The Flying Saucer User's Guide

Getting Started with Flying Saucer

Release R6

May 2006

Table of Contents

An Introduction to Flying Saucer

What It Is

Flying Saucer is a *renderer*, which means it takes XML files as input, and generates a rendered representation of that XML as output. The output may go to the screen (in a GUI), to an image, or to a PDF file. Because we believe most people will rely on conventional practices, our main target for content is XHTML, an XML document format that standardizes HTML. However, we accept any well-formed XML for rendering as long as CSS is provided that tells us how to lay it out. In the case of XHTML, default stylesheets are provided out of the box.

Internally, Flying Saucer works with an XML document and uses CSS to determine how to lay it out visually. The rules for layout come from the CSS 2.1 Specification, and according to that spec, element nodes and attributes are matched to CSS selectors, where each selector identifies some formatting rules. We can't cover how to use CSS here—it's a long and complex specification—but there are many good books available, and tutorials on the web. Check out the W3Schools CSS Tutorial for a starting point.

What It Does

Flying Saucer takes XML and CSS as input (where the CSS might be embedded in the document, or linked from it) and generates rendered content. Our current major output formats are in a GUI interface (a panel) and in PDF; we can also render to image formats, e.g. render the page and save as an image.

If rendering to a GUI, hyperlinks work so you can navigate between pages. As with HTML, you can also render forms, capture output, and create applications that way. In a GUI, Flying Saucer provides a *read-only* view of the output; we cannot replace a text area, say, or Swing's `JEditorPane` or `JTextPane`. However, for static content, or content created by you, Flying Saucer can be used for help documents, tutorials, books, splash screens, presentations, and much more.

We can also render to PDF. For PDF, the layout rules come from the CSS. The difference is the rendered output uses the iText library to generate PDF.

Last, we have utility classes to render output to an image file. With this, you could use XML and CSS to layout printable content—for example, a flyer, a poster, business cards, etc.—and save them as images you can print out or email. It's also a nice way to create thumbnail or reduced-size images of pages.

What It's Good For

Flying Saucer can be used for any Java application that requires some sort of styled text. This can be as simple as a chat program or as complicated as a complete ebook reader with dynamic stylesheets. Flying Saucer is very forward thinking and is designed to be a part of the next generation of applications that combine rich desktop clients with web content. Some examples of application types are:

- chat programs
- online music stores
- a Gutenberg eBook reader
- a distributed dictionary application
- Sherlock style map and movie search
- Konfabulator and Dashboard components
- an RSS reader
- a Friendster client
- an eBay client
- a splash screen
- an about box
- a helpfile viewer
- a javadoc viewer
- report generation and viewing
- a stock ticker / weather reporter

Where the Saucer Does not Fly (what it can't do)

Being honorable people, we must admit what Flying Saucer cannot do for you:

- It cannot be used for user-editable content; output is read-only.
- We render well-formed XML; XHTML is a well-formed document standard. We can't render most HTML "in the wild". At best, you can "clean up" old HTML with TagSoup or JTidy and hope for the best. But without a bunch of work, you won't be able to use Flying Saucer for a real web browser component. However, note that's not a technical limitation, just a lack of time and resources.
- HTML plugins, like applets, Flash programs, etc. are not supported.
- Scripting (e.g. JavaScript) is not supported. We ignore script tags.
- Dynamic changes to the content requires a re-layout (quick, but noticeable), that is, you can't dynamically change the DOM and see results live.
- Most DOM callbacks used in JavaScript are not yet implemented (onLoad, onClick, onBlur, etc.).

These limitations all have a pragmatic origin. Josh Marinacci, the founder of and original lead developer for the Flying Saucer project, realized that writing a fully capable HTML browser component (like Firefox's Gecko engine) could take many man-years of development. But if one focused on well-formed XHTML only, and stuck to the CSS spec, you could cover most of the useful stuff you want to do with a rendering engine, and do it in a reasonable amount of time. So it's not impossible to

add scripting, DHTML, plugins to Flying Saucer, we've just deferred this until someone has the time and energy to get it to work—that way, we stay focused on the goal, which is pure CSS 2.1 support for well-formed XML.

Of course, you can help fix any of these things. Contributors welcome!

License and Dependencies

Flying Saucer itself is licensed under the GNU Lesser General Public License. You can use Flying Saucer in any way you want as long as you respect the terms of the license. A copy of the LGPL is provided under LGPL.txt in our distribution.

Flying Saucer uses a couple of FOSS packages to get the job done. A list of these, along with the license they each have, is listed in the LICENSE file in our distribution.

Requirements for Running and Using Flying Saucer

Flying Saucer is built and tested on Java 1.4 and has some dependencies on libraries only available in 1.4. In principle, you should be able to backport it to 1.3 (or earlier?), but we've not tried that and don't maintain a 1.3 branch.

Basic requirements:

- Java Runtime Environment 1.4 or above (or JDK of course)
- core-renderer.jar (our distribution)
- CSS Parser Our distribution includes a modified version, redistributed as allowed by the CSS Parser license
- iText (also at iText PDF, distributed unmodified
- Ant if you want to build from source
- JUnit if you want to run unit tests on source

iText is not necessary at runtime if you are not generating PDFs, but is necessary for the build to satisfy compile-time dependencies.

In theory, you could substitute another CSS parser that is SAC (Simple API for CSS) compliant, but we've had little success with this due to variations in implementation.

Most of Flying Saucer does not rely on advanced Java features. It should be usable on alternate Java implementations, such as GNU Classpath.

Setting your Classpath

You only need the `core-renderer.jar` and the `cssparser-0-9-4-fs.jar` in your CLASSPATH. If you want PDF output, add `itext-paulo-155.jar`. If you want anti-aliasing using the Minium toolkit, add `minium.jar`. That is all you need for your own programs. You also need an XML parser to be in your classpath, but this already included in recent versions of the JRE. To run the browser or use any of it's support classes you will need the `browser.jar` file.

To summarize, the easiest CLASSPATH to set is:

- `core-renderer.jar` (required)
- `cssparser-0-9-4-fs.jar` (required)
- `itext-paulo-155.jar`
- `minium.jar`

Using Flying Saucer

Basic Usage

- Rendering to a Swing Panel
- Showing an About Box

To make life easier for our end-users, we have created a special Java package, `org.xhtmlrenderer.simple`, which contains classes you can use to get up and running without any hassle.

In addition, in a separate branch of our source tree, we created some sample single-class Java programs to show different uses of Flying Saucer.

To understand where to start, you have to look at how Flying Saucer works. The input is a document, identifed by a Uniform Resource Identifier (URL) or Uniform Resource Locator (URI). That document must be well-formed XML. The document is loaded and elements are matched against the CSS provided for the document. For XHTML, we have specifications for how CSS is specified—as linked stylesheets, as inline styles, and as style attributes. For XML, we support linked stylesheets via the xml-stylesheet processing instruction.

Once we've matched CSS, we run through a layout phase, where we calculate the size and position, as well as display attributes, of all visible elements. The layout is then used to render to some output sink. The standard output sink is a Swing `JPanel` subclass we call `org.xhtmlrenderer.swing.BasicPanel`, or its extended (and more powerful) child, `XHTMLPanel`.

Rendering to a Swing Panel

In fact, to make it really easy, both `org.xhtmlrenderer.swing.BasicPanel`, and its child `org.xhtmlrenderer.simple.XHTMLPanel`, allow you to set the document in one call. In fact, to display a page in a Swing `JFrame`, the code is very simple. Take a look at out our `SinglePageFrame` example in the samples directory. The important stuff happens in the `run()` method.

```
private void run(String[] args) throws Exception {
    loadAndCheckArgs(args);
    //
    // Create a JPanel subclass to render the page
    //
    XHTMLPanel panel = new XHTMLPanel();
    //
    // Set the XHTML document to render. We use the simple
    // of the API call, which uses a File reference. There
    // are a variety of overloads for setDocument().
```

```
    //
    panel.setDocument(new File(fileName));
    //
    // Put our panel in a scrolling pane. You can use
    // a regular JScrollPane here, or our FSScrollPane.
    // FSScrollPane is already set up to move the correct
    // amount when scrolling 1 line or 1 page
    //
    FSScrollPane scroll = new FSScrollPane(panel);
    JFrame frame = new JFrame("Flying Saucer Single Page D
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(scroll);
    frame.pack();
    frame.setSize(1024, 768);
    frame.setVisible(true);
}
```

The basic process is:

- create a `BasicPanel` or `XHTMLPanel` instance
- add it to a Swing `JScrollPane` or an `FSScrollPane` (unless your pages will fit without scrolling
- add the panel to a container—a `JFrame`, another panel, etc.
- call `setDocument()` to load and render your document

That's it! You can now display XHTML and CSS in your Swing applications.

What about AWT or alternate GUI toolkits for Java? Our basic rendering routine writes to a "renderer". Right now we support the concept of rendering to an *output device*. We have two output device implementations: one for Java2D (essentially a canvas, or `Graphics2D` instance) and the other for PDF (using iText). When we render to a Swing panel, we are still painting on a Java2D canvas, so in principle, you should be able to port this rendering to other 2D output surfaces. If you do, we'd like to hear from you! Just note there is no explicit technical limitation that forces you to use Swing—it's just easy, and easy to make it look good.

Showing an About Box

The AboutBox is a prefab component which displays an XHTML document and automatically scrolls it. It is primarily useful for Help->About menu items and splash screens.

Here is an example of adding an about box to a menu item's action listener.

```
import org.xhtmlrenderer.demo.aboutbox.AboutBox;
......
JMenuItem about = new JMenuItem("About...");
about.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
```

```
        AboutBox ab = new AboutBox("About Flying Saucer",
            "demos/about/index.xhtml");
        ab.setVisible(true);
    }
});
```

Creating PDF Files

In release R6, we added the ability to generate PDF files from XML/CSS input. This means that just by starting with XHTML and CSS, you can create portable PDF documents that will are readable by the standard Adobe Acrobat Reader

PDF files are treated as *paged* media, as defined by the CSS 2.1 Specification, in the section Paged media. This means that some CSS attributes that apply to paged media (as opposed to visual media, like a browser) are used to control PDF output. Flying Saucer supports the `@page` rule, which means that page size, page margins and page break controls are all supported for PDF output.

Questions and answers about using Flying Saucer for PDF output:

- How do you control page size?
- How do you control page size on PDF output?
- How do you control page margins on PDF output?
- What controls pagination?
- What about PDF bookmarks?
- What about embedded images? Are images downscaled?
- Does Flying Saucer support PDF form components?

How do you control page size?

What CSS attributes correspond to "page size" (e.g. letter, legal, A4) in CSS and XHTML?

The size property as documented in the CSS3 Paged Media module. Everything in the spec is implemented except auto page handling (the default stylesheet currently sets letter sized paper with a half inch margin)

How do you control page size on PDF output?

What CSS attributes correspond to "page size" as we understand that in a word processor, e.g. US Letter, Legal, or A4?

CSS 2.1 does not support a page size output. Although Flying Saucer currently targets the 2.1 spec, in this case we brought in a CSS3 property, *size*. You specify this as part of the `@page` rule.

```
@page {
 size: 8.5in 11in;
}
```

or

```
@page {
 size: letter;
}
```

It's described in more details in the CSS3 specification.

How do you control page margins on PDF output?

What CSS attributes correspond to "margin" as we understand that in a word processor, e.g. left and right margin of 1inch? is this padding or margin on the `body` *element?*

You can set margin, padding, and border in a @page rule (also part of the CSS3 Paged Media module) i.e.

```
@page {
 margin: 1in;
}
```

`:first`, `:right`, `:left` pseudo-pages are supported. CSS3 named pages are not supported in release R6.

For purposes of pagination, there's nothing special about `<body>` (e.g. if `<body>` spans 20 pages, your top and bottom margins will appear on pages 1 and 20 respectively).

What controls pagination?

Is there a default pagination (whatever fits in the renderable page boundaries)—but then what is a "page" size? how can I (in the current code) implement a forced break? which page-break... does Flying Saucer support right now?

Flying Saucer supports all of the CSS page-break properties.

The only limitation is that page-break-before/after: avoid only considers siblings vs. all margins which meet at that location (as the spec dictates).

If a rule cannot be satisfied (e.g. a prex. `<div style="page-break-inside: avoid;">` spans three pages), the rule is simply dropped as if it never existed.

With the exception of relatively positioned inline content, positioned/floated content will paginate just like content in the normal flow.

What about PDF bookmarks?

For PDF, what sorts of PDF-specific things does Flying Saucer support, e.g. do bookmarks work? is there support for TOCs, footnotes?

In release R6, Flying Saucer supports bookmarks. TODO: I'll send an example later, but
basically you define the bookmark structure in `<head>` and then define
internal links in the document itself to establish a target.

The iText library has a really dizzying array of features so we can get a lot
stuff for free just by providing the XML "API" for it (see `AcroForms` comment
below).

What about embedded images? Are images downscaled?

Are referenced images altered when embedded in the course of generating PDF?

No. PDF has its own way of representing image data, but no image fidelity is lost and the image isn't otherwise modified (e.g. GIFs are stored in a compressed, lossless format; the size of a JPEG on disk will be the same size as the embedded image in the PDF).

For intrinsic width/height calculations we assume a resolution of 96 DPI, but setting an explicit width/height makes it possible to use an arbitrary DPI.

Does Flying Saucer support PDF form components?

What happens with form components when generating a PDF? Is this supported at all (if I can't a non-editable form in my PDF output, say, printable form for handwritten entry?

Replaced elements are @OutputDevice@-specific. The PDF renderer doesn't use Graphics2D. At this point, an `<input>` element will be treated like regular content. Adding AcroForm support is high on the list of priorities for the Flying Saucer team.

Rendering to an Image

Graphics2DRenderer

Sample Applications

- The Browser
- The About Box
- Eeze

The Browser

The About Box

Eeze

Configuration

The Flying Saucer Configuration File

The renderer works with a simple, `java.util.Properties` -based configuration system—no XML! Our `org.xhtmlrenderer.util.Confuration` class loads properties on first access and makes them available at runtime.

When you are using the renderer, `Configuration` needs to know where to find the properties file. If you are running from the renderer JAR file, our default properties will be read from there. If you have unpacked, or re-packed, the JAR, the location of the file is currently hard-coded as /resources/conf/xhtmlrenderer.conf. This path must be on the CLASSPATH as it is loaded as a system resource using a ClassLoader. You need to add the parent directory for /resources to your classpath, or include /resources in your JAR with no parent directory.

You can change the default properties for the application right in the .conf file. However, this is not a good idea, as you will have to merge your changes back on new releases. Plus, it will make reporting bugs more complicated. Instead, you can use one of two override mechanisms for changing the value of individual properties.

Override with Second Conf File

Your override file should just re-assign values for properties originally given in xhtmlrenderer.conf. Please see the documentation in that file for a description of what each property affects. As of R3, we look for a specific override file in a specific location, e.g.

```
$user.home/.flyingsaucer/local.xhtmlrenderer.conf
```

The user.home variable is a system property. If you call the `System.getProperty("user.home")` from within any Java program on your machine, you will find out where this is. The location is usually `c:\Documents And Settings\{username}` and under the `/usr` directory on UNIX systems. Try that method call to see where it is on your machine.

Override with System Properties

You can also override properties one-by-one on the command line, using System properties. To override a property using the System properties, just re-define the property on the command line. e.g.

```
java -Dxr.property-name=new_value org.xhtmlrenderer.HTMLPa
```

You can override as many properties as you like. Note that overrides are driven by the property names in the default configuration file. Specifying a property name not

in that file will have no effect—the property will not be loaded or available for lookup. Logging output is also controlled in this Configuration file.

If you think an override is not taking, you can change the logging behavior of the Configuration class. Because of inter-dependencies between Configuration and the logging system, this is a special-case key, using the System property show-config. The allowed values are from the `java.util.logging.Level` class. Use ALL to show a lot of detail about Configuration startup, OFF for none, and INFO for regular output, like this

```
java -Dshow-config=ALL org.xhtmlrenderer.HTMLPanel
```

This will output messages to the console as Configuration is loading and looking for overrides to the default property values for the renderer.

We have just started using the Configuration system late in preparing release R4. Some runtime behavior that should be configurable (like XHTML parser) is not. If you would like to see some behavior made configurable, shoot us an email.

Looking up Configuration at Runtime

To access a parameter from Configuration at runtime, just use on of the many static methods on the Configuration class. All of these just take the full name of the property:

- String Configuration.valueFor(String)
- String Configuration.valueFor(String key, String default)
- byte Configuration.valueAsByte(String key, byte default)
- double Configuration.valueAsDouble(String key, double default)
- float Configuration.valueAsFloat(String key, float default)
- int Configuration.valueAsInt(String key, int default)
- long Configuration.valueAsLong(String key, long default)
- short Configuration.valueAsShort(String key, short default)
- boolean Configuration.isTrue(String key, boolean default)
- boolean Configuration.isFalse(String key, boolean default)

Logging

The renderer uses the java.util.logging package for logging information and exceptions at runtime. Logging behavior (output) is controlled via the main configuration file. The defaults may be overridden just like any other configuration properties.

Please review the java.util.logging package docs before proceeding.

We log to a set of hierarchies. The internal code—everything between a request to load a page and the page rendering—is logged to a subhierarchy of "plumbing", e.g. plumbing.load. Our convention is that WARNING and SEVERE levels are very important and should always be logged. INFO messages are useful and but can be

excluded if you want a quiet ride. Anything below INFO (FINE, FINER, FINEST) is generally only interesting for core renderer developers. We don't guarrantee that anything below INFO will be useful, correct, practical or informative. You can usually leave log levels at INFO for most purposes.

If you are modifying the renderer core code and want to add log messages, we recommend you always use the `org.xhtmlrenderer.XRLog` class. Using this class ensures that our log configuration is read properly before we write anything out to the log system. The class is pretty self-explanatory, and all logging methods in it are static. If for some reason you need to use the `java.util.logging.Logger` class directly, please use XRLog.getLogger() to retrieve the instance to use.