jsoup cookbook

Introduction

1. Parsing and traversing a Document

Input

2. Parse a document from a String
3. Parsing a body fragment
4. Load a Document from a URL
5. Load a Document from a File

Extracting data

6. Use DOM methods to navigate a document
7. Use selector-syntax to find elements
8. Extract attributes, text, and HTML from elements
9. Working with URLs
10. Example program: list links

Modifying data

11. Set attribute values
12. Set the HTML of an element
13. Setting the text content of elements

Cleaning HTML

14. Sanitize untrusted HTML (to prevent XSS)

Introduction

Parsing and traversing a Document

To parse a HTML document:

```
String html = "<html><head><title>First parse</title></head>"
  + "<body><p>Parsed HTML into a doc.</p></body></html>";
Document doc = Jsoup.parse(html);
```

(See parsing a document from a string for more info.)

The parser will make every attempt to create a clean parse from the HTML you provide, regardless of whether the HTML is well-formed or not. It handles:

- unclosed tags (e.g. `<p>Lorem <p>Ipsum` parses to `<p>Lorem</p> <p>Ipsum</p>`)
- implicit tags (e.g. a naked `<td>Table data</td>` is wrapped into a `<table><tr><td>?`)
- reliably creating the document structure (`html` containing a `head` and `body`, and only appropriate elements within the head)

The object model of a document

- Documents consist of Elements and TextNodes (and a couple of other misc nodes: see the nodes package tree).
- The inheritance chain is: `Document` extends `Element` extends `Node`. `TextNode` extends `Node`.
- An Element contains a list of children Nodes, and has one parent Element. They also have provide a filtered list of child Elements only.

See also

- Extracting data: DOM navigation
- Extracting data: Selector syntax

Input

Parse a document from a String

Problem

You have HTML in a Java String, and you want to parse that HTML to get at its contents, or to make sure it's well formed, or to modify it. The String may have come from user input, a file, or from the web.

Solution

Use the static `Jsoup.parse(String html)` method, or `Jsoup.parse(String html, String baseUri)` if the page came from the web, and you want to get at absolute URLs (see [working-with-urls]).

```
String html = "<html><head><title>First parse</title></head>"
  + "<body><p>Parsed HTML into a doc.</p></body></html>";
Document doc = Jsoup.parse(html);
```

Description

The `parse(String html, String baseUri)` method parses the input HTML into a new `Document`. The `base URI` argument is used to resolve relative URLs into absolute URLs, and should be set to the URL where the document was fetched from. If that's not applicable, or if you know the HTML has a `base` element, you can use the `parse(String html)` method.

As long as you pass in a non-null string, you're guaranteed to have a successful, sensible parse, with a Document containing (at least) a `head` and a `body` element. (**BETA**: if you do get an exception raised, or a bad parse-tree, please file a bug.)

Once you have a Document, you can get get at the data using the appropriate methods in Document and its supers `Element` and `Node`.

Parsing a body fragment

Problem

You have a fragment of body HTML (e.g. a `div` containing a couple of `p` tags; as opposed to a full HTML document) that you want to parse. Perhaps it was provided by a user submitting a comment, or editing the body of a page in a CMS.

Solution

Use the `Jsoup.parseBodyFragment(String html)` method.

```
String html = "<div><p>Lorem ipsum.</p>";
Document doc = Jsoup.parseBodyFragment(html);
Element body = doc.body();
```

Description

The `parseBodyFragment` method creates an empty shell document, and inserts the parsed HTML into the `body` element. If you used the normal `Jsoup.parse(String html)` method, you would generally get the same result, but explicitly treating the input as a body fragment ensures that any bozo HTML provided by the user is parsed into the `body` element.

The `Document.body()` method retrieves the element children of the document's `body` element; it is equivalent to `doc.getElementsByTag("body")`.

Stay safe

If you are going to accept HTML input from a user, you need to be careful to avoid cross-site scripting attacks. See the documentation for the `Whitelist` based cleaner, and clean the input with `clean(String bodyHtml, Whitelist whitelist)`.

Load a Document from a URL

Problem

You need to fetch and parse a HTML document from the web, and find data within it (screen scraping).

Solution

Use the `Jsoup.connect(String url)` method:

```
Document doc = Jsoup.connect("http://example.com/").get();
String title = doc.title();
```

Description

The `connect(String url)` method creates a new `Connection`, and `get()` fetches and parses a HTML file. If an error occurs whilst fetching the URL, it will throw an `IOException`, which you should handle appropriately.

The `Connection` interface is designed for method chaining to build specific requests:

```
Document doc = Jsoup.connect("http://example.com")
  .data("query", "Java")
  .userAgent("Mozilla")
  .cookie("auth", "token")
  .timeout(3000)
  .post();
```

This method only suports web URLs (`http` and `https` protocols); if you need to load from a file, use the `parse(File in, String charsetName)` method instead.

Load a Document from a File

Problem

You have a file on disk that contains HTML, that you'd like to load and parse, and then maybe manipulate or extract data from.

Solution

Use the static `Jsoup.parse(File in, String charsetName, String baseUri)` method:

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");
```

Description

The `parse(File in, String charsetName, String baseUri)` method loads and parses a HTML file. If an error occurs whilst loading the file, it will throw an `IOException`, which you should handle appropriately.

The `baseUri` parameter is used by the parser to resolve relative URLs in the document before a `<base href>` element is found. If that's not a concern for you, you can pass an empty string instead.

There is a sister method `parse(File in, String charsetName)` which uses the file's location as the `baseUri`. This is useful if you are working on a filesystem-local site and the relative links it points to are also on the filesystem.

Extracting data

Use DOM methods to navigate a document

Problem

You have a HTML document that you want to extract data from. You know generally the structure of the HTML document.

Solution

Use the DOM-like methods available after parsing HTML into a `Document`.

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");

Element content = doc.getElementById("content");
Elements links = content.getElementsByTag("a");
for (Element link : links) {
  String linkHref = link.attr("href");
  String linkText = link.text();
}
```

Description

Elements provide a range of DOM-like methods to find elements, and extract and manipulate their data. The DOM getters are contextual: called on a parent Document they find matching elements under the document; called on a child element they find elements under that child. In this way you can winnow in on the data you want.

Finding elements

- `getElementById(String id)`
- `getElementsByTag(String tag)`
- `getElementsByClass(String className)`
- `getElementsByAttribute(String key)` (and related methods)
- Element siblings: `siblingElements()`, `firstElementSibling()`, `lastElementSibling()`; `nextElementSibling()`, `previousElementSibling()`
- Graph: `parent()`, `children()`, `child(int index)`

Element data

- `attr(String key)` to get and `attr(String key, String value)` to set attributes
- `attributes()` to get all attributes
- `id()`, `className()` and `classNames()`
- `text()` to get and `text(String value)` to set the text content
- `html()` to get and `html(String value)` to set the inner HTML content
- `outerHtml()` to get the outer HTML value
- `data()` to get data content (e.g. of `script` and `style` tags)
- `tag()` and `tagName()`

Manipulating HTML and text

- `append(String html)`, `prepend(String html)`
- `appendText(String text)`, `prependText(String text)`
- `appendElement(String tagName)`, `prependElement(String tagName)`
- `html(String value)`

Use selector-syntax to find elements

Problem

You want to find or manipulate elements using a CSS or jquery-like selector syntax.

Solution

Use the `Element.select(String selector)` and `Elements.select(String selector)` methods:

```
File input = new File("/tmp/input.html");
Document doc = Jsoup.parse(input, "UTF-8", "http://example.com/");

Elements links = doc.select("a[href]"); // a with href
Elements pngs = doc.select("img[src$=.png]");
  // img with src ending .png

Element masthead = doc.select("div.masthead").first();
  // div with class=masthead

Elements resultLinks = doc.select("h3.r > a"); // direct a after h3
```

Description

jsoup elements support a CSS (or jquery) like selector syntax to find matching elements, that allows very powerful and robust queries.

The `select` method is available in a `Document`, `Element`, or in `Elements`. It is contextual, so you can filter by selecting from a specific element, or by chaining select calls.

Select returns a list of Elements (as `Elements`), which provides a range of methods to extract and manipulate the results.

Selector overview

- `tagname`: find elements by tag, e.g. `a`
- `ns|tag`: find elements by tag in a namespace, e.g. `fb|name` finds `<fb:name>` elements
- `#id`: find elements by ID, e.g. `#logo`
- `.class`: find elements by class name, e.g. `.masthead`
- `[attribute]`: elements with attribute, e.g. `[href]`
- `[^attr]`: elements with an attribute name prefix, e.g. `[^data-]` finds elements with HTML5 dataset attributes
- `[attr=value]`: elements with attribute value, e.g. `[width=500]`
- `[attr^=value]`, `[attr$=value]`, `[attr*=value]`: elements with attributes that start with, end with, or contain the value, e.g. `[href*=/path/]`
- `[attr~=regex]`: elements with attribute values that match the regular expression; e.g. `img[src~=(?i)\.(png|jpe?g)]`
- `*`: all elements, e.g. `*`

Selector combinations

- `el#id`: elements with ID, e.g. `div#logo`
- `el.class`: elements with class, e.g. `div.masthead`

- `el[attr]`: elements with attribute, e.g. `a[href]`
- Any combination, e.g. `a[href].highlight`
- `ancestor child`: child elements that descend from ancestor, e.g. `.body p` finds `p` elements anywhere under a block with class "body"
- `parent > child`: child elements that descend directly from parent, e.g. `div.content > p` finds `p` elements; and `body > *` finds the direct children of the body tag
- `siblingA + siblingB`: finds sibling B element immediately preceded by sibling A, e.g. `div.head + div`
- `siblingA ~ siblingX`: finds sibling X element preceded by sibling A, e.g. `h1 ~ p`
- `el, el, el`: group multiple selectors, find unique elements that match any of the selectors; e.g. `div.masthead, div.logo`

Pseudo selectors

- `:lt(n)`: find elements whose sibling index (i.e. its position in the DOM tree relative to its parent) is less than `n`; e.g. `td:lt(3)`
- `:gt(n)`: find elements whose sibling index is greater than `n`; e.g. `div p:gt(2)`
- `:eq(n)`: find elements whose sibling index is equal to `n`; e.g. `form input:eq(1)`
- `:has(seletor)`: find elements that contain elements matching the selector; e.g. `div:has(p)`
- `:not(selector)`: find elements that do not match the selector; e.g. `div:not(.logo)`
- `:contains(text)`: find elements that contain the given text. The search is case-insensitive; e.g. `p:contains(jsoup)`
- `:containsOwn(text)`: find elements that directly contain the given text
- `:matches(regex)`: find elements whose text matches the specified regular expression; e.g. `div:matches((?i)login)`
- `:matchesOwn(regex)`: find elements whose own text matches the specified regular expression
- Note that the above indexed pseudo-selectors are 0-based, that is, the first element is at index 0, the second at 1, etc

See the `Selector` API reference for the full supported list and details.

Working with URLs

Problem

You have a HTML document that contains relative URLs, which you need to resolve to absolute URLs.

Solution

1. Make sure you specify a `base URI` when parsing the document (which is implicit when loading from a URL), and
2. Use the `abs:` attribute prefix to resolve an absolute URL from an attribute:

```
Document doc = Jsoup.connect("http://jsoup.org").get();

Element link = doc.select("a").first();
String relHref = link.attr("href"); // == "/"
String absHref = link.attr("abs:href"); // "http://jsoup.org/"
```

Description

In HTML elements, URLs are often written relative to the document's location: `<a href="/download">...</a>`. When you use the `Node.attr(String key)` method to get a href attribute, it will be returned as it is specified in the source HTML.

If you want to get an absolute URL, there is a attribute key prefix `abs:` that will cause the attribute value to be resolved against the document's base URI (original location): `attr("abs:href")`

For this use case, it is important to specify the base URI when parsing the document.

If you don't want to use the `abs:` prefix, there is also a method `Node.absUrl(String key)` which does the same thing, but accesses via the natural attribute key.

Example program: list links

This example program demonstrates how to fetch a page from a URL; extract links, images, and other pointers; and examine their URLs and text.

Specify the URL to fetch as the program's sole argument.

```
package org.jsoup.examples;

import org.jsoup.Jsoup;
import org.jsoup.helper.Validate;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import java.io.IOException;

/**
 * Example program to list links from a URL.
 */
public class ListLinks {
    public static void main(String[] args) throws IOException {
        Validate.isTrue(args.length == 1, "usage: supply url to fetch");
        String url = args[0];
        print("Fetching %s...", url);

        Document doc = Jsoup.connect(url).get();
        Elements links = doc.select("a[href]");
        Elements media = doc.select("[src]");
```

```
        Elements imports = doc.select("link[href]");

        print("\nMedia: (%d)", media.size());
        for (Element src : media) {
            if (src.tagName().equals("img"))
                print(" * %s: <%s> %sx%s (%s)",
                        src.tagName(), src.attr("abs:src"), src.attr("width"),
                                                                src.attr("height"),
                        trim(src.attr("alt"), 20));
            else
                print(" * %s: <%s>", src.tagName(), src.attr("abs:src"));
        }

        print("\nImports: (%d)", imports.size());
        for (Element link : imports) {
            print(" * %s <%s> (%s)", link.tagName(),link.attr("abs:href"),
                                                                 link.attr("rel"));
        }

        print("\nLinks: (%d)", links.size());
        for (Element link : links) {
            print(" * a: <%s>  (%s)", link.attr("abs:href"), trim(link.text(), 35));
        }
    }

    private static void print(String msg, Object... args) {
        System.out.println(String.format(msg, args));
    }

    private static String trim(String s, int width) {
        if (s.length() > width)
            return s.substring(0, width-1) + ".";
        else
            return s;
    }
}
```

[org/jsoup/examples/ListLinks.java]

Example output (trimmed)

```
Fetching http://news.ycombinator.com/...  Media: (38)  * img:
<http://ycombinator.com/images/y18.gif> 18x18 ()  * img:
<http://ycombinator.com/images/s.gif> 10x1 ()  * img:
<http://ycombinator.com/images/grayarrow.gif> x ()  * img:
<http://ycombinator.com/images/s.gif> 0x10 ()  * script:
<http://www.co2stats.com/propres.php?s=1138>  * img:
<http://ycombinator.com/images/s.gif> 15x1 ()  * img:
<http://ycombinator.com/images/hnsearch.png> x ()  * img:
<http://ycombinator.com/images/s.gif> 25x1 ()  * img:
<http://mixpanel.com/site_media/images/mixpanel_partner_logo_borderless.gif> x (Analytics
by Mixpan.)   Imports: (2)  * link <http://ycombinator.com/news.css> (stylesheet)  * link
<http://ycombinator.com/favicon.ico> (shortcut icon)   Links: (141)  * a:
<http://ycombinator.com>  ()  * a: <http://news.ycombinator.com/news>  (Hacker News)  *
a: <http://news.ycombinator.com/newest>  (new)  * a:
<http://news.ycombinator.com/newcomments>  (comments)  * a:
<http://news.ycombinator.com/leaders>  (leaders)  * a: <http://news.ycombinator.com/jobs>
(jobs)  * a: <http://news.ycombinator.com/submit>  (submit)  * a:
<http://news.ycombinator.com/x?fnid=JKhQjfU7gW>  (login)  * a:
<http://news.ycombinator.com/vote?for=1094578&dir=up&whence=%6e%65%77%73>  ()  * a:
<http://www.readwriteweb.com/archives/facebook_gets_faster_debuts_homegrown_php_compiler.
php?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+readwriteweb+%28ReadWriteW
eb%29&utm_content=Twitter>  (Facebook speeds up PHP)  * a:
<http://news.ycombinator.com/user?id=mcxx>  (mcxx)  * a:
<http://news.ycombinator.com/item?id=1094578>  (9 comments)  * a:
<http://news.ycombinator.com/vote?for=1094649&dir=up&whence=%6e%65%77%73>  ()  * a:
<http://groups.google.com/group/django-developers/msg/a65fbbc8effcd914>  ("Tough. Django
produces XHTML.")  * a: <http://news.ycombinator.com/user?id=andybak>  (andybak)  * a:
<http://news.ycombinator.com/item?id=1094649>  (3 comments)  * a:
<http://news.ycombinator.com/vote?for=1093927&dir=up&whence=%6e%65%77%73>  ()  * a:
<http://news.ycombinator.com/x?fnid=p2sdPLE7Ce>  (More)  * a:
<http://news.ycombinator.com/lists>  (Lists)  * a: <http://news.ycombinator.com/rss>
```

```
(RSS)  * a: <http://ycombinator.com/bookmarklet.html>  (Bookmarklet)  * a:
<http://ycombinator.com/newsguidelines.html>  (Guidelines)  * a:
<http://ycombinator.com/newsfaq.html>  (FAQ)  * a: <http://ycombinator.com/newsnews.html>
(News News)  * a: <http://news.ycombinator.com/item?id=363>  (Feature Requests)  * a:
<http://ycombinator.com>  (Y Combinator)  * a: <http://ycombinator.com/w2010.html>
(Apply)  * a: <http://ycombinator.com/lib.html>  (Library)  * a:
<http://www.webmynd.com/html/hackernews.html>  ()  * a: <http://mixpanel.com/?from=yc>
()
```

Modifying data

Set attribute values

Problem

You have a parsed document that you would like to update attribute values on, before saving it out to disk, or sending it on as a HTTP response.

Solution

Use the attribute setter methods `Element.attr(String key, String value)`, and `Elements.attr(String key, String value)`.

If you need to modify the `class` attribute of an element, use the `Element.addClass(String className)` and `Element.removeClass(String className)` methods.

The `Elements` collection has bulk attribue and class methods. For example, to add a `rel="nofollow"` attribute to every `a` element inside a div:

```
doc.select("div.comments a").attr("rel", "nofollow");
```

Description

Like the other methods in `Element`, the `attr` methods return the current `Element` (or `Elements` when working on a collection from a select). This allows convenient method chaining:

```
doc.select("div.masthead").attr("title", "jsoup").addClass("round-
box");
```

Set the HTML of an element

Problem

You need to modify the HTML of an element.

Solution

Use the HTML setter methods in `Element`:

```
Element div = doc.select("div").first(); // <div></div>
div.html("<p>lorem ipsum</p>"); // <div><p>lorem ipsum</p></div>
div.prepend("<p>First</p>");
div.append("<p>Last</p>");
// now: <div><p>First</p><p>lorem ipsum</p><p>Last</p></div>

Element span = doc.select("span").first(); // <span>One</span>
span.wrap("<li><a href='http://example.com/'></a></li>");
// now: <li><a href="http://example.com"><span>One</span></a></li>
```

Discussion

- `Element.html(String html)` clears any existing inner HTML in an element, and replaces it with parsed HTML.
- `Element.prepend(String first)` and `Element.append(String last)` add HTML to the start or end of an element's inner HTML, respectively
- `Element.wrap(String around)` wraps HTML around the **outer** HTML of an element.

See also

You can also use the `Element.prependElement(String tag)` and `Element.appendElement(String tag)` methods to create new elements and insert them into the document flow as a child element.

Setting the text content of elements

Problem

You need to modify the text content of a HTML document.

Solution

Use the text setter methods of `Element`:

```
Element div = doc.select("div").first(); // <div></div>
div.text("five > four"); // <div>five &gt; four</div>
div.prepend("First ");
div.append(" Last");
// now: <div>First five &gt; four Last</div>
```

Discussion

The text setter methods mirror the HTML setter methods:

- `Element.text(String text)` clears any existing inner HTML in an element, and replaces it with the supplied text.
- `Element.prepend(String first)` and `Element.append(String last)` add text nodes to the start or end of an element's inner HTML, respectively

The text should be supplied unencoded: characters like <, > etc will be treated as literals, not HTML.

Cleaning HTML

Sanitize untrusted HTML (to prevent XSS)

Problem

You want to allow untrusted users to supply HTML for output on your website (e.g. as comment submission). You need to clean this HTML to avoid cross-site scripting (XSS) attacks.

Solution

Use the jsoup HTML `Cleaner` with a configuration specified by a `Whitelist`.

```
String unsafe =
  "<p><a href='http://example.com/'
onclick='stealCookies()'>Link</a></p>";
String safe = Jsoup.clean(unsafe, Whitelist.basic());
// now: <p><a href="http://example.com/" rel="nofollow">Link</a></p>
```

Discussion

A cross-site scripting attack against your site can really ruin your day, not to mention your users'. Many sites avoid XSS attacks by not allowing HTML in user submitted content: they enforce plain text only, or use an alternative markup syntax like wiki-text or Markdown. These are seldom optimal solutions for the user, as they lower expressiveness, and force the user to learn a new syntax.

A better solution may be to use a rich text WYSIWYG editor (like CKEditor or TinyMCE). These output HTML, and allow the user to work visually. However, their validation is done on the client side: you need to apply a server-side validation to clean up the input and ensure the HTML is safe to place on your site. Otherwise, an attacker can avoid the client-side Javascript validation and inject unsafe HMTL directly into your site

The jsoup whitelist sanitizer works by parsing the input HTML (in a safe, sand-boxed environment), and then iterating through the parse tree and only allowing known-safe tags and attributes (and values) through into the cleaned output.

It does not use regular expressions, which are inappropriate for this task.

jsoup provides a range of `Whitelist` configurations to suit most requirements; they can be modified if necessary, but take care.

The cleaner is useful not only for avoiding XSS, but also in limiting the range of elements the user can provide: you may be OK with textual `a`, `strong` elements, but not structural `div` or `table` elements.

See also

- See the XSS cheat sheet and filter evasion guide, as an example of how regular-expression filters don't work, and why a safe whitelist parser-based sanitizer is the correct approach.
- See the `Cleaner` reference if you want to get a `Document` instead of a String return
- See the `Whitelist` reference for the different canned options, and to create a custom whitelist
- The nofollow link attribute