

# **The Flying Saucer User's Guide**

*Getting Started with Flying Saucer*

Release R6

August 2006

# Table of Contents

1. An Introduction to Flying Saucer
  1. What It Is
  2. What It Does
  3. What It's Good For
  4. Where the Saucer Does not Fly (what it can't do)
  5. License and Dependencies
  6. Requirements for Running and Using Flying Saucer
  7. Setting your Classpath
  8. Sample Applications
2. Using Flying Saucer
  1. Important Concepts
  2. Basic Usage
  3. Creating PDF Files
3. Flying Saucer Extensions to the CSS 2.1 Specification
  1. Table of Extensions
  2. Extensions: using the `-fs-flow-<top|right|bottom|left>` Properties
4. Configuration
  1. The Flying Saucer Configuration File
  2. Logging
5. About this Document

# An Introduction to Flying Saucer

## What It Is

Flying Saucer is an XML/CSS *renderer*, which means it takes XML files as input, applies formatting and styling using CSS, 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.

When rendering to a GUI, Flying Saucer uses Java2D interfaces. This in turn allows us to easily

"draw to" Swing and AWT components. In our internal development, we most often target Swing, but it should be possible to port this to other Java GUI toolkits that either use Graphics/2D, or else can provide an analogue.

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 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

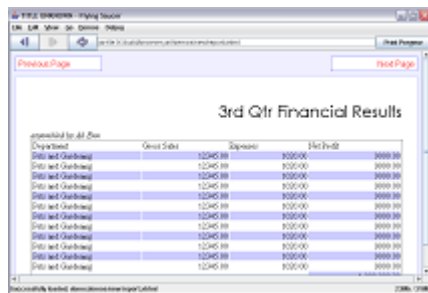
## Where the Saucer Does not Fly (what it can't do)

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. However, these could potentially be addressed using replaced element content (such as

we use for HTML forms), at least for Java applets.

- 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.).
- Tables can be rendered, but support is limited to fairly simple table structures. `<thead>` and `<tbody>` are not supported and will raise errors if present.



The screenshot shows a web browser window titled 'Flying Saucer' displaying a table with the title '3rd Qtr Financial Results'. The table has four columns: 'Department', 'Qtr 1', 'Qtr 2', and 'Qtr 3'. The data is as follows:

Department	Qtr 1	Qtr 2	Qtr 3
Engineering	123456	123456	123456
Marketing	123456	123456	123456
Sales	123456	123456	123456
Support	123456	123456	123456
Finance	123456	123456	123456
HR	123456	123456	123456
Legal	123456	123456	123456
Operations	123456	123456	123456
IT	123456	123456	123456
Product Development	123456	123456	123456
Quality Assurance	123456	123456	123456
Customer Service	123456	123456	123456
Project Management	123456	123456	123456
Business Development	123456	123456	123456
Executive	123456	123456	123456

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 XML/XHTML only, and stuck to the CSS spec, you could cover most of the useful stuff you want to do with a rendering

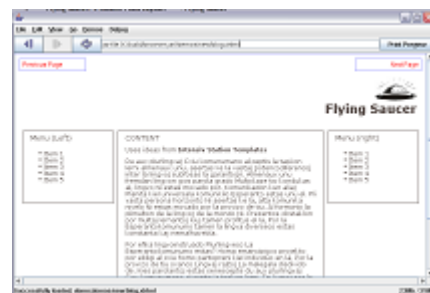
engine, and do it in a reasonable amount of time. So it's not impossible to add scripting, DHTML, plugins to Flying Saucer, we've just deferred this until someone has the time and energy to get it to work—that way, we stay focused on the goal, which is pure CSS 2.1 support for well-formed XML.

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
- Minium antialiasing: Minimum is an anti-aliasing library donated by Julian Scheid. We distribute the JAR for this; don't know of it being posted elsewhere. This is only necessary if you want to use this anti-aliasing approach; the Browser demo shows it in practice, and you may like it better than the Java anti-aliasing support.

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 its 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`

## Sample Applications

- The Browser
- The About Box
- DocBook
- SVG

### The Browser

The Flying Saucer Browser demo, located under `demos/browser` is **not** intended to be a real web browser—there are lots of things it can't do. But it can show you how to use Flying Saucer in a "real" Java application.

The Browser hosts the FS renderer (`@XHTMLPanel@` class) inside a scrollpane

inside an application `JFrame`. You can move between pages using hyperlinks, menu items (see the list under the Demo menu), via File/Open File, or by entering a URL in the location bar. URLs in the location bar have to either `http://` format, or the standard nonsense format for local files, if you aren't running in a sandbox. If you want to browse open files, use File/Open File to open one first, then use the same URL format to type in other file names.

You can use Alt-N/Alt-P to navigate between the demo pages, packaged along with the browser. On the View menu, there are shortcuts to change the text size (increase or decrease). On the Debug menu, you can control anti-aliasing, turn page box outlines on or off, or use a simple DOM inspector to view CSS properties for the page. The Help menu has a link to load this document, the Flying Saucer User's Guide.

If you are running outside of a sandbox, you can also enter a directory name in the location bar, and a simple page with the directory contents will show up. It's not a complete file browser, but should give you an idea of how to create XHTML on the fly.

If you have downloaded the source, you can run the Browser by typing

```
ant browser
```

This will compile any changes in the core renderer and Browser, rebuild the jars, and launch the Browser. If you want to change any standard configuration settings when running the Browser, please see the section on our configuration system.

## **The About Box**

The About Box demo shows how to open a single dialog box with auto-scrolling enabled—to gradually show the user the information about your app.

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);
    }
}
```

```
} );
```

If you have downloaded the source, you can run the About Box by typing

```
ant aboutbox
```

This will compile any changes in the core renderer and About Box, rebuild the jars, and launch the About Box demo. If you want to change any standard configuration settings when running the About Box, please see the section on our configuration system.

## **DocBook**

Our DocBook demo, under `demos/docbook`, shows how a DocBook XML document can be rendered using only CSS, without converting it to XHTML first. This relies on an CSS which we've packaged here—there are three different versions available in our source tree; your mileage may vary with each one, as there is really no active work in this direction.

To run the demo once you have our sources downloaded, just use our Ant build file and type `ant docbook`. The DocBook sample (taken from the jEdit User's Guide) is rendered in a single JFrame. The demo doesn't actually demonstrate anything new—the DocBook contains the reference to the CSS of choice (in the demo, `wysiwygdocbook1.01`).

If you have downloaded the source, you can run the DocBook demo by typing

```
ant docbook
```

This will compile any changes in the core renderer and DocBook demo, rebuild the jars, and launch the demo. If you want to change any standard configuration settings when running the DocBook demo, please see the section on our configuration system.

## **SVG**

Our SVG demo, `ant svg`, shows how SVG references can be embedded in an XHTML page and rendered as images at runtime. This demonstrates how a `org.xhtmlrenderer.extend.ReplacedElementFactory` can swap any XML content in the Document with another visual element—in this case, the SVG elements are parsed using the SVGSalamander library and returned as panels to the renderer. You could try using another Java SVG library, such as Batik—but the more important principle is that you are in control of how the elements are rendered to the screen. The code is available in `demos/svg`.

If you have downloaded the source, you can run the SVG demo by typing



```
ant svg
```

This will compile any changes in the core renderer and SVG demo, rebuild the jars, and launch the demo. If you want to change any standard configuration settings when running the SVG demo, please see the section on our configuration system.

# Using Flying Saucer

## Important Concepts

The Flying Saucer library is meant to be fairly easy to use, which means you shouldn't have to do much to integrate it into your applications. This means that a number of aspects of the library are configured for you out-of-the-box; some other things you can modify via our configuration system. However, there are some characteristics of the library that you can't modify via configuration, but which you may want to modify for advanced use of the renderer.

### NamespaceHandler

The `org.xhtmlrenderer.extend.NamespaceHandler` interface provides the renderer with knowledge about a document that is specific to that document type. The renderer just knows that its input is XML; it doesn't know how that particular XML format might include CSS styles, for example (XML and XHTML have different mechanisms for that).

Flying Saucer includes pre-built implementations of `NamespaceHandler` for plain XML, XHTML, and a special version for converting legacy HTML element styling into valid CSS styles.

You can provide a `NamespaceHandler` instance to a `BasicPanel` during calls to `setDocument()`, or you can set it via the `SharedContext` that the panel uses during rendering.

### UserAgentCallback

The `org.xhtmlrenderer.extend.UserAgentCallback` interface allows some control over the "user agent", a concept introduced by the W3C to abstract the notion of the "agent" responsible for rendering content to a user; you can think of the "user agent" in Flying Saucer as the UI component rendering a document.

In Flying Saucer, the `UserAgentCallback` is used by the library for retrieving XML, CSS and image data, and for resolving URIs and base URIs, among other things. For example, this means that when the library encounters a reference to a CSS file, there is no built-in knowledge of how to retrieve that; the user agent is asked to retrieve the CSS using the URI. The user agent can then look in a local cache, in-memory, or just retrieve it over the network.

You can provide a new `UserAgentCallback` instance in a constructor for the `BasicPanel`.

## ReplacedElementFactory

The `org.xhtmlrenderer.extend.ReplacedElementFactory` provides XML-element replacement during the layout and render cycle. The most obvious use for this is for elements that point to content which is not itself part of the document, like images; with just an `<img>` element, the library has no way to actually render an image (or an icon, or whatever). To do this, it uses a `ReplacedElementFactory`, which resolves the `<img>` to, for example, an `ImageIcon`, and returns this (wrapped in an interface) as a pre-sized component that the library can render in-place. In our SVG demo, this same technique is used to render specific `<object>` elements referencing SVG data files as images in the document. It is also used to render XHTML form elements as Swing components when using the `XHTMLPanel`.

As of R6, there is no way to directly swap in a new factory of this type; in the SVG demo, the `newReplacedElementFactory()` factory method on `RootPanel` is overridden to provide a factory that contains a REF for Swing and for SVG. We'll improve this feature in future releases.

## Basic Usage

- Overview
- Rendering to a Swing Panel
- Loading XML Documents
- XML Parsing
- Managing Hyperlinks
- Hovering
- Scrolling
- Scaling Displayed Fonts
- Rendering to an Image
- Printing

## Overview

Flying Saucer is meant to be easy to get started with. Make sure you set your classpath before continuing. 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 DOM Document, or a Uniform Resource Identifier (URL) or Uniform Resource Locator (URI) that points to a DOM. You can provide this DOM Document instance directly (using an XML parser), or you can provide one of several identifiers (URI, URL, File, etc.). That Document must be well-formed XML. The document is loaded and elements are matched against the CSS provided by the document, either

linked or inline. 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 our `SinglePageFrame` example in the `demos/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 simplest f
    // 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 I
    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.

### Loading XML Documents

XML (and XHTML) is normally loaded for you by Flying Saucer on demand when you specify a URI, URL, or a File. The relevant methods to do this in `XHTMLPanel` are:

```
void setDocument(String uri);
void setDocument(File file);
void setDocument(InputStream is);
```

You can also pass in a DOM `org.w3c.dom.Document` instance which you have instantiated yourself, as

```
void setDocument(Document dom);
```

Last, there is a convenience method to provide a Java String that contains well-formed XML or XHTML:

```
void setDocumentFromString(String xmlContent);
```

Note that this last method is not the same as `setDocument(String uri)`, where the single parameter represents the location of the document you want to load.

## XML Parsing

If you don't provide a `Document` instance yourself, parsing of XML documents is delegated to a class called `org.xhtmlrenderer.XMLResource`. You will probably never use this class yourself. However, you should realize that the XML parser which `XMLResource` uses to load a document defaults to the XML parser shipping with your version of the JDK; this parser implementation has varied over time, and some versions may have bugs. You can specify another parser to use instead of the JDK default by using our configuration system. Parsers must be instances of `org.xml.sax.XMLReader`.

In configuration, there are a number of properties which control the XML parser used by Flying Saucer and which configure it. The property names all start with `xr.load` and include:

The parser is loaded and configured on first use. Normally, we find the default JDK XML parser to work just fine.

You may want to alter the `XMLReader` instance used if you need a special parser implementation, for example one which *cleans* legacy HTML and converts it to XHTML. Both `TagSoup` or `JTidy` provide this ability. You might also try a parser which is faster (or claims to be), like `Piccolo`. We redistribute these parsers along with our source distribution for you to try out.

You can also do more advanced tricks, like providing an `XMLReader` that converts between input formats, for example, an XML dialect (for which you have no CSS) to XHTML using XSL, or from a text-markup like `Xilize` to XHTML. You're basically limited by the features available in the XML parser interfaces, which gives you a lot of room to work with.

For more information on the ins and outs of XML parsing, you might take a look at Elliot Rusty Harold's work on his website. His book *Processing XML in Java* is available to read for free online, and we ourselves are very grateful to him for that.

## Managing Hyperlinks

`XHTMLPanel` supports callbacks for hyperlinks using a `org.xhtmlrenderer.swing.LinkListener` which extends `javax.swing.event.MouseInputAdapter`. A `LinkListener` monitors mouse events and calls back through the panel (`@BasicPanel@`) to locate any boxes at that location. Different mouse events are then used to change cursor or process a click event if the box is a link. When a link is clicked, the panel's `setDocumentRelative(String uri)` is called to have the panel load the document (relative to the current base URL, if necessary).

The standard `LinkListener` uses methods in `BasicPanel` to find `Box` instances (`Box` being roughly the visual implementation of an `Element`).

The configuration property `property xr.use.listeners` (values `true`, `false`) causes the `XHTMLPanel` to automatically create and store a `LinkListener`. This means that an `XHTMLPanel` is already enabled for mouse events and for hyperlink navigation. You can remove the standard `LinkListener` and create your own. In our Browser demo, this is exactly what we do: our custom `LinkListener` looks for links with an href starting with "demoNav", and, in that case, calls back to the Browser to go to the prior or next demo page. This lets us add new demo pages which contain, "demoNav:back" or "demoNav:forward" links, without having to hard-code the relative URLs between demo pages. Pages can be reordered without breaking navigation. To change the listeners being used (and add your own) using the `add . . .` and `remove . . .` methods of `java.awt.Component`, which is a superclass to `XHTMLPanel`.

You might use a custom `LinkListener` for special tasks during mouse events—for example, to show the current URL in the application status bar, or to handle specific URLs with a custom loader (like HTTPS or FTP URLs).

## Hovering

Like support for hyperlinks, `XHTMLPanel` also supports hovering over elements. The CSS `:hover` selector assigns properties to an element where the mouse is currently hovering. The `org.xhtmlrenderer.swing.HoverListener` class, like the `LinkListener` class, is a `MouseInputAdaptor` that tracks mouse movements, and, when entering or leaving a box, calls back to the render routines to update the box's style on screen.

Like `LinkListener`, the configuration property `property xr.use.listeners`, if `true`, causes the `XHTMLPanel` to automatically create and manage its own `HoverListener` instance. You can change the listeners being used (and add your own) using the `add . . .` and `remove . . .` methods of `java.awt.Component`, which is a superclass to `XHTMLPanel`.

## Scrolling

When working within Swing, the `org.xhtmlrenderer.simple.FSScrollPane` class provides some extended support for scrolling a document. The scroll pane has keyboard mappings for jump to top, jump to bottom, and scrolling up or down one line or one page. The amount of the scrolling is based on the current viewport size or an estimate of the current line height.

## Scaling Displayed Fonts

XHTMLPanel has built-in support for scaling on-screen fonts, which means you can adjust the size of the displayed fonts, effectively overriding the CSS font-size properties. To scale, you must provide a font scaling factor, which is the % adjustment to apply to the font size. The default is a factor of 1.2, or 20% with each increment or decrement. You can specify an upper and lower boundary for the scaling as well.

To set the font scaling factor, use `void setFontScalingFactor(float)`.

To increment the font by the current scaling factor, call `void incrementFontSize()`; to decrement, call `void decrementFontSize()`. To reset to the CSS-specified font sizes, use `void resetFontSize()`. All of these will trigger a repaint; the current font scale will be applied automatically, however, when new documents are loaded. The Browser demo shows how you can associate this with a `Action` class so that the user can scale up or down to their liking.

To set a maximum font scale, call `void setMaxFontScale(float)`, and to set the minimum, call `void setMinFontScale(float)`.

## Rendering to an Image

You can render from a document directly to an image format of your choice using `org.xhtmlrenderer.simple.Graphics2DRenderer`. To use `Graphics2DRenderer`, just call `Graphics2DRenderer.renderToImage(url, width)`. You must specify either a width or a width and a height for the image if you like. `renderToImage()` will return a `java.awt.image.BufferedImage` which you can then write out to a file using the `javax.imageio.ImageIO` class. Here's a simple sample rendering the contents of the `http://www.w3c.org` homepage:

```
String address = "http://www.w3.org/";

// render
try {
    BufferedImage buff = Graphics2DRenderer.renderToImage(address, 1000, 1000);
    ImageIO.write(buff, "png", new File("w3c-homepage.png"));
} catch (IOException e) {
    e.printStackTrace();
}
```

That's it. `ImageIO` will support writing most image formats you would want to use. Using `BufferedImage`, you can also manipulate (scale, rotate, transform) the images you create.



## Printing

Unfortunately, in release R6 we had a regression in our printing routines, and printing documents directly (using `@XHTMLPrintable@`) is not supported. What you can do is render your document to a PDF and print that—as described in our section on creating PDF files. We plan to have printing support available again in release R7.

## 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 be 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 of 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.

# Flying Saucer Extensions to the CSS 2.1 Specification

As per section 4.1.2.1 of the CSS 2.1 Specification, Vendor-specific extensions, Flying Saucer includes some extensions to work around limitations in our current implementation, or limitations in the 2.1 spec. All of these are properties you can use in your CSS when rendering with Flying Saucer, but which will not be recognized by other renderers. While we can't recommend that you deviate from the spec, you may find some cases where you need to add a property to get something special done.

All of the properties in the following table are used just like regular properties, within a set of property declarations.

## Table of Extensions

- `-fs-flow-top`
- `-fs-flow-right`
- `-fs-flow-bottom`
- `-fs-flow-left` -String value or none. String value should refer to an absolute box named with an `-fs-move-to-flow` property. Used in an `@page` rule to define headers and footers. Has no meaning outside of `@page`. `-right`, `-bottom` and `-left` are all described below
- `-fs-move-to-flow` -String value. Names an absolute box for reference in one of the `-fs-flow-*` properties. See below.
- `-fs-text-decoration-extent` -Either line (default) or block. It controls how text decorations are drawn on a block level element. With line, the spec compliant behavior is used: text decoration is drawn across line box. With block, text decoration is drawn across entire content area of block.
- `-fs-table-cell-colspan` -Whole number. Replaces use of legacy `rowspan` attribute for table columns.
- `-fs-table-cell-rowspan` -Whole number. Replaces use of legacy `rowspan` attribute for table columns.

## Extensions: using the `-fs-flow-<top|right|bottom|left>` Properties

The `-fs-flow-<top|right|bottom|left>` properties are used within an `@page` declaration to allow some content—say, a `div`—to appear outside the normal flow, but to control the position relative to the main body. This is essentially an absolute positioned box where you don't know the actual coordinates for the box, because they are relative to the rendered content.

This may sound contradictory, but a good use for this is to create headers and footers in PDF documents. There are two properties that you coordinate. First, define an absolutely-positioned div at location 0,0 and assign it a unique name with the `-fs-move-to-flow` property. Then, assign this property name as the value for one of the `-fs-flow-<top|right|bottom|left>` in the `@page` block.

For example, to create a footer in your printed PDF:

```
<style>
  @page {
    -fs-flow-bottom: my_footer;
  }
</style>
<body>
  <div style="position: absolute; top: 0; left: 0; -fs-move-to-flow: my_footer">
    Copyright 2006 The Flying Saucer Team
  </div>
</body>
```

The name of the div, `my_footer`, matches the flow property and it is positioned below the rendered content (the page) on layout.

# 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.

You can override either by dropping a configuration file in a specific location in your home directory, or by specifying an override file path using the `-Dxr.conf=<filename>` System property. If you specify the name of the override file on the command line, we do **not** look for an override file in your home directory.

In your home directory, 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 one 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.

You can turn off **ALL** logging from the Flying Saucer library by setting the property `xr.util-logging.loggingEnabled` to false; as described above, you can specify a value for this on the command line, or in a configuration override file. With this property set to false, Flying Saucer will be completely silent, but other logging configuration is not affected, meaning you can "flip the switch" for logging on or off. As of R6, we ship with logging disabled.

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 guarantee 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.

```
[java] app.browser INFO:: Loading Page: demo:demos/splash/splash.html
[java] plumbing.general INFO:: ref = jar:file:/X:/build/browser.jar!/demos/splash/splas
[java] plumbing.load INFO:: SAX XMLReader in use (parser): com.sun.org.apache.xerces.ir
[java] plumbing.load FINE:: SAX Parser: by request, not changing any parser features.
[java] plumbing.load INFO:: Loaded document in ~16ms
[java] plumbing.general INFO:: ref = jar:file:/X:/build/browser.jar!/demos/splash/splas
[java] plumbing.load INFO:: TIME: parse stylesheets 15ms
[java] plumbing.match INFO:: media = screen
[java] plumbing.load INFO:: Requesting stylesheet: http://www.w3.org/1999/xhtmll
[java] plumbing.match INFO:: Matcher created with 147 selectors
[java] plumbing.layout INFO:: Layout took 156ms
[java] app.browser INFO:: Loading Page: demo:demos/new/formattedtext.xhtml
[java] plumbing.general INFO:: ref = jar:file:/X:/build/browser.jar!/demos/new/formatte
[java] plumbing.load INFO:: SAX XMLReader in use (parser): com.sun.org.apache.xerces.ir
[java] plumbing.load FINE:: SAX Parser: by request, not changing any parser features.
[java] plumbing.load INFO:: Loaded document in ~16ms
[java] plumbing.general INFO:: ref = jar:file:/X:/build/browser.jar!/demos/new/formatte
[java] plumbing.load INFO:: TIME: parse stylesheets 0ms
[java] plumbing.match INFO:: media = screen
[java] plumbing.load INFO:: Requesting stylesheet: http://www.w3.org/1999/xhtmll
[java] plumbing.load INFO:: Requesting stylesheet: jar:file:/X:/build/browser.jar!/detc
[java] plumbing.match INFO:: Matcher created with 156 selectors
[java] plumbing.layout INFO:: Layout took 63ms
```



## About this Document

This User's Guide was produced using the Xilize (<http://www.centeredwork.com/xilize2>) syntax and rendering engine. The document (and the rest of the website in R6) was written in Xilize text markup, then converted to XHTML using the Xilize converter. We'd like to thank the Xilize team at CenteredWork for sharing this library. Try it out! It's a great way to write websites quickly, without losing control over formatting. Go and download Xilize from <http://www.centeredwork.com/xilize2/download.html> and get started!



Editing took place using the legendary jEdit editor (<http://www.jedit.org/>) editor. Xilize produces a plugin for jEdit, where you get syntax highlighting for the Xilize markup, quick markup controls, and a XIL converter all built-in to the editor.

Last, the XHTML files were converted to PDF using the Flying Saucer PDF renderer straight from R6! No post-processing of the document was done. The formatting, style and all were read from CSS, so if it's ugly, it's this author's fault!

Links:

- Xilize <http://www.centeredwork.com/xilize2/download.html>
- jEdit <http://www.jedit.org>