OpenERP Web Documentation

Release 6.1.1

OpenERP S.A.

October 19, 2016

Contents

1 Getting Started with OpenERP Web **3**
- 1.1 Installing . . . 3
- 1.2 Launching . . . 3

2 Deploying OpenERP Web **5**
- 2.1 In-depth configuration . . . 5
- 2.2 Deployment Options . . . 5

3 OpenERP Web as a widgets provider **7**

4 Developing OpenERP Web Addons **9**
- 4.1 Structure . . . 9
- 4.2 Namespacing . . . 10
- 4.3 Creating new standard roles . . . 11
- 4.4 Frequent development tasks . . . 13
- 4.5 Translations . . . 15
- 4.6 Utility behaviors . . . 17
- 4.7 Testing . . . 17

5 OpenERP Web Core and standard addons **19**
- 5.1 Standard Views . . . 19
- 5.2 Internal API Doc . . . 23
- 5.3 Testing . . . 24

6 The OpenERP Web open-source project **25**
- 6.1 Getting involved . . . 25
- 6.2 OpenERP Web over time . . . 30

7 Main differences with the 6.0 client **31**

8 Indices and tables **33**

Contents:

CHAPTER 1

Getting Started with OpenERP Web

1.1 Installing

1.2 Launching

Deploying OpenERP Web

2.1 In-depth configuration

SSL, basic proxy (link to relevant section), links to sections and example files for various servers and proxies, WSGI integration/explanation (if any), ...

2.2 Deployment Options

2.2.1 Serving via WSGI

Apache mod_wsgi

NGinx mod_wsgi

uWSGI

Gunicorn

FastCGI, SCGI, or AJP

2.2.2 Behind a proxy

Apache mod_proxy

NGinx HttpProxy

CHAPTER 3

OpenERP Web as a widgets provider

- Using a readonly view as a widget
  - Site example
  - iGoogle example
  - social site example e.g. Facebook app?
- Write-access widgets (e.g. contact form)
- Multiple widgets on the same page
- JSON-RPC2 API description for third-parties?

CHAPTER 4

Developing OpenERP Web Addons

An OpenERP Web addon is simply a Python package with an openerp descriptor (a `__openerp__.py` file) which follows a few structural and namespacing rules.

4.1 Structure

```
<addon name>
  +-- __openerp__.py
  +-- controllers/
  +-- static/
       +-- lib/
       +-- src/
            +-- css/
            +-- img/
            +-- js/
            +-- xml/
       +-- test/
  +-- test/
```

`__openerp__.py` The addon's descriptor, contains the following information:

> **`name: str`** The addon name, in plain, readable english
>
> **`version: str`** The addon version, following Semantic Versioning rules
>
> **`depends: [str]`** A list of addons this addon needs to work correctly. `base` is an implied dependency if the list is empty.
>
> **`css: [str]`** An ordered list of CSS files this addon provides and needs. The file paths are relative to the addon's root. Because the Web Client *may* perform concatenations and other various optimizations on CSS files, the order is important.
>
> **`js: [str]`** An ordered list of Javascript files this addon provides and needs (including dependencies files). As with CSS files, the order is important as the Web Client *may* perform contatenations and minimizations of files.
>
> **`active: bool`** Whether this addon should be enabled by default any time it is found, or whether it will be enabled through other means (on a by-need or by-installation basis for instance).

`controllers/` All of the Python controllers and JSON-RPC endpoints.

`static/` The static files directory, may be served via a separate web server.

`static/lib/` Third-party libraries used by the addon.

`static/src/{css,js,img,xml}` Location for (respectively) the addon's static CSS files, its JS files, its various image resources as well as the template files

`static/test` Javascript tests files

`test/` The directories in which all tests for the addon are located.

Some of these are guidelines (and not enforced by code), but it's suggested that these be followed. Code which does not fit into these categories can go wherever deemed suitable.

4.2 Namespacing

4.2.1 Python

Because addons are also Python packages, they're inherently namespaced and nothing special needs to be done on that front.

4.2.2 JavaScript

The JavaScript side of an addon has to live in the namespace `openerp.$addon_name`. For instance, everything created by the addon `base` lives in `openerp.base`.

The root namespace of the addon is a function which takes a single parameter `openerp`, which is an OpenERP client instance. Objects (as well as functions, registry instances, etc...) should be added on the correct namespace on that object.

The root function will be called by the OpenERP Web client when initializing the addon.

```
// root namespace of the openerp.example addon
/** @namespace */
openerp.example = function (openerp) {
    // basic initialization code (e.g. templates loading)
    openerp.example.SomeClass = openerp.base.Class.extend(
        /** @lends openerp.example.SomeClass# */{
        /**
         * Description for SomeClass's constructor here
         *
         * @constructs
         */
        init: function () {
            // SomeClass initialization code
        }
        // rest of SomeClass
    });

    // access an object in an other addon namespace to replace it
    openerp.base.SearchView = openerp.base.SearchView.extend({
        init: function () {
            this._super.apply(this, arguments);
            console.log('Search view initialized');
        }
    });
}
```

4.3 Creating new standard roles

4.3.1 Widget

This is the base class for all visual components. It provides a number of services for the management of a DOM subtree:

- Rendering with QWeb
- Parenting-child relations
- Life-cycle management (including facilitating children destruction when a parent object is removed)
- DOM insertion, via jQuery-powered insertion methods. Insertion targets can be anything the corresponding jQuery method accepts (generally selectors, DOM nodes and jQuery objects):

  `appendTo()` Renders the widget and inserts it as the last child of the target, uses .appendTo()

  `prependTo()` Renders the widget and inserts it as the first child of the target, uses .prependTo()

  `insertAfter()` Renders the widget and inserts it as the preceding sibling of the target, uses .insertAfter()

  `insertBefore()` Renders the widget and inserts it as the following sibling of the target, uses .insertBefore()

`Widget()` inherits from `SessionAware()`, so subclasses can easily access the RPC layers.

Subclassing Widget

`Widget()` is subclassed in the standard manner (via the `extend()` method), and provides a number of abstract properties and concrete methods (which you may or may not want to override). Creating a subclass looks like this:

```
var MyWidget = openerp.base.Widget.extend({
    // QWeb template to use when rendering the object
    template: "MyQWebTemplate",

    init: function(parent) {
        this._super(parent);
        // insert code to execute before rendering, for object
        // initialization
    },
    start: function() {
        this._super();
        // post-rendering initialization code, at this point
        // ``this.$element`` has been initialized
        this.$element.find(".my_button").click(/* an example of event binding * /);

        // if ``start`` is asynchronous, return a promise object so callers
        // know when the object is done initializing
        return this.rpc(/* ... */)
    }
});
```

The new class can then be used in the following manner:

```
// Create the instance
var my_widget = new MyWidget(this);
// Render and insert into DOM
my_widget.appendTo(".some-div");
```

After these two lines have executed (and any promise returned by `appendTo` has been resolved if needed), the widget is ready to be used.

Note: the insertion methods will start the widget themselves, and will return the result of `start()`.

If for some reason you do not want to call these methods, you will have to first call `render()` on the widget, then insert it into your DOM and start it.

If the widget is not needed anymore (because it's transient), simply terminate it:

```
my_widget.stop();
```

will unbind all DOM events, remove the widget's content from the DOM and destroy all widget data.

4.3.2 Views

Views are the standard high-level component in OpenERP. A view type corresponds to a way to display a set of data (coming from an OpenERP model).

In OpenERP Web, views are standard objects registered against a dedicated object registry, so the `ViewManager()` knows where to find and how to call them.

Although not mandatory, it is recommended that views inherit from `openerp.base.View()`, which provides a view useful services to its children.

Registering a view

This is the first task to perform when creating a view, and the simplest by far: simply call `openerp.base.views.add(name, object_path)` to register the object of path `object_path` as the view for the view name `name`.

The view name is the name you gave to your new view in the OpenERP server.

From that point onwards, OpenERP Web will be able to find your object and instantiate it.

Standard view behaviors

In the normal OpenERP Web flow, views have to implement a number of methods so view managers can correctly communicate with them:

`start()` This method will always be called after creating the view (via its constructor), but not necessarily immediately.

It is called with no arguments and should handle the heavy setup work, including remote call (to load the view's setup data from the server via e.g. `fields_view_get`, for instance).

`start` should return a promise object which *must* be resolved when the view's setup is completed. This promise is used by view managers to know when they can start interacting with the view.

`do_hide()` Called by the view manager when it wants to replace this view by an other one, but wants to keep this view around to re-activate it later.

Should put the view in some sort of hibernation mode, and *must* hide its DOM elements.

`do_show()` Called when the view manager wants to re-display the view after having hidden it. The view should refresh its data display upon receiving this notification

do_search(domain: Array, context: Object, group_by: Array) If the view is searchable, this method is called to notify it of a search against it.

It should use the provided query data to perform a search and refresh its internal content (and display).

All views are searchable by default, but they can be made non-searchable by setting the property `searchable` to `false`.

This can be done either on the view class itself (at the same level as defining e.g. the `start` method) or at the instance level (in the class's `init`), though you should generally set it on the class.

4.4 Frequent development tasks

There are a number of tasks which OpenERP Web developers do or will need to perform quite regularly. To make these easier, we have written a few guides to help you get started:

4.4.1 Creating a new client action

Client actions are the client-side of OpenERP's "Server Actions": instead of allowing for semi-arbitrary code to be executed in the server, they allow for execution of client-customized code.

On the server side, a client action is an action of type `ir.actions.client`, which has (at most) two properties: a mandatory `tag`, which is an arbitrary string by which the client will identify the action, and an optional `params` which is simply a map of keys and values sent to the client as-is (this way, client actions can be made generic and reused in multiple contexts).

General Structure

In the OpenERP Web code, a client action only requires two pieces of information:

- Mapping the action's `tag` to an OpenERP Web object
- The OpenERP Web object itself, which must inherit from `openerp.web.Widget()`

Our example will be the actual code for the widgets client action (a client action displaying a `res.widget` object, used in the homepage dashboard of the web client):

```
// Registers the object 'openerp.web_dashboard.Widget' to the client
// action tag 'board.home.widgets'
openerp.web.client_actions.add(
    'board.home.widgets', 'openerp.web_dashboard.Widget');
// This object inherits from View, but only Widget is required
openerp.web_dashboard.Widget = openerp.web.View.extend({
    template: 'HomeWidget'
});
```

At this point, the generic `Widget` lifecycle takes over, the template is rendered, inserted in the client DOM, bound on the object's `$element` property and the object is started.

If the client action takes parameters, these parameters are passed in as a second positional parameter to the constructor:

```
init: function (parent, params) {
    // execute the Widget's init
    this._super(parent);
    // board.home.widgets only takes a single param, the identifier of the
    // res.widget object it should display. Store it for later
    this.widget_id = params.widget_id;
}
```

More complex initialization (DOM manipulations, RPC requests, ...) should be performed in the `start()` method.

Note: As required by `Widget`'s contract, if `start` executes any asynchronous code it should return a `$.Deferred` so callers know when it's ready for interaction.

Although generally speaking client actions are not really interacted with.

```
start: function () {
    return $.when(
        this._super(),
        // Simply read the res.widget object this action should display
        new openerp.web.DataSet(this, 'res.widget').read_ids(
            [this.widget_id], ['title'], this.on_widget_loaded));
}
```

The client action can then behave exactly as it wishes to within its root (`this.$element`). In this case, it performs further renderings once its widget's content is retrieved:

```
on_widget_loaded: function (widgets) {
    var widget = widgets[0];
    var url = _.sprintf(
        '/web_dashboard/widgets/content?session_id=%s&widget_id=%d',
        this.session.session_id, widget.id);
    this.$element.html(QWeb.render('HomeWidget.content', {
        widget: widget,
        url: url
    }));
}
```

4.4.2 Adding a sidebar to a view

Initialization

Each view has the responsibility to create its sidebar (or not) if and only if the `sidebar` flag is set in its options.

In that case, it should use the `sidebar_id` value (from its options) to initialize the sidebar at the right position in the DOM:

```
if (this.options.sidebar && this.options.sidebar_id) {
    this.sidebar = new openerp.web.Sidebar(this, this.options.sidebar_id);
    this.sidebar.start();
}
```

Because the sidebar is an old-style widget, it must be started after being initialized.

Sidebar communication protocol

In order to behave correctly, a sidebar needs informations from its parent view.

This information is extracted via a very basic protocol consisting of a property and two methods:

dataset
the view's dataset, used to fetch the currently active model and provide it to remote action handlers as part of the basic context

get_selected_ids()
Used to query the parent view for the set of currently available record identifiers. Used to setup the basic context's `active_id` and `active_ids` keys.

> **Warning:** *get_selected_ids()* must return at least one id

Returns an array of at least one id

Return type Array<Number>

sidebar_context()
Queries the view for additional context data to provide to the sidebar.

`View()` provides a default NOOP implementation, which simply resolves to an empty object.

Returns a promise yielding an object on success, this object is mergeed into the sidebar's own context

Return type $.Deferred<Object>

Programmatic folding and unfolding

The sidebar object starts folded. It provides three methods to handle its folding status:

do_toggle()
Toggles the status of the sidebar

do_fold()
Forces the sidebar closed if it's currently open

do_unfold()
Forces the sidebar open if it's currently closed

4.5 Translations

OpenERP Web should provide most of the tools needed to correctly translate your addons via the tool of your choice (OpenERP itself uses Launchpad's own translation tool.

4.5.1 Making strings translatable

QWeb

QWeb automatically marks all text nodes (any text which is not in an XML attribute and not part of an XML tag) as translatable, and handles the replacement for you. There is nothing special to do to mark template text as translatable

JavaScript

OpenERP Web provides two functions to translate human-readable strings in javascript code. These functions should be "imported" in your module by aliasing them to their bare name:

```
var _t = openerp.web._t,
   _tl = openerp.web._tl;
```

importing those functions under any other name is not guaranteed to work.

Note: only import them if necessary, and only the necessary one(s), no need to clutter your module's namespace for nothing

`openerp.web._t(s)`
Base translation function, eager, works much like `gettext(3)`

Return type String

`openerp.web._lt(s)`
Lazy equivalent to `_t()`, this function will postpone fetching the translation to its argument until the last possible moment.

To use in contexts evaluated before the translation database can be fetched, usually your module's toplevel and the attributes of classes defined in it (class attributes, not instance attributes set in the constructor).

Return type LazyString

Text formatting & translations

A difficulty when translating is integrating data (from the code) into the translated string. In OpenERP Web addons, this should be done by wrapping the text to translate in an `sprintf(3)` call. For OpenERP Web, `sprintf(3)` is provided by underscore.string.

As much as possible, you should use the "named argument" form of sprintf:

```
var translated_string = _.str.sprintf(
    _t("[%(first_record)d to %(last_record)d] of %(records_count)d"), {
        first_record: first + 1,
        last_record: last,
        records_count: total
    }));
```

named arguments make the string to translate much clearer for translators, and allows them to "move" sections around based on the requirements of their language (not all language order text like english).

Named arguments are specified using the following pattern: `%($name)$type` where

`$name` the name of the argument, this is the key in the object/dictionary provided as second parameter to `sprintf`

`$type` a type/format specifier, see the list for all possible types.

Note: positional arguments are acceptable if the translated string has *a single* argument and its content is easy to guess from the text around it. Named arguments should still be preferred.

Warning: you should *never* use string concatenation as it robs the translator of context and make result in a completely incorrect translation

Extracting strings

Once strings have been marked for translation, they need to be extracted into POT (Portable Object Template) files, from which most translation tools can build a database.

This can be done via the provided **gen_translations.sh**.

It can be called either as *gen_translations.sh -a* or by providing two parameters, a path to the addons and the complete path in which to put the extracted POT file.

-a
Extracts translations from all standard OpenERP Web addons (addons bundled with OpenERP Web itself) and puts the extracted templates into the right directory for Rosetta to handle them

4.6 Utility behaviors

4.6.1 JavaScript

- All javascript objects inheriting from `openerp.base.BasicConroller()` will have all methods starting with `on_` or `do_` bound to their `this`. This means they don't have to be manually bound (via `_.bind` or `$.proxy`) in order to be useable as bound event handlers (event handlers keeping their object as `this` rather than taking whatever `this` object they were called with).

  Beware that this is only valid for methods starting with `do_` and `on_`, any other method will have to be bound manually.

4.7 Testing

4.7.1 Python

OpenERP Web uses unittest2 for its testing needs. We selected unittest2 rather than unittest for the following reasons:

- autodiscovery (similar to nose, via the `unit2` CLI utility) and pluggable test discovery.
- new and improved assertions (with improvements in type-specific inequality reportings) including pluggable custom types equality assertions
- neveral new APIs, most notably assertRaises context manager, cleanup function registration, test skipping and class- and module-level setup and teardown
- finally, unittest2 is a backport of Python 3's unittest. We might as well get used to it.

To run tests on addons (from the root directory of OpenERP Web) is as simple as typing `PYTHONPATH=. unit2 discover -s addons`[1]. To test an addon which does not live in the `addons` directory, simply replace `addons` by the directory in which your own addon lives.

[1] the `-s` parameter tells `unit2` to start trying to find tests in the provided directory (here we're testing addons). However a side-effect of that is to set the `PYTHONPATH` there as well, so it will fail to find (and import) `openerpweb`.

The `-t` parameter lets us set the `PYTHONPATH` independently, but it doesn't accept multiple values and here we really want to have both `.` and `addons` on the `PYTHONPATH`.

The solution is to set the `PYTHONPATH` to `.` on start, and the `start-directory` to `addons`. This results in a correct `PYTHONPATH` within `unit2`.

Note: unittest2 is entirely compatible with nose (or the other way around). If you want to use nose as your test runner (due to its addons for instance) you can simply install it and run `nosetests addons` instead of the `unit2` command, the result should be exactly the same.

4.7.2 Python

- Addons lifecycle (loading, execution, events, ...)
  - Python-side
  - JS-side
- Handling static files
- Overridding a Python controller (object?)
- Overridding a Javascript controller (object?)
- Extending templates .. how do you handle deploying static files via e.g. a separate lighttpd?
- Python public APIs
- QWeb templates description?
- OpenERP Web modules (from OpenERP modules)

CHAPTER 5

OpenERP Web Core and standard addons

- General organization and core ideas (design philosophies)
- Internal documentation, autodoc, Python and JS domains
- QWeb code documentation/description
- Documentation of the OpenERP APIs and choices taken based on that?
- Style guide and coding conventions (PEP8? More)
- Test frameworks in JS?

5.1 Standard Views

5.1.1 Search View

The OpenERP search view really is a sub-view, used in support of views acting on collections of records (list view or graph view, for instance).

Its main goal is to collect information from its widgets (themselves collecting information from the users) and make those available to the rest of the client.

The search view's root is `SearchView()`. This object should never need to be created or managed directly, its lifecycle should be driven by the `ViewManager()`.

The search view defines a number of internal and external protocols to communicate with the objects around and within it. Most of these protocols are informal, and types available for inheritance are more mixins than mandatory.

Events

`on_loaded`

Fires when the search view receives its view data (the result of `fields_view_get`). Hooking up before the event allows for altering view data before it can be used.

By the time `on_loaded` is done, the search view is guaranteed to be fully set up and ready to use.

`on_search`

Event triggered after a user asked for a search. The search view fires this event after collecting all input data (contexts, domains and group_by contexts). Note that the search view does *not* merge those (or otherwise evaluate them), they are returned as provided by the various inputs within the view.

`on_clear`

Triggered after a user asked for a form clearing.

Input management

An important concept in the search view is that of input. It is both an informal protocol and an abstract type that can be inherited from.

Inputs are widgets which can contain user data (a char widget for instance, or a selection box). They are capable of action and of reaction: `registration`

This is an input action. Inputs have to register themselves to the main view (which they receive as a constructor argument). This is performed by pushing themselves on the `openerp.base.SearchView.inputs` array.

`get_context`

An input reaction. When it needs to collect contexts, the view calls `get_context()` on all its inputs.

Inputs can react in the following manners:

- Return a context (an object), this is the "normal" response if the input holds a value.
- Return a value that evaluates as false (generally `null`). This value indicates the input does not contain any value and will not affect the results of the search.
- Raise `openerp.base.search.Invalid()` to indicate that it holds a value but this value can not be used in the search (because it is incorrectly formatted or nonsensical). Raising `Invalid()` is guaranteed to cancel the search process.

  `Invalid()` takes three mandatory arguments: an identifier (a name for instance), the invalid value, and a validation message indicating the issue.

`get_domain`

The second input reaction, the possible behaviors of inputs are the same as for `get_context`.

The `openerp.base.search.Input()` type implements registration on its own, but its implementations of `get_context` and `get_domain` simply raise errors and *must* be overridden.

One last action is for filters, as an activation order has to be kept on them for some controls (to establish the correct grouping sequence, for instance).

To that end, filters can call `openerp.base.Search.do_toggle_filter()`, providing themselves as first argument.

Filters calling `do_toggle_filter()` also need to implement a method called `is_enabled()`, which the search view will use to know the current status of the filter.

The search view automatically triggers a search after calls to `do_toggle_filter()`.

Life cycle

The search view has a pretty simple and linear life cycle, in three main steps:

`init()`

Nothing interesting happens here

`start()`

Called by the main view's creator, this is the main initialization step for the list view.

It begins with a remote call to fetch the view's descriptors (`fields_view_get`).

Once the remote call is complete, the `on_loaded` even happens, holding three main operations:

`make_widgets()`

Builds and returns the top-level widgets of the search view. Because it returns an array of widget lines (a 2-dimensional matrix of widgets) it should be called recursively by container widgets (`openerp.base.search.Group()` for instance).

`render()`

Called by the search view on all top-level widgets. Container widgets should recursively call this method on their own children widgets.

Widgets are provided with a mapping of `{name: value}` holding default values for the search view. They can freely pick their initial values from there, but must pass the mapping to their children widgets if they have any.

`start()`

The last operation of the search view startup is to initialize all its widgets in order. This is again done recursively (the search view starts its children, which have to start their own children).

`stop()`

Used before discarding a search view, allows the search view to disable its events and pass the message to its own widgets, gracefully shutting down the whole view.

Widgets

In a search view, the widget is simply a unit of display.

All widgets must be able to react to three events, which will be called in this order:

`render()`

Called with a map of default values. The widget must return a `String`, which is its HTML representation. That string can be empty (if the widget should not be represented).

Widgets are responsible for asking their children for rendering, and for passing along the default values.

`start()`

Called without arguments. At this point, the widget has been fully rendered and can set its events up, if any.

The widget is responsible for starting its children, if it has any.

`stop()`

Gives the widget the opportunity to unbind its events, remove itself from the DOM and perform any other cleanup task it may have.

Even if the widget does not do anything itself, it is responsible for shutting down its children.

An abstract type is available and can be inherited from, to simplify the implementation of those tasks:

Inputs

The search namespace (`openerp.base.search`) provides two more abstract types, used to implement input widgets:

- `openerp.base.search.Input()` is the most basic input type, it only implements *input registration*.

  If inherited from, descendant classes should not call its implementations of `get_context()` and `get_domain()`.

- `openerp.base.search.Field()` is used to implement more "field" widgets (which allow the user to input potentially complex values).

  It provides various services for its subclasses:

  - Sets up the field attributes, using attributes from the field and the view node.
  - It fills the widget with `Filter()` if the field has any child filter.
  - It automatically generates an identifier based on the field type and the field name, using `make_id()`.
  - It sets up a basic (overridable) `template` attribute, combined with the previous tasks, this makes subclasses of `Field()` render themselves "for free".
  - It provides basic implementations of `get_context` and `get_domain`, both hinging on the subclasses implementing `get_value()` (which should return a correct, converted Javascript value):

    `get_context()`

    Checks if the field has a non-`null` and non-empty (`String`) value, and that the field has a `context` attr.

    If both conditions are fullfilled, returns the context.

    `get_domain()`

    Only requires that the field has a non-`null` and non-empty value.

    If the field has a `filter_domain`, returns it immediately. Otherwise, builds a context using the field's name, the field `operator` and the field value, and returns it.

5.1.2 List View

OpenERP Web's list views don't actually exist as such in OpenERP itself: a list view is an OpenERP tree view in the `view_mode` form.

The overall purpose of a list view is to display collections of objects in two main forms: per-object, where each object is a row in and of itself, and grouped, where multiple objects are represented with a single row providing an aggregated view of all grouped objects.

These two forms can be mixed within a single list view, if needed.

The root of a list view is `openerp.base.ListView()`, which may need to be overridden (partially or fully) to control list behavior in non-view cases (when using a list view as sub-component of a form widget for instance).

Creation and Initialization

As with most OpenERP Web views, the list view's `init()` takes quite a number of arguments.

While most of them are the standard view constructor arguments (`view_manager`, `session`, `element_id`, `dataset` and an optional `view_id`), the list view adds a number of options for basic customization (without having to override methods or templates):

`selectable` (default: `true`) Indicates that the list view should allow records to be selected individually. Displays selection check boxes to the left of all record rows, and allows for the triggering of the *selection event*.

`deletable` (default: `true`) Indicates that the list view should allow records to be removed individually. Displays a deletion button to the right of all record rows, and allows for the triggering of the *deletion event*.

`header` (default: `true`) Indicates that list columns should bear a header sporting their name (for non-action columns).

`addable` (default: `"New"`) Indicates that a record addition/creation button should be displayed in the list's header, along with its label. Also allows for the triggering of the *record addition event*.

`sortable` (default: `true`) Indicates that the list view can be sorted per-column (by clicking on its column headers).

`reorderable` (default: `true`) Indicates that the list view records can be reordered (and re-sequenced) by drag and drop.

Events

Addition

The addition event is used to add a record to an existing list view. The default behavior is to switch to the form view, on a new record.

Addition behavior can be overridden by replacing the `do_add_record()` method.

Selection

The selection event is triggered when a given record is selected in the list view.

It can be overridden by replacing the `do_select()` method.

The default behavior is simply to hide or display the list-wise deletion button depending on whether there are selected records or not.

Deletion

The deletion event is triggered when the user tries to remove 1..n records from the list view, either individually or globally (via the header button).

Deletion can be overridden by replacing the `do_delete()` method. By default, this method calls `unlink()` in order to remove the records entirely.

Note: the list-wise deletion button (next to the record addition button) simply proxies to `do_delete()` after obtaining all selected record ids, but it is possible to override it alone by replacing `do_delete_selected()`.

5.2 Internal API Doc

5.2.1 Python

These classes should be moved to other sections of the doc as needed, probably.

5.3 Testing

5.3.1 Python

Testing for the OpenERP Web core is similar to *testing addons*: the tests live in `openerpweb.tests`, unittest2 is the testing framework and tests can be run via either unittest2 (`unit2 discover`) or via nose (`nosetests`).

Tests for the OpenERP Web core can also be run using `setup.py test`.

CHAPTER 6

The OpenERP Web open-source project

6.1 Getting involved

6.1.1 Translations

6.1.2 Bug reporting

6.1.3 Source code repository

6.1.4 Merge proposals

6.1.5 Coding issues and coding conventions

Javascript coding

These are a number of guidelines for javascript code. More than coding conventions, these are warnings against potentially harmful or sub-par constructs.

Ideally, you should be able to configure your editor or IDE to warn you against these kinds of issues.

Use `var` for *all* declarations

In javascript (as opposed to Python), assigning to a variable which does not already exist and is not explicitly declared (via `var`) will implicitly create a global variable. This is bad for a number of reasons:

- It leaks information outside function scopes
- It keeps memory of previous run, with potentially buggy behaviors
- It may conflict with other functions with the same issue
- It makes code harder to statically check (via e.g. IDE inspectors)

Note: It is perfectly possible to use `var` in `for` loops:

```
for (var i = 0; i < some_array.length; ++i) {
    // code here
}
```

this is not an issue

All local *and global* variables should be declared via `var`.

Note: generally speaking, you should not need globals in OpenERP Web: you can just declare a variable local to your top-level function. This way, if your widget/addon is instantiated several times on the same page (because it's used in embedded mode) each instance will have its own internal but global-to-its-objects data.

Do not leave trailing commas in object literals

While it is legal to leave trailing commas in Python dictionaries, e.g.

```
foo = {
    'a': 1,
    'b': 2,
}
```

and it's valid in ECMAScript 5 and most browsers support it in Javascript, you should *never* use trailing commas in Javascript object literals:

- Internet Explorer does *not* support trailing commas (at least until and including Internet Explorer 8), and trailing comma will cause hard-to-debug errors in it
- JSON does not accept trailing comma (it is a syntax error), and using them in object literals puts you at risks of using them in literal JSON strings as well (though there are few reasons to write JSON by hand)

Never use `for ... in` to iterate on arrays

Iterating over an object with for...in is a bit tricky already, it is far more complex than in Python (where it Just Works™) due to the interaction of various Javascript features, but to iterate on arrays it becomes downright deadly and errorneous: `for...in` really iterates over an *object*'s *properties*.

With an array, this has the following consequences:

- It does not necessarily iterate in numerical order, nor does it iterate in any kind of set order. The order is implementation-dependent and may vary from one run to the next depending on a number of reasons and implementation details.
- If properties are added to an array, to `Array.prototype` or to `Object.prototype` (the latter two should not happen in well-behaved javascript code, but you never know...) those properties *will* be iterated over by `for...in`. While `Object.hasOwnProperty` will guard against iterating prototype properties, they will not guard against properties set on the array instance itself (as memoizers for instance).

  Note that this includes setting negative keys on arrays.

For this reason, `for...in` should **never** be used on array objects. Instead, you should use either a normal `for` or (even better, unless you have profiled the code and found a hotspot) one of Underscore's array iteration methods (_.each, _.map, _.filter, etc...).

Underscore is guaranteed to be bundled and available in OpenERP Web scopes.

Use `hasOwnProperty` when iterating on an object with `for ... in`

`for...in` is Javascript's built-in facility for iterating over and object's properties.

It is also fairly tricky to use: it iterates over *all* non-builtin properties of your objects [1], which includes methods of an object's class.

As a result, when iterating over an object with `for...in` the first line of the body *should* generally be a call to Object.hasOwnProperty. This call will check whether the property was set directly on the object or comes from the object's class:

```
for(var key in ob) {
    if (!ob.hasOwnProperty(key)) {
        // comes from ob's class
        continue;
    }
    // do stuff with key
}
```

Since properties can be added directly to e.g. `Object.prototype` (even though it's usually considered bad style), you should not assume you ever know which properties `for...in` is going to iterate over.

An alternative is to use Underscore's iteration methods, which generally work over objects as well as arrays:

Instead of

```
for (var key in ob) {
    if (!ob.hasOwnProperty(key)) { continue; }
    var value = ob[key];
    // Do stuff with key and value
}
```

you could write:

```
_.each(ob, function (value, key) {
    // do stuff with key and value
});
```

and not worry about the details of the iteration: underscore should do the right thing for you on its own [2].

6.1.6 Writing documentation

The OpenERP Web project documentation uses Sphinx for the literate documentation (this document for instance), the development guides (for Python and Javascript alike) and the Python API documentation (via autodoc).

For the Javascript API, documentation should be written using the JsDoc Toolkit.

Guides and main documentation

The meat and most important part of all documentation. Should be written in plain English, using reStructuredText and taking advantage of Sphinx's extensions, especially cross-references.

[1] More precisely, it iterates over all *enumerable* properties. It just happens that built-in properties (such as `String.indexOf` or `Object.toString`) are set to non-enumerable.

The enumerability of a property can be checked using Object.propertyIsEnumeable.

Before ECMAScript 5, it was not possible for user-defined properties to be non-enumerable in a portable manner. ECMAScript 5 introduced Object.defineProperty which lets user code create non-enumerable properties (and more, read-only properties for instance, or implicit getters and setters). However, support for these is not fully complete at this point, and they are not being used in OpenERP Web code anyway.

[2] While using underscore is generally the preferred method (simpler, more reliable and easier to write than a *correct* `for...in` iteration), it is also probably slower (due to the overhead of calling a bunch of functions).

As a result, if you profile some code and find out that an underscore method adds unacceptable overhead in a tight loop, you may want to replace it with a `for...in` (or a regular `for` statement for arrays).

Python API Documentation

All public objects in Python code should have a docstring written in RST, using Sphinx's Python domain [3]:

- Functions and methods documentation should be in their own docstring, using Sphinx's info fields

  For parameters types, built-in and stdlib types should be using the combined syntax:

```
:param dict foo: what the purpose of foo is
```

  unless a more extensive explanation needs to be given (e.g. the specification that the input should be a list of 3-tuple needs to use `:type:` even though all types involved are built-ins). Any other type should be specified in full using the `:type:` field

```
:param foo: what the purpose of foo is
:type foo: some.addon.Class
```

  Mentions of other methods (including within the same class), modules or types in descriptions (of anything, including parameters) should be cross-referenced.

- Classes should likewise be documented using their own docstring, and should include the documentation of their construction (`__init__` and `__new__`), using the info fields as well.
- Attributes (class and instance) should be documented in their class's docstring via the `.. attribute::` directive, following the class's own documentation.
- The relation between modules and module-level attributes is similar: modules should be documented in their own docstring, public module attributes should be documented in the module's docstring using the `.. data::` directive.

Javascript API documentation

Javascript API documentation uses JsDoc, a javascript documentation toolkit with a syntax similar to (and inspired by) JavaDoc's.

Due to limitations of JsDoc, the coding patterns in OpenERP Web and the Sphinx integration, there are a few peculiarities to be aware of when writing javascript API documentation:

- Namespaces and classes *must* be explicitly marked up even if they are not documented, or JsDoc will not understand what they are and will not generate documentation for their content.

  As a result, the bare minimum for a namespace is:

```
/** @namespace */
foo.bar.baz = {};
```

  while for a class it is:

```
/** @class */
foo.bar.baz.Qux = [...]
```

- Because the OpenERP Web project uses John Resig's Class implementation instead of direct prototypal inheritance [4], JsDoc fails to infer class scopes (and constructors or super classes, for that matter) and has to be told explicitly.

  See *Documenting a Class* for the complete rundown.

[3] Because Python is the default domain, the `py:` markup prefix is optional and should be left out.

[4] Resig's Class still uses prototypes under the hood, it doesn't reimplement its own object system although it does add several helpers such as the `_super()` instance method.

- Much like the JavaDoc, JsDoc does not include a full markup language. Instead, comments are simply marked up in HTML.

  This has a number of inconvenients:

  - Complex documentation comments become nigh-unreadable to read in text editors (as opposed to IDEs, which may handle rendering documentation comments on the fly)
  - Though cross-references are supported by JsDoc (via `@link` and `@see`), they only work within the JsDoc
  - More general impossibility to integrate correctly with Sphinx, and e.g. reference JavaScript objects from a tutorial, or have all the documentation live at the same place.

  As a result, JsDoc comments should be marked up using RST, not HTML. They may use Sphinx's cross-references as well.

Documenting a Class

The first task when documenting a class using JsDoc is to *mark* that class, so JsDoc knows it can be used to instantiate objects (and, more importantly as far as it's concerned, should be documented with methods and attributes and stuff).

This is generally done through the `@class` tag, but this tag has a significant limitation: it "believes" the constructor and the class are one and the same [5]. This will work for constructor-less classes, but because OpenERP Web uses Resig's class the constructor is not the class itself but its `init()` method.

Because this pattern is common in modern javascript code bases, JsDoc supports it: it is possible to mark an arbitrary instance method as the *class specification* by using the `@constructs` tag.

Warning: `@constructs` is a class specification in and of itself, it *completely replaces* the class documentation. Using both a class documentation (even without `@class` itself) and a constructor documentation is an *error* in JsDoc and will result in incorrect behavior and broken documentation.

The second issue is that Resig's class uses an object literal to specify instance methods, and because JsDoc does not know anything about Resig's class, it does not know about the role of the object literal.

As with constructors, though, JsDoc provides a pluggable way to tell it about methods: the `@lends` tag. It specifies that the object literal "lends" its properties to the class being built.

`@lends` must be specified right before the opening brace of the object literal (between the opening paren of the `#extend` call and the brace), and takes the full qualified name of the class being created as a parameter, followed by the character # or by `.prototype`. This latter part tells JsDoc these are instance methods, not class (static) methods..

Finally, specifying a class's superclass is done through the `@extends` tag, which takes a fully qualified class name as a parameter.

Here are a class without a constructor, and a class with one, so that everything is clear (these are straight from the OpenERP Web source, with the descriptions and irrelevant atttributes stripped):

```
/**
 * <Insert description here, not below>
 *
 * @class
 * @extends openerp.base.search.Field
 */
openerp.base.search.CharField = openerp.base.search.Field.extend(
    /** @lends openerp.base.search.CharField# */ {
```

[5] Which is the case in normal Javascript semantics. Likewise, the `.prototype` / # pattern we will see later on is due to JsDoc defaulting to the only behavior it can rely on: "normal" Javascript prototype-based type creation.

```
        // methods here
});
```

```
openerp.base.search.Widget = openerp.base.Controller.extend(
    /** @lends openerp.base.search.Widget# */{
    /**
     * <Insert description here, not below>
     *
     * @constructs
     * @extends openerp.base.Controller
     *
     * @param view the ancestor view of this widget
     */
    init: function (view) {
        // construction of the instance
    },
    // bunch of other methods
});
```

6.2 OpenERP Web over time

6.2.1 Release process

OpenSUSE packaging: http://blog.lowkster.com/2011/04/packaging-python-packages-in-opensuse.html

6.2.2 Roadmap

6.2.3 Release notes

CHAPTER 7

Main differences with the 6.0 client

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index

Symbols

-a
    gen_translations.sh command line option, 17

D

dataset (None attribute), 14
do_fold() (built-in function), 15
do_toggle() (built-in function), 15
do_unfold() (built-in function), 15

G

gen_translations.sh command line option
    -a, 17
get_selected_ids() (built-in function), 15

O

openerp.web._lt() (openerp.web method), 16
openerp.web._t() (openerp.web method), 16

S

sidebar_context() (built-in function), 15