Javascript Cheatsheet

There are many ways to solve a problem in JavaScript, and in Odoo. However, the Odoo framework was designed to be extensible (this is a pretty big constraint), and some common problems have a nice standard solution. The standard solution has probably the advantage of being easy to understand for an odoo developers, and will probably keep working when Odoo is modified.

This document tries to explain the way one could solve some of these issues. Note that this is not a reference. This is just a random collection of recipes, or explanations on how to proceed in some cases.

First of all, remember that the first rule of customizing odoo with JS is: *try to do it in python*. This may seem strange, but the python framework is quite extensible, and many behaviours can be done simply with a touch of xml or python. This has usually a lower cost of maintenance than working with JS:

> the JS framework tends to change more, so JS code needs to be more frequently updated

> it is often more difficult to implement a customized behaviour if it needs to communicate with the server and properly integrate with the javascript framework. There are many small details taken care by the framework that customized code needs to replicate. For example, responsiveness, or updating the url, or displaying data without flickering.

> This document does not really explain any concepts. This is more a cookbook. For more details, please consult the javascript reference page (see **Javascript Reference (javascript_reference.html)**)

Creating a new field widget

This is probably a really common usecase: we want to display some information in a form view in a really specific (maybe business dependent) way. For example, assume that we want to change the text color depending on some business condition.

This can be done in three steps: creating a new widget, registering it in the field registry, then adding the widget to the field in the form view

creating a new widget:
> This can be done by extending a widget:

```
var FieldChar = require('web.basic_fields').FieldChar;

var CustomFieldChar = FieldChar.extend({
    _renderReadonly: function () {
        // implement some custom logic here
    },
});
```

registering it in the field registry:

The web client needs to know the mapping between a widget name and its actual class. This is done by a registry:

```
var fieldRegistry = require('web.field_registry');

fieldRegistry.add('my-custom-field', CustomFieldChar);
```

adding the widget in the form view

```
<field name="somefield" widget="my-custom-field"/>
```

Note that only the form, list and kanban views use this field widgets registry. These views are tightly integrated, because the list and kanban views can appear inside a form view).

Modifying an existing field widget

Another use case is that we want to modify an existing field widget. For example, the voip addon in odoo need to modify the FieldPhone widget to add the possibility to easily call the given number on voip. This is done by *including* the FieldPhone widget, so there is no need to change any existing form view.

Field Widgets (instances of (subclass of) AbstractField) are like every other widgets, so they can be monkey patched. This looks like this:

```
var basic_fields = require('web.basic_fields');
var Phone = basic_fields.FieldPhone;

Phone.include({
    events: _.extend({}, Phone.prototype.events, {
        'click': '_onClick',
    }),

    _onClick: function (e) {
        if (this.mode === 'readonly') {
            e.preventDefault();
            var phoneNumber = this.value;
            // call the number on voip...
        }
    },
});
```

Note that there is no need to add the widget to the registry, since it is already registered.

Modifying a main widget from the interface

Another common usecase is the need to customize some elements from the user interface. For example, adding a message in the home menu. The usual process in this case is again to *include* the widget. This is the only way to do it, since there are no registries for those widgets. This is usually done with code looking like this:

```
var HomeMenu = require('web_enterprise.HomeMenu');

HomeMenu.include({
    render: function () {
        this._super();
        // do something else here...
    },
});
```

Creating a new view (from scratch)

Creating a new view is a more advanced topic. This cheatsheet will only highlight the steps that will probably need to be done (in no particular order):

adding a new view type to the field `type` of `ir.ui.view`:

```
class View(models.Model):
    _inherit = 'ir.ui.view'

    type = fields.Selection(selection_add=[('map', "Map")])
```

adding the new view type to the field `view_mode` of `ir.actions.act_window.view`:

```
class ActWindowView(models.Model):
    _inherit = 'ir.actions.act_window.view'

    view_mode = fields.Selection(selection_add=[('map', "Map")])
```

creating the four main pieces which makes a view (in JavaScript):

we need a view (a subclass of `AbstractView`, this is the factory), a renderer (from `AbstractRenderer`), a controller (from `AbstractController`) and a model (from `AbstractModel`). I suggest starting by simply extending the superclasses:

```
    var AbstractController = require('web.AbstractController');
    var AbstractModel = require('web.AbstractModel');
    var AbstractRenderer = require('web.AbstractRenderer');
    var AbstractView = require('web.AbstractView');

    var MapController = AbstractController.extend({});
    var MapRenderer = AbstractRenderer.extend({});
    var MapModel = AbstractModel.extend({});

    var MapView = AbstractView.extend({
        config: {
            Model: MapModel,
            Controller: MapController,
            Renderer: MapRenderer,
        },
    });
```

adding the view to the registry:

As usual, the mapping between a view type and the actual class needs to be updated:

```
    var viewRegistry = require('web.view_registry');

    viewRegistry.add('map', MapView);
```

implementing the four main classes:

The `View` class needs to parse the `arch` field and setup the other three classes. The `Renderer` is in charge of representing the data in the user interface, the `Model` is supposed to talk to the server, to load data and process it. And the `Controller` is there to coordinate, to talk to the web client, …

creating some views in the database:

```
    <record id="customer_map_view" model="ir.ui.view">
        <field name="name">customer.map.view</field>
        <field name="model">res.partner</field>
        <field name="arch" type="xml">
            <map latitude="partner_latitude" longitude="partner_longitude">
                <field name="name"/>
            </map>
        </field>
    </record>
```

Customizing an existing view

Assume we need to create a custom version of a generic view. For example, a kanban view with some extra *ribbon-like* widget on top (to display some specific custom information). In that case, this can be done with 3 steps: extend the kanban view (which also probably mean

extending controllers/renderers and/or models), then registering the view in the view registry, and finally, using the view in the kanban arch (a specific example is the helpdesk dashboard).

extending a view:

Here is what it could look like:

```
var HelpdeskDashboardRenderer = KanbanRenderer.extend({
    ...
});


var HelpdeskDashboardModel = KanbanModel.extend({
    ...
});


var HelpdeskDashboardController = KanbanController.extend({
    ...
});


var HelpdeskDashboardView = KanbanView.extend({
    config: _.extend({}, KanbanView.prototype.config, {
        Model: HelpdeskDashboardModel,
        Renderer: HelpdeskDashboardRenderer,
        Controller: HelpdeskDashboardController,
    }),
});
```

adding it to the view registry:

as usual, we need to inform the web client of the mapping between the name of the views and the actual class.

```
var viewRegistry = require('web.view_registry');
viewRegistry.add('helpdesk_dashboard', HelpdeskDashboardView);
```

using it in an actual view:

we now need to inform the web client that a specific `ir.ui.view` needs to use our new class. Note that this is a web client specific concern. From the point of view of the server, we still have a kanban view. The proper way to do this is by using a special attribute `js_class` (which will be renamed someday into `widget`, because this is really not a good name) on the root node of the arch:

```
<record id="helpdesk_team_view_kanban" model="ir.ui.view" >
    ...
    <field name="arch" type="xml">
        <kanban js_class="helpdesk_dashboard">
            ...
        </kanban>
    </field>
</record>
```

> Note: you can change the way the view interprets the arch structure. However, from the server point of view, this is still a view of the same base type, subjected to the same rules (rng validation, for example). So, your views still need to have a valid arch field.

Promises and asynchronous code

For a very good and complete introduction to promises, please read this excellent article
[https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/async%20%26%20performance/ch3.md (https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/async%20%26%20performance/ch3.md)](https://github.com/getify/You-Dont-Know-JS/blob/1st-ed/async%20%26%20performance/ch3.md)

Creating new Promises

turn a constant into a promise

There are 2 static functions on Promise that create a resolved or rejected promise based
on a constant:

```
var p = Promise.resolve({blabla: '1'}); // creates a resolved promise
p.then(function (result) {
    console.log(result); // --> {blabla: '1'};
});


var p2 = Promise.reject({error: 'error message'}); // creates a rejected promise
p2.catch(function (reason) {
    console.log(reason); // --> {error: 'error message');
});
```

> Note that even if the promises are created already resolved or rejected, the `then` or `catch`
> handlers will still be called asynchronously.

based on an already asynchronous code

Suppose that in a function you must do a rpc, and when it is completed set the result on
this. The `this._rpc` is a function that returns a `Promise`.

```
function callRpc() {
    var self = this;
    return this._rpc(...).then(function (result) {
        self.myValueFromRpc = result;
    });
}
```

for callback based function

Suppose that you were using a function `this.close` that takes as parameter a callback
that is called when the closing is finished. Now suppose that you are doing that in a
method that must send a promise that is resolved when the closing is finished.

```
1  function waitForClose() {
2      var self = this;
3      return new Promise (function(resolve, reject) {
4          self.close(resolve);
5      });
6  }
```

line 2: we save the `this` into a variable so that in an inner function, we can access the scope of our component

line 3: we create and return a new promise. The constructor of a promise takes a function as parameter. This function itself has 2 parameters that we called here `resolve` and `reject`

> `resolve` is a function that, when called, puts the promise in the resolved state.
>
> `reject` is a function that, when called, puts the promise in the rejected state. We do not use reject here and it can be omitted.

line 4: we are calling the function close on our object. It takes a function as parameter (the callback) and it happens that resolve is already a function, so we can pass it directly. To be clearer, we could have written:

```
return new Promise (function (resolve) {
    self.close(function () {
        resolve();
    });
});
```

creating a promise generator (calling one promise after the other *in sequence* and waiting for the last one)

Suppose that you need to loop over an array, do an operation *in sequence* and resolve a promise when the last operation is done.

```
function doStuffOnArray(arr) {
    var done = Promise.resolve();
    arr.forEach(function (item) {
        done = done.then(function () {
            return item.doSomethingAsynchronous();
        });
    });
    return done;
}
```

This way, the promise you return is effectively the last promise.

creating a promise, then resolving it outside the scope of its definition (anti-pattern)

> we do not recommend using this, but sometimes it is useful. Think carefully for alternatives first...

```
...
var resolver, rejecter;
var prom = new Promise(function (resolve, reject){
    resolver = resolve;
    rejecter = reject;
});
...

resolver("done"); // will resolve the promise prom with the result "done"
rejecter("error"); // will reject the promise prom with the reason "error"
```

Waiting for Promises

waiting for a number of Promises

if you have multiple promises that all need to be waited, you can convert them into a single promise that will be resolved when all the promises are resolved using Promise.all(arrayOfPromises).

```
var prom1 = doSomethingThatReturnsAPromise();
var prom2 = Promise.resolve(true);
var constant = true;

var all = Promise.all([prom1, prom2, constant]); // all is a promise
// results is an array, the individual results correspond to the index of their
// promise as called in Promise.all()
all.then(function (results) {
    var prom1Result = results[0];
    var prom2Result = results[1];
    var constantResult = results[2];
});
return all;
```

waiting for a part of a promise chain, but not another part

If you have an asynchronous process that you want to wait to do something, but you also want to return to the caller before that something is done.

```
function returnAsSoonAsAsyncProcessIsDone() {
    var prom = AsyncProcess();
    prom.then(function (resultOfAsyncProcess) {
            return doSomething();
    });
    /* returns prom which will only wait for AsyncProcess(),
       and when it will be resolved, the result will be the one of AsyncProcess */
    return prom;
}
```

Error handling

in general in promises

The general idea is that a promise should not be rejected for control flow, but should only be rejected for errors. When that is the case, you would have multiple resolutions of your promise with, for instance status codes that you would have to check in the `then` handlers and a single `catch` handler at the end of the promise chain.

```
function a() {
    x.y();  // <-- this is an error: x is undefined
    return Promise.resolve(1);
}
function b() {
    return Promise.reject(2);
}

a().catch(console.log);            // will log the error in a
a().then(b).catch(console.log);    // will log the error in a, the then is not execu
b().catch(console.log);            // will log the rejected reason of b (2)
Promise.resolve(1)
        .then(b)                   // the then is executed, it executes b
        .then(...)                 // this then is not executed
        .catch(console.log);       // will log the rejected reason of b (2)
```

in Odoo specifically

In Odoo, it happens that we use promise rejection for control flow, like in mutexes and other concurrency primitives defined in module `web.concurrency` We also want to execute the catch for *business* reasons, but not when there is a coding error in the definition of the promise or of the handlers. For this, we have introduced the concept of `guardedCatch` . It is called like `catch` but not when the rejected reason is an error

```
function blabla() {
    if (someCondition) {
        return Promise.reject("someCondition is truthy");
    }
    return Promise.resolve();
}

// ...

var promise = blabla();
promise.then(function (result) { console.log("everything went fine"); })
// this will be called if blabla returns a rejected promise, but not if it has an e
promise.guardedCatch(function (reason) { console.log(reason); });

// ...

var anotherPromise =
        blabla().then(function () { console.log("everything went fine"); })
                // this will be called if blabla returns a rejected promise,
                // but not if it has an error
                .guardedCatch(console.log);
```

```
var promiseWithError = Promise.resolve().then(function () {
    x.y();  // <-- this is an error: x is undefined
});
promiseWithError.guardedCatch(function (reason) {console.log(reason);}); // will n
promiseWithError.catch(function (reason) {console.log(reason);}); // will be calle
```

Testing asynchronous code

using promises in tests

In the tests code, we support the latest version of Javascript, including primitives like `async` and `await` . This makes using and waiting for promises very easy. Most helper methods also return a promise (either by being marked `async` or by returning a promise directly.

```
var testUtils = require('web.test_utils');
QUnit.test("My test", async function (assert) {
    // making the function async has 2 advantages:
    // 1) it always returns a promise so you don't need to define `var done = asser
    // 2) it allows you to use the `await`
    assert.expect(1);

    var form = await testUtils.createView({ ... });
    await testUtils.form.clickEdit(form);
    await testUtils.form.click('jquery selector');
    assert.containsOnce('jquery selector');
    form.destroy();
});

QUnit.test("My test — no async — no done", function (assert) {
    // this function is not async, but it returns a promise.
    // QUnit will wait for for this promise to be resolved.
    assert.expect(1);

    return testUtils.createView({ ... }).then(function (form) {
        return testUtils.form.clickEdit(form).then(function () {
            return testUtils.form.click('jquery selector').then(function () {
                assert.containsOnce('jquery selector');
                form.destroy();
            });
        });
    });
});


QUnit.test("My test — no async", function (assert) {
    // this function is not async and does not return a promise.
    // we have to use the done function to signal QUnit that the test is async and
    assert.expect(1);
    var done = assert.async();

    testUtils.createView({ ... }).then(function (form) {
        testUtils.form.clickEdit(form).then(function () {
            testUtils.form.click('jquery selector').then(function () {
            assert.containsOnce('jquery selector');
            form.destroy();
            done();
            });
        });
    });
});
```

as you can see, the nicer form is to use `async/await` as it is clearer and shorter to write.