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 Op
This allows providing more seamless features by integrating your module deeper into the web client.

A Basic Module

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

```
web_example
├── __init__.py
└── __openerp__.py
```

section diff file
__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.

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 roo

```
web_example
├── __init__.py
├── __openerp__.py
└── static
```

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

section diff file
__openerp__.py

```
    'name': "Web Example",
    'description': "Basic example of a (future) web module",
    'category': 'Hidden',
    '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
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 con
`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, u
your module. However there are still a few more steps to running javascript code.

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.

section diff file
static/src/js/first_module.js

```
// 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 patter

section diff file
__openerp__.py

```
    '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
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 browse
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 r
server). This is done by providing a javascript module:

section diff file
static/src/js/first_module.js

```
// static/src/js/first_module.js
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 depen
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:

section diff file
__openerp__.py

```
    '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.xml

```
<!-- 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):

section diff file
static/src/js/first_module.js

```
// 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);
    };
};
```

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
make the message appear, as usual, in the browser's console.

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 *Wi*

will simply use its **start()** to add some content to its DOM:

◉ section ○ diff ○ file
static/src/js/first_module.js

```
// 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({
        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 i

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:

◉ section ○ diff ○ file
static/src/css/web_example.css

```
.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):

◉ section ○ diff ○ file
__openerp__.py

```
    '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 hier 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 tc 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 templa 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 othe engines may not provide, and has special integration to OpenERP Web widgets.

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

◉ section ○ diff ○ file
__openerp__.py

```
    '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'],
}
```

static/src/xml/web_example.xml

```
<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>
```

The template can then easily be hooked in the widget:

section diff file
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'
    });
};
```

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

section diff file
static/src/css/web_example.css

```
    color: white;
    background-color: black;
    height: 100%;
}
.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 wi 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 hig 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:

section diff file
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'
        },
        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 comes some actual logic:

section diff file
static/src/js/first_module.js

```
            '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 `_watc` 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 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 " cleanup).

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

Burning through the skies

All work so far has been "local" outside of the original impetus provided by the client action: the widget is self-contained and, once started, does not communicate v outside itself. Not only that, but it has no persistence: if the user leaves the stopwatch screen (to go and see his inbox, or do some well-deserved accounting, for inst was being timed will be lost.

To prevent this irremediable loss, we can use OpenERP's support for storing data as a model, allowing so that we don't lose our data and can later retrieve, query a First let's create a basic OpenERP model in which our data will be stored:

◉ section ○ diff ○ file
__init__.py

```
# __init__.py
from openerp.osv import orm, fields


class Times(orm.Model):
    _name = 'web_example.stopwatch'

    _columns = {
        'time': fields.integer("Time", required=True,
                               help="Measured time in milliseconds"),
        'user_id': fields.many2one('res.users', "User", required=True,
                                   help="User who registered the measurement")
    }
```

then let's add saving times to the database every time the stopwatch is stopped, using **the "high-level" Model API**:

◉ section ○ diff ○ file
static/src/js/first_module.js

```
            this._start = null;
            this._watch = null;
        },
        current: function () {
            // Subtracting javascript dates returns the difference in milliseconds
            return new Date() - this._start;
        },
        update_counter: function (time) {
            var h, m, s;
            s = time / 1000;
            m = Math.floor(s / 60);
            s -= 60*m;
            h = Math.floor(m / 60);
```

```
                .removeClass('oe_web_example_stopped');
            this._start = new Date();
            // Update the UI to the current time
            this.update_counter(this.current());
            // Update the counter at 30 FPS (33ms/frame)
            this._watch = setInterval(function () {
                    this.update_counter(this.current());
                }.bind(this),
                33);
        },
        watch_stop: function () {
            clearInterval(this._watch);
            var time = this.current();
            this.update_counter(time);
            this._start = this._watch = null;
            this.$el.removeClass('oe_web_example_started')
                    .addClass('oe_web_example_stopped');
            new instance.web.Model('web_example.stopwatch').call('create', [{
                user_id: instance.session.uid,
                time: time,
            }]);
        },
        destroy: function () {
            if (this._watch) {
```

A look at the “Network” tab of your preferred browser’s developer tools while playing with the stopwatch will show that the save (creation) request is indeed sent (a
though we’re ignoring the response at this point).

These saved data should now be loaded and displayed when first opening the action, so the user can see his previously recorded times. This is done by overloading t
method: the purpose of **`start()`** is to perform *asynchronous* initialization steps, so the rest of the web client knows to “wait” and gets a readiness signal. In this cas
data recorded previously using the **`Query()`** interface and add this data to an ordered list added to the widget’s template:

section diff file

static/src/js/first_module.js

```
            this._super.apply(this, arguments);
            this._start = null;
            this._watch = null;
            this.model = new instance.web.Model('web_example.stopwatch');
        },
        start: function () {
            var display = this.display_record.bind(this);
            return this.model.query()
                .filter([['user_id', '=', instance.session.uid]])
                .all().done(function (records) {
                    _(records).each(display);
                });
        },
        current: function () {
            // Subtracting javascript dates returns the difference in milliseconds
            return new Date() - this._start;
        },
        display_record: function (record) {
            $('<li>')
                .text(this.format_time(record.time))
                .appendTo(this.$('.oe_web_example_saved'));
        },
        format_time: function (time) {
            var h, m, s;
            s = time / 1000;
            m = Math.floor(s / 60);
            s -= 60*m;
            h = Math.floor(m / 60);
            m -= 60*h;
            return _.str.sprintf("%02d:%02d:%02d", h, m, s);
        },
        update_counter: function (time) {
            this.$('.oe_web_example_timer').text(this.format_time(time));
        },
        watch_start: function () {
            this.$el.addClass('oe_web_example_started')


            this._start = this._watch = null;
            this.$el.removeClass('oe_web_example_started')
                    .addClass('oe_web_example_stopped');
            this.model.call('create', [{
                user_id: instance.session.uid,
                time: time,
            }]);
```

static/src/xml/web_example.xml

```
    <p class="oe_web_example_stop">
        <button type="button">Stop</button>
    </p>
    <ol class="oe_web_example_saved"></ol>
</div>
</templates>
```

And for consistency's sake (so that the display a user leaves is pretty much the same as the one he comes back to), newly created records should also automatically list:

section diff file
static/src/js/first_module.js

```
            33);
        },
        watch_stop: function () {
            var self = this;
            clearInterval(this._watch);
            var time = this.current();
            this.update_counter(time);
            this._start = this._watch = null;
            this.$el.removeClass('oe_web_example_started')
                    .addClass('oe_web_example_stopped');
            var record = {
                user_id: instance.session.uid,
                time: time,
            };
            this.model.call('create', [record]).done(function () {
                self.display_record(record);
            });
        },
        destroy: function () {
            if (this._watch) {
```

Note that we're only displaying the record once we know it's been saved from the database (the `create` call has returned without error).

Mic check, is this working?

So far, features have been implemented, code has been worked and tentatively tried. However, there is no guarantee they will *keep working* as new changes are pe features added, ...

The original author (you, dear reader) could keep a notebook with a list of workflows to check, to ensure everything keeps working. And follow the notebook day a time something is changed in the module.

That gets repetitive after a while. And computers are good at doing repetitive stuff, as long as you tell them how to do it.

So let's add test to the module, so that in the future the computer can take care of ensuring what works today keeps working tomorrow.

Note:

Here we're writing tests after having implemented the widget. This may or may not work, we may need to alter bits and pieces of code to get them in a testable sta testing methodology is TDD where the tests are written first, and the code necessary to make these tests pass is written afterwards.

Both methods have their opponents and detractors, advantages and inconvenients. Pick the one you prefer.

The first step of *Testing in OpenERP Web* is to set up the basic testing structure:

1. Creating a javascript file

   section diff file
   static/src/tests/timer.js

2. Containing a test section (and a few tests to make sure the tests are correctly run)

   section diff file
   static/src/tests/timer.js

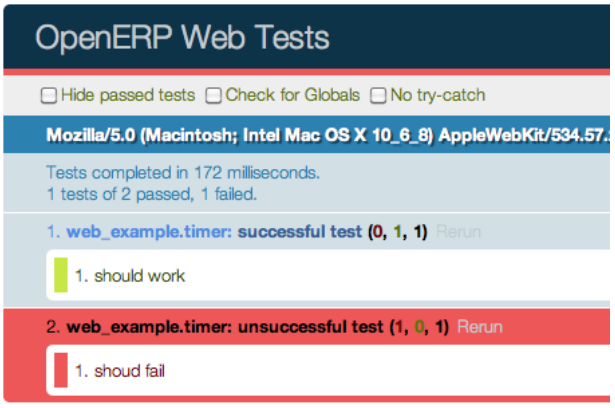
   ```
   openerp.testing.section('timer', function (test) {
       test('successful test', function () {
           ok(true, "should work");
       });
       test('unsuccessful test', function () {
           ok(false, "shoud fail");
       });
   });
   ```

3. Then declaring the test file in the module's manifest

   section diff file
   __openerp__.py

   ```
       'js': ['static/src/js/first_module.js'],
       'css': ['static/src/css/web_example.css'],
       'qweb': ['static/src/xml/web_example.xml'],
       'test': ['static/src/tests/timer.js'],
   }
   ```

4. And finally — after restarting OpenERP — navigating to the test runner at `/web/tests` and selecting your soon-to-be-tested module:



the testing result do indeed match the test.

The simplest tests to write are for synchronous pure functions. Synchronous means no RPC call or any other such thing (e.g. `setTimeout`), only direct data process means no side-effect: the function takes some input, manipulates it and yields an output.

In our widget, only `format_time` fits the bill: it takes a duration (in milliseconds) and returns an `hours:minutes:second` formatting of it. Let's test it:

section diff file
static/src/tests/timer.js

```
openerp.testing.section('timer', function (test) {
    test('format_time', function (instance) {
        var w = new instance.web_example.Action();

        strictEqual(
            w.format_time(0),
            '00:00:00');
        strictEqual(
            w.format_time(543),
            '00:00:00',
            "should round sub-second times down to zero");
        strictEqual(
            w.format_time(5340),
            '00:00:05',
            "should floor sub-second extents to the previous second");
        strictEqual(
            w.format_time(60000),
            '00:01:00');
        strictEqual(
            w.format_time(3600000),
            '01:00:00');
        strictEqual(
            w.format_time(86400000),
            '24:00:00');
        strictEqual(
            w.format_time(604800000),
            '168:00:00');

        strictEqual(
            w.format_time(22733958),
            '06:18:53');
        strictEqual(
            w.format_time(41676639),
            '11:34:36');
        strictEqual(
            w.format_time(57802094),
            '16:03:22');
        strictEqual(
            w.format_time(73451828),
            '20:24:11');
        strictEqual(
            w.format_time(84092336),
            '23:21:32');
    });
});
```

This series of simple tests passes with no issue. The next easy-ish test type is to test basic DOM alterations from provided input, such as (for our widget) updating th displaying a record to the records list: while it's not pure (it alters the DOM "in-place") it has well-delimited side-effects and these side-effects come solely from the

Because these methods alter the widget's DOM, the widget needs a DOM. Looking up *a widget's lifecycle*, the widget really only gets its DOM when adding it to the However a side-effect of this is to **`start()`** it, which for us means going to query the user's times.

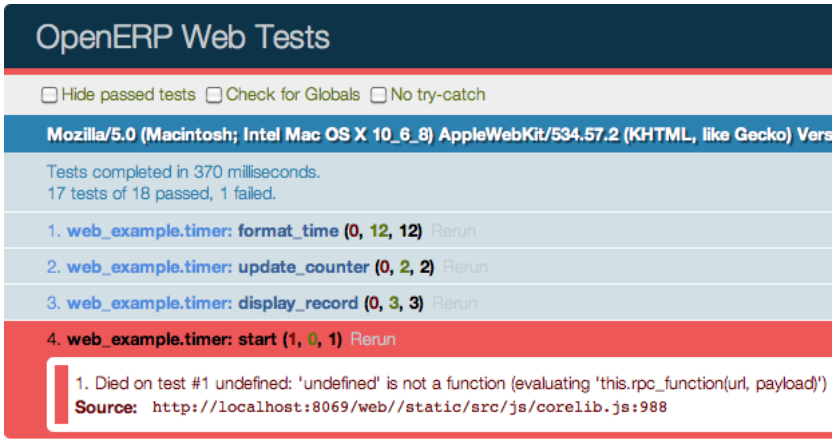
We don't have any records to get in our test, and we don't want to test the initialization yet! So let's cheat a bit: we can manually **`set a widget's DOM`**, let's create matching what each method expects then call the method:

section diff file
static/src/tests/timer.js

```
            w.format_time(84092336),
            '23:21:32');
    });
    test('update_counter', function (instance, $fixture) {
        var w = new instance.web_example.Action();
        // $fixture is a DOM tree whose content gets cleaned up before
        // each test, so we can add whatever we need to it
        $fixture.append('<div class="oe_web_example_timer">');
        // Then set it on the widget
        w.setElement($fixture);

        // Update the counter with a known value
        w.update_counter(22733958);
        // And check the DOM matches
        strictEqual($fixture.text(), '06:18:53');

        w.update_counter(73451828)
        strictEqual($fixture.text(), '20:24:11');
    });
    test('display_record', function (instance, $fixture) {
        var w = new instance.web_example.Action();
        $fixture.append('<ol class="oe_web_example_saved">')
        w.setElement($fixture);

        w.display_record({time: 41676639});
        w.display_record({time: 84092336});

        var $lis = $fixture.find('li');
        strictEqual($lis.length, 2, "should have printed 2 records");
        strictEqual($lis[0].textContent, '11:34:36');
        strictEqual($lis[1].textContent, '23:21:32');
    });
});
```

The next group of patches (in terms of setup/complexity) is RPC tests: testing components/methods which perform network calls (RPC requests). In our module, s
`watch_stop` are in that case: `start` fetches the user's recorded times and `watch_stop` creates a new record with the current watch.

By default, tests don't allow RPC requests and will generate an error when trying to perform one:



To allow them, the test case (or the test suite) has to explicitly opt into **`rpc support`** by adding the `rpc: 'mock'` option to the test case, and providing its own "rpc

section diff file
static/src/tests/timer.js

```
        strictEqual($lis[0].textContent, '11:34:36');
        strictEqual($lis[1].textContent, '23:21:32');
    });
    test('start', {templates: true, rpc: 'mock', asserts: 3}, function (instance, $fixture, mock) {
        // Rather odd-looking shortcut for search+read in a single RPC call
        mock('/web/dataset/search_read', function () {
            // ignore parameters, just return a pair of records.
            return {records: [
                {time: 22733958},
                {time: 84092336}
            ]};
        });

        var w = new instance.web_example.Action();
        return w.appendTo($fixture)
        .then(function () {
            var $lis = $fixture.find('li');
            strictEqual($lis.length, 2);
            strictEqual($lis[0].textContent, '06:18:53');
            strictEqual($lis[1].textContent, '23:21:32');
        });
    });
});
```

Note:

By defaut, tests cases don't load templates either. We had not needed to perform any template rendering before here, so we must now enable templates loading via `corresponding option`.

Our final test requires altering the module's code: asynchronous tests use *deferred* to know when a test ends and the other one can start (otherwise test content wil linearly and the assertions of a test will be executed during the next test or worse), but although `watch_stop` performs an asynchronous `create` operation it doesn' we can synchronize on. We simply need to return its result:

section diff file
static/src/js/first_module.js

```
                user_id: instance.session.uid,
                time: time,
            };
            return this.model.call('create', [record]).done(function () {
                self.display_record(record);
            });
        },
```

This makes no difference to the original code, but allows us to write our test:

section diff file
static/src/tests/timer.js

```
            strictEqual($lis[1].textContent, '23:21:32');
        });
    });
    test('watch_stop', {templates: true, rpc: 'mock', asserts: 3}, function (instance, $fix, mock) {
        var created = false;
        mock('web_example.stopwatch:create', function (args, kwargs) {
            created = true;
            // return a fake id (unused)
            return 42;
        });
        mock('/web/dataset/search_read', function () {
            return {records: []};
        });

        var w = new instance.web_example.Action();
        return w.appendTo($fix)
        .then(function () {
            // Virtual start point 5s before 'now'
            w._start = new Date() - 5000;
            return w.watch_stop();
        })
        .done(function () {
            ok(created, "should have called create()");
            strictEqual($fix.find('.oe_web_example_timer').text(),
                        '00:00:05',
                        "should have updated the timer");
            strictEqual($fix.find('li')[0].textContent,
                        '00:00:05',
                        "should have added the new time to the list");
        });
    });
});
```

[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 pa (e.g. events handling).