QWeb

QWeb is the template engine used by the OpenERP Web Client. It is an XML-based templating language, similar to Genshi, Thymeleaf or Facelets with a few pecu

- It's implemented fully in javascript and rendered in the browser.
- Each template file (XML files) contains multiple templates, where template engine usually have a 1:1 mapping between template files and templates.
- It has special support in OpenERP Web's `Widget`, though it can be used outside of OpenERP Web (and it's possible to use `Widget` without relying on the QWeb

The rationale behind using QWeb instead of a more popular template syntax is that its extension mechanism is very similar to the openerp view inheritance mecha openerp views a QWeb template is an xml tree and therefore xpath or dom manipulations are easy to performs on it.

Here's an example demonstrating most of the basic QWeb features:

```
<templates>
  <div t-name="example_template" t-attf-class="base #{cls}">
    <h4 t-if="title"><t t-esc="title"/></h4>
    <ul>
      <li t-foreach="items" t-as="item" t-att-class="item_parity">
        <t t-call="example_template.sub">
          <t t-set="arg" t-value="item_value"/>
        </t>
      </li>
    </ul>
  </div>
  <t t-name="example_template.sub">
    <t t-esc="arg.name"/>
    <dl>
      <t t-foreach="arg.tags" t-as="tag" t-if="tag_index lt 5">
        <dt><t t-esc="tag"/></dt>
        <dd><t t-esc="tag_value"/></dd>
      </t>
    </dl>
  </t>
</templates>
```

rendered with the following context:

```
{
    "class1": "foo",
    "title": "Random Title",
    "items": [
        { "name": "foo", "tags": {"bar": "baz", "qux": "quux"} },
        { "name": "Lorem", "tags": {
                "ipsum": "dolor",
                "sit": "amet",
                "consectetur": "adipiscing",
                "elit": "Sed",
                "hendrerit": "ullamcorper",
                "ante": "id",
                "vestibulum": "Lorem",
                "ipsum": "dolor",
                "sit": "amet"
            }
        }
    ]
}
```

will yield this section of HTML document (reformated for readability):

```
<div class="base foo">
    <h4>Random Title</h4>
    <ul>
        <li class="even">
            foo
            <dl>
                <dt>bar</dt>
                <dd>baz</dd>
                <dt>qux</dt>
                <dd>quux</dd>
            </dl>
        </li>
        <li class="odd">
            Lorem
            <dl>
                <dt>ipsum</dt>
                <dd>dolor</dd>
                <dt>sit</dt>
                <dd>amet</dd>
                <dt>consectetur</dt>
                <dd>adipiscing</dd>
                <dt>elit</dt>
                <dd>Sed</dd>
                <dt>hendrerit</dt>
                <dd>ullamcorper</dd>
            </dl>
        </li>
    </ul>
```

```
</div>
```

API

While QWeb implements a number of attributes and methods for customization and configuration, only two things are really important to the user:

class `QWeb2.Engine`()

The QWeb "renderer", handles most of QWeb's logic (loading, parsing, compiling and rendering templates).

OpenERP Web instantiates one for the user, and sets it to `instance.web.qweb`. It also loads all the template files of the various modules into that QWeb instar

A **`QWeb2.Engine()`** also serves as a "template namespace".

`QWeb2.Engine.render`(*template*[, *context*])

Renders a previously loaded template to a String, using `context` (if provided) to find the variables accessed during template rendering (e.g. strings to displ

Arguments:
- **template** (*String*) – the name of the template to render
- **context** (*Object*) – the basic namespace to use for template rendering

Returns: String

The engine exposes an other method which may be useful in some cases (e.g. if you need a separate template namespace with, in OpenERP Web, Kanban viev **`QWeb2.Engine()`** instance so their templates don't collide with more general "module" templates):

`QWeb2.Engine.add_template`(*templates*)

Loads a template file (a collection of templates) in the QWeb instance. The templates can be specified as:

An XML string
: QWeb will attempt to parse it to an XML document then load it.

A URL
: QWeb will attempt to download the URL content, then load the resulting XML string.

A `Document` or `Node`
: QWeb will traverse the first level of the document (the child nodes of the provided root) and load any named template or template override.

A **`QWeb2.Engine()`** also exposes various attributes for behavior customization:

`QWeb2.Engine.prefix`

Prefix used to recognize *directives* during parsing. A string. By default, `t`.

`QWeb2.Engine.debug`

Boolean flag putting the engine in "debug mode". Normally, QWeb intercepts any error raised during template execution. In debug mode, it leaves all exc without intercepting them.

`QWeb2.Engine.jQuery`

The jQuery instance used during *template inheritance* processing. Defaults to `window.jQuery`.

`QWeb2.Engine.preprocess_node`

A `Function`. If present, called before compiling each DOM node to template code. In OpenERP Web, this is used to automatically translate text content a in templates. Defaults to `null`.

Directives

A basic QWeb template is nothing more than an XHTML document (as it must be valid XML), which will be output as-is. But the rendering can be customized wit called "directives". Directives are attributes elements prefixed by **`prefix`** (this document will use the default prefix `t`, as does OpenERP Web).

A directive will usually control or alter the output of the element it is set on. If no suitable element is available, the prefix itself can be used as a "no-operation" elem supporting directives (or internal content, which will be rendered). This means:

```
<t>Something something</t>
```

will simply output the string "Something something" (the element itself will be skipped and "unwrapped"):

```
var e = new QWeb2.Engine();
e.add_template('<templates>\
    <t t-name="test1"><t>Test 1</t></t>\
    <t t-name="test2"><span>Test 2</span></t>\
</templates>');
e.render('test1'); // Test 1
e.render('test2'); // <span>Test 2</span>
```

Note:

The conventions used in directive descriptions are the following:

- directives are described as compound functions, potentially with optional sections. Each section of the function name is an attribute of the element bearing the

- a special parameter is `BODY`, which does not have a name and designates the content of the element.
- special parameter types (aside from `BODY` which remains untyped) are `Name`, which designates a valid javascript variable name, `Expression` which designates a expression, and `Format` which designates a Ruby-style format string (a literal string with `#{Expression}` inclusions executed and replaced by their result)

Note:

`Expression` actually supports a few extensions on the javascript syntax: because some syntactic elements of javascript are not compatible with XML and must be substitutions are performed from forms which don't need to be escaped. Thus the following "keyword operators" are available in an `Expression`: `and` (maps to `&&`), `gt` (maps to `>`), `gte` (maps to `>=`), `lt` (maps to `<`) and `lte` (maps to `<=`).

Defining Templates

`t-name=name`

Parameters: **name** (*String*) –
an arbitrary javascript string. Each template name is unique in a given **`QWeb2.Engine()`** instance, defining a new template with an existing na the previous one without warning.
When multiple templates are related, it is customary to use dotted names as a kind of "namespace" e.g. `foo` and `foo.bar` which will be used ei sub-widget of the widget used by `foo`.

Templates can only be defined as the children of the document root. The document root's name is irrelevant (it's not checked) but is usually `<templates>` for s

```
<templates>
    <t t-name="template1">
        <!-- template code -->
    </t>
</templates>
```

t-name can be used on an element with an output as well:

```
<templates>
    <div t-name="template2">
        <!-- template code -->
    </div>
</templates>
```

which ensures the template has a single root (if a template has multiple roots and is then passed directly to jQuery, odd things occur).

Output

`t-esc=content`

Parameters: **content** (*Expression*) –

Evaluates, html-escapes and outputs `content`.

`t-escf=content`

Parameters: **content** (*Format*) –

Similar to *t-esc* but evaluates a `Format` instead of just an expression.

`t-raw=content`

Parameters: **content** (*Expression*) –

Similar to *t-esc* but does *not* html-escape the result of evaluating `content`. Should only ever be used for known-secure content, or will be an XSS attack vector.

`t-rawf=content`

Parameters: **content** (*Format*) –

`Format`-based version of *t-raw*.

`t-att=map`

Parameters: **map** (*Expression*) –

Evaluates `map` expecting an `Object` result, sets each key:value pair as an attribute (and its value) on the holder element:

```
<span t-att="{foo: 3, bar: 42}"/>
```

will yield

```
<span foo="3" bar="42"/>
```

`t-att-ATTNAME=value`

Parameters:
- **ATTNAME** (*Name*) –
- **value** (*Expression*) –

Evaluates `value` and sets it on the attribute `ATTNAME` on the holder element.

If `value`'s result is `undefined`, suppresses the creation of the attribute.

`t-attf-ATTNAME=value`

Parameters:
- **ATTNAME** (*Name*) –
- **value** (*Format*) –

Similar to *t-att-** but the value of the attribute is specified via a `Format` instead of an expression. Useful for specifying e.g. classes mixing literal classes and com

Flow Control

`t-set=name (t-value=value | BODY)`

Parameters:
- **name** (*Name*) –
- **value** (*Expression*) –
- **BODY** –

Creates a new binding in the template context. If `value` is specified, evaluates it and sets it to the specified `name`. Otherwise, processes `BODY` and uses that instea

`t-if=condition`

Parameters: **condition** (*Expression*) –

Evaluates `condition`, suppresses the output of the holder element and its content of the result is falsy.

`t-foreach=iterable [t-as=name]`

Parameters:
- **iterable** (*Expression*) –
- **name** (*Name*) –

Evaluates `iterable`, iterates on it and evaluates the holder element and its body once per iteration round.

If `name` is not specified, computes a `name` based on `iterable` (by replacing non-`Name` characters by _).

If `iterable` yields a `Number`, treats it as a range from 0 to that number (excluded).

While iterating, *t-foreach* adds a number of variables in the context:

`#{name}`
If iterating on an array (or a range), the current value in the iteration. If iterating on an *object*, the current key.

`#{name}_all`
The collection being iterated (the array generated for a `Number`)

`#{name}_value`
The current iteration value (current item for an array, value for the current item for an object)

`#{name}_index`
The 0-based index of the current iteration round.

`#{name}_first`
Whether the current iteration round is the first one.

`#{name}_parity`
`"odd"` if the current iteration round is odd, `"even"` otherwise. `0` is considered even.

`t-call=template [BODY]`

Parameters:
- **template** (*String*) –
- **BODY** –

Calls the specified `template` and returns its result. If `BODY` is specified, it is evaluated *before* calling `template` and can be used to specify e.g. parameters. This u
call-template with with-param in XSLT.

Template Inheritance and Extension

`t-extend=template BODY`

Parameters: **template** (*String*) – name of the template to extend

Works similarly to OpenERP models: if used on its own, will alter the specified template in-place; if used in conjunction with *t-name* will create a new templat
one as a base.

`BODY` should be a sequence of *t-jquery* alteration directives.

Note:
The inheritance in the second form is *static*: the parent template is copied and transformed when *t-extend* is called. If it is altered later (by a *t-extend* without a
changes will *not* appear in the "child" templates.

`t-jquery=selector [t-operation=operation] BODY`

Parameters:
- **selector** (*String*) – a CSS selector into the parent template
- **operation** – one of `append`, `prepend`, `before`, `after`, `inner` or `replace`.
- **BODY** – `operation` argument, or alterations to perform

- If `operation` is specified, applies the selector to the parent template to find a *context node*, then applies `operation` (as a jQuery operation) to the *context n*
as parameter.

Note:
`replace` maps to jQuery's replaceWith(newContent), `inner` maps to html(htmlString).

- If `operation` is not provided, `BODY` is evaluated as javascript code, with the *context node* as `this`.

Warning:
While this second form is much more powerful than the first, it is also much harder to read and maintain and should be avoided. It is usually possible to e
replace it with a sequence of `t-jquery:t-operation:`.

Escape Hatches / debugging

`t-log=expression`

Parameters: **expression** (*Expression*) –

Evaluates the provided expression (in the current template context) and logs its result via `console.log`.

`t-debug`

Injects a debugger breakpoint (via the `debugger;` statement) in the compiled template output.

`t-js=context BODY`

Parameters:
- **context** (*Name*) –
- **BODY** – javascript code

Injects the provided `BODY` javascript code into the compiled template, passing it the current template context using the name specified by `context`.