**WIDGETS**

Odoo has many widgets that are available and ready to use, such as many2many\_tags, image, Html, handle, etc. each widget has a unique functionality written in Javascript and XML files in the **web** module.

For example, let's take the **many2many\_tags** widget.

The Javascript file is written at **/web/static/src/js/fields/relational\_fields.js**

The XML file or template is written at **/web/static/src/xml/base.xml.**

**Template**

Templates are XML files that will be rendered when the widget is running. To modify the many2many\_tags widget, first, we must create a template. Create an XML file with the name of template.xml or another name in your module or addon, precisely in the /static/src/xml directory, then copy and paste the code below.

**<?xml version="1.0" encoding="UTF-8"?>**

**<templates id="template" xml:space="preserve">**

**<t t-name="FieldMany2ManyTagsLink">**

**<t t-foreach="elements" t-as="el">**

**<t t-set="color" t-value="el[colorField] || 0"/>**

**<t t-set="colornames" t-value="['No color', 'Red', 'Orange', 'Yellow', 'Light blue', 'Dark purple', 'Salmon pink', 'Medium blue', 'Dark blue', 'Fushia', 'Green', 'Purple']"/>**

**<div t-attf-class="badge badge-pill dropdown o\_tag\_color\_#{color}" t-att-data-color="color" t-att-data-index="el\_index" t-att-data-id="el.id" t-attf-title="Tag color: #{colornames[color]}">**

**<t t-set="\_badge\_text">**

**<a href="#" class="o\_external\_link" t-att-modelid="el.id">**

**<span class="o\_badge\_text" t-att-title="el.display\_name" ><span role="img" t-attf-aria-label="Tag color: #{colornames[color]}"/><t t-esc="el.display\_name"/></span>**

**</a>**

**</t>**

**<t t-if="colorField">**

**<a href="#" class="dropdown-toggle o-no-caret" data-toggle="dropdown">**

**<t t-raw="\_badge\_text"/>**

**</a>**

**</t>**

**<t t-else="">**

**<t t-raw="\_badge\_text"/>**

**</t>**

**<a t-if="!readonly" href="#" class="fa fa-times o\_delete" title="Delete" aria-label="Delete"></a>**

**</div>**

**</t>**

**</t>**

**</templates>**

The code above is not much different from the original many2many\_tags template, I only change the name to FieldMany2ManyTagsLink, add the o\_external\_link class as a trigger when we click the widget, and add the modelid attribute to store the model’s primary key.

Next, create a javascript file in your module or addon precisely in the /static/src/js/ directory with any name like widget.js.

**Define a Javascript Module**

First, you should define a javascript module by writing the code below.

**odoo.define('many2many\_tags\_link.widget', function (require) {**

**"use strict";**

**// place your code here**

**});**

Pay attention to the first parameter in the define function above. Usually, the formula is **your\_module\_name.any\_text**. The purpose of the code above is so the module that you write is easy to be **overridden** by other modules, and so that your code is executed in the proper order by odoo. See the next section.

**Dependency**

Dependencies are like the import function in python. To **override the many2many\_tags** widget, at least we must import some javascript module like the code below.

**odoo.define('many2many\_tags\_link.widget', function (require) {**

**"use strict";**

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

**var AbstractField = require('web.AbstractField');**

**var registry = require('web.field\_registry');**

**var relational\_fields = require('web.relational\_fields');**

**});**

From the code above your code will not be loaded by odoo before the web.core, web.AbstractField etc code has finished loading. This is how dependency work in Odoo.

Pay attention to the relational\_fields variable. If you notice, the web.relational\_fields is the code that was written in the /web/static/src/js/fields/relational\_fields.js file. So in odoo, dependencies are not written based on the file name where the code is written. But based on what text you insert as a parameter in the odoo.define function. I mean to import some odoo’s javasrcript module we don’t have to write the file name completely including its directory.

**Include or Extend**

Include is used if we want to change the functionality of a widget everywhere, in any view. While the extend is the same as inheritance, we can determine where this change is applied manually. Because I don’t want all views that use the many2many\_tags widget can be clicked on all views, then I will use the extend, so I can determine which many2many\_tags widget views that can be clicked and which are not. Look at the code below.

**odoo.define('many2many\_tags\_link.widget', function (require) {**

**"use strict";**

**// load all dependencies that we need, think of it as the same as import function in python**

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

**var AbstractField = require('web.AbstractField');**

**var registry = require('web.field\_registry');**

**var relational\_fields = require('web.relational\_fields');**

**var \_t = core.\_t;**

**var qweb = core.qweb;**

**// start override the relational\_fields module by calling the include or extend function**

**// the relational\_fields variable has many objects, if necessary try the console.log to show them**

**// in this code we try to override the FieldMany2ManyTags object**

**// if you want to use an include function where the changes will take effect all over the place where we use this code**

**// var FieldMany2ManyTagsLink = relational\_fields.FieldMany2ManyTags.include**

**// here we are using the extend function because we want the changes only to take effect in certain places**

**var FieldMany2ManyTagsLink = relational\_fields.FieldMany2ManyTags.extend({**

**tag\_template: "FieldMany2ManyTagsLink", // call template or xml file**

**events: \_.extend({}, AbstractField.prototype.events, { // this is list of action**

**'click .o\_delete': '\_onDeleteTag', // this action is exist in original odoo's source code**

**'click .o\_external\_link': '\_openRelated', // I add this action, if we click the widget odoo will call the \_openRelated method**

**}),**

**\_openRelated: function (event) {**

**// Actually I copy the concept of the \_onClick function on the FieldMany2One object**

**// in the same file where the original many2many\_tag widget code was written**

**// see the /web/static/src/js/fields/relational\_fields.js file**

**event.preventDefault();**

**event.stopPropagation();**

**var self = this;**

**var modelid = parseInt(event.currentTarget.getAttribute('modelid'));**

**if (this.mode === 'readonly' && !this.noOpen && modelid) {**

**this.\_rpc({**

**model: this.field.relation,**

**method: 'get\_formview\_action',**

**args: [[modelid]],**

**context: this.record.getContext(this.recordParams),**

**})**

**.then(function (action) {**

**self.trigger\_up('do\_action', {action: action});**

**});**

**}**

**},**

**});**

**// this code is written so that our widget can be called by writing**

**// <field name="sales\_person\_ids" widget="many2many\_tags\_link" />**

**// in the ERP XML file**

**// if we use the extend function and this code is not written, the widget cannot be used**

**registry**

**.add('many2many\_tags\_link', FieldMany2ManyTagsLink);**

**// This code is written so that the widget that we write can be used as a dependency by other modules and can be overridden**

**// the widget keeps running if we don't write this**

**return {**

**FieldMany2ManyTagsLink: FieldMany2ManyTagsLink,**

**}**

**});**

**CREATING NEW WIDGET**

There are 3 areas where javascript is used in odoo, it’s the backend, the point of sale, and the frontend (website or e-commerce).

A widget is a part of odoo which is used to render the value of a field. For example, the **Many2many** field by default will be displayed in the form of a **table**, but if we use the **many2many\_tags** widget the value of the field will be displayed in the form of a chip or **badge**.

Another example of a widget is the **image** **widget**, which is usually used to display **images** on the master product.

To create a widget we must load the javascript file first. Create a js file, with any name for example widget\_one.js. Put this file in the your\_module\_name/static/src/js/ directory. Before writing the widget code, we should test whether the widget\_one.js file was successfully loaded by odoo or not by writing the console.log code as below.

**console.log('hello world !');**

Next, to load the javascript file we have to create an XML file, usually, the file name is assets.xml which is placed in the view directory, but we can also use another name. In this XML file, create a template that inherits the web.assets\_backend template, then write the code to load the widget\_one.js file that we created earlier like in the code below.

**<?xml version="1.0" encoding="utf-8"?>**

**<odoo>**

**<data>**

**<template id="tutorial\_javascript\_assets\_backend" inherit\_id="web.assets\_backend" name="assets backend">**

**<xpath expr="script[last()]" position="after">**

**<script type="text/javascript" src="/tutorial\_javascript/static/src/js/widget\_one.js"/>**

**</xpath>**

**</template>**

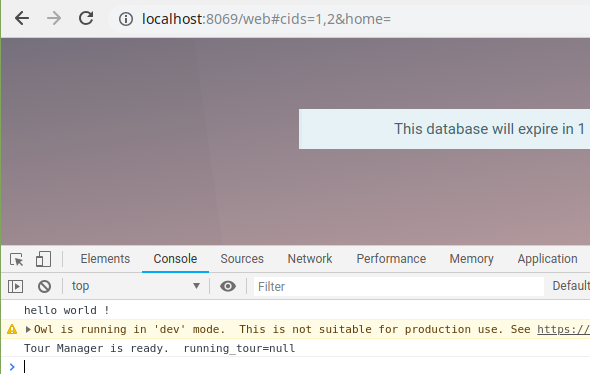
**</data>**

**</odoo>**

When you create a template to load a javascript file, like in the code above, pay attention to the **inherit\_id=”web.assets\_backend”** section. If we inherit to the **web.assets\_backend template,** odoo will put our javascript file in the **backend/ERP**. Therefore, the code that we write **will not run on point of sale or e-commerce**.

The code used to load the javascript file is the same as the HTML code in general, it’s the script tag. You need to pay attention to the src attribute. In this attribute, we must write the path where the file is stored completely, including the module name. In this example, my module name is **tutorial\_javascript**.

Then load the XML file in the \_\_manifest\_\_.py file, restart the odoo service, then install the module. When we refresh the browser, if the javascript file that we created is successfully loaded by odoo, the message should appear on the console, as shown below.



Next, in the **widget\_one.js** file, call the **odoo.define** function with the first argument being a **string**. This string will be used as a marker so the widget that we write can be **inherited** or **overridden** by other modules. This string must be **unique**, therefore usually the writing format is the name of the module followed by any text.

The second argument is a function where our’s widget code should be written.

**odoo.define('tutorial\_javascript.widget\_one', function (require) {**

**"use strict";**

**console.log('Place your widget code here');**

**});**

All widgets must extend to the **AbstractField** **object** or its child. It also must be added to the **field\_registry** **object**. So let’s import those two objects first.

**odoo.define('tutorial\_javascript.widget\_one', function (require) {**

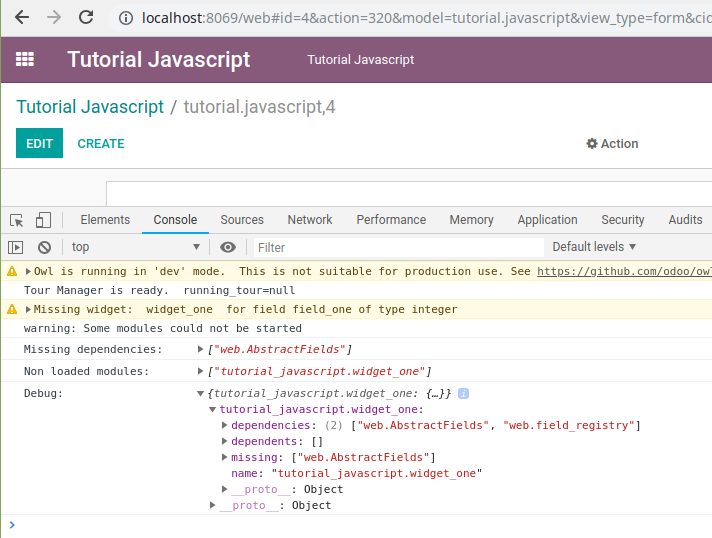
**"use strict";**

**var AbstractField = require('web.AbstractField');**

**var FieldRegistry = require('web.field\_registry');**

**});**

When creating a widget, always enter **debug mode and activate the developer tools**, to find out if there are **errors**. This is an image example that shows the odoo error message when we create a widget.



The above error message occurs because of the **web.AbstractFields** object was not found due to a **typo**, the correct object should be **web.AbstractField** without the ‘s’ at the end of the object name.

we will create a simple widget, for example, to display a short message. The view of a widget can be generated via a template in XML format, or via javascript code by overriding the **\_render, \_renderEdit, or \_renderReadonly** methods.

To create a template, first, create an XML file then write code like the code below.

**<?xml version="1.0" encoding="UTF-8"?>**

**<template>**

**<t t-name="WidgetOneTemplate">**

**<div>**

**<t t-if="widget.mode == 'edit' ">**

**<p>This is the widget view in edit mode (the user is editing the document)</p>**

**</t>**

**<t t-if="widget.mode == 'readonly' ">**

**<p>This is the widget view in readonly mode (the user only sees the document)</p>**

**</t>**

**</div>**

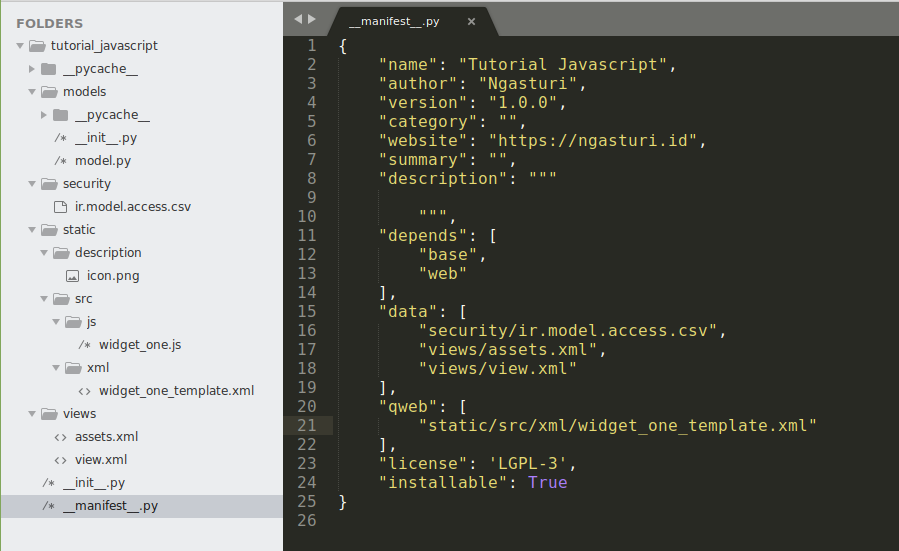
**</t>**

**</template>**

A widget has 2 modes, it is the edit mode where the user is pressing the Create or the Edit button, so he can change the contents of the document. And the readonly mode where the user can only see the document. So we have to prepare 2 different views for these two modes.

Next, save the XML file above in the your\_module\_name/static/src/xml directory. And don’t forget to load it in the \_\_manifest\_\_.py file.

The directory structure of the module we created and the contents of the \_\_manifest\_\_.py file will look like below.



Then we just need to write the actual widget code. Please read the code below and pay attention to the comments section.

**odoo.define('tutorial\_javascript.widget\_one', function (require) {**

**"use strict";**

**// import the required object to create a widget**

**var AbstractField = require('web.AbstractField');**

**var FieldRegistry = require('web.field\_registry');**

**// create an object with any name**

**// don't forget to extend to the web.AbstractField object or its child**

**var WidgetOne = AbstractField.extend({**

**template: 'WidgetOneTemplate', // fill with the template name that will be rendered by odoo**

**});**

**// register the widget to web.field\_registry object**

**// so we can use our widget in odoo's view/xml file**

**// with the code like below**

**// <field name="field\_one" widget="widget\_one" />**

**// the 'widget\_one' name is up to you, as long as it's always connected/without spaces**

**FieldRegistry.add('widget\_one', WidgetOne);**

**// return the widget object**

**// so it can be inherited or overridden by another module**

**return WidgetOne;**

**});**

Then we can directly use the widget that we have created in the odoo xml/view file using the code below.

**<record id="tutorial\_javascript\_form" model="ir.ui.view">**

**<field name="name">tutorial.javascript.form</field>**

**<field name="model">tutorial.javascript</field>**

**<field name="arch" type="xml">**

**<form>**

**<sheet>**

**<group>**

**<field name="field\_one" widget="widget\_one" />**

**</group>**

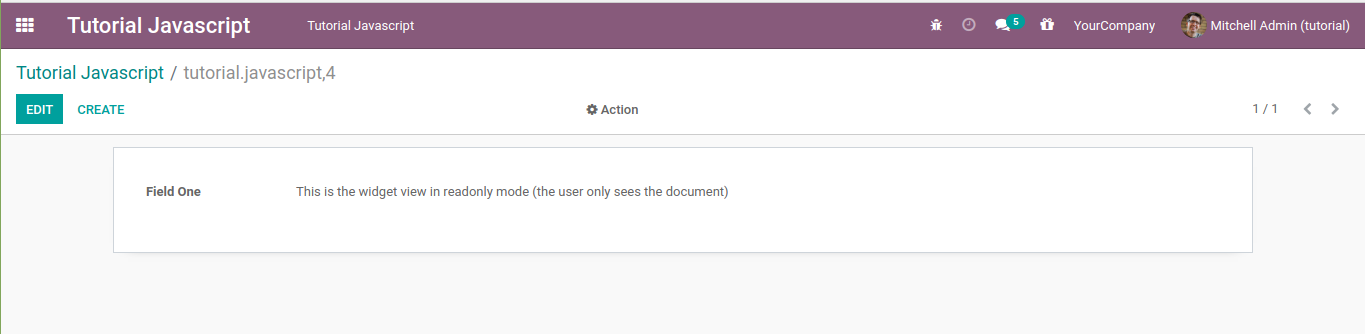
**</sheet>**

**</form>**

**</field>**

**</record>**

If your code doesn’t have an error it will look like this.



In edit mode, the widget template will always appear. But in readonly mode, if the value of the field that uses the widget that we create is null or False, the view will be blank. So make sure to fill the field value so the readonly example view model can be seen. You can change it via the database or give the field a default value like in the code below.

**field\_one = fields.Integer('Field One', default=1)**

You need to remember that when you edit any files in the static directory, in this case including the js, CSS, and XML files, you don’t need to upgrade the module that you created. Just restart your odoo service and refresh the browser.

When you create a field with the type of **integer**, when you add it to the XML/view file, odoo will **automatically** **render** the **value** of that field in an input element in **edit** mode, so the user can change the value **directly** on the input element earlier. In this, we will try to change the **appearance** of the **integer** field above, which initially can be changed **freely** by the **user**, to be a little more **limited**, by adding 2 **buttons**. The – buttons to **decrease** the value and the + buttons to **increase** the value. So if the user wants to change the integer field value, he must click the – button or the + button, it can no longer be typed freely.

First, let’s change the WidgetOneTemplate template we created before to display two buttons ( – and + ) and an input field that is disabled in edit mode. Then only it displays the value in readonly mode, without buttons and input field.

**<?xml version="1.0" encoding="UTF-8"?>**

**<template>**

**<t t-name="WidgetOneTemplate">**

**<div>**

**<t t-if="widget.mode == 'edit' ">**

**<div class="input-group">**

**<div class="input-group-prepend">**

**<button class="btn btn-danger btn-minus"> - </button>**

**</div>**

**<input type="text" class="form-control" t-att-value="widget.value" disabled="disabled" />**

**<div class="input-group-append">**

**<button class="btn btn-success btn-plus"> + </button>**

**</div>**

**</div>**

**</t>**

**<t t-if="widget.mode == 'readonly' ">**

**<span t-esc="widget.value" />**

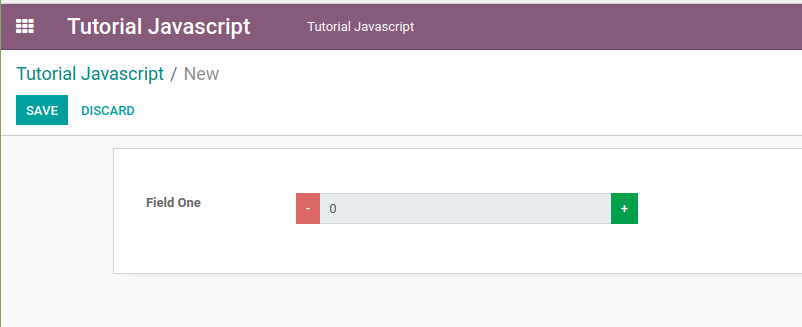
**</t>**

**</div>**

**</t>**

**</template>**

In edit mode, it will look like the image below.



Then in the widget\_one.js file let’s add an event for each button, like in the code below.

**var WidgetOne = AbstractField.extend({**

**template: 'WidgetOneTemplate', // fill with the template name that will be rendered by odoo**

**events: { // list of event, like jquery event**

**'click .btn-minus': 'btn\_minus\_action',**

**'click .btn-plus': 'btn\_plus\_action',**

**},**

**btn\_minus\_action: function(){**

**var new\_value = this.value - 1;**

**this.\_setValue(new\_value.toString());**

**console.log(this.value);**

**},**

**btn\_plus\_action: function(){**

**var new\_value = this.value + 1;**

**this.\_setValue(new\_value.toString());**

**console.log(this.value);**

**},**

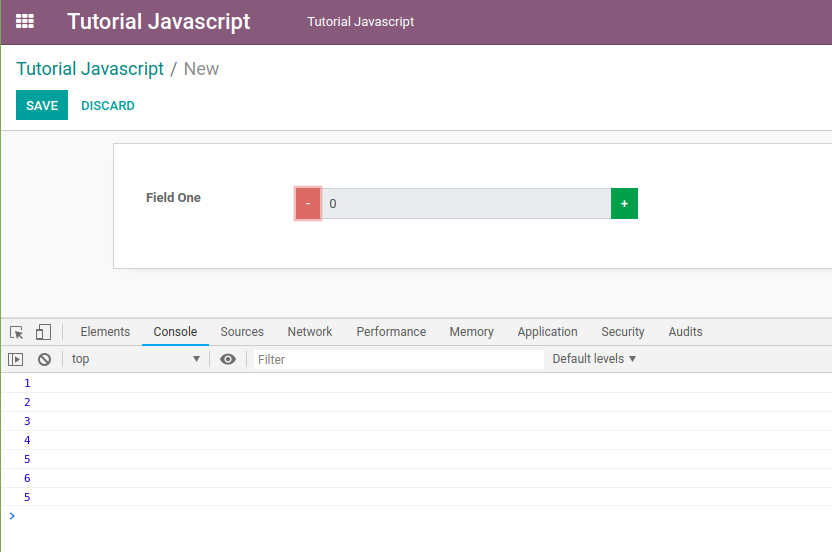
**});**

The events in odoo are almost the same as the events in jquery, except that they are written in reverse. In **jquery**, if we want to write the **click** **event** on a button with the **btn**-**minus** class we can write it like this.

**$('.btn-minus.').click(btn\_minus\_action);**

To change the value of a field we can use **this.\_setValue(new\_value) method**, but there are a few things that you need to pay attention to regarding the **data type** of this **new\_value** **variable**, **depending** on the **data** **type** of the **fields** that use the **widget** that we **create**. For example, in an integer field the variable that we pass to **this.\_setValue()** method must be a **string**. If it is a **number** it will cause an **error**. Strange? Not really. Because by **default** we can input the field with the type of **integer** with a **value** like 123,456.78 which of course is a **string**. And odoo can still **process** that **value** as a **number**. Because in **shadow**, **odoo** will **parse** the **new\_value** variable in the form of a **string** **to** **a** **number** so it can be processed in addition, subtraction, etc. operations, even though the view display 123,456.78 which of course is not a valid numeric value in javascript. Therefore, we have to change the **new\_value** variable from **number** **data** **type** **to** **string** **data** **type** with the **toString()** method.

After refreshing the browser, we can check if it was successfully added.



It is **not working** yet. On the console, the **value has been changed but on the user interface** the value is still **zero**.

It turns out that when the value of a field has been changed, the **user interface is not automatically re-rendered by odoo**. In order for the user interface to be re-rendered when the value of a field changes, we must **override the \_render or the \_renderEdit** methods if you want to manage the display logic between the edit and readonly mode through javascript code. But, we used the display logic between the edit and the readonly mode is managed in the XML file with the **t-if attribute, we will override the \_render method only.**

There are **several ways to change the appearance of the user interface**, **the easiest way is to use the qweb.** So let’s import the qweb and then override the \_render method like in the code below.

**odoo.define('tutorial\_javascript.widget\_one', function (require) {**

**"use strict";**

**// import the required object to create a widget**

**var AbstractField = require('web.AbstractField');**

**var FieldRegistry = require('web.field\_registry');**

**// import qweb to render a view**

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

**var qweb = core.qweb;**

**// create an object with any name**

**// don't forget to extend to the web.AbstractField object or its child**

**var WidgetOne = AbstractField.extend({**

**template: 'WidgetOneTemplate', // fill with the template name that will be rendered by odoo**

**events: { // list of event, like jquery event**

**'click .btn-minus': 'btn\_minus\_action',**

**'click .btn-plus': 'btn\_plus\_action',**

**},**

**btn\_minus\_action: function(){**

**var new\_value = this.value - 1;**

**this.\_setValue(new\_value.toString());**

**},**

**btn\_plus\_action: function(){**

**var new\_value = this.value + 1;**

**this.\_setValue(new\_value.toString());**

**},**

**\_render: function () {**

**// re-render the view if the field value is changed**

**console.log(this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this})));**

**},**

**});**

**// register the widget to web.field\_registry object**

**// so we can use our widget in odoo's view/xml file**

**// with the code below**

**// <field name="field\_one" widget="widget\_one" />**

**// the 'widget\_one' name is up to you, as long as it's always connected/without spaces**

**FieldRegistry.add('widget\_one', WidgetOne);**

**// return the widget object**

**// so it can be inherited or overridden by another module**

**return WidgetOne;**

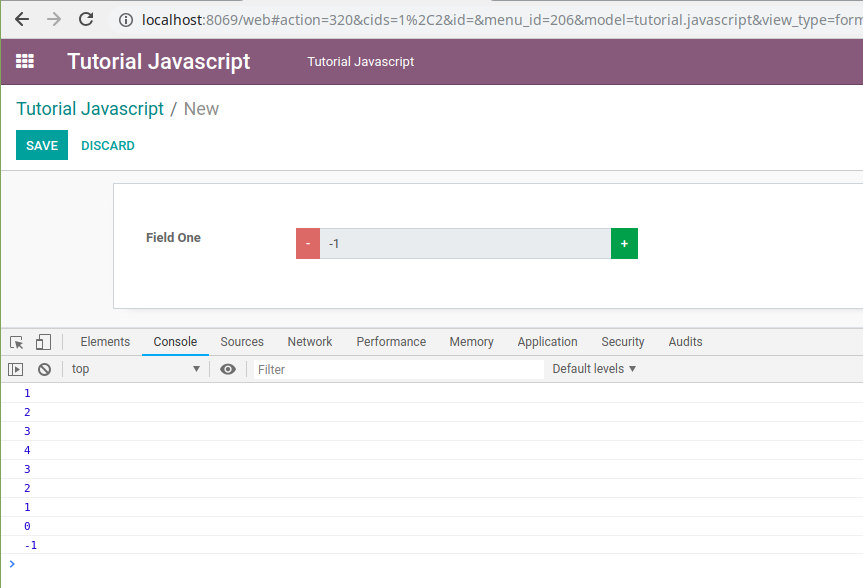
**});**

The **first parameter** that we have to pass is the qweb.render method is the **name of the template** to be rendered. Meanwhile, the **second parameter** is the **object/data** that we want to display. In the second parameter, the **key of the object we set to widget** because in the **WidgetOneTemplate** template we use the widget.value code to display the value of the field that uses the widget that we create. We can change the name of this key, for example, like in the code below.

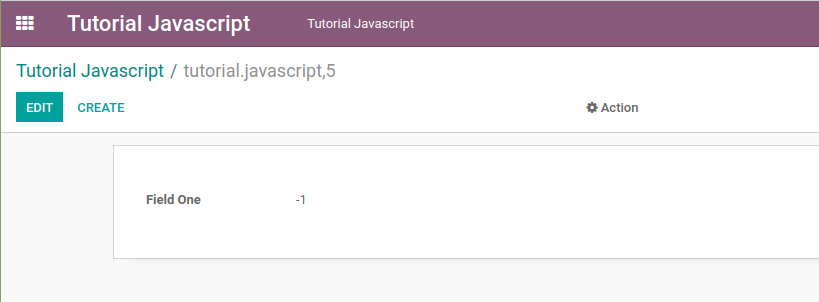
**this.$el.html($(qweb.render(this.template, {'data': this})));**

But in the **WidgetOneTemplate** template, we also have to change the code to **data.value**, otherwise it will cause an error.

After restarting our widget will be displayed as below



After Rendering, check whether the value is stored or not by clicking the Save Button



We can use the attrs, options, or context attributes, which are usually included when we write a field in the xml/view file. For this, we will use the options attribute to detect whether the user configures the addition or subtraction value or not. If the user configures it in the options attribute like in the code below.

**<field name="field\_one" widget="widget\_one" options="{'step': 1000}"/>**

We will make the value of the field that uses the widget\_one widget will be increased or decreased by the **step value** configured by the user in the XML file, which is 1000. But if the user does not configure it. We can save the **step** **value** **as** **a** **property** **of** **the** **widget**, then we can use the **init** **method** which will be **called** **first** **automatically** **by** **odoo** to detect whether the user configures the step value or not.

**odoo.define('tutorial\_javascript.widget\_one', function (require) {**

**"use strict";**

**// import the required object to create a widget**

**var AbstractField = require('web.AbstractField');**

**var FieldRegistry = require('web.field\_registry');**

**// import qweb to render a view**

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

**var qweb = core.qweb;**

**// create an object with any name**

**// don't forget to extend to the web.AbstractField object or its child**

**var WidgetOne = AbstractField.extend({**

**step: 1, // default value, if user not configure it in xml file**

**template: 'WidgetOneTemplate', // fill with the template name that will be rendered by odoo**

**events: { // list of event, like jquery event**

**'click .btn-minus': 'btn\_minus\_action',**

**'click .btn-plus': 'btn\_plus\_action',**

**},**

**init: function () {**

**// the 'init' method is called first**

**this.\_super.apply(this, arguments);**

**if(this.nodeOptions.step){**

**// if user configure the 'step' value in xml file**

**// change the default value to user's desired value**

**this.step = this.nodeOptions.step;**

**}**

**},**

**btn\_minus\_action: function(){**

**var new\_value = this.value - this.step;**

**this.\_setValue(new\_value.toString());**

**},**

**btn\_plus\_action: function(){**

**var new\_value = this.value + this.step;**

**this.\_setValue(new\_value.toString());**

**},**

**\_render: function () {**

**// re-render the view if the field value is changed**

**console.log(this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this})));**

**},**

**});**

**// register the widget to web.field\_registry object**

**// so we can use our widget in odoo's view/xml file**

**// with the code like below**

**// <field name="field\_one" widget="widget\_one" />**

**// the 'widget\_one' name is up to you, as long as it's always connected/without spaces**

**FieldRegistry.add('widget\_one', WidgetOne);**

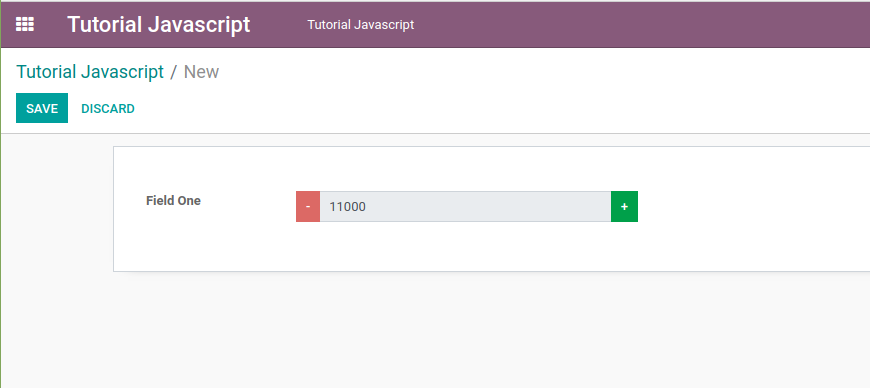
**// return the widget object**

**// so it can be inherited or overridden by another module**

**return WidgetOne;**

**});**

Now if the user configures the step value to 1000 in the XML file in the options attribute, the field value will be increased or decreased by 1000, like in the image below.



Then what if we want to add the **thousands separator? To format currency?**

We can use the **web.field\_utils object to format currencies**. Therefore, let’s import it first.

**var field\_utils = require('web.field\_utils');**

Then let’s change the **\_render** method to include the **formatted** **value** **with** **a** **thousand** **separator**.

**\_render: function () {**

**// re-render the view if the field value is changed**

**// format the value to include the thousand separator**

**var formated\_value = field\_utils.format[this.formatType](this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this, 'formated\_value': formated\_value})));**

**},**

Then let’s modify the **WidgetOneTemplate** template to **display** the **formatted** value with a thousand **separator**, **instead** of the **original** value.

**<?xml version="1.0" encoding="UTF-8"?>**

**<template>**

**<t t-name="WidgetOneTemplate">**

**<div>**

**<t t-if="widget.mode == 'edit' ">**

**<div class="input-group">**

**<div class="input-group-prepend">**

**<button class="btn btn-danger btn-minus"> - </button>**

**</div>**

**<input type="text" class="form-control" t-att-value="formated\_value" disabled="disabled" />**

**<div class="input-group-append">**

**<button class="btn btn-success btn-plus"> + </button>**

**</div>**

**</div>**

**</t>**

**<t t-if="widget.mode == 'readonly' ">**

**<span t-esc="formated\_value" />**

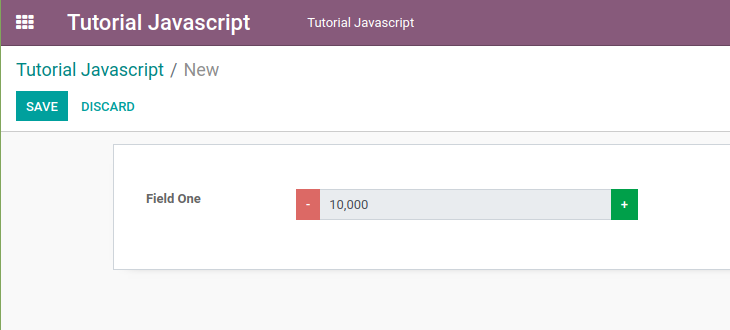
**</t>**

**</div>**

**</t>**

**</template>**

The final result will be like the below image



**USE OF JQUERY**

We can get **value from other fields/widgets**, and change the value of other fields/widgets that are still on the **same page using jquery events**.

First, let’s add a new button to the widget that we have created above (widget\_one)

**<?xml version="1.0" encoding="UTF-8"?>**

**<template>**

**<t t-name="WidgetOneTemplate">**

**<div>**

**<t t-if="widget.mode == 'edit' ">**

**<div class="input-group">**

**<div class="input-group-prepend">**

**<button class="btn btn-danger btn-minus"> - </button>**

**</div>**

**<input type="text" class="form-control" t-att-value="formated\_value" disabled="disabled" />**

**<div class="input-group-append">**

**<button class="btn btn-success btn-plus"> + </button>**

**<button class="btn btn-primary btn-copy"><i class="fa fa-copy"></i></button>**

**</div>**

**</div>**

**</t>**

**<t t-if="widget.mode == 'readonly' ">**

**<span t-esc="formated\_value" />**

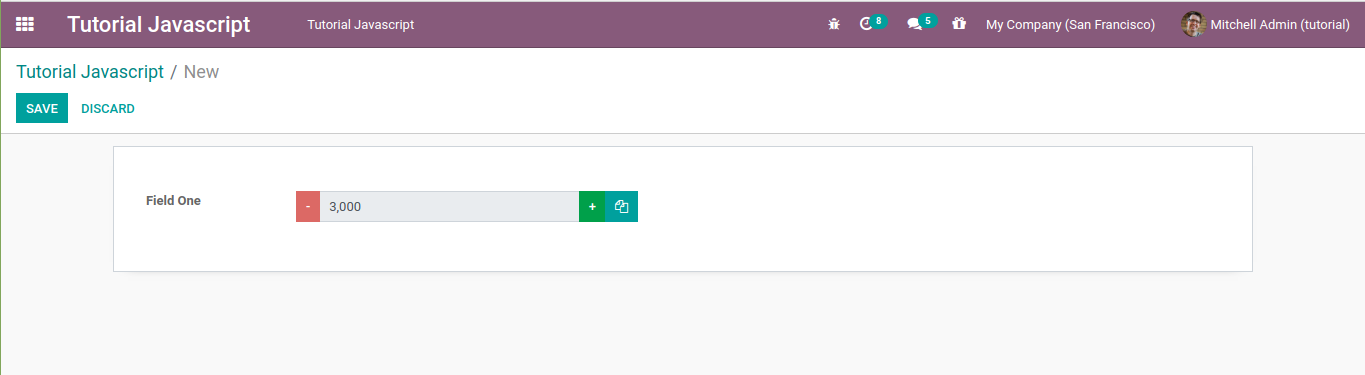
**</t>**

**</div>**

**</t>**

**</template>**

After refreshing, it will look like below



We can write jquery code in any method, even outside the widget, but if we want to access the elements of our widget, make sure the template is already rendered, for example, we can write the jquery code at the bottom of the **\_render** method.

The best way to add a jquery event to a widget is to use the **this.$el.find(element).click()**, as shown in the code below.

**\_render: function () {**

**// re-render the view if the field value is changed**

**// format the value to include the thousand separator**

**var formated\_value = field\_utils.format[this.formatType](this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this, 'formated\_value': formated\_value})));**

**this.$el.find('.btn-copy').click(function(){**

**alert('click on copy button');**

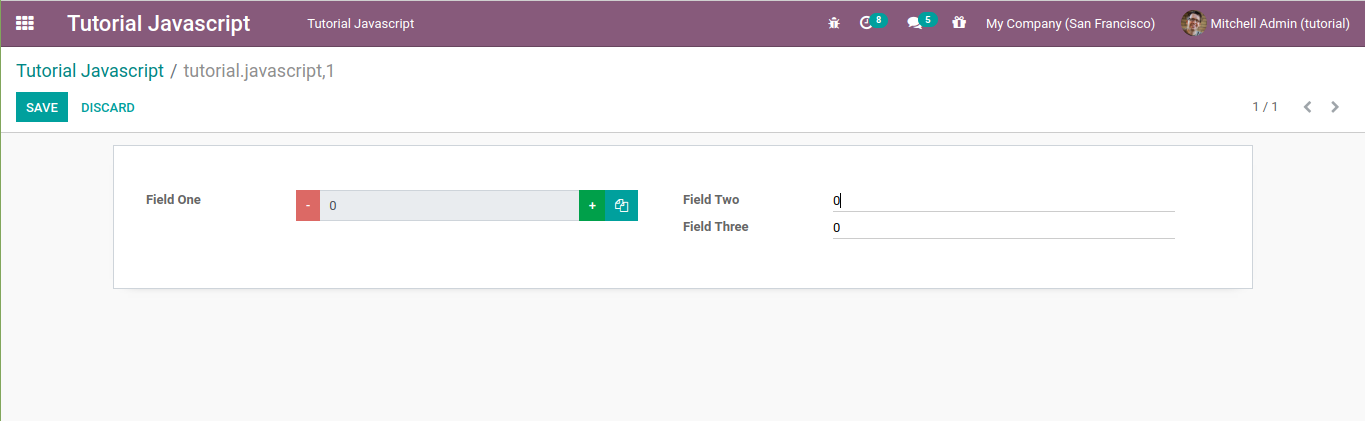
**});**

**},**

With the **this.$el.find(element)** code, odoo will ensure that the click event above will only be executed in one view/field only. I recommend you to always use **this.$el.find(element)** code if you want to write events with jquery, or to get the values of the elements in the widget. As a precaution, if the widget that we made is used more than one field in one form, or there are other widgets that use the same class.

what if we want to access values from other fields?

As an example, let’s add 2 more integer fields in the form that we have created so that the form view will be like the image below.



Then in the jquery click event that we have created, add the console.log command as shown in the code below.

**\_render: function () {**

**// re-render the view if the field value is changed**

**// format the value to include the thousand separator**

**var self = this;**

**var formated\_value = field\_utils.format[this.formatType](this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this, 'formated\_value': formated\_value})));**

**this.$el.find('.btn-copy').click(function(){**

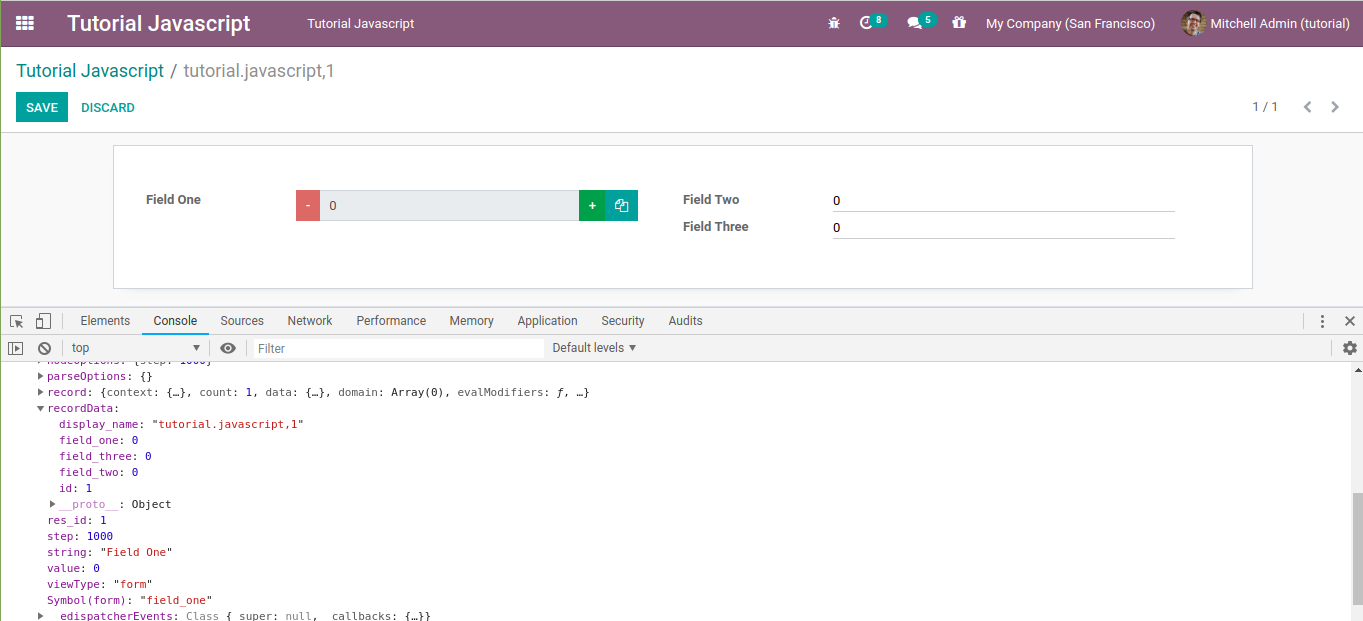
**console.log(self);**

**});**

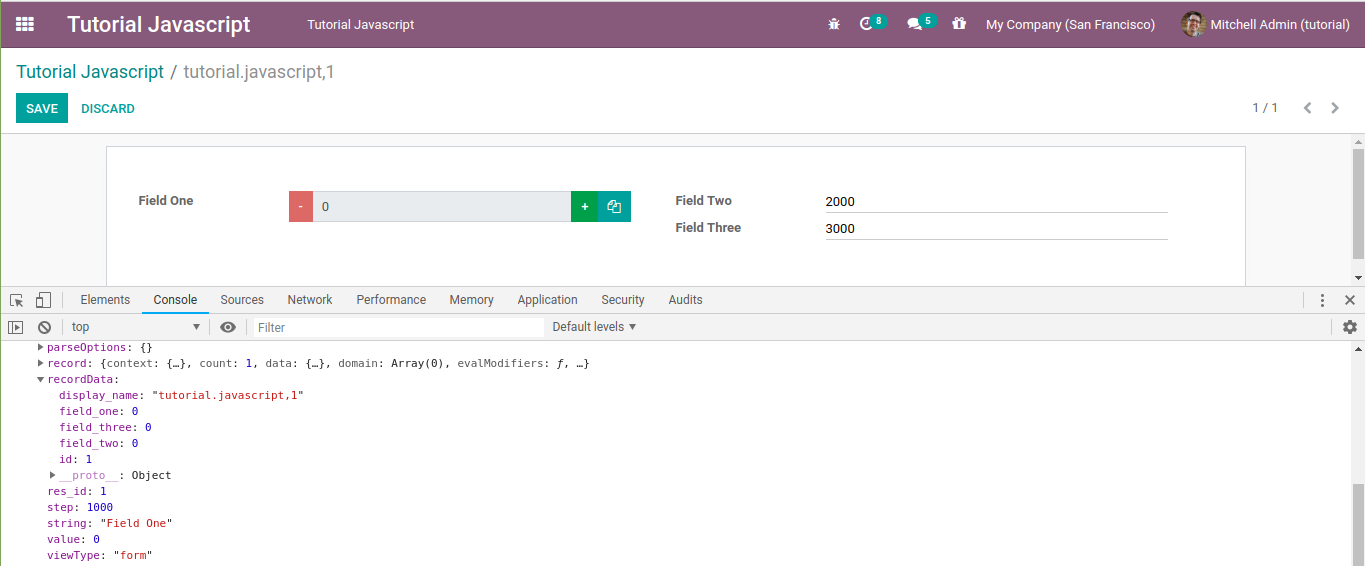
**},**

**IMPORTANT !!!** When we write the jquery event followed by an anonymous function, please **store the value of the widget in a temporary variable**, in the example code above I **save the current widget value into a variable with the name of self** because, in an anonymous function, **the this variable no longer points to the widget that we created, but points to the element that we add an event, it is the button with btn-copy class.**

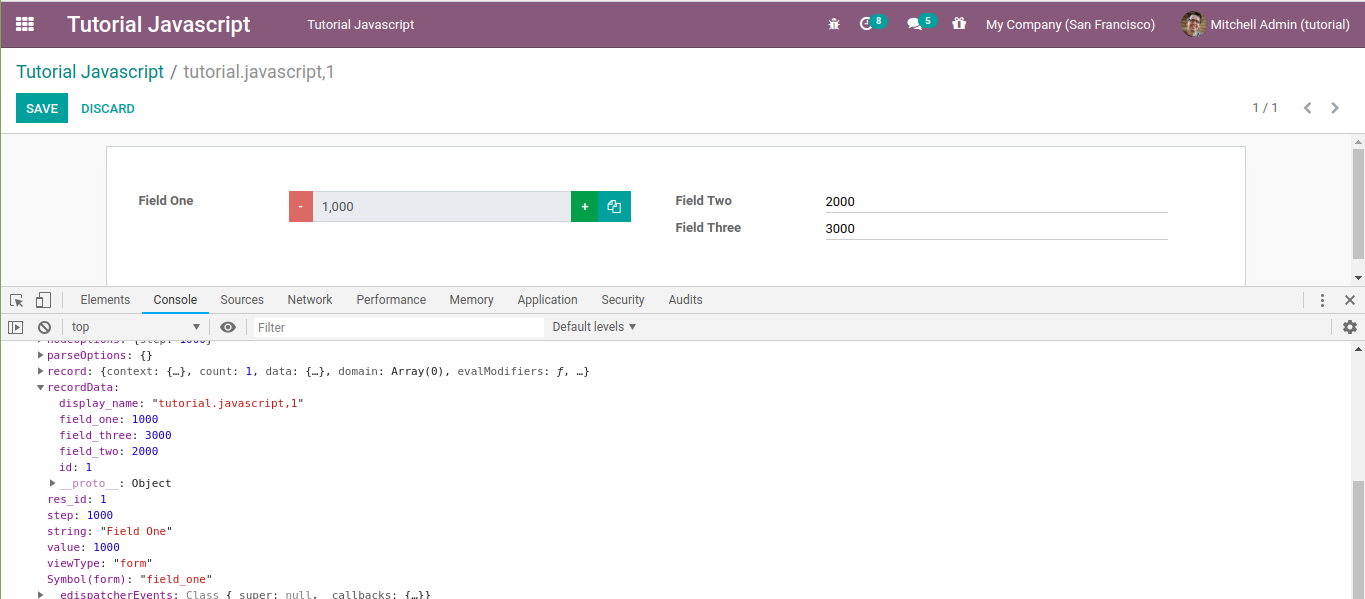
After refreshing and clicking the copy button, the self variable is printed on the console as below so that we can get the values in the record



To get the value of a field on a form, we can see it in the **recordData** or **record** property as shown above. In the picture above the values of **Field** **Two** and **Field** **Three** are still **0,** now let’s change the values of the two fields above then click the copy button again.



In the user interface the values of Field Two and Field Three have changed, but in the console tab, the values are still 0.Now let’s click the + or – button then click the copy button again.



Now the values of Field Two and Field Three in the console are the **same as the field values in the user interface**.

Next, let’s test it by moving the console.log code above in another method, for example when the user clicks the + button as shown in the code below.

**btn\_plus\_action: function(){**

**console.log(this);**

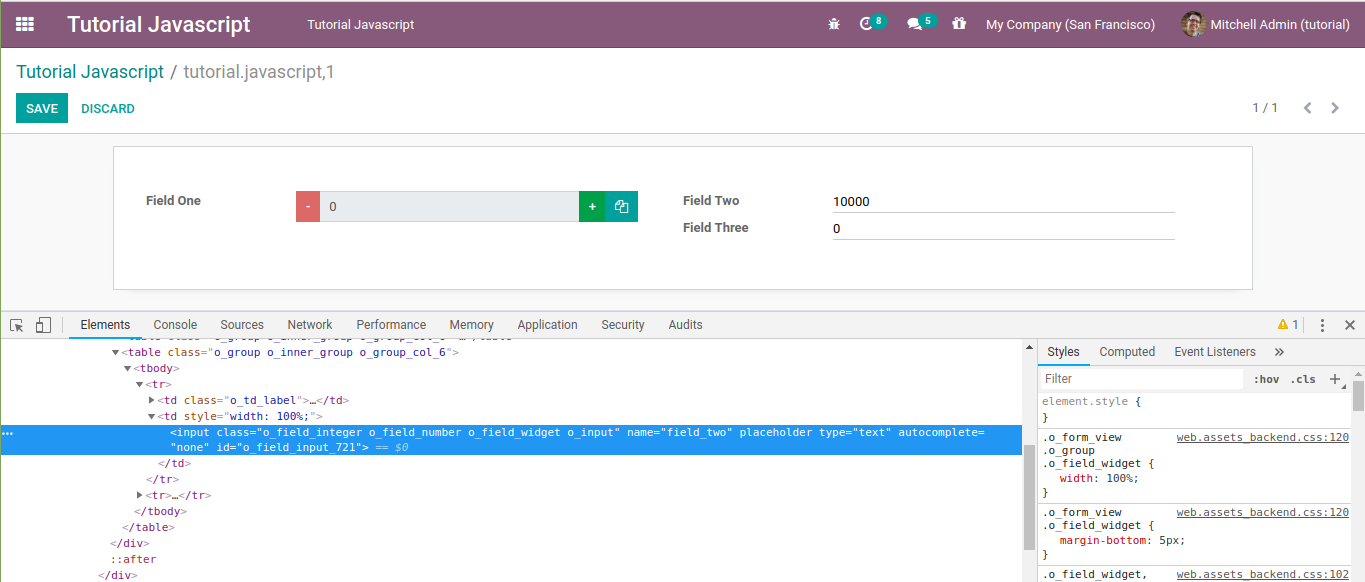
**var new\_value = this.value + this.step;**

**this.\_setValue(new\_value.toString());**

**},**

The **values of all fields should match the values displayed in the user interface**. Next try removing or **commenting the this.\_setValue(new\_value.toString());** code above. Are the values of all fields in the console the same as the values in the user interface?

It’s a bit **tricky to get the value of a field by accessing the this/self object.** However, depending on the field type, sometimes **we can still use jquery.** Please inspect the input element of Field Two then pay attention to the attribute of that element.



From the picture above, the input element of an integer field has a name attribute with the same value as the field name. So that we can access the value of that field with a jquery code like this.

**\_render: function () {**

**// re-render the view if the field value is changed**

**// format the value to include the thousand separator**

**var self = this;**

**var formated\_value = field\_utils.format[this.formatType](this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this, 'formated\_value': formated\_value})));**

**this.$el.find('.btn-copy').click(function(){**

**var field\_two\_val = $('[name=field\_two]').val();**

**console.log(field\_two\_val);**

**});**

**},**

Then what if we want to change other value fields with javascript?

To change other field values, we cannot change the **recordData** object directly, like in the **self.recordData.field\_three = 3000;** code, but we must call the **self.trigger\_up(‘field\_changed’, values)** method like in the code below.

**\_render: function () {**

**// re-render the view if the field value is changed**

**// format the value to include the thousand separator**

**var self = this;**

**var formated\_value = field\_utils.format[this.formatType](this.value);**

**this.$el.html($(qweb.render(this.template, {'widget': this, 'formated\_value': formated\_value})));**

**this.$el.find('.btn-copy').click(function(){**

**// we can also use this code**

**// self.$el.find('input').val();**

**// if we want to access the field one value with jquery**

**// by accessing the widget element**

**var field\_one\_val = self.value;**

**var field\_two\_val = $('[name=field\_two]').val();**

**var field\_three\_val = field\_one\_val + parseInt(field\_two\_val);**

**self.trigger\_up('field\_changed', {**

**dataPointID: self.dataPointID,**

**viewType: self.viewType,**

**changes: {'field\_three': field\_three\_val},**

**});**

**});**

**},**

The **viewType variable is the view type where our widget is used, usually a form or a tree.**

What you have to pay attention to is the changes variable which must be an object with a key is the name of the other field that you want to change its value, **while the value depends on the type of the field.** Since in this example, the **field\_three** type is an **integer**, the **value** must be a **number**.