Javascript Reference

This document presents the Odoo Javascript framework. This framework is not a large application in term of lines of code, but it is quite generic, because it is basically a machine to turn a declarative interface description into a live application, able to interact with every model and records in the database. It is even possible to use the web client to modify the interface of the web client.

Overview

The Javascript framework is designed to work with three main use cases:

the *web client*: this is the private web application, where one can view and edit business data. This is a single page application (the page is never reloaded, only the new data is fetched from the server whenever it is needed)

the *website*: this is the public part of Odoo. It allows an unidentified user to browse some content, to shop or to perform many actions, as a client. This is a classical website: various routes with controllers and some javascript to make it work.

the *point of sale*: this is the interface for the point of sale. It is a specialized single page application.

Some javascript code is common to these three use cases, and is bundled together (see below in the assets section). This document will focus mostly on the web client design.

Web client

Single Page Application

In short, the *webClient*, instance of *WebClient* is the root component of the whole user interface. Its responsibility is to orchestrate all various subcomponents, and to provide services, such as rpcs, local storage and more.
In runtime, the web client is a single page application. It does not need to request a full page from the server each time the user perform an action. Instead, it only requests what it needs, and then replaces/updates the view. Also, it manages the url: it is kept in sync with the web client state.
It means that while a user is working on Odoo, the web client class (and the action manager) actually creates and destroys many sub components. The state is highly dynamic, and each widget could be destroyed at any time.

Overview of web client JS code

Here, we give a very quick overview on the web client code, in the *web/static/src/js* addon. Note that it is deliberately not exhaustive. We only cover the most important files/folders.

boot.js: this is the file that defines the module system. It needs to be loaded first.

core/: this is a collection of lower level building blocks. Notably, it contains the class system, the widget system, concurrency utilities, and many other class/functions.

chrome/: in this folder, we have most large widgets which make up most of the user interface.

chrome/abstract_web_client.js and *chrome/web_client.js*: together, these files define the WebClient widget, which is the root widget for the web client.

chrome/action_manager.js: this is the code that will convert an action into a widget (for example a kanban or a form view)

chrome/search_X.js all these files define the search view (it is not a view in the point of view of the web client, only from the server point of view)

fields: all main view field widgets are defined here

views: this is where the views are located

Assets Management

Managing assets in Odoo is not as straightforward as it is in some other apps. One of the reason is that we have a variety of situations where some, but not all the assets are required. For example, the needs of the web client, the point of sale, the website or even the mobile application are different. Also, some assets may be large, but are seldom needed. In that case, we sometimes want them to be loaded lazily.
The main idea is that we define a set of *bundles* in xml. A bundle is here defined as a collection of files (javascript, css, scss). In Odoo, the most important bundles are defined in the file *addons/web/views/webclient_templates.xml*. It looks like this:

```
<template id="web.assets_common" name="Common Assets (used in backend interface and website)">
    <link rel="stylesheet" type="text/css" href="/web/static/lib/jquery.ui/jquery-ui.css"/>
    ...
    <script type="text/javascript" src="/web/static/src/js/boot.js"></script>
    ...
</template>
```

The files in a bundle can then be inserted into a template by using the *t-call-assets* directive:

```
<t t-call-assets="web.assets_common" t-js="false"/>
<t t-call-assets="web.assets_common" t-css="false"/>
```

Here is what happens when a template is rendered by the server with these directives:

all the *scss* files described in the bundle are compiled into css files. A file named *file.scss* will be compiled in a file named *file.scss.css*.

if we are in *debug=assets* mode

the *t-call-assets* directive with the *t-js* attribute set to false will be replaced by a list of stylesheet tags pointing to the css files

the *t-call-assets* directive with the *t-css* attribute set to false will be replaced by a list of script tags pointing to the js files

if we are not in *debug=assets* mode

the css files will be concatenated and minified, then a stylesheet tag is generated

the js files are concatenated and minified, then a script tag is generated

Note that the assets files are cached, so in theory, a browser should only load them once.

Main bundles

When the Odoo server is started, it checks the timestamp of each file in a bundle, and if necessary, will create/recreate the corresponding bundles.
Here are some important bundles that most developers will need to know:

web.assets_common: this bundle contains most assets which are common to the web client, the website, and also the point of sale. This is supposed to contain lower level building blocks for the odoo framework. Note that it contains the *boot.js* file, which defines the odoo module system.

web.assets_backend: this bundle contains the code specific to the web client (notably the web client/action manager/views)

web.assets_frontend: this bundle is about all that is specific to the public website: ecommerce, forum, blog, event management, ...

Adding files in an asset bundle

The proper way to add a file located in *addons/web* to a bundle is simple: it is just enough to add a *script* or a *stylesheet* tag to the bundle in the file *webclient_templates.xml*. But when we work in a different addon, we need to add a file from that addon. In that case, it should be done in three steps:

1. add a *assets.xml* file in the *views/* folder
2. add the string 'views/assets.xml' in the 'data' key in the manifest file
3. create an inherited view of the desired bundle, and add the file(s) with an xpath expression. For example,

```
<template id="assets_backend" name="helpdesk assets" inherit_id="web.assets_backend">
    <xpath expr="//script[last()]" position="after">
        <link rel="stylesheet" type="text/scss" href="/helpdesk/static/src/scss/helpdesk.scss"/>
        <script type="text/javascript" src="/helpdesk/static/src/js/helpdesk_dashboard.js"></script>
    </xpath>
</template>
```

> Note that the files in a bundle are all loaded immediately when the user loads the odoo web client. This means that the files are transferred through the network everytime (except when the browser cache is active). In some cases, it may be better to lazyload some assets. For example, if a widget requires a large library, and that widget is not a core part of the experience, then it may be a good idea to only load the library when the widget is actually created. The widget class has actually builtin support just for this use case. (see section **QWeb Template Engine**)

What to do if a file is not loaded/updated

There are many different reasons why a file may not be properly loaded. Here are a few things you can try to solve the issue:

once the server is started, it does not know if an asset file has been modified. So, you can simply restart the server to regenerate the assets.

check the console (in the dev tools, usually opened with F12) to make sure there are no obvious errors

try to add a console.log at the beginning of your file (before any module definition), so you can see if a file has been loaded or not

in the user interface, in debug mode (INSERT LINK HERE TO DEBUG MODE), there is an option to force the server to update its assets files.

use the *debug=assets* mode. This will actually bypass the asset bundles (note that it does not actually solve the issue. The server still uses outdated bundles)

finally, the most convenient way to do it, for a developer, is to start the server with the *–dev=all* option. This activates the file watcher options, which will automatically invalidate assets when necessary. Note that it does not work very well if the OS is Windows.

remember to refresh your page!

or maybe to save your code file...

> Once an asset file has been recreated, you need to refresh the page, to reload the proper files (if that does not work, the files may be cached).

Javascript Module System

Once we are able to load our javascript files into the browser, we need to make sure they are loaded in the correct order. In order to do that, Odoo has defined a small module system (located in the file *addons/web/static/src/js/boot.js*, which needs to be loaded first).
The Odoo module system, inspired by AMD, works by defining the function *define* on the global odoo object. We then define each javascript module by calling that function. In the Odoo framework, a module is a piece of code that will be executed as soon as possible. It has a name and potentially some dependencies. When its dependencies are loaded, a module will then be loaded as well. The value of the module is then the return value of the function defining the module.
As an example, it may look like this:

```
// in file a.js
odoo.define('module.A', function (require) {
    "use strict";

    var A = ...;

    return A;
});

// in file b.js
odoo.define('module.B', function (require) {
    "use strict";

    var A = require('module.A');

    var B = ...; // something that involves A

    return B;
});
```

An alternative way to define a module is to give explicitly a list of dependencies in the second argument.

```
odoo.define('module.Something', ['module.A', 'module.B'], function (require) {
    "use strict";

    var A = require('module.A');
    var B = require('module.B');

    // some code
});
```

If some dependencies are missing/non ready, then the module will simply not be loaded. There will be a warning in the console after a few seconds.
Note that circular dependencies are not supported. It makes sense, but it means that one needs to be careful.

Defining a module

The *odoo.define* method is given three arguments:

moduleName: the name of the javascript module. It should be a unique string. The convention is to have the name of the odoo addon followed by a specific description. For example, 'web.Widget' describes a module defined in the *web* addon, which exports a *Widget* class (because the first letter is capitalized)
If the name is not unique, an exception will be thrown and displayed in the console.

dependencies: the second argument is optional. If given, it should be a list of strings, each corresponding to a javascript module. This describes the dependencies that are required to be loaded before the module is executed. If the dependencies are not explicitly given here, then the module system will extract them from the function by calling toString on it, then using a regexp to find all *require* statements.

```
odoo.define('module.Something', ['web.ajax'], function (require) {
  "use strict";

  var ajax = require('web.ajax');

  // some code here
  return something;
});
```

finally, the last argument is a function which defines the module. Its return value is the value of the module, which may be passed to other modules requiring it. Note that there is a small exception for asynchronous modules, see the next section.

If an error happens, it will be logged (in debug mode) in the console:

`Missing dependencies` : These modules do not appear in the page. It is possible that the JavaScript file is not in the page or that the module name is wrong

`Failed modules` : A javascript error is detected

`Rejected modules` : The module returns a rejected Promise. It (and its dependent modules) is not loaded.

`Rejected linked modules` : Modules who depend on a rejected module

`Non loaded modules` : Modules who depend on a missing or a failed module

Asynchronous modules

It can happen that a module needs to perform some work before it is ready. For example, it could do a rpc to load some data. In that case, the module can simply return a promise. In that case, the module system will simply wait for the promise to complete before registering the module.

```
odoo.define('module.Something', function (require) {
    "use strict";

    var ajax = require('web.ajax');

    return ajax.rpc(...).then(function (result) {
        // some code here
        return something;
    });
});
```

Best practices

remember the convention for a module name: *addon name* suffixed with *module name*.

declare all your dependencies at the top of the module. Also, they should be sorted alphabetically by module name. This makes it easier to understand your module.

declare all exported values at the end

try to avoid exporting too many things from one module. It is usually better to simply export one thing in one (small/smallish) module.

asynchronous modules can be used to simplify some use cases. For example, the *web.dom_ready* module returns a promise which will be resolved when the dom is actually ready. So, another module that needs the DOM could simply have a `require('web.dom_ready')` statement somewhere, and the code will only be executed when the DOM is ready.

try to avoid defining more than one module in one file. It may be convenient in the short term, but this is actually harder to maintain.

Class System

Odoo was developed before ECMAScript 6 classes were available. In Ecmascript 5, the standard way to define a class is to define a function and to add methods on its prototype object. This is fine, but it is slightly complex when we want to use inheritance, mixins.
For these reasons, Odoo decided to use its own class system, inspired by John Resig. The base Class is located in *web.Class*, in the file *class.js*.

Creating a subclass

Let us discuss how classes are created. The main mechanism is to use the *extend* method (this is more or less the equivalent of *extend* in ES6 classes).

```
var Class = require('web.Class');

var Animal = Class.extend({
    init: function () {
        this.x = 0;
        this.hunger = 0;
    },
    move: function () {
        this.x = this.x + 1;
        this.hunger = this.hunger + 1;
    },
    eat: function () {
        this.hunger = 0;
    },
});
```

In this example, the *init* function is the constructor. It will be called when an instance is created. Making an instance is done by using the *new* keyword.

Inheritance

It is convenient to be able to inherit an existing class. This is simply done by using the *extend* method on the superclass. When a method is called, the framework will secretly rebind a special method: *_super* to the currently called method. This allows us to use *this._super* whenever we need to call a parent method.

```
var Animal = require('web.Animal');

var Dog = Animal.extend({
    move: function () {
        this.bark();
        this._super.apply(this, arguments);
    },
    bark: function () {
        console.log('woof');
    },
});

var dog = new Dog();
dog.move()
```

Mixins

The odoo Class system does not support multiple inheritance, but for those cases when we need to share some behaviour, we have a mixin system: the *extend* method can actually take an arbitrary number of arguments, and will combine all of them in the new class.

```
var Animal = require('web.Animal');
var DanceMixin = {
    dance: function () {
        console.log('dancing...');
    },
};

var Hamster = Animal.extend(DanceMixin, {
    sleep: function () {
        console.log('sleeping');
    },
});
```

In this example, the *Hamster* class is a subclass of Animal, but it also mix the DanceMixin in.

Patching an existing class

It is not common, but we sometimes need to modify another class *in place*. The goal is to have a mechanism to change a class and all future/present instances. This is done by using the *include* method:

```
var Hamster = require('web.Hamster');

Hamster.include({
    sleep: function () {
        this._super.apply(this, arguments);
        console.log('zzzz');
    },
});
```

This is obviously a dangerous operation and should be done with care. But with the way Odoo is structured, it is sometimes necessary in one addon to modify the behavior of a widget/class defined in another addon. Note that it will modify all instances of the class, even if they have already been created.

Widgets

The *Widget* class is really an important building block of the user interface. Pretty much everything in the user interface is under the control of a widget. The Widget class is defined in the module *web.Widget*, in *widget.js*.
In short, the features provided by the Widget class include:

- parent/child relationships between widgets (*PropertiesMixin*)
- extensive lifecycle management with safety features (e.g. automatically destroying children widgets during the destruction of a parent)
- automatic rendering with qweb (qweb.html#reference-qweb)
- various utility functions to help interacting with the outside environment.

Here is an example of a basic counter widget:

```
var Widget = require('web.Widget');

var Counter = Widget.extend({
    template: 'some.template',
    events: {
        'click button': '_onClick',
    },
    init: function (parent, value) {
        this._super(parent);
        this.count = value;
    },
    _onClick: function () {
        this.count++;
        this.$('.val').text(this.count);
    },
});
```

For this example, assume that the template *some.template* (and is properly loaded: the template is in a file, which is properly defined in the *qweb* key in the module manifest) is given by:

```
<div t-name="some.template">
    <span class="val"><t t-esc="widget.count"/></span>
    <button>Increment</button>
</div>
```

This example widget can be used in the following manner:

```
// Create the instance
var counter = new Counter(this, 4);
// Render and insert into DOM
counter.appendTo(".some-div");
```

This example illustrates a few of the features of the *Widget* class, including the event system, the template system, the constructor with the initial *parent* argument.

Widget Lifecycle

Like many component systems, the widget class has a well defined lifecycle. The usual lifecycle is the following: *init* is called, then *willStart*, then the rendering takes place, then *start* and finally *destroy*.

`Widget.init(parent)`

this is the constructor. The init method is supposed to initialize the base state of the widget. It is synchronous and can be overridden to take more parameters from the widget's creator/parent

Arguments:

`parent` (**`Widget()`**) – the new widget's parent, used to handle automatic destruction and event propagation. Can be **`null`** for the widget to have no parent.

`Widget.willStart()`

this method will be called once by the framework when a widget is created and in the process of being appended to the DOM. The *willStart* method is a hook that should return a promise. The JS framework will wait for this promise to complete before moving on to the rendering step. Note that at this point, the widget does not have a DOM root element. The *willStart* hook is mostly useful to perform some asynchronous work, such as fetching data from the server

`[Rendering]()`

This step is automatically done by the framework. What happens is that the framework checks if a template key is defined on the widget. If that is the case, then it will render that template with the *widget* key bound to the widget in the rendering context (see the example above: we use *widget.count* in the QWeb template to read the value from the widget). If no template is defined, we read the *tagName* key and create a corresponding DOM element. When the rendering is done, we set the result as the $el property of the widget. After this, we automatically bind all events in the events and custom_events keys.

`Widget.start()`

when the rendering is complete, the framework will automatically call the *start* method. This is useful to perform some specialized post-rendering work. For example, setting up a library.

Must return a promise to indicate when its work is done.

Returns:

promise

`Widget.destroy()`

This is always the final step in the life of a widget. When a widget is destroyed, we basically perform all necessary cleanup operations: removing the widget from the component tree, unbinding all events, ...

Automatically called when the widget's parent is destroyed, must be called explicitly if the widget has no parent or if it is removed but its parent remains.

Note that the willStart and start method are not necessarily called. A widget can be created (the *init* method will be called) and then destroyed (*destroy* method) without ever having been appended to the DOM. If that is the case, the willStart and start will not even be called.

Widget API

`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:

`Widget.id`

Used to generate an `id` attribute on the generated DOM root. Note that this is rarely needed, and is probably not a good idea if a widget can be used more than once.

`Widget.className`

Used to generate a `class` attribute on the generated DOM root. Note that it can actually contain more than one css class: *'some-class other-class'*

`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.

`Widget.el`

raw DOM element set as root to the widget (only available after the start lifecycle method)

Widget.$el

jQuery wrapper around **el**. (only available after the start lifecycle method)

Widget.template

Should be set to the name of a QWeb template (qweb.html#reference-qweb). 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.

Widget.xmlDependencies

List of paths to xml files that need to be loaded before the widget can be rendered. This will not induce loading anything that has already been loaded. This is useful when you want to load your templates lazily, or if you want to share a widget between the website and the web client interface.

```
var EditorMenuBar = Widget.extend({
    xmlDependencies: ['/web_editor/static/src/xml/editor.xml'],
    ...
```

Widget.events

Events are a mapping of an event selector (an event name and an optional CSS selector separated by a space) to a callback. The callback can be the name of a widget's method or a function object. 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. If the selector is left out (only an event name is specified), the event will be set directly on the widget's DOM root.

Note: the use of an inline function is discouraged, and will probably be removed sometimes in the future.

Widget.custom_events

this is almost the same as the *events* attribute, but the keys are arbitrary strings. They represent business events triggered by some sub widgets. When an event is triggered, it will 'bubble up' the widget tree (see the section on component communication for more details).

Widget.isDestroyed()

Returns:

true if the widget is being or has been destroyed, **false** otherwise

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

this helper method is similar to **Backbone.View.$**

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

Inserting a widget in the DOM

Widget.appendTo(*element*)

Renders the widget and inserts it as the last child of the target, uses .appendTo() (https://api.jquery.com/appendTo/)

Widget.prependTo(*element*)

Renders the widget and inserts it as the first child of the target, uses .prependTo() (https://api.jquery.com/prependTo/)

Widget.insertAfter(*element*)

Renders the widget and inserts it as the preceding sibling of the target, uses .insertAfter() (https://api.jquery.com/insertAfter/)

Widget.insertBefore(*element*)

Renders the widget and inserts it as the following sibling of the target, uses .insertBefore() (https://api.jquery.com/insertBefore/)

All of these methods accept whatever the corresponding jQuery method accepts (CSS selectors, DOM nodes or jQuery objects). They all return a promise and are charged with three tasks:

rendering the widget's root element via **renderElement()**

inserting the widget's root element in the DOM using whichever jQuery method they match

starting the widget, and returning the result of starting it

Widget Guidelines

Identifiers (**id** attribute) should be avoided. In generic applications and modules, **id** limits the re-usability of components and tends to make code more brittle. Most of the time, they can be replaced with nothing, classes or keeping a reference to a DOM node or jQuery element.

If an **id** is absolutely necessary (because a third-party library requires one), the id should be partially generated using **_.uniqueId()** e.g.:

```
this.id = _.uniqueId('my-widget-');
```

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).

Global selectors should be avoided. Because a component may be used several times in a single page (an example in Odoo is dashboards), queries should be restricted to a given component's scope. Unfiltered selections such as `$(selector)` or `document.querySelectorAll(selector)` will generally lead to unintended or incorrect behavior. Odoo Web's `Widget()` has an attribute providing its DOM root (`$el`), and a shortcut to select nodes directly (`$()`).

More generally, never assume your components own or controls anything beyond its own personal `$el` (so, avoid using a reference to the parent widget)

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 life cycle.

Make sure to wait for start to be finished before using $el e.g.:

```
var Widget = require('web.Widget');

var AlmostCorrectWidget = Widget.extend({
    start: function () {
        this.$el.hasClass(....) // in theory, $el is already set, but you don't know what the parent will
        return this._super.apply(arguments);
    },
});

var IncorrectWidget = Widget.extend({
    start: function () {
        this._super.apply(arguments); // the parent promise is lost, nobody will wait for the start of th
        this.$el.hasClass(....)
    },
});

var CorrectWidget = Widget.extend({
    start: function () {
        var self = this;
        return this._super.apply(arguments).then(function() {
            self.$el.hasClass(....) // this works, no promise is lost and the code executes in a controll
        });
    },
});
```

QWeb Template Engine

The web client uses the QWeb (qweb.html) template engine to render widgets (unless they override the *renderElement* method to do something else). The Qweb JS template engine is based on XML, and is mostly compatible with the python implementation.

Now, let us explain how the templates are loaded. Whenever the web client starts, a rpc is made to the */web/webclient/qweb* route. The server will then return a list of all templates defined in data files for each installed modules. The correct files are listed in the *qweb* entry in each module manifest.

The web client will wait for that list of template to be loaded, before starting its first widget.

This mechanism works quite well for our needs, but sometimes, we want to lazy load a template. For example, imagine that we have a widget which is rarely used. In that case, maybe we prefer to not load its template in the main file, in order to make the web client slightly lighter. In that case, we can use the *xmlDependencies* key of the Widget:

```
var Widget = require('web.Widget');

var Counter = Widget.extend({
    template: 'some.template',
    xmlDependencies: ['/myaddon/path/to/my/file.xml'],

    ...

});
```

With this, the *Counter* widget will load the xmlDependencies files in its *willStart* method, so the template will be ready when the rendering is performed.

Event system

There are currently two event systems supported by Odoo: a simple system which allows adding listeners and triggering events, and a more complete system that also makes events 'bubble up'.
Both of these event systems are implemented in the *EventDispatcherMixin*, in the file *mixins.js*. This mixin is included in the *Widget* class.

Base Event system

This event system was historically the first. It implements a simple bus pattern. We have 4 main methods:

on: this is used to register a listener on an event.

off: useful to remove events listener.

once: this is used to register a listener that will only be called once.

trigger: trigger an event. This will cause each listeners to be called.

Here is an example on how this event system could be used:

```
var Widget = require('web.Widget');
var Counter = require('myModule.Counter');

var MyWidget = Widget.extend({
    start: function () {
        this.counter = new Counter(this);
        this.counter.on('valuechange', this, this._onValueChange);
        var def = this.counter.appendTo(this.$el);
        return Promise.all([def, this._super.apply(this, arguments)]);
    },
    _onValueChange: function (val) {
        // do something with val
    },
});

// in Counter widget, we need to call the trigger method:

... this.trigger('valuechange', someValue);
```

> **⚠ Warning**
> the use of this event system is discouraged, we plan to replace each *trigger* method by the *trigger_up* method from the extended event system

Extended Event System

The custom event widgets is a more advanced system, which mimic the DOM events API. Whenever an event is triggered, it will 'bubble up' the component tree, until it reaches the root widget, or is stopped.

trigger_up: this is the method that will create a small *OdooEvent* and dispatch it in the component tree. Note that it will start with the component that triggered the event

custom_events: this is the equivalent of the *event* dictionary, but for odoo events.

The OdooEvent class is very simple. It has three public attributes: *target* (the widget that triggered the event), *name* (the event name) and *data* (the payload). It also has 2 methods: *stopPropagation* and *is_stopped*.

The previous example can be updated to use the custom event system:

```
var Widget = require('web.Widget');
var Counter = require('myModule.Counter');

var MyWidget = Widget.extend({
    custom_events: {
        valuechange: '_onValueChange'
    },
    start: function () {
        this.counter = new Counter(this);
        var def = this.counter.appendTo(this.$el);
        return Promise.all([def, this._super.apply(this, arguments)]);
    },
    _onValueChange: function(event) {
        // do something with event.data.val
    },
});

// in Counter widget, we need to call the trigger_up method:

... this.trigger_up('valuechange', {value: someValue});
```

Registries

A common need in the Odoo ecosystem is to extend/change the behaviour of the base system from the outside (by installing an application, i.e. a different module). For example, one may need to add a new widget type in some views. In that case, and many others, the usual process is to create the desired component, then add it to a registry (registering step), to make the rest of the web client aware of its existence.

There are a few registries available in the system:

field registry (exported by `web.field_registry`)

The field registry contains all field widgets known to the web client. Whenever a view (typically form or list/kanban) needs a field widget, this is where it will look. A typical use case look like this:

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

var FieldPad = ...;

fieldRegistry.add('pad', FieldPad);
```

Note that each value should be a subclass of *AbstractField*

view registry

This registry contains all JS views known to the web client (and in particular, the view manager). Each value of this registry should be a subclass of *AbstractView*.

action registry

We keep track of all client actions in this registry. This is where the action manager looks up whenever it needs to create a client action. In version 11, each value should simply be a subclass of *Widget*. However, in version 12, the values are required to be *AbstractAction*.

Communication between widgets

There are many ways to communicate between components.

From a parent to its child

This is a simple case. The parent widget can simply call a method on its child:

```
this.someWidget.update(someInfo);
```

From a widget to its parent/some ancestor

In this case, the widget's job is simply to notify its environment that something happened. Since we do not want the widget to have a reference to its parent (this would couple the widget with its parent's implementation), the best way to proceed is usually to trigger an event, which will bubble up the component tree, by using the `trigger_up` method:

```
this.trigger_up('open_record', { record: record, id: id});
```

This event will be triggered on the widget, then will bubble up and be eventually caught by some upstream widget:

```
var SomeAncestor = Widget.extend({
    custom_events: {
        'open_record': '_onOpenRecord',
    },
    _onOpenRecord: function (event) {
        var record = event.data.record;
        var id = event.data.id;
        // do something with the event.
    },
});
```

Cross component

Cross component communication can be achieved by using a bus. This is not the preferred form of communication, because it has the disadvantage of making the code harder to maintain. However, it has the advantage of decoupling the components. In that case, this is simply done by triggering and listening to events on a bus. For example:

```
// in WidgetA
var core = require('web.core');

var WidgetA = Widget.extend({
    ...
    start: function () {
        core.bus.on('barcode_scanned', this, this._onBarcodeScanned);
    },
});

// in WidgetB
var WidgetB = Widget.extend({
    ...
    someFunction: function (barcode) {
        core.bus.trigger('barcode_scanned', barcode);
    },
});
```

```
In this example, we use the bus exported by *web.core*, but this is not
required. A bus could be created for a specific purpose.
```

Services

In version 11.0, we introduced the notion of *service*. The main idea is to give to sub components a controlled way to access their environment, in a way that allow the framework enough control, and which is testable.
The service system is organized around three ideas: services, service providers and widgets. The way it works is that widgets trigger (with *trigger_up*) events, these events bubble up to a service provider, which will ask a service to perform a task, then maybe return an answer.

Service

A service is an instance of the *AbstractService* class. It basically only has a name and a few methods. Its job is to perform some work, typically something depending on the environment.
For example, we have the *ajax* service (job is to perform a rpc), the *localStorage* (interact with the browser local storage) and many others.
Here is a simplified example on how the ajax service is implemented:

```
var AbstractService = require('web.AbstractService');

var AjaxService = AbstractService.extend({
    name: 'ajax',
    rpc: function (...) {
        return ...;
    },
});
```

This service is named 'ajax' and define one method, *rpc*.

Service Provider

For services to work, it is necessary that we have a service provider ready to dispatch the custom events. In the *backend* (web client), this is done by the main web client instance. Note that the code for the service provider comes from the *ServiceProviderMixin*.

Widget

The widget is the part that requests a service. In order to do that, it simply triggers an event *call_service* (typically by using the helper function *call*). This event will bubble up and communicate the intent to the rest of the system.
In practice, some functions are so frequently called that we have some helpers functions to make them easier to use. For example, the *_rpc* method is a helper that helps making a rpc.

```
var SomeWidget = Widget.extend({
    _getActivityModelViewID: function (model) {
        return this._rpc({
            model: model,
            method: 'get_activity_view_id'
        });
    },
});
```

> **⚠ Warning**
> If a widget is destroyed, it will be detached from the main component tree and will not have a parent. In that case, the events will not bubble up, which means that the work will not be done. This is usually exactly what we want from a destroyed widget.

RPCs

The rpc functionality is supplied by the ajax service. But most people will probably only interact with the *_rpc* helpers.

There are typically two usecases when working on Odoo: one may need to call a method on a (python) model (this goes through a controller *call_kw*), or one may need to directly call a controller (available on some route).

Calling a method on a python model:

```
return this._rpc({
    model: 'some.model',
    method: 'some_method',
    args: [some, args],
});
```

Directly calling a controller:

```
return this._rpc({
    route: '/some/route/',
    params: { some: kwargs},
});
```

Notifications

The Odoo framework has a standard way to communicate various information to the user: notifications, which are displayed on the top right of the user interface.
There are two types of notifications:

notification: useful to display some feedback. For example, whenever a user unsubscribed to a channel.

warning: useful to display some important/urgent information. Typically most kind of (recoverable) errors in the system.

Also, notifications can be used to ask a question to the user without disturbing its workflow. Imagine a phone call received through VOIP: a sticky notification could be displayed with two buttons *Accept* and *Decline*.

Notification system

The notification system in Odoo is designed with the following components:

a *Notification* widget: this is a simple widget that is meant to be created and displayed with the desired information

a *NotificationService*: a service whose responsibility is to create and destroy notifications whenever a request is done (with a custom_event). Note that the web client is a service provider.

a client action *display_notification*: this allows to trigger the display of a notification from python (e.g. in the method called when the user clicked on a button of type object).

two helper functions in *ServiceMixin*: *do_notify* and *do_warn*

Displaying a notification

The most common way to display a notification is by using two methods that come from the *ServiceMixin*:

***do_notify(title, message, sticky, className)*:**

Display a notification of type *notification*.

title: string. This will be displayed on the top as a title

message: string, the content of the notification

sticky: boolean, optional. If true, the notification will stay until the user dismisses it. Otherwise, the notification will be automatically closed after a short delay.

className: string, optional. This is a css class name that will be automatically added to the notification. This could be useful for styling purpose, even though its use is discouraged.

***do_warn(title, message, sticky, className)*:**

Display a notification of type *warning*.

title: string. This will be displayed on the top as a title

message: string, the content of the notification

sticky: boolean, optional. If true, the notification will stay until the user dismisses it. Otherwise, the notification will be automatically closed after a short delay.

className: string, optional. This is a css class name that will be automatically added to the notification. This could be useful for styling purpose, even though its use is discouraged.

Here are two examples on how to use these methods:

```
// note that we call _t on the text to make sure it is properly translated.
this.do_notify(_t("Success"), _t("Your signature request has been sent."));

this.do_warn(_t("Error"), _t("Filter name is required."));
```

Here an example in python:

```
# note that we call _(string) on the text to make sure it is properly translated.
def show_notification(self):
    return {
        'type': 'ir.actions.client',
        'tag': 'display_notification',
        'params': {
            'title': _('Success'),
            'message': _('Your signature request has been sent.'),
            'sticky': False,
        }
    }
```

Systray

The Systray is the right part of the menu bar in the interface, where the web client displays a few widgets, such as a messaging menu.
When the SystrayMenu is created by the menu, it will look for all registered widgets and add them as a sub widget at the proper place.
There is currently no specific API for systray widgets. They are supposed to be simple widgets, and can communicate with their environment just like other widgets with the *trigger_up* method.

Adding a new Systray Item

There is no systray registry. The proper way to add a widget is to add it to the class variable SystrayMenu.items.

```
var SystrayMenu = require('web.SystrayMenu');

var MySystrayWidget = Widget.extend({
    ...
});

SystrayMenu.Items.push(MySystrayWidget);
```

Ordering

Before adding the widget to himself, the Systray Menu will sort the items by a sequence property. If that property is not present on the prototype, it will use 50 instead. So, to position a systray item to be on the right, one can set a very high sequence number (and conversely, a low number to put it on the left).

```
MySystrayWidget.prototype.sequence = 100;
```

Translation management

Some translations are made on the server side (basically all text strings rendered or processed by the server), but there are strings in the static files that need to be translated. The way it currently works is the following:

- each translatable string is tagged with the special function *_t* (available in the JS module *web.core*
- these strings are used by the server to generate the proper PO files
- whenever the web client is loaded, it will call the route */web/webclient/translations*, which returns a list of all translatable terms
- in runtime, whenever the function *_t* is called, it will look up in this list in order to find a translation, and return it or the original string if none is found.

Note that translations are explained in more details, from the server point of view, in the document Translating Modules (translations.html).
There are two important functions for the translations in javascript: *_t* and *_lt*. The difference is that *_lt* is lazily evaluated.

```
var core = require('web.core');

var _t = core._t;
var _lt = core._lt;

var SomeWidget = Widget.extend({
    exampleString: _lt('this should be translated'),
    ...
    someMethod: function () {
        var str = _t('some text');
        ...
    },
});
```

In this example, the *_lt* is necessary because the translations are not ready when the module is loaded.
Note that translation functions need some care. The string given in argument should not be dynamic.

Session

There is a specific module provided by the web client which contains some information specific to the user current *session*. Some notable keys are

- uid: the current user ID (its ID as a *res.users*)
- user_name: the user name, as a string
- the user context (user ID, language and timezone)
- partner_id: the ID of the partner associated to the current user
- db: the name of the database currently being in use

Adding information to the session

When the /web route is loaded, the server will inject some session information in the template a script tag. The information will be read from the method *session_info* of the model *ir.http*. So, if one wants to add a specific information, it can be done by overriding the session_info method and adding it to the dictionary.

```
from odoo import models
from odoo.http import request


class IrHttp(models.AbstractModel):
    _inherit = 'ir.http'

    def session_info(self):
        result = super(IrHttp, self).session_info()
        result['some_key'] = get_some_value_from_db()
        return result
```

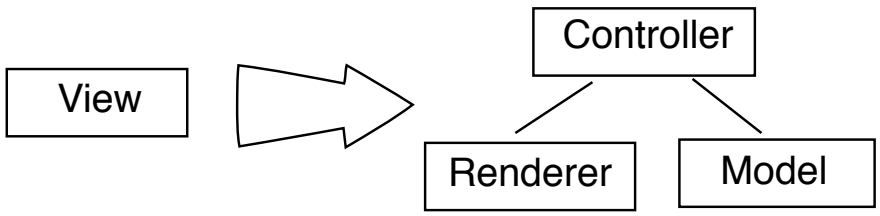
Now, the value can be obtained in javascript by reading it in the session:

```
var session = require('web.session');
var myValue = session.some_key;
...
```

Note that this mechanism is designed to reduce the amount of communication needed by the web client to be ready. It is more appropriate for data which is cheap to compute (a slow session_info call will delay the loading for the web client for everyone), and for data which is required early in the initialization process.

Views

The word ‘view’ has more than one meaning. This section is about the design of the javascript code of the views, not the structure of the *arch* or anything else.
In 2017, Odoo replaced the previous view code with a new architecture. The main need was to separate the rendering logic from the model logic.
Views (in a generic sense) are now described with 4 pieces: a View, a Controller, a Renderer and a Model. The API of these 4 pieces is described in the AbstractView, AbstractController, AbstractRenderer and AbstractModel classes.



the View is the factory. Its job is to get a set of fields, arch, context and some other parameters, then to construct a Controller/Renderer/Model triplet.
The view’s role is to properly setup each piece of the MVC pattern, with the correct information. Usually, it has to process the arch string and extract the data necessary for each other parts of the view.
Note that the view is a class, not a widget. Once its job has been done, it can be discarded.

the Renderer has one job: representing the data being viewed in a DOM element. Each view can render the data in a different way. Also, it should listen on appropriate user actions and notify its parent (the Controller) if necessary.
The Renderer is the V in the MVC pattern.

the Model: its job is to fetch and hold the state of the view. Usually, it represents in some way a set of records in the database. The Model is the owner of the ‘business data’. It is the M in the MVC pattern.

the Controller: its job is to coordinate the renderer and the model. Also, it is the main entry point for the rest of the web client. For example, when the user changes something in the search view, the *update* method of the controller will be called with the appropriate information.
It is the C in the MVC pattern.

> The JS code for the views has been designed to be usable outside of the context of a view manager/action manager. They could be used in a client action, or, they could be displayed in the public website (with some work on the assets).

Field Widgets

A good part of the web client experience is about editing and creating data. Most of that work is done with the help of field widgets, which are aware of the field type and of the specific details on how a value should be displayed and edited.

AbstractField

The *AbstractField* class is the base class for all widgets in a view, for all views that support them (currently: Form, List, Kanban).
There are many differences between the v11 field widgets and the previous versions. Let us mention the most important ones:

- the widgets are shared between all views (well, Form/List/Kanban). No need to duplicate the implementation anymore. Note that it is possible to have a specialized version of a widget for a view, by prefixing it with the view name in the view registry: *list.many2one* will be chosen in priority over *many2one*.
- the widgets are no longer the owner of the field value. They only represent the data and communicate with the rest of the view.
- the widgets do no longer need to be able to switch between edit and readonly mode. Now, when such a change is necessary, the widget will be destroyed and rerendered again. It is not a problem, since they do not own their value anyway
- the field widgets can be used outside of a view. Their API is slightly awkward, but they are designed to be standalone.

Decorations

Like the list view, field widgets have a simple support for decorations. The goal of decorations is to have a simple way to specify a text color depending on the record current state. For example,

```
<field name="state" decoration-danger="amount &lt; 10000"/>
```

The valid decoration names are:

- decoration-bf
- decoration-it
- decoration-danger
- decoration-info
- decoration-muted
- decoration-primary
- decoration-success
- decoration-warning

Each decoration *decoration-X* will be mapped to a css class *text-X*, which is a standard bootstrap css class (except for *text-it* and *text-bf*, which are handled by odoo and correspond to italic and bold, respectively). Note that the value of the decoration attribute should be a valid python expression, which will be evaluated with the record as evaluation context.

Non relational fields

We document here all non relational fields available by default, in no particular order.

integer (FieldInteger)

This is the default field type for fields of type *integer*.

Supported field types: *integer*

Options:

type: setting the input type (*text* by default, can be set on *number*)

On edit mode, the field is rendered as an input with the HTML attribute type setted on *number* (so user can benefit the native support, especially on mobile). In this case, the default formatting is disabled to avoid incompability.

```
<field name="int_value" options='{"type": "number"}'/>
```

step: set the step to the value up and down when the user click on buttons (only for input of type number, 1 by default)

```
<field name="int_value" options='{"type": "number", "step": 100}'/>
```

format: should the number be formatted. (true by default)

By default, numbers are formatted according to locale parameters.

This option will prevent the field's value from being formatted.

```
<field name="int_value" options='{"format": false}'/>
```

float (FieldFloat)

This is the default field type for fields of type *float*.

Supported field types: *float*

Attributes:

digits: displayed precision

```
<field name="factor" digits="[42,5]"/>
```

Options:

type: setting the input type (*text* by default, can be set on *number*)

On edit mode, the field is rendered as an input with the HTML attribute type setted on *number* (so user can benefit the native support, especially on mobile). In this case, the default formatting is disabled to avoid incompability.

```
<field name="int_value" options='{"type": "number"}'/>
```

step: set the step to the value up and down when the user click on buttons (only for input of type number, 1 by default)

```
  <field name="int_value" options='{"type": "number", "step": 0.1}'/>

- format: should the number be formatted. (true by default)

By default, numbers are formatted according to locale parameters.
    This option will prevent the field's value from being formatted.

.. code-block:: xml

    <field name="int_value" options='{"format": false}'/>
```

float_time (FieldFloatTime)

The goal of this widget is to display properly a float value that represents a time interval (in hours). So, for example, 0.5 should be formatted as 0:30, or 4.75 correspond to 4:45.

Supported field types: *float*

float_factor (FieldFloatFactor)

This widget aims to display properly a float value that converted using a factor given in its options. So, for example, the value saved in database is 0.5 and the factor is 3, the widget value should be formatted as 1.5.

Supported field types: *float*

float_toggle (FieldFloatToggle)

The goal of this widget is to replace the input field by a button containing a range of possible values (given in the options). Each click allows the user to loop in the range. The purpose here is to restrict the field value to a predefined selection. Also, the widget support the factor conversion as the *float_factor* widget (Range values should be the result of the conversion).

Supported field types: *float*

```
<field name="days_to_close" widget="float_toggle" options='{"factor": 2, "range": [0, 4, 8]}'/>
```

boolean (FieldBoolean)

This is the default field type for fields of type *boolean*.

Supported field types: *boolean*

char (FieldChar)

This is the default field type for fields of type *char*.

Supported field types: *char*

date (FieldDate)

This is the default field type for fields of type *date*. Note that it also works with datetime fields. It uses the session timezone when formatting dates.

Supported field types: *date*, *datetime*

Options:

datepicker: extra settings for the datepicker (https://github.com/Eonasdan/bootstrap-datetimepicker) widget.

```
<field name="datefield" options='{"datepicker": {"daysOfWeekDisabled": [0, 6]}}'/>
```

datetime (FieldDateTime)

This is the default field type for fields of type *datetime*.

Supported field types: *date*, *datetime*

Options:

datepicker: extra settings for the datepicker (https://github.com/Eonasdan/bootstrap-datetimepicker) widget.

```
<field name="datetimefield" options='{"datepicker": {"daysOfWeekDisabled": [0, 6]}}'/>
```

daterange (FieldDateRange)

This widget allows the user to select start and end date into a single picker.

Supported field types: *date*, *datetime*

Options:

related_start_date: apply on end date field to get start date value which is used to display range in the picker.

related_end_date: apply on start date field to get end date value which is used to display range in the picker.

picker_options: extra settings for picker.

```
<field name="start_date" widget="daterange" options='{"related_end_date": "end_date"}'/>
```

remaining_days (RemainingDays)

This widget can be used on date and datetime fields. In readonly, it displays the delta (in days) between the value of the field and today. The widget is intended to be used for informative purpose: therefore the value cannot be modified in edit mode.

Supported field types: *date*, *datetime*

monetary (FieldMonetary)

This is the default field type for fields of type 'monetary'. It is used to display a currency. If there is a currency fields given in option, it will use that, otherwise it will fall back to the default currency (in the session)

Supported field types: *monetary*, *float*

Options:

currency_field: another field name which should be a many2one on currency.

```
<field name="value" widget="monetary" options="{'currency_field': 'currency_id'}"/>
```

text (FieldText)

This is the default field type for fields of type *text*.

Supported field types: *text*

handle (HandleWidget)

This field's job is to be displayed as a *handle*, and allows reordering the various records by drag and dropping them.

> **⚠ Warning**
> It has to be specified on the field by which records are sorted.

> **⚠ Warning**
> Having more than one field with a handle widget on the same list is not supported.

Supported field types: *integer*

email (FieldEmail)

This field displays email address. The main reason to use it is that it is rendered as an anchor tag with the proper href, in readonly mode.

Supported field types: *char*

phone (FieldPhone)

This field displays a phone number. The main reason to use it is that it is rendered as an anchor tag with the proper href, in readonly mode, but only in some cases: we only want to make it clickable if the device can call this particular number.

Supported field types: *char*

url (UrlWidget)

This field displays an url (in readonly mode). The main reason to use it is that it is rendered as an anchor tag with the proper css classes and href.

Also, the text of the anchor tag can be customized with the *text* attribute (it won't change the href value).

```
<field name="foo" widget="url" text="Some URL"/>

Options:

- website_path: (default:false) by default, the widget forces (if not already
  the case) the href value to begin with http:// except if this option is set
  to true, thus allowing redirections to the database's own website.

- Supported field types: *char*
```

domain (FieldDomain)

The "Domain" field allows the user to construct a technical-prefix domain thanks to a tree-like interface and see the selected records in real time. In debug mode, an input is also there to be able to enter the prefix char domain directly (or to build advanced domains the tree-like interface does not allow to).

Note that this is limited to 'static' domain (no dynamic expression, or access to context variable).

Supported field types: *char*

link_button (LinkButton)

The LinkButton widget actually simply displays a span with an icon and the text value as content. The link is clickable and will open a new browser window with its value as url.

Supported field types: *char*

image (FieldBinaryImage)

This widget is used to represent a binary value as an image. In some cases, the server returns a 'bin_size' instead of the real image (a bin_size is a string representing a file size, such as 6.5kb). In that case, the widget will make an image with a source attribute corresponding to an image on the server.

Supported field types: *binary*

Options:

preview_image: if the image is only loaded as a 'bin_size', then this option is useful to inform the web client that the default field name is not the name of the current field, but the name of another field.

accepted_file_extensions: the file extension the user can pick from the file input dialog box (default value is `image/*`) (cf: `accept` attribute on <input type="file"/>)

```
<field name="image" widget='image' options='{"preview_image":"image_128"}'/>
```

binary (FieldBinaryFile)

Generic widget to allow saving/downloading a binary file.

Supported field types: *binary*

Options:

accepted_file_extensions: the file extension the user can pick from the file input dialog box (cf: `accept` attribute on <input type="file"/>)

Attribute:

filename: saving a binary file will lose its file name, since it only saves the binary value. The filename can be saved in another field. To do that, an attribute filename should be set to a field present in the view.

```
<field name="datas" filename="datas_fname"/>
```

priority (PriorityWidget)

This widget is rendered as a set of stars, allowing the user to click on it to select a value or not. This is useful for example to mark a task as high priority.

Note that this widget also works in 'readonly' mode, which is unusual.

Supported field types: *selection*

attachment_image (AttachmentImage)

Image widget for many2one fields. If the field is set, this widget will be rendered as an image with the proper src url. This widget does not have a different behaviour in edit or readonly mode, it is only useful to view an image.

Supported field types: *many2one*

```
<field name="displayed_image_id" widget="attachment_image"/>
```

image_selection (ImageSelection)

Allow the user to select a value by clicking on an image.

Supported field types: *selection*

Options: a dictionary with a mapping from a selection value to an object with the url for an image (*image_link*) and a preview image (*preview_link*).

Note that this option is not optional!

```
<field name="external_report_layout" widget="image_selection" options="{
    'background': {
        'image_link': '/base/static/img/preview_background.png',
        'preview_link': '/base/static/pdf/preview_background.pdf'
    },
    'standard': {
        'image_link': '/base/static/img/preview_standard.png',
        'preview_link': '/base/static/pdf/preview_standard.pdf'
    }
}"/>
```

label_selection (LabelSelection)

This widget renders a simple non-editable label. This is only useful to display some information, not to edit it.

Supported field types: *selection*

Options:

classes: a mapping from a selection value to a css class

```
<field name="state" widget="label_selection" options="{
    'classes': {'draft': 'default', 'cancel': 'default', 'none': 'danger'}
}"/>
```

state_selection (StateSelectionWidget)

This is a specialized selection widget. It assumes that the record has some hardcoded fields, present in the view: *stage_id*, *legend_normal*, *legend_blocked*, *legend_done*. This is mostly used to display and change the state of a task in a project, with additional information displayed in the dropdown.

Supported field types: *selection*

```
<field name="kanban_state" widget="state_selection"/>
```

kanban_state_selection (StateSelectionWidget)

This is exactly the same widget as state_selection

Supported field types: *selection*

boolean_favorite (FavoriteWidget)

This widget is displayed as an empty (or not) star, depending on a boolean value. Note that it also can be edited in readonly mode.

Supported field types: *boolean*

boolean_button (FieldBooleanButton)

The Boolean Button widget is meant to be used in a stat button in a form view. The goal is to display a nice button with the current state of a boolean field (for example, 'Active'), and allow the user to change that field when clicking on it.

Note that it also can be edited in readonly mode.

Supported field types: *boolean*

Options:

terminology: it can be either 'active', 'archive', 'close' or a customized mapping with the keys *string_true*, *string_false*, *hover_true*, *hover_false*

```
<field name="active" widget="boolean_button" options='{"terminology": "archive"}'/>
```

boolean_toggle (BooleanToggle)

Displays a toggle switch to represent a boolean. This is a subfield of FieldBoolean, mostly used to have a different look.

statinfo (StatInfo)

This widget is meant to represent statistical information in a *stat button*. It is basically just a label with a number.

Supported field types: *integer, float*

Options:

label_field: if given, the widget will use the value of the label_field as text.

```
<button name="%(act_payslip_lines)d"
    icon="fa-money"
    type="action">
    <field name="payslip_count" widget="statinfo"
        string="Payslip"
        options="{'label_field': 'label_tasks'}"/>
</button>
```

percentpie (FieldPercentPie)

This widget is meant to represent statistical information in a *stat button*. This is similar to a statinfo widget, but the information is represented in a *pie* chart (empty to full). Note that the value is interpreted as a percentage (a number between 0 and 100).

Supported field types: *integer, float*

```
<field name="replied_ratio" string="Replied" widget="percentpie"/>
```

progressbar (FieldProgressBar)

Represent a value as a progress bar (from 0 to some value)

Supported field types: *integer, float*

Options:

editable: boolean if value is editable

current_value: get the current_value from the field that must be present in the view

max_value: get the max_value from the field that must be present in the view

edit_max_value: boolean if the max_value is editable

title: title of the bar, displayed on top of the bar –> not translated, use parameter (not option) “title” instead

```
<field name="absence_of_today" widget="progressbar"
    options="{'current_value': 'absence_of_today', 'max_value': 'total_employee', 'editable': false}"/>
```

toggle_button (FieldToggleBoolean)

This widget is intended to be used on boolean fields. It toggles a button switching between a green bullet / gray bullet. It also set up a tooltip, depending on the value and some options.

Supported field types: *boolean*

Options:

active: the string for the tooltip that should be set when boolean is true

inactive: the tooltip that should be set when boolean is false

```
<field name="payslip_status" widget="toggle_button"
    options='{"active": "Reported in last payslips", "inactive": "To Report in Payslip"}'
/>
```

dashboard_graph (JournalDashboardGraph)

This is a more specialized widget, useful to display a graph representing a set of data. For example, it is used in the accounting dashboard kanban view.

It assumes that the field is a JSON serialization of a set of data.

Supported field types: *char*

Attribute

graph_type: string, can be either ‘line’ or ‘bar’

```
<field name="dashboard_graph_data"
    widget="dashboard_graph"
    graph_type="line"/>
```

ace (AceEditor)

This widget is intended to be used on Text fields. It provides Ace Editor for editing XML and Python.

Supported field types: *char, text*

badge (FieldBadge)

Displays the value inside a bootstrap badge pill.

Supported field types: *char*, *selection*, *many2one*

By default, the badge has a lightgrey background, but it can be customized by using the decoration-X mechanism. For instance, to display a red badge under a given condition:

```
<field name="foo" widget"badge" decoration-danger="state == 'cancel'"/>
```

Relational fields

***class* web.relational_fields.FieldSelection()**

Supported field types: *selection*

web.relational_fields.FieldSelection.placeholder

a string which is used to display some info when no value is selected

```
<field name="tax_id" widget="selection" placeholder="Select a tax"/>
```

radio (FieldRadio)

This is a subfield of FielSelection, but specialized to display all the valid choices as radio buttons.

Note that if used on a many2one records, then more rpcs will be done to fetch the name_gets of the related records.

Supported field types: *selection, many2one*

Options:

horizontal: if true, radio buttons will be displayed horizontally.

```
<field name="recommended_activity_type_id" widget="radio"
    options="{'horizontal':true}"/>
```

selection_badge (FieldSelectionBadge)

This is a subfield of FieldSelection, but specialized to display all the valid choices as rectangular badges.

Supported field types: *selection, many2one*

```
<field name="recommended_activity_type_id" widget="selection_badge"/>
```

many2one (FieldMany2One)

Default widget for many2one fields.

Supported field types: *many2one*

Attributes:

can_create: allow the creation of related records (take precedence over no_create option)

can_write: allow the editing of related records (default: true)

Options:

no_create: prevent the creation of related records

quick_create: allow the quick creation of related records (default: true)

no_quick_create: prevent the quick creation of related records (don't ask me)

no_create_edit: same as no_create, maybe...

create_name_field: when creating a related record, if this option is set, the value of the *create_name_field* will be filled with the value of the input (default: *name*)

always_reload: boolean, default to false. If true, the widget will always do an additional name_get to fetch its name value. This is used for the situations where the name_get method is overridden (please do not do that)

no_open: boolean, default to false. If set to true, the many2one will not redirect on the record when clicking on it (in readonly mode)

```
<field name="currency_id" options="{'no_create': True, 'no_open': True}"/>
```

list.many2one (ListFieldMany2One)

Default widget for many2one fields (in list view).

Specialization of many2one field for list views. The main reason is that we need to render many2one fields (in readonly mode) as a text, which does not allow opening the related records.

Supported field types: *many2one*

many2one_barcode (FieldMany2OneBarcode)

Widget for many2one fields allows to open the camera from a mobile device (Android/iOS) to scan a barcode.

Specialization of many2one field where the user is allowed to use the native camera to scan a barcode. Then it uses name_search to search this value.

If this widget is set and user is not using the mobile application, it will fallback to regular many2one (FieldMany2One)

Supported field types: *many2one*

many2one_avatar (Many2OneAvatar)

This widget is only supported on many2one fields pointing to a model which inherits from 'image.mixin'. In readonly, it displays the image of the related record next to its display_name. Note that the display_name isn't a clickable link in this case. In edit, it behaves exactly like the regular many2one.

Supported field types: *many2one*

many2one_avatar_user (Many2OneAvatarUser)

This widget is a specialization of the Many2OneAvatar. When the avatar is clicked, we open a chat window with the corresponding user. This widget can only be set on many2one fields pointing to the 'res.users' model.

Supported field types: *many2one* (pointing to 'res.users')

many2one_avatar_employee (Many2OneAvatarEmployee)

Same as Many2OneAvatarUser, but for many2one fields pointing to 'hr.employee'.

Supported field types: *many2one* (pointing to 'hr.employee')

kanban.many2one (KanbanFieldMany2One)

Default widget for many2one fields (in kanban view). We need to disable all editing in kanban views.

Supported field types: *many2one*

many2many (FieldMany2Many)

Default widget for many2many fields.

Supported field types: *many2many*

Attributes:

mode: string, default view to display

domain: restrict the data to a specific domain

Options:

create_text: allow the customization of the text displayed when adding a new record

many2many_binary (FieldMany2ManyBinaryMultiFiles)

This widget helps the user to upload or delete one or more files at the same time.

Note that this widget is specific to the model 'ir.attachment'.

Supported field types: *many2many*

Options:

accepted_file_extensions: the file extension the user can pick from the file input dialog box (cf: `accept` attribute on <input type="file"/>)

many2many_tags (FieldMany2ManyTags)

Display many2many as a list of tags.

Supported field types: *many2many*

Options:

create: domain determining whether or not new tags can be created (default: True).

```
<field name="category_id" widget="many2many_tags" options="{'create': [['some_other_field', '>', 24]]}"/>
```

color_field: the name of a numeric field, which should be present in the view. A color will be chosen depending on its value.

```
<field name="category_id" widget="many2many_tags" options="{'color_field': 'color'}"/>
```

no_edit_color: set to True to remove the possibility to change the color of the tags (default: False).

```
<field name="category_id" widget="many2many_tags" options="{'color_field': 'color', 'no_edit_color': True}"
```

form.many2many_tags (FormFieldMany2ManyTags)

Specialization of many2many_tags widget for form views. It has some extra code to allow editing the color of a tag.

Supported field types: *many2many*

kanban.many2many_tags (KanbanFieldMany2ManyTags)

Specialization of many2many_tags widget for kanban views.

Supported field types: *many2many*

many2many_checkboxes (FieldMany2ManyCheckBoxes)

This field displays a list of checkboxes and allow the user to select a subset of the choices.

Supported field types: *many2many*

one2many (FieldOne2Many)

Default widget for one2many fields.

It usually displays data in a sub list view, or a sub kanban view.

Supported field types: *one2many*

Options:

create: domain determining whether or not related records can be created (default: True).

delete: domain determining whether or not related records can be deleted (default: True).

```
<field name="turtles" options="{'create': [['some_other_field', '>', 24]]}"/>
```

create_text: a string that is used to customize the 'Add' label/text.

```
<field name="turtles" options="{\'create_text\': \'Add turtle\'}">
```

statusbar (FieldStatus)

This is a really specialized widget for the form views. It is the bar on top of many forms which represent a flow, and allow selecting a specific state.

Supported field types: *selection, many2one*

reference (FieldReference)

The FieldReference is a combination of a select (for the model) and a FieldMany2One (for its value). It allows the selection of a record on an arbitrary model.

Supported field types: *char, reference*

Client actions

The idea of a client action is a customized widget that is integrated in the web client interface, just like a *act_window_action*. This is useful when you need a component that is not closely linked to an existing view or a specific model. For example, the Discuss application is actually a client action.

A client action is a term that has various meanings, depending on the context:

from the perspective of the server, it is a record of the model *ir_action*, with a field *tag* of type char

from the perspective of the web client, it is a widget, which inherit from the class AbstractAction, and is supposed to be registered in the action registry under the corresponding key (from the field char)

Whenever a menu item is associated to a client action, opening it will simply fetch the action definition from the server, then lookup into its action registry to get the Widget definition at the appropriate key, and finally, it will instantiate and append the widget to the proper place in the DOM.

Adding a client action

A client action is a widget which will control the part of the screen below the menu bar. It can have a control panel, if necessary. Defining a client action can be done in two steps: implementing a new widget, and registering the widget in the action registry.

Implementing a new client action.

This is done by creating a widget:

```
var AbstractAction = require('web.AbstractAction');

var ClientAction = AbstractAction.extend({
    hasControlPanel: true,
    ...
});
```

Registering the client action:

As usual, we need to make the web client aware of the mapping between client actions and the actual class:

```
var core = require('web.core');

core.action_registry.add('my-custom-action', ClientAction);
```

Then, to use the client action in the web client, we need to create a client action record (a record of the model `ir.actions.client`) with the proper `tag` attribute:

```
<record id="my_client_action" model="ir.actions.client">
    <field name="name">Some Name</field>
    <field name="tag">my-custom-action</field>
</record>
```

Using the control panel

By default, the client action does not display a control panel. In order to do that, several steps should be done.

Set the *hasControlPanel* to *true*. In the widget code:

```
var MyClientAction = AbstractAction.extend({
    hasControlPanel: true,
    loadControlPanel: true, // default: false
    ...
});
```

> **⚠ Warning**
> when the `loadControlPanel` is set to true, the client action will automatically get the content of a search view or a control panel view. In this case, a model name should be specified like this:
>
> ```
> init: function (parent, action, options) {
>     ...
>     this.controlPanelParams.modelName = 'model.name';
>     ...
> }
> ```

Call the method *updateControlPanel* whenever we need to update the control panel. For example:

```
var SomeClientAction = Widget.extend({
    hasControlPanel: true,
    ...
    start: function () {
        this._renderButtons();
        this._update_control_panel();
        ...
    },
    do_show: function () {
         ...
         this._update_control_panel();
    },
    _renderButtons: function () {
        this.$buttons = $(QWeb.render('SomeTemplate.Buttons'));
        this.$buttons.on('click', ...);
    },
    _update_control_panel: function () {
        this.updateControlPanel({
            cp_content: {
               $buttons: this.$buttons,
            },
        });
    }
```

The `updateControlPanel` is the main method to customize the content in controlpanel. For more information, look into the control_panel_renderer.js (https://github.com/odoo/odoo/blob/13.0/addons/web/static/src/js/views/control_panel/control_panel_renderer.js#L130)

file.