HTML Templates in GAVO DaCHS

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2016-07-29

Contents

While you do not have to write any HTML to publish data using DaCHS, and indeed we discourage spending much effort on fancy web pages – the VO is about generically usable APIs, not fancy, but non-interoperable web pages –, for some special effects or even the occasional custom UI many aspects of DaCHS web browser appearance can be overridden (and new behaviour introduced) through its nevow-based templating system. This article discusses how to do that.

DaCHS's System Templates

Overriding System Templates

DaCHS web pages are generated from templates delivered in the distribution's `gavo/resources/templates` subdirectory. Since that might actually be in an "egg" (compressed archive) on your installation, the easiest way to get at them is by using `gavo admin dumpDF templates/<template-name>`. The main reason to get them is to override them locally. To do that, put the template into `<dachs-root>/web/templates`.

For instance, if you wanted to change the sidebar, on a default installation you would type:

```
$ cd /var/gavo/web/templates
$ gavo admin dumpDF templates/sidebar.html > sidebar.html
```

While changes to templates will (almost) always directly show up in the pages served by DaCHS, if you freshly override a system template, you will need to re-start the server to make it realise it should load the template from a different source.

The content of the template files must be well-formed XML, preferably valid XHTML plus some extra elements and attributes from the `http://nevow.com/ns/nevow/0.1` namespace, and DaCHS will throw errors on non-well-formed XML. You may want to use run XML validators – as in:

```
xmlstarlet val -e sidebar.html
```

(requires the xmlstarlet package) – on the file before deploying them, although DaCHS error messages should typically give you a useful hint on where you got it wrong in case of errors.

While the sidebar's (and some other templates') source may not immediately figure as HTML, you can simply start adding HTML. Try, perhaps, adding:

```
<p class="sidebaritem">Hello world.</p>
```

above the opening div in sidebar.html and reload a page showing the sidebar. Your new material should show in the page.

Note that, if you override a template, it is a good idea to evaluate changes in the upstream templates after updating DaCHS. You can do this by running:

```
svn log https://svn.ari.uni-heidelberg.de/svn/gavo/python/trunk/gavo/resources/templates
```

(perhaps narrowing it down to the file you overrode). We try to avoid breaking user templates, but as there's no regression testing against that in place, and sometimes templates might reference functionality that has been dropped. So, it may and therefore will still happen (which is an important reason for trying to avoid overriding system templates and asking on dachs-support if someone knows another way to achive some desired effect. Plus, of course, by following upstream changes you can follow improvements we make – the popular "Send via SAMP" functionality, for instance, required a template change.

Templates Commonly Used

Ignoring some templates used for administration and internal purposes, here is a list of templates used by DaCHS:

defaultresponse.html: used by the form renderer for both the form itself and the result page. This is what you want to start from when doing custom UIs.

examples.html: used to format DALI examples (see Writing Examples)

loginout.html: used when logging out; note that DaCHS uses HTTP basic authentication; so, this is not a page you can use to explain to your users what to do to log in, they'll just see the browser's authentication window.

maintenance.html: used when the service is taken offline by creating a `state/MAINT` file. The `mainText` data item contains the contents of that file

rdinfo.html: the `...browse/<rdId>` pages are generated from these

root-tree.html: see The Root Template

root.html: see The Root Template

serviceinfo.html: used by the info renderer

sidebar.html: used by several browser-oriented renderers to create the sidebar

tableinfo.html: used by the tableinfo renderer (which is what makes the `tableinfo/<table name>` pages)

The Root Template

The `root.html` template is what is rendered when users request the root resource of your DaCHS installation, so it is a fairly good target for changing (and it's also fairly safe to override). While you could put any well-formed XHTML there, we recommend you base your template on either the distributed `root.html` template distribution, which mainly gives a list of all services available on the system, or `root-tree.html`, which organizes services in trees and only downloads metadata as necessary (a simple non-javascript fallback is part of the distributed template, too). The latter is intended for use on sites that have more then a few tens of services, when the plain `root.html` would expand to several 100s of kilobytes.

Note that the root page (and a few more similarly static documents) are fairly aggressively cached by DaCHS. This means that changes to the docment will not usually become immediately visible in your browser. However, documents

are not cached when you are logged in. As you write on your root page, it is therefore advisable to log in as administrator (see the [web]adminpasswd config item; e.g., click on the little [s] on the root page and select "Log in" from the sidebar); if you're happy with your design, run `gavo serve expire //services` (the root page cache is controlled by the //services RD), and it will be shown to non-logged users, too.

The Templating Language

DaCHS employs nevow's stan as its templating language. Our upstream, the Twisted community, has since integrated it into its web2 framework; since for DaCHS there is no compelling reason to migrate (but a fairly compelling reason not to, since we want to avoid unnecessarily breaking templates), we will keep nevow for the time being.

The great thing about nevow is that the templates are valid, verifyable XHTML. This is because nevow just adds a few elements and attributes in a separate namespace. This must be declared alongside the XHTML namespace; a standalone template will thus have to start like this:

```
<html
  xmlns:n="http://nevow.com/ns/nevow/0.1"
  xmlns="http://www.w3.org/1999/xhtml">
```

(prepending this with a doctype declaration like:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

is optional[1]); in the following, we assume the `n` prefix for the nevow namespace throughout, and you'd be devious to use something else.

The nevow namespace contains the following names (examples below; you're not supposed to understand things here, this is just a reference):

- `n:invisible` – an element that doesn't appear in the output. this is useful when you need some render or data attribute but have no element to "hang it on".
- `n:attr` – an element to add computed attributes to the embedding element. If you know `xsl.attribute`: same thing.
- `n:slot` – an element replacing itself with the value a mapping in the context data has for its `name` attribute.

- `n:render` – a universal (i.e., can go on any HTML tag) attribute changing the way the DOM tree is built below that element. To avoid confusion with DaCHS renderers, we will call the values of these attributes "nevow renderers" in the following.
- `n:pattern` – a universal attribute a bit like a tree-typed argument to a nevow renderer
- `n:data` – a universal attribute pulling a piece of data from python into the rendering context for use by nevow renderers.

-- that's really all it takes for a powerful XML templating language.

In-Python Stan

While it's normally preferable to load templates from XML file, sometimes you want to inline HTML fragments. Stan makes that relatively easy and save by providing a simple and elegant DOM-like language. Essentially, one says:

```
from nevow import tags as T
```

and then has the usual HTML elements as names within T. Calling them as functions will add attributes, indexing them will add children. This may sound odd but allows relatively nice expressions such as:

```
T.body[
  T.p["See also ", T.a(href="http://example.com")["here"], ". or"],
  T.ul[[
    T.li["or ", T.a(href=link)[anchor]
      for link, anchor in linklist]]]]
```

This T is already in the namespace of the renderer function (see below).

More information and this (as well as the whole template language) is available in Meet Stan.

Template Language Example

To see how this works, consider the `serviceinfo.html` template (see Overriding System Templates to see how to get it). The first two "magic" thing are children of `head`:

```
<title n:render="title"/>
...
<n:invisible n:render="commonhead"/>
```

The first is an empty title saying it wants to be rendered by `title`. That is a DaCHS-defined nevow renderer that says "put the service title as a text child of the embedding element (and delete any previous child)". If you are curious, the code implementing that is in `metarender.ServiceInfoRenderer` and looks like this:

```
def render_title(self, ctx, data):
  return ctx.tag["Information on Service '%s'"%
    base.getMetaText(self.service, "title")]
```

Below is a list the more Common Nevow Renderers.

The other magic thing is the `n:invisible`. As said above, that tag doesn't turn up in the result DOM, it's just there so we can call the `commonhead` nevow renderer. It pulls in all the common CSS and javascript declarations all (or most) DaCHS HTML pages want.

So: Nevow renderers essentially are just python functions putting things into tags.

The next magic thing is:

```
<body n:render="withsidebar">
```

Again, this is asks to pull in a whole lot of elements, this time from a template (you probably guessed that it is `sidebar.html`), except that this renderer does not discard children present in the template but adds them in a `div`. Perhaps the implementation helps here (don't sweat the details at this point; the code is in web.grend.GavoRenderMixin):

```
def render_withsidebar(self, ctx, data):
  oldChildren = ctx.tag.children
  ctx.tag.children = []
  return ctx.tag(class_="container")[
    self._sidebar,
    T.div(id="body")[
      T.a(name="body"),
      oldChildren
    ]]
```

(where `self._sidebar` contains a parse of the sidebar template).

Going on, you'll see:

```
<div class="useservice" n:data="browserURL" n:render="ifdata">
```

This is the first example for `n:data`. What happens here is that I put a thing into the render context, the `browserURL`. Again, that's mapped to a python function, which again comes from `web.metarender.ServiceInfoRenderer`. It looks like this:

```
def data_browserURL(self, ctx, data):
  return self.service.getBrowserURL()
```

– where the method returns a URL suitable for a web browser, or `None` if the service doesn't have a renderer for which a browser will display something useful (e.g., a service only having an `ssap.xml` renderer). This last part is used in the `n:render="ifdata"`. This nevow renderer (if you're curious, it's in web.grend and thus available on all DaCHS renderers) will return an empty string if the context object evaluates to false, while otherwise returning the tree below it.

The net effect of this construct is that the "Use this service in your browser" button that is generated in what's below will only be shown if the service actualy is usable in a web browser.

The image is put in literally, but the link it points to has to be taken from the context object (the `browserURL`). To get it into the attribute, nevow uses a trick you may know from XSLT:

```
<a>
  <n:attr name="href" n:render="string"/>
  <img src="/static/img/usesvcbutton.png" class="silentlink"
    alt="[Use this service from your browser]"/>
</a>
```

Essentially, there's an element, `n:attr`, that, when rendered, will end up as an attribute on the embedding element (the `a`). The name of that attribute is taken from the `name` attribute of `n:attr`, i.e., `href` in this case. The value of that attribute can be rendered in an arbitrary way, but in this case, I'm just using the `string` renderer (provided by nevow itself), which takes the context object, stringifies it and dumps it into the DOM tree. Had I written:

```
<a>
  <n:attr name="href">http://foo.bar/baz</n:attr>
  a link
</a>
```

the result would have been:

```
<a href="http://foo.bar/baz">a link</a>
```

– while introducing constants in this way doesn't make much sense, it might be useful if, for instance, the context object were a relative link, in which case you could have written:

```
<a>
  <n:attr name="href">http://foo.bar/baz<n:invisible
    n:render="string"/></n:attr>
  a link
</a>
```

If this seems spooky to you, play around with it for a moment – it's really not as hard as it might seem.

What follows in `serviceinfo.html` is just more of the same, with some additional nevow renderers explained below. The next new thing – and the last major concept to understand for nevew renderering – is this:

```
<ul n:data="rendAvail" n:render="sequence">
  <li n:render="mapping" n:pattern="item">
    <em><n:slot name="rendName"/></em> --
      <n:slot name="rendExpl"/></li>
</ul>
```

Let's take this step by step. First, we put something into the context object: `rendAvail`. This is again taken from the DaCHS renderer (still the `ServiceInfoRenderer`), which reads:

```
def data_rendAvail(self, ctx, data):
  return [{"rendName": rend,
      "rendExpl": RendExplainer.explain(rend, self.service)}
    for rend in self.service.allowed]
```

`RendExplainer.explain` is a function returning DOM fragments explaining what a renderer does, with embedded links and similar as necessary. So, what comes back is a structure looking like this:

```
[
  {"rendName": "form", "rendExpl": "This is browseable"},
  {"rendName": "scs.xml", "rendExpl": "This is not browseable"}]
```

-- a sequence of mappings.

Such sequences you can renderer using the nevow-provided `sequence` nevow renderer. This renderer looks for an element that has `n:pattern="sequence"` in the DOM tree below it, and renders one copy of of that each for each element of

the context object (which must be a sequence) with, and that's the brilliant part, the context object set to that object. You may want to read that sentence again...

So, with example sequene above the:

```
<li n:render="mapping" n:pattern="item">
  <em><n:slot name="rendName"/></em> --
    <n:slot name="rendExpl"/></li>
```

in the body of the `ul` element will be rendered twice, first with:

```
{"rendName": "form", "rendExpl": "This is browseable"}
```

as the context object and then with:

```
{"rendName": "scs.xml", "rendExpl": "This is not browseable"}]
```

These are mappings in python lingo, and so it may not surprise you that to make something sensible of that, nevow provides a `mapping` nevow renderer. It's a bit like `sequence` in that it inspects the tree below it. Unlike it, it doesn't look for `item` patterns but for `n:slot`, which has a `name` attribute – the element is then replaced by the value of the named key in the mapping.

One last new thing is a bit further down:

```
<n:invisible n:render="ifmeta copyright">
  <h2>Copyright, License, Acknowledgements</h2>
  <p n:render="metahtml">copyright</p>
</n:invisible>
```

Here, the render attribute contains a blank. Nevow interprets that as a parameterised renderer; in effect, the parameter(s) are passed to the renderer. It probably helps to see how this is implemented:

```
def render_ifmeta(self, metaName):
  if self.getMeta(metaName) is not None:
    return lambda ctx, data: ctx.tag
  else:
    return lambda ctx, data: ""
```

Anyway, parameterised nevow renderers need this extra argument, and you'll get weird errors if you don't provide it.

And that's almost all you need to know about nevow's templating language, except for what additional render and data functions there are. The more useful of which are covered in the next chapter.

Common Nevow Renderers

In DaCHS templates, nevow renderers can come from nevow itself, the active renderer (that's usually `form` for templates or a variety thereof), or the active service element in the RD (`customRF` and `customDF` for nevow renderers and data functions). Unfortunately, we've added nevow renderers and data functions in a somewhat ad-hoc fashion, and so they're not really documented terribly well.

Until we get to imporve that, here's an overview of the nevow renderers (and data functions) we believe are most useful in practice.

string: (from nevow) takes the unicode of a context object and places that in the DOM

sequence: (from nevow) iterates over the context object, rendering `item` patterns in its subtree once per item.

mapping: (from nevow) fills `n:slot` elements in its subtree with values from its (dict) context object, taking the slot's name attributes as keys

meta (data function): (from GavoRenderMixin) puts the named item from the current meta parent (usually, the service) into the context object. This could look like this:

```
<p n:data="meta creationDate" n:render="ifdata">
  <n:span class="date" n:render="string"/></p>
```

(you don't usally need this, prefer the meta render function)

meta: (from GavoRenderMixin) renders the meta item named in the text content as text.

metahtml: (from GavoRenderMixin) renders the meta item named in the text content as HTML; this automatically handles meta sequences and several special cases, so it's the preferred way of including metadata in templates.

rootlink: (from GavoRenderMixin) this was intended for running DaCHS "off-root" (i.e., as if in a subdirectory of the server). Since that's not how DaCHS was deployed in practice, it's not used consistently, so there's no point in using it right now.

getconfig: (from GavoRenderMixin) takes a config key from its content and renders it as the value of this configuration item, e.g.:

```
Finish up in
<span n:render="getconfig">[async]defaultExecTime</span> seconds
```

If you leave out the section, [general] is assumed as usual.

ifmeta: (from GavoRenderMixin) a parameterised renderer, i.e. used like this:

```
<p n:render="ifmeta coverage"><stuff/></p>
```

With this, stuff is rendered if a coverage meta item is present in the service.

ifownmeta: (from GavoRenderMixin) like ifmeta, except meta items are not inherited (e.g., title, which service typically gets from the embedding resource)

ifdata: (from GavoRenderMixin) only renders the embedded subtree if the context object evaluates to True

ifnodata: (from GavoRenderMixin) only renders the embedded subtree if the context object evaluates to False

ifslot: (from GavoRenderMixin) a parameterised renderer. It only renders the embedded subtree (usually a slot) if the named key exists in the context object (which must be a dict-like thing)

ifnoslot: (from GavoRenderMixin) the reverse of ifslot.

ifadmin: (from GavoRenderMixin) renders the embedded subtree if there is a logged user and that user is gavoadmin.

explodableMeta: (from GavoRenderMixin) used in the sidebar – see there for what it does.

authinfo: (from GavoRenderMixin) returns material letting an anonymous user log in or a logged in user log out.

prependsite: (from GavoRenderMixin) for templates intended to be shared between sites, this prepends the current site's short name (config item [web]sitename) to the element content.

withsidebar: (from GavoRenderMixin) used on the `body` element, adds the standard DaCHS sidebar to a page.

resulttable: (from HTMLResultRenderMixin) renders an HTML table out of a `result` context object (this is what draws the table on default HTML results)

resultline: (from HTMLResultRenderMixin) renders the first line of the result table in a key-value manner (this is sometimes used with the qp renderer)

parpair: (from HTMLResultRenderMixin) expects a (key, value) pair in the context and renders it as key:value in the current tag (this is used to render the current parameters; to avoid showing lots of non-informative lines, this will render nothing if value is None.

ifresult: (from HTMLResultRenderMixin) swallows the child tree if no result is available

ifnoresult: (from HTMLResultRenderMixin) renders its content only if a result table is available but contains no rows.

result: (from HTMLResultRenderMixin) a data function putting the current result into the context. If no result is available (e.g., because no query has been sent), `None` will be put into the context data.

queryseq: (from HTMLResultRenderMixin) a data function putting a sequence of (key, value) query parameters into the context data.

param: (from HTMLResultRenderMixin) a parameterised renderer that takes a python format string and formats the value of the param named in the content with it, for instance:

```
<span n:render="param %5.2f">rv</span>
```

resultmeta: (from service.SvcResult) a data function on results (i.e., this must be in an element with `n:data="result"`) putting a dictionary with at least the key `itemsMatched` into the context.

inputRec: (from service.SvcResult) a data function on results (i.e., this must be in an element with `n:data="result"`) putting a the parameters dictionary of the inputs table (typcially, the input parameters) into the context.

table: (from service.SvcResult) a data function on results (i.e., this must be in an element with `n:data="result"`) putting a the result's primary table into the context

Overriding the Default Response Template

Sometimes you want to change something in the appearance of a service that cannot be done with either service/@customCSS or the tricks described in How do I add an image to a query form?. You will then need to override the default-response template. As an example, you can refer to apfs/res/apfs_new RD; it has a custom template sitting next to the RD as `response.template`, which is declared as a custom service template in the service element via:

```
<template key="response">res/response.template</template>
```

(the path is relative to the RD's resdir).

What we wanted to do here is change then title depending on input parameters (sometimes it's "apparent places", sometimes "intermediate places", and I also wanted to have a warning if users chose a non-single star for the ephemeris generated, as that is probably inaccurate.

A word of warning: Do not introduce a custom template lightly. We do change `defaultresponse.html` occasionally, and you should probably follow these changes in your templates. So, before you do a custom, service-specific template, ask on the dachs-support mailing list if someone knows a smarter solution to your problem.

In the APFS case, what made us use a custom template is teh varying title; that's really hard to do in some other way because the title is service metadata, and the service is persistent within DaCHS, possibly even used concurrently by concurrent requests. Manipulating its metadata per-request is therefore a bad idea.

So, what did we do? First, get the default response template to have something to start with. We recommend to put it into a subdirectory `/res`, so assuming you are in the RD's resdir, you'd do:

```
$ mkdir -p res
$ gavo admin dumpDF templates/defaultresponse.html > res/response.html
```

That done, you can place the template declaration above into the service definition and start changing the template. Changes should be immediately visible on reload.

To give you an impression of the interplay between the template and the RD, here is a closer look at the apfs/res/apfs_new RD and the associated template at https://svn.ari.uni-heidelberg.de/svn/gavo/hdinputs/apfs/res/response.html There, we first had to add a custom render function rendering the title to the service:

```
<customRF name="apfsTitle">
  # wants n:data="result"
  if data.queryMeta.get("columnSet")==set(["equ"]):
    denom = "Apparent Places"
  elif data.queryMeta.get("columnSet")==set(["equ", "cio"]):
    denom = "Apparent and Intermediate Places"
  else:
    denom = "Intermediate Places"
  star = str(data.inputTable.getMeta("forStar"))
  return ctx.tag[denom+" for star \catname "+star]
</customRF>
```

We're getting a parameter from `queryMeta` here, an ugly piece of misdesign keeping lots of information on the request. Before you repeat that stunt, ask on dachs-support, because there are most certainly better ways do do that. The remaining part is a fairly straightforward nevow renderer, perhaps except for the call of the catname macro, which is here because in this particular case the

render function is used by two services and we've put it into a STREAM to allow reuse.

In comparison, the necessary changes in `response.template` are minor:

```
<h1 n:render="apfsTitle"/>
```

is all that's needed. The template goes on saying:

```
<p n:data="inputRec" n:render="mapping">
  ICRS position is &alpha;=<n:slot name="alpha"/>,
  &delta;=<n:slot name="delta"/>.
</p>
```

This displays the position of the star for which the ephemeris is computed for. We do this here because the input to this service is something like a catalog number, and alpha and delta are actually computed by the service's input data descriptor.

Finally, when the result is in context, we do:

```
<p style="border:2px solid red; max-width:400px; padding:5pt"
  n:render="multiplicityWarning"/>
```

We would do this today by attaching a `_warning` meta item on the result, but since that was not available when we wrote the RD, we added a custom render function like this:

```
<customRF name="multiplicityWarning">
  # wants n:data="result"
  if data.inputTable.getMeta("multiple"):
    return ctx.tag["Warning:  This star"
      " is identified by Hipparcos as being a ",
      T.strong["multiple star."],
      " This means that proper motions given in the Hipparcos"
      " catalogue are unreliable when applied to extended periods"
      " of time.  Thus, the places given below will be severely"
      " wrong unless the orbital period of the object is sufficiently"
      " long (i.e., hundreds of years)."]
  return ""
</customRF>
```

The `multiple` meta is put to the table further up in the computation. Note how we leave as much as possible to the template; for instance, the style is set there rather in the render function.

Custom Template, Custom Core

For exotic, custom services you can combine a custom template with a python (or custom) core. Here is a stripped-down example to get you started; if you want to run this, you will have to install pyephem (if you port this to astropy and share your port with us, you'll receive good Karma).

First, the RD; the thing to watch out for is the creation of the output table, getting the input params, and setting the output params. Note that you'd normally generate table rows, which would be dicts added through addRow:

```
<resource schema="neptune">
  <!-- Metadata omitted.  Thou shalt not do this. -->
  <service id="s">
    <template key="response">response.html</template>
    <pythonCore>
      <inputTable>
        <inputKey name="for_date" type="timestamp"
          tablehead="Date"
          description="Date to compute ephemeris for">
          <property key="defaultForForm">1969-06-04</property>
        </inputKey>
        <inputKey name="lat" unit="deg" tablehead="Latitude">
          <property key="defaultForForm">49.4294</property>
        </inputKey>
        <inputKey name="long" unit="deg" tablehead="Longitude">
          <property key="defaultForForm">11.00417</property>
        </inputKey>
      </inputTable>

      <outputTable>
        <param name="next_rising" type="timestamp"/>
        <param name="next_setting" type="timestamp"/>
      </outputTable>

      <coreProc>
        <setup>
          <code>
            import ephem
          </code>
        </setup>
        <code>
          res = rsc.TableForDef(self.outputTable)
          n = ephem.Neptune(inputTable.getParam("for_date"))
          obs = ephem.Observer()
          obs.lat = inputTable.getParam("lat")*DEG
          obs.long = inputTable.getParam("long")*DEG
          obs.date = inputTable.getParam("for_date")
          res.setParam("next_rising", obs.next_rising(n).datetime())
          res.setParam("next_setting", obs.next_setting(n).datetime())
          return res
        </code>
```

```
      </coreProc>
    </pythonCore>
  </service>
</resource>
```

And here's the template:

```
<html xmlns="http://www.w3.org/1999/xhtml"
              xmlns:n="http://nevow.com/ns/nevow/0.1">
<head n:render="commonhead">
<title>Nepune rising (and setting)</title>
<style type="text/css">
  .horizontime {
    font-size: 300%;
    color: #ff6688;
    font-family: fantasy;
    white-space: nowrap;
  }
</style>
</head>
<body n:data="result">
  <p style="text-align:center">
    Neptune rising:
      <span class="horizontime" n:render="param %s">next_rising</span>
  </p>
  <p style="text-align:center">
    Neptune setting:
      <span class="horizontime" n:render="param %s">next_setting</span>
  </p>
  <n:invisible n:render="form genForm"/>
</body>
</html>
```

Suggested exercise: write two renderers such that you can write:

```
<span class="horizontime" n:render="rising_time"/>
```

and get the rising time formatted the way you want it. If you get stuck, ask back, and we'll provide a solution and a new exercise.

[1] One reason *not* to declare a DTD is that careless clients other than web browsers (on system with misconfigured system catalogs) might stumble across such a page and try do download the DTD; the W3C was so pissed with clients doing that that they are now severely slowing them down by delaying delivery of the DTDs, which sometimes leads to surprising client behaviour.