which have default values. The component's implementation contains an HTML5 canvas, with ID, width, height, and style configured from the component's relevant attributes. That takes care of my first requirement (configurable attributes) for the canvas component.

The canvas composite component inserts its *children* — meaning anything in the body of the `<h5:canvas>` tag — into the HTML5 canvas tag (`<canvas>`). That means any text in the body of the tag will be displayed if a browser doesn't support HTML5 canvas. That takes care of the second requirement (use the body of the component's tag for the *canvas not supported* message).

The canvas component contains an `<h:outputScript>` tag that imports the canvas's JavaScript, which is specified with the canvas component's `library` and `script` attributes. Notice that the `library` and `script` attributes default to `html5` and `canvasDefault.js`, respectively. That takes care of the third requirement (automatically import the canvas's JavaScript).

Finally, the canvas component invokes a JavaScript method named `init()`, passing the canvas's ID, pen color, and line width. The `init()` method obtains and initializes the canvas's context. Notice I said *method*, not *function*, because the `init()` method belongs to an object. That object's name is derived from the canvas component's `script` attribute. For example, for the smiley canvas in Listing 3, I specify a `script` value of `smiley.js`, so the smiley canvas component will call `smiley.js.init()` — the `init()` method of an object named `js`, contained in an object named `smiley`. If you don't specify the `script` value explicitly, it defaults to `canvasDefault.js`, so the JavaScript method would be `canvasDefault.js.init()`. Calling those methods instead of global functions lets me fulfill the fourth requirement: support multiple canvases in a single page.

The default JavaScript for the canvas component is shown in Listing 5:

Listing 5. The default canvas JavaScript (resources/html5/canvasDefault.js)

```
if (!canvasDefault) var canvasDefault = {}

if (!canvasDefault.js) {
  canvasDefault.js = {
    init : function(canvasId, penColor, lineWidth) {
      var canvas, context, painting;

      canvas = document.getElementById(canvasId);
      if (canvas == null) {
        alert("Canvas " + canvasId + " not found")
      }

      context = canvas.getContext("2d")
      if (context == null)
        return;

      painting = false;

      context.strokeStyle = penColor
      context.lineWidth = lineWidth
      context.font = "15px Helvetica"

      canvas.addEventListener("mousedown", function(ev) {
        painting = true
        context.beginPath()
        context.moveTo(ev.offsetX, ev.offsetY)
      }, false)

      canvas.addEventListener("mousemove", function(ev) {
        updateReadout(ev.offsetX, ev.offsetY)

        if (painting) {
          paint(ev.offsetX, ev.offsetY)
        }
        function updateReadout(x, y) {
          context.clearRect(0, 0, 100, 20)
          context.fillText("x: " + x + ",  y: " + y, 5, 15)
        }
        function paint(x, y) {
          context.lineTo(ev.offsetX, ev.offsetY)
          context.stroke()
        }

      }, false)

      canvas.addEventListener("mouseup", function() {
        painting = false
        context.closePath()
      }, false)
    }
  }
}
```

In Listing 5, I create an object named `canvasDefault`, containing an object named `js`, which contains an `init()` method. I do that to *namespace* the `init()` method, so it doesn't get overridden by another global `init()` function. That way, I can have multiple canvases in a single page, all of which have their own implementations of an `init()` function.

Listing 6 shows the JavaScript for the smiley canvas:

Listing 6. The smiley canvas JavaScript (resources/application/smiley.js)

```
if (!smiley) var smiley = {}

if (!smiley.js) {
  smiley.js = {
    init : function(canvasId, penColor, lineWidth) {
      var canvas, context

      canvas = document.getElementById(canvasId);
      if (canvas == null) {
        alert("Canvas " + canvasId + " not found")
      }

      context = canvas.getContext("2d");
      if (context == null)
        return

      // smiley face code originally downloaded
      // from thinkvitamin.com

      // Create the face

      context.strokeStyle = "#000000";
      context.fillStyle = "#AAAAFF";
```

```
        context.beginPath();
        context.arc(100,100,50,0,Math.PI*2,true);
        context.closePath();
        context.stroke();
        context.fill();

        // eyes
        context.strokeStyle = "#000000";
        context.fillStyle = "#FFFFFF";
        context.beginPath();
        context.arc(80,80,8,0,Math.PI*2,true);
        context.closePath();
        context.stroke();
        context.fill();

        context.fillStyle = "#0000FF";
        context.beginPath();
        context.arc(80,80,5,0,Math.PI*2,true);
        context.closePath();
        context.fill();

        context.strokeStyle = "#000000";
        context.fillStyle = "#FFFFFF";
        context.beginPath();
        context.arc(120,80,8,0,Math.PI*2,true);
        context.closePath();
        context.stroke();
        context.fill();

        context.fillStyle = "#0000FF";
        context.beginPath();
        context.arc(120,80,5,0,Math.PI*2,true);
        context.closePath();
        context.fill();

        // nose
        context.fillStyle = "#000000";
        context.beginPath();
        context.moveTo(93,100);
        context.lineTo(100,93);
        context.lineTo(107,100);
        context.closePath();
        context.fill();

        // smile
        context.strokeStyle = "#000000";
        context.beginPath();
        context.moveTo(70,110);

        context.quadraticCurveTo(100,150,130,110);
        context.closePath();
        context.stroke();
      }
    }
  }
```

Listing 6 adheres to the same namespacing convention that I used in Listing 5. Subsequent JavaScript for other types of canvases must follow the same convention.

Conclusion

In this article, I've introduced you to HTML5 and shown you how to implement a JSF

2 HTML5 canvas composite component that makes it easy for JSF developers and page authors to use HTML5 canvases. I showed you how to pass information, obtained from the JSF expression language, to the JavaScript associated with a composite component, and how to namespace JavaScript functions so that identically named functions do not clobber each other. In the next installment of *JSF 2 fu*, I'll show you how to implement another HTML5 component, and how to put multiple HTML5 components in a reusable library that you can distribute to other developers in a JAR file.

Downloads

Description	Name	Size	Download method
Sample code for this article	j-jsf2fu-1010.zip	49KB	HTTP

Information about download methods

Resources

Learn

- Exploring HTML5 with JavaServer Faces 2.0: Check out this slide presentation by Roger Kitain, the JSF co-spec lead.
- HTML5Rocks: Google's HTML5 site offers tutorials and other HTML5 resources.
- HTML5 tag reference: W3schools documents HTML5's elements, including the canvas element.
- The HTML5 Specification: The official specification for the HTML5 standard — still in progress as of September 2010 — is here.
- HTML5 Tutorial: This site publishes numerous HTML5 tutorials.
- JavaScript DOM: Javadoc-like documentation of the JavaScript DOM.
- "XHTML5 in a nutshell" (Sergey Mavrody, The WHATWG Blog, July 2010): Read about polyglot HTML5.
- PhoneGap: A JavaScript framework for building cross-platform mobile apps.
- HTML5 + Quake II: See Quake II implemented using GWT and HTML5.
- Google Chrome Developer Tools: The Google Chrome Developer Tools include a JavaScript debugger.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Discuss

- Get involved in the My developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

David Geary

Author, speaker, and consultant David Geary is the president of Clarity Training, Inc., where he teaches developers to implement Web applications using JSF and Google Web Toolkit (GWT). He was on the JSTL 1.0 and JSF 1.0/2.0 Expert Groups, co-authored Sun's Web Developer Certification Exam, and has contributed to open source projects, including Apache Struts and Apache Tiles. David's *Graphic Java Swing* was one of the best-selling Java books of all time, and *Core*

JSF (co-written with Cay Horstman), is the best-selling JSF book. David also speaks regularly at conferences and user groups. He has been a regular on the NFJS tour since 2003, is a three-time Java University instructor, and a three-time JavaOne Rock Star.

Trademarks