OpenERP Web Developers Documentation

Release 7.0

OpenERP s.a.

August 29, 2016

Contents:

CHAPTER 1

Building an OpenERP Web module

There is no significant distinction between an OpenERP Web module and an OpenERP module, the web part is mostly additional data and code inside a regular OpenERP module. This allows providing more seamless features by integrating your module deeper into the web client.

1.1 A Basic Module

A very basic OpenERP module structure will be our starting point:

```
web_example
-- __init__.py
-- __openerp__.py
```

```
# __openerp__.py
{
    'name': "Web Example",
    'description': "Basic example of a (future) web module",
    'category': 'Hidden',
    'depends': ['base'],
}
```

This is a sufficient minimal declaration of a valid OpenERP module.

1.2 Web Declaration

There is no such thing as a "web module" declaration. An OpenERP module is automatically recognized as "web-enabled" if it contains a `static` directory at its root, so:

```
web_example
-- __init__.py
-- __openerp__.py
-- static
```

is the extent of it. You should also change the dependency to list `web`:

```
--- web_example/__openerp__.py
+++ web_example/__openerp__.py
@@ -1,7 +1,7 @@
 # __openerp__.py
 {
     'name': "Web Example",
     'description': "Basic example of a (future) web module",
     'category': 'Hidden',
-    'depends': ['base'],
+    'depends': ['web'],
 }
```

Note: This does not matter in normal operation so you may not realize it's wrong (the web module does the loading of everything else, so it can only be loaded), but when e.g. testing the loading process is slightly different than normal, and incorrect dependency may lead to broken code.

This makes the "web" discovery system consider the module as having a "web part", and check if it has web controllers to mount or javascript files to load. The content of the `static/` folder is also automatically made available to web browser at the URL `$module-name/static/$file-path`. This is sufficient to provide pictures (of cats, usually) through your module. However there are still a few more steps to running javascript code.

1.3 Getting Things Done

The first one is to add javascript code. It's customary to put it in `static/src/js`, to have room for e.g. other file types, or third-party libraries.

```
// static/src/js/first_module.js
console.log("Debug statement: file loaded");
```

The client won't load any file unless specified, thus the new file should be listed in the module's manifest file, under a new key `js` (a list of file names, or glob patterns):

```
--- web_example/__openerp__.py
+++ web_example/__openerp__.py
@@ -1,7 +1,8 @@
 # __openerp__.py
 {
     'name': "Web Example",
     'description': "Basic example of a (future) web module",
     'category': 'Hidden',
     'depends': ['web'],
+    'js': ['static/src/js/first_module.js'],
 }
```

At this point, if the module is installed and the client reloaded the message should appear in your browser's development console.

Note: Because the manifest file has been edited, you will have to restart the OpenERP server itself for it to be taken in account.

You may also want to open your browser's console *before* reloading, depending on the browser messages printed while the console is closed may not work or may not appear after opening it.

Note: If the message does not appear, try cleaning your browser's caches and ensure the file is correctly loaded from the server logs or the "resources" tab of your browser's developers tools.

At this point the code runs, but it runs only once when the module is initialized, and it can't get access to the various APIs of the web client (such as making RPC requests to the server). This is done by providing a javascript module:

```
--- web_example/static/src/js/first_module.js
+++ web_example/static/src/js/first_module.js
@@ -1,2 +1,4 @@
 // static/src/js/first_module.js
-console.log("Debug statement: file loaded");
+openerp.web_example = function (instance) {
+    console.log("Module loaded");
+};
```

If you reload the client, you'll see a message in the console exactly as previously. The differences, though invisible at this point, are:

- All javascript files specified in the manifest (only this one so far) have been fully loaded
- An instance of the web client and a namespace inside that instance (with the same name as the module) have been created and are available for use

The latter point is what the `instance` parameter to the function provides: an instance of the OpenERP Web client, with the contents of all the new module's dependencies loaded in and initialized. These are the entry points to the web client's APIs.

To demonstrate, let's build a simple client action: a stopwatch

First, the action declaration:

```
--- web_example/__openerp__.py
+++ web_example/__openerp__.py
@@ -1,8 +1,9 @@
 # __openerp__.py
 {
     'name': "Web Example",
     'description': "Basic example of a (future) web module",
     'category': 'Hidden',
     'depends': ['web'],
+    'data': ['web_example.xml'],
     'js': ['static/src/js/first_module.js'],
 }
```

```
<!-- web_example/web_example.xml -->
<openerp>
    <data>
        <record model="ir.actions.client" id="action_client_example">
            <field name="name">Example Client Action</field>
            <field name="tag">example.action</field>
        </record>
        <menuitem action="action_client_example"
                  id="menu_client_example"/>
    </data>
</openerp>
```

then set up the client action hook to register a function (for now):

```
--- web_example/static/src/js/first_module.js
+++ web_example/static/src/js/first_module.js
@@ -1,4 +1,7 @@
 // static/src/js/first_module.js
 openerp.web_example = function (instance) {
-    console.log("Module loaded");
+    instance.web.client_actions.add('example.action', 'instance.web_example.Action');
+    instance.web_example.Action = function (parent, action) {
+        console.log("Executed the action", action);
+    };
 };
```

Updating the module (in order to load the XML description) and re-starting the server should display a new menu *Example Client Action* at the top-level. Opening said menu will make the message appear, as usual, in the browser's console.

1.4 Paint it black

The next step is to take control of the page itself, rather than just print little messages in the console. This we can do by replacing our client action function by a Widget. Our widget will simply use its `start()` to add some content to its DOM:

```
--- web_example/static/src/js/first_module.js
+++ web_example/static/src/js/first_module.js
@@ -1,7 +1,11 @@
 // static/src/js/first_module.js
 openerp.web_example = function (instance) {
-    instance.web.client_actions.add('example.action', 'instance.web_example.Action');
-    instance.web_example.Action = function (parent, action) {
-        console.log("Executed the action", action);
-    };
+    instance.web.client_actions.add('example.action', 'instance.web_example.Action');
+    instance.web_example.Action = instance.web.Widget.extend({
+        className: 'oe_web_example',
+        start: function () {
+            this.$el.text("Hello, world!");
+            return this._super();
+        }
+    });
 };
```

after reloading the client (to update the javascript file), instead of printing to the console the menu item clears the whole screen and displays the specified message in the page.

Since we've added a class on the widget's *DOM root* we can now see how to add a stylesheet to a module: first create the stylesheet file:

```
.openerp .oe_web_example {
    color: white;
    background-color: black;
    height: 100%;
    font-size: 400%;
}
```

then add a reference to the stylesheet in the module's manifest (which will require restarting the OpenERP Server to see the changes, as usual):

```
--- web_example/__openerp__.py
+++ web_example/__openerp__.py
@@ -1,9 +1,10 @@
 # __openerp__.py
 {
     'name': "Web Example",
     'description': "Basic example of a (future) web module",
     'category': 'Hidden',
     'depends': ['web'],
     'data': ['web_example.xml'],
     'js': ['static/src/js/first_module.js'],
+    'css': ['static/src/css/web_example.css'],
 }
```

the text displayed by the menu item should now be huge, and white-on-black (instead of small and black-on-white). From there on, the world's your canvas.

Note: Prefixing CSS rules with both `.openerp` (to ensure the rule will apply only within the confines of the OpenERP Web client) and a class at the root of your own hierarchy of widgets is strongly recommended to avoid "leaking" styles in case the code is running embedded in an other web page, and does not have the whole screen to itself.

So far we haven't built much (any, really) DOM content. It could all be done in `start()` but that gets unwieldy and hard to maintain fast. It is also very difficult to extend by third parties (trying to add or change things in your widgets) unless broken up into multiple methods which each perform a little bit of the rendering.

The first way to handle this method is to delegate the content to plenty of sub-widgets, which can be individually overridden. An other method [1] is to use a template to render a widget's DOM.

OpenERP Web's template language is QWeb. Although any templating engine can be used (e.g. mustache or _.template) QWeb has important features which other template engines may not provide, and has special integration to OpenERP Web widgets.

Adding a template file is similar to adding a style sheet:

```
<templates>
<div t-name="web_example.action" class="oe_web_example oe_web_example_stopped">
    <h4 class="oe_web_example_timer">00:00:00</h4>
    <p class="oe_web_example_start">
        <button type="button">Start</button>
    </p>
    <p class="oe_web_example_stop">
        <button type="button">Stop</button>
    </p>
</div>
</templates>
```

```
--- web_example/__openerp__.py
+++ web_example/__openerp__.py
@@ -1,10 +1,11 @@
 # __openerp__.py
 {
     'name': "Web Example",
     'description': "Basic example of a (future) web module",
     'category': 'Hidden',
     'depends': ['web'],
     'data': ['web_example.xml'],
     'js': ['static/src/js/first_module.js'],
     'css': ['static/src/css/web_example.css'],
+    'qweb': ['static/src/xml/web_example.xml'],
 }
```

[1] they are not alternative solutions: they work very well together. Templates are used to build "just DOM", sub-widgets are used to build DOM subsections *and* delegate part of the behavior (e.g. events handling).

The template can then easily be hooked in the widget:

```
--- web_example/static/src/js/first_module.js
+++ web_example/static/src/js/first_module.js
@@ -1,11 +1,7 @@
 // static/src/js/first_module.js
 openerp.web_example = function (instance) {
     instance.web.client_actions.add('example.action', 'instance.web_example.Action');
     instance.web_example.Action = instance.web.Widget.extend({
+        template: 'web_example.action'
-        className: 'oe_web_example',
-        start: function () {
-            this.$el.text("Hello, world!");
-            return this._super();
-        }
     });
 };
```

And finally the CSS can be altered to style the new (and more complex) template-generated DOM, rather than the code-generated one:

```
--- web_example/static/src/css/web_example.css
+++ web_example/static/src/css/web_example.css
@@ -1,6 +1,13 @@
 .openerp .oe_web_example {
     color: white;
     background-color: black;
     height: 100%;
-    font-size: 400%;
 }
+.openerp .oe_web_example h4 {
+    margin: 0;
+    font-size: 200%;
+}
+.openerp .oe_web_example.oe_web_example_started .oe_web_example_start button,
+.openerp .oe_web_example.oe_web_example_stopped .oe_web_example_stop button {
+    display: none
+}
```

Note: The last section of the CSS change is an example of "state classes": a CSS class (or set of classes) on the root of the widget, which is toggled when the state of the widget changes and can perform drastic alterations in rendering (usually showing/hiding various elements).

This pattern is both fairly simple (to read and understand) and efficient (because most of the hard work is pushed to the browser's CSS engine, which is usually highly optimized, and done in a single repaint after toggling the class).

The last step (until the next one) is to add some behavior and make our stopwatch watch. First hook some events on the buttons to toggle the widget's state:

```
--- web_example/static/src/js/first_module.js
+++ web_example/static/src/js/first_module.js
@@ -1,7 +1,19 @@
 // static/src/js/first_module.js
 openerp.web_example = function (instance) {
     instance.web.client_actions.add('example.action', 'instance.web_example.Action');
     instance.web_example.Action = instance.web.Widget.extend({
-        template: 'web_example.action'
+        template: 'web_example.action',
+        events: {
+            'click .oe_web_example_start button': 'watch_start',
+            'click .oe_web_example_stop button': 'watch_stop'
+        },
+        watch_start: function () {
+            this.$el.addClass('oe_web_example_started')
+                    .removeClass('oe_web_example_stopped');
+        },
+        watch_stop: function () {
+            this.$el.removeClass('oe_web_example_started')
+                    .addClass('oe_web_example_stopped');
+        },
     });
 };
```

This demonstrates the use of the "events hash" and event delegation to declaratively handle events on the widget's DOM. And already changes the button displayed in the UI. Then comes some actual logic:

```
--- web_example/static/src/js/first_module.js
+++ web_example/static/src/js/first_module.js
@@ -1,19 +1,52 @@
 // static/src/js/first_module.js
 openerp.web_example = function (instance) {
     instance.web.client_actions.add('example.action', 'instance.web_example.Action');
     instance.web_example.Action = instance.web.Widget.extend({
         template: 'web_example.action',
         events: {
             'click .oe_web_example_start button': 'watch_start',
             'click .oe_web_example_stop button': 'watch_stop'
         },
+        init: function () {
+            this._super.apply(this, arguments);
+            this._start = null;
+            this._watch = null;
+        },
+        update_counter: function () {
+            var h, m, s;
+            // Subtracting javascript dates returns the difference in milliseconds
+            var diff = new Date() - this._start;
+            s = diff / 1000;
+            m = Math.floor(s / 60);
+            s -= 60*m;
+            h = Math.floor(m / 60);
+            m -= 60*h;
+            this.$('.oe_web_example_timer').text(
+                _.str.sprintf("%02d:%02d:%02d", h, m, s));
+        },
         watch_start: function () {
             this.$el.addClass('oe_web_example_started')
                     .removeClass('oe_web_example_stopped');
+            this._start = new Date();
+            // Update the UI to the current time
+            this.update_counter();
+            // Update the counter at 30 FPS (33ms/frame)
+            this._watch = setInterval(
+                this.proxy('update_counter'),
+                33);
         },
         watch_stop: function () {
+            clearInterval(this._watch);
+            this.update_counter();
+            this._start = this._watch = null;
             this.$el.removeClass('oe_web_example_started')
                     .addClass('oe_web_example_stopped');
         },
+        destroy: function () {
+            if (this._watch) {
+                clearInterval(this._watch);
+            }
+            this._super();
+        }
     });
 };
```

- An initializer (the `init` method) is introduced to set-up a few internal variables: `_start` will hold the start of the timer (as a javascript Date object), and `_watch` will hold a ticker to update the interface regularly and display the "current time".
- `update_counter` is in charge of taking the time difference between "now" and `_start`, formatting as `HH:MM:SS` and displaying the result on screen.
- `watch_start` is augmented to initialize `_start` with its value and set-up the update of the counter display every 33ms.
- `watch_stop` disables the updater, does a final update of the counter display and resets everything.
- Finally, because javascript Interval and Timeout objects execute "outside" the widget, they will keep going even after the widget has been destroyed (especially an issue with intervals as they repeat indefinitely). So `_watch` *must* be cleared when the widget is destroyed (then the `_super` must be called as well in order to perform the "normal" widget cleanup).

Starting and stopping the watch now works, and correctly tracks time since having started the watch, neatly formatted.

CHAPTER 2

Widget

class `openerp.web.`**`Widget`**`()`

This is the base class for all visual components. It corresponds to an MVC view. It provides a number of services to handle a section of a page:

- 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()

- Backbone-compatible shortcuts

2.1 DOM Root

A *`Widget()`* is responsible for a section of the page materialized by the DOM root of the widget. The DOM root is available via the `el` and `$el` attributes, which are respectively the raw DOM Element and the jQuery wrapper around the DOM element.

There are two main ways to define and generate this DOM root:

`openerp.web.Widget.`**`template`**

Should be set to the name of a QWeb template (a `String()`). If set, the template will be rendered after the widget has been initialized but before it has been started. The root element generated by the template will be set as the DOM root of the widget.

`openerp.web.Widget.`**`tagName`**

Used if the widget has no template defined. Defaults to `div`, will be used as the tag name to create the DOM element to set as the widget's DOM root. It is possible to further customize this generated DOM root with the following attributes:

`openerp.web.Widget.`**`id`**

Used to generate an `id` attribute on the generated DOM root.

`openerp.web.Widget.`**`className`**
Used to generate a `class` attribute on the generated DOM root.

`openerp.web.Widget.`**`attributes`**
Mapping (object literal) of attribute names to attribute values. Each of these k:v pairs will be set as a DOM attribute on the generated DOM root.

None of these is used in case a template is specified on the widget.

The DOM root can also be defined programmatically by overridding

`openerp.web.Widget.`**`renderElement`**`()`
Renders the widget's DOM root and sets it. The default implementation will render a set template or generate an element as described above, and will call *`setElement()`* on the result.

Any override to *`renderElement()`* which does not call its `_super` **must** call *`setElement()`* with whatever it generated or the widget's behavior is undefined.r

Note: The default *`renderElement()`* can be called repeatedly, it will *replace* the previous DOM root (using `replaceWith`). However, this requires that the widget correctly sets and unsets its events (and children widgets). Generally, *`renderElement()`* should not be called repeatedly unless the widget advertizes this feature.

2.1.1 Accessing DOM content

Because a widget is only responsible for the content below its DOM root, there is a shortcut for selecting sub-sections of a widget's DOM:

`openerp.web.Widget.`**`$`**`(`*selector*`)`
Applies the CSS selector specified as parameter to the widget's DOM root.

```
this.$(selector);
```

is functionally identical to:

```
this.$el.find(selector);
```

Arguments

- **`selector`** (*`String`*) – CSS selector

Returns jQuery object

Note: this helper method is compatible with `Backbone.View.$`

2.1.2 Resetting the DOM root

`openerp.web.Widget.`**`setElement`**`(`*element*`)`
Re-sets the widget's DOM root to the provided element, also handles re-setting the various aliases of the DOM root as well as unsetting and re-setting delegated events.

Arguments

- **`element`** (*`Element`*) – a DOM element or jQuery object to set as the widget's DOM root

Note: should be mostly compatible with Backbone's setElement

2.2 DOM events handling

A widget will generally need to respond to user action within its section of the page. This entails binding events to DOM elements.

To this end, *Widget()* provides an shortcut:

openerp.web.Widget.**events**

Events are a mapping of `event selector` (an event name and a CSS selector separated by a space) to a callback. The callback can be either a method name in the widget or a function. In either case, the `this` will be set to the widget:

```
events: {
    'click p.oe_some_class a': 'some_method',
    'change input': function (e) {
        e.stopPropagation();
    }
},
```

The selector is used for jQuery's event delegation, the callback will only be triggered for descendants of the DOM root matching the selector [0]. If the selector is left out (only an event name is specified), the event will be set directly on the widget's DOM root.

openerp.web.Widget.**delegateEvents**()

This method is in charge of binding *events* to the DOM. It is automatically called after setting the widget's DOM root.

It can be overridden to set up more complex events than the *events* map allows, but the parent should always be called (or *events* won't be handled correctly).

openerp.web.Widget.**undelegateEvents**()

This method is in charge of unbinding *events* from the DOM root when the widget is destroyed or the DOM root is reset, in order to avoid leaving "phantom" events.

It should be overridden to un-set any event set in an override of *delegateEvents()*.

Note: this behavior should be compatible with Backbone's delegateEvents, apart from not accepting any argument.

2.3 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) {
```

[0] not all DOM events are compatible with events delegation

```
        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.destroy();
```

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

CHAPTER 3

Asynchronous Operations

As a language (and runtime), javascript is fundamentally single-threaded. This means any blocking request or computation will blocks the whole page (and, in older browsers, the software itself even preventing users from switching to an other tab): a javascript environment can be seen as an event-based runloop where application developers have no control over the runloop itself.

As a result, performing long-running synchronous network requests or other types of complex and expensive accesses is frowned upon and asynchronous APIs are used instead.

Asynchronous code rarely comes naturally, especially for developers used to synchronous server-side code (in Python, Java or C#) where the code will just block until the deed is gone. This is increased further when asynchronous programming is not a first-class concept and is instead implemented on top of callbacks-based programming, which is the case in javascript.

The goal of this guide is to provide some tools to deal with asynchronous systems, and warn against systematic issues or dangers.

3.1 Deferreds

Deferreds are a form of promises. OpenERP Web currently uses jQuery's deferred.

The core idea of deferreds is that potentially asynchronous methods will return a `Deferred()` object instead of an arbitrary value or (most commonly) nothing.

This object can then be used to track the end of the asynchronous operation by adding callbacks onto it, either success callbacks or error callbacks.

A great advantage of deferreds over simply passing callback functions directly to asynchronous methods is the ability to *compose them*.

3.1.1 Using deferreds

Deferreds's most important method is `Deferred.then()`. It is used to attach new callbacks to the deferred object.

- the first parameter attaches a success callback, called when the deferred object is successfully resolved and provided with the resolved value(s) for the asynchronous operation.
- the second parameter attaches a failure callback, called when the deferred object is rejected and provided with rejection values (often some sort of error message).

Callbacks attached to deferreds are never "lost": if a callback is attached to an already resolved or rejected deferred, the callback will be called (or ignored) immediately. A deferred is also only ever resolved or rejected once, and is either resolved or rejected: a given deferred can not call a single success callback twice, or call both a success and a failure callbacks.

`then()` should be the method you'll use most often when interacting with deferred objects (and thus asynchronous APIs).

3.1.2 Building deferreds

After using asynchronous APIs may come the time to build them: for mocks, to compose deferreds from multiple source in a complex manner, in order to let the current operations repaint the screen or give other events the time to unfold, ...

This is easy using jQuery's deferred objects.

Note: this section is an implementation detail of jQuery Deferred objects, the creation of promises is not part of any standard (even tentative) that I know of. If you are using deferred objects which are not jQuery's, their API may (and often will) be completely different.

Deferreds are created by invoking their constructor [1] without any argument. This creates a `Deferred()` instance object with the following methods:

`Deferred.resolve()`

As its name indicates, this method moves the deferred to the "Resolved" state. It can be provided as many arguments as necessary, these arguments will be provided to any pending success callback.

`Deferred.reject()`

Similar to `resolve()`, but moves the deferred to the "Rejected" state and calls pending failure handlers.

`Deferred.promise()`

Creates a readonly view of the deferred object. It is generally a good idea to return a promise view of the deferred to prevent callers from resolving or rejecting the deferred in your stead.

`reject()` and `resolve()` are used to inform callers that the asynchronous operation has failed (or succeeded). These methods should simply be called when the asynchronous operation has ended, to notify anybody interested in its result(s).

3.1.3 Composing deferreds

What we've seen so far is pretty nice, but mostly doable by passing functions to other functions (well adding functions post-facto would probably be a chore... still, doable).

Deferreds truly shine when code needs to compose asynchronous operations in some way or other, as they can be used as a basis for such composition.

There are two main forms of compositions over deferred: multiplexing and piping/cascading.

Deferred multiplexing

The most common reason for multiplexing deferred is simply performing 2+ asynchronous operations and wanting to wait until all of them are done before moving on (and executing more stuff).

The jQuery multiplexing function for promises is `when()`.

[1] or simply calling `Deferred()` as a function, the result is the same

Note: the multiplexing behavior of jQuery's `when()` is an (incompatible, mostly) extension of the behavior defined in CommonJS Promises/B.

This function can take any number of promises [2] and will return a promise.

This returned promise will be resolved when *all* multiplexed promises are resolved, and will be rejected as soon as one of the multiplexed promises is rejected (it behaves like Python's `all()`, but with promise objects instead of boolean-ish).

The resolved values of the various promises multiplexed via `when()` are mapped to the arguments of `when()`'s success callback, if they are needed. The resolved values of a promise are at the same index in the callback's arguments as the promise in the `when()` call so you will have:

```
$.when(p0, p1, p2, p3).then(
        function (results0, results1, results2, results3) {
    // code
});
```

Warning: in a normal mapping, each parameter to the callback would be an array: each promise is conceptually resolved with an array of 0..n values and these values are passed to `when()`'s callback. But jQuery treats deferreds resolving a single value specially, and "unwraps" that value.
For instance, in the code block above if the index of each promise is the number of values it resolves (0 to 3), `results0` is an empty array, `results2` is an array of 2 elements (a pair) but `results1` is the actual value resolved by `p1`, not an array.

Deferred chaining

A second useful composition is starting an asynchronous operation as the result of an other asynchronous operation, and wanting the result of both: with the tools described so far, handling e.g. OpenERP's search/read sequence with this would require something along the lines of:

```
var result = $.Deferred();
Model.search(condition).then(function (ids) {
    Model.read(ids, fields).then(function (records) {
        result.resolve(records);
    });
});
return result.promise();
```

While it doesn't look too bad for trivial code, this quickly gets unwieldy.

But `then()` also allows handling this kind of chains: it returns a new promise object, not the one it was called with, and the return values of the callbacks is actually important to it: whichever callback is called,

- If the callback is not set (not provided or left to null), the resolution or rejection value(s) is simply forwarded to `then()`'s promise (it's essentially a noop)
- If the callback is set and does not return an observable object (a deferred or a promise), the value it returns (`undefined` if it does not return anything) will replace the value it was given, e.g.

[2] or not-promises, the CommonJS Promises/B role of `when()` is to be able to treat values and promises uniformly: `when()` will pass promises through directly, but non-promise values and objects will be transformed into a resolved promise (resolving themselves with the value itself).
jQuery's `when()` keeps this behavior making deferreds easy to build from "static" values, or allowing defensive code where expected promises are wrapped in `when()` just in case.

```
promise.then(function () {
    console.log('called');
});
```

will resolve with the sole value `undefined`.

- If the callback is set and returns an observable object, that object will be the actual resolution (and result) of the pipe. This means a resolved promise from the failure callback will resolve the pipe, and a failure promise from the success callback will reject the pipe.

  This provides an easy way to chain operation successes, and the previous piece of code can now be rewritten:

```
return Model.search(condition).then(function (ids) {
    return Model.read(ids, fields);
});
```

  the result of the whole expression will encode failure if either `search` or `read` fails (with the right rejection values), and will be resolved with `read`'s resolution values if the chain executes correctly.

`then()` is also useful to adapt third-party promise-based APIs, in order to filter their resolution value counts for instance (to take advantage of *`when()`* 's special treatment of single-value promises).

3.1.4 jQuery.Deferred API

`when`(*deferreds...*)

Arguments

- **`deferreds`** – deferred objects to multiplex

Returns a multiplexed deferred

Return type *`Deferred()`*

class **`Deferred`**()

`Deferred.`**`then`**(*doneCallback*[, *failCallback*])

Attaches new callbacks to the resolution or rejection of the deferred object. Callbacks are executed in the order they are attached to the deferred.

To provide only a failure callback, pass `null` as the `doneCallback`, to provide only a success callback the second argument can just be ignored (and not passed at all).

Returns a new deferred which resolves to the result of the corresponding callback, if a callback returns a deferred itself that new deferred will be used as the resolution of the chain.

Arguments

- **`doneCallback`** (*`Function`*) – function called when the deferred is resolved
- **`failCallback`** (*`Function`*) – function called when the deferred is rejected

Returns the deferred object on which it was called

Return type *`Deferred()`*

`Deferred.`**`done`**(*doneCallback*)

Attaches a new success callback to the deferred, shortcut for `deferred.then(doneCallback)`.

This is a jQuery extension to CommonJS Promises/A providing little value over calling *`then()`* directly, it should be avoided.

Arguments

- **`doneCallback`** (*`Function`*) – function called when the deferred is resolved

Returns the deferred object on which it was called

Return type *`Deferred()`*

`Deferred.`**`fail`**(*failCallback*)

Attaches a new failure callback to the deferred, shortcut for `deferred.then(null, failCallback)`.

A second jQuery extension to Promises/A. Although it provides more value than *`done()`*, it still is not much and should be avoided as well.

Arguments

- **`failCallback`** (*`Function`*) – function called when the deferred is rejected

Returns the deferred object on which it was called

Return type *`Deferred()`*

`Deferred.`**`promise`**()

Returns a read-only view of the deferred object, with all mutators (resolve and reject) methods removed.

`Deferred.`**`resolve`**(*value...*)

Called to resolve a deferred, any value provided will be passed onto the success handlers of the deferred object.

Resolving a deferred which has already been resolved or rejected has no effect.

`Deferred.`**`reject`**(*value...*)

Called to reject (fail) a deferred, any value provided will be passed onto the failure handler of the deferred object.

Rejecting a deferred which has already been resolved or rejected has no effect.

CHAPTER 4

RPC Calls

Building static displays is all nice and good and allows for neat effects (and sometimes you're given data to display from third parties so you don't have to make any effort), but a point generally comes where you'll want to talk to the world and make some network requests.

OpenERP Web provides two primary APIs to handle this, a low-level JSON-RPC based API communicating with the Python section of OpenERP Web (and of your addon, if you have a Python part) and a high-level API above that allowing your code to talk directly to the OpenERP server, using familiar-looking calls.

All networking APIs are asynchronous. As a result, all of them will return `Deferred()` objects (whether they resolve those with values or not). Understanding how those work before before moving on is probably necessary.

4.1 High-level API: calling into OpenERP models

Access to OpenERP object methods (made available through XML-RPC from the server) is done via the `openerp.web.Model()` class. This class maps onto the OpenERP server objects via two primary methods, `call()` and `query()`.

`call()` is a direct mapping to the corresponding method of the OpenERP server object. Its usage is similar to that of the OpenERP Model API, with three differences:

- The interface is asynchronous, so instead of returning results directly RPC method calls will return `Deferred()` instances, which will themselves resolve to the result of the matching RPC call.
- Because ECMAScript 3/Javascript 1.5 doesnt feature any equivalent to `__getattr__` or `method_missing`, there needs to be an explicit method to dispatch RPC methods.
- No notion of pooler, the model proxy is instantiated where needed, not fetched from an other (somewhat global) object

```
var Users = new Model('res.users');

Users.call('change_password', ['oldpassword', 'newpassword'],
                  {context: some_context}).then(function (result) {
    // do something with change_password result
});
```

`query()` is a shortcut for a builder-style interface to searches (`search` + `read` in OpenERP RPC terms). It returns a `Query()` object which is immutable but allows building new `Query()` instances from the first one, adding new properties or modifiying the parent object's:

```
Users.query(['name', 'login', 'user_email', 'signature'])
     .filter([['active', '=', true], ['company_id', '=', main_company]])
     .limit(15)
     .all().then(function (users) {
    // do work with users records
});
```

The query is only actually performed when calling one of the query serialization methods, `all()` and `first()`. These methods will perform a new RPC call every time they are called.

For that reason, it's actually possible to keep "intermediate" queries around and use them differently/add new specifications on them.

class `openerp.web.`**`Model`**(*name*)

`openerp.web.Model.`**`name`**
name of the OpenERP model this object is bound to

`openerp.web.Model.`**`call`**(*method[, args][, kwargs]*)
Calls the `method` method of the current model, with the provided positional and keyword arguments.

Arguments

- **`method`** (*`String`*) – method to call over rpc on the `name`
- **`args`** (*`Array<>`*) – positional arguments to pass to the method, optional
- **`kwargs`** (*`Object<>`*) – keyword arguments to pass to the method, optional

Return type Deferred<>

`openerp.web.Model.`**`query`**(*fields*)

Arguments

- **`fields`** (*`Array<String>`*) – list of fields to fetch during the search

Returns a `Query()` object representing the search to perform

class `openerp.web.`**`Query`**(*fields*)
The first set of methods is the "fetching" methods. They perform RPC queries using the internal data of the object they're called on.

`openerp.web.Query.`**`all`**()
Fetches the result of the current `Query()` object's search.

Return type Deferred<Array<>>

`openerp.web.Query.`**`first`**()
Fetches the **first** result of the current `Query()`, or `null` if the current `Query()` does have any result.

Return type Deferred<Object | null>

`openerp.web.Query.`**`count`**()
Fetches the number of records the current `Query()` would retrieve.

Return type Deferred<Number>

`openerp.web.Query.`**`group_by`**(*grouping...*)
Fetches the groups for the query, using the first specified grouping parameter

Arguments

- **`grouping`** (*`Array<String>`*) – Lists the levels of grouping asked of the server. Grouping can actually be an array or varargs.

Return type Deferred<Array<openerp.web.QueryGroup>> | null

The second set of methods is the "mutator" methods, they create a **new** `Query()` object with the relevant (internal) attribute either augmented or replaced.

`openerp.web.Query.`**`context`**`(`*ctx*`)`
: Adds the provided `ctx` to the query, on top of any existing context

`openerp.web.Query.`**`filter`**`(`*domain*`)`
: Adds the provided domain to the query, this domain is `AND`-ed to the existing query domain.

`opeenrp.web.Query.`**`offset`**`(`*offset*`)`
: Sets the provided offset on the query. The new offset *replaces* the old one.

`openerp.web.Query.`**`limit`**`(`*limit*`)`
: Sets the provided limit on the query. The new limit *replaces* the old one.

`openerp.web.Query.`**`order_by`**`(`*fields…*`)`
: Overrides the model's natural order with the provided field specifications. Behaves much like Django's QuerySet.order_by:

  - Takes 1..n field names, in order of most to least importance (the first field is the first sorting key). Fields are provided as strings.
  - A field specifies an ascending order, unless it is prefixed with the minus sign "-" in which case the field is used in the descending order

  Divergences from Django's sorting include a lack of random sort (`?` field) and the inability to "drill down" into relations for sorting.

4.1.1 Aggregation (grouping)

OpenERP has powerful grouping capacities, but they are kind-of strange in that they're recursive, and level n+1 relies on data provided directly by the grouping at level n. As a result, while `read_group` works it's not a very intuitive API.

OpenERP Web 7.0 eschews direct calls to `read_group` in favor of calling a method of `Query()`, much in the way it is one in SQLAlchemy [1]:

```
some_query.group_by(['field1', 'field2']).then(function (groups) {
    // do things with the fetched groups
});
```

This method is asynchronous when provided with 1..n fields (to group on) as argument, but it can also be called without any field (empty fields collection or nothing at all). In this case, instead of returning a Deferred object it will return `null`.

When grouping criterion come from a third-party and may or may not list fields (e.g. could be an empty list), this provides two ways to test the presence of actual subgroups (versus the need to perform a regular query for records):

- A check on `group_by`'s result and two completely separate code paths

```
    var groups;
    if (groups = some_query.group_by(gby)) {
        groups.then(function (gs) {
            // groups
        });
    }
    // no groups
```

[1] with a small twist: SQLAlchemy's `orm.query.Query.group_by` is not terminal, it returns a query which can still be altered.

- Or a more coherent code path using *when()*'s ability to coerce values into deferreds:

```
    $.when(some_query.group_by(gby)).then(function (groups) {
        if (!groups) {
            // No grouping
        } else {
            // grouping, even if there are no groups (groups
            // itself could be an empty array)
        }
    });
```

The result of a (successful) *group_by()* is an array of `QueryGroup()`.

4.2 Low-level API: RPC calls to Python side

While the previous section is great for calling core OpenERP code (models code), it does not work if you want to call the Python side of OpenERP Web.

For this, a lower-level API exists on on `Connection()` objects (usually available through `openerp.connection`): the `rpc` method.

This method simply takes an absolute path (which is the combination of the Python controller's `_cp_path` attribute and the name of the method you want to call) and a mapping of attributes to values (applied as keyword arguments on the Python method [2]). This function fetches the return value of the Python methods, converted to JSON.

For instance, to call the `resequence` of the `DataSet` controller:

```
openerp.connection.rpc('/web/dataset/resequence', {
    model: some_model,
    ids: array_of_ids,
    offset: 42
}).then(function (result) {
    // resequenced on server
});
```

[2] except for `context`, which is extracted and stored in the request object itself.

CHAPTER 5

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 peculiarities:

- 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 integration).

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 mechanism. Like 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>
```

5.1 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 instance.

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 display).

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 views get their own `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 exceptions go through 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 and some attributes in templates. Defaults to `null`.

5.2 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 with bits of logic 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" element solely for 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 directive.
- 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 valid javascript 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 escaped, text 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 `&&`), `or` (maps to `||`), `gt` (maps to `>`), `gte` (maps to `>=`), `lt` (maps to `<`) and `lte` (maps to `<=`).

5.2.1 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 name will overwrite 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 either by `foo` or by a 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 simplicity.

```
<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).

5.2.2 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 computed ones.

5.2.3 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 instead.

`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 usage is similar to call-template with with-param in XSLT.

5.2.4 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 template using the old 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 *t-name*), these 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 node*, passing `BODY` 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 either avoid it or replace it with a sequence of `t-jquery:t-operation:`.

5.2.5 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`.

CHAPTER 6

Client actions

Client actions are the client-side version 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).

6.1 General Structure

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

- Mapping the action's `tag` to an object
- Providing said object. Two different types of objects can be mapped to a client action:
  - An OpenERP Web widget, which must inherit from *`openerp.web.Widget()`*
  - A regular javascript function

The major difference is in the lifecycle of these:

- if the client action maps to a function, the function will be called when executing the action. The function can have no further interaction with the Web Client itself, although it can return an action which will be executed after it.

  The function takes 2 parameters: the ActionManager calling it and the descriptor for the current action (the `ir.actions.client` dictionary).

- if, on the other hand, the client action maps to a *`Widget()`*, that *`Widget()`* will be instantiated and added to the web client's canvas, with the usual *`Widget()`* lifecycle (essentially, it will either take over the content area of the client or it will be integrated within a dialog).

For example, to create a client action displaying a `res.widget` object:

```
// Registers the object 'openerp.web_dashboard.Widget' to the client
// action tag 'board.home.widgets'
instance.web.client_actions.add(
    'board.home.widgets', 'instance.web_dashboard.Widget');
instance.web_dashboard.Widget = instance.web.Widget.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 `$el` property and the object is started.

The second parameter to the constructor is the descriptor for the action itself, which contains any parameter provided:

```
init: function (parent, action) {
    // 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 = action.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.

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

The client action can then behave exactly as it wishes to within its root (`this.$el`). 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.$el.html(QWeb.render('HomeWidget.content', {
        widget: widget,
        url: url
    }));
}
```

CHAPTER 7

Guidelines and Recommendations

7.1 Web Module Recommendations

7.1.1 Identifiers (`id` attribute) should be avoided

In generic applications and modules, `@id` limits the reusabily of components and tends to make code more brittle.

Just about all the time, they can be replaced with nothing, with classes or with keeping a reference to a DOM node or a jQuery element around.

Note: If it is *absolutely necessary* to have an `@id` (because a third-party library requires one and can't take a DOM element), it should be generated with _.uniqueId or some other similar method.

7.1.2 Avoid predictable/common CSS class names

Class names such as "content" or "navigation" might match the desired meaning/semantics, but it is likely an other developer will have the same need, creating a naming conflict and unintended behavior. Generic class names should be prefixed with e.g. the name of the component they belong to (creating "informal" namespaces, much as in C or Objective-C)

7.1.3 Global selectors should be avoided

Because a component may be used several times in a single page (an example in OpenERP is dashboards), queries should be restricted to a given component's scope. Unfiltered selections such as `$(selector)` or `document.querySelectorAll(selector)` will generally lead to unintended or incorrect behavior.

OpenERP Web's *`Widget()`* has an attribute providing its DOM root `Widget.$el`, and a shortcut to select nodes directly *`Widget.$`*.

More generally, never assume your components own or controls anything beyond its own personal DOM.

7.1.4 Understand deferreds

Deferreds, promises, futures, …

Known under many names, these objects are essential to and (in OpenERP Web) widely used for making asynchronous javascript operations palatable and understandable.

7.2 OpenERP Web guidelines

- HTML templating/rendering should use QWeb unless absolutely trivial.
- All interactive components (components displaying information to the screen or intercepting DOM events) must inherit from `Widget` and correctly implement and use its API and lifecycle.
- All css classes must be prefixed with *oe_* .
- Asynchronous functions (functions which call *session.rpc* directly or indirectly at the very least) *must* return deferreds, so that callers of overriders can correctly synchronize with them.

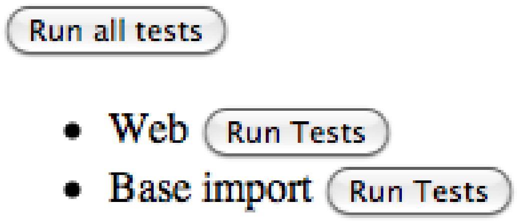
CHAPTER 8

Testing in OpenERP Web

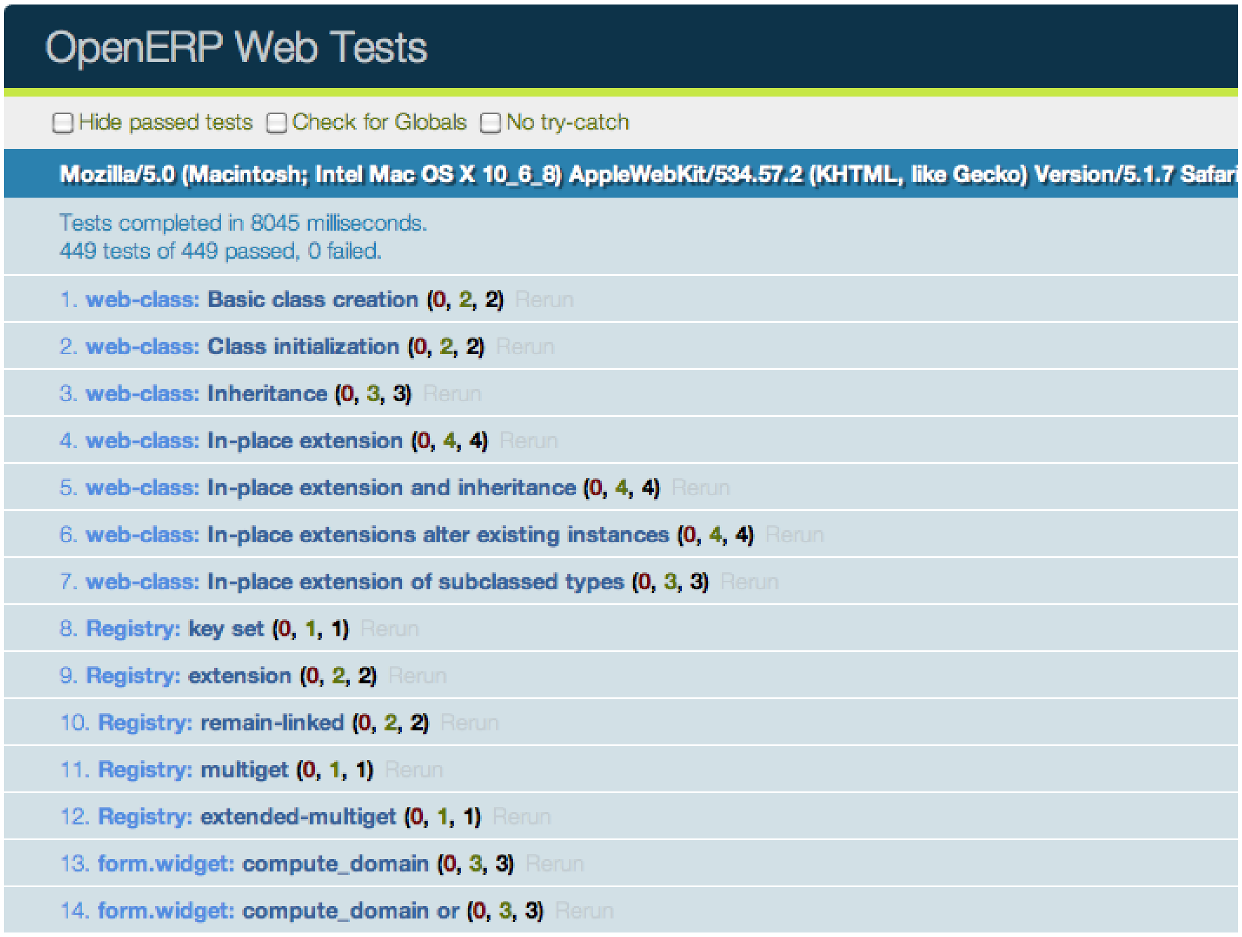
8.1 Javascript Unit Testing

OpenERP Web 7.0 includes means to unit-test both the core code of OpenERP Web and your own javascript modules. On the javascript side, unit-testing is based on QUnit with a number of helpers and extensions for better integration with OpenERP.

To see what the runner looks like, find (or start) an OpenERP server with the web client enabled, and navigate to `/web/tests` e.g. on OpenERP's CI. This will show the runner selector, which lists all modules with javascript unit tests, and allows starting any of them (or all javascript tests in all modules at once).



Clicking any runner button will launch the corresponding tests in the bundled QUnit runner:



OpenERP Web Tests

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/534.57.2 (KHTML, like Gecko) Version/5.1.7 Safari

Tests completed in 8045 milliseconds.
449 tests of 449 passed, 0 failed.

1. web-class: Basic class creation (0, 2, 2) Rerun
2. web-class: Class initialization (0, 2, 2) Rerun
3. web-class: Inheritance (0, 3, 3) Rerun
4. web-class: In-place extension (0, 4, 4) Rerun
5. web-class: In-place extension and inheritance (0, 4, 4) Rerun
6. web-class: In-place extensions alter existing instances (0, 4, 4) Rerun
7. web-class: In-place extension of subclassed types (0, 3, 3) Rerun
8. Registry: key set (0, 1, 1) Rerun
9. Registry: extension (0, 2, 2) Rerun
10. Registry: remain-linked (0, 2, 2) Rerun
11. Registry: multiget (0, 1, 1) Rerun
12. Registry: extended-multiget (0, 1, 1) Rerun
13. form.widget: compute_domain (0, 3, 3) Rerun
14. form.widget: compute_domain or (0, 3, 3) Rerun

8.2 Writing a test case

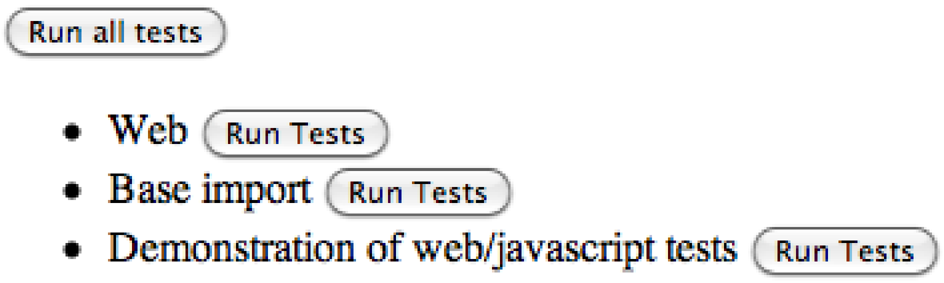
The first step is to list the test file(s). This is done through the `test` key of the openerp manifest, by adding javascript files to it (next to the usual YAML files, if any):

```
{
    'name': "Demonstration of web/javascript tests",
    'category': 'Hidden',
    'depends': ['web'],
    'test': ['static/test/demo.js'],
}
```

and to create the corresponding test file(s)

Note: Test files which do not exist will be ignored, if all test files of a module are ignored (can not be found), the test runner will consider that the module has no javascript tests.

After that, refreshing the runner selector will display the new module and allow running all of its (0 so far) tests:



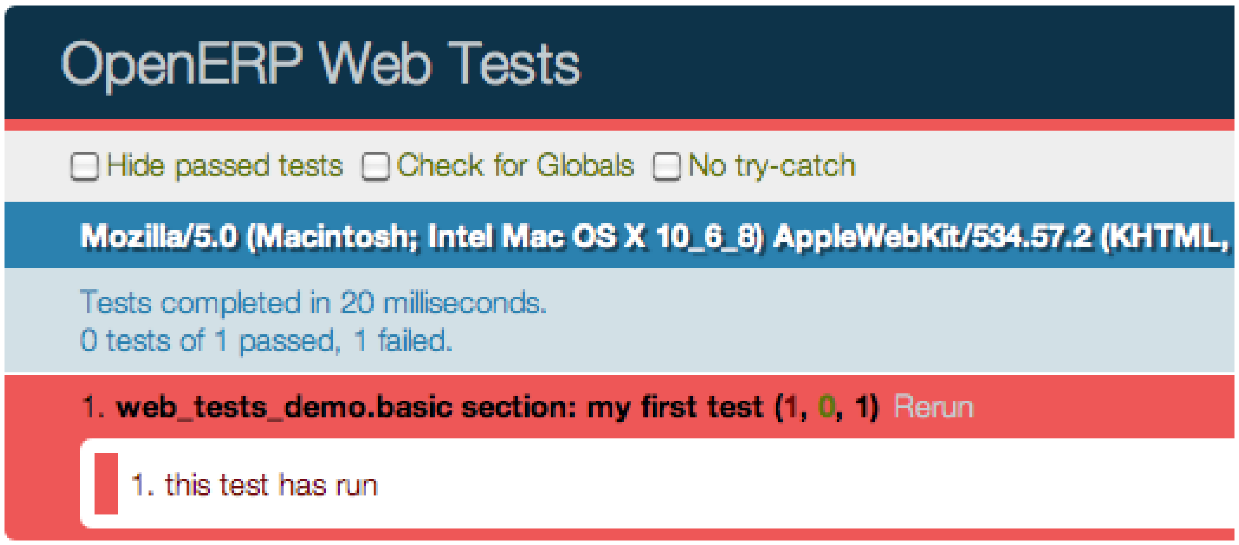
The next step is to create a test case:

```
openerp.testing.section('basic section', function (test) {
    test('my first test', function () {
        ok(false, "this test has run");
    });
});
```

All testing helpers and structures live in the `openerp.testing` module. OpenERP tests live in a *section()*, which is itself part of a module. The first argument to a section is the name of the section, the second one is the section body.

test, provided by the *section()* to the callback, is used to register a given test case which will be run whenever the test runner actually does its job. OpenERP Web test case use standard QUnit assertions within them.

Launching the test runner at this point will run the test and display the corresponding assertion message, with red colors indicating the test failed:



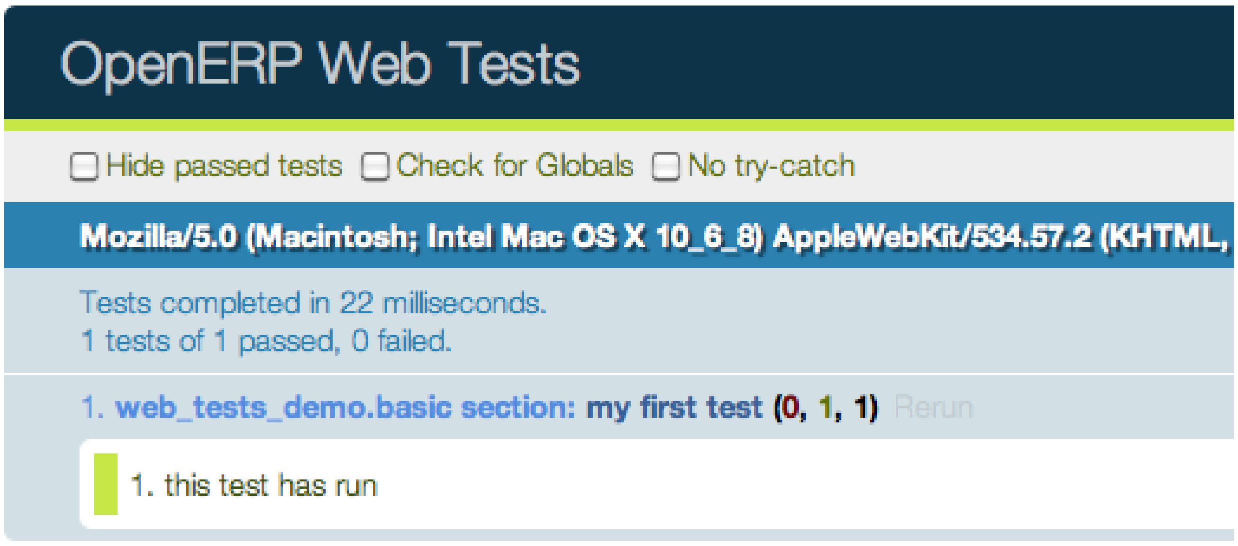
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/534.57.2 (KHTML,

Tests completed in 20 milliseconds.
0 tests of 1 passed, 1 failed.

1. web_tests_demo.basic section: my first test (1, 0, 1)

1. this test has run

Fixing the test (by replacing `false` to `true` in the assertion) will make it pass:



OpenERP Web Tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_6_8) AppleWebKit/534.57.2 (KHTML,

Tests completed in 22 milliseconds.
1 tests of 1 passed, 0 failed.

1. web_tests_demo.basic section: my first test (0, 1, 1) Rerun

1. this test has run

8.3 Assertions

As noted above, OpenERP Web's tests use qunit assertions. They are available globally (so they can just be called without references to anything). The following list is available:

ok (*state*[, *message*])
: checks that `state` is truthy (in the javascript sense)

strictEqual (*actual*, *expected*[, *message*])
: checks that the actual (produced by a method being tested) and expected values are identical (roughly equivalent to `ok(actual === expected, message)`)

notStrictEqual (*actual*, *expected*[, *message*])
: checks that the actual and expected values are *not* identical (roughly equivalent to `ok(actual !== expected, message)`)

deepEqual (*actual*, *expected*[, *message*])
: deep comparison between actual and expected: recurse into containers (objects and arrays) to ensure that they have the same keys/number of elements, and the values match.

notDeepEqual (*actual*, *expected*[, *message*])
: inverse operation to `deepEqual()`

throws (*block[, expected][, message]*)
: checks that, when called, the `block` throws an error. Optionally validates that error against `expected`.

  Arguments

  - **block** (*Function*) –
  - **expected** (*Error | RegExp*) – if a regexp, checks that the thrown error's message matches the regular expression. If an error type, checks that the thrown error is of that type.

equal (*actual*, *expected*[, *message*])
: checks that `actual` and `expected` are loosely equal, using the == operator and its coercion rules.

notEqual (*actual*, *expected*[, *message*])
: inverse operation to `equal()`

8.4 Getting an OpenERP instance

The OpenERP instance is the base through which most OpenERP Web modules behaviors (functions, objects, . . .) are accessed. As a result, the test framework automatically builds one, and loads the module being tested and all of its dependencies inside it. This new instance is provided as the first positional parameter to your test cases. Let's observe by adding javascript code (not test code) to the test module:

```
{
    'name': "Demonstration of web/javascript tests",
    'category': 'Hidden',
    'depends': ['web'],
    'js': ['static/src/js/demo.js'],
    'test': ['static/test/demo.js'],
}
```

```
// src/js/demo.js
openerp.web_tests_demo = function (instance) {
    instance.web_tests_demo = {
        value_true: true,
        SomeType: instance.web.Class.extend({
            init: function (value) {
                this.value = value;
            }
        })
    };
};
```

and then adding a new test case, which simply checks that the `instance` contains all the expected stuff we created in the module:

```
// test/demo.js
test('module content', function (instance) {
    ok(instance.web_tests_demo.value_true, "should have a true value");
    var type_instance = new instance.web_tests_demo.SomeType(42);
    strictEqual(type_instance.value, 42, "should have provided value");
});
```

8.5 DOM Scratchpad

As in the wider client, arbitrarily accessing document content is strongly discouraged during tests. But DOM access is still needed to e.g. fully initialize `widgets` before testing them.

Thus, a test case gets a DOM scratchpad as its second positional parameter, in a jQuery instance. That scratchpad is fully cleaned up before each test, and as long as it doesn't do anything outside the scratchpad your code can do whatever it wants:

```
// test/demo.js
test('DOM content', function (instance, $scratchpad) {
    $scratchpad.html('<div><span class="foo bar">ok</span></div>');
    ok($scratchpad.find('span').hasClass('foo'),
       "should have provided class");
});
test('clean scratchpad', function (instance, $scratchpad) {
    ok(!$scratchpad.children().length, "should have no content");
    ok(!$scratchpad.text(), "should have no text");
});
```

Note: The top-level element of the scratchpad is not cleaned up, test cases can add text or DOM children but shoud not alter `$scratchpad` itself.

8.6 Loading templates

To avoid the corresponding processing costs, by default templates are not loaded into QWeb. If you need to render e.g. widgets making use of QWeb templates, you can request their loading through the *`templates`* option to the *`test case function`*.

This will automatically load all relevant templates in the instance's qweb before running the test case:

```
{
    'name': "Demonstration of web/javascript tests",
    'category': 'Hidden',
    'depends': ['web'],
    'js': ['static/src/js/demo.js'],
    'test': ['static/test/demo.js'],
    'qweb': ['static/src/xml/demo.xml'],
}
```

```
<!-- src/xml/demo.xml -->
<templates id="template" xml:space="preserve">
    <t t-name="DemoTemplate">
        <t t-foreach="5" t-as="value">
            <p><t t-esc="value"/></p>
        </t>
    </t>
</templates>
```

```
// test/demo.js
test('templates', {templates: true}, function (instance) {
    var s = instance.web.qweb.render('DemoTemplate');
    var texts = $(s).find('p').map(function () {
        return $(this).text();
    }).get();

    deepEqual(texts, ['0', '1', '2', '3', '4']);
});
```

8.7 Asynchronous cases

The test case examples so far are all synchronous, they execute from the first to the last line and once the last line has executed the test is done. But the web client is full of asynchronous code, and thus test cases need to be async-aware.

This is done by returning a *`deferred`* from the case callback:

```
// test/demo.js
test('asynchronous', {
    asserts: 1
}, function () {
    var d = $.Deferred();
    setTimeout(function () {
        ok(true);
        d.resolve();
    }, 100);
    return d;
});
```

This example also uses the `options parameter` to specify the number of assertions the case should expect, if less or more assertions are specified the case will count as failed.

Asynchronous test cases *must* specify the number of assertions they will run. This allows more easily catching situations where e.g. the test architecture was not warned about asynchronous operations.

Note: Asynchronous test cases also have a 2 seconds timeout: if the test does not finish within 2 seconds, it will be considered failed. This pretty much always means the test will not resolve. This timeout *only* applies to the test itself, not to the setup and teardown processes.

Note: If the returned deferred is rejected, the test will be failed unless `fail_on_rejection` is set to `false`.

8.8 RPC

An important subset of asynchronous test cases is test cases which need to perform (and chain, to an extent) RPC calls.

Note: Because they are a subset of asynchronous cases, RPC cases must also provide a valid `assertions count`.

By default, test cases will fail when trying to perform an RPC call. The ability to perform RPC calls must be explicitly requested by a test case (or its containing test suite) through `rpc`, and can be one of two modes: `mock` or `rpc`.

8.8.1 Mock RPC

The preferred (and fastest from a setup and execution time point of view) way to do RPC during tests is to mock the RPC calls: while setting up the test case, provide what the RPC responses "should" be, and only test the code between the "user" (the test itself) and the RPC call, before the call is effectively done.

To do this, set the `rpc option` to `mock`. This will add a third parameter to the test case callback:

mock(*rpc_spec*, *handler*)

Can be used in two different ways depending on the shape of the first parameter:

- If it matches the pattern `model:method` (if it contains a colon, essentially) the call will set up the mocking of an RPC call straight to the OpenERP server (through XMLRPC) as performed via e.g. `openerp.web.Model.call()`.

  In that case, `handler` should be a function taking two arguments `args` and `kwargs`, matching the corresponding arguments on the server side and should simply return the value as if it were returned by the Python XMLRPC handler:

```
test('XML-RPC', {rpc: 'mock', asserts: 3}, function (instance, $s, mock) {
    // set up mocking
    mock('people.famous:name_search', function (args, kwargs) {
        strictEqual(kwargs.name, 'bob');
        return [
            [1, "Microsoft Bob"],
            [2, "Bob the Builder"],
            [3, "Silent Bob"]
        ];
    });

    // actual test code
    return new instance.web.Model('people.famous')
        .call('name_search', {name: 'bob'}).then(function (result) {
            strictEqual(result.length, 3, "shoud return 3 people");
            strictEqual(result[0][1], "Microsoft Bob",
                "the most famous bob should be Microsoft Bob");
        });
});
```

- Otherwise, if it matches an absolute path (e.g. `/a/b/c`) it will mock a JSON-RPC call to a web client controller, such as `/web/webclient/translations`. In that case, the handler takes a single `params` argument holding all of the parameters provided over JSON-RPC.

  As previously, the handler should simply return the result value as if returned by the original JSON-RPC handler:

```
    test('JSON-RPC', {rpc: 'mock', asserts: 3, templates: true}, function (instance, $s, mock) {
        var fetched_dbs = false, fetched_langs = false;
        mock('/web/database/get_list', function () {
            fetched_dbs = true;
            return ['foo', 'bar', 'baz'];
        });
        mock('/web/session/get_lang_list', function () {
            fetched_langs = true;
            return [['vo_IS', 'Hopelandic / Vonlenska']];
        });

        // widget needs that or it blows up
        instance.webclient = {toggle_bars: openerp.testing.noop};
        var dbm = new instance.web.DatabaseManager({});
        return dbm.appendTo($s).then(function () {
            ok(fetched_dbs, "should have fetched databases");
            ok(fetched_langs, "should have fetched languages");
            deepEqual(dbm.db_list, ['foo', 'bar', 'baz']);
        });
    });
```

Note: Mock handlers can contain assertions, these assertions should be part of the assertions count (and if multiple calls are made to a handler containing assertions, it multiplies the effective number of assertions).

8.8.2 Actual RPC

A more realistic (but significantly slower and more expensive) way to perform RPC calls is to perform actual calls to an actually running OpenERP server. To do this, set the `rpc option` to `rpc`, it will not provide any new parameter but will enable actual RPC, and the automatic creation and destruction of databases (from a specified source) around tests.

First, create a basic model we can test stuff with:

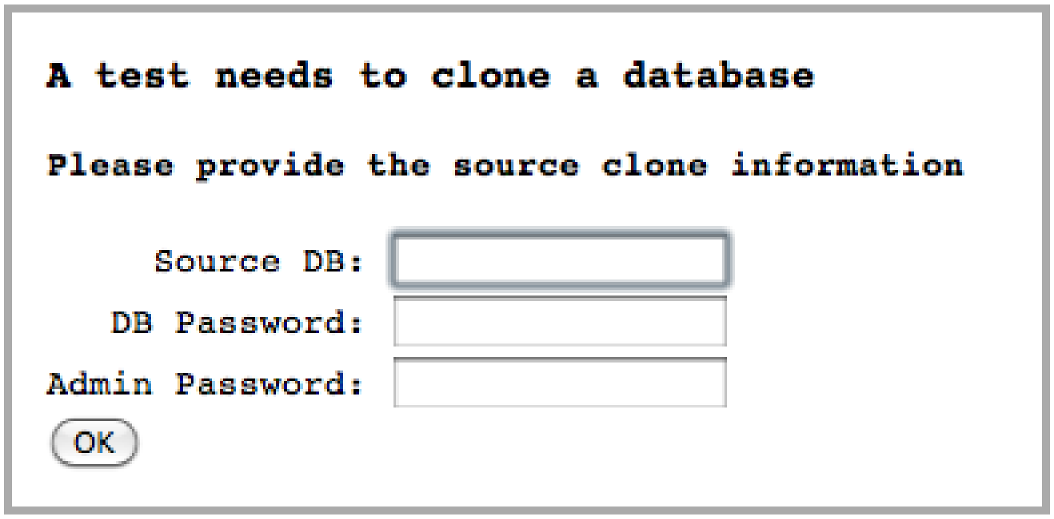
```
from openerp.osv import orm, fields

class TestObject(orm.Model):
    _name = 'web_tests_demo.model'

    _columns = {
        'name': fields.char("Name", required=True),
        'thing': fields.char("Thing"),
        'other': fields.char("Other", required=True)
    }
    _defaults = {
        'other': "bob"
    }
```

then the actual test:

```
test('actual RPC', {rpc: 'rpc', asserts: 4}, function (instance) {
    var Model = new instance.web.Model('web_tests_demo.model');
    return Model.call('create', [{name: "Bob"}])
        .then(function (id) {
            return Model.call('read', [[id]]);
        }).then(function (records) {
            strictEqual(records.length, 1);
            var record = records[0];
            strictEqual(record.name, "Bob");
            strictEqual(record.thing, false);
            // default value
            strictEqual(record.other, 'bob');
        });
});
```

This test looks like a "mock" RPC test but for the lack of mock response (and the different `rpc` type), however it has further ranging consequences in that it will copy an existing database to a new one, run the test in full on that temporary database and destroy the database, to simulate an isolated and transactional context and avoid affecting other tests. One of the consequences is that it takes a *long* time to run (5~10s, most of that time being spent waiting for a database duplication).

Furthermore, as the test needs to clone a database, it also has to ask which database to clone, the database/super-admin password and the password of the `admin` user (in order to authenticate as said user). As a result, the first time the test runner encounters an `rpc: "rpc"` test configuration it will produce the following prompt:



A test needs to clone a database

Please provide the source clone information

Source DB:

DB Password:

Admin Password:

and stop the testing process until the necessary information has been provided.

The prompt will only appear once per test run, all tests will use the same "source" database.

Note: The handling of that information is currently rather brittle and unchecked, incorrect values will likely crash the runner.

Note: The runner does not currently store this information (for any longer than a test run that is), the prompt will have to be filled every time.

8.9 Testing API

`openerp.testing.`**`section`**`(`*name*[, *options*], *body*`)`

A test section, serves as shared namespace for related tests (for constants or values to only set up once). The `body` function should contain the tests themselves.

Note that the order in which tests are run is essentially undefined, do *not* rely on it.

Arguments

- **`name`** (*`String`*) –
- **`options`** (`TestOptions`) –
- **`body`** (Function<*`case()`*, void>) –

`openerp.testing.`**`case`**`(`*name*[, *options*], *callback*`)`

Registers a test case callback in the test runner, the callback will only be run once the runner is started (or maybe not at all, if the test is filtered out).

Arguments

- **`name`** (*`String`*) –
- **`options`** (`TestOptions`) –
- **`callback`** (*`Function<instance, $, Function<String, Function, void>>`*) –

class `TestOptions``()`

the various options which can be passed to *`section()`* or *`case()`*. Except for *`setup`* and *`teardown`*, an option on *`case()`* will overwrite the corresponding option on *`section()`* so e.g. *`rpc`* can be set for a *`section()`* and then differently set for some *`case()`* of that *`section()`*

`TestOptions.`**`asserts`**

An integer, the number of assertions which should run during a normal execution of the test. Mandatory for asynchronous tests.

`TestOptions.`**`setup`**

Test case setup, run right before each test case. A section's *`setup()`* is run before the case's own, if both are specified.

`TestOptions.`**`teardown`**

Test case teardown, a case's *`teardown()`* is run before the corresponding section if both are present.

`TestOptions.`**`fail_on_rejection`**
If the test is asynchronous and its resulting promise is rejected, fail the test. Defaults to `true`, set to `false` to not fail the test in case of rejection:

```
// test/demo.js
test('unfail rejection', {
    asserts: 1,
    fail_on_rejection: false
}, function () {
    var d = $.Deferred();
    setTimeout(function () {
        ok(true);
        d.reject();
    }, 100);
    return d;
});
```

`TestOptions.`**`rpc`**
RPC method to use during tests, one of `"mock"` or `"rpc"`. Any other value will disable RPC for the test (if they were enabled by the suite for instance).

`TestOptions.`**`templates`**
Whether the current module (and its dependencies)'s templates should be loaded into QWeb before starting the test. A boolean, `false` by default.

The test runner can also use two global configuration values set directly on the `window` object:

- `oe_all_dependencies` is an `Array` of all modules with a web component, ordered by dependency (for a module `A` with dependencies `A'`, any module of `A'` must come before `A` in the array)
- `oe_db_info` is an object with 3 keys `source`, `supadmin` and `password`. It is used to pre-configure *actual RPC* tests, to avoid a prompt being displayed (especially for headless situations).

8.10 Running through Python

The web client includes the means to run these tests on the command-line (or in a CI system), but while actually running it is pretty simple the setup of the pre-requisite parts has some complexities.

1. Install unittest2 and QUnitSuite in your Python environment. Both can trivially be installed via pip or easy_install.

   The former is the unit-testing framework used by OpenERP, the latter is an adapter module to run qunit test suites and convert their result into something unittest2 can understand and report.

2. Install PhantomJS. It is a headless browser which allows automating running and testing web pages. QUnitSuite uses it to actually run the qunit test suite.

   The PhantomJS website provides pre-built binaries for some platforms, and your OS's package management probably provides it as well.

   If you're building PhantomJS from source, I recommend preparing for some knitting time as it's not exactly fast (it needs to compile both Qt and Webkit, both being pretty big projects).

   Note: Because PhantomJS is webkit-based, it will not be able to test if Firefox, Opera or Internet Explorer can correctly run the test suite (and it is only an approximation for Safari and Chrome). It is therefore recommended to *also* run the test suites in actual browsers once in a while.

Note: The version of PhantomJS this was build through is 1.7, previous versions *should* work but are not actually supported (and tend to just segfault when something goes wrong in PhantomJS itself so they're a pain to debug).

3. Set up OpenERP Command, which will be used to actually run the tests: running the qunit test suite requires a running server, so at this point OpenERP Server isn't able to do it on its own during the building/testing process.
4. Install a new database with all relevant modules (all modules with a web component at least), then restart the server

Note: For some tests, a source database needs to be duplicated. This operation requires that there be no connection to the database being duplicated, but OpenERP doesn't currently break existing/outstanding connections, so restarting the server is the simplest way to ensure everything is in the right state.

5. Launch `oe run-tests -d $DATABASE -mweb` with the correct addons-path specified (and replacing `$DATABASE` by the source database you created above)

Note: If you leave out `-mweb`, the runner will attempt to run all the tests in all the modules, which may or may not work.

If everything went correctly, you should now see a list of tests with (hopefully) `ok` next to their names, closing with a report of the number of tests run and the time it took:

```
test_empty_find (openerp.addons.web.tests.test_dataset.TestDataSetController) ... ok
test_ids_shortcut (openerp.addons.web.tests.test_dataset.TestDataSetController) ... ok
test_regular_find (openerp.addons.web.tests.test_dataset.TestDataSetController) ... ok
web.testing.stack: direct, value, success ... ok
web.testing.stack: direct, deferred, success ... ok
web.testing.stack: direct, value, error ... ok
web.testing.stack: direct, deferred, failure ... ok
web.testing.stack: successful setup ... ok
web.testing.stack: successful teardown ... ok
web.testing.stack: successful setup and teardown ... ok

[snip ~150 lines]

test_convert_complex_context (openerp.addons.web.tests.test_view.DomainsAndContextsTest) ... ok
test_convert_complex_domain (openerp.addons.web.tests.test_view.DomainsAndContextsTest) ... ok
test_convert_literal_context (openerp.addons.web.tests.test_view.DomainsAndContextsTest) ... ok
test_convert_literal_domain (openerp.addons.web.tests.test_view.DomainsAndContextsTest) ... ok
test_retrieve_nonliteral_context (openerp.addons.web.tests.test_view.DomainsAndContextsTest) ... ok
test_retrieve_nonliteral_domain (openerp.addons.web.tests.test_view.DomainsAndContextsTest) ... ok

----------------------------------------------------------------------
Ran 181 tests in 15.706s

OK
```

Congratulation, you have just performed a successful "offline" run of the OpenERP Web test suite.

Note: Note that this runs all the Python tests for the `web` module, but all the web tests for all of OpenERP. This can be surprising.

CHAPTER 9

Search View

OpenERP Web 7.0 implements a unified facets-based search view instead of the previous form-like search view (composed of buttons and multiple fields). The goal for this change is twofold:

- Avoid the common issue of users confusing the search view with a form view and trying to create their records through it (or entering all their data, hitting the `Create` button expecting their record to be created and losing everything).
- Improve the looks and behaviors of the view, and the fit within OpenERP Web's new design.

The internal structure of the faceted search is inspired by VisualSearch [1].

As does VisualSearch, the new search view is based on Backbone and makes significant use of Backbone's models and collections (OpenERP Web's widgets make a good replacement for Backbone's own views). As a result, understanding the implementation details of the OpenERP Web 7 search view also requires a basic understanding of Backbone's models, collections and events.

Note: This document may mention *fetching* data. This is a shortcut for "returning a `Deferred()` to [whatever is being fetched]". Unless further noted, the function or method may opt to return nothing by fetching `null` (which can easily be done by returning `$.when(null)`, which simply wraps the `null` in a Deferred).

9.1 Working with the search view: creating new inputs

The primary component of search views, as with all other OpenERP views, are inputs. The search view has two types of inputs — filters and fields — but only one is easly customizable: fields.

The mapping from OpenERP field types (and widgets) to search view objects is stored in the `openerp.web.search.fields` `Registry()` where new field types and widgets can be added.

Search view inputs have four main roles:

9.1.1 Loading defaults

Once the search view has initialized all its inputs, it will call `facet_for_defaults()` on each input, passing it a mapping (a javascript object) of `name:value` extracted from the action's context.

This method should fetch a `Facet()` (or an equivalent object) for the field's default value if applicable (if a default value for the field is found in the `defaults` mapping).

[1] the original view was implemented on top of a monkey-patched VisualSearch, but as our needs diverged from VisualSearch's goal this made less and less sense ultimately leading to a clean-room reimplementation

A default implementation is provided which checks if `defaults` contains a non-falsy value for the field's `@name` and calls `openerp.web.search.Input.facet_for()` with that value.

There is no default implementation of `openerp.web.search.Input.facet_for()` [2], but `openerp.web.search.Field()` provides one, which uses the value as-is to fetch a `Facet()`.

9.1.2 Providing completions

An important component of the new search view is the auto-completion pane, and the task of providing completion items is delegated to inputs through the `complete()` method.

This method should take a single argument (the string being typed by the user) and should fetch an `Array` of possible completions [3].

A default implementation is provided which fetches nothing.

A completion item is a javascript object with two keys (technically it can have any number of keys, but only these two will be used by the search view):

`label`

The string which will be displayed in the completion pane. It may be formatted using HTML (inline only), as a result if `value` is interpolated into it it *must* be escaped. `_.escape` can be used for this.

`facet`

Either a `Facet()` object or (more commonly) the corresponding attributes object. This is the facet which will be inserted into the search query if the completion item is selected by the user.

If the `facet` is not provided (not present, `null`, `undefined` or any other falsy value), the completion item will not be selectable and will act as a section title of sort (the `label` will be formatted differently). If an input *may* fetch multiple completion items, it *should* prefix those with a section title using its own name. This has no technical consequence but is clearer for users.

Note: If a field is `invisible`, its completion function will *not* be called.

9.1.3 Providing drawer/supplementary UI

For some inputs (fields or not), interaction via autocompletion may be awkward or even impossible.

These may opt to being rendered in a "drawer" as well or instead. In that case, they will undergo the normal widget lifecycle and be rendered inside the drawer.

Any input can note its desire to be rendered in the drawer by returning a truthy value from `in_drawer()`.

By default, `in_drawer()` returns the value of `_in_drawer`, which is `false`. The behavior can be toggled either by redefining the attribute to `true` (either on the class or on the input), or by overriding `in_drawer()` itself.

The input will be rendered in the full width of the drawer, it will be started only once (per view).

Todo

drawer API (if a widget wants to close the drawer in some way), part of the low-level SearchView API/interactions?

[2] In case you are extending the search view with a brand new type of input

[3] Ideally this array should not hold more than about 10 items, but the search view does not put any constraint on this at the moment. Note that this may change.

Todo

handle filters and filter groups via a “driver” input which dynamically collects, lays out and renders filters? => exercises drawer thingies

Note: An `invisible` input will not be inserted into the drawer.

9.1.4 Converting from facet objects

Ultimately, the point of the search view is to allow searching. In OpenERP this is done via domains. On the other hand, the OpenERP Web 7 search view’s state is modelled after a collection of *`Facet()`*, and each field of a search view may have special requirements when it comes to the domains it produces [5].

So there needs to be some way of mapping *`Facet()`* objects to OpenERP search data.

This is done via an input’s `get_domain()` and `get_context()`. Each takes a *`Facet()`* and returns whatever it’s supposed to generate (a domain or a context, respectively). Either can return `null` if the current value does not map to a domain or context, and can throw an `Invalid()` exception if the value is not valid at all for the field.

Note: The *`Facet()`* object can have any number of values (from 1 upwards)

Note: There is a third conversion method, `get_groupby()`, which returns an `Array` of groupby domains rather than a single context. At this point, it is only implemented on (and used by) filters.

9.2 Programmatic interactions: internal model

This new searchview is built around an instance of *`SearchQuery()`* available as `openerp.web.SearchView.query`.

The query is a backbone collection of *`Facet()`* objects, which can be interacted with directly by external objects or search view controls (e.g. widgets displayed in the drawer).

class `openerp.web.search.`**`SearchQuery`**`()`

The current search query of the search view, provides convenience behaviors for manipulating *`Facet()`* on top of the usual backbone collection methods.

The query ensures all of its facets contain at least one *`FacetValue()`* instance. Otherwise, the facet is automatically removed from the query.

`openerp.web.search.SearchQuery.`**`add`**`(`*values*, *options*`)`

Overridden from the base `add` method so that adding a facet which is *already* in the collection will merge the value of the new facet into the old one rather than add a second facet with different values.

Arguments

- **`values`** – facet, facet attributes or array thereof

Returns the collection itself

[5] search view fields may also bundle context data to add to the search context

`openerp.web.search.SearchQuery.`**`toggle`**`(`*value*, *options*`)`

Convenience method for toggling facet values in a query: removes the values (through the facet itself) if they are present, adds them if they are not. If the facet itself is not in the collection, adds it automatically.

A toggling is atomic: only one change event will be triggered on the facet regardless of the number of values added to or removed from the facet (if the facet already exists), and the facet is only removed from the query if it has no value *at the end* of the toggling.

Arguments

- **`value`** – facet or facet attributes

Returns the collection

class `openerp.web.search.`**`Facet`**`()`

A backbone model representing a single facet of the current research. May map to a search field, or to a more complex or fuzzier input (e.g. a custom filter or an advanced search).

`category`

The displayed name of the facet, as a `String`. This is a backbone model attribute.

`field`

The `Input()` instance which originally created the facet [4], used to delegate some operations (such as serializing the facet's values to domains and contexts). This is a backbone model attribute.

`values`

`FacetValues()` as a javascript attribute, stores all the values for the facet and helps propagate their events to the facet. Is also available as a backbone attribute (via `#get` and `#set`) in which cases it serializes to and deserializes from javascript arrays (via `Collection#toJSON` and `Collection#reset`).

`[icon]`

optional, a single ASCII letter (a-z or A-Z) mapping to the bundled mnmliconsRegular icon font.

When a facet with an `icon` attribute is rendered, the icon is displayed (in the icon font) in the first section of the facet instead of the `category`.

By default, only filters make use of this facility.

class `openerp.web.search.`**`FacetValues`**`()`

Backbone collection of *`FacetValue()`* instances.

class `openerp.web.search.`**`FacetValue`**`()`

Backbone model representing a single value within a facet, represents a pair of (displayed name, logical value).

`label`

Backbone model attribute storing the "displayable" representation of the value, visually output to the user. Must be a string.

`value`

Backbone model attribute storing the logical/internal value (of itself), will be used by `Input()` to serialize to domains and contexts.

Can be of any type.

[4] `field` does not actually need to be an instance of `Input()`, nor does it need to be what created the facet, it just needs to provide the three facet-serialization methods `get_domain()`, `get_context()` and `get_gropuby()`, existing `Input()` subtypes merely provide convenient base implementation for those methods.

Complex search view inputs (especially those living in the drawer) may prefer using object literals with the right slots returning closed-over values or some other scheme un-bound to an actual `Input()`, as `CustomFilters()` and `Advanced()` do.

9.3 Field services

`Field()` provides a default implementation of `get_domain()` and `get_context()` taking care of most of the peculiarities pertaining to OpenERP's handling of fields in search views. It also provides finer hooks to let developers of new fields and widgets customize the behavior they want without necessarily having to reimplement all of `get_domain()` or `get_context()`:

`openerp.web.search.Field.`**`get_context`**(*facet*)

If the field has no `@context`, simply returns `null`. Otherwise, calls *`value_from()`* once for each *`FacetValue()`* of the current *`Facet()`* (in order to extract the basic javascript object from the *`FacetValue()`* then evaluates `@context` with each of these values set as `self`, and returns the union of all these contexts.

Arguments

- **`facet`** (`openerp.web.search.Facet`) –

Returns a context (literal or compound)

`openerp.web.search.Field.`**`get_domain`**(*facet*)

If the field has no `@filter_domain`, calls *`make_domain()`* once with each *`FacetValue()`* of the current *`Facet()`* as well as the field's `@name` and either its `@operator` or *`default_operator`*.

If the field has an `@filter_value`, calls *`value_from()`* once per *`FacetValue()`* and evaluates `@filter_value` with each of these values set as `self`.

In either case, "ors" all of the resulting domains (using `|`) if there is more than one *`FacetValue()`* and returns the union of the result.

Arguments

- **`facet`** (`openerp.web.search.Facet`) –

Returns a domain (literal or compound)

`openerp.web.search.Field.`**`make_domain`**(*name*, *operator*, *facetValue*)

Builds a literal domain from the provided data. Calls *`value_from()`* on the *`FacetValue()`* and evaluates and sets it as the domain's third value, uses the other two parameters as the first two values.

Can be overridden to build more complex default domains.

Arguments

- **`name`** (*`String`*) – the field's name
- **`operator`** (*`String`*) – the operator to use in the field's domain
- **`facetValue`** (`openerp.web.search.FacetValue`) –

Returns Array<(String, String, Object)>

`openerp.web.search.Field.`**`value_from`**(*facetValue*)

Extracts a "bare" javascript value from the provided *`FacetValue()`*, and returns it.

The default implementation will simply return the `value` backbone property of the argument.

Arguments

- **`facetValue`** (`openerp.web.search.FacetValue`) –

Returns Object

`openerp.web.search.Field.`**`default_operator`**

Operator used to build a domain when a field has no `@operator` or `@filter_domain`. `"="` for `Field()`

9.4 Arbitrary data storage

`Facet()` and *`FacetValue()`* objects (and structures) provided by your widgets should never be altered by the search view (or an other widget). This means you are free to add arbitrary fields in these structures if you need to (because you have more complex needs than the attributes described in this document).

Ideally this should be avoided, but the possibility remains.

9.5 Changes

Todo

merge in changelog instead?

The displaying of the search view was significantly altered from OpenERP Web 6.1 to OpenERP Web 7.

As a result, while the external API used to interact with the search view does not change many internal details — including the interaction between the search view and its widgets — were significantly altered:

9.5.1 Internal operations

- `openerp.web.SearchView.do_clear()` has been removed
- `openerp.web.SearchView.do_toggle_filter()` has been removed

9.5.2 Widgets API

- `openerp.web.search.Widget.render()` has been removed
- `openerp.web.search.Widget.make_id()` has been removed
- Search field objects are not openerp widgets anymore, their `start` is not generally called
- `clear()` has been removed since clearing the search view now simply consists of removing all search facets
- `get_domain()` and `get_context()` now take a *`Facet()`* as parameter, from which it's their job to get whatever value they want
- `get_groupby()` has been added. It returns an `Array()` of context-like constructs. By default, it does not do anything in `Field()` and it returns the various contexts of its enabled filters in `FilterGroup()`.

9.5.3 Filters

- `openerp.web.search.Filter.is_enabled()` has been removed
- `FilterGroup()` instances are still rendered (and started) in the "advanced search" drawer.

9.5.4 Fields

- `get_value` has been replaced by *`value_from()`* as it now takes a *`FacetValue()`* argument (instead of no argument). It provides a default implementation returning the `value` property of its argument.
- The third argument to *`make_domain()`* is now a *`FacetValue()`* so child classes have all the information they need to derive the "right" resulting domain.

9.5.5 Custom filters

Instead of being an intrinsic part of the search view, custom filters are now a special case of filter groups. They are treated specially still, but much less so than they used to be.

9.5.6 Many To One

- Because the autocompletion service is now provided by the search view itself, `openerp.web.search.ManyToOneField.setup_autocomplete()` has been removed.

9.5.7 Advanced Search

- The advanced search is now a more standard `Input()` configured to be rendered in the drawer.
- `Field()` are now standard widgets, with the "right" behaviors (they don't rebind their `$element` in `start()`)
- The ad-hoc optional setting of the openerp field descriptor on a `Field()` has been removed, the field descriptor is now passed as second argument to the `Field()`'s constructor, and bound to its `field`.
- Instead of its former domain triplet `(field, operator, value)`, `get_proposition()` now returns an object with two fields `label` and `value`, respectively a human-readable version of the proposition and the corresponding domain triplet for the proposition.

List View

10.1 Style Hooks

The list view provides a few style hook classes for re-styling of list views in various situations:

`.oe_list`

The root element of the list view, styling rules should be rooted on that class.

`table.oe_list_content`

The root table for the listview, accessory components may be generated or added outside this section, this is the list view "proper".

`.oe_list_buttons`

The action buttons array for the list view, with its sub-elements

`.oe_list_add`

The default "Create"/"Add" button of the list view

`.oe_alternative`

The "alternative choice" for the list view, by default text along the lines of "or import" with a link.

`.oe_list_field_cell`

The cell (`td`) for a given field of the list view, cells which are *not* fields (e.g. name of a group, or number of items in a group) will not have this class. The field cell can be further specified:

`.oe_number`

Numeric cell types (integer and float)

`.oe_button`

Action button (`button` tag in the view) inside the cell

`.oe_readonly`

Readonly field cell

`.oe_list_field_$type`

Additional class for the precise type of the cell, `$type` is the field's @widget if there is one, otherwise it's the field's type.

`.oe_list_record_selector`

Selector cells

10.1.1 Editable list view

The editable list view module adds a few supplementary style hook classes, for edition situations:

`.oe_list_editable`

Added to the `.oe_list` when the list is editable (however that was done). The class may be removed on-the-fly if the list becomes non-editable.

`.oe_editing`

Added to both `.oe_list` and `.oe_list_button` (as the buttons may be outside of the list view) when a row of the list is currently being edited.

`tr.oe_edition`

Class set on the row being edited itself. Note that the edition form is *not* contained within the row, this allows for styling or modifying the row while it's being edited separately. Mostly for fields which can not be edited (e.g. read-only fields).

10.2 Columns display customization

The list view provides a registry to *`openerp.web.list.Column()`* objects allowing for the customization of a column's display (e.g. so that a binary field is rendered as a link to the binary file directly in the list view).

The registry is `instance.web.list.columns`, the keys are of the form `tag.type` where `tag` can be `field` or `button`, and `type` can be either the field's type or the field's `@widget` (in the view).

Most of the time, you'll want to define a `tag.widget` key (e.g. `field.progressbar`).

class `openerp.web.list.`**`Column`**(*id*, *tag*, *attrs*)

`openerp.web.list.Column.`**`format`**(*record_data*, *options*)

Top-level formatting method, returns an empty string if the column is invisible (unless the `process_modifiers=false` option is provided); returns `options.value_if_empty` or an empty string if there is no value in the record for the column.

Otherwise calls *`_format()`* and returns its result.

This method only needs to be overridden if the column has no concept of values (and needs to bypass that check), for a button for instance.

Otherwise, custom columns should generally override *`_format()`* instead.

Returns String

`openerp.web.list.Column.`**`_format`**(*record_data*, *options*)

Never called directly, called if the column is visible and has a value.

The default implementation calls `format_value()` and htmlescapes the result (via `_.escape`).

Note that the implementation of *`_format()`* *must* escape the data provided to it, its output will *not* be escaped by *`format()`*.

Returns String

10.3 Editable list view

List view edition is an extension to the base listview providing the capability of inline record edition by delegating to an embedded form view.

10.3.1 Editability status

The editability status of a list view can be queried through the `editable()` method, will return a falsy value if the listview is not currently editable.

The editability status is based on three flags:

`tree/@editable`

> If present, can be either `"top"` or `"bottom"`. Either will make the list view editable, with new records being respectively created at the top or at the bottom of the view.

`context.set_editable`

> Boolean flag extracted from a search context (during the `do_search`()` handler), `true` will make the view editable (from the top), `false` or the absence of the flag is a noop.

`defaults.editable`

> Like `tree/@editable`, one of absent (`null`)), `"top"` or `"bottom"`, fallback for the list view if none of the previous two flags are set.

These three flags can only *make* a listview editable, they can *not* override a previously set flag. To do that, a listview user should instead cancel *the edit:before event*.

The editable list view module adds a number of methods to the list view, on top of implementing the *`EditorDelegate()`* protocol:

10.3.2 Interaction Methods

`openerp.web.ListView.`**`ensure_saved`**`()`

> Attempts to resolve the pending edition, if any, by saving the edited row's current state.
>
> **Returns** delegate resolving to all editions having been saved, or rejected if a pending edition could not be saved (e.g. validation failure)

`openerp.web.ListView.`**`start_edition`**`(`*[record][, options]*`)`

> Starts editing the provided record inline, through an overlay form view of editable fields in the record.
>
> If no record is provided, creates a new one according to the editability configuration of the list view.
>
> This method resolves any pending edition when invoked, before starting a new edition.
>
> **Arguments**
>
> - **`record`** (`Record()`) – record to edit, or null to create a new record
> - **`options`** (`EditOptions`) –
>
> **Returns** delegate to the form used for the edition

`openerp.web.ListView.`**`save_edition`**`()`

> Resolves the pending edition.

Returns delegate to the save being completed, resolves to an object with two attributes `created` (flag indicating whether the saved record was just created or was updated) and `record` the reloaded record having been edited.

`openerp.web.ListView.`**`cancel_edition`**([*force=false*])
Cancels pending edition, cleans up the list view in case of creation (removes the empty record being created).

Arguments

- **`force`** (*`Boolean`*) – doesn't check if the user has added any data, discards the edition unconditionally

10.3.3 Utility Methods

`openerp.web.ListView.`**`get_cells_for`**(*row*)
Extracts the cells from a listview row, and puts them in a {fieldname: cell} mapping for analysis and manipulation.

Arguments

- **`row`** (*`jQuery`*) –

Return type Object

`openerp.web.ListView.`**`with_event`**(*event_name, event, action[, args][, trigger_params]*)
Executes `action` in the context of the view's editor, bracketing it with cancellable event signals.

Arguments

- **`event_name`** (*`String`*) – base name for the bracketing event, will be postfixed by `:before` and `:after` before being called (respectively before and after `action` is executed)
- **`event`** (*`Object`*) – object passed to the `:before` event handlers.
- **`action`** (*`Function`*) – function called with the view's editor as its `this`. May return a deferred.
- **`args`** (*`Array`*) – arguments passed to `action`
- **`trigger_params`** (*`Array`*) – arguments passed to the `:after` event handler alongside the results of `action`

10.3.4 Behavioral Customizations

`openerp.web.ListView.`**`handle_onwrite`**(*record*)
Implements the handling of the `onwrite` listview attribute: calls the RPC methods specified by `@onwrite`, and if that method returns an array of ids loads or reloads the records corresponding to those ids.

Arguments

- **`record`** (*`openerp.web.list.Record`*) – record being written having triggered the `onwrite` callback

Returns deferred to all reloadings being done

10.3.5 Events

For simpler interactions by/with external users of the listview, the view provides a number of dedicated events to its lifecycle.

Note: if an event is defined as *cancellable*, it means its first parameter is an object on which the `cancel` attribute can be set. If the `cancel` attribute is set, the view will abort its current behavior as soon as possible, and rollback any state modification.

Generally speaking, an event should only be cancelled (by setting the `cancel` flag to `true`), uncancelling an event is undefined as event handlers are executed on a first-come-first-serve basis and later handlers may re-cancel an uncancelled event.

`edit:before` *cancellable*

Invoked before the list view starts editing a record.

Provided with an event object with a single property `record`, holding the attributes of the record being edited (`record` is empty *but not null* for a new record)

`edit:after`

Invoked after the list view has gone into an edition state, provided with the attributes of the record being edited (see `edit:before`) as first parameter and the form used for the edition as second parameter.

`save:before` *cancellable*

Invoked right before saving a pending edition, provided with an event object holding the listview's editor (`editor`) and the edition form (`form`)

`save:after`

Invoked after a save has been completed

`cancel:before` *cancellable*

Invoked before cancelling a pending edition, provided with the same information as `save:before`.

`cancel:after`

Invoked after a pending edition has been cancelled.

10.3.6 DOM events

The list view has grown hooks for the `keyup` event on its edition form (during edition): any such event bubbling out of the edition form will be forwarded to a method `keyup_EVENTNAME`, where `EVENTNAME` is the name of the key in `$.ui.keyCode`.

The method will also get the event object (originally passed to the `keyup` handler) as its sole parameter.

The base editable list view has handlers for the `ENTER` and `ESCAPE` keys.

10.4 Editor

The list-edition modules does not generally interact with the embedded formview, delegating instead to its *`Editor()`*.

class `openerp.web.list.`**`Editor`**(*parent*[, *options*])

The editor object provides a more convenient interface to form views, and simplifies the usage of form views for semi-arbitrary edition of stuff.

However, the editor does *not* task itself with being internally consistent at this point: calling e.g. `edit()` multiple times in a row without saving or cancelling each edit is undefined.

Arguments

- **`parent`** (*`Widget()`*) –
- **`options`** (`EditorOptions`) –

`openerp.web.list.Editor.`**`is_editing`**([*record_state*])

Indicates whether the editor is currently in the process of providing edition for a record.

Can be filtered by the state of the record being edited (whether it's a record being *created* or a record being *altered*), in which case it asserts both that an edition is underway and that the record being edited respectively does not yet exist in the database or already exists there.

Arguments

- **`record_state`** (*`String`*) – state of the record being edited. Either `"new"` or `"edit"`.

Return type Boolean

`openerp.web.list.Editor.`**`edit`**(*record*, *configureField*[, *options*])

Loads the provided record into the internal form view and displays the form view.

Will also attempt to focus the first visible field of the form view.

Arguments

- **`record`** (*`Object`*) – record to load into the form view (key:value mapping similar to the result of a `read`)
- **`configureField`** (*`Function<String, openerp.web.form.Field>`*) – function called with each field of the form view right after the form is displayed, lets whoever called this method do some last-minute configuration of form fields.
- **`options`** (`EditOptions`) –

Returns jQuery delegate to the form object

`openerp.web.list.Editor.`**`save`**()

Attempts to save the internal form, then hide it

Returns delegate to the record under edition (with `id` added for a creation). The record is not updated from when it was passed in, aside from the `id` attribute.

`openerp.web.list.Editor.`**`cancel`**([*force=false*])

Attemps to cancel the edition of the internal form, then hide the form

Arguments

- **`force`** (*`Boolean`*) – unconditionally cancels the edition of the internal form, even if the user has already entered data in it.

Returns delegate to the record under edition

class **`EditorOptions`**()

`EditorOptions.`**`formView`**
Form view (sub)-class to instantiate and delegate edition to.

By default, `FormView()`

`EditorOptions.`**`delegate`**
Object used to get various bits of information about how to display stuff.

By default, uses the editor's parent widget. See *`EditorDelegate()`* for the methods and attributes to provide.

class **`EditorDelegate`**`()`
Informal protocol defining the methods and attributes expected of the *`Editor()`*'s delegate.

`EditorDelegate.`**`dataset`**
The dataset passed to the form view to synchronize the form view and the outer widget.

`EditorDelegate.`**`edition_view`**`(`*editor*`)`
Called by the *`Editor()`* object to get a form view (JSON) to pass along to the form view it created.

The result should be a valid form view, see Form Notes for various peculiarities of the form view format.

Arguments

- **`editor`** (*`Editor()`*) – editor object asking for the view

Returns form view

Return type Object

`EditorDelegate.`**`prepends_on_create`**`()`
By default, the *`Editor()`* will append the ids of newly created records to the *`EditorDelegate.dataset`*. If this method returns `true`, it will prepend these ids instead.

Returns whether new records should be prepended to the dataset (instead of appended)

Return type Boolean

class **`EditOptions`**`()`
Options object optionally passed into a method starting an edition to configure its setup and behavior

`focus_field`
Name of the field to set focus on after setting up the edition of the record.

If this option is not provided, or the requested field can not be focused (invisible, readonly or not in the view), the first visible non-readonly field is focused.

10.5 Changes from 6.1

- The editable listview behavior has been rewritten pretty much from scratch, any code touching on editability will have to be modified
  - The overloading of `Groups()` and `List()` for editability has been drastically simplified, and most of the behavior has been moved to the list view itself. Only `row_clicked()` is still overridden.
  - A new method `get_row_for(record) -> jQuery(tr) | null` has been added to both ListView.List and ListView.Group, it can be called from the list view to get the table row matching a record (if such a row exists).
- `do_button_action()`'s core behavior has been split away to `handle_button()`. This allows bypassing overrides of `do_button_action()` in a parent class.

  Ideally, `handle_button()` should not be overridden.

- Modifiers handling has been improved (all modifiers information should now be available through `modifiers_for()`, not just `invisible`)
- Changed some handling of the list view's record: a record may now have no id, and the listview will handle that correctly (for new records being created) as well as correctly handle the `id` being set.
- Extended the internal collections structure of the list view with #find, #succ and #pred.

CHAPTER 11

Notes on the usage of the Form View as a sub-widget

11.1 Undocumented stuff

- `initial_mode` *option* defines the starting mode of the form view, one of `view` and `edit` (?). Default value is `view` (non-editable form).
- `embedded_view` *attribute* has to be set separately when providing a view directly, no option available for that usage.
  - View arch **must** contain node with `@class="oe_form_container"`, otherwise everything will break without any info
  - Root element of view arch not being `form` may or may not work correctly, no idea.
  - Freeform views => `@version="7.0"`
- Form is not entirely loaded (some widgets may not appear) unless `on_record_loaded` is called (or `do_show`, which itself calls `on_record_loaded`).
- "Empty" form => `on_button_new` (...), or manually call `default_get` + `on_record_loaded`
- Form fields default to width: 100%, padding, !important margin, can be reached via `.oe_form_field`
- Form *will* render buttons and a pager, offers options to locate both outside of form itself (`$buttons` and `$pager`), providing empty jquery objects (`$()`) seems to stop displaying both but not sure if there are deleterious side-effects.

  Other options:

  - Pass in `$(document.createDocumentFragment)` to ensure it's a DOM-compatible tree completely outside of the actual DOM.
  - ???
- readonly fields probably don't have a background, beware if need of overlay
  - What is the difference between `readonly` and `effective_readonly`?
- No facilities for DOM events handling/delegations e.g. handling keyup/keydown/keypress from a form fields into the form's user.
  - Also no way to reverse from a DOM node (e.g. DOMEvent#target) back to a form view field easily

CHAPTER 12

API changes from OpenERP Web 6.1 to 7.0

12.1 Supported browsers

The OpenERP Web Client supports the following web browsers:

- Internet Explorer 9+
- Google Chrome 22+
- Firefox 13+
- Any browser using the latest version of Chrome Frame

12.2 DataSet -> Model

The 6.1 `DataSet` API has been deprecated in favor of the smaller and more orthogonal Model API, which more closely matches the API in OpenERP Web's Python side and in OpenObject addons and removes most stateful behavior of DataSet.

12.2.1 Migration guide

- Actual arbitrary RPC calls can just be remapped on a *`Model()`* instance:

```
dataset.call(method, args)
```

  or

```
dataset.call_and_eval(method, args)
```

  can be replaced by calls to *`openerp.web.Model.call()`*:

```
model.call(method, args)
```

  If callbacks are passed directly to the older methods, they need to be added to the new one via `.then()`.

  Note: The `context_index` and `domain_index` features were not ported, context and domain now need to be passed in "in full", they won't be automatically filled with the user's current context.

- Shorcut methods (`name_get`, `name_search`, `unlink`, `write`, ...) should be ported to *`openerp.web.Model.call()`*, using the server's original signature. On the other hand, the non-shortcut equivalents can now use keyword arguments (see *`call()`*'s signature for details)
- `read_slice`, which allowed a single round-trip to perform a search and a read, should be reimplemented via *`Query()`* objects (see: *`query()`*) for clearer and simpler code. `read_index` should be replaced by a *`Query()`* as well, combining `offset()` and *`first()`*.

12.2.2 Rationale

Renaming

> The name *DataSet* exists in the CS community consciousness, and (as its name implies) it's a set of data (often fetched from a database, maybe lazily). OpenERP Web's dataset behaves very differently as it does not store (much) data (only a bunch of ids and just enough state to break things). The name "Model" matches the one used on the Python side for the task of building an RPC proxy to OpenERP objects.

API simplification

> `DataSet` has a number of methods which serve as little more than shortcuts, or are there due to domain and context evaluation issues in 6.1.
>
> The shortcuts really add little value, and OpenERP Web 6.2 embeds a restricted Python evaluator (in javascript) meaning most of the context and domain parsing & evaluation can be moved to the javascript code and does not require cooperative RPC bridging.

12.3 DataGroup -> also Model

Alongside the deprecation of `DataSet` for *`Model()`*, OpenERP Web 7.0 removes `DataGroup` and its subtypes as public objects in favor of a single method on *`Query()`*: *`group_by()`*.

12.3.1 Migration guide

12.3.2 Rationale

While the `DataGroup` API worked (mostly), it is quite odd and alien-looking, a bit too Smalltalk-inspired (behaves like a self-contained flow-control structure for reasons which may or may not have been good).

Because it is heavily related to `DataSet` (as it *yields* `DataSet` objects), deprecating `DataSet` automatically deprecates `DataGroup` (if we want to stay consistent), which is a good time to make the API more imperative and look more like what most developers are used to.

But as `DataGroup` users in 6.1 were rare (and there really was little reason to use it), it has been removed as a public API.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Index