odoo

Release 8.0

December 31, 2015

1 Tutorials **1**
1.1 Theme Tutorial . . . 1
1.1.1 An introduction for web designers . . . 1
1.1.2 Thinking “modular” . . . 4
1.1.3 Create a theme module . . . 9
1.1.4 Structure of an Odoo page . . . 11
1.1.5 Create a specific page layout . . . 13
1.1.6 Add Styles . . . 15
1.1.7 Create Snippets . . . 17
1.1.8 Snippet options . . . 18
1.1.9 Editing Reference Guide . . . 22
1.1.10 SEO best practice . . . 23
1.2 Building a Website . . . 25
1.2.1 Creating a basic module . . . 25
1.2.2 A demonstration module . . . 25
1.2.3 To the browser . . . 26
1.2.4 Templates . . . 26
1.2.5 Storing data in Odoo . . . 27
1.2.6 Website support . . . 29
1.2.7 URLs and routing . . . 30
1.2.8 Field edition . . . 32
1.2.9 Administration and ERP integration . . . 33
1.3 Building a Module . . . 39
1.3.1 Start/Stop the Odoo server . . . 40
1.3.2 Build an Odoo module . . . 40
1.3.3 Basic views . . . 43
1.3.4 Relations between models . . . 45
1.3.5 Inheritance . . . 46
1.3.6 Computed fields and default values . . . 49
1.3.7 Onchange . . . 50
1.3.8 Model constraints . . . 51
1.3.9 Advanced Views . . . 51
1.3.10 Workflows . . . 54
1.3.11 Security . . . 55
1.3.12 Wizards . . . 56
1.3.13 Internationalization . . . 57
1.3.14 Reporting . . . 57
1.3.15 WebServices . . . 59
1.4 Building Interface Extensions . . . 60
1.4.1 A Simple Module . . . 61
1.4.2 Odoo JavaScript Module . . . 64
1.4.3 Classes . . . 64
1.4.4 Widgets Basics . . . 66
1.4.5 The QWeb Template Engine . . . 69

1.4.6 Widget Helpers . . . 73
1.4.7 Widget Events and Properties . . . 75
1.4.8 Modify existing widgets and classes . . . 76
1.4.9 Translations . . . 77
1.4.10 Communication with the Odoo Server . . . 77
1.4.11 Exercises . . . 79
1.4.12 Existing web components . . . 80

2 Web Service API **87**
2.1 Connection . . . 87
2.1.1 Configuration . . . 87
2.1.2 Logging in . . . 87
2.2 Calling methods . . . 88
2.2.1 List records . . . 88
2.2.2 Count records . . . 88
2.2.3 Read records . . . 88
2.2.4 Listing record fields . . . 89
2.2.5 Search and read . . . 89
2.2.6 Create records . . . 90
2.2.7 Update records . . . 90
2.2.8 Delete records . . . 91
2.2.9 Inspection and introspection . . . 91
2.2.10 Workflow manipulations . . . 92
2.2.11 Report printing . . . 93

3 Setting Up **95**
3.1 Installing Odoo . . . 95
3.1.1 Demo . . . 95
3.1.2 SaaS . . . 95
3.1.3 Packaged installers . . . 96
3.1.4 Source Install . . . 97
3.2 Deploying Odoo . . . 100
3.2.1 dbfilter . . . 100
3.2.2 PostgreSQL . . . 100
3.2.3 HTTPS . . . 101
3.2.4 Builtin server . . . 101
3.2.5 Odoo as a WSGI Application . . . 101
3.2.6 Serving Static Files . . . 102
3.2.7 Security . . . 102

4 Reference **103**
4.1 ORM API . . . 103
4.1.1 Recordsets . . . 103
4.1.2 Environment . . . 105
4.1.3 Common ORM methods . . . 106
4.1.4 Creating Models . . . 106
4.1.5 Compatibility between new API and old API . . . 109
4.1.6 Model Reference . . . 110
4.1.7 Method decorators . . . 117
4.1.8 Fields . . . 120
4.1.9 Inheritance and extension . . . 125
4.1.10 Domains . . . 128
4.1.11 Porting from the old API to the new API . . . 129
4.2 Data Files . . . 131
4.2.1 Structure . . . 131
4.2.2 Core operations . . . 131
4.2.3 Shortcuts . . . 133
4.2.4 CSV data files . . . 134
4.3 Actions . . . 136

4.3.1 Window Actions (ir.actions.act_window) . . . 136
4.3.2 URL Actions (ir.actions.act_url) . . . 137
4.3.3 Server Actions (ir.actions.server) . . . 138
4.3.4 Report Actions (ir.actions.report.xml) . . . 140
4.3.5 Client Actions (ir.actions.client) . . . 140
4.4 Views . . . 141
4.4.1 Common Structure . . . 141
4.4.2 Inheritance . . . 141
4.4.3 Lists . . . 142
4.4.4 Forms . . . 144
4.4.5 Graphs . . . 151
4.4.6 Kanban . . . 151
4.4.7 Calendar . . . 153
4.4.8 Gantt . . . 153
4.4.9 Diagram . . . 154
4.4.10 Search . . . 154
4.4.11 QWeb . . . 156
4.5 Modules . . . 156
4.5.1 Manifest . . . 156
4.6 Command-line interface: odoo.py . . . 157
4.6.1 Running the server . . . 157
4.6.2 Scaffolding . . . 160
4.6.3 Configuration file . . . 160
4.7 Security in Odoo . . . 161
4.7.1 Access Control . . . 161
4.7.2 Record Rules . . . 161
4.7.3 Field Access . . . 162
4.7.4 Workflow transition rules . . . 162
4.8 Testing Modules . . . 162
4.8.1 Running tests . . . 163
4.9 Web Controllers . . . 164
4.9.1 Routing . . . 164
4.9.2 Request . . . 164
4.9.3 Response . . . 166
4.9.4 Controllers . . . 167
4.10 QWeb . . . 167
4.10.1 data output . . . 168
4.10.2 conditionals . . . 168
4.10.3 loops . . . 169
4.10.4 attributes . . . 170
4.10.5 setting variables . . . 170
4.10.6 calling sub-templates . . . 171
4.10.7 Python . . . 171
4.10.8 Javascript . . . 174
4.11 Javascript . . . 177
4.11.1 Widgets . . . 177
4.11.2 RPC . . . 182
4.11.3 Web Client . . . 185
4.11.4 Testing in Odoo Web Client . . . 185
4.12 Translating Modules . . . 194
4.12.1 Exporting translatable term . . . 194
4.12.2 Implicit exports . . . 195
4.12.3 Explicit exports . . . 195
4.13 QWeb Reports . . . 195
4.13.1 Report . . . 196
4.13.2 Report template . . . 196
4.13.3 Paper Format . . . 198
4.13.4 Custom Reports . . . 199

4.13.5 Reports are web pages . . . 199
4.14 Workflows . . . 199
4.14.1 Basics . . . 201
4.14.2 Activities . . . 202
4.14.3 Transitions . . . 204
4.15 Odoo Guidelines . . . 205
4.15.1 Module structure . . . 205
4.15.2 XML files . . . 207
4.15.3 Python . . . 208
4.15.4 Javascript and CSS . . . 211
4.15.5 Git . . . 211
4.16 Upgrade API . . . 212
4.16.1 Introduction . . . 212
4.16.2 The methods . . . 212

5 Business Mementoes **217**

Python Module Index **219**

CHAPTER

ONE

TUTORIALS

1.1 Theme Tutorial

Odoo celebrates freedom. Freedom for the designer to go further and freedom for the user to customize everything according to their needs.

Ready to create your own theme? Great. Here are some things you should know before you begin. This tutorial is a guide to creating an Odoo theme.



Warning: This tutorial requires having installed Odoo v8.0 and the Theme Support Engine (website_less) module. https://www.odoo.com/apps/8.0/website_less/

1.1.1 An introduction for web designers

If you are a web designer using Odoo for the first time, you are in the right place. This introduction will outline the basics of Odoo theme creation.

Note: Odoo's team has created a framework that's powerful and easy to use. There's no need to know special syntaxes to use this set of tools.

From common CMS to Odoo

Note: If you always think and work in the same way, you'll probably get the same results. If you want something completely new, then try something different.

> Where is my header.php file?

This is usually the first question from a web designer used to working with Wordpress or Joomla and coming to Odoo for the first time.



Indeed, when using common CMSs, you have to code several files (like header.php, page.php, post.php, etc.) in order to create a basic structure for your website. With those systems, this base structure acts as a design foundation that you have to update over time to ensure compatibility within your CMS. So, even after you have spent hours coding the files, you have not even started on the design yet.

This **does not** apply to creating Odoo themes.

Note: We think that theme design should be simple (and powerful). When we created our Website Builder, we decided to start from scratch instead of relying on what already existed. This approach gave us the freedom to focus on the things that are really important for designers: styles, content and the logic behind them. No more struggling with technical stuff.

Odoo default theme structure

Odoo comes with a default theme structure. It is a very basic "theme" that provides minimal structure and layout. When you create a new theme, you are actually extending this. Indeed it's always enabled in your setup and it acts exactly like the CMS's base structure we mentioned above, except that you don't have to create or maintain it. It will upgrade automatically within your Odoo installation and, since it is included in the Website Builder module, everything is smoothly integrated by default.

As a result, you are totally free to focus on design while this structure does the job of providing integrations and functionality.



Main features:

- Basic layouts for pages, blog and eCommerce
- Website Builder integration

- Basic Snippets
- Automatic Less/Sass compiling
- Automatic Js and CSS minification and combination

Main technologies:

- Twitter Bootstrap
- jQuery
- jQuery UI
- underscore.js

1.1.2 Thinking "modular"

An Odoo theme is not a folder containing HTML or PHP files, it's a modular framework written in XML. Never worked with XML files before? Don't worry, after following the tutorial, you'll be able to create your first theme with only basic knowledge of HTML.

Using classical web design workflows, you usually code the layout of the entire page. The result of this is a "static" web page. You can update the content, of course, but your client will need you to work on making even basic changes.

Creating themes for Odoo is a total change of perspective. Instead of defining the complete layout for a page, you can create blocks (snippets) at let the user choose where to "drag&drop" them, creating the page layout on their own. We call this modular design.

Imagine an Odoo theme as a "list" of elements and options that you have to create and style. As a designer, your goal is to style these elements in order to achieve a wonderful result, regardless of where the end user chooses to place them.

Let's take a tour of our "list" elements:

Odoo's XML files, an overview

Any Odoo XML file starts with encoding specifications. After that, you have to write your code inside a `<data>` tag, placed into an `</openerp>` tag.

```
[XML]
<?xml version="1.0" encoding="utf-8" ?>
<openerp>
 <data>
  ## YOUR CODE HERE
 </data>
</openerp>
```

Almost every element and option that you create has to be placed inside a `<template>` tag, like in this example.

```
[XML]
<template id="my_title" name="My title">
  <h1>This is an HTML block</h1>
  <h2 class="lead">And this is a subtitle</h2>
</template>
```

Important: don't misunderstand what `template` means. A template tag only defines a piece of html code or options - but it does not necessarily coincide with a visual arrangement of elements.

The previous code defines a title, but it will not be displayed anywhere because that *template* is not associated with any part of the **Odoo default structure**. In order to do that you can use **xpath**, **qWeb** or a combination of both. Keep reading the tutorial to learn to how properly extend it with your own code.

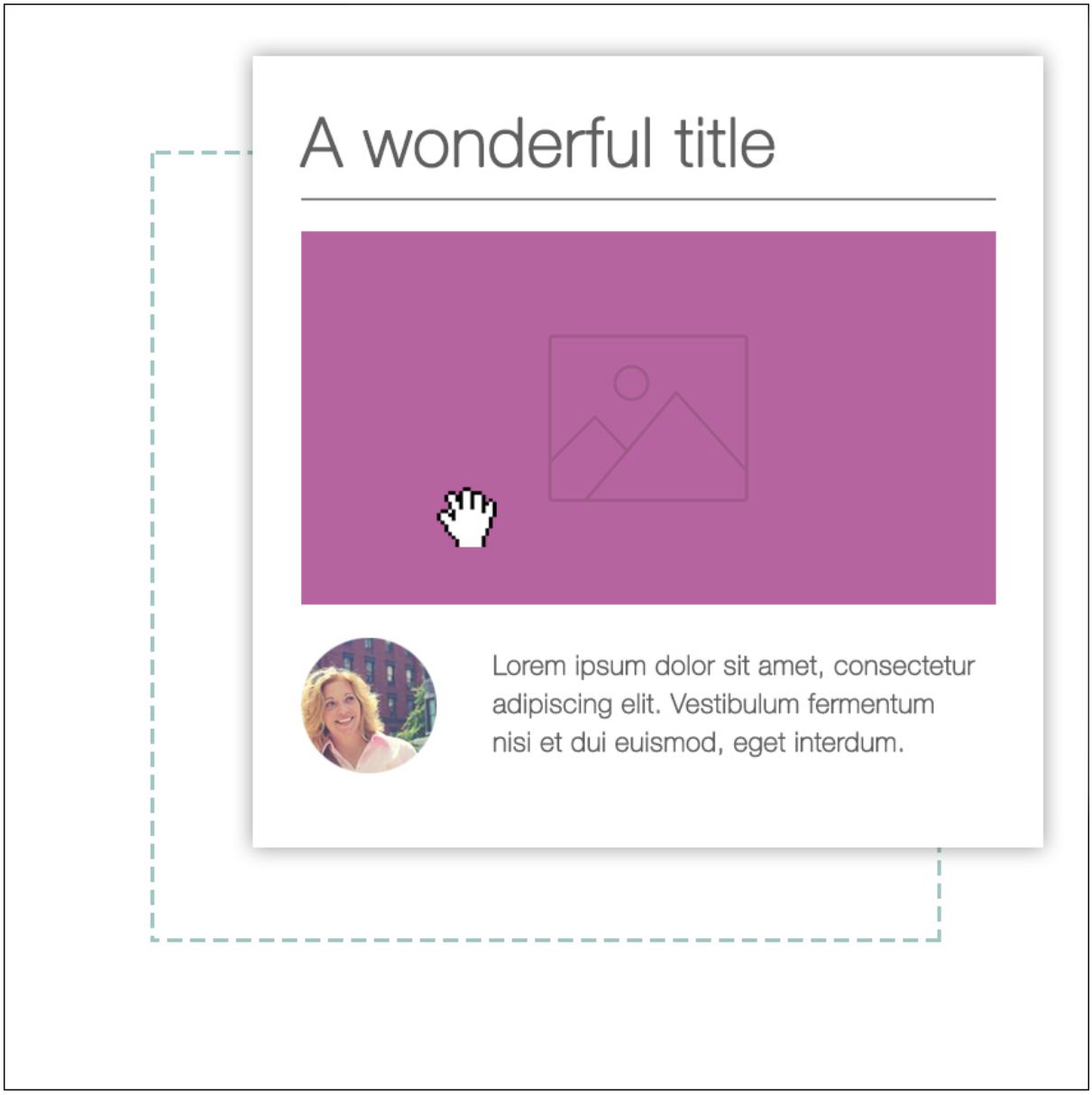


Fig. 1.1: Snippets (or building-blocks)
A piece of HTML code. The user will drag&drop, modify and combine them using our built-in Website Builder interface. You can define sets of options and styles for each snippet. The user will choose from them according to their needs.

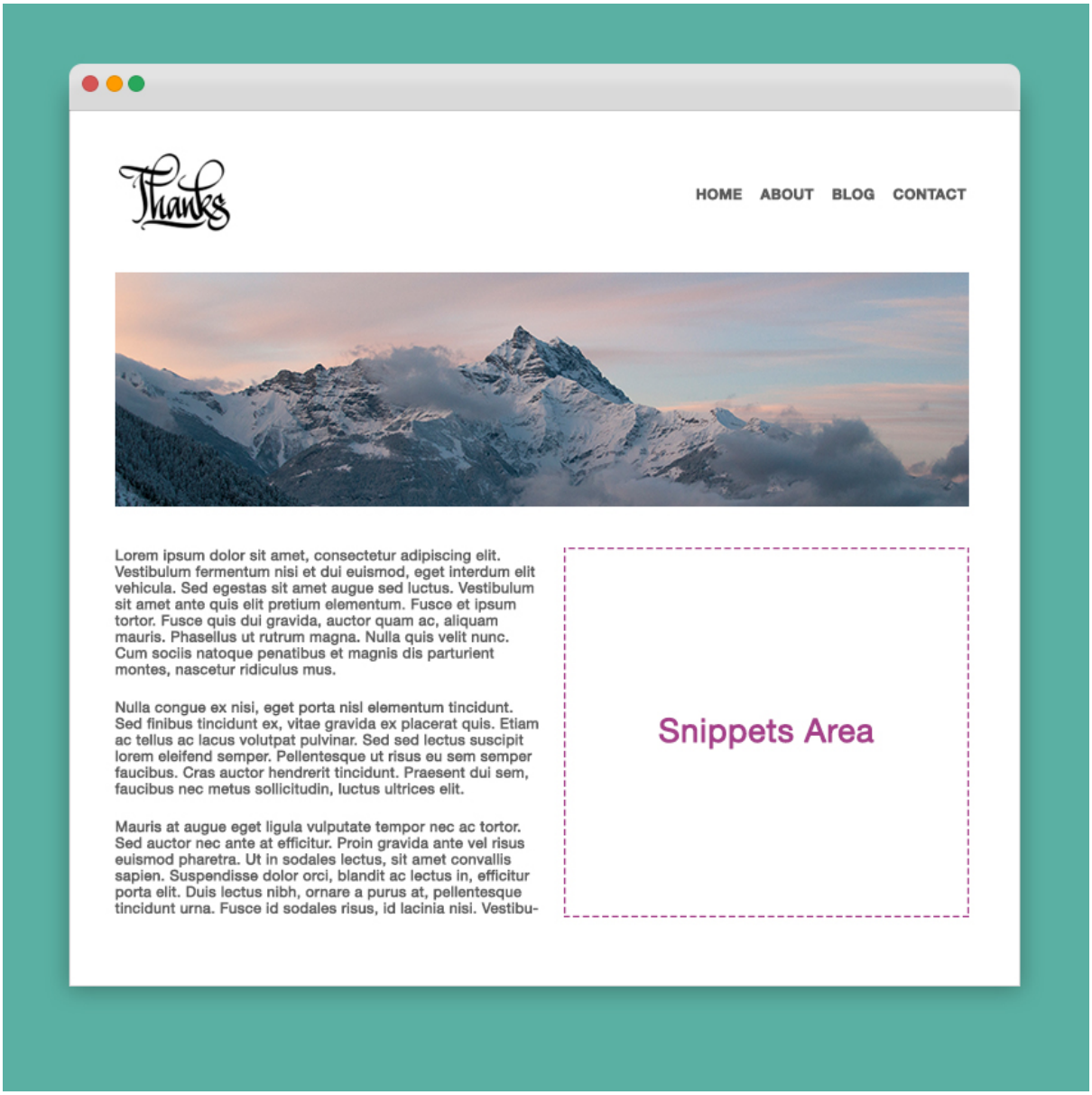


Fig. 1.2: Pages

These are normal web pages, except that they will be editable by the final user and that you can define an empty area that the user can "fill" by dragging snippets into it.

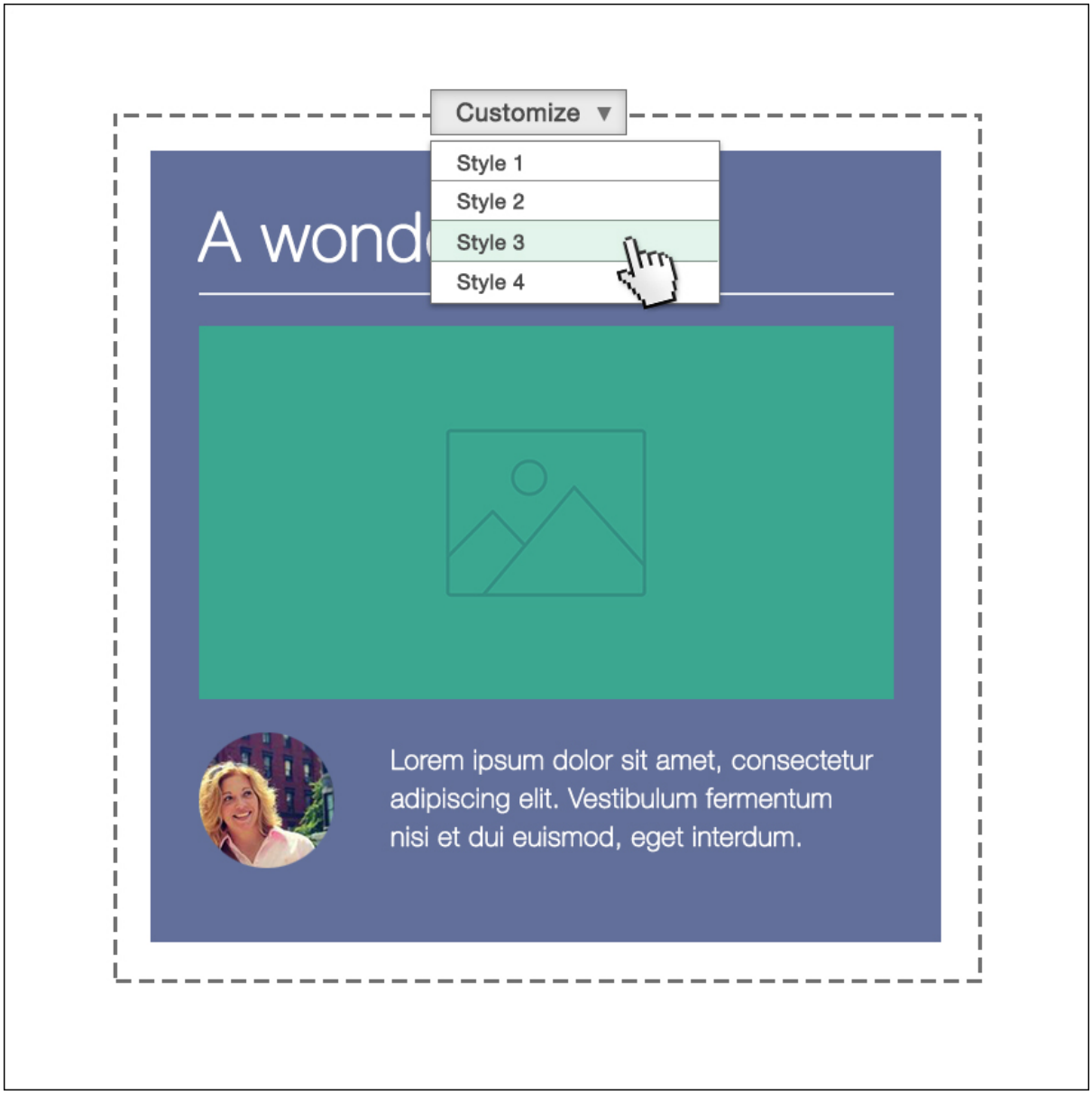


Fig. 1.3: Styles

Styles are defined using standard CSS files (or Less/Sass). You can define a style as **default** or **optional**. The default styles are always active in your theme, the optional styles can be enabled or disabled by the user.



Fig. 1.4: Functionalities

Thanks to Odoo's modularity, everything can be personalized even more. This means there are endless possibilities for your creativity. Adding functionalities is easy and it's simple to provide the end user with customizable options.

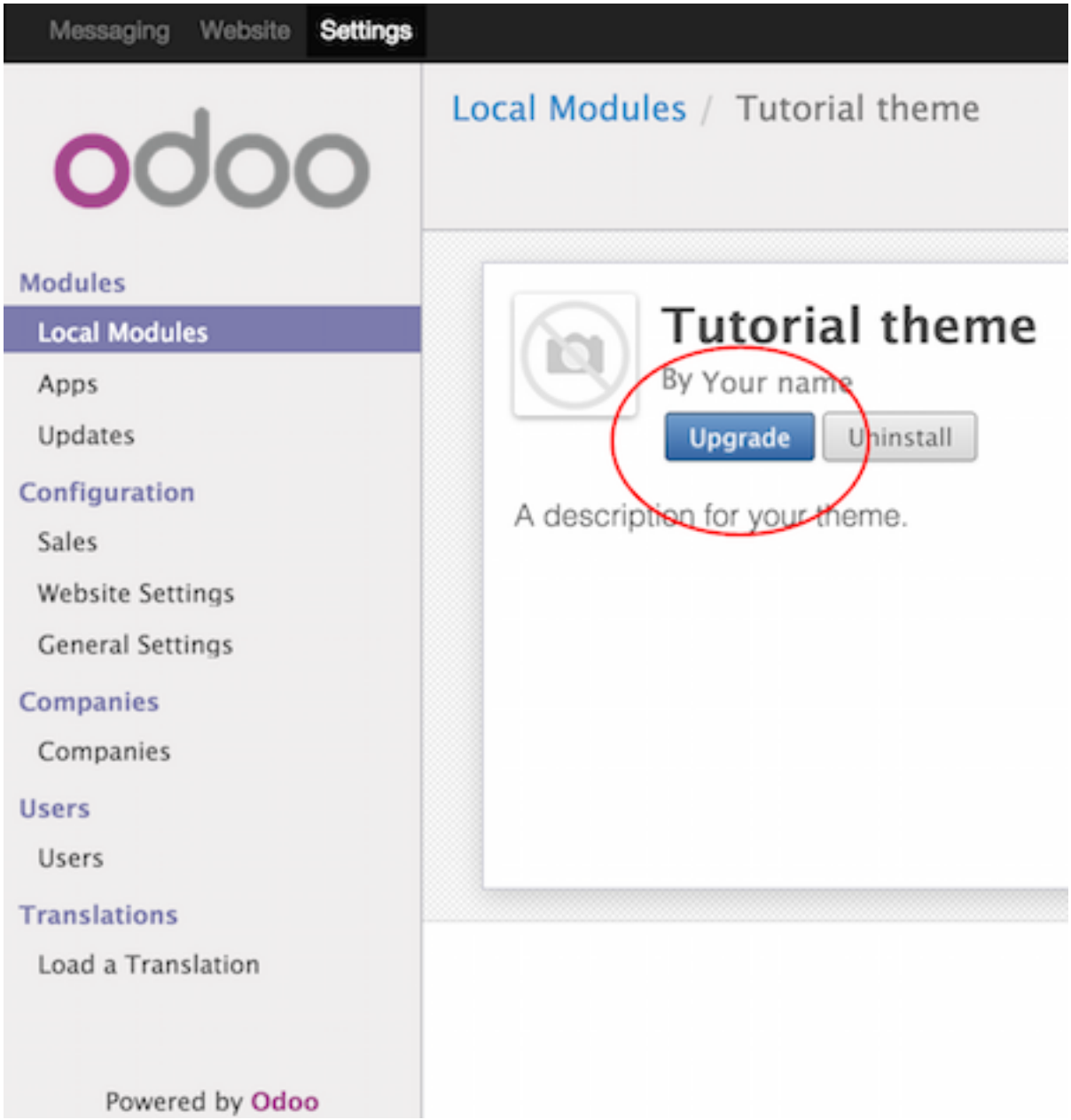
Keep reading the tutorial to learn to how properly extend it with your own code.

Update your theme

Since XML files are only loaded when you install the theme, you will have to force reloading every time you make changes on an xml file.

To do that, click on the Upgrade button in the module's page.





1.1.3 Create a theme module

Odoo's themes are packaged like modules. Even if you are designing a very simple website for your company or client, you need to package the theme like an Odoo module.

`main folder` Create a folder and name it like this: `theme_` followed by your theme's name.

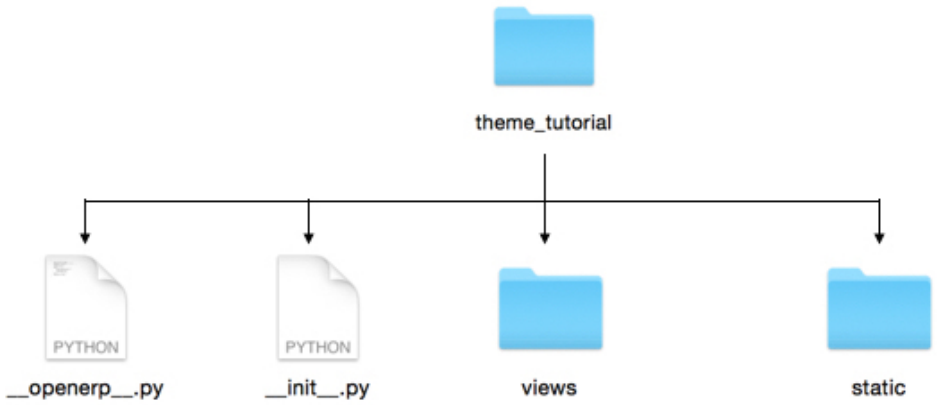
__openerp__.py Create an empty document and save it to your folder as `__openerp__.py`. This will contain the configuration info for your theme.

__init__.py Create another empty file and name it `__init__.py`. It's a mandatory system file. Create and leave it blank.

views and static folders Create them in the main folder. In `views` you'll place your xml files that define your snippets, your pages and your options. `static` folder is the right place for your style , images and custom js code.

Important: Use two underscore characters at the beginning and two at the end of openerp and init file names.

The final result should be something like this:



Edit __openerp__.py

Open the `__openerp__.py` you created and copy/paste the following:

```
{
  'name':'Tutorial theme',
  'description': 'A description for your theme.',
  'version':'1.0',
  'author':'Your name',

  'data': [
  ],
  'category': 'Theme/Creative',
  'depends': ['website', 'website_less'],
}
```

Replace the first four property's values with anything you like. These values will be used to identify your new theme in Odoo's backend.

The `data` property will contain the xml files list. Right now it's empty, but we will add any new files created.

`application: True` is mandatory.

`category` defines your module category (always "Theme") and, after a slash, the subcategory. You can use one subcategory from the Odoo Apps categories list. (https://www.odoo.com/apps/themes)

`depends` specifies the modules needed by our theme to work properly. For our tutorial theme, we only need website and website_less. If you need blogging or eCommerce features as well, you have to add those modules too.

```
...
'depends': ['website', 'website_blog', 'sale'],
...
```

Installing your theme

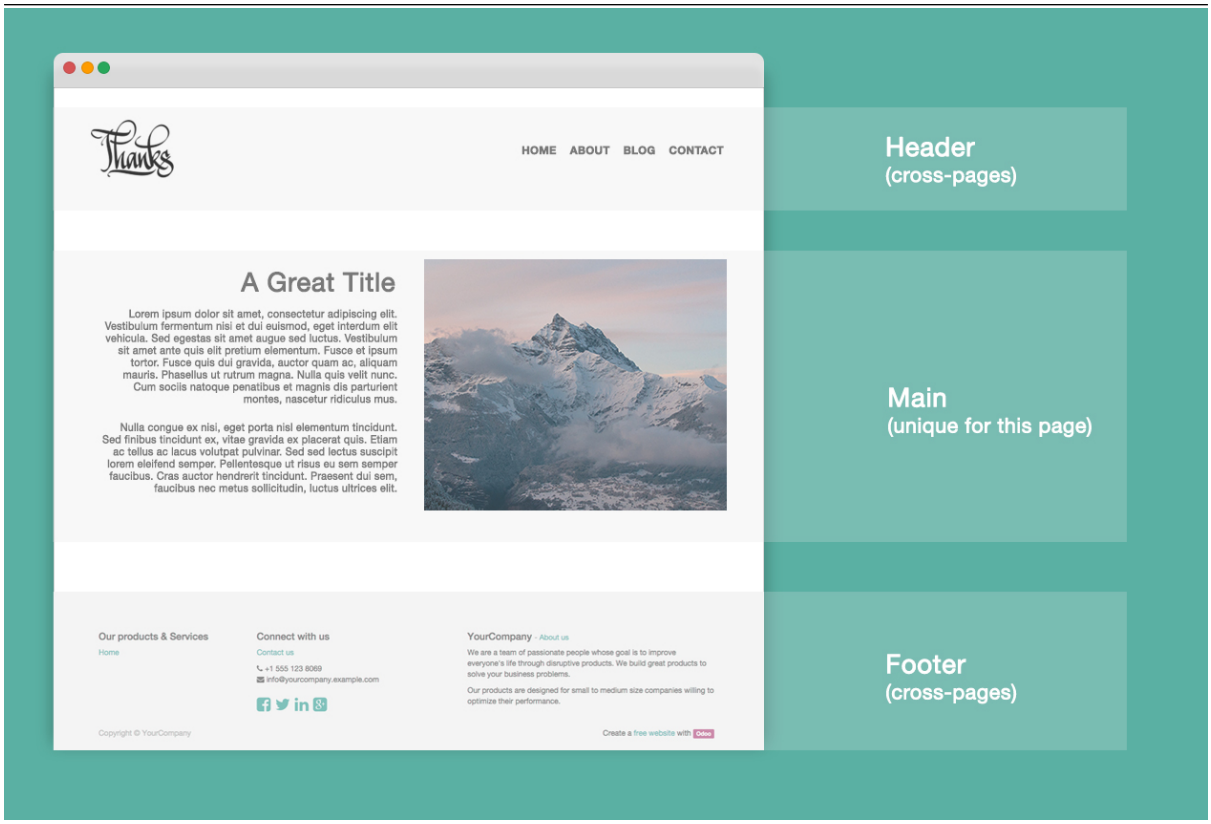
To install your theme, you just place your theme folder inside addons in your Odoo installation.

After that, navigate to the Settings page, look for your theme and click on the install button.

1.1.4 Structure of an Odoo page

An Odoo page is the visual result of a combination of 2 kind of elements, **cross-pages** and **unique**. By default, Odoo provides you with a **Header** and a **Footer** (cross-pages) and a unique main element that contains the content that makes your page unique.

Note: Cross-pages elements will be the same on every page. Unique elements are related to a specific page only.



To inspect the default layout, simply create a new page using the Website Builder. Click on *Content* → *New Page* and add a page name. Inspect the page using your browser.

```
<div id="wrapwrap">
  <header />
  <main />
  <footer />
</div>
```

Extend the default Header

By default, Odoo header contains a responsive navigation menu and the company's logo. You can easily add new elements or style the existing one.

To do so, create a **layout.xml** file in your **views** folder and add the default Odoo xml markup.

```
<?xml version="1.0" encoding="utf-8" ?>
<openerp>
  <data>
```

```
  </data>
</openerp>
```

Create a new template into the <data> tag, copy-pasting the following code.

```
<!-- Customize header  -->
<template id="custom_header" inherit_id="website.layout" name="Custom Header">

  <!-- Assign an id  -->
  <xpath expr="//div[@id='wrapwrap']/header" position="attributes">
    <attribute name="id">my_header</attribute>
  </xpath>

  <!-- Add an element after the top menu  -->
  <xpath expr="//div[@id='wrapwrap']/header/div" position="after">
    <div class="container">
      <div class="alert alert-info mt16" role="alert">
        <strong>Welcome</strong> in our website!
      </div>
    </div>
  </xpath>
</template>
```

The first xpath will add the id my_header to the header. It's the best option if you want to target css rules to that element and avoid these affecting other content on the page.

> **Warning:** Be careful replacing default elements attributes. As your theme will extend the default one, your changes will take priority in any future Odoo's update.

The second xpath will add a welcome message just after the navigation menu.

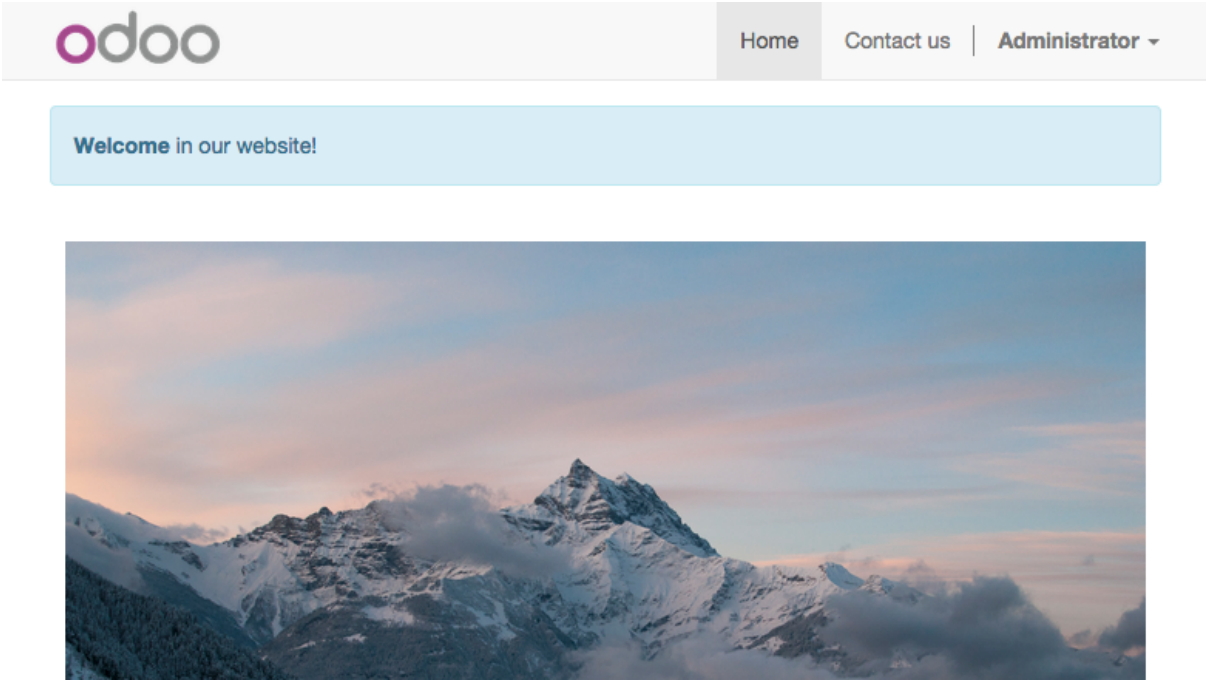
The last step is to add layout.xml to the list of xml files used by the theme. To do that, edit your __openerp__.py file like this

```
'data': [ 'views/layout.xml' ],
```

Update your theme



Great! We successfully added an id to the header and an element after the navigation menu. These changes will be applied to each page of the website.



1.1.5 Create a specific page layout

Imagine that we want to create a specific layout for a Services page. For this page, we need to add a list of services to the top and give the client the possibility of setting the rest of the page's layout using snippets.

Inside your *views* folder, create a **pages.xml** file and add the default Odoo markup. Inside `<data>` create a `<template>` tag, set the `page` attribute to `True` and add your code into it.

```
<?xml version="1.0" encoding="utf-8" ?>
<openerp>
  <data>
    <!-- === Services Page === -->
    <template name="Services page" id="website.services" page="True">
      <h1>Our Services</h1>
        <ul class="services">
          <li>Cloud Hosting</li>
          <li>Support</li>
          <li>Unlimited space</li>
        </ul>
      </template>
    </data>
  </openerp>
```

The page title will be the template ID. In our case *Services* (from `website.services`)

We successfully created a new page layout, but we haven't told the system **how to use it**. To do that, we can use **QWeb**. Wrap the html code into a `<t>` tag, like in this example.

```
<!-- === Services Page === -->
<template name="Services page" id="website.services" page="True">
  <t t-call="website.layout">
    <div id="wrap">
      <div class="container">
        <h1>Our Services</h1>
        <ul class="services">
          <li>Cloud Hosting</li>
          <li>Support</li>
          <li>Unlimited space</li>
        </ul>
```

```
      </div>
    </div>
  </t>
</template>
```

Using `<t t-call="website.layout">` we will extend the Odoo default page layout with our code.

As you can see, we wrapped our code into two `<div>`, one with ID `wrap` and the other one with class `container`. This is to provide a minimal layout.

The next step is to add an empty area that the user can fill with snippets. To achieve this, just create a `div` with `oe_structure` class just before closing the `div#wrap` element.

```
<?xml version="1.0" encoding="utf-8" ?>
<openerp>
<data>

<!-- === Services Page === -->
<template name="Services page" id="website.services" page="True">
  <t t-call="website.layout">
   <div id="wrap">
     <div class="container">
       <h1>Our Services</h1>
       <ul class="services">
         <li>Cloud Hosting</li>
         <li>Support</li>
         <li>Unlimited space</li>
       </ul>
       <!-- === Snippets' area === -->
       <div class="oe_structure" />
     </div>
   </div>
  </t>
</template>

</data>
</openerp>
```

Tip: You can create as many snippet areas as you like and place them anywhere in your pages.

Our page is almost ready. Now all we have to do is add **pages.xml** in our **__openerp__.py** file

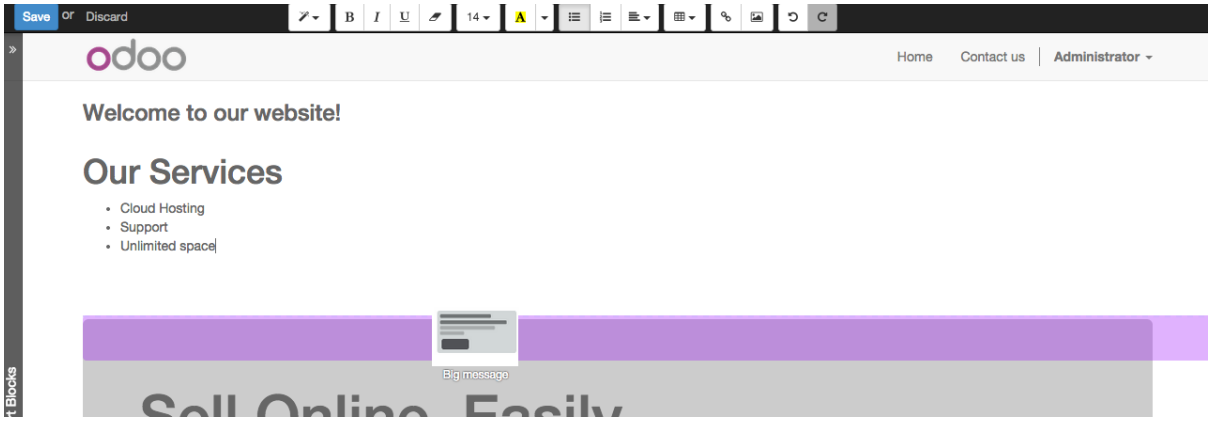
```
'data': [
  'views/layout.xml',
  'views/pages.xml'
],
```

Update your theme



Great, our Services page is ready and you'll be able to access it by navigating to `/yourwebsite/page/services`.

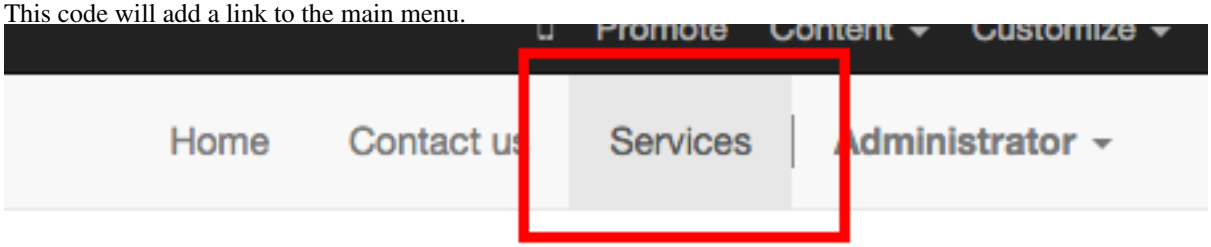
You will notice that it's possible to drag/drop snippets underneath the *Our Services* list.



Now let's go back to our *pages.xml* and, after our page template, copy/paste the following code.

```
<record id="services_page_link" model="website.menu">
  <field name="name">Services</field>
  <field name="url">/page/services</field>
  <field name="parent_id" ref="website.main_menu" />
  <field name="sequence" type="int">99</field>
</record>
```

This code will add a link to the main menu.



The **sequence** attribute defines the link's position in the top menu. In our example, we set the value to `99` in order to place it last. I you want to place it in a particular position, you have to replace the value according to your needs.

As you can see inspecting the *data.xml* file in the `website` module, the **Home** link is set to `10` and the **Contact** us one is set to `60` by default. If, for example, you want to place your link in the **middle**, you can set your link's sequence value to `40`.

1.1.6 Add Styles

Odoo includes Bootstrap by default. This means that you can take advantage of all Bootstrap styles and layout functionalities out of the box.

Of course Bootstrap is not enough if you want to provide a unique design. The following steps will guide you through how to add custom styles to your theme. The final result won't be pretty, but will provide you with enough information to build upon on your own.

Let's start by creating an empty file called **style.less** and place it in a folder called **less** in your static folder. The following rules will style our *Services* page. Copy and paste it, then save the file.

```
.services {
    background: #EAEAEA;
    padding: 1em;
    margin: 2em 0 3em;
    li {
        display: block;
        position: relative;
        background-color: #16a085;
```

```
        color: #FFF;
        padding: 2em;
        text-align: center;
        margin-bottom: 1em;
        font-size: 1.5em;
    }
}
```

Our file is ready but it is not included in our theme yet.

Let's navigate to the view folder and create an XML file called *assets.xml*. Add the default Odoo xml markup and copy/paste the following code. Remember to replace `theme folder` with your theme's main folder name.

```
<template id="mystyle" name="My style" inherit_id="website.assets_frontend">
    <xpath expr="link[last()]" position="after">
        <link href="/theme folder/static/less/style.less" rel="stylesheet" type="text/less"/>
    </xpath>
</template>
```

We just created a template specifying our less file. As you can see, our template has a special attribute called `inherit_id`. This attribute tells Odoo that our template is referring to another one in order to operate.

In this case, we are referring to `assets_frontend` template, located in the `website` module. `assets_frontend` specifies the list of assets loaded by the website builder and our goal is to add our less file to this list.

This can be achieved using xpath with the attributes `expr="link[last()]"` and `position="after"`, which means "*take my style file and place it after the last link in the list of the assets*".

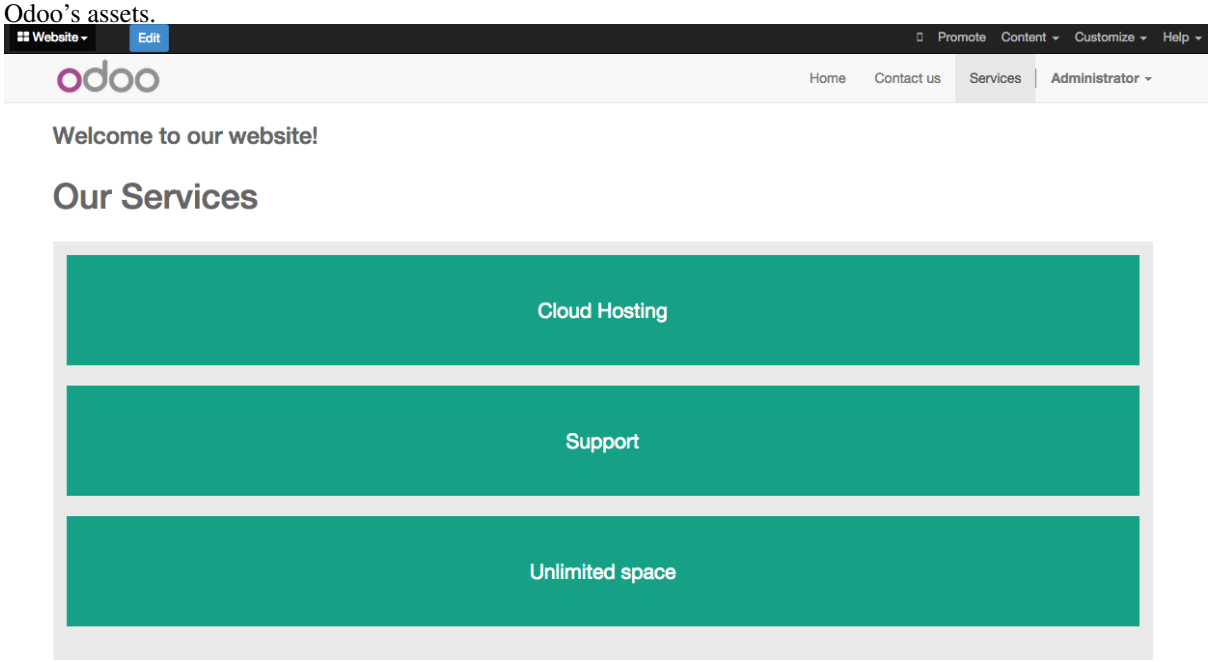
Placing it after the last one, we ensure that our file will be loaded at the end and take priority.

Finally add **assets.xml** in your **__openerp__.py** file.

Update your theme



Our less file is now included in our theme, it will be automatically compiled, minified and combined with all Odoo's assets.



1.1.7 Create Snippets

Since snippets are how users design and layout pages, they are the most important element of your design. Let's create a snippet for our Service page. The snippet will display three testimonials and it will be editable by the end user using the Website Builder UI. Navigate to the view folder and create an XML file called **snippets.xml**. Add the default Odoo xml markup and copy/paste the following code. The template contains the HTML markup that will be displayed by the snippet.

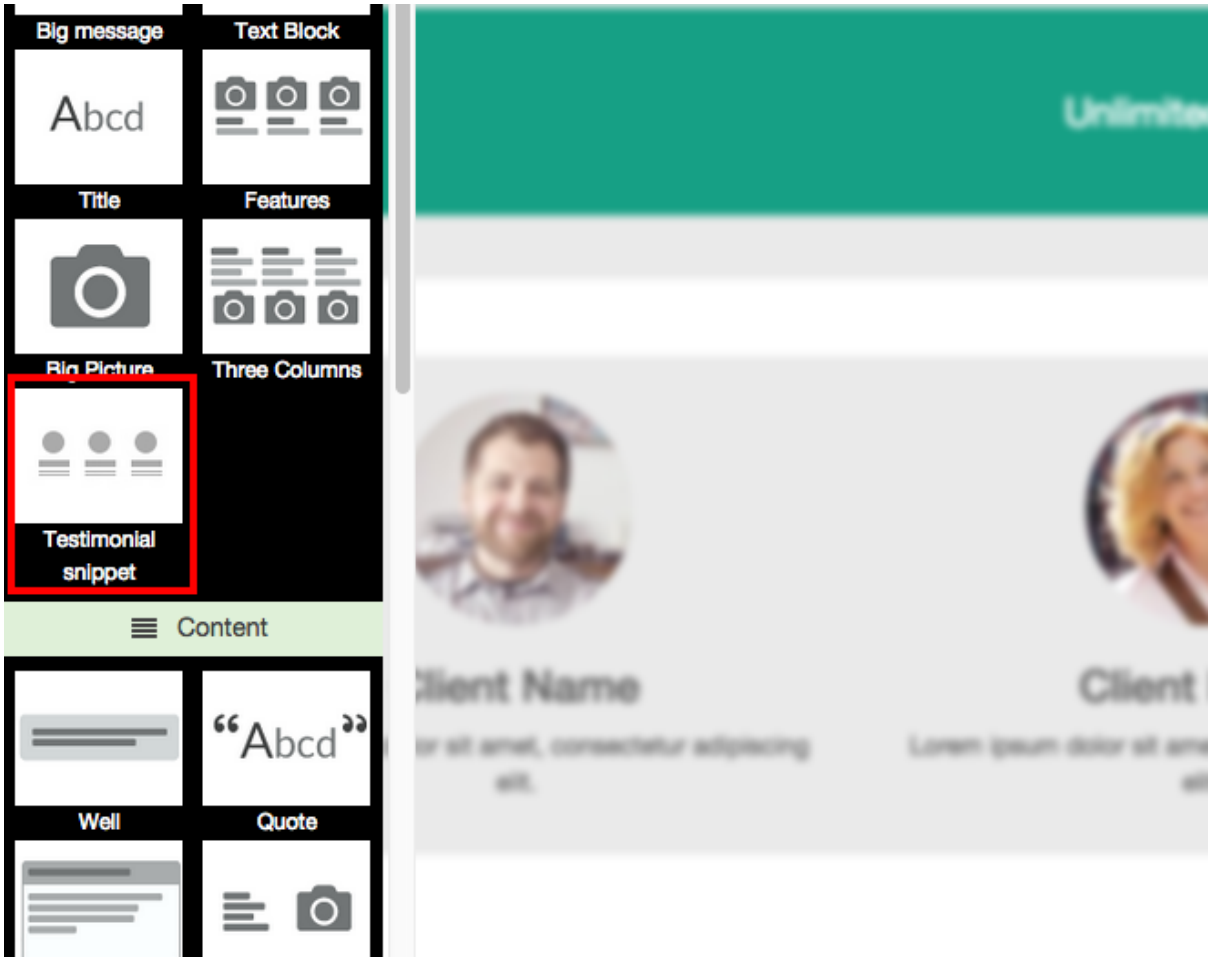
```
<template id="snippet_testimonial" name="Testimonial snippet">
  <section class="snippet_testimonial">
    <div class="container text-center">
      <div class="row">
        <div class="col-md-4">
          <img alt="client" class="img-circle" src="/theme_tutorial/static/src/img/client_1.jpg"/>
          <h3>Client Name</h3>
          <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
        </div>
        <div class="col-md-4">
          <img alt="client" class="img-circle" src="/theme_tutorial/static/src/img/client_2.jpg"/>
          <h3>Client Name</h3>
          <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
        </div>
        <div class="col-md-4">
          <img alt="client" class="img-circle" src="/theme_tutorial/static/src/img/client_3.jpg"/>
          <h3>Client Name</h3>
          <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.</p>
        </div>
      </div>
    </div>
  </section>
</template>
```

As you can see, we used Bootstrap default classes for our three columns. It's not just about layout, these classes **will be triggered by the Website Builder to make them resizable by the user**.

The previous code will create the snippet's content, but we still need to place it into the editor bar, so the user will be able to drag&drop it into the page. Copy/paste this template in your **snippets.xml** file.

```
<template id="place_into_bar" inherit_id="website.snippets" name="Place into bar">
  <xpath expr="//div[@id='snippet_structure']" position="inside">
    <t t-snippet="theme_tutorial.snippet_testimonial"
       t-thumbnail="/theme_tutorial/static/src/img/ui/snippet_thumb.jpg"/>
  </xpath>
</template>
```

Using xpath, we are targeting a particular element with id `snippet_structure`. This means that the snippet will appear in the Structure tab. If you want to change the destination tab, you have just to replace the `id` value in the xpath expression.



Tab Name	Xpath expression
Structure	`//div[@id='snippet_structure']`
Content	`//div[@id='snippet_content']`
Feature	`//div[@id='snippet_feature']`
Effect	`//div[@id='snippet_effect']`

The `<t>` tag will call our snippet's template and will assign a thumbnail placed in the img folder. You can now drag your snippet from the snippet bar, drop it in your page and see the result.



1.1.8 Snippet options

Options allow publishers to edit a snippet's appearance using the Website Builder's UI. Using Website Builder functionalities, you can create snippet options easily and automatically add them to the UI.

Options group properties

Options are wrapped in groups. Groups can have properties that define how the included options will interact with the user interface.

data-selector=" css selector(s) " Bind all the options included into the group to a particular element.

data-js=" custom method name " Is used to bind custom Javascript methods.

data-drop-in=" css selector(s) " Defines the list of elements where the snippet can be dropped into.

data-drop-near=" css selector(s) " Defines the list of elements that the snippet can be dropped beside.

Default option methods

Options apply standard CSS classes to the snippet. Depending on the method that you choose, the UI will behave differently.

data-select_class=" class name " More data-select_class in the same group defines a list of classes that the user can choose to apply. Only one option can be enabled at a time.

data-toggle_class=" class name " The data-toggle_class is used to apply one or more CSS classes from the list to a snippet. Multiple selections can be applied at once.

Let's demonstrate how default options work with a basic example.

We start by adding a new file in our views folder - name it **options.xml** and add the default Odoo XML markup. Create a new template copy/pasting the following

```
<template id="snippet_testimonial_opt" name="Snippet Testimonial Options" inherit_id="website_les
  <xpath expr="//div[@data-js='background']" position="after">
    <div data-selector=".snippet_testimonial"> <!-- Options group -->
      <li class="dropdown-submenu">
        <a href="#">Your Option</a>
        <ul class="dropdown-menu"> <!-- Options list -->
          <li data-select_class="opt_shadow"><a>Shadow Images</a></li>
          <li data-select_class="opt_grey_bg"><a>Grey Bg</a></li>
          <li data-select_class=""><a>None</a></li>
        </ul>
      </li>
    </div>
  </xpath>
 </template>
```

Note: The previous template will inherit the default **snippet_options template** adding our options after the **background** options (xpath expr attribute). To place your options in a particular order, inspect the **snippet_options template** from the **website module** and add your options before/after the desired position.

As you can see, we wrapped all our options inside a DIV tag that will group our options and that will target them to the right selector (`data-selector=".snippet_testimonial"`).

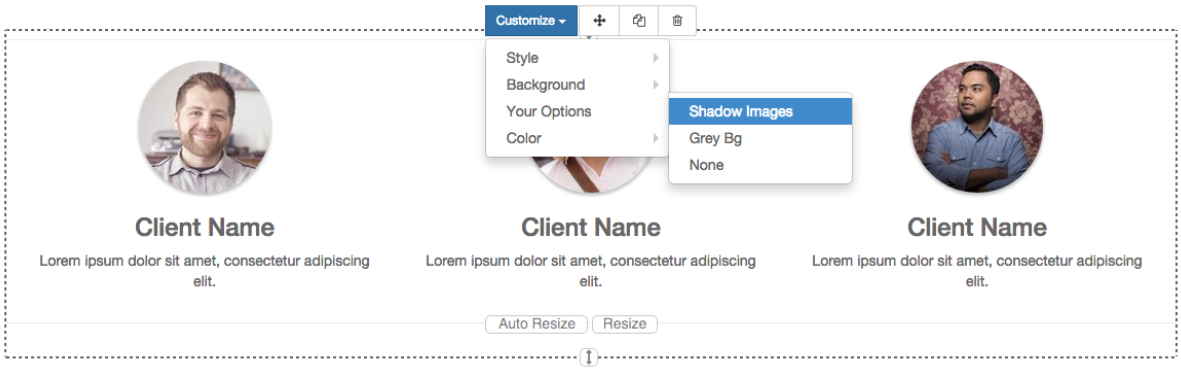
To define our options we applied `data-select_class` attributes to the `li` elements. When the user selects an option, the class contained in the attribute will automatically be applied to the element.

Since `select_class` method avoids multiple selections, the last "empty" option will reset the snippet to default.

Add **options.xml** to `__openerp__.py` and update your theme.



Dropping our snippet onto the page, you will notice that our new options are automatically added to the customize menu. Inspecting the page, you will also notice that the class will be applied to the element when selecting an option.



Let's create some css rules in order to provide a visual feedback for our options. Open our **style.less** file and add the following

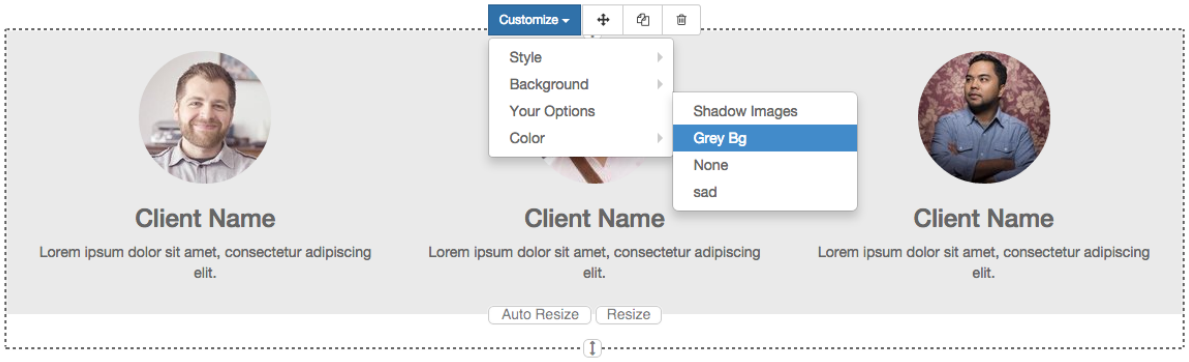
```
.snippet_testimonial {
  border: 1px solid #EAEAEA;
  padding: 20px;
}

// These lines will add a default style for our snippet. Now let's create our custom rules for the

.snippet_testimonial {
  border: 1px solid #EAEAEA;
  padding: 20px;

  &.opt_shadow img {
    box-shadow: 0 2px 5px rgba(51, 51, 51, 0.4);
  }

  &.opt_grey_bg {
    border: none;
    background-color: #EAEAEA;
  }
}
```



Great! We successfully created options for our snippet.

Any time the publisher clicks on an option, the system will add the class specified in the data-select_class attribute.

By replacing `data-select_class` with `data-toggle_class` you will be able to select more classes at the same time.

Javascript Options

`data-select_class` and `data-toggle_class` are great if you need to perform simple class change operations. But what if your snippet's customization needs something more?

As we said before, `data-js` propriety can be assigned to an options group in order to define a custom method. Let's create one for our *testimonials snippet* by adding a `data-js` attribute to the option's group div that we created earlier.

```
<div data-js="snippet_testimonial_options" data-selector=".snippet_testimonial">
  [...]
</div>
```

Done. From now on, the Website Builder will look for a `snippet_testimonial_options` method each time the publisher enters in edit mode.

Let's go one step further by creating a javascript file, name it **tutorial_editor.js** and place it into the **static** folder. Copy/paste the following code

```
(function() {
    'use strict';
    var website = openerp.website;
    website.openerp_website = {};
})();
```

Great, we successfully created our javascript editor file. This file will contain all the javascript functions used by our snippets in edit mode. Let's create a new function for our testimonial snippet using the `snippet_testimonial_options` method that we created before.

```
(function() {
    'use strict';
    var website = openerp.website;
    website.openerp_website = {};

    website.snippet.options.snippet_testimonial_options = website.snippet.Option.extend({
        on_focus: function() {
            alert("On focus!");
        }
    })
})();
```

As you will notice, we used a method called `on_focus` to trigger our function. The Website Builder provides several events you can use to trigger your custom functions.

Event	Description
`start`	Fires when the publisher selects the snippet for the first time in an editing session or when the snippet is drag-dropped into the page
`on_focus`	Fires each time the snippet is selected by the user or when the snippet is drag-dropped into the page.
`on_blur`	This event occurs when a snippet loses focus.
`on_clone`	Fires just after a snippet is duplicated. A new js variable is created ($clone) containing the cloned element.
`on_remove`	It occurs just before that the snippet is removed.
`drop_and_build_snippet`	Fires just after that the snippet is drag and dropped into a drop zone. When this event is triggered, the content is already inserted in the page.
`clean_for_save`	It trigger before the publisher save the page.

Let's add our new javascript files to the editor assets list. Go back to **assets.xml** and create a new template like the previous one. This time we have to inherit `assets_editor` instead of `assets_frontend`.

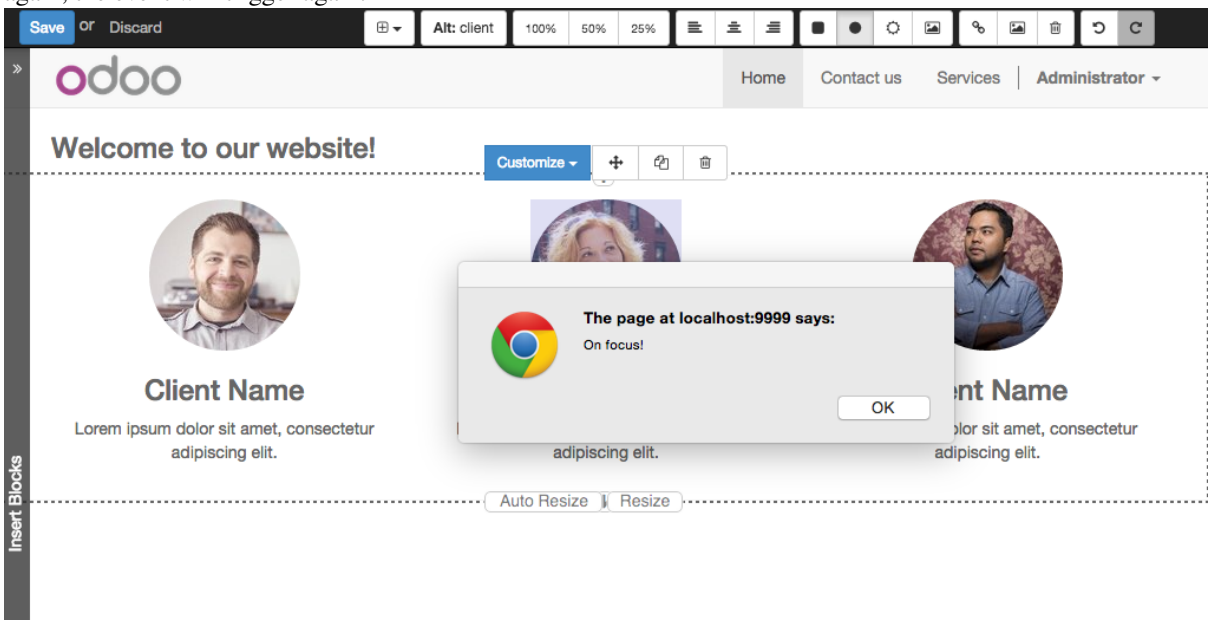
```
<template id="my_js" inherit_id="website_less.assets_editor" name="My Js">
  <xpath expr="script[last()]" position="after">
    <script type="text/javascript" src="/theme_tutorial/static/src/js/tutorial_editor.js" />
```

```
  </xpath>
</template>
```

Update your theme



Let's test our new javascript function. Enter in Edit mode and drop into the page. You should now see the javascript alert that we bound on the `on_focus` event. If you close it, then click outside of your snippet and then click in it again, the event will trigger again.



1.1.9 Editing Reference Guide

Basically all the elements in a page can be edited by the publisher. Besides that, some element types and css classes will trigger special Website Builder functionalities when edited.

Layout

`<section />` Any section element can be edited like a block of content. The publisher can move or duplicate it. It's also possible to set a background image or color. Section is the standard main container of any snippet.

`.row > .col-md-*` Any medium bootstrap columns directly descending from a .row element, will be resizable by the publisher.

`contenteditable="False"` This attribute will prevent editing to the element and all its children.

`contenteditable="True"` Apply it to an element inside a contenteditable="False" element in order to create an exception and make the element and its children editable.

`<a href="#" />` In Edit Mode, any link can be edited and styled. Using the "Link Modal" it's also possible to replace it with a button.

Media

`<span class="fa" />` Pictogram elements. Editing this element will open the Pictogram library to replace the icon. It's also possible to transform the elements using CSS.

<img /> Once clicked, the Image Library will open and you can replace images. Transformation is also possible for this kind of element.

```
<div class="media_iframe_video" data-src="[your url]" >
  <div class="css_editable_mode_display"/>
  <div class="media_iframe_video_size"/>
  <iframe src="[your url]"/>
</div>
```

This html structure will create an `<iframe>` element editable by the publisher.

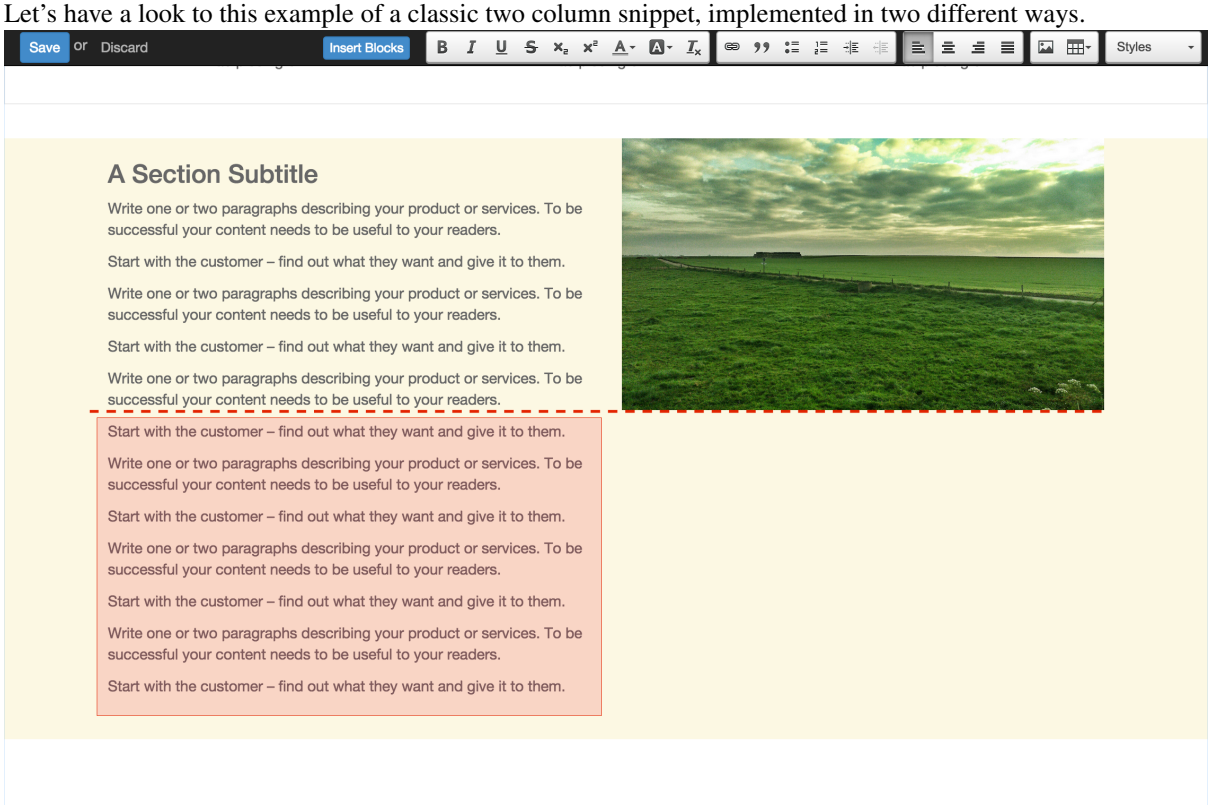
1.1.10 SEO best practice

Facilitate content insertion

Modern search engine algorithms increasingly focus on content, which means there is less focus on **keyword saturation** and more focus on whether or not the content is **actually relevant to the keywords**.

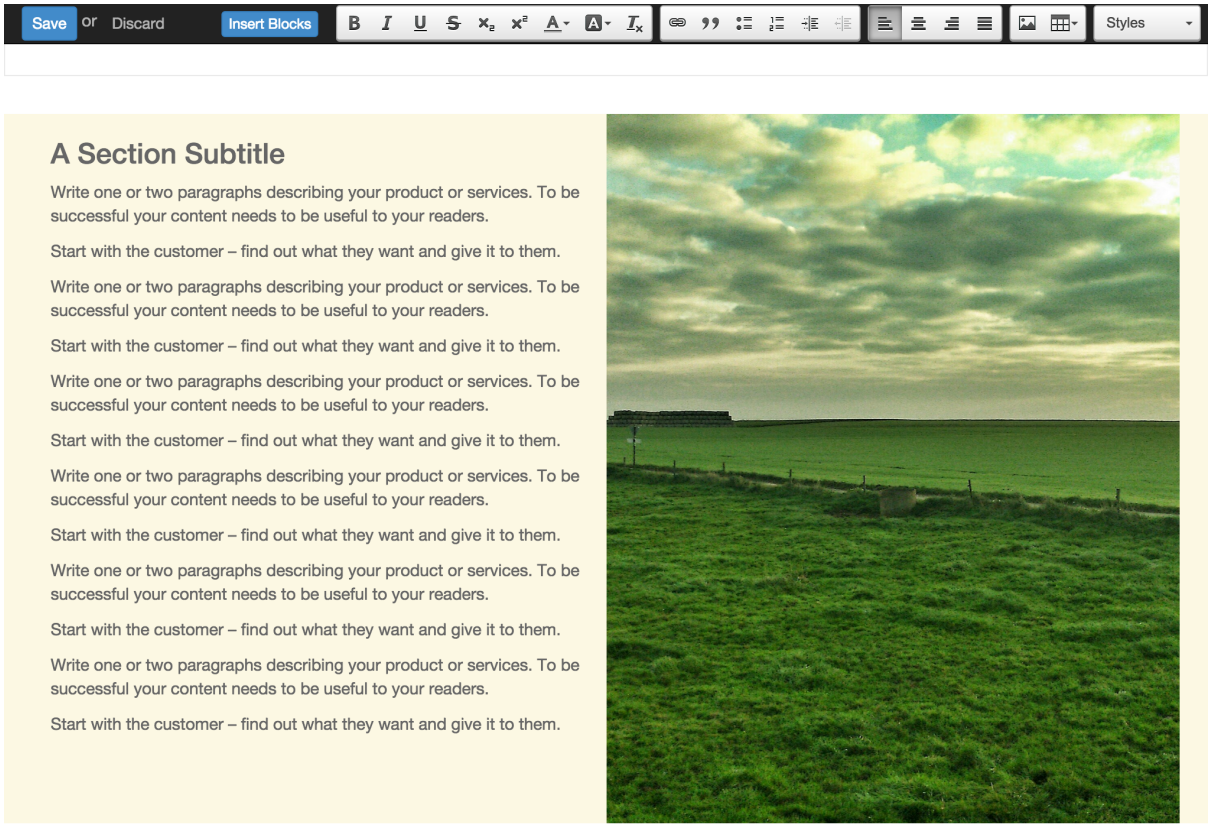
As content is so important for SEO, you should concentrate on giving publishers the tools to easily insert it. It is important that your snippets are "content-responsive", meaning that they should fit the publisher's content regardless of size.

Let's have a look to this example of a classic two column snippet, implemented in two different ways.



Bad

Using fixed image, the publisher will be forced to limit the text in order to follow the layout.



Good

Using background images that fit the column height, the publisher will be free to add the content regardless of the image's height.

Page segmentation

Basically, page segmentation means that a page is divided into several separate parts and these parts are treated as separate entries by search engines. When you design pages or snippets, you should be sure to use the right tags in order to facilitate search engine indexing.

<article> Specifies an independent block of content. Within it should be a piece of self-contained content that should make sense on its own. You can nest `<article>` elements within one another. In this case, it's implied that the nested elements are related to the outer `<article>` element.

<header> Indicates the header section of a self-contained block of content (an `<article>`).

<section> Is the snippet default tag and it specifies a subsection of a block of content. It can be used to split `<article>` content into several parts. It's advisable to use a heading element (`<h1>` – `<h6>`) to define the section's topic.

<hgroup> Is used to wrap a section of headings (`<h1>` - `<h6>`). A great example would be an article with both a headline and sub-headline at the top:

```
<hgroup>
  <h1>Main Title</h1>
  <h2>Subheading</h2>
</hgroup>
```

Describe your page

Define keywords

You should use appropriate, relevant keywords and synonyms for those keywords. You can define them for each page using the built-in "Promote" function found in the bar at the top.

Define a title and a description

Define them using the "Promote" function. Keep your page titles short and include the main keyword phrase for the page. Good titles evoke an emotional response, ask a question or promise something.

Descriptions, while not important to search engine rankings, are extremely important in gaining user click-through. These are an opportunity to advertise content and to let people searching know exactly whether the given page contains the information they're looking for. It is important that titles and descriptions on each page are unique.

1.2 Building a Website

Warning:
- This guide assumes basic knowledge of Python
- This guide assumes *an installed Odoo*

1.2.1 Creating a basic module

In Odoo, tasks are performed by creating modules.

Modules customize the behavior of an Odoo installation, either by adding new behaviors or by altering existing ones (including behaviors added by other modules).

Odoo's scaffolding can setup a basic module. To quickly get started simply invoke:

```
$ ./odoo.py scaffold Academy my-modules
```

This will automatically create a `my-modules` *module directory* with an `academy` module inside. The directory can be an existing module directory if you want, but the module name must be unique within the directory.

1.2.2 A demonstration module

We have a "complete" module ready for installation.

Although it does absolutely nothing we can install it:

- start the Odoo server

```
$ ./odoo.py --addons-path addons,my-modules
```

- go to http://localhost:8069
- create a new database including demonstration data
- to go *Settings* → *Modules* → *Local Modules*
- in the top-right corner remove the *Installed* filter and search for *academy*
- click the *Install* button for the *Academy* module

1.2.3 To the browser

Controllers interpret browser requests and send data back.

Add a simple controller and ensure it is imported by `__init__.py` (so Odoo can find it): *academy/controllers.py*

```
# -*- coding: utf-8 -*-
from openerp import http

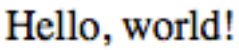
class Academy(http.Controller):
    @http.route('/academy/academy/', auth='public')
    def index(self, **kw):
        return "Hello, world"

#     @http.route('/academy/academy/objects/', auth='public')
#     def list(self, **kw):
```

Shut down your server (`^C`) then restart it:

```
$ ./odoo.py --addons-path addons,my-modules
```

and open a page to http://localhost:8069/academy/academy/, you should see your "page" appear:



1.2.4 Templates

Generating HTML in Python isn't very pleasant.

The usual solution is templates, pseudo-documents with placeholders and display logic. Odoo allows any Python templating system, but provides its own *QWeb* templating system which integrates with other features.

Create a template and ensure the template file is registered in the `__openerp__.py` manifest, and alter the controller to use our template: *academy/controllers.py*

```
class Academy(http.Controller):
    @http.route('/academy/academy/', auth='public')
    def index(self, **kw):
        return http.request.render('academy.index', {
            'teachers': ["Diana Padilla", "Jody Caroll", "Lester Vaughn"],
        })

#     @http.route('/academy/academy/objects/', auth='public')
#     def list(self, **kw):
```

academy/templates.xml

```
<openerp>
    <data>
        <template id="index">
            <title>Academy</title>
            <t t-foreach="teachers" t-as="teacher">
```

```
            <p><t t-esc="teacher"/></p>
        </t>
    </template>
    <!-- <template id="object"> -->
    <!--   <h1><t t-esc="object.display_name"/></h1> -->
    <!--   <dl> -->
```

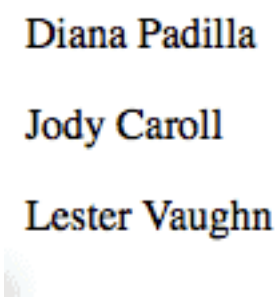
The templates iterates (`t-foreach`) on all the teachers (passed through the *template context*), and prints each teacher in its own paragraph.

Finally restart Odoo and update the module's data (to install the template) by going to *Settings* → *Modules* → *Local Modules* → *Academy* and clicking *Upgrade*.

Tip: Alternatively, Odoo can be restarted *and update modules at the same time*:

```
$ odoo.py --addons-path addons,my-modules -d academy -u academy
```

Going to http://localhost:8069/academy/academy/ should now result in:



1.2.5 Storing data in Odoo

Odoo models map to database tables.

In the previous section we just displayed a list of string entered statically in the Python code. This doesn't allow modifications or persistent storage so we'll now move our data to the database.

Defining the data model

Define a teacher model, and ensure it is imported from `__init__.py` so it is correctly loaded: *academy/models.py*

```
from openerp import models, fields, api

class Teachers(models.Model):
    _name = 'academy.teachers'

    name = fields.Char()
```

Then setup *basic access control* for the model and add them to the manifest: *academy/__openerp__.py*

```
    # always loaded
    'data': [
        'security/ir.model.access.csv',
        'templates.xml',
    ],
    # only loaded in demonstration mode
```

academy/security/ir.model.access.csv

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_academy_teachers,access_academy_teachers,model_academy_teachers,,1,0,0,0
```

this simply gives read access (`perm_read`) to all users (`group_id:id` left empty).

Note: *Data files* (XML or CSV) must be added to the module manifest, Python files (models or controllers) don't but have to be imported from `__init__.py` (directly or indirectly)

Warning: the administrator user bypasses access control, they have access to all models even if not given access

Demonstration data

The second step is to add some demonstration data to the system so it's possible to test it easily. This is done by adding a `demo` *data file*, which must be linked from the manifest: *academy/demo.xml*

```
<openerp>
    <data>
        <record id="padilla" model="academy.teachers">
            <field name="name">Diana Padilla</field>
        </record>
        <record id="carroll" model="academy.teachers">
            <field name="name">Jody Carroll</field>
        </record>
        <record id="vaughn" model="academy.teachers">
            <field name="name">Lester Vaughn</field>
        </record>
    </data>
</openerp>
```

Tip: *Data files* can be used for demo and non-demo data. Demo data are only loaded in "demonstration mode" and can be used for flow testing and demonstration, non-demo data are always loaded and used as initial system setup.

In this case we're using demonstration data because an actual user of the system would want to input or import their own teachers list, this list is only useful for testing.

Accessing the data

The last step is to alter model and template to use our demonstration data:

1. fetch the records from the database instead of having a static list
2. Because *`search()`* returns a set of records matching the filter ("all records" here), alter the template to print each teacher's `name`

academy/controllers.py

```
class Academy(http.Controller):
    @http.route('/academy/academy/', auth='public')
    def index(self, **kw):
        Teachers = http.request.env['academy.teachers']
        return http.request.render('academy.index', {
            'teachers': Teachers.search([])
        })

#     @http.route('/academy/academy/objects/', auth='public')
```

academy/templates.xml

```
        <template id="index">
            <title>Academy</title>
            <t t-foreach="teachers" t-as="teacher">
                <p><t t-esc="teacher.id"/> <t t-esc="teacher.name"/></p>
            </t>
        </template>
        <!-- <template id="object"> -->
```

Restart the server and update the module (in order to update the manifest and templates and load the demo file) then navigate to http://localhost:8069/academy/academy/. The page should look slightly different: names should simply be prefixed by a number (the database identifier for the teacher).

1.2.6 Website support

Odoo bundles a module dedicated to building websites.

So far we've used controllers fairly directly, but Odoo 8 added deeper integration and a few other services (e.g. default styling, theming) via the `website` module.

1. first, add `website` as a dependency to `academy`
2. then add the `website=True` flag on the controller, this sets up a few new variables on *the request object* and allows using the website layout in our template
3. use the website layout in the template

academy/__openerp__.py

```
    'version': '0.1',

    # any module necessary for this one to work correctly
    'depends': ['website'],

    # always loaded
    'data': [
```

academy/controllers.py

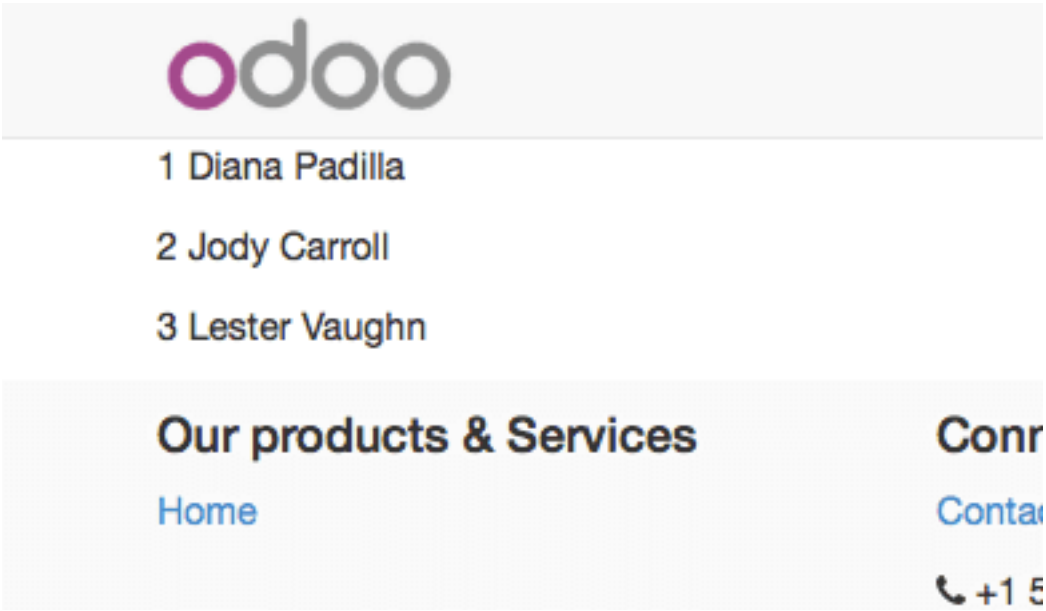
```
from openerp import http

class Academy(http.Controller):
    @http.route('/academy/academy/', auth='public', website=True)
    def index(self, **kw):
        Teachers = http.request.env['academy.teachers']
        return http.request.render('academy.index', {
```

academy/templates.xml

```
<openerp>
    <data>
        <template id="index">
            <t t-call="website.layout">
                <t t-set="title">Academy</t>
                <div class="oe_structure">
                    <div class="container">
                        <t t-foreach="teachers" t-as="teacher">
                            <p><t t-esc="teacher.id"/> <t t-esc="teacher.name"/></p>
                        </t>
                    </div>
                </div>
            </t>
        </template>
        <!-- <template id="object"> -->
```

After restarting the server while updating the module (in order to update the manifest and template) access http://localhost:8069/academy/academy/ should yield a nicer looking page with branding and a number of built-in page elements (top-level menu, footer, ...)



The website layout also provides support for edition tools: click *Sign In* (in the top-right), fill the credentials in (`admin` / `admin` by default) then click *Log In*.

You're now in Odoo "proper": the administrative interface. For now click on the *Website* menu item (top-left corner.

We're back in the website but as an administrator, with access to advanced edition features provided by the *website* support:

- a template code editor (*Customize* → *HTML Editor*) where you can see and edit all templates used for the current page
- the *Edit* button in the top-left switches to "edition mode" where blocks (snippets) and rich text edition are available
- a number of other features such as mobile preview or SEO (Search Engine Optimization)

1.2.7 URLs and routing

Controller methods are associated with *routes* via the `route()` decorator which takes a routing string and a number of attributes to customise its behavior or security.

We've seen a "literal" routing string, which matches a URL section exactly, but routing strings can also use converter patterns which match bits of URLs and make those available as local variables. For instance we can create a new controller method which takes a bit of URL and prints it out: *academy/controllers.py*

```
            'teachers': Teachers.search([])
        })

    @http.route('/academy/<name>/', auth='public', website=True)
    def teacher(self, name):
        return '<h1>{}</h1>'.format(name)


#     @http.route('/academy/academy/objects/', auth='public')
#     def list(self, **kw):
#         return http.request.render('academy.listing', {
```

restart Odoo, access http://localhost:8069/academy/Alice/ and http://localhost:8069/academy/Bob/ and see the difference.

As the name indicates, converter patterns don't just do extraction, they also do *validation* and *conversion*, so we can change the new controller to only accept integers: *academy/controllers.py*

```
            'teachers': Teachers.search([])
        })

    @http.route('/academy/<int:id>/', auth='public', website=True)
    def teacher(self, id):
        return '<h1>{} ({})</h1>'.format(id, type(id).__name__)


#     @http.route('/academy/academy/objects/', auth='public')
```

Restart Odoo, access http://localhost:8069/academy/2, note how the old value was a string, but the new one was converted to an integers. Try accessing http://localhost:8069/academy/Carol/ and note that the page was not found: since "Carol" is not an integer, the route was ignored and no route could be found.

Odoo provides an additional converter called `model` which provides records directly when given their id. Let's use this to create a generic page for teacher biographies: *academy/controllers.py*

```
            'teachers': Teachers.search([])
        })

    @http.route('/academy/<model("academy.teachers"):teacher>/', auth='public', website=True)
    def teacher(self, teacher):
        return http.request.render('academy.biography', {
            'person': teacher
        })


#     @http.route('/academy/academy/objects/', auth='public')
```

academy/templates.xml

```
                </div>
            </t>
        </template>
        <template id="biography">
            <t t-call="website.layout">
                <t t-set="title">Academy</t>
                <div class="oe_structure"/>
                <div class="oe_structure">
                    <div class="container">
                        <p><t t-esc="person.id"/> <t t-esc="person.name"/></p>
                    </div>
                </div>
                <div class="oe_structure"/>
            </t>
        </template>
        <!-- <template id="object"> -->
        <!--   <h1><t t-esc="object.display_name"/></h1> -->
        <!--   <dl> -->
```

then change the list of model to link to our new controller: *academy/templates.xml*

```
                <div class="oe_structure">
                    <div class="container">
                        <t t-foreach="teachers" t-as="teacher">
                            <p><a t-attf-href="/academy/{{ slug(teacher) }}">
                              <t t-esc="teacher.name"/></a>
                            </p>
                        </t>
```

```
            </div>
        </div>
```

```
        <div class="oe_structure"/>
        <div class="oe_structure">
            <div class="container">
                <h3><t t-esc="person.name"/></h3>
            </div>
        </div>
        <div class="oe_structure"/>
```

Restart Odoo and upgrade the module, then you can visit each teacher's page. As an exercise, try adding blocks to a teacher's page to write a biography, then go to another teacher's page and so forth. You will discover, that your biography is shared between all teachers, because blocks are added to the *template*, and the *biography* template is shared between all teachers, when one page is edited they're all edited at the same time.

1.2.8 Field edition

Data which is specific to a record should be saved on that record, so let us add a new biography field to our teachers: *academy/models.py*

```
    _name = 'academy.teachers'

    name = fields.Char()
    biography = fields.Html()
```

academy/templates.xml

```
        <div class="oe_structure">
            <div class="container">
                <h3><t t-esc="person.name"/></h3>
                <div><t t-esc="person.biography"/></div>
            </div>
        </div>
        <div class="oe_structure"/>
```

Restart Odoo and update the views, reload the teacher's page and... the field is invisible since it contains nothing.

For record fields, templates can use a special `t-field` directive which allows editing the field content from the website using field-specific interfaces. Change the *person* template to use `t-field`: *academy/templates.xml*

```
        <div class="oe_structure"/>
        <div class="oe_structure">
            <div class="container">
                <h3 t-field="person.name"/>
                <div t-field="person.biography"/>
            </div>
        </div>
        <div class="oe_structure"/>
```

Restart Odoo and upgrade the module, there is now a placeholder under the teacher's name and a new zone for blocks in *Edit* mode. Content dropped there is stored in the corresponding teacher's `biography` field, and thus specific to that teacher.

The teacher's name is also editable, and when saved the change is visible on the index page.

`t-field` can also take formatting options which depend on the exact field. For instance if we display the modification date for a teacher's record: *academy/templates.xml*

```
        <div class="oe_structure">
            <div class="container">
                <h3 t-field="person.name"/>
                <p>Last modified: <i t-field="person.write_date"/></p>
```

```
                    <div t-field="person.biography"/>
                </div>
            </div>
```

it is displayed in a very "computery" manner and hard to read, but we could ask for a human-readable version: *academy/templates.xml*

```
            <div class="oe_structure">
                <div class="container">
                    <h3 t-field="person.name"/>
                    <p>Last modified: <i t-field="person.write_date" t-field-options='{"format
                    <div t-field="person.biography"/>
                </div>
            </div>
```

or a relative display: *academy/templates.xml*

```
            <div class="oe_structure">
                <div class="container">
                    <h3 t-field="person.name"/>
                    <p>Last modified: <i t-field="person.write_date" t-field-options='{"widget
                    <div t-field="person.biography"/>
                </div>
            </div>
```

1.2.9 Administration and ERP integration

A brief and incomplete introduction to the Odoo administration

The Odoo administration was briefly seen during the *website support* section. We can go back to it using *Administrator* → *Administrator* in the menu (or *Sign In* if you're signed out).

The conceptual structure of the Odoo backend is simple:

1. first are menus, a tree (menus can have sub-menus) of records. Menus without children map to…
2. actions. Actions have various types: links, reports, code which Odoo should execute or data display. Data display actions are called *window actions*, and tell Odoo to display a given *model* according to a set of views…
3. a view has a type, a broad category to which it corresponds (a list, a graph, a calendar) and an *architecture* which customises the way the model is displayed inside the view.

Editing in the Odoo administration

By default, an Odoo model is essentially invisible to a user. To make it visible it must be available through an action, which itself needs to be reachable, generally through a menu.

Let's create a menu for our model: *academy/__openerp__.py*

```
    'data': [
        'security/ir.model.access.csv',
        'templates.xml',
        'views.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
```

academy/views.xml

```
<openerp>
 <data>
  <record id="action_academy_teachers" model="ir.actions.act_window">
    <field name="name">Academy teachers</field>
    <field name="res_model">academy.teachers</field>
  </record>

  <menuitem sequence="0" id="menu_academy" name="Academy"/>
  <menuitem id="menu_academy_content" parent="menu_academy"
            name="Academy Content"/>
  <menuitem id="menu_academy_content_teachers"
            parent="menu_academy_content"
            action="action_academy_teachers"/>
```

then accessing http://localhost:8069/web/ in the top left should be a menu *Academy*, which is selected by default, as it is the first menu, and having opened a listing of teachers. From the listing it is possible to *Create* new teacher records, and to switch to the "form" by-record view.

If there is no definition of how to present records (a *view*) Odoo will automatically create a basic one on-the-fly. In our case it works for the "list" view for now (only displays the teacher's name) but in the "form" view the HTML `biography` field is displayed side-by-side with the `name` field and not given enough space. Let's define a custom form view to make viewing and editing teacher records a better experience: *academy/views.xml*

```
    <field name="name">Academy teachers</field>
    <field name="res_model">academy.teachers</field>
  </record>

  <record id="academy_teacher_form" model="ir.ui.view">
    <field name="name">Academy teachers: form</field>
    <field name="model">academy.teachers</field>
    <field name="arch" type="xml">
      <form>
        <sheet>
          <label for="name"/> <field name="name"/>
          <label for="biography"/>
          <field name="biography"/>
        </sheet>
      </form>
    </field>
  </record>

  <menuitem sequence="0" id="menu_academy" name="Academy"/>
  <menuitem id="menu_academy_content" parent="menu_academy"
```

Relations between models

We have seen a pair of "basic" fields stored directly in the record. There are *a number of basic fields*. The second broad categories of fields are *relational* and used to link records to one another (within a model or across models).

For demonstration, let's create a *courses* model. Each course should have a `teacher` field, linking to a single teacher record, but each teacher can teach many courses: *academy/models.py*

```
    name = fields.Char()
    biography = fields.Html()

class Courses(models.Model):
    _name = 'academy.courses'

    name = fields.Char()
    teacher_id = fields.Many2one('academy.teachers', string="Teacher")
```

academy/security/ir.model.access.csv

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_academy_teachers,access_academy_teachers,model_academy_teachers,,1,0,0,0
access_academy_courses,access_academy_courses,model_academy_courses,,1,0,0,0
```

let's also add views so we can see and edit a course's teacher: *academy/views.xml*

```
      </form>
    </field>
  </record>

  <record id="action_academy_courses" model="ir.actions.act_window">
    <field name="name">Academy courses</field>
    <field name="res_model">academy.courses</field>
  </record>
  <record id="academy_course_search" model="ir.ui.view">
    <field name="name">Academy courses: search</field>
    <field name="model">academy.courses</field>
    <field name="arch" type="xml">
      <search>
        <field name="name"/>
        <field name="teacher_id"/>
      </search>
    </field>
  </record>
  <record id="academy_course_list" model="ir.ui.view">
    <field name="name">Academy courses: list</field>
    <field name="model">academy.courses</field>
    <field name="arch" type="xml">
      <tree string="Courses">
        <field name="name"/>
        <field name="teacher_id"/>
      </tree>
    </field>
  </record>
  <record id="academy_course_form" model="ir.ui.view">
    <field name="name">Academy courses: form</field>
    <field name="model">academy.courses</field>
    <field name="arch" type="xml">
      <form>
        <sheet>
          <label for="name"/>
          <field name="name"/>
          <label for="teacher_id"/>
          <field name="teacher_id"/>
        </sheet>
      </form>
    </field>
  </record>

  <menuitem sequence="0" id="menu_academy" name="Academy"/>
  <menuitem id="menu_academy_content" parent="menu_academy"
            name="Academy Content"/>
  <menuitem id="menu_academy_content_courses"
            parent="menu_academy_content"
            action="action_academy_courses"/>
  <menuitem id="menu_academy_content_teachers"
            parent="menu_academy_content"
```

It should also be possible to create new courses directly from a teacher's page, or to see all the courses they teach, so add *the inverse relationship* to the *teachers* model: *academy/models.py*

```
    name = fields.Char()
    biography = fields.Html()
```

```
    course_ids = fields.One2many('academy.courses', 'teacher_id', string="Courses")

class Courses(models.Model):
    _name = 'academy.courses'
```

academy/views.xml

```
    <field name="arch" type="xml">
      <form>
        <sheet>
          <label for="name"/> <field name="name"/>

          <label for="biography"/>
          <field name="biography"/>

          <field name="course_ids">
            <tree string="Courses" editable="bottom">
              <field name="name"/>
            </tree>
          </field>
        </sheet>
      </form>
```

Discussions and notifications

Odoo provides technical models, which don't directly fulfill business needs but which add capabilities to business objects without having to build them by hand.

One of these is the *Chatter* system, part of Odoo's email and messaging system, which can add notifications and discussion threads to any model. The model simply has to `_inherit` `mail.thread`, and add the `message_ids` field to its form view to display the discussion thread. Discussion threads are per-record.

For our academy, it makes sense to allow discussing courses to handle e.g. scheduling changes or discussions between teachers and assistants: *academy/models.py*

```
class Courses(models.Model):
    _name = 'academy.courses'
    _inherit = 'mail.thread'

    name = fields.Char()
    teacher_id = fields.Many2one('academy.teachers', string="Teacher")
```

academy/views.xml

```
          <field name="name"/>
          <label for="teacher_id"/>
          <field name="teacher_id"/>
        </sheet>
        <div class="oe_chatter">
          <field name="message_follower_ids" widget="mail_followers"/>
          <field name="message_ids" widget="mail_thread"/>
        </div>
      </form>
    </field>
```

At the bottom of each course form, there is now a discussion thread and the possibility for users of the system to leave messages and follow or unfollow discussions linked to specific courses.

Selling courses

Odoo also provides business models which allow using or opting in business needs more directly. For instance the `website_sale` module sets up an e-commerce site based on the products in the Odoo system. We can easily make course subscriptions sellable by making our courses specific kinds of products.

Rather than the previous classical inheritance, this means replacing our *course* model by the *product* model, and extending products in-place (to add anything we need to it).

First of all we need to add a dependency on `website_sale` so we get both products (via `sale`) and the ecommerce interface: *academy/__openerp__.py*

```
    'version': '0.1',

    # any module necessary for this one to work correctly
    'depends': ['website_sale'],

    # always loaded
    'data': [
```

restart Odoo, update your module, there is now a *Shop* section in the website, listing a number of pre-filled (via demonstration data) products.

The second step is to replace the *courses* model by `product.template`, and add a new category of product for courses: *academy/__openerp__.py*

```
        'security/ir.model.access.csv',
        'templates.xml',
        'views.xml',
        'data.xml',
    ],
    # only loaded in demonstration mode
    'demo': [
```

academy/data.xml

```
<openerp><data>
  <record model="product.public.category" id="category_courses">
    <field name="name">Courses</field>
    <field name="parent_id" ref="website_sale.categ_others"/>
  </record>
</data></openerp>
```

academy/demo.xml

```
        <record id="vaughn" model="academy.teachers">
            <field name="name">Lester Vaughn</field>
        </record>

        <record id="course0" model="product.template">
            <field name="name">Course 0</field>
            <field name="teacher_id" ref="padilla"/>
            <field name="public_categ_ids" eval="[(4, ref('academy.category_courses'), False)]"/>
            <field name="website_published">True</field>
            <field name="list_price" type="float">0</field>
            <field name="type">service</field>
        </record>
        <record id="course1" model="product.template">
            <field name="name">Course 1</field>
            <field name="teacher_id" ref="padilla"/>
            <field name="public_categ_ids" eval="[(4, ref('academy.category_courses'), False)]"/>
            <field name="website_published">True</field>
            <field name="list_price" type="float">0</field>
            <field name="type">service</field>
        </record>
```

```
        <record id="course2" model="product.template">
            <field name="name">Course 2</field>
            <field name="teacher_id" ref="vaughn"/>
            <field name="public_categ_ids" eval="[(4, ref('academy.category_courses'), False)]"/>
            <field name="website_published">True</field>
            <field name="list_price" type="float">0</field>
            <field name="type">service</field>
        </record>
    </data>
</openerp>
```

academy/models.py

```
    name = fields.Char()
    biography = fields.Html()

    course_ids = fields.One2many('product.template', 'teacher_id', string="Courses")

class Courses(models.Model):
    _inherit = 'product.template'

    teacher_id = fields.Many2one('academy.teachers', string="Teacher")
```

academy/security/ir.model.access.csv

```
id,name,model_id:id,group_id:id,perm_read,perm_write,perm_create,perm_unlink
access_academy_teachers,access_academy_teachers,model_academy_teachers,,1,0,0,0
```

academy/views.xml

```
      </form>
    </field>
  </record>

  <menuitem sequence="0" id="menu_academy" name="Academy"/>
  <menuitem id="menu_academy_content" parent="menu_academy"
            name="Academy Content"/>
  <menuitem id="menu_academy_content_teachers"
            parent="menu_academy_content"
```

With this installed, a few courses are now available in the *Shop*, though they may have to be looked for.

Note:

- to extend a model in-place, it's *inherited* without giving it a new *_name*
- `product.template` already uses the discussions system, so we can remove it from our extension model
- we're creating our courses as *published* by default so they can be seen without having to log in

---

Altering existing views

So far, we have briefly seen:

- the creation of new models
- the creation of new views
- the creation of new records
- the alteration of existing models

We're left with the alteration of existing records and the alteration of existing views. We'll do both on the *Shop* pages.

View alteration is done by creating *extension* views, which are applied on top of the original view and alter it. These alteration views can be added or removed without modifying the original, making it easier to try things out and roll changes back.

Since our courses are free, there is no reason to display their price on the shop page, so we're going to alter the view and hide the price if it's 0. The first task is finding out which view displays the price, this can be done via *Customize* → *HTML Editor* which lets us read the various templates involved in rendering a page. Going through a few of them, "Product item" looks a likely culprit.

Altering view architectures is done in 3 steps:

1. Create a new view
2. Extend the view to modify by setting the new view's `inherit_id` to the modified view's external id
3. In the architecture, use the `xpath` tag to select and alter elements from the modified view

academy/templates.xml

```
            <div class="oe_structure"/>
        </t>
    </template>

    <template id="product_item_hide_no_price" inherit_id="website_sale.products_item">
        <xpath expr="//div[hasclass('product_price')]/b" position="attributes">
            <attribute name="t-if">product.price &gt; 0</attribute>
        </xpath>
    </template>

    <!-- <template id="object"> -->
    <!--   <h1><t t-esc="object.display_name"/></h1> -->
    <!--   <dl> -->
```

The second thing we will change is making the product categories sidebar visible by default: *Customize* → *Product Categories* lets you toggle a tree of product categories (used to filter the main display) on and off.

This is done via the `customize_show` and `active` fields of extension templates: an extension template (such as the one we've just created) can be *customize_show=True*. This choice will display the view in the *Customize* menu with a check box, allowing administrators to activate or disable them (and easily customize their website pages).

We simply need to modify the *Product Categories* record and set its default to *active="True"*: *academy/templates.xml*

```
        </xpath>
    </template>

    <record id="website_sale.products_categories" model="ir.ui.view">
        <field name="active" eval="True"/>
    </record>

    <!-- <template id="object"> -->
    <!--   <h1><t t-esc="object.display_name"/></h1> -->
    <!--   <dl> -->
```

With this, the *Product Categories* sidebar will automatically be enabled when the *Academy* module is installed.

1.3 Building a Module

Warning: This tutorial requires *having installed Odoo*

1.3.1 Start/Stop the Odoo server

Odoo uses a client/server architecture in which clients are web browsers accessing the Odoo server via RPC.

Business logic and extension is generally performed on the server side, although supporting client features (e.g. new data representation such as interactive maps) can be added to the client.

In order to start the server, simply invoke the command *odoo.py* in the shell, adding the full path to the file if necessary:

```
odoo.py
```

The server is stopped by hitting `Ctrl-C` twice from the terminal, or by killing the corresponding OS process.

1.3.2 Build an Odoo module

Both server and client extensions are packaged as *modules* which are optionally loaded in a *database*.

Odoo modules can either add brand new business logic to an Odoo system, or alter and extend existing business logic: a module can be created to add your country's accounting rules to Odoo's generic accounting support, while the next module adds support for real-time visualisation of a bus fleet.

Everything in Odoo thus starts and ends with modules.

Composition of a module

An Odoo module can contain a number of elements:

Business objects declared as Python classes, these resources are automatically persisted by Odoo based on their configuration

Data files XML or CSV files declaring metadata (views or workflows), configuration data (modules parameterization), demonstration data and more

Web controllers Handle requests from web browsers

Static web data Images, CSS or javascript files used by the web interface or website

Module structure

Each module is a directory within a *module directory*. Module directories are specified by using the `--addons-path` option.

Tip: most command-line options can also be set using *a configuration file*

An Odoo module is declared by its *manifest*. See the *manifest documentation* information about it.

A module is also a Python package with a `__init__.py` file, containing import instructions for various Python files in the module.

For instance, if the module has a single `mymodule.py` file `__init__.py` might contain:

```
from . import mymodule
```

Odoo provides a mechanism to help set up a new module, *odoo.py* has a subcommand *scaffold* to create an empty module:

```
$ odoo.py scaffold <module name> <where to put it>
```

The command creates a subdirectory for your module, and automatically creates a bunch of standard files for a module. Most of them simply contain commented code or XML. The usage of most of those files will be explained along this tutorial.

M odule creation

Use the command line above to create an empty module Open Academy, and install it in Odoo.

Object-Relational Mapping

A key component of Odoo is the ORM (Object-Relational Mapping) layer. This layer avoids having to write most SQL (Structured Query Language) by hand and provides extensibility and security services[1].

Business objects are declared as Python classes extending *Model* which integrates them into the automated persistence system.

Models can be configured by setting a number of attributes at their definition. The most important attribute is *_name* which is required and defines the name for the model in the Odoo system. Here is a minimally complete definition of a model:

```
from openerp import models
class MinimalModel(models.Model):
    _name = 'test.model'
```

Model fields

Fields are used to define what the model can store and where. Fields are defined as attributes on the model class:

```
from openerp import models, fields

class LessMinimalModel(models.Model):
    _name = 'test.model2'

    name = fields.Char()
```

Common Attributes

Much like the model itself, its fields can be configured, by passing configuration attributes as parameters:

```
name = field.Char(required=True)
```

Some attributes are available on all fields, here are the most common ones:

string (unicode, default: field's name) The label of the field in UI (visible by users).

required (bool, default: False) If `True`, the field can not be empty, it must either have a default value or always be given a value when creating a record.

help (unicode, default: '') Long-form, provides a help tooltip to users in the UI.

index (bool, default: False) Requests that Odoo create a database index on the column

Simple fields

There are two broad categories of fields: "simple" fields which are atomic values stored directly in the model's table and "relational" fields linking records (of the same model or of different models).

Example of simple fields are *Boolean*, *Date*, *Char*.

[1] writing raw SQL queries is possible, but requires care as it bypasses all Odoo authentication and security mechanisms.

Reserved fields

Odoo creates a few fields in all models[2]. These fields are managed by the system and shouldn't be written to. They can be read if useful or necessary:

id (Id) the unique identifier for a record in its model

create_date (*Datetime*) creation date of the record

create_uid (*Many2one*) user who created the record

write_date (*Datetime*) last modification date of the record

write_uid (*Many2one*) user who last modified the record

Special fields

By default, Odoo also requires a `name` field on all models for various display and search behaviors. The field used for these purposes can be overridden by setting *`_rec_name`*.

D efine a model

Define a new data model *Course* in the *openacademy* module. A course has a title and a description. Courses must have a title.

Data files

Odoo is a highly data driven system. Although behavior is customized using Python code part of a module's value is in the data it sets up when loaded.

Tip: some modules exist solely to add data into Odoo

Module data is declared via *data files*, XML files with `<record>` elements. Each `<record>` element creates or updates a database record.

```
<openerp>
    <data>
        <record model="{model name}" id="{record identifier}">
            <field name="{a field name}">{a value}</field>
        </record>
    </data>
<openerp>
```

- `model` is the name of the Odoo model for the record
- `id` is an external identifier, it allows referring to the record (without having to know its in-database identifier)
- `<field>` elements have a `name` which is the name of the field in the model (e.g. `description`). Their body is the field's value.

Data files have to be declared in the manifest file to be loaded, they can be declared in the `'data'` list (always loaded) or in the `'demo'` list (only loaded in demonstration mode).

D efine demonstration data

Create demonstration data filling the *Courses* model with a few demonstration courses.

---

[2] it is possible to *disable the automatic creation of some fields*

Actions and Menus

Actions and menus are regular records in database, usually declared through data files. Actions can be triggered in three ways:

1. by clicking on menu items (linked to specific actions)
2. by clicking on buttons in views (if these are connected to actions)
3. as contextual actions on object

Because menus are somewhat complex to declare there is a `<menuitem>` shortcut to declare an `ir.ui.menu` and connect it to the corresponding action more easily.

```
<record model="ir.actions.act_window" id="action_list_ideas">
    <field name="name">Ideas</field>
    <field name="res_model">idea.idea</field>
    <field name="view_mode">tree,form</field>
</record>
<menuitem id="menu_ideas" parent="menu_root" name="Ideas" sequence="10"
          action="action_list_ideas"/>
```

Danger: The action must be declared before its corresponding menu in the XML file.
Data files are executed sequentially, the action's `id` must be present in the database before the menu can be created.

D efine new menu entries

Define new menu entries to access courses and sessions under the OpenAcademy menu entry. A user should be able to

- display a list of all the courses
- create/modify courses

1.3.3 Basic views

Views define the way the records of a model are displayed. Each type of view represents a mode of visualization (a list of records, a graph of their aggregation, ...). Views can either be requested generically via their type (e.g. *a list of partners*) or specifically via their id. For generic requests, the view with the correct type and the lowest priority will be used (so the lowest-priority view of each type is the default view for that type).

View inheritance allows altering views declared elsewhere (adding or removing content).

Generic view declaration

A view is declared as a record of the model `ir.ui.view`. The view type is implied by the root element of the `arch` field:

```
<record model="ir.ui.view" id="view_id">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
    <field name="arch" type="xml">
        <!-- view content: <form>, <tree>, <graph>, ... -->
    </field>
</record>
```

Danger: The view's content is XML.
The `arch` field must thus be declared as `type="xml"` to be parsed correctly.

Tree views

Tree views, also called list views, display records in a tabular form.

Their root element is `<tree>`. The simplest form of the tree view simply lists all the fields to display in the table (each field as a column):

```
<tree string="Idea list">
    <field name="name"/>
    <field name="inventor_id"/>
</tree>
```

Form views

Forms are used to create and edit single records.

Their root element is `<form>`. They composed of high-level structure elements (groups, notebooks) and interactive elements (buttons and fields):

```
<form string="Idea form">
    <group colspan="4">
        <group colspan="2" col="2">
            <separator string="General stuff" colspan="2"/>
            <field name="name"/>
            <field name="inventor_id"/>
        </group>

        <group colspan="2" col="2">
            <separator string="Dates" colspan="2"/>
            <field name="active"/>
            <field name="invent_date" readonly="1"/>
        </group>

        <notebook colspan="4">
            <page string="Description">
                <field name="description" nolabel="1"/>
            </page>
        </notebook>

        <field name="state"/>
    </group>
</form>
```

C ustomise form view using XML

Create your own form view for the Course object. Data displayed should be: the name and the description of the course.

N otebooks

In the Course form view, put the description field under a tab, such that it will be easier to add other tabs later, containing additional information.

Form views can also use plain HTML for more flexible layouts:

```
<form string="Idea Form">
    <header>
        <button string="Confirm" type="object" name="action_confirm"
                states="draft" class="oe_highlight" />
        <button string="Mark as done" type="object" name="action_done"
                states="confirmed" class="oe_highlight"/>
        <button string="Reset to draft" type="object" name="action_draft"
```

```
                states="confirmed,done" />
        <field name="state" widget="statusbar"/>
    </header>
    <sheet>
        <div class="oe_title">
            <label for="name" class="oe_edit_only" string="Idea Name" />
            <h1><field name="name" /></h1>
        </div>
        <separator string="General" colspan="2" />
        <group colspan="2" col="2">
            <field name="description" placeholder="Idea description..." />
        </group>
    </sheet>
</form>
```

Search views

Search views customize the search field associated with the list view (and other aggregated views). Their root element is `<search>` and they're composed of fields defining which fields can be searched on:

```
<search>
    <field name="name"/>
    <field name="inventor_id"/>
</search>
```

If no search view exists for the model, Odoo generates one which only allows searching on the `name` field.

S earch courses

Allow searching for courses based on their title or their description.

1.3.4 Relations between models

A record from a model may be related to a record from another model. For instance, a sale order record is related to a client record that contains the client data; it is also related to its sale order line records.

C reate a session model

For the module Open Academy, we consider a model for *sessions*: a session is an occurrence of a course taught at a given time for a given audience.

Create a model for *sessions*. A session has a name, a start date, a duration and a number of seats. Add an action and a menu item to display them. Make the new model visible via a menu item.

Relational fields

Relational fields link records, either of the same model (hierarchies) or between different models.

Relational field types are:

Many2one(other_model, ondelete='set null') A simple link to an other object:

```
    print foo.other_id.name
```

See also:

foreign keys

One2many(other_model, related_field) A virtual relationship, inverse of a *Many2one*. A *One2many* behaves as a container of records, accessing it results in a (possibly empty) set of records:

```
    for other in foo.other_ids:
        print other.name
```

> **Danger:** Because a *One2many* is a virtual relationship, there *must* be a *Many2one* field in the *other_model*, and its name *must* be *related_field*

Many2many(other_model) Bidirectional multiple relationship, any record on one side can be related to any number of records on the other side. Behaves as a container of records, accessing it also results in a possibly empty set of records:

```
    for other in foo.other_ids:
        print other.name
```

M any2one relations

Using a many2one, modify the *Course* and *Session* models to reflect their relation with other models:

- A course has a *responsible* user; the value of that field is a record of the built-in model `res.users`.
- A session has an *instructor*; the value of that field is a record of the built-in model `res.partner`.
- A session is related to a *course*; the value of that field is a record of the model `openacademy.course` and is required.
- Adapt the views.

I nverse one2many relations

Using the inverse relational field one2many, modify the models to reflect the relation between courses and sessions.

M ultiple many2many relations

Using the relational field many2many, modify the *Session* model to relate every session to a set of *attendees*. Attendees will be represented by partner records, so we will relate to the built-in model `res.partner`. Adapt the views accordingly.

1.3.5 Inheritance

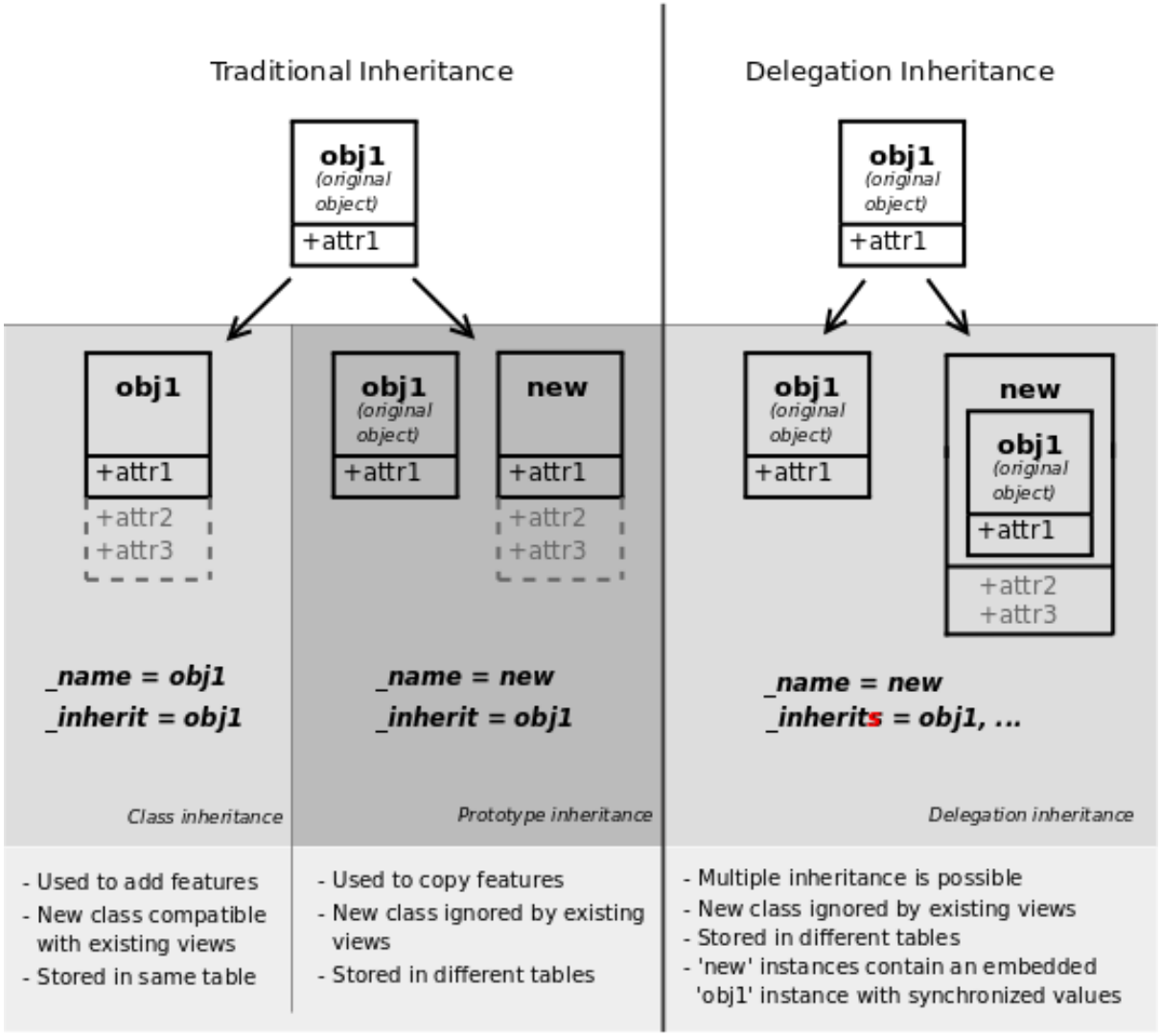
Model inheritance

Odoo provides two *inheritance* mechanisms to extend an existing model in a modular way.

The first inheritance mechanism allows a module to modify the behavior of a model defined in another module:

- add fields to a model,
- override the definition of fields on a model,
- add constraints to a model,
- add methods to a model,
- override existing methods on a model.

The second inheritance mechanism (delegation) allows to link every record of a model to a record in a parent model, and provides transparent access to the fields of the parent record.



See also:

- *_inherit*
- *_inherits*

View inheritance

Instead of modifying existing views in place (by overwriting them), Odoo provides view inheritance where children "extension" views are applied on top of root views, and can add or remove content from their parent.

An extension view references its parent using the `inherit_id` field, and instead of a single view its `arch` field is composed of any number of `xpath` elements selecting and altering the content of their parent view:

```
<!-- improved idea categories list -->
<record id="idea_category_list2" model="ir.ui.view">
    <field name="name">id.category.list2</field>
    <field name="model">idea.category</field>
    <field name="inherit_id" ref="id_category_list"/>
    <field name="arch" type="xml">
        <!-- find field description and add the field
             idea_ids after it -->
        <xpath expr="//field[@name='description']" position="after">
          <field name="idea_ids" string="Number of ideas"/>
        </xpath>
```

```
    </field>
</record>
```

expr An XPath expression selecting a single element in the parent view. Raises an error if it matches no element or more than one

position Operation to apply to the matched element:

inside appends `xpath`'s body at the end of the matched element

replace replaces the matched element by the `xpath`'s body

before inserts the `xpath`'s body as a sibling before the matched element

after inserts the `xpaths`'s body as a sibling after the matched element

attributes alters the attributes of the matched element using special `attribute` elements in the `xpath`'s body

---

Tip: When matching a single element, the `position` attribute can be set directly on the element to be found. Both inheritances below will give the same result.

```
<xpath expr="//field[@name='description']" position="after">
    <field name="idea_ids" />
</xpath>

<field name="description" position="after">
    <field name="idea_ids" />
</field>
```

A lter existing content

- Using model inheritance, modify the existing *Partner* model to add an `instructor` boolean field, and a many2many field that corresponds to the session-partner relation
- Using view inheritance, display this fields in the partner form view

---

Domains

In Odoo, *Domains* are values that encode conditions on records. A domain is a list of criteria used to select a subset of a model's records. Each criteria is a triple with a field name, an operator and a value.

For instance, when used on the *Product* model the following domain selects all *services* with a unit price over *1000*:

```
[('product_type', '=', 'service'), ('unit_price', '>', 1000)]
```

By default criteria are combined with an implicit AND. The logical operators `&` (AND), `|` (OR) and `!` (NOT) can be used to explicitly combine criteria. They are used in prefix position (the operator is inserted before its arguments rather than between). For instance to select products "which are services *OR* have a unit price which is *NOT* between 1000 and 2000":

```
['|',
    ('product_type', '=', 'service'),
    '!', '&',
        ('unit_price', '>=', 1000),
        ('unit_price', '<', 2000)]
```

A `domain` parameter can be added to relational fields to limit valid records for the relation when trying to select records in the client interface.

D omains on relational fields

When selecting the instructor for a *Session*, only instructors (partners with `instructor` set to `True`) should be visible.

M ore complex domains

Create new partner categories *Teacher / Level 1* and *Teacher / Level 2*. The instructor for a session can be either an instructor or a teacher (of any level).

1.3.6 Computed fields and default values

So far fields have been stored directly in and retrieved directly from the database. Fields can also be *computed*. In that case, the field's value is not retrieved from the database but computed on-the-fly by calling a method of the model.

To create a computed field, create a field and set its attribute `compute` to the name of a method. The computation method should simply set the value of the field to compute on every record in `self`.

> **Danger:** `self` is a collection
> The object `self` is a *recordset*, i.e., an ordered collection of records. It supports the standard Python operations on collections, like `len(self)` and `iter(self)`, plus extra set operations like `recs1 + recs2`. Iterating over `self` gives the records one by one, where each record is itself a collection of size 1. You can access/assign fields on single records by using the dot notation, like `record.name`.

```
import random
from openerp import models, fields

class ComputedModel(models.Model):
    _name = 'test.computed'

    name = fields.Char(compute='_compute_name')

    @api.multi
    def _compute_name(self):
        for record in self:
            record.name = str(random.randint(1, 1e6))
```

Dependencies

The value of a computed field usually depends on the values of other fields on the computed record. The ORM expects the developer to specify those dependencies on the compute method with the decorator *depends()*. The given dependencies are used by the ORM to trigger the recomputation of the field whenever some of its dependencies have been modified:

```
from openerp import models, fields, api

class ComputedModel(models.Model):
    _name = 'test.computed'

    name = fields.Char(compute='_compute_name')
    value = fields.Integer()

    @api.depends('value')
    def _compute_name(self):
        for record in self:
            self.name = "Record with value %s" % self.value
```

C omputed fields

- Add the percentage of taken seats to the *Session* model
- Display that field in the tree and form views
- Display the field as a progress bar

Default values

Any field can be given a default value. In the field definition, add the option `default=X` where `X` is either a Python literal value (boolean, integer, float, string), or a function taking a recordset and returning a value:

```
name = fields.Char(default="Unknown")
user_id = fields.Many2one('res.users', default=lambda self: self.env.user)
```

Note: The object `self.env` gives access to request parameters and other useful things:

- `self.env.cr` or `self._cr` is the database *cursor* object; it is used for querying the database
- `self.env.uid` or `self._uid` is the current user's database id
- `self.env.user` is the current user's record
- `self.env.context` or `self._context` is the context dictionary
- `self.env.ref(xml_id)` returns the record corresponding to an XML id
- `self.env[model_name]` returns an instance of the given model

A ctive objects – Default values

- Define the start_date default value as today (see *`Date`*).
- Add a field `active` in the class Session, and set sessions as active by default.

1.3.7 Onchange

The "onchange" mechanism provides a way for the client interface to update a form whenever the user has filled in a value in a field, without saving anything to the database.

For instance, suppose a model has three fields `amount`, `unit_price` and `price`, and you want to update the price on the form when any of the other fields is modified. To achieve this, define a method where `self` represents the record in the form view, and decorate it with *`onchange()`* to specify on which field it has to be triggered. Any change you make on `self` will be reflected on the form.

```
<!-- content of form view -->
<field name="amount"/>
<field name="unit_price"/>
<field name="price" readonly="1"/>
```

```
# onchange handler
@api.onchange('amount', 'unit_price')
def _onchange_price(self):
    # set auto-changing field
    self.price = self.amount * self.unit_price
    # Can optionally return a warning and domains
    return {
        'warning': {
            'title': "Something bad happened",
            'message': "It was very bad indeed",
        }
    }
```

For computed fields, valued `onchange` behavior is built-in as can be seen by playing with the *Session* form: change the number of seats or participants, and the `taken_seats` progressbar is automatically updated.

W arning

Add an explicit onchange to warn about invalid values, like a negative number of seats, or more participants than seats.

1.3.8 Model constraints

Odoo provides two ways to set up automatically verified invariants: *Python constraints* and *SQL constraints*.

A Python constraint is defined as a method decorated with *constrains()*, and invoked on a recordset. The decorator specifies which fields are involved in the constraint, so that the constraint is automatically evaluated when one of them is modified. The method is expected to raise an exception if its invariant is not satisfied:

```
from openerp.exceptions import ValidationError

@api.constrains('age')
def _check_something(self):
    for record in self:
        if record.age > 20:
            raise ValidationError("Your record is too old: %s" % record.age)
    # all records passed the test, don't return anything
```

A dd Python constraints

Add a constraint that checks that the instructor is not present in the attendees of his/her own session.

SQL constraints are defined through the model attribute *_sql_constraints*. The latter is assigned to a list of triples of strings `(name, sql_definition, message)`, where `name` is a valid SQL constraint name, `sql_definition` is a table_constraint expression, and `message` is the error message.

A dd SQL constraints

With the help of PostgreSQL's documentation , add the following constraints:

1. CHECK that the course description and the course title are different
2. Make the Course's name UNIQUE

E xercise 6 - Add a duplicate option

Since we added a constraint for the Course name uniqueness, it is not possible to use the "duplicate" function anymore (*Form → Duplicate*).

Re-implement your own "copy" method which allows to duplicate the Course object, changing the original name into "Copy of [original name]".

1.3.9 Advanced Views

Tree views

Tree views can take supplementary attributes to further customize their behavior:

colors mappings of colors to conditions. If the condition evaluates to `True`, the corresponding color is applied to the row:

```
    <tree string="Idea Categories" colors="blue:state=='draft';red:state=='trashed'">
        <field name="name"/>
        <field name="state"/>
    </tree>
```

Clauses are separated by `;`, the color and condition are separated by `:`.

editable Either "top" or "bottom". Makes the tree view editable in-place (rather than having to go through the form view), the value is the position where new rows appear.

L ist coloring

Modify the Session tree view in such a way that sessions lasting less than 5 days are colored blue, and the ones lasting more than 15 days are colored red.

Calendars

Displays records as calendar events. Their root element is `<calendar>` and their most common attributes are:

color The name of the field used for *color segmentation*. Colors are automatically distributed to events, but events in the same color segment (records which have the same value for their `@color` field) will be given the same color.

date_start record's field holding the start date/time for the event

date_stop (optional) record's field holding the end date/time for the event

field (to define the label for each calendar event)

```
<calendar string="Ideas" date_start="invent_date" color="inventor_id">
    <field name="name"/>
</calendar>
```

C alendar view

Add a Calendar view to the *Session* model enabling the user to view the events associated to the Open Academy.

Search views

Search view `<field>` elements can have a `@filter_domain` that overrides the domain generated for searching on the given field. In the given domain, `self` represents the value entered by the user. In the example below, it is used to search on both fields `name` and `description`.

Search views can also contain `<filter>` elements, which act as toggles for predefined searches. Filters must have one of the following attributes:

domain add the given domain to the current search

context add some context to the current search; use the key `group_by` to group results on the given field name

```
<search string="Ideas">
    <field name="name"/>
    <field name="description" string="Name and description"
           filter_domain="['|', ('name', 'ilike', self), ('description', 'ilike', self)]"/>
    <field name="inventor_id"/>
    <field name="country_id" widget="selection"/>

    <filter name="my_ideas" string="My Ideas"
            domain="[('inventor_id', '=', uid)]"/>
    <group string="Group By">
        <filter name="group_by_inventor" string="Inventor"
```

```
                context="{'group_by': 'inventor_id'}"/>
    </group>
</search>
```

To use a non-default search view in an action, it should be linked using the `search_view_id` field of the action record.

The action can also set default values for search fields through its `context` field: context keys of the form `search_default_`*`field_name`* will initialize *field_name* with the provided value. Search filters must have an optional `@name` to have a default and behave as booleans (they can only be enabled by default).

S earch views

1. Add a button to filter the courses for which the current user is the responsible in the course search view. Make it selected by default.
2. Add a button to group courses by responsible user.

Gantt

Horizontal bar charts typically used to show project planning and advancement, their root element is `<gantt>`.

```
<gantt string="Ideas" date_start="invent_date" color="inventor_id">
    <level object="idea.idea" link="id" domain="[]">
        <field name="inventor_id"/>
    </level>
</gantt>
```

G antt charts

Add a Gantt Chart enabling the user to view the sessions scheduling linked to the Open Academy module. The sessions should be grouped by instructor.

Graph views

Graph views allow aggregated overview and analysis of models, their root element is `<graph>`.

Graph views have 4 display modes, the default mode is selected using the `@type` attribute.

Pivot a multidimensional table, allows the selection of filers and dimensions to get the right aggregated dataset before moving to a more graphical overview

Bar (default) a bar chart, the first dimension is used to define groups on the horizontal axis, other dimensions define aggregated bars within each group.

By default bars are side-by-side, they can be stacked by using `@stacked="True"` on the `<graph>`

Line 2-dimensional line chart

Pie 2-dimensional pie

Graph views contain `<field>` with a mandatory `@type` attribute taking the values:

`row` (default) the field should be aggregated by default

`measure` the field should be aggregated rather than grouped on

```
<graph string="Total idea score by Inventor">
    <field name="inventor_id"/>
    <field name="score" type="measure"/>
</graph>
```

Warning: Graph views perform aggregations on database values, they do not work with non-stored computed fields.

G raph view

Add a Graph view in the Session object that displays, for each course, the number of attendees under the form of a bar chart.

Kanban

Used to organize tasks, production processes, etc... their root element is `<kanban>`.

A kanban view shows a set of cards possibly grouped in columns. Each card represents a record, and each column the values of an aggregation field.

For instance, project tasks may be organized by stage (each column is a stage), or by responsible (each column is a user), and so on.

Kanban views define the structure of each card as a mix of form elements (including basic HTML) and *QWeb*.

K anban view

Add a Kanban view that displays sessions grouped by course (columns are thus courses).

1.3.10 Workflows

Workflows are models associated to business objects describing their dynamics. Workflows are also used to track processes that evolve over time.

A lmost a workflow

Add a `state` field to the *Session* model. It will be used to define a workflow-ish.

A sesion can have three possible states: Draft (default), Confirmed and Done.

In the session form, add a (read-only) field to visualize the state, and buttons to change it. The valid transitions are:

- Draft -> Confirmed
- Confirmed -> Draft
- Confirmed -> Done
- Done -> Draft

Workflows may be associated with any object in Odoo, and are entirely customizable. Workflows are used to structure and manage the lifecycles of business objects and documents, and define transitions, triggers, etc. with graphical tools. Workflows, activities (nodes or actions) and transitions (conditions) are declared as XML records, as usual. The tokens that navigate in workflows are called workitems.

Warning: A workflow associated with a model is only created when the model's records are created. Thus there is no workflow instance associated with session instances created before the workflow's definition

---

W orkflow

Replace the ad-hoc *Session* workflow by a real workflow. Transform the *Session* form view so its buttons call the workflow instead of the model's methods.

A utomatic transitions

Automatically transition sessions from *Draft* to *Confirmed* when more than half the session's seats are reserved.

S erver actions

Replace the Python methods for synchronizing session state by server actions.

Both the workflow and the server actions could have been created entirely from the UI.

1.3.11 Security

Access control mechanisms must be configured to achieve a coherent security policy.

Group-based access control mechanisms

Groups are created as normal records on the model `res.groups`, and granted menu access via menu definitions. However even without a menu, objects may still be accessible indirectly, so actual object-level permissions (read, write, create, unlink) must be defined for groups. They are usually inserted via CSV files inside modules. It is also possible to restrict access to specific fields on a view or object using the field's groups attribute.

Access rights

Access rights are defined as records of the model `ir.model.access`. Each access right is associated to a model, a group (or no group for global access), and a set of permissions: read, write, create, unlink. Such access rights are usually created by a CSV file named after its model: `ir.model.access.csv`.

```
id,name,model_id/id,group_id/id,perm_read,perm_write,perm_create,perm_unlink
access_idea_idea,idea.idea,model_idea_idea,base.group_user,1,1,1,0
access_idea_vote,idea.vote,model_idea_vote,base.group_user,1,1,1,0
```

A dd access control through the OpenERP interface

Create a new user "John Smith". Then create a group "OpenAcademy / Session Read" with read access to the *Session* model.

A dd access control through data files in your module

Using data files,

- Create a group *OpenAcademy / Manager* with full access to all OpenAcademy models
- Make *Session* and *Course* readable by all users

Record rules

A record rule restricts the access rights to a subset of records of the given model. A rule is a record of the model `ir.rule`, and is associated to a model, a number of groups (many2many field), permissions to which the restriction applies, and a domain. The domain specifies to which records the access rights are limited.

Here is an example of a rule that prevents the deletion of leads that are not in state `cancel`. Notice that the value of the field `groups` must follow the same convention as the method *`write()`* of the ORM.

```
<record id="delete_cancelled_only" model="ir.rule">
    <field name="name">Only cancelled leads may be deleted</field>
    <field name="model_id" ref="crm.model_crm_lead"/>
    <field name="groups" eval="[(4, ref('base.group_sale_manager'))]"/>
```

```
    <field name="perm_read" eval="0"/>
    <field name="perm_write" eval="0"/>
    <field name="perm_create" eval="0"/>
    <field name="perm_unlink" eval="1" />
    <field name="domain_force">[('state','=','cancel')]</field>
</record>
```

R ecord rule

Add a record rule for the model Course and the group "OpenAcademy / Manager", that restricts `write` and `unlink` accesses to the responsible of a course. If a course has no responsible, all users of the group must be able to modify it.

1.3.12 Wizards

Wizards describe interactive sessions with the user (or dialog boxes) through dynamic forms. A wizard is simply a model that extends the class `TransientModel` instead of *`Model`*. The class `TransientModel` extends *`Model`* and reuse all its existing mechanisms, with the following particularities:

- Wizard records are not meant to be persistent; they are automatically deleted from the database after a certain time. This is why they are called *transient*.
- Wizard models do not require explicit access rights: users have all permissions on wizard records.
- Wizard records may refer to regular records or wizard records through many2one fields, but regular records *cannot* refer to wizard records through a many2one field.

We want to create a wizard that allow users to create attendees for a particular session, or for a list of sessions at once.

D efine the wizard

Create a wizard model with a many2one relationship with the *Session* model and a many2many relationship with the *Partner* model.

Launching wizards

Wizards are launched by `ir.actions.act_window` records, with the field `target` set to the value `new`. The latter opens the wizard view into a popup window. The action may be triggered by a menu item.

There is another way to launch the wizard: using an `ir.actions.act_window` record like above, but with an extra field `src_model` that specifies in the context of which model the action is available. The wizard will appear in the contextual actions of the model, above the main view. Because of some internal hooks in the ORM, such an action is declared in XML with the tag `act_window`.

```
<act_window id="launch_the_wizard"
            name="Launch the Wizard"
            src_model="context_model_name"
            res_model="wizard_model_name"
            view_mode="form"
            target="new"
            key2="client_action_multi"/>
```

Wizards use regular views and their buttons may use the attribute `special="cancel"` to close the wizard window without saving.

L aunch the wizard

1. Define a form view for the wizard.
2. Add the action to launch it in the context of the *Session* model.

3. Define a default value for the session field in the wizard; use the context parameter `self._context` to retrieve the current session.

R egister attendees

Add buttons to the wizard, and implement the corresponding method for adding the attendees to the given session.

R egister attendees to multiple sessions

Modify the wizard model so that attendees can be registered to multiple sessions.

1.3.13 Internationalization

Each module can provide its own translations within the i18n directory, by having files named LANG.po where LANG is the locale code for the language, or the language and country combination when they differ (e.g. pt.po or pt_BR.po). Translations will be loaded automatically by Odoo for all enabled languages. Developers always use English when creating a module, then export the module terms using Odoo's gettext POT export feature (*Settings* → *Translations* → *Import/Export* → *Export Translation* without specifying a language), to create the module template POT file, and then derive the translated PO files. Many IDE's have plugins or modes for editing and merging PO/POT files.

Tip: The GNU gettext format (Portable Object) used by Odoo is integrated into LaunchPad, making it an online collaborative translation platform.

```
|- idea/ # The module directory
   |- i18n/ # Translation files
      | - idea.pot # Translation Template (exported from Odoo)
      | - fr.po # French translation
      | - pt_BR.po # Brazilian Portuguese translation
      | (...)
```

Tip: By default Odoo's POT export only extracts labels inside XML files or inside field definitions in Python code, but any Python string can be translated this way by surrounding it with the function `openerp._()` (e.g. `_("Label")`)

T ranslate a module

Choose a second language for your Odoo installation. Translate your module using the facilities provided by Odoo.

1.3.14 Reporting

Printed reports

Odoo 8.0 comes with a new report engine based on *QWeb*, Twitter Bootstrap and Wkhtmltopdf.

A report is a combination two elements:

- an `ir.actions.report.xml`, for which a `<report>` shortcut element is provided, it sets up various basic parameters for the report (default type, whether the report should be saved to the database after generation,...)

```
    <report
        id="account_invoices"
        model="account.invoice"
        string="Invoices"
```

```
        report_type="qweb-pdf"
        name="account.report_invoice"
        file="account.report_invoice"
        attachment_use="True"
        attachment="(object.state in ('open','paid')) and
            ('INV'+(object.number or '').replace('/','')+'.pdf')"
    />
```

- A standard *QWeb view* for the actual report:

```
    <t t-call="report.html_container">
        <t t-foreach="docs" t-as="o">
            <t t-call="report.external_layout">
                <div class="page">
                    <h2>Report title</h2>
                </div>
            </t>
        </t>
    </t>

    the standard rendering context provides a number of elements, the most
    important being:

    ``docs``
        the records for which the report is printed
    ``user``
        the user printing the report
```

Because reports are standard web pages, they are available through a URL and output parameters can be manipulated through this URL, for instance the HTML version of the *Invoice* report is available through http://localhost:8069/report/html/account.report_invoice/1 (if `account` is installed) and the PDF version through http://localhost:8069/report/pdf/account.report_invoice/1.

Danger: If it appears that your PDF report is missing the styles (i.e. the text appears but the style/layout is different from the html version), probably your wkhtmltopdf process cannot reach your web server to download them.
If you check your server logs and see that the CSS styles are not being downloaded when generating a PDF report, most surely this is the problem.
The wkhtmltopdf process will use the `web.base.url` system parameter as the *root path* to all linked files, but this parameter is automatically updated each time the Administrator is logged in. If your server resides behind some kind of proxy, that could not be reachable. You can fix this by adding one of these system parameters:

- `report.url`, pointing to an URL reachable from your server (probably `http://localhost:8069` or something similar). It will be used for this particular purpose only.
- `web.base.url.freeze`, when set to `True`, will stop the automatic updates to `web.base.url`.

C reate a report for the Session model

For each session, it should display session's name, its start and end, and list the session's attendees.

Dashboards

D efine a Dashboard

Define a dashboard containing the graph view you created, the sessions calendar view and a list view of the courses (switchable to a form view). This dashboard should be available through a menuitem in the menu, and automatically displayed in the web client when the OpenAcademy main menu is selected.

1.3.15 WebServices

The web-service module offer a common interface for all web-services :

- XML-RPC
- JSON-RPC

Business objects can also be accessed via the distributed object mechanism. They can all be modified via the client interface with contextual views.

Odoo is accessible through XML-RPC/JSON-RPC interfaces, for which libraries exist in many languages.

XML-RPC Library

The following example is a Python program that interacts with an Odoo server with the library `xmlrpclib`:

```
import xmlrpclib

root = 'http://%s:%d/xmlrpc/' % (HOST, PORT)

uid = xmlrpclib.ServerProxy(root + 'common').login(DB, USER, PASS)
print "Logged in as %s (uid: %d)" % (USER, uid)

# Create a new note
sock = xmlrpclib.ServerProxy(root + 'object')
args = {
    'color' : 8,
    'memo' : 'This is a note',
    'create_uid': uid,
}
note_id = sock.execute(DB, uid, PASS, 'note.note', 'create', args)
```

A dd a new service to the client

Write a Python program able to send XML-RPC requests to a PC running Odoo (yours, or your instructor's). This program should display all the sessions, and their corresponding number of seats. It should also create a new session for one of the courses.

JSON-RPC Library

The following example is a Python program that interacts with an Odoo server with the standard Python libraries `urllib2` and `json`:

```
import json
import random
import urllib2

def json_rpc(url, method, params):
    data = {
        "jsonrpc": "2.0",
        "method": method,
        "params": params,
        "id": random.randint(0, 1000000000),
    }
    req = urllib2.Request(url=url, data=json.dumps(data), headers={
        "Content-Type":"application/json",
    })
    reply = json.load(urllib2.urlopen(req))
    if reply.get("error"):
        raise Exception(reply["error"])
```

```
    return reply["result"]

def call(url, service, method, *args):
    return json_rpc(url, "call", {"service": service, "method": method, "args": args})

# log in the given database
url = "http://%s:%s/jsonrpc" % (HOST, PORT)
uid = call(url, "common", "login", DB, USER, PASS)

# create a new note
args = {
    'color' : 8,
    'memo' : 'This is another note',
    'create_uid': uid,
}
note_id = call(url, "object", "execute", DB, uid, PASS, 'note.note', 'create', args)
```

Here is the same program, using the library jsonrpclib:

```
import jsonrpclib

# server proxy object
url = "http://%s:%s/jsonrpc" % (HOST, PORT)
server = jsonrpclib.Server(url)

# log in the given database
uid = server.call(service="common", method="login", args=[DB, USER, PASS])

# helper function for invoking model methods
def invoke(model, method, *args):
    args = [DB, uid, PASS, model, method] + list(args)
    return server.call(service="object", method="execute", args=args)

# create a new note
args = {
    'color' : 8,
    'memo' : 'This is another note',
    'create_uid': uid,
}
note_id = invoke('note.note', 'create', args)
```

Examples can be easily adapted from XML-RPC to JSON-RPC.

Note: There are a number of high-level APIs in various languages to access Odoo systems without *explicitly* going through XML-RPC or JSON-RPC, such as:

- https://github.com/akretion/ooor
- https://github.com/syleam/openobject-library
- https://github.com/nicolas-van/openerp-client-lib
- https://pypi.python.org/pypi/oersted/

1.4 Building Interface Extensions

This guide is about creating modules for Odoo's web client.

To create websites with Odoo, see Building a Website; to add business capabilities or extend existing business systems of Odoo, see Building a Module.

Warning: This guide assumes knowledge of:
- Javascript basics and good practices
- jQuery
- Underscore.js

It also requires *an installed Odoo*, and Git.

1.4.1 A Simple Module

Let's start with a simple Odoo module holding basic web component configuration and letting us test the web framework.

The example module is available online and can be downloaded using the following command:

```
$ git clone http://github.com/odoo/petstore
```

This will create a `petstore` folder wherever you executed the command. You then need to add that folder to Odoo's *`addons path`*, create a new database and install the `oepetstore` module.

If you browse the `petstore` folder, you should see the following content:

```
oepetstore
|-- images
|   |-- alligator.jpg
|   |-- ball.jpg
|   |-- crazy_circle.jpg
|   |-- fish.jpg
|   `-- mice.jpg
|-- __init__.py
|-- oepetstore.message_of_the_day.csv
|-- __openerp__.py
|-- petstore_data.xml
|-- petstore.py
|-- petstore.xml
`-- static
    `-- src
        |-- css
        |   `-- petstore.css
        |-- js
        |   `-- petstore.js
        `-- xml
            `-- petstore.xml
```

The module already holds various server customizations. We'll come back to these later, for now let's focus on the web-related content, in the `static` folder.

Files used in the "web" side of an Odoo module must be placed in a `static` folder so they are available to a web browser, files outside that folder can not be fetched by browsers. The `src/css`, `src/js` and `src/xml` sub-folders are conventional and not strictly necessary.

`oepetstore/static/css/petstore.css` currently empty, will hold the CSS for pet store content

`oepetstore/static/xml/petstore.xml` Mostly empty, will hold *QWeb* templates

`oepetstore/static/js/petstore.js` The most important (and interesting) part, contains the logic of the application (or at least its web-browser side) as javascript. It should currently look like:

```
openerp.oepetstore = function(instance, local) {
    var _t = instance.web._t,
        _lt = instance.web._lt;
    var QWeb = instance.web.qweb;

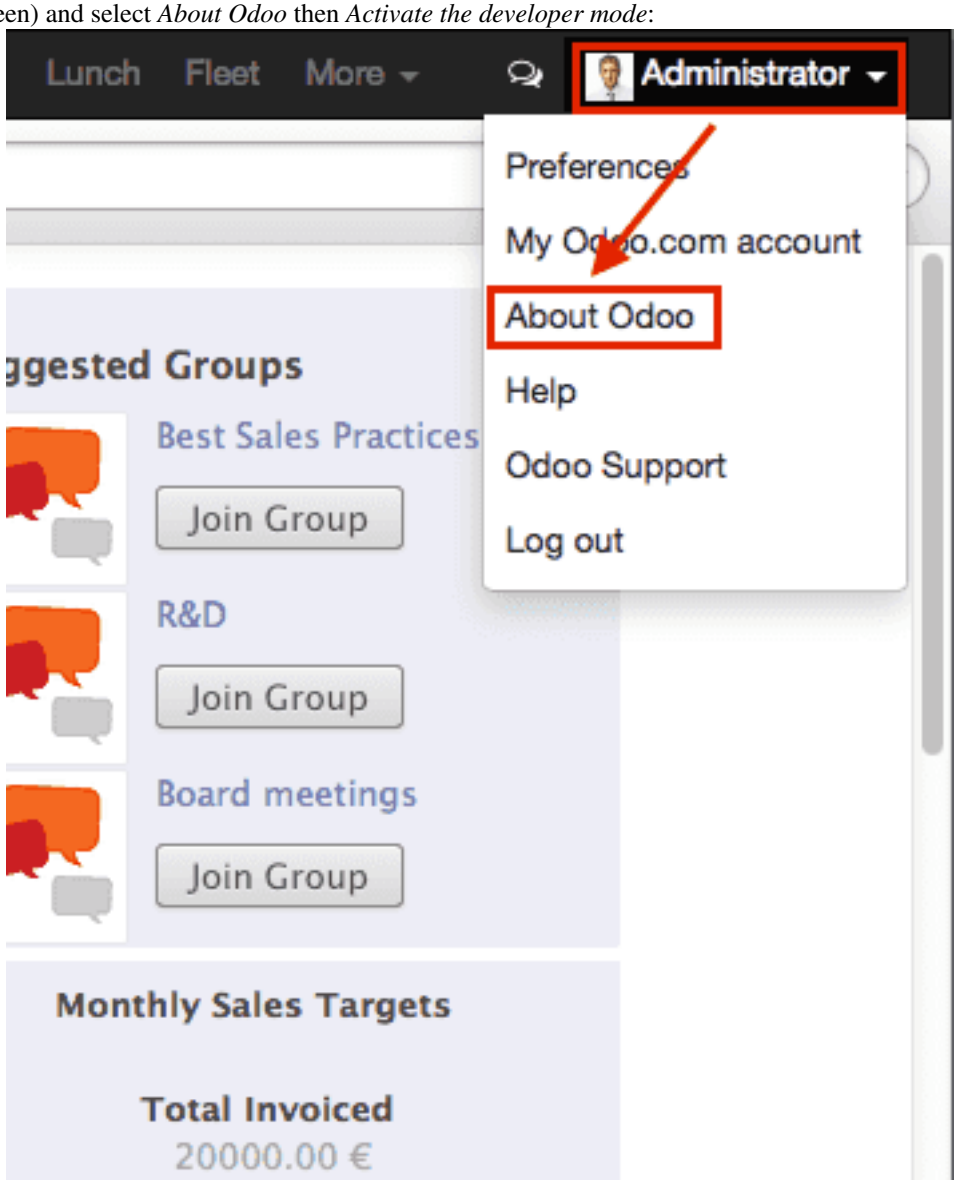
    local.HomePage = instance.Widget.extend({
        start: function() {
```

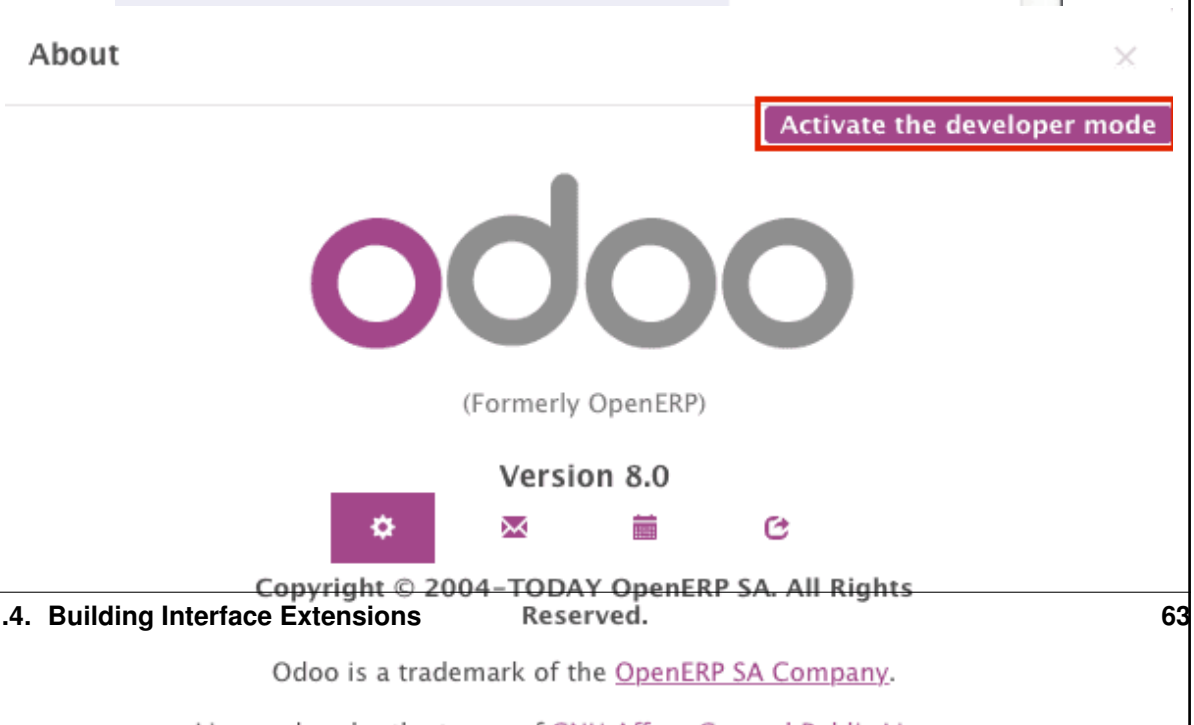
```
                console.log("pet store home page loaded");
            },
        });

        instance.web.client_actions.add(
            'petstore.homepage', 'instance.oepetstore.HomePage');
    }
```

Which only prints a small message in the browser's console.

Warning: All JavaScript files are concatenated and minified to improve application load time.
One of the drawback is debugging becomes more difficult as individual files disappear and the code is made significantly less readable. It is possible to disable this process by enabling the "developer mode": log into your Odoo instance (user *admin* password *admin* by default) open the user menu (in the top-right corner of the Odoo screen) and select *About Odoo* then *Activate the developer mode*:





1.4.2 Odoo JavaScript Module

Javascript doesn't have built-in modules. As a result variables defined in different files are all mashed together and may conflict. This has given rise to various module patterns used to build clean namespaces and limit risks of naming conflicts.

The Odoo framework uses one such pattern to define modules within web addons, in order to both namespace code and correctly order its loading.

`oepetstore/static/js/petstore.js` contains a module declaration:

```
openerp.oepetstore = function(instance, local) {
    local.xxx = ...;
}
```

In Odoo web, modules are declared as functions set on the global `openerp` variable. The function's name must be the same as the addon (in this case `oepetstore`) so the framework can find it, and automatically initialize it.

When the web client loads your module it will call the root function and provide two parameters:

- the first parameter is the current instance of the Odoo web client, it gives access to various capabilities defined by the Odoo (translations, network services) as well as objects defined by the core or by other modules.
- the second parameter is your own local namespace automatically created by the web client. Objects and variables which should be accessible from outside your module (either because the Odoo web client needs to call them or because others may want to customize them) should be set inside that namespace.

1.4.3 Classes

Much as modules, and contrary to most object-oriented languages, javascript does not build in *classes*[3] although it provides roughly equivalent (if lower-level and more verbose) mechanisms.

For simplicity and developer-friendliness Odoo web provides a class system based on John Resig's Simple JavaScript Inheritance.

New classes are defined by calling the `extend()` method of `openerp.web.Class()`:

```
var MyClass = instance.web.Class.extend({
    say_hello: function() {
        console.log("hello");
    },
});
```

The `extend()` method takes a dictionary describing the new class's content (methods and static attributes). In this case, it will only have a `say_hello` method which takes no parameters.

Classes are instantiated using the `new` operator:

```
var my_object = new MyClass();
my_object.say_hello();
// print "hello" in the console
```

And attributes of the instance can be accessed via `this`:

```
var MyClass = instance.web.Class.extend({
    say_hello: function() {
        console.log("hello", this.name);
    },
```

[3] as a separate concept from instances. In many languages classes are full-fledged objects and themselves instance (of metaclasses) but there remains two fairly separate hierarchies between classes and instances

```
});

var my_object = new MyClass();
my_object.name = "Bob";
my_object.say_hello();
// print "hello Bob" in the console
```

Classes can provide an initializer to perform the initial setup of the instance, by defining an `init()` method. The initializer receives the parameters passed when using the `new` operator:

```
var MyClass = instance.web.Class.extend({
    init: function(name) {
        this.name = name;
    },
    say_hello: function() {
        console.log("hello", this.name);
    },
});

var my_object = new MyClass("Bob");
my_object.say_hello();
// print "hello Bob" in the console
```

It is also possible to create subclasses from existing (used-defined) classes by calling `extend()` on the parent class, as is done to subclass `Class()`:

```
var MySpanishClass = MyClass.extend({
    say_hello: function() {
        console.log("hola", this.name);
    },
});

var my_object = new MySpanishClass("Bob");
my_object.say_hello();
// print "hola Bob" in the console
```

When overriding a method using inheritance, you can use `this._super()` to call the original method:

```
var MySpanishClass = MyClass.extend({
    say_hello: function() {
        this._super();
        console.log("translation in Spanish: hola", this.name);
    },
});

var my_object = new MySpanishClass("Bob");
my_object.say_hello();
// print "hello Bob \n translation in Spanish: hola Bob" in the console
```

Warning: `_super` is not a standard method, it is set on-the-fly to the next method in the current inheritance chain, if any. It is only defined during the *synchronous* part of a method call, for use in asynchronous handlers (after network calls or in `setTimeout` callbacks) a reference to its value should be retained, it should not be accessed via `this`:

```
// broken, will generate an error
say_hello: function () {
    setTimeout(function () {
        this._super();
    }.bind(this), 0);
}

// correct
say_hello: function () {
    // don't forget .bind()
    var _super = this._super.bind(this);
    setTimeout(function () {
        _super();
    }.bind(this), 0);
}
```

1.4.4 Widgets Basics

The Odoo web client bundles jQuery for easy DOM manipulation. It is useful and provides a better API than standard W3C DOM[4], but insufficient to structure complex applications leading to difficult maintenance.

Much like object-oriented desktop UI toolkits (e.g. Qt, Cocoa or GTK), Odoo Web makes specific components responsible for sections of a page. In Odoo web, the base for such components is the *`Widget()`* class, a component specialized in handling a page section and displaying information for the user.

Your First Widget

The initial demonstration module already provides a basic widget:

```
local.HomePage = instance.Widget.extend({
    start: function() {
        console.log("pet store home page loaded");
    },
});
```

It extends *`Widget()`* and overrides the standard method *`start()`*, which — much like the previous `MyClass` — does little for now.

This line at the end of the file:

```
instance.web.client_actions.add(
    'petstore.homepage', 'instance.oepetstore.HomePage');
```

registers our basic widget as a client action. Client actions will be explained later, for now this is just what allows our widget to be called and displayed when we select the *Pet Store* → *Pet Store* → *Home Page* menu.

Warning: because the widget will be called from outside our module, the web client needs its "fully qualified" name, not the local version.

[4] as well as papering over cross-browser differences, although this has become less necessary over time

Display Content

Widgets have a number of methods and features, but the basics are simple:

- set up a widget
- format the widget's data
- display the widget

The `HomePage` widget already has a *`start()`* method. That method is part of the normal widget lifecycle and automatically called once the widget is inserted in the page. We can use it to display some content.

All widgets have a *`$el`* which represents the section of page they're in charge of (as a jQuery object). Widget content should be inserted there. By default, *`$el`* is an empty `<div>` element.

A `<div>` element is usually invisible to the user if it has no content (or without specific styles giving it a size) which is why nothing is displayed on the page when `HomePage` is launched.

Let's add some content to the widget's root element, using jQuery:

```
local.HomePage = instance.Widget.extend({
    start: function() {
        this.$el.append("<div>Hello dear Odoo user!</div>");
    },
});
```

That message will now appear when you open *Pet Store* → *Pet Store* → *Home Page*

Note: to refresh the javascript code loaded in Odoo Web, you will need to reload the page. There is no need to restart the Odoo server.

The `HomePage` widget is used by Odoo Web and managed automatically. To learn how to use a widget "from scratch" let's create a new one:

```
local.GreetingsWidget = instance.Widget.extend({
    start: function() {
        this.$el.append("<div>We are so happy to see you again in this menu!</div>");
    },
});
```

We can now add our `GreetingsWidget` to the `HomePage` by using the `GreetingsWidget`'s *`appendTo()`* method:

```
local.HomePage = instance.Widget.extend({
    start: function() {
        this.$el.append("<div>Hello dear Odoo user!</div>");
        var greeting = new local.GreetingsWidget(this);
        return greeting.appendTo(this.$el);
    },
});
```

- `HomePage` first adds its own content to its DOM root
- `HomePage` then instantiates `GreetingsWidget`
- Finally it tells `GreetingsWidget` where to insert itself, delegating part of its *`$el`* to the `GreetingsWidget`.

When the *`appendTo()`* method is called, it asks the widget to insert itself at the specified position and to display its content. The *`start()`* method will be called during the call to *`appendTo()`*.

To see what happens under the displayed interface, we will use the browser's DOM Explorer. But first let's alter our widgets slightly so we can more easily find where they are, by *`adding a class to their root elements`*:

```
local.HomePage = instance.Widget.extend({
    className: 'oe_petstore_homepage',
    ...
});
local.GreetingsWidget = instance.Widget.extend({
    className: 'oe_petstore_greetings',
    ...
});
```

If you can find the relevant section of the DOM (right-click on the text then *Inspect Element*), it should look like this:

```
<div class="oe_petstore_homepage">
    <div>Hello dear Odoo user!</div>
    <div class="oe_petstore_greetings">
        <div>We are so happy to see you again in this menu!</div>
    </div>
</div>
```

Which clearly shows the two `<div>` elements automatically created by *`Widget()`*, because we added some classes on them.

We can also see the two message-holding divs we added ourselves

Finally, note the `<div class="oe_petstore_greetings">` element which represents the `GreetingsWidget` instance is *inside* the `<div class="oe_petstore_homepage">` which represents the `HomePage` instance, since we appended

Widget Parents and Children

In the previous part, we instantiated a widget using this syntax:

```
new local.GreetingsWidget(this);
```

The first argument is `this`, which in that case was a `HomePage` instance. This tells the widget being created which other widget is its *parent*.

As we've seen, widgets are usually inserted in the DOM by another widget and *inside* that other widget's root element. This means most widgets are "part" of another widget, and exist on behalf of it. We call the container the *parent*, and the contained widget the *child*.

Due to multiple technical and conceptual reasons, it is necessary for a widget to know who is its parent and who are its children.

`getParent()` can be used to get the parent of a widget:

```
    local.GreetingsWidget = instance.Widget.extend({
        start: function() {
            console.log(this.getParent().$el );
            // will print "div.oe_petstore_homepage" in the console
        },
    });
```

`getChildren()` can be used to get a list of its children:

```
    local.HomePage = instance.Widget.extend({
        start: function() {
            var greeting = new local.GreetingsWidget(this);
            greeting.appendTo(this.$el);
            console.log(this.getChildren()[0].$el);
            // will print "div.oe_petstore_greetings" in the console
        },
    });
```

When overriding the *init()* method of a widget it is *of the utmost importance* to pass the parent to the `this._super()` call, otherwise the relation will not be set up correctly:

```
local.GreetingsWidget = instance.Widget.extend({
    init: function(parent, name) {
        this._super(parent);
        this.name = name;
    },
});
```

Finally, if a widget does not have a parent (e.g. because it's the root widget of the application), `null` can be provided as parent:

```
new local.GreetingsWidget(null);
```

Destroying Widgets

If you can display content to your users, you should also be able to erase it. This is done via the *destroy()* method:

```
greeting.destroy();
```

When a widget is destroyed it will first call *destroy()* on all its children. Then it erases itself from the DOM. If you have set up permanent structures in *init()* or *start()* which must be explicitly cleaned up (because the garbage collector will not handle them), you can override *destroy()*.

Danger: when overriding *destroy()*, `_super()` *must always* be called otherwise the widget and its children are not correctly cleaned up leaving possible memory leaks and "phantom events", even if no error is displayed

1.4.5 The QWeb Template Engine

In the previous section we added content to our widgets by directly manipulating (and adding to) their DOM:

```
this.$el.append("<div>Hello dear Odoo user!</div>");
```

This allows generating and displaying any type of content, but gets unwieldy when generating significant amounts of DOM (lots of duplication, quoting issues, ...)

As many other environments, Odoo's solution is to use a template engine. Odoo's template engine is called *QWeb*.

QWeb is an XML-based templating language, similar to Genshi, Thymeleaf or Facelets. It has the following characteristics:

- It's implemented fully in JavaScript and rendered in the browser
- Each template file (XML files) contains multiple templates
- It has special support in Odoo Web's *Widget()*, though it can be used outside of Odoo's web client (and it's possible to use *Widget()* without relying on QWeb)

Note: The rationale behind using QWeb instead of existing javascript template engines is the extensibility of pre-existing (third-party) templates, much like Odoo *views*.

Most javascript template engines are text-based which precludes easy structural extensibility where an XML-based templating engine can be generically altered using e.g. XPath or CSS and a tree-alteration DSL (or even just XSLT). This flexibility and extensibility is a core characteristic of Odoo, and losing it was considered unacceptable.

Using QWeb

First let's define a simple QWeb template in the almost-empty `oepetstore/static/src/xml/petstore.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
    <t t-name="HomePageTemplate">
        <div style="background-color: red;">This is some simple HTML</div>
    </t>
</templates>
```

Now we can use this template inside of the `HomePage` widget. Using the `QWeb` loader variable defined at the top of the page, we can call to the template defined in the XML file:

```
local.HomePage = instance.Widget.extend({
    start: function() {
        this.$el.append(QWeb.render("HomePageTemplate"));
    },
});
```

`QWeb.render()` looks for the specified template, renders it to a string and returns the result.

However, because *`Widget()`* has special integration for QWeb the template can be set directly on the widget via its *`template`* attribute:

```
local.HomePage = instance.Widget.extend({
    template: "HomePageTemplate",
    start: function() {
        ...
    },
});
```

Although the result looks similar, there are two differences between these usages:

- with the second version, the template is rendered right before *`start()`* is called
- in the first version the template's content is added to the widget's root element, whereas in the second version the template's root element is directly *set as* the widget's root element. Which is why the "greetings" sub-widget also gets a red background

Warning: templates should have a single non-`t` root element, especially if they're set as a widget's *`template`*. If there are multiple "root elements", results are undefined (usually only the first root element will be used and the others will be ignored)

QWeb Context

QWeb templates can be given data and can contain basic display logic.

For explicit calls to `QWeb.render()`, the template data is passed as second parameter:

```
QWeb.render("HomePageTemplate", {name: "Klaus"});
```

with the template modified to:

```
<t t-name="HomePageTemplate">
    <div>Hello <t t-esc="name"/></div>
</t>
```

will result in:

```
<div>Hello Klaus</div>
```

When using *Widget()*'s integration it is not possible to provide additional data to the template. The template will be given a single `widget` context variable, referencing the widget being rendered right before *start()* is called (the widget's state will essentially be that set up by *init()*):

```
<t t-name="HomePageTemplate">
    <div>Hello <t t-esc="widget.name"/></div>
</t>
```

```
local.HomePage = instance.Widget.extend({
    template: "HomePageTemplate",
    init: function(parent) {
        this._super(parent);
        this.name = "Mordecai";
    },
    start: function() {
    },
});
```

Result:

```
<div>Hello Mordecai</div>
```

Template Declaration

We've seen how to *render* QWeb templates, let's now see the syntax of the templates themselves.

A QWeb template is composed of regular XML mixed with QWeb *directives*. A QWeb directive is declared with XML attributes starting with `t-`.

The most basic directive is `t-name`, used to declare new templates in a template file:

```
<templates>
    <t t-name="HomePageTemplate">
        <div>This is some simple HTML</div>
    </t>
</templates>
```

`t-name` takes the name of the template being defined, and declares that it can be called using `QWeb.render()`. It can only be used at the top-level of a template file.

Escaping

The `t-esc` directive can be used to output text:

```
<div>Hello <t t-esc="name"/></div>
```

It takes a Javascript expression which is evaluated, the result of the expression is then HTML-escaped and inserted in the document. Since it's an expression it's possible to provide just a variable name as above, or a more complex expression like a computation:

```
<div><t t-esc="3+5"/></div>
```

or method calls:

```
<div><t t-esc="name.toUpperCase()"/></div>
```

Outputting HTML

To inject HTML in the page being rendered, use `t-raw`. Like `t-esc` it takes an arbitrary Javascript expression as parameter, but it does not perform an HTML-escape step.

```
<div><t t-raw="name.link(user_account)"/></div>
```

Danger: `t-raw` *must not* be used on any data which may contain non-escaped user-provided content as this leads to cross-site scripting vulnerabilities

Conditionals

QWeb can have conditional blocks using `t-if`. The directive takes an arbitrary expression, if the expression is falsy (`false`, `null`, `0` or an empty string) the whole block is suppressed, otherwise it is displayed.

```
<div>
    <t t-if="true == true">
        true is true
    </t>
    <t t-if="true == false">
        true is not true
    </t>
</div>
```

Note: QWeb doesn't have an "else" structure, use a second `t-if` with the original condition inverted. You may want to store the condition in a local variable if it's a complex or expensive expression.

Iteration

To iterate on a list, use `t-foreach` and `t-as`. `t-foreach` takes an expression returning a list to iterate on `t-as` takes a variable name to bind to each item during iteration.

```
<div>
    <t t-foreach="names" t-as="name">
        <div>
            Hello <t t-esc="name"/>
        </div>
    </t>
</div>
```

Note: `t-foreach` can also be used with numbers and objects (dictionaries)

Defining attributes

QWeb provides two related directives to define computed attributes: `t-att-`*name* and `t-attf-`*name*. In either case, *name* is the name of the attribute to create (e.g. `t-att-id` defines the attribute `id` after rendering).

`t-att-` takes a javascript expression whose result is set as the attribute's value, it is most useful if all of the attribute's value is computed:

```
<div>
    Input your name:
    <input type="text" t-att-value="defaultName"/>
</div>
```

`t-attf-` takes a *format string*. A format string is literal text with interpolation blocks inside, an interpolation block is a javascript expression between `{{` and `}}`, which will be replaced by the result of the expression. It is most useful for attributes which are partially literal and partially computed such as a class:

```
<div t-attf-class="container {{ left ? 'text-left' : '' }} {{ extra_class }}">
    insert content here
</div>
```

Calling other templates

Templates can be split into sub-templates (for simplicity, maintainability, reusability or to avoid excessive markup nesting).

This is done using the `t-call` directive, which takes the name of the template to render:

```
<t t-name="A">
    <div class="i-am-a">
        <t t-call="B"/>
    </div>
</t>
<t t-name="B">
    <div class="i-am-b"/>
</t>
```

rendering the `A` template will result in:

```
<div class="i-am-a">
    <div class="i-am-b"/>
</div>
```

Sub-templates inherit the rendering context of their caller.

To Learn More About QWeb

For a QWeb reference, see *QWeb*.

Exercise

U sage of QWeb in Widgets

Create a widget whose constructor takes two parameters aside from `parent`: `product_names` and `color`.

- `product_names` should an array of strings, each one the name of a product
- `color` is a string containing a color in CSS color format (ie: `#000000` for black).

The widget should display the given product names one under the other, each one in a separate box with a background color with the value of `color` and a border. You should use QWeb to render the HTML. Any necessary CSS should be in `oepetstore/static/src/css/petstore.css`.

Use the widget in `HomePage` with half a dozen products.

1.4.6 Widget Helpers

`Widget`'s jQuery Selector

Selecting DOM elements within a widget can be performed by calling the `find()` method on the widget's DOM root:

```
this.$el.find("input.my_input")...
```

But because it's a common operation, *`Widget()`* provides an equivalent shortcut through the *`$()`* method:

```
local.MyWidget = instance.Widget.extend({
    start: function() {
        this.$("input.my_input")...
    },
});
```

Warning: The global jQuery function `$()` should *never* be used unless it is absolutely necessary: selection on a widget's root are scoped to the widget and local to it, but selections with `$()` are global to the page/application and may match parts of other widgets and views, leading to odd or dangerous side-effects. Since a widget should generally act only on the DOM section it owns, there is no cause for global selection.

Easier DOM Events Binding

We have previously bound DOM events using normal jQuery event handlers (e.g. `.click()` or `.change()`) on widget elements:

```
local.MyWidget = instance.Widget.extend({
    start: function() {
        var self = this;
        this.$(".my_button").click(function() {
            self.button_clicked();
        });
    },
    button_clicked: function() {
        ..
    },
});
```

While this works it has a few issues:

1. it is rather verbose
2. it does not support replacing the widget's root element at runtime as the binding is only performed when `start()` is run (during widget initialization)
3. it requires dealing with `this`-binding issues

Widgets thus provide a shortcut to DOM event binding via *`events`*:

```
local.MyWidget = instance.Widget.extend({
    events: {
        "click .my_button": "button_clicked",
    },
    button_clicked: function() {
        ..
    }
});
```

`events` is an object (mapping) of an event to the function or method to call when the event is triggered:

- the key is an event name, possibly refined with a CSS selector in which case only if the event happens on a selected sub-element will the function or method run: `click` will handle all clicks within the widget, but `click .my_button` will only handle clicks in elements bearing the `my_button` class
- the value is the action to perform when the event is triggered

  It can be either a function:

```
    events: {
        'click': function (e) { /* code here */ }
    }
```

  or the name of a method on the object (see example above).

In either case, the `this` is the widget instance and the handler is given a single parameter, the jQuery event object for the event.

1.4.7 Widget Events and Properties

Events

Widgets provide an event system (separate from the DOM/jQuery event system described above): a widget can fire events on itself, and other widgets (or itself) can bind themselves and listen for these events:

```
local.ConfirmWidget = instance.Widget.extend({
    events: {
        'click button.ok_button': function () {
            this.trigger('user_chose', true);
        },
        'click button.cancel_button': function () {
            this.trigger('user_chose', false);
        }
    },
    start: function() {
        this.$el.append("<div>Are you sure you want to perform this action?</div>" +
            "<button class='ok_button'>Ok</button>" +
            "<button class='cancel_button'>Cancel</button>");
    },
});
```

This widget acts as a facade, transforming user input (through DOM events) into a documentable internal event to which parent widgets can bind themselves.

`trigger()` takes the name of the event to trigger as its first (mandatory) argument, any further arguments are treated as event data and passed directly to listeners.

We can then set up a parent event instantiating our generic widget and listening to the `user_chose` event using `on()`:

```
local.HomePage = instance.Widget.extend({
    start: function() {
        var widget = new local.ConfirmWidget(this);
        widget.on("user_chose", this, this.user_chose);
        widget.appendTo(this.$el);
    },
    user_chose: function(confirm) {
        if (confirm) {
            console.log("The user agreed to continue");
        } else {
            console.log("The user refused to continue");
        }
    },
});
```

`on()` binds a function to be called when the event identified by `event_name` is. The `func` argument is the function to call and `object` is the object to which that function is related if it is a method. The bound function will be called with the additional arguments of `trigger()` if it has any. Example:

```
start: function() {
    var widget = ...
    widget.on("my_event", this, this.my_event_triggered);
    widget.trigger("my_event", 1, 2, 3);
},
my_event_triggered: function(a, b, c) {
    console.log(a, b, c);
    // will print "1 2 3"
}
```

Note: Triggering events on an other widget is generally a bad idea. The main exception to that rule is `openerp.web.bus` which exists specifically to broadcasts evens in which any widget could be interested throughout the Odoo web application.

Properties

Properties are very similar to normal object attributes in that they allow storing data on a widget instance, however they have the additional feature that they trigger events when set:

```
start: function() {
    this.widget = ...
    this.widget.on("change:name", this, this.name_changed);
    this.widget.set("name", "Nicolas");
},
name_changed: function() {
    console.log("The new value of the property 'name' is", this.widget.get("name"));
}
```

- `set()` sets the value of a property and triggers `change:`*propname* (where *propname* is the property name passed as first parameter to `set()`) and `change`
- `get()` retrieves the value of a property.

Exercise

W idget Properties and Events

Create a widget `ColorInputWidget` that will display 3 `<input type="text">`. Each of these `<input>` is dedicated to type a hexadecimal number from 00 to FF. When any of these `<input>` is modified by the user the widget must query the content of the three `<input>`, concatenate their values to have a complete CSS color code (ie: `#00FF00`) and put the result in a property named `color`. Please note the jQuery `change()` event that you can bind on any HTML `<input>` element and the `val()` method that can query the current value of that `<input>` could be useful to you for this exercise.

Then, modify the `HomePage` widget to instantiate `ColorInputWidget` and display it. The `HomePage` widget should also display an empty rectangle. That rectangle must always, at any moment, have the same background color as the color in the `color` property of the `ColorInputWidget` instance.

Use QWeb to generate all HTML.

1.4.8 Modify existing widgets and classes

The class system of the Odoo web framework allows direct modification of existing classes using the `include()` method:

```
var TestClass = instance.web.Class.extend({
    testMethod: function() {
        return "hello";
    },
});

TestClass.include({
    testMethod: function() {
        return this._super() + " world";
    },
});
```

```
console.log(new TestClass().testMethod());
// will print "hello world"
```

This system is similar to the inheritance mechanism, except it will alter the target class in-place instead of creating a new class.

In that case, `this._super()` will call the original implementation of a method being replaced/redefined. If the class already had sub-classes, all calls to `this._super()` in sub-classes will call the new implementations defined in the call to `include()`. This will also work if some instances of the class (or of any of its sub-classes) were created prior to the call to `include()`.

1.4.9 Translations

The process to translate text in Python and JavaScript code is very similar. You could have noticed these lines at the beginning of the `petstore.js` file:

```
var _t = instance.web._t,
    _lt = instance.web._lt;
```

These lines are simply used to import the translation functions in the current JavaScript module. They are used thus:

```
this.$el.text(_t("Hello user!"));
```

In Odoo, translations files are automatically generated by scanning the source code. All piece of code that calls a certain function are detected and their content is added to a translation file that will then be sent to the translators. In Python, the function is `_()`. In JavaScript the function is `_t()` (and also `_lt()`).

`_t()` will return the translation defined for the text it is given. If no translation is defined for that text, it will return the original text as-is.

Note: To inject user-provided values in translatable strings, it is recommended to use _.str.sprintf with named arguments *after* the translation:

```
this.$el.text(_.str.sprintf(
    _t("Hello, %(user)s!"), {
    user: "Ed"
}));
```

This makes translatable strings more readable to translators, and gives them more flexibility to reorder or ignore parameters.

`_lt()` ("lazy translate") is similar but somewhat more complex: instead of translating its parameter immediately, it returns an object which, when converted to a string, will perform the translation.

It is used to define translatable terms before the translations system is initialized, for class attributes for instance (as modules are loaded before the user's language is configured and translations are downloaded).

1.4.10 Communication with the Odoo Server

Contacting Models

Most operations with Odoo involve communicating with *models* implementing business concern, these models will then (potentially) interact with some storage engine (usually PostgreSQL).

Although jQuery provides a $.ajax function for network interactions, communicating with Odoo requires additional metadata whose setup before every call would be verbose and error-prone. As a result, Odoo web provides higher-level communication primitives.

To demonstrate this, the file `petstore.py` already contains a small model with a sample method:

```
class message_of_the_day(models.Model):
    _name = "oepetstore.message_of_the_day"

    @api.model
    def my_method(self):
        return {"hello": "world"}

    message = fields.Text(),
    color = fields.Char(size=20),
```

This declares a model with two fields, and a method `my_method()` which returns a literal dictionary.

Here is a sample widget that calls `my_method()` and displays the result:

```
local.HomePage = instance.Widget.extend({
    start: function() {
        var self = this;
        var model = new instance.web.Model("oepetstore.message_of_the_day");
        model.call("my_method", {context: new instance.web.CompoundContext()}).then(function(resul
            self.$el.append("<div>Hello " + result["hello"] + "</div>");
            // will show "Hello world" to the user
        });
    },
});
```

The class used to call Odoo models is *openerp.Model()*. It is instantiated with the Odoo model's name as first parameter (`oepetstore.message_of_the_day` here).

`call()` can be used to call any (public) method of an Odoo model. It takes the following positional arguments:

name The name of the method to call, `my_method` here

args an array of positional arguments to provide to the method. Because the example has no positional argument to provide, the `args` parameter is not provided.

Here is an other example with positional arguments:

```
    @api.model
    def my_method2(self, a, b, c): ...
```

```
    model.call("my_method", [1, 2, 3], ...
    // with this a=1, b=2 and c=3
```

kwargs a mapping of keyword arguments to pass. The example provides a single named argument `context`.

```
    @api.model
    def my_method2(self, a, b, c): ...
```

```
    model.call("my_method", [], {a: 1, b: 2, c: 3, ...
    // with this a=1, b=2 and c=3
```

`call()` returns a deferred resolved with the value returned by the model's method as first argument.

CompoundContext

The previous section used a `context` argument which was not explained in the method call:

```
model.call("my_method", {context: new instance.web.CompoundContext()})
```

The context is like a "magic" argument that the web client will always give to the server when calling a method. The context is a dictionary containing multiple keys. One of the most important key is the language of the user, used by the server to translate all the messages of the application. Another one is the time zone of the user, used to compute correctly dates and times if Odoo is used by people in different countries.

The `argument` is necessary in all methods, otherwise bad things could happen (such as the application not being translated correctly). That's why, when you call a model's method, you should always provide that argument. The solution to achieve that is to use `openerp.web.CompoundContext()`.

`CompoundContext()` is a class used to pass the user's context (with language, time zone, etc...) to the server as well as adding new keys to the context (some models' methods use arbitrary keys added to the context). It is created by giving to its constructor any number of dictionaries or other `CompoundContext()` instances. It will merge all those contexts before sending them to the server.

```
model.call("my_method", {context: new instance.web.CompoundContext({'new_key': 'key_value'})})
```

```
@api.model
def my_method(self):
    print self.env.context
    // will print: {'lang': 'en_US', 'new_key': 'key_value', 'tz': 'Europe/Brussels', 'uid': 1}
```

You can see the dictionary in the argument `context` contains some keys that are related to the configuration of the current user in Odoo plus the `new_key` key that was added when instantiating `CompoundContext()`.

Queries

While *`call()`* is sufficient for any interaction with Odoo models, Odoo Web provides a helper for simpler and clearer querying of models (fetching of records based on various conditions): *`query()`* which acts as a shortcut for the common combination of *`search()`* and :*`read()`*. It provides a clearer syntax to search and read models:

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

versus:

```
model.call('search', [['active', '=', true], ['company_id', '=', main_company]], {limit: 15})
    .then(function (ids) {
        return model.call('read', [ids, ['name', 'login', 'user_email', 'signature']]);
    })
    .then(function (users) {
        // do work with users records
    });
```

- `query()` takes an optional list of fields as parameter (if no field is provided, all fields of the model are fetched). It returns a *`openerp.web.Query()`* which can be further customized before being executed
- *`Query()`* represents the query being built. It is immutable, methods to customize the query actually return a modified copy, so it's possible to use the original and the new version side-by-side. See *`Query()`* for its customization options.

When the query is set up as desired, simply call *`all()`* to execute it and return a deferred to its result. The result is the same as *`read()`*'s, an array of dictionaries where each dictionary is a requested record, with each requested field a dictionary key.

1.4.11 Exercises

M essage of the Day

Create a `MessageOfTheDay` widget displaying the last record of the `oepetstore.message_of_the_day` model. The widget should fetch its record as soon as it is displayed.

Display the widget in the Pet Store home page.

P et Toys List

Create a `PetToysList` widget displaying 5 toys (using their name and their images).

The pet toys are not stored in a new model, instead they're stored in `product.product` using a special category *Pet Toys*. You can see the pre-generated toys and add new ones by going to *Pet Store* → *Pet Store* → *Pet Toys*. You will probably need to explore `product.product` to create the right domain to select just pet toys.

In Odoo, images are generally stored in regular fields encoded as base64, HTML supports displaying images straight from base64 with `<img src="data:mime_type; base64,base64_image_data"/>`

The `PetToysList` widget should be displayed on the home page on the right of the `MessageOfTheDay` widget. You will need to make some layout with CSS to achieve this.

1.4.12 Existing web components

The Action Manager

In Odoo, many operations start from an *action*: opening a menu item (to a view), printing a report, ...

Actions are pieces of data describing how a client should react to the activation of a piece of content. Actions can be stored (and read through a model) or they can be generated on-the fly (locally to the client by javascript code, or remotely by a method of a model).

In Odoo Web, the component responsible for handling and reacting to these actions is the *Action Manager*.

Using the Action Manager

The action manager can be invoked explicitly from javascript code by creating a dictionary describing *an action* of the right type, and calling an action manager instance with it.

`do_action()` is a shortcut of *`Widget()`* looking up the "current" action manager and executing the action:

```
instance.web.TestWidget = instance.Widget.extend({
    dispatch_to_new_action: function() {
        this.do_action({
            type: 'ir.actions.act_window',
            res_model: "product.product",
            res_id: 1,
            views: [[false, 'form']],
            target: 'current',
            context: {},
        });
    },
});
```

The most common action `type` is `ir.actions.act_window` which provides views to a model (displays a model in various manners), its most common attributes are:

`res_model` The model to display in views

`res_id` (optional) For form views, a preselected record in `res_model`

`views` Lists the views available through the action. A list of `[view_id, view_type]`, `view_id` can either be the database identifier of a view of the right type, or `false` to use the view by default for the specified type. View types can not be present multiple times. The action will open the first view of the list by default.

`target` Either `current` (the default) which replaces the "content" section of the web client by the action, or `new` to open the action in a dialog box.

`context` Additional context data to use within the action.

J ump to Product

Modify the `PetToysList` component so clicking on a toy replaces the homepage by the toy's form view.

Client Actions

Throughout this guide, we used a simple `HomePage` widget which the web client automatically starts when we select the right menu item. But how did the Odoo web know to start this widget? Because the widget is registered as a *client action*.

A client action is (as its name implies) an action type defined almost entirely in the client, in javascript for Odoo web. The server simply sends an action tag (an arbitrary name), and optionally adds a few parameters, but beyond that *everything* is handled by custom client code.

Our widget is registered as the handler for the client action through this:

```
instance.web.client_actions.add('petstore.homepage', 'instance.oepetstore.HomePage');
```

`instance.web.client_actions` is a `Registry()` in which the action manager looks up client action handlers when it needs to execute one. The first parameter of `add()` is the name (tag) of the client action, and the second parameter is the path to the widget from the Odoo web client root.

When a client action must be executed, the action manager looks up its tag in the registry, walks the specified path and displays the widget it finds at the end.

Note: a client action handler can also be a regular function, in whch case it'll be called and its result (if any) will be interpreted as the next action to execute.

On the server side, we had simply defined an `ir.actions.client` action:

```
<record id="action_home_page" model="ir.actions.client">
    <field name="tag">petstore.homepage</field>
</record>
```

and a menu opening the action:

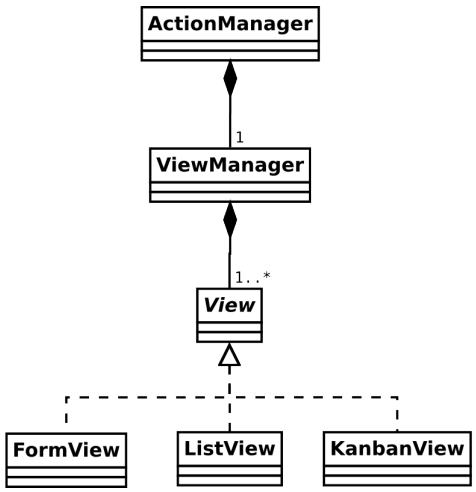
```
<menuitem id="home_page_petstore_menu" parent="petstore_menu"
          name="Home Page" action="action_home_page"/>
```

Architecture of the Views

Much of Odoo web's usefulness (and complexity) resides in views. Each view type is a way of displaying a model in the client.

The View Manager

When an `ActionManager` instance receive an action of type `ir.actions.act_window`, it delegates the synchronization and handling of the views themselves to a *view manager*, which will then set up one or multiple views depending on the original action's requirements:



The Views

Most *Odoo views* are implemented through a subclass of `openerp.web.View()` which provides a bit of generic basic structure for handling events and displaying model information.

The *search view* is considered a view type by the main Odoo framework, but handled separately by the web client (as it's a more permanent fixture and can interact with other views, which regular views don't do).

A view is responsible for loading its own description XML (using *`fields_view_get`*) and any other data source it needs. To that purpose, views are provided with an optional view identifier set as the `view_id` attribute.

Views are also provided with a `DataSet()` instance which holds most necessary model information (the model name and possibly various record ids).

Views may also want to handle search queries by overriding `do_search()`, and updating their `DataSet()` as necessary.

The Form View Fields

A common need is the extension of the web form view to add new ways of displaying fields.

All built-in fields have a default display implementation, a new form widget may be necessary to correctly interact with a new field type (e.g. a GIS field) or to provide new representations and ways to interact with existing field types (e.g. validate *`Char`* fields which should contain email addresses and display them as email links).

To explicitly specify which form widget should be used to display a field, simply use the `widget` attribute in the view's XML description:

```
<field name="contact_mail" widget="email"/>
```

Note:

- the same widget is used in both "view" (read-only) and "edition" modes of a form view, it's not possible to use a widget in one and an other widget in the other
- and a given field (name) can not be used multiple times in the same form
- a widget may ignore the current mode of the form view and remain the same in both view and edition

---

Fields are instantiated by the form view after it has read its XML description and constructed the corresponding HTML representing that description. After that, the form view will communicate with the field objects using some methods. These methods are defined by the `FieldInterface` interface. Almost all fields inherit the `AbstractField` abstract class. That class defines some default mechanisms that need to be implemented by most fields.

Here are some of the responsibilities of a field class:

- The field class must display and allow the user to edit the value of the field.
- It must correctly implement the 3 field attributes available in all fields of Odoo. The `AbstractField` class already implements an algorithm that dynamically calculates the value of these attributes (they can change at any moment because their value change according to the value of other fields). Their values are stored in *Widget Properties* (the widget properties were explained earlier in this guide). It is the responsibility of each field class to check these widget properties and dynamically adapt depending of their values. Here is a description of each of these attributes:
  - `required`: The field must have a value before saving. If `required` is `true` and the field doesn't have a value, the method `is_valid()` of the field must return `false`.
  - `invisible`: When this is `true`, the field must be invisible. The `AbstractField` class already has a basic implementation of this behavior that fits most fields.
  - `readonly`: When `true`, the field must not be editable by the user. Most fields in Odoo have a completely different behavior depending on the value of `readonly`. As example, the `FieldChar` displays an HTML `<input>` when it is editable and simply displays the text when it is read-only. This also means it has much more code it would need to implement only one behavior, but this is necessary to ensure a good user experience.
- Fields have two methods, `set_value()` and `get_value()`, which are called by the form view to give it the value to display and get back the new value entered by the user. These methods must be able to handle the value as given by the Odoo server when a `read()` is performed on a model and give back a valid value for a `write()`. Remember that the JavaScript/Python data types used to represent the values given by `read()` and given to `write()` is not necessarily the same in Odoo. As example, when you read a many2one, it is always a tuple whose first value is the id of the pointed record and the second one is the name get (ie: `(15, "Agrolait")`). But when you write a many2one it must be a single integer, not a tuple anymore. `AbstractField` has a default implementation of these methods that works well for simple data type and set a widget property named `value`.

Please note that, to better understand how to implement fields, you are strongly encouraged to look at the definition of the `FieldInterface` interface and the `AbstractField` class directly in the code of the Odoo web client.

Creating a New Type of Field

In this part we will explain how to create a new type of field. The example here will be to re-implement the `FieldChar` class and progressively explain each part.

Simple Read-Only Field Here is a first implementation that will only display text. The user will not be able to modify the content of the field.

```
local.FieldChar2 = instance.web.form.AbstractField.extend({
    init: function() {
        this._super.apply(this, arguments);
        this.set("value", "");
    },
    render_value: function() {
        this.$el.text(this.get("value"));
    },
});

instance.web.form.widgets.add('char2', 'instance.oepetstore.FieldChar2');
```

In this example, we declare a class named `FieldChar2` inheriting from `AbstractField`. We also register this class in the registry `instance.web.form.widgets` under the key `char2`. That will allow us to use this new field in any form view by specifying `widget="char2"` in the `<field/>` tag in the XML declaration of the view.

In this example, we define a single method: `render_value()`. All it does is display the widget property `value`. Those are two tools defined by the `AbstractField` class. As explained before, the form view will call the method `set_value()` of the field to set the value to display. This method already has a default implementation in `AbstractField` which simply sets the widget property `value`. `AbstractField` also watch the `change:value` event on itself and calls the `render_value()` when it occurs. So, `render_value()` is a convenience method to implement in child classes to perform some operation each time the value of the field changes.

In the `init()` method, we also define the default value of the field if none is specified by the form view (here we assume the default value of a `char` field should be an empty string).

Read-Write Field Read-only fields, which only display content and don't allow the user to modify it can be useful, but most fields in Odoo also allow editing. This makes the field classes more complicated, mostly because fields are supposed to handle both editable and non-editable mode, those modes are often completely different (for design and usability purpose) and the fields must be able to switch between modes at any moment.

To know in which mode the current field should be, the `AbstractField` class sets a widget property named `effective_readonly`. The field should watch for changes in that widget property and display the correct mode accordingly. Example:

```
local.FieldChar2 = instance.web.form.AbstractField.extend({
    init: function() {
        this._super.apply(this, arguments);
        this.set("value", "");
    },
    start: function() {
        this.on("change:effective_readonly", this, function() {
            this.display_field();
            this.render_value();
        });
        this.display_field();
        return this._super();
    },
    display_field: function() {
        var self = this;
        this.$el.html(QWeb.render("FieldChar2", {widget: this}));
        if (! this.get("effective_readonly")) {
            this.$("input").change(function() {
                self.internal_set_value(self.$("input").val());
            });
        }
    },
    render_value: function() {
        if (this.get("effective_readonly")) {
            this.$el.text(this.get("value"));
        } else {
            this.$("input").val(this.get("value"));
        }
    },
});

instance.web.form.widgets.add('char2', 'instance.oepetstore.FieldChar2');
```

```
<t t-name="FieldChar2">
    <div class="oe_field_char2">
        <t t-if="! widget.get('effective_readonly')">
            <input type="text"></input>
        </t>
    </div>
</t>
```

In the `start()` method (which is called immediately after a widget has been appended to the DOM), we bind

on the event `change:effective_readonly`. That allows us to redisplay the field each time the widget property `effective_readonly` changes. This event handler will call `display_field()`, which is also called directly in `start()`. This `display_field()` was created specifically for this field, it's not a method defined in `AbstractField` or any other class. We can use this method to display the content of the field depending on the current mode.

From now on the conception of this field is typical, except there is a lot of verifications to know the state of the `effective_readonly` property:

- In the QWeb template used to display the content of the widget, it displays an `<input type="text" />` if we are in read-write mode and nothing in particular in read-only mode.
- In the `display_field()` method, we have to bind on the `change` event of the `<input type="text" />` to know when the user has changed the value. When it happens, we call the `internal_set_value()` method with the new value of the field. This is a convenience method provided by the `AbstractField` class. That method will set a new value in the `value` property but will not trigger a call to `render_value()` (which is not necessary since the `<input type="text" />` already contains the correct value).
- In `render_value()`, we use a completely different code to display the value of the field depending if we are in read-only or in read-write mode.

C reate a Color Field

Create a `FieldColor` class. The value of this field should be a string containing a color code like those used in CSS (example: `#FF0000` for red). In read-only mode, this color field should display a little block whose color corresponds to the value of the field. In read-write mode, you should display an `<input type="color" />`. That type of `<input />` is an HTML5 component that doesn't work in all browsers but works well in Google Chrome. So it's OK to use as an exercise.

You can use that widget in the form view of the `message_of_the_day` model for its field named `color`. As a bonus, you can change the `MessageOfTheDay` widget created in the previous part of this guide to display the message of the day with the background color indicated in the `color` field.

The Form View Custom Widgets

Form fields are used to edit a single field, and are intrinsically linked to a field. Because this may be limiting, it is also possible to create *form widgets* which are not so restricted and have less ties to a specific lifecycle.

Custom form widgets can be added to a form view through the `widget` tag:

```
<widget type="xxx" />
```

This type of widget will simply be created by the form view during the creation of the HTML according to the XML definition. They have properties in common with the fields (like the `effective_readonly` property) but they are not assigned a precise field. And so they don't have methods like `get_value()` and `set_value()`. They must inherit from the `FormWidget` abstract class.

Form widgets can interact with form fields by listening for their changes and fetching or altering their values. They can access form fields through their `field_manager` attribute:

```
local.WidgetMultiplication = instance.web.form.FormWidget.extend({
    start: function() {
        this._super();
        this.field_manager.on("field_changed:integer_a", this, this.display_result);
        this.field_manager.on("field_changed:integer_b", this, this.display_result);
        this.display_result();
    },
    display_result: function() {
        var result = this.field_manager.get_field_value("integer_a") *
                     this.field_manager.get_field_value("integer_b");
        this.$el.text("a*b = " + result);
```

```
    }
});

instance.web.form.custom_widgets.add('multiplication', 'instance.oepetstore.WidgetMultiplication'
```

`FormWidget` is generally the `FormView()` itself, but features used from it should be limited to those defined by `FieldManagerMixin()`, the most useful being:

- `get_field_value(field_name)()` which returns the value of a field.
- `set_values(values)()` sets multiple field values, takes a mapping of `{field_name: value_to_set}`
- An event `field_changed:`*`field_name`* is triggered any time the value of the field called `field_name` is changed

---

S how Coordinates on Google Map

Add two fields to `product.product` storing a latitude and a longitude, then create a new form widget to display the latitude and longitude of a product's origin on a map

To display the map, use Google Map's embedding:

```
<iframe width="400" height="300" src="https://maps.google.com/?ie=UTF8&amp;ll=XXX,YYY&amp;output=
</iframe>
```

where `XXX` should be replaced by the latitude and `YYY` by the longitude.

Display the two position fields and a map widget using them in a new notebook page of the product's form view.

G et the Current Coordinate

Add a button resetting the product's coordinates to the location of the user, you can get these coordinates using the javascript geolocation API.

Now we would like to display an additional button to automatically set the coordinates to the location of the current user.

To get the coordinates of the user, an easy way is to use the geolocation JavaScript API. See the online documentation to know how to use it.

Please also note that the user should not be able to click on that button when the form view is in read-only mode. So, this custom widget should handle correctly the `effective_readonly` property just like any field. One way to do this would be to make the button disappear when `effective_readonly` is true.

CHAPTER

TWO

WEB SERVICE API

Odoo is usually extended internally via modules, but many of its features and all of its data are also available from the outside for external analysis or integration with various tools. Part of the *Model Reference* API is easily available over XML-RPC and accessible from a variety of languages.

2.1 Connection

2.1.1 Configuration

If you already have an Odoo server installed, you can just use its parameters

Warning: For Odoo Online instances (<domain>.odoo.com), users are created without a *local* password (as a person you are logged in via the Odoo Online authentication system, not by the instance itself). To use XML-RPC on Odoo Online instances, you will need to set a password on the user account you want to use:

- Log in your instance with an administrator account
- Go to *Settings* → *Users* → *Users*
- Click on the user you want to use for XML-RPC access
- Click the *Change Password* button
- Set a *New Password* value then click *Change Password*.

The *server url* is the instance's domain (e.g. *https://mycompany.odoo.com*), the *database name* is the name of the instance (e.g. *mycompany*). The *username* is the configured user's login as shown by the *Change Password* screen.

demo

To make exploration simpler, you can also ask https://demo.odoo.com for a test database:

2.1.2 Logging in

Odoo requires users of the API to be authenticated before they can query most data.

The `xmlrpc/2/common` endpoint provides meta-calls which don't require authentication, such as the authentication itself or fetching version information. To verify if the connection information is correct before trying to authenticate, the simplest call is to ask for the server's version. The authentication itself is done through the `authenticate` function and returns a user identifier (`uid`) used in authenticated calls instead of the login.

```
{
    "server_version": "8.0",
    "server_version_info": [8, 0, 0, "final", 0],
    "server_serie": "8.0",
    "protocol_version": 1,
}
```

2.2 Calling methods

The second endpoint is `xmlrpc/2/object`, is used to call methods of odoo models via the `execute_kw` RPC function.

Each call to `execute_kw` takes the following parameters:

- the database to use, a string
- the user id (retrieved through `authenticate`), an integer
- the user's password, a string
- the model name, a string
- the method name, a string
- an array/list of parameters passed by position
- a mapping/dict of parameters to pass by keyword (optional)

For instance to see if we can read the `res.partner` model we can call `check_access_rights` with `operation` passed by position and `raise_exception` passed by keyword (in order to get a true/false result rather than true/error):

```
true
```

2.2.1 List records

Records can be listed and filtered via *`search()`*.

`search()` takes a mandatory *domain* filter (possibly empty), and returns the database identifiers of all records matching the filter. To list customer companies for instance:

```
[7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23, 28, 74]
```

Pagination

By default a search will return the ids of all records matching the condition, which may be a huge number. `offset` and `limit` parameters are available to only retrieve a subset of all matched records.

```
[13, 20, 30, 22, 29]
```

2.2.2 Count records

Rather than retrieve a possibly gigantic list of records and count them, *`search_count()`* can be used to retrieve only the number of records matching the query. It takes the same *domain* filter as *`search()`* and no other parameter.

```
19
```

Warning: calling `search` then `search_count` (or the other way around) may not yield coherent results if other users are using the server: stored data could have changed between the calls

2.2.3 Read records

Record data is accessible via the *`read()`* method, which takes a list of ids (as returned by *`search()`*) and optionally a list of fields to fetch. By default, it will fetch all the fields the current user can read, which tends to be a huge amount.

```
121
```

Conversedly, picking only three fields deemed interesting.

```
[{"comment": false, "country_id": [21, "Belgium"], "id": 7, "name": "Agrolait"}]
```

Note: even if the `id` field is not requested, it is always returned

2.2.4 Listing record fields

`fields_get()` can be used to inspect a model's fields and check which ones seem to be of interest.

Because it returns a large amount of meta-information (it is also used by client programs) it should be filtered before printing, the most interesting items for a human user are `string` (the field's label), `help` (a help text if available) and `type` (to know which values to expect, or to send when updating a record):

```
{
    "ean13": {
        "type": "char",
        "help": "BarCode",
        "string": "EAN13"
    },
    "property_account_position": {
        "type": "many2one",
        "help": "The fiscal position will determine taxes and accounts used for the partner.",
        "string": "Fiscal Position"
    },
    "signup_valid": {
        "type": "boolean",
        "help": "",
        "string": "Signup Token is Valid"
    },
    "date_localization": {
        "type": "date",
        "help": "",
        "string": "Geo Localization Date"
    },
    "ref_companies": {
        "type": "one2many",
        "help": "",
        "string": "Companies that refers to partner"
    },
    "sale_order_count": {
        "type": "integer",
        "help": "",
        "string": "# of Sales Order"
    },
    "purchase_order_count": {
        "type": "integer",
        "help": "",
        "string": "# of Purchase Order"
    },
```

2.2.5 Search and read

Because it is a very common task, Odoo provides a `search_read()` shortcut which as its name notes is equivalent to a *`search()`* followed by a *`read()`*, but avoids having to perform two requests and keep ids around.

Its arguments are similar to *`search()`*'s, but it can also take a list of `fields` (like *`read()`*, if that list is not provided it will fetch all fields of matched records):

```
[
    {
        "comment": false,
        "country_id": [ 21, "Belgium" ],
        "id": 7,
        "name": "Agrolait"
    },
    {
        "comment": false,
        "country_id": [ 76, "France" ],
        "id": 18,
        "name": "Axelor"
    },
    {
        "comment": false,
        "country_id": [ 233, "United Kingdom" ],
        "id": 12,
        "name": "Bank Wealthy and sons"
    },
    {
        "comment": false,
        "country_id": [ 105, "India" ],
        "id": 14,
        "name": "Best Designers"
    },
    {
        "comment": false,
        "country_id": [ 76, "France" ],
        "id": 17,
        "name": "Camptocamp"
    }
]
```


2.2.6 Create records

Records of a model are created using *create()*. The method will create a single record and return its database identifier.

create() takes a mapping of fields to values, used to initialize the record. For any field which has a default value and is not set through the mapping argument, the default value will be used.

```
78
```

Warning: while most value types are what would be expected (integer for *Integer*, string for *Char* or *Text*),

- *Date*, *Datetime* and `Binary` fields use string values
- *One2many* and *Many2many* use a special command protocol detailed in *the documentation to the write method*.

2.2.7 Update records

Records can be updated using *write()*, it takes a list of records to update and a mapping of updated fields to values similar to *create()*.

Multiple records can be updated simultanously, but they will all get the same values for the fields being set. It is not currently possible to perform "computed" updates (where the value being set depends on an existing value of a record).

```
[[78, "Newer partner"]]
```

2.2.8 Delete records

Records can be deleted in bulk by providing their ids to *unlink()*.

```
[]
```

2.2.9 Inspection and introspection

While we previously used *fields_get()* to query a model and have been using an arbitrary model from the start, Odoo stores most model metadata inside a few meta-models which allow both querying the system and altering models and fields (with some limitations) on the fly over XML-RPC.

ir.model

Provides information about Odoo models via its various fields

name a human-readable description of the model

model the name of each model in the system

state whether the model was generated in Python code (`base`) or by creating an `ir.model` record (`manual`)

field_id list of the model's fields through a *One2many* to *ir.model.fields*

view_ids *One2many* to the *Views* defined for the model

access_ids *One2many* relation to the *Access Control* set on the model

`ir.model` can be used to

- query the system for installed models (as a precondition to operations on the model or to explore the system's content)
- get information about a specific model (generally by listing the fields associated with it)
- create new models dynamically over RPC

Warning:
- "custom" model names must start with `x_`
- the `state` must be provided and `manual`, otherwise the model will not be loaded
- it is not possible to add new *methods* to a custom model, only fields

a custom model will initially contain only the "built-in" fields available on all models:

```
{
    "create_uid": {
        "type": "many2one",
        "string": "Created by"
    },
    "create_date": {
        "type": "datetime",
        "string": "Created on"
    },
    "__last_update": {
        "type": "datetime",
        "string": "Last Modified on"
    },
    "write_uid": {
        "type": "many2one",
```

```
        "string": "Last Updated by"
    },
    "write_date": {
        "type": "datetime",
        "string": "Last Updated on"
    },
    "display_name": {
        "type": "char",
        "string": "Display Name"
    },
    "id": {
        "type": "integer",
        "string": "Id"
    }
}
```

ir.model.fields

Provides information about the fields of Odoo models and allows adding custom fields without using Python code

model_id *Many2one* to *ir.model* to which the field belongs

name the field's technical name (used in `read` or `write`)

field_description the field's user-readable label (e.g. `string` in `fields_get`)

ttype the *type* of field to create

state whether the field was created via Python code (`base`) or via `ir.model.fields` (`manual`)

required, readonly, translate enables the corresponding flag on the field

groups *field-level access control*, a *Many2many* to `res.groups`

selection, size, on_delete, relation, relation_field, domain type-specific properties and customizations, see *the fields documentation* for details

Like custom models, only new fields created with `state="manual"` are activated as actual fields on the model.

> **Warning:** computed fields can not be added via `ir.model.fields`, some field meta-information (defaults, onchange) can not be set either

```
[
    {
        "create_uid": [1, "Administrator"],
        "x_name": "test record",
        "__last_update": "2014-11-12 16:32:13",
        "write_uid": [1, "Administrator"],
        "write_date": "2014-11-12 16:32:13",
        "create_date": "2014-11-12 16:32:13",
        "id": 1,
        "display_name": "test record"
    }
]
```

2.2.10 Workflow manipulations

Workflows can be moved along by sending them *signals*. Instead of using the top-level `execute_kw`, signals are sent using `exec_workflow`.

Signals are sent to a specific record, and possibly trigger a transition on the workflow instance associated with the record.

Warning: this example needs `account` module installed

2.2.11 Report printing

Available reports can be listed by searching the `ir.actions.report.xml` model, fields of interest being

`model` the model on which the report applies, can be used to look for available reports on a specific model

`name` human-readable report name

`report_name` the technical name of the report, used to print it

Reports can be printed over RPC with the following information:

- the name of the report (`report_name`)
- the ids of the records to include in the report

Note: the report is sent as PDF binary data encoded in base64, it must be decoded and may need to be saved to disk before use

---

CHAPTER

THREE

SETTING UP

3.1 Installing Odoo

There are mutliple ways to install Odoo, or not install it at all, depending on the intended use case.

This documents attempts to describe most of the installation options.

Demo the simplest "installation", only suitable for getting a quick feel for Odoo or trying something out

SaaS trivial to start with and fully managed and migrated by Odoo S.A., can be used to both test Odoo and use it for your business, but restricts the flexibility of the system somewhat (check: no custom modules? what else?).

Can be used for both testing Odoo and long-term "production" use.

Packaged installers simple to get started, allows more flexibility in hosting and deploying the system and greater control over where data is stored. The maintenance burden is shifted to the user.

Suitable for testing Odoo, developing modules and can be used for long-term production use with additional deployment and maintenance work.

Source Install harder to get started than *Packaged installers*, provides even greater flexibility: packaged installers don't generally allow multiple running Odoo versions on the same system, and don't provide easy source access to Odoo itself.

Good for developing modules, can be used as base for production deployment.

The source code can be obtained by downloading a tarball or using git. Using git makes it easier to update, switch between multiple versions (including the current development version) or contribute.

docker image if you usually use docker for development or deployment, an official docker base image is available, see the image's help document for more information

3.1.1 Demo

To simply get a quick idea of Odoo, demo instances are available. They are shared instances which only live for a few hours, and can be used to browse around and try things out with no commitment.

Demo instances require no local installation, just a web browser.

3.1.2 SaaS

Odoo's SaaS provides private instances and starts out free. It can be used to discover and test Odoo and do non-code customizations without having to install it locally.

Like demo instances, SaaS instances require no local installation, a web browser is sufficient.

3.1.3 Packaged installers

Odoo provides packaged installers for Windows, deb-based distributions (Debian, Ubuntu, ...) and RPM-based distributions (Fedora, CentOS, RHEL, ...).

These packages automatically set up all dependencies, but may be difficult to keep up-to-date.

Official packages with all relevant dependency requirements are available on https://nightly.odoo.com.

Windows

- download https://nightly.odoo.com/8.0/nightly/exe/odoo_8.0.latest.exe
- run the downloaded file

  Warning: on Windows 8, you may see a warning titled "Windows protected your PC". Click *More Info* then *Run anyway*

- Accept the UAC prompt
- Go through the various installation steps

Odoo will automatically be started at the end of the installation.

Configuration

The *configuration file* can be found at `%PROGRAMFILES%\Odoo 8.0-id\server\openerp-server.conf`.

The configuration file can be edited to connect to a remote Postgresql, edit file locations or set a dbfilter.

To reload the configuration file, restart the Odoo service via *Services* → *odoo server*.

Deb

To install Odoo 8.0 on Debian-based distribution, execute the following commands as root:

```
# wget -O - https://nightly.odoo.com/odoo.key | apt-key add -
# echo "deb http://nightly.odoo.com/8.0/nightly/deb/ ./" >> /etc/apt/sources.list
# apt-get update && apt-get install odoo
```

This will automatically install all dependencies, install Odoo itself as a daemon and automatically start it.

Danger: to print PDF reports, you must install wkhtmltopdf yourself: the version of wkhtmltopdf available in debian repositories does not support headers and footers so it can not be installed automatically. The recommended version is 0.12.1 and is available on the wkhtmltopdf download page, in the archive section. As there is no official release for Debian Jessie, you can find ours on http://nightly.odoo.com/extra/.

Configuration

The *configuration file* can be found at `/etc/odoo/openerp-server.conf`

When the configuration file is edited, Odoo must be restarted using `service`:

```
$ sudo service odoo restart
Restarting odoo: ok
```

RPM

Warning: with RHEL-based distributions (RHEL, CenOS, Scientific Linux), EPEL must be added to the distribution's repositories for all of Odoo's dependencies to be available. For CenOS:

```
$ sudo yum install -y epel-release
```

For other RHEL-based distribution, see the EPEL documentation.

```
$ sudo yum install -y postgresql-server
$ sudo postgresql-setup initdb
$ sudo systemctl enable postgresql
$ sudo systemctl start postgresql
$ sudo yum-config-manager --add-repo=https://nightly.odoo.com/8.0/nightly/rpm/odoo.repo
$ sudo yum install -y odoo
$ sudo systemctl enable odoo
$ sudo systemctl start odoo
```

Danger: to print PDF reports, you must install wkhtmltopdf yourself: the version of wkhtmltopdf available in Fedora/CentOS repositories does not support headers and footers so it can not be installed automatically. Use the version available on the wkhtmltopdf download page.

Configuration

The *configuration file* can be found at `/etc/odoo/openerp-server.conf`

When the configuration file is edited, Odoo must be restarted via SystemD:

```
$ sudo systemctl restart odoo
```

3.1.4 Source Install

The source "installation" really is about not installing Odoo, and running it directly from source instead.

This can be more convenient for module developers as the Odoo source is more easily accessible than using packaged installation (for information or to build this documentation and have it available offline).

It also makes starting and stopping Odoo more flexible and explicit than the services set up by the packaged installations, and allows overriding settings using *command-line parameters* without needing to edit a configuration file.

Finally it provides greater control over the system's set up, and allows more easily keeping (and running) multiple versions of Odoo side-by-side.

There are two way to get the odoo source source zip or git.

- Odoo zip can be downloaded from https://nightly.odoo.com/8.0/nightly/src/odoo_8.0.latest.zip, the zip file then needs to be uncompressed to use its content
- git allows simpler update and easier switching between differents versions of Odoo. It also simplifies maintaining non-module patches and contributions. The primary drawback of git is that it is significantly larger than a tarball as it contains the entire history of the Odoo project.

  The git repository is https://github.com/odoo/odoo.git.

  Downloading it requires a git client (which may be available via your distribution on linux) and can be performed using the following command:

```
$ git clone https://github.com/odoo/odoo.git
```

Installing dependencies

Source installation requires manually installing dependencies:

- Python 2.7.
  - on Linux and OS X, included by default
  - on Windows, use the official Python 2.7.9 installer.

    Warning: select "add python.exe to Path" during installation, and reboot afterwards to ensure the `PATH` is updated

    Note: if Python is already installed, make sure it is 2.7.9, previous versions are less convenient and 3.x versions are not compatible with Odoo

- PostgreSQL, to use a local database

  After installation you will need to create a postgres user: by default the only user is `postgres`, and Odoo forbids connecting as `postgres`.

  - on Linux, use your distribution's package, then create a postgres user named like your login:

```
$ sudo su - postgres -c "createuser -s $USER"
```

    Because the role login is the same as your unix login unix sockets can be use without a password.

  - on OS X, postgres.app is the simplest way to get started, then create a postgres user as on Linux
  - on Windows, use PostgreSQL for windows then
    - add PostgreSQL's `bin` directory (default: `C:\Program Files\PostgreSQL\9.4\bin`) to your `PATH`
    - create a postgres user with a password using the pg admin gui: open pgAdminIII, double-click the server to create a connection, select *Edit* → *New Object* → *New Login Role*, enter the username in the *Role Name* field (e.g. `odoo`), then open the *Definition* tab and enter the password (e.g. `odoo`), then click *OK*.

      The user and password must be passed to Odoo using either the `-w` and `-r` options or *the configuration file*

- Python dependencies listed in the `requirements.txt` file.
  - on Linux, python dependencies may be installable with the system's package manager or using pip.

    For libraries using native code (Pillow, lxml, greenlet, gevent, psycopg2, ldap) it may be necessary to install development tools and native dependencies before pip is able to install the dependencies themselves. These are available in `-dev` or `-devel` packages for Python, Postgres, libxml2, libxslt, libevent, libsasl2 and libldap2. Then the Python dependecies can themselves be installed:

```
$ pip install -r requirements.txt
```

  - on OS X, you will need to install the Command Line Tools (`xcode-select --install`) then download and install a package manager of your choice (homebrew, macports) to install non-Python dependencies. pip can then be used to install the Python dependencies as on Linux:

```
$ pip install -r requirements.txt
```

  - on Windows you need to install some of the dependencies manually, tweak the requirements.txt file, then run pip to install the remaning ones.

    Install `psycopg` using the installer here http://www.stickpeople.com/projects/python/win-psycopg/

    Then edit the requirements.txt file:

    - remove `psycopg2` as you already have it.

* remove the optional `python-ldap`, `gevent` and `psutil` because they require compilation.
* add `pypiwin32` because it's needed under windows.

Then use pip to install the dependencies using the following command from a cmd.exe prompt (replace `\YourOdooPath` by the actual path where you downloaded Odoo):

```
C:\> cd \YourOdooPath
C:\YourOdooPath> C:\Python27\Scripts\pip.exe install -r requirements.txt
```

- *Less CSS* via nodejs
  - on Linux, use your distribution's package manager to install nodejs and npm.

    Warning: In debian wheezy and Ubuntu 13.10 and before you need to install nodejs manually:

    ```
    $ wget -qO- https://deb.nodesource.com/setup | bash -
    $ apt-get install -y nodejs
    ```

    In later debian (>jessie) and ubuntu (>14.04) you may need to add a symlink as npm packages call `node` but debian calls the binary `nodejs`

    ```
    $ apt-get install -y npm
    $ sudo ln -s /usr/bin/nodejs /usr/bin/node
    ```

    Once npm is installed, use it to install less and less-plugin-clean-css:

    ```
    $ sudo npm install -g less less-plugin-clean-css
    ```

  - on OS X, install nodejs via your preferred package manager (homebrew, macports) then install less and less-plugin-clean-css:

    ```
    $ sudo npm install -g less less-plugin-clean-css
    ```

  - on Windows, install nodejs, reboot (to update the `PATH`) and install less and less-plugin-clean-css:

    ```
    C:\> npm install -g less less-plugin-clean-css
    ```

Running Odoo

Once all dependencies are set up, Odoo can be launched by running `odoo.py`.

Configuration can be provided either through *command-line arguments* or through a *configuration file*.

Common necessary configurations are:

- PostgreSQL host, port, user and password.

  Odoo has no defaults beyond psycopg2's defaults: connects over a UNIX socket on port 5432 with the current user and no password. By default this should work on Linux and OS X, but it *will not work* on windows as it does not support UNIX sockets.

- Custom addons path beyond the defaults, to load your own modules

Under Windows a typical way to execute odoo would be:

```
C:\YourOdooPath> python odoo.py -w odoo -r odoo --addons-path=addons,../mymodules --db-filter=mydb
```

Where `odoo`, `odoo` are the postgresql login and password, `../mymodules` a directory with additional addons and `mydb` the default db to serve on localhost:8069

Under Unix a typical way to execute odoo would be:

```
$ ./odoo.py --addons-path=addons,../mymodules --db-filter=mydb$
```

Where `../mymodules` is a directory with additional addons and `mydb` the default db to serve on localhost:8069

3.2 Deploying Odoo

This document describes basic steps to set up Odoo in production. It follows *installation*, but should not be used for development systems.

3.2.1 dbfilter

Odoo is a multi-tenant system: a single Odoo system may run and serve a number of database instances. It is also highly customizable, with customizations (starting from the modules being loaded) depending on the "current database".

This is not an issue when working with the backend (web client) as a logged-in company user: the database can be selected when logging in, and customizations loaded afterwards.

However it is an issue for non-logged users (portal, website) which aren't bound to a database: Odoo need to know which database should be used for the operations or to get the data. If multi-tenancy is not used that is not an issue, there's only one database to use, but if there are multiple databases accessible Odoo needs a rule to know which one it should use.

That is one of the purposes of *`--db-filter`*: it specifies the default database for the Odoo system. The value is a regular expression, possibly including the dynamically injected hostname or subdomain through which the Odoo system is accessed.

If an Odoo hosts multiple databases in production, especially if `website` is used, it **must** use a dbfilter or a number of features will not work correctly or not use at all.

3.2.2 PostgreSQL

By default, PostgreSQL only allows connection over UNIX sockets and loopback connections (from "localhost", the same machine the PostgreSQL server is installed on).

UNIX socket is fine if you want Odoo and PostgreSQL to execute on the same machine, and is the default when no host is provided, but if you want Odoo and PostgreSQL to execute on different machines [1] it will need to listen to network interfaces [2], either:

- only accept loopback connections and use an SSH tunnel between the machine on which Odoo runs and the one on which PostgreSQL runs, then configure Odoo to connect to its end of the tunnel
- accept connections to the machine on which Odoo is installed, possibly over ssl (see PostgreSQL connection settings for details), then configure Odoo to connect over the network

Configuring Odoo

Out of the box, Odoo connects to a local postgres over UNIX socket via port 5432. This can be overridden using *the database options* when your Postgres deployment is not local and/or does not use the installation defaults.

The *packaged installers* will automatically create a new user (`odoo`) and set it as the database user.

- The database management screens are protected by the `admin_passwd` setting. This setting can only be set using configuration files, and is simply checked before performing database alterations. It should be set to a randomly generated value to ensure third parties can not use this interface.

[1] to have multiple Odoo installations use the same PostgreSQL database, or to provide more computing resources to both software.
[2] technically a tool like socat can be used to proxy UNIX sockets across networks, but that is mostly for software which can only be used over UNIX sockets

- all database operations use the *database options*, including the database management screen. For the database management screen to work requires that the user have `createdb` right.
- users can always drop databases they own. For the database management screen to be completely non-functional, the user needs to be created with `no-createdb` and the database must be owned by a different user.

> **Warning:** the user *must not* be a superuser

3.2.3 HTTPS

Whether it's accessed via website/web client or the webservice, Odoo transmits authentication information in cleatext. This means a secure deployment of Odoo must use HTTPS[3]. SSL termination can be implemented via just about any SSL termination proxy, but requires the following setup:

- enable Odoo's *proxy mode*. This should only be enabled when Odoo is behind a reverse proxy
- set up the SSL termination proxy (Nginx termination example)
- set up the proxying itself (Nginx proxying example)
- your SSL termination proxy should also automatically redirect non-secure connections to the secure port

3.2.4 Builtin server

Odoo includes built-in HTTP servers, using either multithreading or multiprocessing.

For production use, it is recommended to use the multiprocessing server as it increases stability, makes somewhat better use of computing resources and can be better monitored and resource-restricted.

- Multiprocessing is enabled by configuring *a non-zero number of worker processes*, the number of workers should be based on the number of cores in the machine (possibly with some room for cron workers depending on how much cron work is predicted)
- Worker limits can be configured based on the hardware configuration to avoid resources exhaustion

> **Warning:** multiprocessing mode currently isn't available on Windows

LiveChat

In multiprocessing, a dedicated LiveChat worker is automatically started and listening on *the longpolling port* but the client will not connect to it.

Instead you must have a proxy redirecting requests whose URL starts with `/longpolling/` to the longpolling port. Other request should be proxied to the *normal HTTP port*

3.2.5 Odoo as a WSGI Application

It is also possible to mount Odoo as a standard WSGI application. Odoo provides the base for a WSGI launcher script as `openerp-wsgi.py`. That script should be customized (possibly after copying it) to correctly set the configuration directly in `openerp.tools.config` rather than through the command-line or a configuration file.

However the WSGI server will only expose the main HTTP endpoint for the web client, website and webservice API. Because Odoo does not control the creation of workers anymore it can not setup cron or livechat workers

[3] or be accessible only over an internal packet-switched network, but that requires secured switches, protections against ARP spoofing and precludes usage of WiFi. Even over secure packet-switched networks, deployment over HTTPS is recommended, and possible costs are lowered as "self-signed" certificates are easier to deploy on a controlled environment than over the internet.

Cron Workers

To run cron jobs for an Odoo deployment as a WSGI application requires

- a classical Odoo (run via `odoo.py`)
- connected to the database in which cron jobs have to be run (via *`odoo.py -d`*)
- which should not be exposed to the network. To ensure cron runners are not network-accessible, it is possible to disable the built-in HTTP server entirely with *`odoo.py --no-xmlrpc`* or setting `xmlrpc = False` in the configuration file

LiveChat

The second problematic subsystem for WSGI deployments is the LiveChat: where most HTTP connections are relatively short and quickly free up their worker process for the next request, LiveChat require a long-lived connection for each client in order to implement near-real-time notifications.

This is in conflict with the process-based worker model, as it will tie up worker processes and prevent new users from accessing the system. However, those long-lived connections do very little and mostly stay parked waiting for notifications.

The solutions to support livechat/motifications in a WSGI application are:

- deploy a threaded version of Odoo (instread of a process-based preforking one) and redirect only requests to URLs starting with `/longpolling/` to that Odoo, this is the simplest and the longpolling URL can double up as the cron instance.
- deploy an evented Odoo via `openerp-gevent` and proxy requests starting with `/longpolling/` to *`the longpolling port`*.

3.2.6 Serving Static Files

For development convenience, Odoo directly serves all static files in its modules. This may not be ideal when it comes to performances, and static files should generally be served by a static HTTP server.

Odoo static files live in each module's `static/` folder, so static files can be served by intercepting all requests to */MODULE*`/static/`*FILE*, and looking up the right module (and file) in the various addons paths.

3.2.7 Security

"Super-admin" password

Configuring Odoo mentioned `admin_passwd` in passing.

This setting is used on all database management screens (to create, delete, dump or restore databases).

If the management screens must not be accessible, or must only be accessible from a selected set of machines, use the proxy server's features to block access to all routes starting with `/web/database` except (maybe) `/web/database/selector` which displays the database-selection screen.

If the database-management screen should be left accessible, the `admin_passwd` setting must be changed from its `admin` default: this password is checked before allowing database-alteration operations.

It should be stored securely, and should be generated randomly e.g.

```
$ python -c 'import base64, os; print(base64.b64encode(os.urandom(24)))'
```

which will generate a 32 characters pseudorandom printable string.

CHAPTER

FOUR

REFERENCE

4.1 ORM API

4.1.1 Recordsets

New in version 8.0: This page documents the New API added in Odoo 8.0 which should be the primary development API going forward. It also provides information about porting from or bridging with the "old API" of versions 7 and earlier, but does not explicitly document that API. See the old documentation for that.

Interaction with models and records is performed through recordsets, a sorted set of records of the same model.

> **Warning:** contrary to what the name implies, it is currently possible for recordsets to contain duplicates. This may change in the future.

Methods defined on a model are executed on a recordset, and their `self` is a recordset:

```
class AModel(models.Model):
    _name = 'a.model'
    def a_method(self):
        # self can be anywhere between 0 records and all records in the
        # database
        self.do_operation()
```

Iterating on a recordset will yield new sets of *a single record* ("singletons"), much like iterating on a Python string yields strings of a single characters:

```
def do_operation(self):
    print self # => a.model(1, 2, 3, 4, 5)
    for record in self:
        print record # => a.model(1), then a.model(2), then a.model(3), ...
```

Field access

Recordsets provide an "Active Record" interface: model fields can be read and written directly from the record, but only on singletons (single-record recordsets). Setting a field's value triggers an update to the database:

```
>>> record.name
Example Name
>>> record.company_id.name
Company Name
>>> record.name = "Bob"
```

Trying to read or write a field on multiple records will raise an error.

Accessing a relational field (*Many2one*, *One2many*, *Many2many*) *always* returns a recordset, empty if the field is not set.

Danger: each assignment to a field triggers a database update, when setting multiple fields at the same time or setting fields on multiple records (to the same value), use *write()*:

```
# 3 * len(records) database updates
for record in records:
    record.a = 1
    record.b = 2
    record.c = 3

# len(records) database updates
for record in records:
    record.write({'a': 1, 'b': 2, 'c': 3})

# 1 database update
records.write({'a': 1, 'b': 2, 'c': 3})
```

Set operations

Recordsets are immutable, but sets of the same model can be combined using various set operations, returning new recordsets. Set operations do *not* preserve order.

- `record in set` returns whether `record` (which must be a 1-element recordset) is present in `set`. `record not in set` is the inverse operation
- `set1 | set2` returns the union of the two recordsets, a new recordset containing all records present in either source
- `set1 & set2` returns the intersection of two recordsets, a new recordset containing only records present in both sources
- `set1 - set2` returns a new recordset containing only records of `set1` which are *not* in `set2`

Other recordset operations

Recordsets are iterable so the usual Python tools are available for transformation (`python:map()`, `python:sorted()`, `ifilter()`, ...) however these return either a `python:list` or an python:iterator, removing the ability to call methods on their result, or to use set operations.

Recordsets therefore provide these operations returning recordsets themselves (when possible):

filtered() returns a recordset containing only records satisfying the provided predicate function. The predicate can also be a string to filter by a field being true or false:

```
# only keep records whose company is the current user's
records.filtered(lambda r: r.company_id == user.company_id)

# only keep records whose partner is a company
records.filtered("partner_id.is_company")
```

sorted() returns a recordset sorted by the provided key function. If no key is provided, use the model's default sort order:

```
# sort records by name
records.sorted(key=lambda r: r.name)
```

mapped() applies the provided function to each record in the recordset, returns a recordset if the results are recordsets:

```
# returns a list of summing two fields for each record in the set
records.mapped(lambda r: r.field1 + r.field2)
```

The provided function can be a string to get field values:

```
# returns a list of names
records.mapped('name')

# returns a recordset of partners
record.mapped('partner_id')

# returns the union of all partner banks, with duplicates removed
record.mapped('partner_id.bank_ids')
```

4.1.2 Environment

The `Environment` stores various contextual data used by the ORM: the database cursor (for database queries), the current user (for access rights checking) and the current context (storing arbitrary metadata). The environment also stores caches.

All recordsets have an environment, which is immutable, can be accessed using `env` and gives access to the current user (`user`), the cursor (`cr`) or the context (`context`):

```
>>> records.env
<Environment object ...>
>>> records.env.user
res.user(3)
>>> records.env.cr
<Cursor object ...)
```

When creating a recordset from an other recordset, the environment is inherited. The environment can be used to get an empty recordset in an other model, and query that model:

```
>>> self.env['res.partner']
res.partner
>>> self.env['res.partner'].search([['is_company', '=', True], ['customer', '=', True]])
res.partner(7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23, 28, 74)
```

Altering the environment

The environment can be customized from a recordset. This returns a new version of the recordset using the altered environment.

sudo() creates a new environment with the provided user set, uses the administrator if none is provided (to bypass access rights/rules in safe contexts), returns a copy of the recordset it is called on using the new environment:

```
# create partner object as administrator
env['res.partner'].sudo().create({'name': "A Partner"})

# list partners visible by the "public" user
public = env.ref('base.public_user')
env['res.partner'].sudo(public).search([])
```

with_context()

1. can take a single positional parameter, which replaces the current environment's context
2. can take any number of parameters by keyword, which are added to either the current environment's context or the context set during step 1

```
# look for partner, or create one with specified timezone if none is
# found
env['res.partner'].with_context(tz=a_tz).find_or_create(email_address)
```

with_env() replaces the existing environment entirely

4.1.3 Common ORM methods

search() Takes a *search domain*, returns a recordset of matching records. Can return a subset of matching records (`offset` and `limit` parameters) and be ordered (`order` parameter):

```
>>> # searches the current model
>>> self.search([('is_company', '=', True), ('customer', '=', True)])
res.partner(7, 18, 12, 14, 17, 19, 8, 31, 26, 16, 13, 20, 30, 22, 29, 15, 23, 28, 74)
>>> self.search([('is_company', '=', True)], limit=1).name
'Agrolait'
```

> **Tip:** to just check if any record matches a domain, or count the number of records which do, use *search_count()*

create() Takes a number of field values, and returns a recordset containing the record created:

```
>>> self.create({'name': "New Name"})
res.partner(78)
```

write() Takes a number of field values, writes them to all the records in its recordset. Does not return anything:

```
self.write({'name': "Newer Name"})
```

browse() Takes a database id or a list of ids and returns a recordset, useful when record ids are obtained from outside Odoo (e.g. round-trip through external system) or *when calling methods in the old API*:

```
>>> self.browse([7, 18, 12])
res.partner(7, 18, 12)
```

exists() Returns a new recordset containing only the records which exist in the database. Can be used to check whether a record (e.g. obtained externally) still exists:

```
if not record.exists():
    raise Exception("The record has been deleted")
```

or after calling a method which could have removed some records:

```
records.may_remove_some()
# only keep records which were not deleted
records = records.exists()
```

ref() Environment method returning the record matching a provided external id:

```
>>> env.ref('base.group_public')
res.groups(2)
```

ensure_one() checks that the recordset is a singleton (only contains a single record), raises an error otherwise:

```
records.ensure_one()
# is equivalent to but clearer than:
assert len(records) == 1, "Expected singleton"
```

4.1.4 Creating Models

Model fields are defined as attributes on the model itself:

```
from openerp import models, fields
class AModel(models.Model):
    _name = 'a.model.name'

    field1 = fields.Char()
```

Warning: this means you can not define a field and a method with the same name, they will conflict

By default, the field's label (user-visible name) is a capitalized version of the field name, this can be overridden with the `string` parameter:

```
field2 = fields.Integer(string="an other field")
```

For the various field types and parameters, see *the fields reference*.

Default values are defined as parameters on fields, either a value:

```
a_field = fields.Char(default="a value")
```

or a function called to compute the default value, which should return that value:

```
a_field = fields.Char(default=compute_default_value)
def compute_default_value(self):
    return self.get_value()
```

Computed fields

Fields can be computed (instead of read straight from the database) using the `compute` parameter. **It must assign the computed value to the field**. If it uses the values of other *fields*, it should specify those fields using *`depends()`*:

```
from openerp import api
total = fields.Float(compute='_compute_total')

@api.depends('value', 'tax')
def _compute_total(self):
    for record in self:
        record.total = record.value + record.value * record.tax
```

- dependencies can be dotted paths when using sub-fields:

```
    @api.depends('line_ids.value')
    def _compute_total(self):
        for record in self:
            record.total = sum(line.value for line in record.line_ids)
```

- computed fields are not stored by default, they are computed and returned when requested. Setting `store=True` will store them in the database and automatically enable searching
- searching on a computed field can also be enabled by setting the `search` parameter. The value is a method name returning a *Domains*:

```
    upper_name = field.Char(compute='_compute_upper', search='_search_upper')

    def _search_upper(self, operator, value):
        if operator == 'like':
            operator = 'ilike'
        return [('name', operator, value)]
```

- to allow *setting* values on a computed field, use the `inverse` parameter. It is the name of a function reversing the computation and setting the relevant fields:

```
    document = fields.Char(compute='_get_document', inverse='_set_document')

    def _get_document(self):
        for record in self:
            with open(record.get_document_path) as f:
                record.document = f.read()
    def _set_document(self):
```

```
        for record in self:
            if not record.document: continue
            with open(record.get_document_path()) as f:
                f.write(record.document)
```

- multiple fields can be computed at the same time by the same method, just use the same method on all fields and set all of them:

```
    discount_value = fields.Float(compute='_apply_discount')
    total = fields.Float(compute='_apply_discount')

    @depends('value', 'discount')
    def _apply_discount(self):
        for record in self:
            # compute actual discount from discount percentage
            discount = self.value * self.discount
            self.discount_value = discount
            self.total = self.value - discount
```

Related fields

A special case of computed fields are *related* (proxy) fields, which provide the value of a sub-field on the current record. They are defined by setting the `related` parameter and like regular computed fields they can be stored:

```
nickname = fields.Char(related='user_id.partner_id.name', store=True)
```

onchange: updating UI on the fly

When a user changes a field's value in a form (but hasn't saved the form yet), it can be useful to automatically update other fields based on that value e.g. updating a final total when the tax is changed or a new invoice line is added.

- computed fields are automatically checked and recomputed, they do not need an `onchange`
- for non-computed fields, the *`onchange()`* decorator is used to provide new field values:

```
    @api.onchange('field1', 'field2') # if these fields are changed, call method
    def check_change(self):
        if self.field1 < self.field2:
            self.field3 = True
```

  the changes performed during the method are then sent to the client program and become visible to the user

- Both computed fields and new-API onchanges are automatically called by the client without having to add them in views
- It is possible to suppress the trigger from a specific field by adding `on_change="0"` in a view:

```
    <field name="name" on_change="0"/>
```

  will not trigger any interface update when the field is edited by the user, even if there are function fields or explicit onchange depending on that field.

Note: `onchange` methods work on virtual records assignment on these records is not written to the database, just used to know which value to send back to the client

---

Low-level SQL

The `cr` attribute on environments is the cursor for the current database transaction and allows executing SQL directly, either for queries which are difficult to express using the ORM (e.g. complex joins) or for performance

reasons:

```
self.env.cr.execute("some_sql", param1, param2, param3)
```

Because models use the same cursor and the `Environment` holds various caches, these caches must be invalidated when *altering* the database in raw SQL, or further uses of models may become incoherent. It is necessary to clear caches when using `CREATE`, `UPDATE` or `DELETE` in SQL, but not `SELECT` (which simply reads the database).

Clearing caches can be performed using the `invalidate_all()` method of the `Environment` object.

4.1.5 Compatibility between new API and old API

Odoo is currently transitioning from an older (less regular) API, it can be necessary to manually bridge from one to the other manually:

- RPC layers (both XML-RPC and JSON-RPC) are expressed in terms of the old API, methods expressed purely in the new API are not available over RPC
- overridable methods may be called from older pieces of code still written in the old API style

The big differences between the old and new APIs are:

- values of the `Environment` (cursor, user id and context) are passed explicitly to methods instead
- record data (*`ids`*) are passed explicitly to methods, and possibly not passed at all
- methods tend to work on lists of ids instead of recordsets

By default, methods are assumed to use the new API style and are not callable from the old API style.

Tip: calls from the new API to the old API are bridged

when using the new API style, calls to methods defined using the old API are automatically converted on-the-fly, there should be no need to do anything special:

```
>>> # method in the old API style
>>> def old_method(self, cr, uid, ids, context=None):
...     print ids

>>> # method in the new API style
>>> def new_method(self):
...     # system automatically infers how to call the old-style
...     # method from the new-style method
...     self.old_method()

>>> env[model].browse([1, 2, 3, 4]).new_method()
[1, 2, 3, 4]
```

Two decorators can expose a new-style method to the old API:

model() the method is exposed as not using ids, its recordset will generally be empty. Its "old API" signature is `cr, uid, *arguments, context`:

```
@api.model
def some_method(self, a_value):
    pass
# can be called as
old_style_model.some_method(cr, uid, a_value, context=context)
```

multi() the method is exposed as taking a list of ids (possibly empty), its "old API" signature is `cr, uid, ids, *arguments, context`:

```
    @api.multi
    def some_method(self, a_value):
        pass
    # can be called as
    old_style_model.some_method(cr, uid, [id1, id2], a_value, context=context)
```

Because new-style APIs tend to return recordsets and old-style APIs tend to return lists of ids, there is also a decorator managing this:

returns() the function is assumed to return a recordset, the first parameter should be the name of the recordset's model or `self` (for the current model).

No effect if the method is called in new API style, but transforms the recordset into a list of ids when called from the old API style:

```
    >>> @api.multi
    ... @api.returns('self')
    ... def some_method(self):
    ...     return self
    >>> new_style_model = env['a.model'].browse(1, 2, 3)
    >>> new_style_model.some_method()
    a.model(1, 2, 3)
    >>> old_style_model = pool['a.model']
    >>> old_style_model.some_method(cr, uid, [1, 2, 3], context=context)
    [1, 2, 3]
```

4.1.6 Model Reference

class `openerp.models.`**`Model`**(*pool*, *cr*)

Main super-class for regular database-persisted OpenERP models.

OpenERP models are created by inheriting from this class:

```
    class user(Model):
        ...
```

The system will later instantiate the class once per database (on which the class' module is installed).

Structural attributes

`_name`

business object name, in dot-notation (in module namespace)

`_rec_name`

Alternative field to use as name, used by osv's name_get() (default: `'name'`)

`_inherit`

- If *`_name`* is set, names of parent models to inherit from. Can be a `str` if inheriting from a single parent
- If *`_name`* is unset, name of a single model to extend in-place

See *Inheritance and extension*.

`_order`

Ordering field when searching without an ordering specified (default: `'id'`)

Type str

`_auto`

Whether a database table should be created (default: `True`)

If set to `False`, override `init()` to create the database table

_table
Name of the table backing the model created when *_auto*, automatically generated by default.

_inherits
dictionary mapping the _name of the parent business objects to the names of the corresponding foreign key fields to use:

```
_inherits = {
    'a.model': 'a_field_id',
    'b.model': 'b_field_id'
}
```

implements composition-based inheritance: the new model exposes all the fields of the *_inherits*-ed model but stores none of them: the values themselves remain stored on the linked record.

> **Warning:** if the same field is defined on multiple *_inherits*-ed

_constraints
list of `(constraint_function, message, fields)` defining Python constraints. The fields list is indicative

Deprecated since version 8.0: use *constrains()*

_sql_constraints
list of `(name, sql_definition, message)` triples defining SQL constraints to execute when generating the backing table

_parent_store
Alongside *parent_left* and *parent_right*, sets up a nested set to enable fast hierarchical queries on the records of the current model (default: `False`)

Type bool

CRUD

create(*vals*) → record
Creates a new record for the model.

The new record is initialized using the values from `vals` and if necessary those from *default_get()*.

Parameters **vals** (*dict*) – values for the model's fields, as a dictionary:

```
{'field_name': field_value, ...}
```

see *write()* for details

Returns new record created

Raises

- **AccessError** –
  - if user has no create rights on the requested object
  - if user tries to bypass access rules for create on the requested object
- **ValidateError** – if user tries to enter invalid value for a field that is not in selection
- **UserError** – if a loop would be created in a hierarchy of objects a result of the operation (such as setting an object as its own parent)

browse([*ids*]) → records
Returns a recordset for the ids provided as parameter in the current environment.

Can take no ids, a single id or a sequence of ids.

unlink()
Deletes the records of the current set

Raises

- **AccessError** –
  - if user has no unlink rights on the requested object
  - if user tries to bypass access rules for unlink on the requested object
- **UserError** – if the record is default property for other records

write(*vals*)
Updates all records in the current set with the provided values.

Parameters **vals** (*dict*) – fields to update and the value to set on them e.g:

```
{'foo': 1, 'bar': "Qux"}
```

will set the field `foo` to `1` and the field `bar` to `"Qux"` if those are valid (otherwise it will trigger an error).

Raises

- **AccessError** –
  - if user has no write rights on the requested object
  - if user tries to bypass access rules for write on the requested object
- **ValidateError** – if user tries to enter invalid value for a field that is not in selection
- **UserError** – if a loop would be created in a hierarchy of objects a result of the operation (such as setting an object as its own parent)

•For numeric fields (*Integer*, *Float*) the value should be of the corresponding type

•For *Boolean*, the value should be a `python:bool`

•For *Selection*, the value should match the selection values (generally `python:str`, sometimes `python:int`)

•For *Many2one*, the value should be the database identifier of the record to set

•Other non-relational fields use a string for value

> **Danger:** for historical and compatibility reasons, *Date* and *Datetime* fields use strings as values (written and read) rather than `date` or `datetime`. These date strings are UTC-only and formatted according to `openerp.tools.misc.DEFAULT_SERVER_DATE_FORMAT` and `openerp.tools.misc.DEFAULT_SERVER_DATETIME_FORMAT`

•*One2many* and *Many2many* use a special "commands" format to manipulate the set of records stored in/associated with the field.

This format is a list of triplets executed sequentially, where each triplet is a command to execute on the set of records. Not all commands apply in all situations. Possible commands are:

(0, _, values) adds a new record created from the provided `value` dict.

(1, id, values) updates an existing record of id `id` with the values in `values`. Can not be used in *create()*.

(2, id, _) removes the record of id `id` from the set, then deletes it (from the database). Can not be used in *create()*.

(3, id, _) removes the record of id `id` from the set, but does not delete it. Can not be used on *One2many*. Can not be used in *create()*.

(4, id, _) adds an existing record of id `id` to the set. Can not be used on *One2many*.

(5, _, _) removes all records from the set, equivalent to using the command `3` on every record explicitly. Can not be used on *One2many*. Can not be used in *create()*.

(6, _, ids) replaces all existing records in the set by the `ids` list, equivalent to using the command `5` followed by a command `4` for each `id` in `ids`. Can not be used on *One2many*.

Note: Values marked as _ in the list above are ignored and can be anything, generally `0` or `False`.

---

read([*fields*])

Reads the requested fields for the records in `self`, low-level/RPC method. In Python code, prefer *browse()*.

Parameters **fields** – list of field names to return (default is all fields)

Returns a list of dictionaries mapping field names to their values, with one dictionary per record

Raises AccessError if user has no read rights on some of the given records

Research

search(*args[, offset=0][, limit=None][, order=None][, count=False]*)

Searches for records based on the `args` *search domain*.

Parameters

- **args** – *A search domain*. Use an empty list to match all records.
- **offset** (*int*) – number of results to ignore (default: none)
- **limit** (*int*) – maximum number of records to return (default: all)
- **order** (*str*) – sort string
- **count** (*bool*) – if True, only counts and returns the number of matching records (default: False)

Returns at most `limit` records matching the search criteria

Raises AccessError

- if user tries to bypass access rules for read on the requested object.

search_count(*args*) → int

Returns the number of records in the current model matching *the provided domain*.

name_search(*name=''*, *args=None*, *operator='ilike'*, *limit=100*) → records

Search for records that have a display name matching the given `name` pattern when compared with the given `operator`, while also matching the optional search domain (`args`).

This is used for example to provide suggestions based on a partial value for a relational field. Sometimes be seen as the inverse function of *name_get()*, but it is not guaranteed to be.

This method is equivalent to calling *search()* with a search domain based on `display_name` and then *name_get()* on the result of the search.

Parameters

- **name** (*str*) – the name pattern to match
- **args** (*list*) – optional search domain (see *search()* for syntax), specifying further restrictions
- **operator** (*str*) – domain operator for matching `name`, such as `'like'` or `'='`.

- **limit** (*int*) – optional max number of records to return

Return type list

Returns list of pairs `(id, text_repr)` for all matching records.

Recordset operations

ids
List of actual record ids in this recordset (ignores placeholder ids for records to create)

ensure_one()
Verifies that the current recorset holds a single record. Raises an exception otherwise.

exists() → records
Returns the subset of records in `self` that exist, and marks deleted records as such in cache. It can be used as a test on records:

```
if record.exists():
    ...
```

By convention, new records are returned as existing.

filtered(*func*)
Select the records in `self` such that `func(rec)` is true, and return them as a recordset.

Parameters **func** – a function or a dot-separated sequence of field names

sorted(*key=None*, *reverse=False*)
Return the recordset `self` ordered by `key`.

Parameters

- **key** – either a function of one argument that returns a comparison key for each record, or `None`, in which case records are ordered according the default model's order
- **reverse** – if `True`, return the result in reverse order

mapped(*func*)
Apply `func` on all records in `self`, and return the result as a list or a recordset (if `func` return recordsets). In the latter case, the order of the returned recordset is arbitrary.

Parameters **func** – a function or a dot-separated sequence of field names

Environment swapping

sudo([*user=SUPERUSER*])
Returns a new version of this recordset attached to the provided user.

By default this returns a *SUPERUSER* recordset, where access control and record rules are bypassed.

Note: Using *sudo* could cause data access to cross the boundaries of record rules, possibly mixing records that are meant to be isolated (e.g. records from different companies in multi-company environments).

It may lead to un-intuitive results in methods which select one record among many - for example getting the default company, or selecting a Bill of Materials.

Note: Because the record rules and access control will have to be re-evaluated, the new recordset will not benefit from the current environment's data cache, so later data access may incur extra delays while re-fetching from the database.

---

with_context(*[context][, **overrides]*) → records
Returns a new version of this recordset attached to an extended context.

The extended context is either the provided `context` in which `overrides` are merged or the *current* context in which `overrides` are merged e.g.:

```
# current context is {'key1': True}
r2 = records.with_context({}, key2=True)
# -> r2._context is {'key2': True}
r2 = records.with_context(key2=True)
# -> r2._context is {'key1': True, 'key2': True}
```

with_env(*env*)
Returns a new version of this recordset attached to the provided environment

> **Warning:** The new environment will not benefit from the current environment's data cache, so later data access may incur extra delays while re-fetching from the database.

Fields and views querying

fields_get(*[fields][, attributes]*)
Return the definition of each field.

The returned value is a dictionary (indiced by field name) of dictionaries. The _inherits'd fields are included. The string, help, and selection (if present) attributes are translated.

Parameters

- **allfields** – list of fields to document, all if empty or not provided
- **attributes** – list of description attributes to return for each field, all if empty or not provided

fields_view_get([*view_id* | *view_type='form'*])
Get the detailed composition of the requested view like fields, model, view architecture

Parameters

- **view_id** – id of the view or None
- **view_type** – type of the view to return if view_id is None ('form', 'tree', ...)
- **toolbar** – true to include contextual actions
- **submenu** – deprecated

Returns dictionary describing the composition of the requested view (including inherited views and extensions)

Raises

- **AttributeError** –
  - if the inherited view has unknown position to work with other than 'before', 'after', 'inside', 'replace'
  - if some tag other than 'position' is found in parent view
- **Invalid ArchitectureError** – if there is view type other than form, tree, calendar, search etc defined on the structure

???

default_get(*fields*) → default_values
Return default values for the fields in `fields_list`. Default values are determined by the context, user defaults, and the model itself.

Parameters **fields_list** – a list of field names

Returns a dictionary mapping each field name to its corresponding default value, if it has one.

copy(*default=None*)
Duplicate record with given id updating it with default values

Parameters **default** (*dict*) – dictionary of field values to override in the original values of the copied record, e.g: `{'field_name': overridden_value, ...}`

Returns new record

name_get() → [(id, name), ...]
Returns a textual representation for the records in `self`. By default this is the value of the `display_name` field.

Returns list of pairs `(id, text_repr)` for each records

Return type list(tuple)

name_create(*name*) → record
Create a new record by calling *create()* with only one value provided: the display name of the new record.

The new record will be initialized with any default values applicable to this model, or provided through the context. The usual behavior of *create()* applies.

Parameters **name** – display name of the record to create

Return type tuple

Returns the *name_get()* pair value of the created record

Automatic fields

id
Identifier *field*

_log_access
Whether log access fields (`create_date`, `write_uid`, ...) should be generated (default: `True`)

create_date
Date at which the record was created

Type `Datetime`

create_uid
Relational field to the user who created the record

Type `res.users`

write_date
Date at which the record was last modified

Type `Datetime`

write_uid
Relational field to the last user who modified the record

Type `res.users`

Reserved field names

A few field names are reserved for pre-defined behaviors beyond that of automated fields. They should be defined on a model when the related behavior is desired:

name
default value for *_rec_name*, used to display records in context where a representative "naming" is necessary.

Type *Char*

active
toggles the global visibility of the record, if `active` is set to `False` the record is invisible in most searches and listing

Type *Boolean*

sequence
Alterable ordering criteria, allows drag-and-drop reordering of models in list views

Type *Integer*

state
lifecycle stages of the object, used by the `states` attribute on *fields*

Type *Selection*

parent_id
used to order records in a tree structure and enables the `child_of` operator in domains

Type *Many2one*

parent_left
used with *_parent_store*, allows faster tree structure access

parent_right
see *parent_left*

4.1.7 Method decorators

This module provides the elements for managing two different API styles, namely the "traditional" and "record" styles.

In the "traditional" style, parameters like the database cursor, user id, context dictionary and record ids (usually denoted as `cr`, `uid`, `context`, `ids`) are passed explicitly to all methods. In the "record" style, those parameters are hidden into model instances, which gives it a more object-oriented feel.

For instance, the statements:

```
model = self.pool.get(MODEL)
ids = model.search(cr, uid, DOMAIN, context=context)
for rec in model.browse(cr, uid, ids, context=context):
    print rec.name
model.write(cr, uid, ids, VALUES, context=context)
```

may also be written as:

```
env = Environment(cr, uid, context)  # cr, uid, context wrapped in env
model = env[MODEL]                   # retrieve an instance of MODEL
recs = model.search(DOMAIN)          # search returns a recordset
for rec in recs:                     # iterate over the records
    print rec.name
recs.write(VALUES)                   # update all records in recs
```

Methods written in the "traditional" style are automatically decorated, following some heuristics based on parameter names.

openerp.api.**multi**(*method*)

Decorate a record-style method where `self` is a recordset. The method typically defines an operation on records. Such a method:

```
@api.multi
def method(self, args):
    ...
```

may be called in both record and traditional styles, like:

```
# recs = model.browse(cr, uid, ids, context)
recs.method(args)

model.method(cr, uid, ids, args, context=context)
```

openerp.api.**model**(*method*)

Decorate a record-style method where `self` is a recordset, but its contents is not relevant, only the model is. Such a method:

```
@api.model
def method(self, args):
    ...
```

may be called in both record and traditional styles, like:

```
# recs = model.browse(cr, uid, ids, context)
recs.method(args)

model.method(cr, uid, args, context=context)
```

Notice that no `ids` are passed to the method in the traditional style.

openerp.api.**depends**(**args*)

Return a decorator that specifies the field dependencies of a "compute" method (for new-style function fields). Each argument must be a string that consists in a dot-separated sequence of field names:

```
pname = fields.Char(compute='_compute_pname')

@api.one
@api.depends('partner_id.name', 'partner_id.is_company')
def _compute_pname(self):
    if self.partner_id.is_company:
        self.pname = (self.partner_id.name or "").upper()
    else:
        self.pname = self.partner_id.name
```

One may also pass a single function as argument. In that case, the dependencies are given by calling the function with the field's model.

openerp.api.**constrains**(**args*)

Decorates a constraint checker. Each argument must be a field name used in the check:

```
@api.one
@api.constrains('name', 'description')
def _check_description(self):
    if self.name == self.description:
        raise ValidationError("Fields name and description must be different")
```

Invoked on the records on which one of the named fields has been modified.

Should raise `ValidationError` if the validation failed.

openerp.api.**onchange**(**args*)

Return a decorator to decorate an onchange method for given fields. Each argument must be a field name:

```
    @api.onchange('partner_id')
    def _onchange_partner(self):
        self.message = "Dear %s" % (self.partner_id.name or "")
```

In the form views where the field appears, the method will be called when one of the given fields is modified. The method is invoked on a pseudo-record that contains the values present in the form. Field assignments on that record are automatically sent back to the client.

The method may return a dictionary for changing field domains and pop up a warning message, like in the old API:

```
    return {
        'domain': {'other_id': [('partner_id', '=', partner_id)]},
        'warning': {'title': "Warning", 'message': "What is this?"},
    }
```

openerp.api.**returns**(*model*, *downgrade=None*, *upgrade=None*)

Return a decorator for methods that return instances of `model`.

Parameters

- **model** – a model name, or `'self'` for the current model
- **downgrade** – a function `downgrade(self, value, *args, **kwargs)` to convert the record-style `value` to a traditional-style output
- **upgrade** – a function `upgrade(self, value, *args, **kwargs)` to convert the traditional-style `value` to a record-style output

The arguments `self`, `*args` and `**kwargs` are the ones passed to the method in the record-style.

The decorator adapts the method output to the api style: `id`, `ids` or `False` for the traditional style, and recordset for the record style:

```
    @model
    @returns('res.partner')
    def find_partner(self, arg):
        ...     # return some record

    # output depends on call style: traditional vs record style
    partner_id = model.find_partner(cr, uid, arg, context=context)

    # recs = model.browse(cr, uid, ids, context)
    partner_record = recs.find_partner(arg)
```

Note that the decorated method must satisfy that convention.

Those decorators are automatically *inherited*: a method that overrides a decorated existing method will be decorated with the same `@returns(model)`.

openerp.api.**one**(*method*)

Decorate a record-style method where `self` is expected to be a singleton instance. The decorated method automatically loops on records, and makes a list with the results. In case the method is decorated with *returns()*, it concatenates the resulting instances. Such a method:

```
    @api.one
    def method(self, args):
        return self.name
```

may be called in both record and traditional styles, like:

```
    # recs = model.browse(cr, uid, ids, context)
    names = recs.method(args)

    names = model.method(cr, uid, ids, args, context=context)
```

Deprecated since version 9.0: *one()* often makes the code less clear and behaves in ways developers and readers may not expect.

It is strongly recommended to use *multi()* and either iterate on the `self` recordset or ensure that the recordset is a single record with *ensure_one()*.

openerp.api.**v7**(*method_v7*)

Decorate a method that supports the old-style api only. A new-style api may be provided by redefining a method with the same name and decorated with *v8()*:

```
@api.v7
def foo(self, cr, uid, ids, context=None):
    ...

@api.v8
def foo(self):
    ...
```

Note that the wrapper method uses the docstring of the first method.

openerp.api.**v8**(*method_v8*)

Decorate a method that supports the new-style api only. An old-style api may be provided by redefining a method with the same name and decorated with *v7()*:

```
@api.v8
def foo(self):
    ...

@api.v7
def foo(self, cr, uid, ids, context=None):
    ...
```

Note that the wrapper method uses the docstring of the first method.

4.1.8 Fields

Basic fields

class openerp.fields.**Field**(*string=None*, ***kwargs*)

The field descriptor contains the field definition, and manages accesses and assignments of the corresponding field on records. The following attributes may be provided when instanciating a field:

Parameters

- **string** – the label of the field seen by users (string); if not set, the ORM takes the field name in the class (capitalized).
- **help** – the tooltip of the field seen by users (string)
- **readonly** – whether the field is readonly (boolean, by default `False`)
- **required** – whether the value of the field is required (boolean, by default `False`)
- **index** – whether the field is indexed in database (boolean, by default `False`)
- **default** – the default value for the field; this is either a static value, or a function taking a recordset and returning a value
- **states** – a dictionary mapping state values to lists of UI attribute-value pairs; possible attributes are: 'readonly', 'required', 'invisible'. Note: Any state-based condition requires the `state` field value to be available on the client-side UI. This is typically done by including it in the relevant views, possibly made invisible if not relevant for the end-user.
- **groups** – comma-separated list of group xml ids (string); this restricts the field access to the users of the given groups only

- **copy** (*bool*) – whether the field value should be copied when the record is duplicated (default: `True` for normal fields, `False` for `one2many` and computed fields, including property fields and related fields)
- **oldname** (*string*) – the previous name of this field, so that ORM can rename it automatically at migration

Computed fields

One can define a field whose value is computed instead of simply being read from the database. The attributes that are specific to computed fields are given below. To define such a field, simply provide a value for the attribute `compute`.

Parameters

- **compute** – name of a method that computes the field
- **inverse** – name of a method that inverses the field (optional)
- **search** – name of a method that implement search on the field (optional)
- **store** – whether the field is stored in database (boolean, by default `False` on computed fields)
- **compute_sudo** – whether the field should be recomputed as superuser to bypass access rights (boolean, by default `False`)

The methods given for `compute`, `inverse` and `search` are model methods. Their signature is shown in the following example:

```
upper = fields.Char(compute='_compute_upper',
                    inverse='_inverse_upper',
                    search='_search_upper')

@api.depends('name')
def _compute_upper(self):
    for rec in self:
        rec.upper = rec.name.upper() if rec.name else False

def _inverse_upper(self):
    for rec in self:
        rec.name = rec.upper.lower() if rec.upper else False

def _search_upper(self, operator, value):
    if operator == 'like':
        operator = 'ilike'
    return [('name', operator, value)]
```

The compute method has to assign the field on all records of the invoked recordset. The decorator *openerp.api.depends()* must be applied on the compute method to specify the field dependencies; those dependencies are used to determine when to recompute the field; recomputation is automatic and guarantees cache/database consistency. Note that the same method can be used for several fields, you simply have to assign all the given fields in the method; the method will be invoked once for all those fields.

By default, a computed field is not stored to the database, and is computed on-the-fly. Adding the attribute `store=True` will store the field's values in the database. The advantage of a stored field is that searching on that field is done by the database itself. The disadvantage is that it requires database updates when the field must be recomputed.

The inverse method, as its name says, does the inverse of the compute method: the invoked records have a value for the field, and you must apply the necessary changes on the field dependencies such that the computation gives the expected value. Note that a computed field without an inverse method is readonly by default.

The search method is invoked when processing domains before doing an actual search on the model. It must return a domain equivalent to the condition: `field operator value`.

Related fields

The value of a related field is given by following a sequence of relational fields and reading a field on the reached model. The complete sequence of fields to traverse is specified by the attribute

Parameters **`related`** – sequence of field names

Some field attributes are automatically copied from the source field if they are not redefined: `string`, `help`, `readonly`, `required` (only if all fields in the sequence are required), `groups`, `digits`, `size`, `translate`, `sanitize`, `selection`, `comodel_name`, `domain`, `context`. All semantic-free attributes are copied from the source field.

By default, the values of related fields are not stored to the database. Add the attribute `store=True` to make it stored, just like computed fields. Related fields are automatically recomputed when their dependencies are modified.

Company-dependent fields

Formerly known as 'property' fields, the value of those fields depends on the company. In other words, users that belong to different companies may see different values for the field on a given record.

Parameters **`company_dependent`** – whether the field is company-dependent (boolean)

Incremental definition

A field is defined as class attribute on a model class. If the model is extended (see *`Model`*), one can also extend the field definition by redefining a field with the same name and same type on the subclass. In that case, the attributes of the field are taken from the parent class and overridden by the ones given in subclasses.

For instance, the second class below only adds a tooltip on the field `state`:

```
class First(models.Model):
    _name = 'foo'
    state = fields.Selection([...], required=True)

class Second(models.Model):
    _inherit = 'foo'
    state = fields.Selection(help="Blah blah blah")
```

class openerp.fields.**Char**(*string=None*, ***kwargs*)
Bases: openerp.fields._String

Basic string field, can be length-limited, usually displayed as a single-line string in clients

Parameters

- **`size`** (*int*) – the maximum size of values stored for that field
- **`translate`** (*bool*) – whether the values of this field can be translated

class openerp.fields.**Boolean**(*string=None*, ***kwargs*)
Bases: *openerp.fields.Field*

class openerp.fields.**Integer**(*string=None*, ***kwargs*)
Bases: *openerp.fields.Field*

class openerp.fields.**Float**(*string=None*, *digits=None*, ***kwargs*)
Bases: *openerp.fields.Field*

The precision digits are given by the attribute

Parameters **digits** – a pair (total, decimal), or a function taking a database cursor and returning a pair (total, decimal)

class openerp.fields.**Text**(*string=None*, ***kwargs*)
Bases: openerp.fields._String

Very similar to *Char* but used for longer contents, does not have a size and usually displayed as a multiline text box.

Parameters **translate** – whether the value of this field can be translated

class openerp.fields.**Selection**(*selection=None*, *string=None*, ***kwargs*)
Bases: *openerp.fields.Field*

Parameters

- **selection** – specifies the possible values for this field. It is given as either a list of pairs (value, string), or a model method, or a method name.
- **selection_add** – provides an extension of the selection in the case of an overridden field. It is a list of pairs (value, string).

The attribute selection is mandatory except in the case of *related fields* or *field extensions*.

class openerp.fields.**Html**(*string=None*, ***kwargs*)
Bases: openerp.fields._String

class openerp.fields.**Date**(*string=None*, ***kwargs*)
Bases: *openerp.fields.Field*

static **context_today**(*record*, *timestamp=None*)
Return the current date as seen in the client's timezone in a format fit for date fields. This method may be used to compute default values.

Parameters **timestamp** (*datetime*) – optional datetime value to use instead of the current date and time (must be a datetime, regular dates can't be converted between timezones.)

Return type str

static **from_string**(*value*)
Convert an ORM value into a date value.

static **to_string**(*value*)
Convert a date value into the format expected by the ORM.

static **today**(**args*)
Return the current day in the format expected by the ORM. This function may be used to compute default values.

class openerp.fields.**Datetime**(*string=None*, ***kwargs*)
Bases: *openerp.fields.Field*

static **context_timestamp**(*record*, *timestamp*)
Returns the given timestamp converted to the client's timezone. This method is *not* meant for use as a _defaults initializer, because datetime fields are automatically converted upon display on client side. For _defaults you fields.datetime.now() should be used instead.

Parameters **timestamp** (*datetime*) – naive datetime value (expressed in UTC) to be converted to the client timezone

Return type datetime

Returns timestamp converted to timezone-aware datetime in context timezone

static **from_string**(*value*)
Convert an ORM value into a datetime value.

static **now**(**args*)
Return the current day and time in the format expected by the ORM. This function may be used to compute default values.

static to_string(*value*)
Convert a `datetime` value into the format expected by the ORM.

Relational fields

class `openerp.fields.`**Many2one**(*comodel_name=None*, *string=None*, ***kwargs*)
Bases: `openerp.fields._Relational`

The value of such a field is a recordset of size 0 (no record) or 1 (a single record).

Parameters

- **comodel_name** – name of the target model (string)
- **domain** – an optional domain to set on candidate values on the client side (domain or string)
- **context** – an optional context to use on the client side when handling that field (dictionary)
- **ondelete** – what to do when the referred record is deleted; possible values are: `'set null'`, `'restrict'`, `'cascade'`
- **auto_join** – whether JOINs are generated upon search through that field (boolean, by default `False`)
- **delegate** – set it to `True` to make fields of the target model accessible from the current model (corresponds to `_inherits`)

The attribute `comodel_name` is mandatory except in the case of related fields or field extensions.

class `openerp.fields.`**One2many**(*comodel_name=None*, *inverse_name=None*, *string=None*, ***kwargs*)
Bases: `openerp.fields._RelationalMulti`

One2many field; the value of such a field is the recordset of all the records in `comodel_name` such that the field `inverse_name` is equal to the current record.

Parameters

- **comodel_name** – name of the target model (string)
- **inverse_name** – name of the inverse `Many2one` field in `comodel_name` (string)
- **domain** – an optional domain to set on candidate values on the client side (domain or string)
- **context** – an optional context to use on the client side when handling that field (dictionary)
- **auto_join** – whether JOINs are generated upon search through that field (boolean, by default `False`)
- **limit** – optional limit to use upon read (integer)

The attributes `comodel_name` and `inverse_name` are mandatory except in the case of related fields or field extensions.

class `openerp.fields.`**Many2many**(*comodel_name=None*, *relation=None*, *column1=None*, *column2=None*, *string=None*, ***kwargs*)
Bases: `openerp.fields._RelationalMulti`

Many2many field; the value of such a field is the recordset.

Parameters **comodel_name** – name of the target model (string)

The attribute `comodel_name` is mandatory except in the case of related fields or field extensions.

Parameters

- **relation** – optional name of the table that stores the relation in the database (string)
- **column1** – optional name of the column referring to "these" records in the table relation (string)
- **column2** – optional name of the column referring to "those" records in the table relation (string)

The attributes relation, column1 and column2 are optional. If not given, names are automatically generated from model names, provided model_name and comodel_name are different!

Parameters

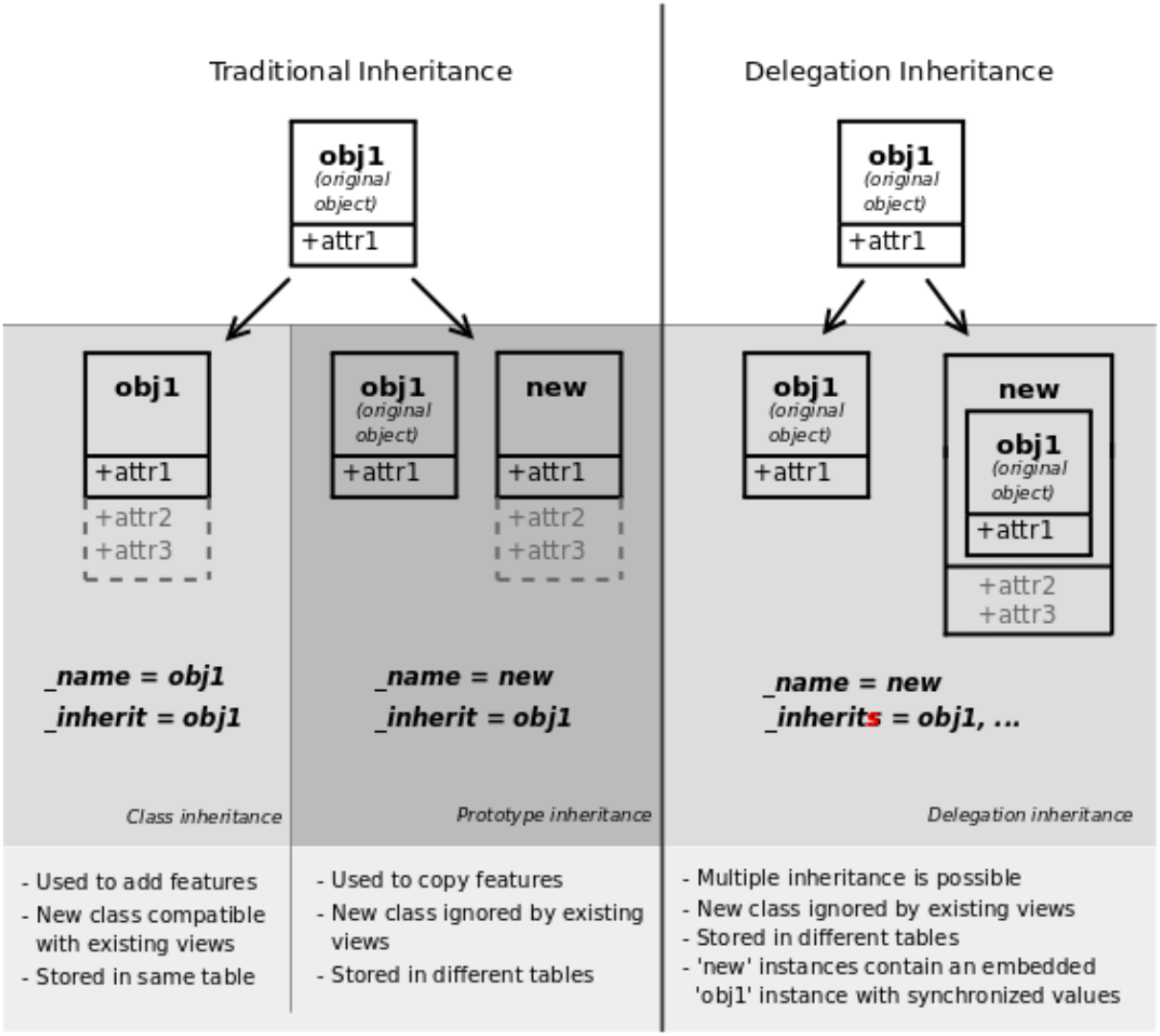
- **domain** – an optional domain to set on candidate values on the client side (domain or string)
- **context** – an optional context to use on the client side when handling that field (dictionary)
- **limit** – optional limit to use upon read (integer)

class openerp.fields.**Reference**(*selection=None*, *string=None*, ***kwargs*)
Bases: *openerp.fields.Selection*

4.1.9 Inheritance and extension

Odoo provides three different mechanisms to extend models in a modular way:

- creating a new model from an existing one, adding new information to the copy but leaving the original module as-is
- extending models defined in other modules in-place, replacing the previous version
- delegating some of the model's fields to records it contains



Classical inheritance

When using the *_inherit* and *_name* attributes together, Odoo creates a new model using the existing one (provided via *_inherit*) as a base. The new model gets all the fields, methods and meta-information (defaults & al) from its base.

```
class Inheritance0(models.Model):
    _name = 'inheritance.0'

    name = fields.Char()

    def call(self):
        return self.check("model 0")

    def check(self, s):
        return "This is {} record {}".format(s, self.name)

class Inheritance1(models.Model):
    _name = 'inheritance.1'
    _inherit = 'inheritance.0'

    def call(self):
        return self.check("model 1")
```

and using them:

```
        a = env['inheritance.0'].create({'name': 'A'})
        a.call()
        b = env['inheritance.1'].create({'name': 'B'})
        b.call()
```

will yield:

```
        This is model 0 record A
        This is model 1 record B
```

the second model has inherited from the first model's `check` method and its `name` field, but overridden the `call` method, as when using standard Python inheritance.

Extension

When using *`_inherit`* but leaving out *`_name`*, the new model replaces the existing one, essentially extending it in-place. This is useful to add new fields or methods to existing models (created in other modules), or to customize or reconfigure them (e.g. to change their default sort order):

```
class Extension0(models.Model):
    _name = 'extension.0'

    name = fields.Char(default="A")

class Extension1(models.Model):
    _inherit = 'extension.0'

    description = fields.Char(default="Extended")
```

```
        record = env['extension.0'].create({})
        record.read()[0]
```

will yield:

```
        {'name': "A", 'description': "Extended"}
```

Note: it will also yield the various *automatic fields* unless they've been disabled

Delegation

The third inheritance mechanism provides more flexibility (it can be altered at runtime) but less power: using the *`_inherits`* a model *delegates* the lookup of any field not found on the current model to "children" models. The delegation is performed via *`Reference`* fields automatically set up on the parent model:

```
class Child0(models.Model):
    _name = 'delegation.child0'

    field_0 = fields.Integer()

class Child1(models.Model):
    _name = 'delegation.child1'

    field_1 = fields.Integer()

class Delegating(models.Model):
    _name = 'delegation.parent'

    _inherits = {
        'delegation.child0': 'child0_id',
```

```
        'delegation.child1': 'child1_id',
    }

    child0_id = fields.Many2one('delegation.child0', required=True, ondelete='cascade')
    child1_id = fields.Many2one('delegation.child1', required=True, ondelete='cascade')
```

```
        record = env['delegation.parent'].create({
            'child0_id': env['delegation.child0'].create({'field_0': 0}).id,
            'child1_id': env['delegation.child1'].create({'field_1': 1}).id,
        })
        record.field_0
        record.field_1
```

will result in:

```
        0
        1
```

and it's possible to write directly on the delegated field:

```
        record.write({'field_1': 4})
```

Warning: when using delegation inheritance, methods are *not* inherited, only fields

4.1.10 Domains

A domain is a list of criteria, each criterion being a triple (either a `list` or a `tuple`) of `(field_name, operator, value)` where:

`field_name` (`str`) a field name of the current model, or a relationship traversal through a *`Many2one`* using dot-notation e.g. `'street'` or `'partner_id.country'`

`operator` (`str`) an operator used to compare the `field_name` with the `value`. Valid operators are:

`=` equals to

`!=` not equals to

`>` greater than

`>=` greater than or equal to

`<` less than

`<=` less than or equal to

`=?` unset or equals to (returns true if `value` is either `None` or `False`, otherwise behaves like =)

`=like` matches `field_name` against the `value` pattern. An underscore _ in the pattern stands for (matches) any single character; a percent sign `%` matches any string of zero or more characters.

`like` matches `field_name` against the `%value%` pattern. Similar to `=like` but wraps `value` with '%' before matching

`not like` doesn't match against the `%value%` pattern

`ilike` case insensitive `like`

`not ilike` case insensitive `not like`

`=ilike` case insensitive `=like`

`in` is equal to any of the items from `value`, `value` should be a list of items

`not in` is unequal to all of the items from `value`

child_of is a child (descendant) of a `value` record.

Takes the semantics of the model into account (i.e following the relationship field named by `_parent_name`).

value variable type, must be comparable (through `operator`) to the named field

Domain criteria can be combined using logical operators in *prefix* form:

'&' logical *AND*, default operation to combine criteria following one another. Arity 2 (uses the next 2 criteria or combinations).

'|' logical *OR*, arity 2.

'!' logical *NOT*, arity 1.

Tip: Mostly to negate combinations of criteria

Individual criterion generally have a negative form (e.g. = -> !=, < -> >=) which is simpler than negating the positive.

Example

To search for partners named *ABC*, from belgium or germany, whose language is not english:

```
[('name','=','ABC'),
 ('language.code','!=','en_US'),
 '|',('country_id.code','=','be'),
     ('country_id.code','=','de')]
```

This domain is interpreted as:

```
    (name is 'ABC')
AND (language is NOT english)
AND (country is Belgium OR Germany)
```

4.1.11 Porting from the old API to the new API

- bare lists of ids are to be avoided in the new API, use recordsets instead
- methods still written in the old API should be automatically bridged by the ORM, no need to switch to the old API, just call them as if they were a new API method. See *Automatic bridging of old API methods* for more details.
- *`search()`* returns a recordset, no point in e.g. browsing its result
- `fields.related` and `fields.function` are replaced by using a normal field type with either a `related=` or a `compute=` parameter
- *`depends()`* on `compute=` methods **must be complete**, it must list **all** the fields and sub-fields which the compute method uses. It is better to have too many dependencies (will recompute the field in cases where that is not needed) than not enough (will forget to recompute the field and then values will be incorrect)
- **remove** all `onchange` methods on computed fields. Computed fields are automatically re-computed when one of their dependencies is changed, and that is used to auto-generate `onchange` by the client
- the decorators *`model()`* and *`multi()`* are for bridging *when calling from the old API context*, for internal or pure new-api (e.g. compute) they are useless
- remove `_default`, replace by `default=` parameter on corresponding fields
- if a field's `string=` is the titlecased version of the field name:

```
    name = fields.Char(string="Name")
```

  it is useless and should be removed

- the `multi=` parameter does not do anything on new API fields use the same `compute=` methods on all relevant fields for the same result
- provide `compute=`, `inverse=` and `search=` methods by name (as a string), this makes them overridable (removes the need for an intermediate "trampoline" function)
- double check that all fields and methods have different names, there is no warning in case of collision (because Python handles it before Odoo sees anything)
- the normal new-api import is `from openerp import fields, models`. If compatibility decorators are necessary, use `from openerp import api, fields, models`
- avoid the *one()* decorator, it probably does not do what you expect
- remove explicit definition of *create_uid*, *create_date*, *write_uid* and *write_date* fields: they are now created as regular "legitimate" fields, and can be read and written like any other field out-of-the-box
- when straight conversion is impossible (semantics can not be bridged) or the "old API" version is not desirable and could be improved for the new API, it is possible to use completely different "old API" and "new API" implementations for the same method name using *v7()* and *v8()*. The method should first be defined using the old-API style and decorated with *v7()*, it should then be re-defined using the exact same name but the new-API style and decorated with *v8()*. Calls from an old-API context will be dispatched to the first implementation and calls from a new-API context will be dispatched to the second implementation. One implementation can call (and frequently does) call the other by switching context.

  > **Danger:** using these decorators makes methods extremely difficult to override and harder to understand and document

- uses of `_columns` or `_all_columns` should be replaced by `_fields`, which provides access to instances of new-style *openerp.fields.Field* instances (rather than old-style `openerp.osv.fields._column`).

  Non-stored computed fields created using the new API style are *not* available in `_columns` and can only be inspected through `_fields`

- reassigning `self` in a method is probably unnecessary and may break translation introspection
- `Environment` objects rely on some threadlocal state, which has to be set up before using them. It is necessary to do so using the `openerp.api.Environment.manage()` context manager when trying to use the new API in contexts where it hasn't been set up yet, such as new threads or a Python interactive environment:

```
>>> from openerp import api, modules
>>> r = modules.registry.RegistryManager.get('test')
>>> cr = r.cursor()
>>> env = api.Environment(cr, 1, {})
Traceback (most recent call last):
  ...
AttributeError: environments
>>> with api.Environment.manage():
...     env = api.Environment(cr, 1, {})
...     print env['res.partner'].browse(1)
...
res.partner(1,)
```

Automatic bridging of old API methods

When models are initialized, all methods are automatically scanned and bridged if they look like models declared in the old API style. This bridging makes them transparently callable from new-API-style methods.

Methods are matched as "old-API style" if their second positional parameter (after `self`) is called either `cr` or `cursor`. The system also recognizes the third positional parameter being called `uid` or `user` and the fourth

being called `id` or `ids`. It also recognizes the presence of any parameter called `context`.

When calling such methods from a new API context, the system will automatically fill matched parameters from the current `Environment` (for `cr`, `user` and `context`) or the current recordset (for `id` and `ids`).

In the rare cases where it is necessary, the bridging can be customized by decorating the old-style method:

- disabling it entirely, by decorating a method with `noguess()` there will be no bridging and methods will be called the exact same way from the new and old API styles
- defining the bridge explicitly, this is mostly for methods which are matched incorrectly (because parameters are named in unexpected ways):

  `cr()` will automatically prepend the current cursor to explicitly provided parameters, positionally

  `cr_uid()` will automatically prepend the current cursor and user's id to explictly provided parameters

  `cr_uid_ids()` will automatically prepend the current cursor, user's id and recordset's ids to explicitly provided parameters

  `cr_uid_id()` will loop over the current recordset and call the method once for each record, prepending the current cursor, user's id and record's id to explicitly provided parameters.

  > **Danger:** the result of this wrapper is *always a list* when calling from a new-API context

  All of these methods have a `_context`-suffixed version (e.g. `cr_uid_context()`) which also passes the current context *by keyword*.

- dual implementations using *`v7()`* and *`v8()`* will be ignored as they provide their own "bridging"

4.2 Data Files

Odoo is greatly data-driven, and a big part of modules definition is thus the definition of the various records it manages: UI (menus and views), security (access rights and access rules), reports and plain data are all defined via records.

4.2.1 Structure

The main way to define data in Odoo is via XML data files: The broad structure of an XML data file is the following:

- The nested root elements `openerp` and `data`
- Any number of operation elements within `data`

```
<!-- the root elements of the data file -->
<openerp><data>
  <operation/>
  ...
</data></openerp>
```

Data files are executed sequentially, operations can only refer to the result of operations defined previously

4.2.2 Core operations

`record`

`record` appropriately defines or updates a database record, it has the following attributes:

`model` (required) name of the model to create (or update)

`id` the external identifier for this record. It is strongly recommended to provide one

- for record creation, allows subsequent definitions to either modify or refer to this record
- for record modification, the record to modify

context context to use when creating the record

forcecreate in update mode whether the record should be created if it doesn't exist

Requires an external id, defaults to `True`.

field

Each record can be composed of `field` tags, defining values to set when creating the record. A `record` with no `field` will use all default values (creation) or do nothing (update).

A `field` has a mandatory `name` attribute, the name of the field to set, and various methods to define the value itself:

Nothing if no value is provided for the field, an implicit `False` will be set on the field. Can be used to clear a field, or avoid using a default value for the field.

search for *relational fields*, should be a *domain* on the field's model.

Will evaluate the domain, search the field's model using it and set the search's result as the field's value. Will only use the first result if the field is a *Many2one*

ref if a `ref` attribute is provided, its value must be a valid external id, which will be looked up and set as the field's value.

Mostly for *Many2one* and *Reference* fields

type if a `type` attribute is provided, it is used to interpret and convert the field's content. The field's content can be provided through an external file using the `file` attribute, or through the node's body.

Available types are:

xml, html extracts the `field`'s children as a single document, evaluates any external id specified with the form `%(external_id)s`. `%%` can be used to output actual % signs.

file ensures that the field content is a valid file path in the current model, saves the pair *module*,*path* as the field value

char sets the field content directly as the field's value without alterations

base64 base64-encodes the field's content, useful combined with the `file` *attribute* to load e.g. image data into attachments

int converts the field's content to an integer and sets it as the field's value

float converts the field's content to a float and sets it as the field's value

list, tuple should contain any number of `value` elements with the same properties as `field`, each element resolves to an item of a generated tuple or list, and the generated collection is set as the field's value

eval for cases where the previous methods are unsuitable, the `eval` attributes simply evaluates whatever Python expression it is provided and sets the result as the field's value.

The evaluation context contains various modules (`time`, `datetime`, `timedelta`, `relativedelta`), a function to resolve external identifiers (`ref`) and the model object for the current field if applicable (`obj`)

delete

The `delete` tag can remove any number of records previously defined. It has the following attributes:

model (required) the model in which a specified record should be deleted

id the external id of a record to remove

search a *domain* to find records of the model to remove

`id` and `search` are exclusive

function

The `function` tag calls a method on a model, with provided parameters. It has two mandatory parameters `model` and `name` specifying respectively the model and the name of the method to call.

Parameters can be provided using `eval` (should evaluate to a sequence of parameters to call the method with) or `value` elements (see `list` values).

workflow

The `workflow` tag sends a signal to an existing workflow. The workflow can be specified via a `ref` attribute (the external id of an existing workflow) or a `value` tag returning the id of a workflow.

The tag also has two mandatory attributes `model` (the model linked to the workflow) and `action` (the name of the signal to send to the workflow).

4.2.3 Shortcuts

Because some important structural models of Odoo are complex and involved, data files provide shorter alternatives to defining them using *record tags*:

menuitem

Defines an `ir.ui.menu` record with a number of defaults and fallbacks:

Parent menu

- If a `parent` attribute is set, it should be the external id of an other menu item, used as the new item's parent
- If no `parent` is provided, tries to interpret the `name` attribute as a `/`-separated sequence of menu names and find a place in the menu hierarchy. In that interpretation, intermediate menus are automatically created
- Otherwise the menu is defined as a "top-level" menu item (*not* a menu with no parent)

Menu name If no `name` attribute is specified, tries to get the menu name from a linked action if any. Otherwise uses the record's `id`

Groups A `groups` attribute is interpreted as a comma-separated sequence of external identifiers for `res.groups` models. If an external identifier is prefixed with a minus (−), the group is *removed* from the menu's groups

action if specified, the `action` attribute should be the external id of an action to execute when the menu is open

id the menu item's external id

template

Creates a *QWeb view* requiring only the `arch` section of the view, and allowing a few *optional* attributes:

id the view's external identifier

name, inherit_id, priority same as the corresponding field on `ir.ui.view` (nb: `inherit_id` should be an external identifier)

primary if set to `True` and combined with a `inherit_id`, defines the view as a primary

groups comma-separated list of group external identifiers

page if set to `"True"`, the template is a website page (linkable to, deletable)

optional `enabled` or `disabled`, whether the view can be disabled (in the website interface) and its default status. If unset, the view is always enabled.

report

Creates a `ir.actions.report.xml` record with a few default values.

Mostly just proxies attributes to the corresponding fields on `ir.actions.report.xml`, but also automatically creates the item in the *More* menu of the report's `model`.

4.2.4 CSV data files

XML data files are flexible and self-descriptive, but very verbose when creating a number of simple records of the same model in bulk.

For this case, data files can also use csv, this is often the case for *access rights*:

- the file name is *`model_name`*`.csv`
- the first row lists the fields to write, with the special field `id` for external identifiers (used for creation or update)
- each row thereafter creates a new record

Here's the first lines of the data file defining US states `res.country.state.csv`

```
"id","country_id:id","name","code"
state_au_1,au,"Australian Capital Territory","ACT"
state_au_2,au,"New South Wales","NSW"
state_au_3,au,"Northern Territory","NT"
state_au_4,au,"Queensland","QLD"
state_au_5,au,"South Australia","SA"
state_au_6,au,"Tasmania","TAS"
state_au_7,au,"Victoria","VIC"
state_au_8,au,"Western Australia","WA"
state_us_1,us,"Alabama","AL"
state_us_2,us,"Alaska","AK"
state_us_3,us,"Arizona","AZ"
state_us_4,us,"Arkansas","AR"
state_us_5,us,"California","CA"
state_us_6,us,"Colorado","CO"
```

rendered in a more readable format:

id	country_id:id	name	code
state_au_1	au	Australian Capital Territory	ACT
state_au_2	au	New South Wales	NSW
state_au_3	au	Northern Territory	NT
state_au_4	au	Queensland	QLD
state_au_5	au	South Australia	SA
state_au_6	au	Tasmania	TAS
state_au_7	au	Victoria	VIC
state_au_8	au	Western Australia	WA
state_us_1	us	Alabama	AL
state_us_2	us	Alaska	AK

Continued on next page

Table 4.1 – continued from previous page

id	country_id:id	name	code
state_us_3	us	Arizona	AZ
state_us_4	us	Arkansas	AR
state_us_5	us	California	CA
state_us_6	us	Colorado	CO
state_us_7	us	Connecticut	CT
state_us_8	us	Delaware	DE
state_us_9	us	District of Columbia	DC
state_us_10	us	Florida	FL
state_us_11	us	Georgia	GA
state_us_12	us	Hawaii	HI
state_us_13	us	Idaho	ID
state_us_14	us	Illinois	IL
state_us_15	us	Indiana	IN
state_us_16	us	Iowa	IA
state_us_17	us	Kansas	KS
state_us_18	us	Kentucky	KY
state_us_19	us	Louisiana	LA
state_us_20	us	Maine	ME
state_us_21	us	Montana	MT
state_us_22	us	Nebraska	NE
state_us_23	us	Nevada	NV
state_us_24	us	New Hampshire	NH
state_us_25	us	New Jersey	NJ
state_us_26	us	New Mexico	NM
state_us_27	us	New York	NY
state_us_28	us	North Carolina	NC
state_us_29	us	North Dakota	ND
state_us_30	us	Ohio	OH
state_us_31	us	Oklahoma	OK
state_us_32	us	Oregon	OR
state_us_33	us	Maryland	MD
state_us_34	us	Massachusetts	MA
state_us_35	us	Michigan	MI
state_us_36	us	Minnesota	MN
state_us_37	us	Mississippi	MS
state_us_38	us	Missouri	MO
state_us_39	us	Pennsylvania	PA
state_us_40	us	Rhode Island	RI
state_us_41	us	South Carolina	SC
state_us_42	us	South Dakota	SD
state_us_43	us	Tennessee	TN
state_us_44	us	Texas	TX
state_us_45	us	Utah	UT
state_us_46	us	Vermont	VT
state_us_47	us	Virginia	VA
state_us_48	us	Washington	WA
state_us_49	us	West Virginia	WV
state_us_50	us	Wisconsin	WI
state_us_51	us	Wyoming	WY

For each row (record):

- the first column is the external id of the record to create or update
- the second column is the external id of the country object to link to (country objects must have been defined

beforehand)

- the third column is the `name` field for `res.country.state`
- the fourth column is the `code` field for `res.country.state`

4.3 Actions

Actions define the behavior of the system in response to user actions: login, action button, selection of an invoice, ...

Actions can be stored in the database or returned directly as dictionaries in e.g. button methods. All actions share two mandatory attributes:

`type` the category of the current action, determines which fields may be used and how the action is interpreted

`name` short user-readable description of the action, may be displayed in the client's interface

A client can get actions in 4 forms:

`False` if any action dialog is currently open, close it

A string if a *client action* matches, interpret as a client action's tag, otherwise treat as a number

A number read the corresponding action record from the database, may be a database identifier or an external id

A dictionary treat as a client action descriptor and execute

4.3.1 Window Actions (`ir.actions.act_window`)

The most common action type, used to present visualisations of a model through *views*: a window action defines a set of view types (and possibly specific views) for a model (and possibly specific record of the model).

Its fields are:

`res_model` model to present views for

`views` a list of `(view_id, view_type)` pairs. The second element of each pair is the category of the view (tree, form, graph, ...) and the first is an optional database id (or `False`). If no id is provided, the client should fetch the default view of the specified type for the requested model (this is automatically done by *`fields_view_get()`*). The first type of the list is the default view type and will be open by default when the action is executed. Each view type should be present at most once in the list

`res_id` (optional) if the default view is `form`, specifies the record to load (otherwise a new record should be created)

`search_view_id` (optional) `(id, name)` pair, `id` is the database identifier of a specific search view to load for the action. Defaults to fetching the default search view for the model

`target` (optional) whether the views should be open in the main content area (`current`) or in a dialog/popup (`new`). Defaults to `current`.

`context` (optional) additional context data to pass to the views

`domain` (optional) filtering domain to implicitly add to all view search queries

`limit` (optional) number of records to display in lists by default. Defaults to 80 in the web client

`auto_search` (optional) whether a search should be performed immediately after loading the default view. Defaults to `True`

For instance, to open customers (partner with the `customer` flag set) with list and form views:

```
{
    "type": "ir.actions.act_window",
    "res_model": "res.partner",
    "views": [[False, "tree"], [False, "form"]],
    "domain": [["customer", "=", true]],
}
```

Or to open the form view of a specific product (obtained separately) in a new dialog:

```
{
    "type": "ir.actions.act_window",
    "res_model": "product.product",
    "views": [[False, "form"]],
    "res_id": a_product_id,
    "target": "new",
}
```

In-database window actions have a few different fields which should be ignored by clients, mostly to use in composing the `views` list:

`view_mode` comma-separated list of view types (tree, form, ...) as a string. All of these types will be present in the generated `views` list (with at least a `False` view_id)

`view_ids` M2M[1] to view objects, defines the initial content of `views`

`view_id` specific view added to the `views` list in case its type is part of the `view_mode` list and not already filled by one of the views in `view_ids`

These are mostly used when defining actions from *Data Files*:

```
<record model="ir.actions.act_window" id="test_action">
    <field name="name">A Test Action</field>
    <field name="res_model">some.model</field>
    <field name="view_mode">graph</field>
    <field name="view_id" ref="my_specific_view"/>
</record>
```

will use the "my_specific_view" view even if that's not the default view for the model.

The server-side composition of the `views` sequence is the following:

- get each `(id, type)` from `view_ids` (ordered by `sequence`)
- if `view_id` is defined and its type isn't already filled, append its `(id, type)`
- for each unfilled type in `view_mode`, append `(False, type)`

4.3.2 URL Actions (`ir.actions.act_url`)

Allow opening a URL (website/web page) via an Odoo action. Can be customized via two fields:

`url` the address to open when activating the action

`target` opens the address in a new window/page if `new`, replaces the current content with the page if `self`. Defaults to `new`

```
{
    "type": "ir.actions.act_url",
    "url": "http://odoo.com",
    "target": "self",
}
```

will replace the current content section by the Odoo home page.

[1] technically not an M2M: adds a sequence field and may be composed of just a view type, without a view id.

4.3.3 Server Actions (`ir.actions.server`)

Allow triggering complex server code from any valid action location. Only two fields are relevant to clients:

`id` the in-database identifier of the server action to run

`context` (optional) context data to use when running the server action

In-database records are significantly richer and can perform a number of specific or generic actions based on their `state`. Some fields (and corresponding behaviors) are shared between states:

`model_id` Odoo model linked to the action, made available in *evaluation contexts*

`condition` (optional) evaluated as Python code using the server action's *evaluation context*. If `False`, prevents the action from running. Default: `True`

Valid action types (`state` field) are extensible, the default types are:

`code`

The default and most flexible server action type, executes arbitrary Python code with the action's *evaluation context*. Only uses one specific type-specific field:

`code` a piece of Python code to execute when the action is called

```
<record model="ir.actions.server" id="print_instance">
    <field name="name">Res Partner Server Action</field>
    <field name="model_id" ref="model_res_partner"/>
    <field name="code">
        print self, object
    </field>
</record>
```

Note: The code segment can define a variable called `action`, which will be returned to the client as the next action to execute:

```
<record model="ir.actions.server" id="print_instance">
    <field name="name">Res Partner Server Action</field>
    <field name="model_id" ref="model_res_partner"/>
    <field name="code">
        if object.some_condition():
            action = {
                "type": "ir.actions.act_window",
                "view_mode": "form",
                "res_model": object._name,
                "res_id": object.id,
            }
    </field>
</record>
```

will ask the client to open a form for the record if it fullfils some condition

---

This tends to be the only action type created from *data files*, other types aside from *multi* are simpler than Python code to define from the UI, but not from *data files*.

`object_create`

Creates a new record, from scratch (via *`create()`*) or by copying an existing record (via *`copy()`*)

`use_create` the creation policy, one of:

`new` creates a record in the model specified by `model_id`

`new_other` creates a record in the model specified by `crud_model_id`

copy_current copies the record on which the action was invoked

copy_other copies an other record, obtained via `ref_object`

fields_lines fields to override when creating or copying the record. *One2many* with the fields:

col1 `ir.model.fields` to set in the model implied by `use_create`

value value for the field, interpreted via `type`

type If `value`, the `value` field is interpreted as a literal value (possibly converted), if `equation` the `value` field is interpreted as a Python expression and evaluated

crud_model_id model in which to create a new record, if `use_create` is set to `new_other`

ref_object *Reference* to an arbitrary record to copy, used if `use_create` is set to `copy_other`

link_new_record boolean flag linking the newly created record to the current one via a many2one field specified through `link_field_id`, defaults to `False`

link_field_id many2one to `ir.model.fields`, specifies the current record's m2o field on which the newly created record should be set (models should match)

object_write

Similar to *object_create* but alters an existing records instead of creating one

use_write write policy, one of:

current write to the current record

other write to an other record selected via `crud_model_id` and `ref_object`

expression write to an other record whose model is selected via `crud_model_id` and whose id is selected by evaluating `write_expression`

write_expression Python expression returning a record or an object id, used when `use_write` is set to `expression` in order to decide which record should be modified

fields_lines see *object_create*

crud_model_id see *object_create*

ref_object see *object_create*

multi

Executes multiple actions one after the other. Actions to execute are defined via the `child_ids` m2m. If sub-actions themselves return actions, the last one will be returned to the client as the multi's own next action

trigger

Sends a signal to a workflow.

wkf_transition_id *Many2one* to a `workflow.transition` to trigger

use_relational_model if `base` (the default), trigger the signal on behalf of the current record. If `relational`, trigger the signal on behalf of a field of the current record selected through `wkf_model_id` and `wkf_field_id`

client_action

Indirection for directly returning an other action defined using `action_id`. Simply returns that action to the client for execution.

Evaluation context

A number of keys are available in the evaluation context of or surrounding server actions:

self the model object linked to the action via `model_id`

object, obj only available if `active_model` and `active_id` are provided (via context) otherwise `None`. The actual record selected by `active_id`

pool the current database registry

datetime, dateutil, time corresponding Python modules

cr the current cursor

user the current user record

context execution context

Warning constructor for the `Warning` exception

4.3.4 Report Actions (`ir.actions.report.xml`)

Triggers the printing of a report

name (mandatory) only useful as a mnemonic/description of the report when looking for one in a list of some sort

model (mandatory) the model your report will be about

report_type (mandatory) either `qweb-pdf` for PDF reports or `qweb-html` for HTML

report_name the name of your report (which will be the name of the PDF output)

groups_id *Many2many* field to the groups allowed to view/use the current report

paperformat_id *Many2one* field to the paper format you wish to use for this report (if not specified, the company format will be used)

attachment_use if set to `True`, the report is only generated once the first time it is requested, and re-printed from the stored report afterwards instead of being re-generated every time.

Can be used for reports which must only be generated once (e.g. for legal reasons)

attachment python expression that defines the name of the report; the record is accessible as the variable `object`

4.3.5 Client Actions (`ir.actions.client`)

Triggers an action implemented entirely in the client.

tag the client-side identifier of the action, an arbitrary string which the client should know how to react to

params (optional) a Python dictionary of additional data to send to the client, alongside the client action tag

```
{
    "type": "ir.actions.client",
    "tag": "pos.ui"
}
```

tells the client to start the Point of Sale interface, the server has no idea how the POS interface works.

4.4 Views

4.4.1 Common Structure

View objects expose a number of fields, they are optional unless specified otherwise.

name (mandatory) only useful as a mnemonic/description of the view when looking for one in a list of some sort

model the model linked to the view, if applicable (it doesn't for QWeb views)

priority client programs can request views by `id`, or by `(model, type)`. For the latter, all the views for the right type and model will be searched, and the one with the lowest `priority` number will be returned (it is the "default view").

`priority` also defines the order of application during *view inheritance*

arch the description of the view's layout

groups_id *Many2many* field to the groups allowed to view/use the current view

inherit_id the current view's parent view, see *Inheritance*, unset by default

mode inheritance mode, see *Inheritance*. If `inherit_id` is unset the `mode` can only be `primary`. If `inherit_id` is set, `extension` by default but can be explicitly set to `primary`

application website feature defining togglable views. By default, views are always applied

4.4.2 Inheritance

View matching

- if a view is requested by `(model, type)`, the view with the right model and type, `mode=primary` and the lowest priority is matched
- when a view is requested by `id`, if its mode is not `primary` its *closest* parent with mode `primary` is matched

View resolution

Resolution generates the final `arch` for a requested/matched `primary` view:

1. if the view has a parent, the parent is fully resolved then the current view's inheritance specs are applied
2. if the view has no parent, its `arch` is used as-is
3. the current view's children with mode `extension` are looked up and their inheritance specs are applied depth-first (a child view is applied, then its children, then its siblings)

The result of applying children views yields the final `arch`

Inheritance specs

There are three types of inheritance specs:

- An `xpath` element with an `expr` attribute. `expr` is an XPath expression[2] applied to the current `arch`, the first node it finds is the match
- a `field` element with a `name` attribute, matches the first `field` with the same `name`

[2] an extension function is added for simpler matching in QWeb views: `hasclass(*classes)` matches if the context node has all the specified classes

- any other element, the first element with the same name and identical attributes (ignoring `position`) is matched

The inheritance spec may have an optional `position` attribute specifying how the matched node should be altered:

`inside` (default) the content of the inheritance spec is appended to the matched node

`replace` the content of the inheritance spec replaces the matched node

`after` the content of the inheritance spec is added to the matched node's parent, after the matched node

`before` the content of the inheritance spec is added to the matched node's parent, before the matched node

`attributes` the content of the inheritance spec should be `attribute` elements with a `name` attribute and an optional body:

- if the `attribute` element has a body, a new attributed named after its `name` is created on the matched node with the `attribute` element's text as value
- if the `attribute` element has no body, the attribute named after its `name` is removed from the matched node. If no such attribute exists, an error is raised

A view's specs are applied sequentially.

4.4.3 Lists

The root element of list views is `<tree>`[3]. The list view's root can have the following attributes:

`editable` by default, selecting a list view's row opens the corresponding *form view*. The `editable` attributes makes the list view itself editable in-place.

Valid values are `top` and `bottom`, making *new* records appear respectively at the top or bottom of the list.

The architecture for the inline *form view* is derived from the list view. Most attributes valid on a *form view*'s fields and buttons are thus accepted by list views although they may not have any meaning if the list view is non-editable

`default_order` overrides the ordering of the view, replacing the model's default order. The value is a comma-separated list of fields, postfixed by `desc` to sort in reverse order:

```
<tree default_order="sequence,name desc">
```

`colors` allows changing the color of a row's text based on the corresponding record's attributes.

Defined as a mapping of colors to Python expressions. Values are of the form: *`color:expr[;...]`*. For each record, pairs are tested in order, the expression is evaluated for the record and if `true` the corresponding color is applied to the row. If no color matches, uses the default text color (black).

- `color` can be any valid CSS color unit.
- `expr` should be a Python expression evaluated with the current record's attributes as context values. Other context values are `uid` (the id of the current user) and `current_date` (the current date as a string of the form `yyyy-MM-dd`)

`fonts` allows changing a row's font style based on the corresponding record's attributes.

The format is the same as for `color`, but the `color` of each pair is replaced by `bold`, `italic` or `underline`, the expression evaluating to `true` will apply the corresponding style to the row's text. Contrary to `colors`, multiple pairs can match each record

`create`, `edit`, `delete` allows *dis*abling the corresponding action in the view by setting the corresponding attribute to `false`

[3] for historical reasons, it has its origin in tree-type views later repurposed to a more table/list-type display

on_write only makes sense on an `editable` list. Should be the name of a method on the list's model. The method will be called with the `id` of a record after having created or edited that record (in database).

The method should return a list of ids of other records to load or update.

string alternative translatable label for the view

Deprecated since version 8.0: not displayed anymore

Possible children elements of the list view are:

button displays a button in a list cell

icon icon to use to display the button

string

- if there is no `icon`, the button's text
- if there is an `icon`, `alt` text for the icon

type type of button, indicates how it clicking it affects Odoo:

workflow (default) sends a signal to a workflow. The button's `name` is the workflow signal, the row's record is passed as argument to the signal

object call a method on the list's model. The button's `name` is the method, which is called with the current row's record id and the current context.

action load an execute an `ir.actions`, the button's `name` is the database id of the action. The context is expanded with the list's model (as `active_model`), the current row's record (`active_id`) and all the records currently loaded in the list (`active_ids`, may be just a subset of the database records matching the current search)

name see `type`

args see `type`

attrs dynamic attributes based on record values.

A mapping of attributes to domains, domains are evaluated in the context of the current row's record, if `True` the corresponding attribute is set on the cell.

Possible attributes are `invisible` (hides the button) and `readonly` (disables the button but still shows it)

states shorthand for `invisible` `attrs`: a list of states, comma separated, requires that the model has a `state` field and that it is used in the view.

Makes the button `invisible` if the record is *not* in one of the listed states

context merged into the view's context when performing the button's Odoo call

confirm confirmation message to display (and for the user to accept) before performing the button's Odoo call

field defines a column where the corresponding field should be displayed for each record. Can use the following attributes:

name the name of the field to display in the current model. A given name can only be used once per view

string the title of the field's column (by default, uses the `string` of the model's field)

invisible fetches and stores the field, but doesn't display the column in the table. Necessary for fields which shouldn't be displayed but are used by e.g. `@colors`

groups lists the groups which should be able to see the field

widget alternate representations for a field's display. Possible list view values are:

progressbar displays `float` fields as a progress bar.

many2onebutton replaces the m2o field's value by a checkmark if the field is filled, and a cross if it is not

handle for `sequence` fields, instead of displaying the field's value just displays a dra&drop icon

sum, avg displays the corresponding aggregate at the bottom of the column. The aggregation is only computed on *currently displayed* records. The aggregation operation must match the corresponding field's `group_operator`

attrs dynamic attributes based on record values. Only effects the current field, so e.g. `invisible` will hide the field but leave the same field of other records visible, it will not hide the column itself

---

Note: if the list view is `editable`, any field attribute from the *form view* is also valid and will be used when setting up the inline form view

---

4.4.4 Forms

Form views are used to display the data from a single record. Their root element is `<form>`. They are composed of regular HTML with additional structural and semantic components.

Structural components

Structural components provide structure or "visual" features with little logic. They are used as elements or sets of elements in form views.

notebook defines a tabbed section. Each tab is defined through a `page` child element. Pages can have the following attributes:

string (required) the title of the tab

accesskey an HTML accesskey

attrs standard dynamic attributes based on record values

group used to define column layouts in forms. By default, groups define 2 columns and most direct children of groups take a single column. `field` direct children of groups display a label by default, and the label and the field itself have a colspan of 1 each.

The number of columns in a `group` can be customized using the `col` attribute, the number of columns taken by an element can be customized using `colspan`.

Children are laid out horizontally (tries to fill the next column before changing row).

Groups can have a `string` attribute, which is displayed as the group's title

newline only useful within `group` elements, ends the current row early and immediately switches to a new row (without filling any remaining column beforehand)

separator small horizontal spacing, with a `string` attribute behaves as a section title

sheet can be used as a direct child to `form` for a narrower and more responsive form layout

header combined with `sheet`, provides a full-width location above the sheet itself, generally used to display workflow buttons and status widgets

Semantic components

Semantic components tie into and allow interaction with the Odoo system. Available semantic components are:

button call into the Odoo system, similar to *list view buttons*

field renders (and allow edition of, possibly) a single field of the current record. Possible attributes are:

name (mandatory) the name of the field to render

widget fields have a default rendering based on their type (e.g. *Char*, *Many2one*). The `widget` attributes allows using a different rendering method and context.

options JSON object specifying configuration option for the field's widget (including default widgets)

class HTML class to set on the generated element, common field classes are:

oe_inline prevent the usual line break following fields

oe_left, oe_right floats the field to the corresponding direction

oe_read_only, oe_edit_only only displays the field in the corresponding form mode

oe_no_button avoids displaying the navigation button in a *Many2one*

oe_avatar for image fields, displays images as "avatar" (square, 90x90 maximum size, some image decorations)

groups only displays the field for specific users

on_change calls the specified method when this field's value is edited, can generate update other fields or display warnings for the user

Deprecated since version 8.0: Use *openerp.api.onchange()* on the model

attrs dynamic meta-parameters based on record values

domain for relational fields only, filters to apply when displaying existing records for selection

context for relational fields only, context to pass when fetching possible values

readonly display the field in both readonly and edition mode, but never make it editable

required generates an error and prevents saving the record if the field doesn't have a value

nolabel don't automatically display the field's label, only makes sense if the field is a direct child of a `group` element

placeholder help message to display in *empty* fields. Can replace field labels in complex forms. *Should not* be an example of data as users are liable to confuse placeholder text with filled fields

mode for *One2many*, display mode (view type) to use for the field's linked records. One of `tree`, `form`, `kanban` or `graph`. The default is `tree` (a list display)

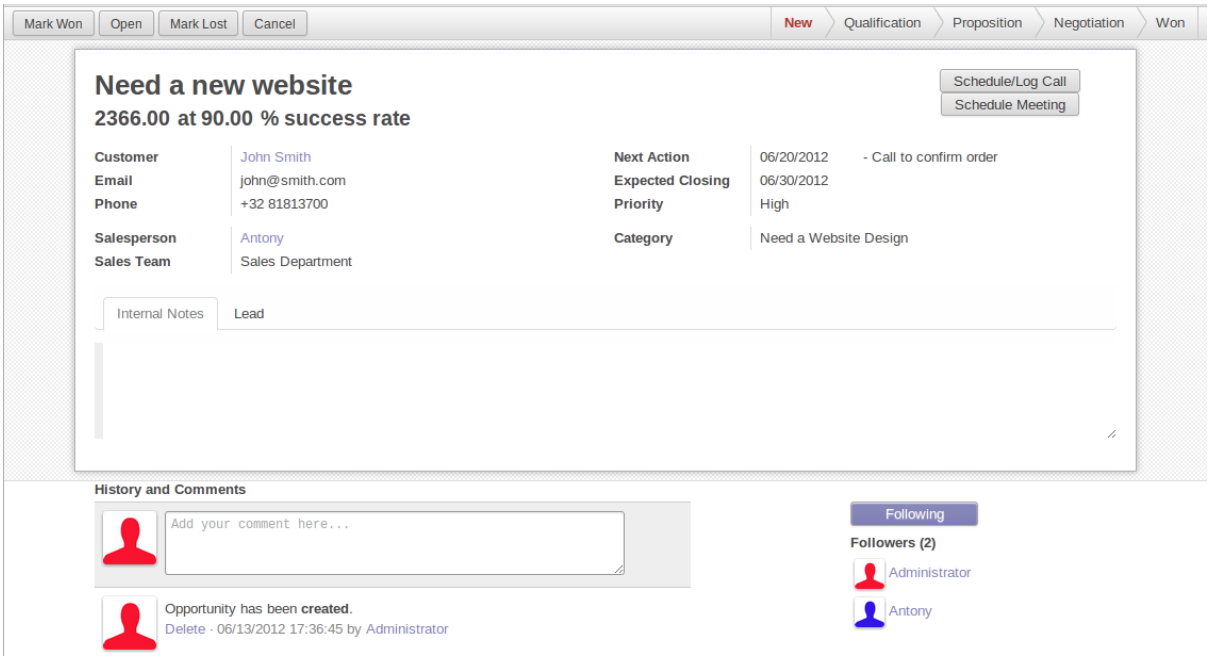
help tooltip displayed for users when hovering the field or its label

filename for binary fields, name of the related field providing the name of the file

password indicates that a *Char* field stores a password and that its data shouldn't be displayed

Business Views guidelines

Business views are targeted at regular users, not advanced users. Examples are: Opportunities, Products, Partners, Tasks, Projects, etc.



In general, a business view is composed of

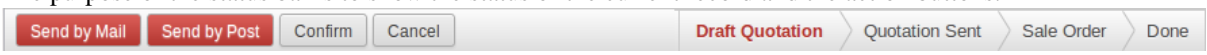
1. a status bar on top (with technical or business flow),
2. a sheet in the middle (the form itself),
3. a bottom part with History and Comments.

Technically, the new form views are structured as follows in XML:

```
<form>
    <header> ... content of the status bar  ... </header>
    <sheet>  ... content of the sheet       ... </sheet>
    <div class="oe_chatter"> ... content of the bottom part ... </div>
</form>
```

The Status Bar

The purpose of the status bar is to show the status of the current record and the action buttons.



The Buttons The order of buttons follows the business flow. For instance, in a sale order, the logical steps are:

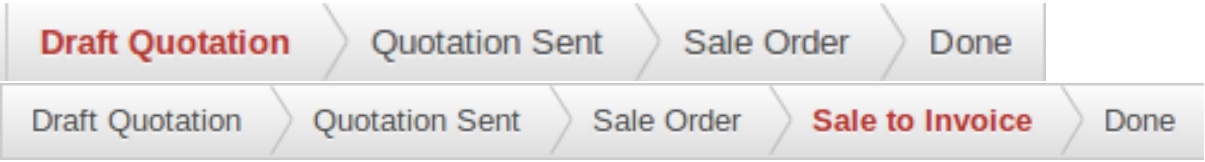
1. Send the quotation
2. Confirm the quotation
3. Create the final invoice
4. Send the goods

Highlighted buttons (in red by default) emphasize the logical next step, to help the user. It is usually the first active button. On the other hand, *cancel* buttons *must* remain grey (normal). For instance, in Invoice the button *Refund* must never be red.

Technically, buttons are highlighted by adding the class “oe_highlight”:

```
<button class="oe_highlight" name="..." type="..." states="..."/>
```

The Status Uses the `statusbar` widget, and shows the current state in red. States common to all flows (for instance, a sale order begins as a quotation, then we send it, then it becomes a full sale order, and finally it is done) should be visible at all times but exceptions or states depending on particular sub-flow should only be visible when current.



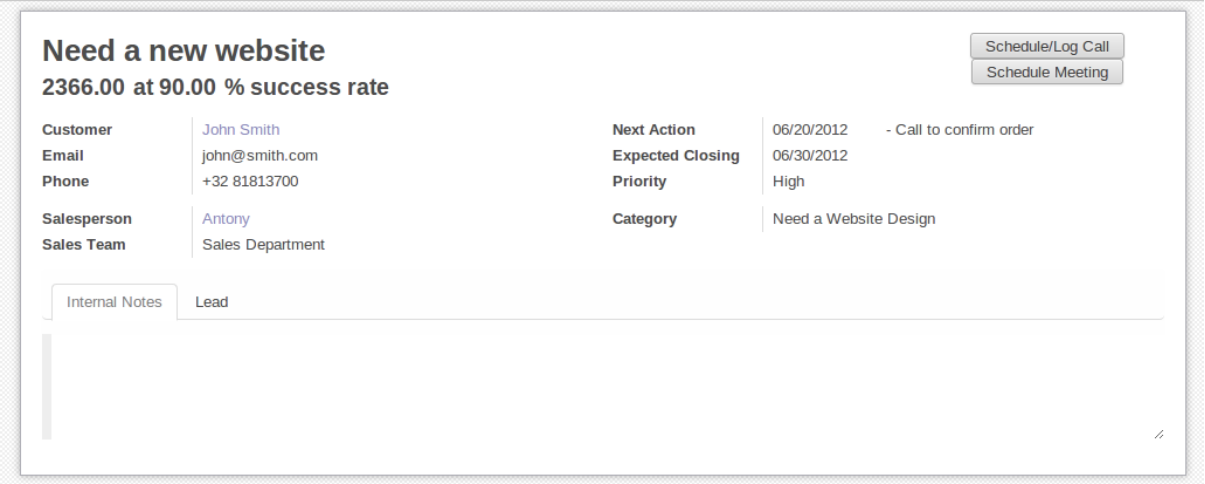
The states are shown following the order used in the field (the list in a selection field, etc). States that are always visible are specified with the attribute `statusbar_visible`.

`statusbar_colors` can be used to give a custom color to specific states.

```
<field name="state" widget="statusbar"
    statusbar_visible="draft,sent,progress,invoiced,done"
    statusbar_colors="{'shipping_except':'red','waiting_date':'blue'}"/>
```

The Sheet

All business views should look like a printed sheet:



1. Elements inside a `<form>` or `<page>` do not define groups, elements inside them are laid out according to normal HTML rules. They content can be explicitly grouped using `<group>` or regular `<div>` elements.
2. By default, the element `<group>` defines two columns inside, unless an attribute `col="n"` is used. The columns have the same width (1/n th of the group's width). Use a `<group>` element to produce a column of fields.
3. To give a title to a section, add a `string` attribute to a `<group>` element:

```
    <group string="Time-sensitive operations">
```

   this replaces the former use of `<separator string="XXX"/>`.

4. The `<field>` element does not produce a label, except as direct children of a `<group>` element[4]. Use `<label for="field_name>` to produce a label of a field.

Sheet Headers Some sheets have headers with one or more fields, and the labels of those fields are only shown in edit mode.

[4] for backwards compatibility reasons

View mode	Edit mode
Need a new website 2366.00 at 90.00 % success rate	Subject Need a new website Expected Revenue 2366.00 € at 90.00 % success rate

Use HTML text, `<div>`, `<h1>`, `<h2>`... to produce nice headers, and `<label>` with the class `oe_edit_only` to only display the field's label in edit mode. The class `oe_inline` will make fields inline (instead of blocks): content following the field will be displayed on the same line rather than on the line below it. The form above is produced by the following XML:

```
<label for="name" class="oe_edit_only"/>
<h1><field name="name"/></h1>

<label for="planned_revenue" class="oe_edit_only"/>
<h2>
    <field name="planned_revenue" class="oe_inline"/>
    <field name="company_currency" class="oe_inline oe_edit_only"/> at
    <field name="probability" class="oe_inline"/> % success rate
</h2>
```

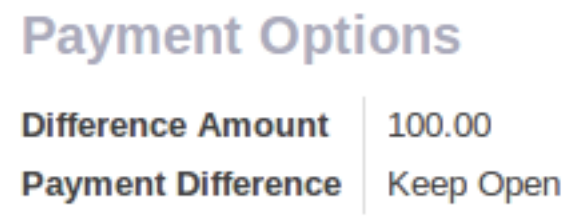
Button Box Many relevant actions or links can be displayed in the form. For example, in Opportunity form, the actions "Schedule a Call" and "Schedule a Meeting" have an important place in the use of the CRM. Instead of placing them in the "More" menu, put them directly in the sheet as buttons (on the top right) to make them more visible and more easily accessible.



Technically, the buttons are placed inside a `<div>` to group them as a block on the right-hand side of the sheet.

```
<div class="oe_button_box oe_right">
    <button string="Schedule/Log Call" name="..." type="action"/>
    <button string="Schedule Meeting" name="action_makeMeeting" type="object"/>
</div>
```

Groups and Titles A column of fields is now produced with a `<group>` element, with an optional title.

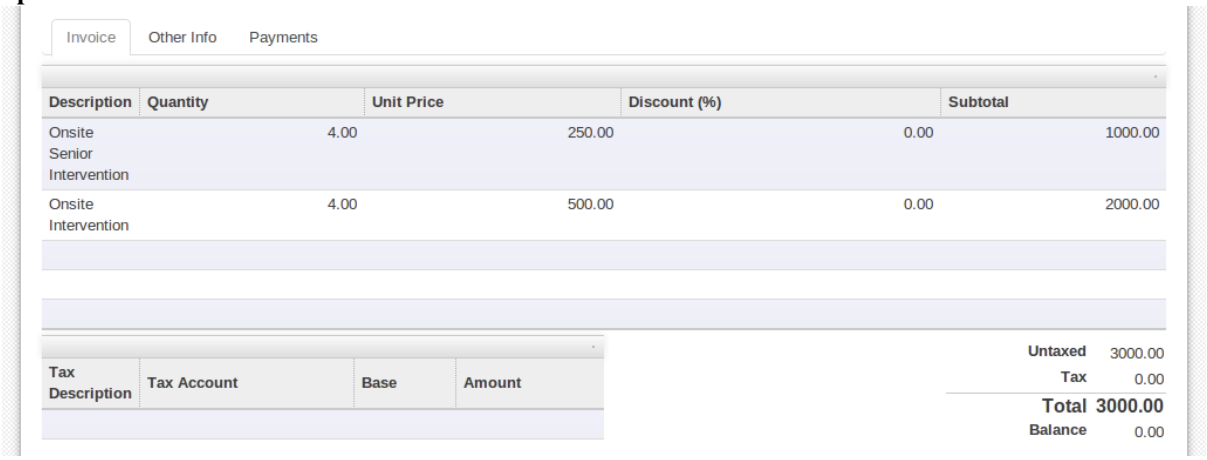



```
<group string="Payment Options">
    <field name="writeoff_amount"/>
    <field name="payment_option"/>
</group>
```

It is recommended to have two columns of fields on the form. For this, simply put the `<group>` elements that contain the fields inside a top-level `<group>` element.

To make *view extension* simpler, it is recommended to put a `name` attribute on `<group>` elements, so new fields can easily be added at the right place.

Special Case: Subtotals Some classes are defined to render subtotals like in invoice forms:

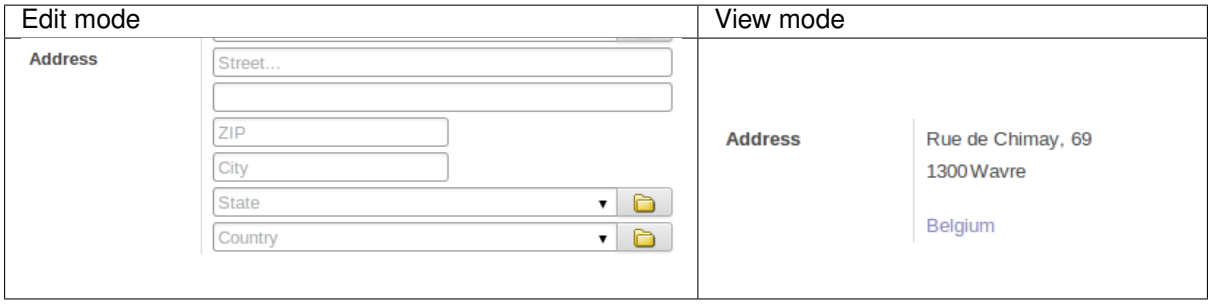



```
<group class="oe_subtotal_footer">
    <field name="amount_untaxed"/>
    <field name="amount_tax"/>
    <field name="amount_total" class="oe_subtotal_footer_separator"/>
    <field name="residual" style="margin-top: 10px"/>
</group>
```

Placeholders and Inline Fields Sometimes field labels make the form too complex. One can omit field labels, and instead put a placeholder inside the field. The placeholder text is visible only when the field is empty. The placeholder should tell what to place inside the field, it *must not* be an example as they are often confused with filled data.

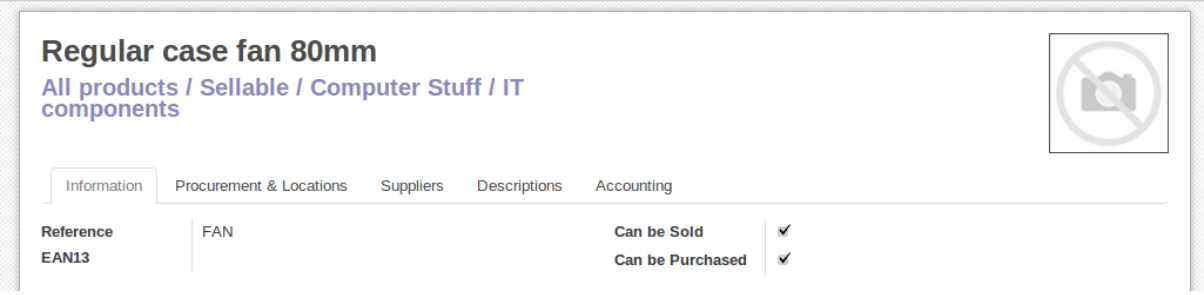
One can also group fields together by rendering them "inline" inside an explicit block element like `<div>`. This allows grouping semantically related fields as if they were a single (composite) fields.

The following example, taken from the *Leads* form, shows both placeholders and inline fields (zip and city).




```
<group>
    <label for="street" string="Address"/>
    <div>
        <field name="street" placeholder="Street..."/>
        <field name="street2"/>
        <div>
            <field name="zip" class="oe_inline" placeholder="ZIP"/>
            <field name="city" class="oe_inline" placeholder="City"/>
        </div>
        <field name="state_id" placeholder="State"/>
        <field name="country_id" placeholder="Country"/>
    </div>
</group>
```

Images Images, like avatars, should be displayed on the right of the sheet. The product form looks like:



The form above contains a <sheet> element that starts with:

```
<field name="product_image" widget="image" class="oe_avatar oe_right"/>
```

Tags Most *Many2many* fields, like categories, are better rendered as a list of tags. Use the widget `many2many_tags` for this:




```
<field name="category_id" widget="many2many_tags"/>
```

Configuration forms guidelines

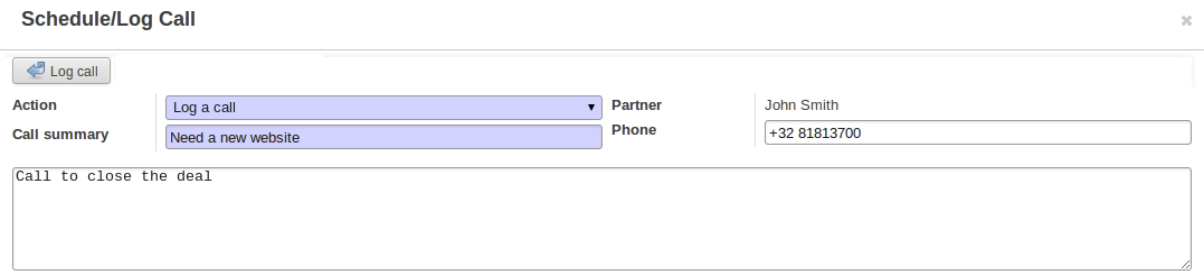
Examples of configuration forms: Stages, Leave Type, etc. This concerns all menu items under Configuration of each application (like Sales/Configuration).



1. no header (because no state, no workflow, no button)
2. no sheet

Dialog forms guidelines

Example: "Schedule a Call" from an opportunity.



1. avoid separators (the title is already in the popup title bar, so another separator is not relevant)

2. avoid cancel buttons (user generally close the popup window to get the same effect)
3. action buttons must be highlighted (red)
4. when there is a text area, use a placeholder instead of a label or a separator
5. like in regular form views, put buttons in the <header> element

Configuration Wizards guidelines

Example: Settings / Configuration / Sales.

1. always in line (no popup)
2. no sheet
3. keep the cancel button (users cannot close the window)
4. the button "Apply" must be red

4.4.5 Graphs

The graph view is used to visualize aggregations over a number of records or record groups. Its root element is `<graph>` which can take the following attributes:

`type` one of `bar` (default), `pie`, `line` and `pivot`, the type of graph to use (`pivot` technically isn't a graph type, it displays the aggregation as a pivot table)

`stacked` only used for `bar` charts. If present and set to `True`, stacks bars within a group

The only allowed element within a graph view is `field` which can have the following attributes:

`name` (required) the name of a field to use in a graph view. If used for grouping (rather than aggregating)

`type` indicates whether the field should be used as a grouping criteria or as an aggregated value within a group. Possible values are:

`row` (default) groups by the specified field. All graph types support at least one level of grouping, some may support more. For pivot tables, each group gets its own row.

`col` only used by pivot tables, creates column-wise groups

`measure` field to aggregate within a group

`interval` on date and datetime fields, groups by the specified interval (`day`, `week`, `month`, `quarter` or `year`) instead of grouping on the specific datetime (fixed second resolution) or date (fixed day resolution).

> **Warning:** graph view aggregations are performed on database content, non-stored function fields can not be used in graph views

4.4.6 Kanban

The kanban view is a kanban board visualisation: it displays records as "cards", halfway between a *list view* and a non-editable *form view*. Records may be grouped in columns for use in workflow visualisation or manipulation (e.g. tasks or work-progress management), or ungrouped (used simply to visualize records).

The root element of the Kanban view is `<kanban>`, it can use the following attributes:

`default_group_by` whether the kanban view should be grouped if no grouping is specified via the action or the current research. Should be the name of the field to group by when no grouping is otherwise specified

`default_order` cards sorting order used if the user has not already sorted the records (via the list view)

`class` adds HTML classes to the root HTML element of the Kanban view

quick_create whether it should be possible to create records without switching to the form view. By default, `quick_create` is enabled when the Kanban view is grouped, and disabled when not.

Set to `true` to always enable it, and to `false` to always disable it.

Possible children of the view element are:

field declares fields to aggregate or to use in kanban *logic*. If the field is simply displayed in the kanban view, it does not need to be pre-declared.

Possible attributes are:

name (required) the name of the field to fetch

sum, avg, min, max, count displays the corresponding aggregation at the top of a kanban column, the field's value is the label of the aggregation (a string). Only one aggregate operation per field is supported.

templates defines a list of *QWeb* templates. Cards definition may be split into multiple templates for clarity, but kanban views *must* define at least one root template `kanban-box`, which will be rendered once for each record.

The kanban view uses mostly-standard *javascript qweb* and provides the following context variables:

instance the current *Web Client* instance

widget the current *KanbanRecord()*, can be used to fetch some meta-information. These methods are also available directly in the template context and don't need to be accessed via `widget`

record an object with all the requested fields as its attributes. Each field has two attributes `value` and `raw_value`, the former is formatted according to current user parameters, the latter is the direct value from a *read()* (except for date and datetime fields that are formatted according to user's locale)

read_only_mode self-explanatory

buttons and fields

While most of the Kanban templates are standard *QWeb*, the Kanban view processes `field`, `button` and `a` elements specially:

- by default fields are replaced by their formatted value, unless they match specific kanban view widgets
- buttons and links with a `type` attribute become perform Odoo-related operations rather than their standard HTML function. Possible types are:

  action, object standard behavior for *Odoo buttons*, most attributes relevant to standard Odoo buttons can be used.

  open opens the card's record in the form view in read-only mode

  edit opens the card's record in the form view in editable mode

  delete deletes the card's record and removes the card

Javascript API

class KanbanRecord()

`Widget()` handling the rendering of a single record to a card. Available within its own rendering as `widget` in the template context.

kanban_color(*raw_value*)

Converts a color segmentation value to a kanban color class `oe_kanban_color_`*color_index*. The built-in CSS provides classes up to a `color_index` of 9.

kanban_getcolor (*raw_value*)
Converts a color segmentation value to a color index (between 0 and 9 by default). Color segmentation values can be either numbers or strings.

kanban_image (*model, field, id[, cache][, options]*)
Generates the URL to the specified field as an image access.

Arguments

- **model** (*String*) – model hosting the image
- **field** (*String*) – name of the field holding the image data
- **id** – identifier of the record contaning the image to display
- **cache** (*Number*) – caching duration (in seconds) of the browser default should be overridden. `0` disables caching entirely

Returns an image URL

kanban_text_ellipsis (*string*[, *size=160*])
clips text beyond the specified size and appends an ellipsis to it. Can be used to display the initial part of potentially very long fields (e.g. descriptions) without the risk of unwieldy cards

4.4.7 Calendar

Calendar views display records as events in a daily, weekly or monthly calendar. Their root element is `<calendar>`. Available attributes on the calendar view are:

date_start (required) name of the record's field holding the start date for the event

date_stop name of the record's field holding the end date for the event, if `date_stop` is provided records become movable (via drag and drop) directly in the calendar

date_delay alternative to `date_stop`, provides the duration of the event instead of its end date

color name of a record field to use for *color segmentation*. Records in the same color segment are allocated the same highlight color in the calendar, colors are allocated semi-randomly.

event_open_popup opens the event in a dialog instead of switching to the form view, enabled by default

quick_add enables quick-event creation on click: only asks the user for a `name` and tries to create a new event with just that and the clicked event time. Falls back to a full form dialog if the quick creation fails

display format string for event display, field names should be within brackets `[` and `]`

all_day name of a boolean field on the record indicating whether the corresponding event is flagged as day-long (and duration is irrelevant)

4.4.8 Gantt

Gantt views appropriately display Gantt charts (for scheduling).

The root element of gantt views is `<gantt/>`, it has no children but can take the following attributes:

date_start (required) name of the field providing the start datetime of the event for each record.

date_stop name of the field providing the end duration of the event for each record. Can be replaced by `date_delay`. One (and only one) of `date_stop` and `date_delay` must be provided.

If the field is `False` for a record, it's assumed to be a "point event" and the end date will be set to the start date

date_delay name of the field providing the duration of the event

progress name of a field providing the completion percentage for the record's event, between 0 and 100

default_group_by name of a field to group tasks by

4.4.9 Diagram

The diagram view can be used to display directed graphs of records. The root element is `<diagram>` and takes no attributes.

Possible children of the diagram view are:

`node` (required, 1) Defines the nodes of the graph. Its attributes are:

`object` the node's Odoo model

`shape` conditional shape mapping similar to colors and fonts in *the list view*. The only valid shape is `rectangle` (the default shape is an ellipsis)

`bgcolor` same as `shape`, but conditionally maps a background color for nodes. The default background color is white, the only valid alternative is `grey`.

`arrow` (required, 1) Defines the directed edges of the graph. Its attributes are:

`object` (required) the edge's Odoo model

`source` (required) *`Many2one`* field of the edge's model pointing to the edge's source node record

`destination` (required) *`Many2one`* field of the edge's model pointing to the edge's destination node record

`label` Python list of attributes (as quoted strings). The corresponding attributes's values will be concatenated and displayed as the edge's label

`label` Explanatory note for the diagram, the `string` attribute defines the note's content. Each `label` is output as a paragraph in the diagram header, easily visible but without any special emphasis.

4.4.10 Search

Search views are a break from previous view types in that they don't display *content*: although they apply to a specific model, they are used to filter other view's content (generally aggregated views e.g. *Lists* or *Graphs*). Beyond that difference in use case, they are defined the same way.

The root element of search views is `<search>`. It takes no attributes.

Possible children elements of the search view are:

`field` fields define domains or contexts with user-provided values. When search domains are generated, field domains are composed with one another and with filters using **AND**.

Fields can have the following attributes:

`name` the name of the field to filter on

`string` the field's label

`operator` by default, fields generate domains of the form `[(`*`name`*`,` *`operator`*`,` *`provided_value`*`)]` where `name` is the field's name and `provided_value` is the value provided by the user, possibly filtered or transformed (e.g. a user is expected to provide the *label* of a selection field's value, not the value itself).

The `operator` attribute allows overriding the default operator, which depends on the field's type (e.g. = for float fields but `ilike` for char fields)

`filter_domain` complete domain to use as the field's search domain, can use a `self` variable to inject the provided value in the custom domain. Can be used to generate significantly more flexible domains than `operator` alone (e.g. searches on multiple fields at once)

If both `operator` and `filter_domain` are provided, `filter_domain` takes precedence.

`context` allows adding context keys, including the user-provided value (which as for `domain` is available as a `self` variable). By default, fields don't generate domains.

Note: the domain and context are inclusive and both are generated if if a `context` is specified. To only generate context values, set `filter_domain` to an empty list: `filter_domain="[]"`

groups make the field only available to specific users

widget use specific search widget for the field (the only use case in standard Odoo 8.0 is a `selection` widget for *`Many2one`* fields)

domain if the field can provide an auto-completion (e.g. *`Many2one`*), filters the possible completion results.

filter a filter is a predefined toggle in the search view, it can only be enabled or disabled. Its main purposes are to add data to the search context (the context passed to the data view for searching/filtering), or to append new sections to the search filter.

Filters can have the following attributes:

string (required) the label of the filter

domain an Odoo *domain*, will be appended to the action's domain as part of the search domain

context a Python dictionary, merged into the action's domain to generate the search domain

name logical name for the filter, can be used to *enable it by default*, can also be used as *inheritance hook*

help a longer explanatory text for the filter, may be displayed as a tooltip

groups makes a filter only available to specific users

icon an icon to display next to the label, if there's sufficient space

Deprecated since version 7.0.

Tip: New in version 7.0.

Sequences of filters (without non-filters separating them) are treated as inclusively composited: they will be composed with `OR` rather than the usual `AND`, e.g.

```
<filter domain="[('state', '=', 'draft')]"/>
<filter domain="[('state', '=', 'done')]"/>
```

if both filters are selected, will select the records whose `state` is `draft` or `done`, but

```
<filter domain="[('state', '=', 'draft')]"/>
<separator/>
<filter domain="[('delay', '<', 15)]"/>
```

if both filters are selected, will select the records whose `state` is `draft` **and** `delay` is below 15.

separator can be used to separates groups of filters in simple search views

group can be used to separate groups of filters, more readable than `separator` in complex search views

Search defaults

Search fields and filters can be configured through the action's `context` using `search_default_`*`name`* keys. For fields, the value should be the value to set in the field, for filters it's a boolean value. For instance, assuming `foo` is a field and `bar` is a filter an action context of:

```
{
  'search_default_foo': 'acro',
  'search_default_bar': 1
}
```

will automatically enable the `bar` filter and search the `foo` field for *acro*.

4.4.11 QWeb

QWeb views are standard *QWeb* templates inside a view's `arch`. They don't have a specific root element.

A QWeb view can only contain a single template[5], and the template's name *must* match the view's complete (including module name) external id.

template should be used as a shortcut to define QWeb views.

4.5 Modules

4.5.1 Manifest

The manifest file serves to declare a python package as an Odoo module and to specify module metadata.

It is a file called `__openerp__.py` and contains a single Python dictionary, where each key specifies module metadatum.

```
{
    'name': "A Module",
    'version': '1.0',
    'depends': ['base'],
    'author': "Author Name",
    'category': 'Category',
    'description': """
    Description text
    """,
    # data files always loaded at installation
    'data': [
        'mymodule_view.xml',
    ],
    # data files containing optionally loaded demonstration data
    'demo': [
        'demo_data.xml',
    ],
}
```

Available manifest fields are:

`name` (`str`, required) the human-readable name of the module

`version` (`str`) this module's version, should follow semantic versioning rules

`description` (`str`) extended description for the module, in reStructuredText

`author` (`str`) name of the module author

`website` (`str`) website URL for the module author

`license` (`str`, defaults: `AGPL-3`) distribution license for the module

`category` (`str`, default: `Uncategorized`) classification category within Odoo, rough business domain for the module.

Although using existing categories is recommended, the field is freeform and unknown categories are created on-the-fly. Category hierarchies can be created using the separator `/` e.g. `Foo / Bar` will create a category `Foo`, a category `Bar` as child category of `Foo`, and will set `Bar` as the module's category.

`depends` (`list(str)`) Odoo modules which must be loaded before this one, either because this module uses features they create or because it alters resources they define.

When a module is installed, all of its dependencies are installed before it. Likewise dependencies are loaded before a module is loaded.

[5] or no template if it's an inherited view, then *it should only contain xpath elements*

data (list(str)) List of data files which must always be installed or updated with the module. A list of paths from the module root directory

demo (list(str)) List of data files which are only installed or updated in *demonstration mode*

auto_install (bool, default: False) If `True`, this module will automatically be installed if all of its dependencies are installed.

It is generally used for "link modules" implementing synergic integration between two otherwise independent modules.

For instance `sale_crm` depends on both `sale` and `crm` and is set to `auto_install`. When both `sale` and `crm` are installed, it automatically adds CRM campaigns tracking to sale orders without either `sale` or `crm` being aware of one another

4.6 Command-line interface: odoo.py

4.6.1 Running the server

-d <database>, **--database** <database>
database used when installing or updating modules.

-i <modules>, **--init** <modules>
comma-separated list of modules to install before running the server (requires *-d*).

-u <modules>, **--update** <modules>
comma-separated list of modules to update before running the server (requires *-d*).

--addons-path <directories>
comma-separated list of directories in which modules are stored. These directories are scanned for modules (nb: when and why?)

--workers <count>
if `count` is not 0 (the default), enables multiprocessing and sets up the specified number of HTTP workers (sub-processes processing HTTP and RPC requests).

Note: multiprocessing mode is only available on Unix-based systems

---

A number of options allow limiting and recyling workers:

--limit-request <limit>
Number of requests a worker will process before being recycled and restarted.

Defaults to 8196.

--limit-memory-soft <limit>
Maximum allowed virtual memory per worker. If the limit is exceeded, the worker is killed and recycled at the end of the current request.

Defaults to 640MB.

--limit-memory-hard <limit>
Hard limit on virtual memory, any worker exceeding the limit will be immediately killed without waiting for the end of the current request processing.

Defaults to 768MB.

--limit-time-cpu <limit>
Prevents the worker from using more than <limit> CPU seconds for each request. If the limit is exceeded, the worker is killed.

Defaults to 60.

--limit-time-real <limit>
Prevents the worker from taking longer than <limit> seconds to process a request. If the limit is exceeded, the worker is killed.

Differs from *--limit-time-cpu* in that this is a "wall time" limit including e.g. SQL queries.

Defaults to 120.

--max-cron-threads <count>
number of workers dedicated to cron jobs. Defaults to 2. The workers are threads in multithreading mode and processes in multiprocessing mode.

For multiprocessing mode, this is in addition to the HTTP worker processes.

-c <config>, **--config** <config>
provide an alternate configuration file

-s, --save
saves the server configuration to the current configuration file (*$HOME*/.openerp_serverrc by default, overridable using *-c*)

--proxy-mode
enables the use of X-Forwarded-* headers through Werkzeug's proxy support.

> **Warning:** proxy mode *must not* be enabled outside of a reverse proxy scenario

--test-enable
runs tests after installing modules

--debug
when an unexpected error is raised (not a warning or an access error), automatically starts python:pdb before logging and returning the error

database

-r <user>, **--db_user** <user>
database username, used to connect to PostgreSQL.

-w <password>, **--db_password** <password>
database password, if using password authentication.

--db_host <hostname>
host for the database server

- localhost on Windows
- UNIX socket otherwise

--db_port <port>
port the database listens on, defaults to 5432

--db-filter <filter>
hides databases that do not match <filter>. The filter is a regular expression, with the additions that:

- %h is replaced by the whole hostname the request is made on.
- %d is replaced by the subdomain the request is made on, with the exception of www (so domain odoo.com and www.odoo.com both match the database odoo)

--db-template <template>
when creating new databases from the database-management screens, use the specified template database. Defaults to template1.

built-in HTTP

--no-xmlrpc
do not start the HTTP or long-polling workers (may still start cron workers)

> **Warning:** has no effect if `--test-enable` is set, as tests require an accessible HTTP server

--xmlrpc-interface <interface>
TCP/IP address on which the HTTP server listens, defaults to `0.0.0.0` (all addresses)

--xmlrpc-port <port>
Port on which the HTTP server listens, defaults to 8069.

--longpolling-port <port>
TCP port for long-polling connections in multiprocessing or gevent mode, defaults to 8072. Not used in default (threaded) mode.

logging

By default, Odoo displays all logging of level `info` except for workflow logging (`warning` only), and log output is sent to `stdout`. Various options are available to redirect logging to other destinations and to customize the amout of logging output

--logfile <file>
sends logging output to the specified file instead of stdout. On Unix, the file can be managed by external log rotation programs and will automatically be reopened when replaced

--logrotate
enables log rotation daily, keeping 30 backups. Log rotation frequency and number of backups is not configurable.

--syslog
logs to the system's event logger: syslog on unices and the Event Log on Windows.

Neither is configurable

--log-db <dbname>
logs to the `ir.logging` model (`ir_logging` table) of the specified database. The database can be the name of a database in the "current" PostgreSQL, or a PostgreSQL URI for e.g. log aggregation

--log-handler <handler-spec>
LOGGER`:`*LEVEL*, enables `LOGGER` at the provided `LEVEL` e.g. `openerp.models:DEBUG` will enable all logging messages at or above `DEBUG` level in the models.

- The colon `:` is mandatory
- The logger can be omitted to configure the root (default) handler
- If the level is omitted, the logger is set to `INFO`

The option can be repeated to configure multiple loggers e.g.

```
$ odoo.py --log-handler :DEBUG --log-handler werkzeug:CRITICAL --log-handler openerp.fields:W
```

--log-request
enable DEBUG logging for RPC requests, equivalent to `--log-handler=openerp.http.rpc.request:DEBUG`

--log-response
enable DEBUG logging for RPC responses, equivalent to `--log-handler=openerp.http.rpc.response:DEBUG`

--log-web
enables DEBUG logging of HTTP requests and responses, equivalent to `--log-handler=openerp.http:DEBUG`

--log-sql
enables DEBUG logging of SQL querying, equivalent to `--log-handler=openerp.sql_db:DEBUG`

--log-level <level>
Shortcut to more easily set predefined levels on specific loggers. "real" levels (`critical`, `error`, `warn`, `debug`) are set on the `openerp` and `werkzeug` loggers (except for `debug` which is only set on `openerp`).

Odoo also provides debugging pseudo-levels which apply to different sets of loggers:

debug_sql sets the SQL logger to `debug`

equivalent to `--log-sql`

debug_rpc sets the `openerp` and HTTP request loggers to `debug`

equivalent to `--log-level debug --log-request`

debug_rpc_answer sets the `openerp` and HTTP request and response loggers to `debug`

equivalent to `--log-level debug --log-request --log-response`

Note: In case of conflict between *`--log-level`* and *`--log-handler`*, the latter is used

Advanced options

--auto-reload
enable auto-reloading of python files and xml files without having to restart the server. Requires pyinotify.

4.6.2 Scaffolding

Scaffolding is the automated creation of a skeleton structure to simplify bootstrapping (of new modules, in the case of Odoo). While not necessary it avoids the tedium of setting up basic structures and looking up what all starting requirements are.

Scaffolding is available via the **`odoo.py scaffold`** subcommand.

-t <template>
a template directory, files are passed through jinja2 then copied to the `destination` directory

name
the name of the module to create, may munged in various manners to generate programmatic names (e.g. module directory name, model names, ...)

destination
directory in which to create the new module, defaults to the current directory

4.6.3 Configuration file

Most of the command-line options can also be specified via a configuration file. Most of the time, they use similar names with the prefix - removed and other - are replaced by _ e.g. `--db-template` becomes `db_template`.

Some conversions don't match the pattern:

- `--db-filter` becomes `dbfilter`
- `--no-xmlrpc` corresponds to the `xmlrpc` boolean
- logging presets (all options starting with `--log-` except for `--log-handler` and `--log-db`) just add content to `log_handler`, use that directly in the configuration file
- `--smtp` is stored as `smtp_server`
- `--database` is stored as `dbname`

- `--debug` is stored as `debug_mode` (a boolean)
- `--i18n-import` and `--i18n-export` aren't available at all from configuration files

The default configuration file is *$HOME/*`.openerp_serverrc` which can be overridden using *`--config`*. Specifying *`--save`* will save the current configuration state back to that file.

4.7 Security in Odoo

Aside from manually managing access using custom code, Odoo provides two main data-driven mechanisms to manage or restrict access to data.

Both mechanisms are linked to specific users through *groups*: a user belongs to any number of groups, and security mechanisms are associated to groups, thus applying security mechamisms to users.

4.7.1 Access Control

Managed by the `ir.model.access` records, defines access to a whole model.

Each access control has a model to which it grants permissions, the permissions it grants and optionally a group.

Access controls are additive, for a given model a user has access all permissions granted to any of its groups: if the user belongs to one group which allows writing and another which allows deleting, they can both write and delete.

If no group is specified, the access control applies to all users, otherwise it only applies to the members of the given group.

Available permissions are creation (`perm_create`), searching and reading (`perm_read`), updating existing records (`perm_write`) and deleting existing records (`perm_unlink`)

4.7.2 Record Rules

Record rules are conditions that records must satisfy for an operation (create, read, update or delete) to be allowed. It is applied record-by-record after access control has been applied.

A record rule has:

- a model on which it applies
- a set of permissions to which it applies (e.g. if `perm_read` is set, the rule will only be checked when reading a record)
- a set of user groups to which the rule applies, if no group is specified the rule is *global*
- a *domain* used to check whether a given record matches the rule (and is accessible) or does not (and is not accessible). The domain is evaluated with two variables in context: `user` is the current user's record and `time` is the time module

Global rules and group rules (rules restricted to specific groups versus groups applying to all users) are used quite differently:

- Global rules are subtractive, they *must all* be matched for a record to be accessible
- Group rules are additive, if *any* of them matches (and all global rules match) then the record is accessible

This means the first *group rule* restricts access, but any further *group rule* expands it, while *global rules* can only ever restrict access (or have no effect).

> **Warning:** record rules do not apply to the Administrator user
> although access rules do

4.7.3 Field Access

New in version 7.0.

An ORM *Field* can have a `groups` attribute providing a list of groups (as a comma-separated string of external identifiers).

If the current user is not in one of the listed groups, he will not have access to the field:

- restricted fields are automatically removed from requested views
- restricted fields are removed from *fields_get()* responses
- attempts to (explicitly) read from or write to restricted fields results in an access error

4.7.4 Workflow transition rules

Workflow transitions can be restricted to a specific group. Users outside the group can not trigger the transition.

4.8 Testing Modules

Odoo provides support for testing modules using unittest2.

To write tests, simply define a `tests` sub-package in your module, it will be automatically inspected for test modules. Test modules should have a name starting with `test_` and should be imported from `tests/__init__.py`, e.g.

```
your_module
|-- ...
`-- tests
    |-- __init__.py
    |-- test_bar.py
    `-- test_foo.py
```

and `__init__.py` contains:

```
from . import test_foo, test_bar
```

Warning: test modules which are not imported from `tests/__init__.py` will not be run

Changed in version 8.0: previously, the test runner would only run modules added to two lists `fast_suite` and `checks` in `tests/__init__.py`. In 8.0 it will run all imported modules

The test runner will simply run any test case, as described in the official unittest documentation, but Odoo provides a number of utilities and helpers related to testing Odoo content (modules, mainly):

class `openerp.tests.common.`**`TransactionCase`**(*methodName='runTest'*)

TestCase in which each test method is run in its own transaction, and with its own cursor. The transaction is rolled back and the cursor is closed after each test.

`browse_ref`(*xid*)

Returns a record object for the provided external identifier

Parameters **`xid`** – fully-qualified external identifier, in the form *module.identifier*

Raise ValueError if not found

Returns `BaseModel`

`ref`(*xid*)

Returns database ID for the provided external identifier, shortcut for `get_object_reference`

Parameters **`xid`** – fully-qualified external identifier, in the form *module.identifier*

Raise ValueError if not found

Returns registered id

class openerp.tests.common.**SingleTransactionCase**(*methodName='runTest'*)
TestCase in which all test methods are run in the same transaction, the transaction is started with the first test method and rolled back at the end of the last.

browse_ref(*xid*)
Returns a record object for the provided external identifier

Parameters **xid** – fully-qualified external identifier, in the form *module.identifier*

Raise ValueError if not found

Returns BaseModel

ref(*xid*)
Returns database ID for the provided external identifier, shortcut for get_object_reference

Parameters **xid** – fully-qualified external identifier, in the form *module.identifier*

Raise ValueError if not found

Returns registered id

By default, tests are run once right after the corresponding module has been installed. Test cases can also be configured to run after all modules have been installed, and not run right after the module installation:

openerp.tests.common.**at_install**(*flag*)
Sets the at-install state of a test, the flag is a boolean specifying whether the test should (True) or should not (False) run during module installation.

By default, tests are run right after installing the module, before starting the installation of the next module.

openerp.tests.common.**post_install**(*flag*)
Sets the post-install state of a test. The flag is a boolean specifying whether the test should or should not run after a set of module installations.

By default, tests are *not* run after installation of all modules in the current installation set.

The most common situation is to use *TransactionCase* and test a property of a model in each method:

```
class TestModelA(common.TransactionCase):
    def test_some_action(self):
        record = self.env['model.a'].create({'field': 'value'})
        record.some_action()
        self.assertEqual(
            record.field,
            expected_field_value)

    # other tests...
```

4.8.1 Running tests

Tests are automatically run when installing or updating modules if *--test-enable* was enabled when starting the Odoo server.

As of Odoo 8, running tests outside of the install/update cycle is not supported.

4.9 Web Controllers

4.9.1 Routing

openerp.http.**route**(*route=None*, ***kw*)

Decorator marking the decorated method as being a handler for requests. The method must be part of a subclass of `Controller`.

Parameters

- **route** – string or array. The route part that will determine which http requests will match the decorated method. Can be a single string or an array of strings. See werkzeug's routing documentation for the format of route expression (http://werkzeug.pocoo.org/docs/routing/).
- **type** – The type of request, can be `'http'` or `'json'`.
- **auth** – The type of authentication method, can on of the following:
  - `user`: The user must be authenticated and the current request will perform using the rights of the user.
  - `public`: The user may or may not be authenticated. If she isn't, the current request will perform using the shared Public user.
  - `none`: The method is always active, even if there is no database. Mainly used by the framework and authentication modules. There request code will not have any facilities to access the database nor have any configuration indicating the current database nor the current user.
- **methods** – A sequence of http methods this route applies to. If not specified, all methods are allowed.
- **cors** – The Access-Control-Allow-Origin cors directive value.

4.9.2 Request

The request object is automatically set on `openerp.http.request` at the start of the request

class openerp.http.**WebRequest**(*httprequest*)

Parent class for all Odoo Web request types, mostly deals with initialization and setup of the request object (the dispatching itself has to be handled by the subclasses)

Parameters **httprequest** (`werkzeug.wrappers.BaseRequest`) – a wrapped werkzeug Request object

httprequest
the original `werkzeug.wrappers.Request` object provided to the request

params
`Mapping` of request parameters, not generally useful as they're provided directly to the handler method as keyword arguments

env
The `Environment` bound to current request. Raises a `RuntimeError` if the current requests is not bound to a database.

context
`Mapping` of context values for the current request

session
a `OpenERPSession` holding the HTTP session data for the current http session

cr
`Cursor` initialized for the current method call.

Accessing the cursor when the current request uses the `none` authentication will raise an exception.

debug
Indicates whether the current request is in "debug" mode

session_id
opaque identifier for the `OpenERPSession` instance of the current request

Deprecated since version 8.0: Use the `sid` attribute on *session*

registry
The registry to the database linked to this request. Can be `None` if the current request uses the `none` authentication.

Deprecated since version 8.0: use *env*

db
The database linked to this request. Can be `None` if the current request uses the `none` authentication.

httpsession
HTTP session data

Deprecated since version 8.0: Use *session* instead.

class `openerp.http.`**HttpRequest**(**args*)
Handler for the `http` request type.

matched routing parameters, query string parameters, form parameters and files are passed to the handler method as keyword arguments.

In case of name conflict, routing parameters have priority.

The handler method's result can be:

- a falsy value, in which case the HTTP response will be an HTTP 204 (No Content)
- a werkzeug Response object, which is returned as-is
- a `str` or `unicode`, will be wrapped in a Response object and interpreted as HTML

make_response(*data*, *headers=None*, *cookies=None*)
Helper for non-HTML responses, or HTML responses with custom response headers or cookies.

While handlers can just return the HTML markup of a page they want to send as a string if non-HTML data is returned they need to create a complete response object, or the returned data will not be correctly interpreted by the clients.

Parameters

- **data** (*basestring*) – response body
- **headers** (`[(name, value)]`) – HTTP headers to set on the response
- **cookies** (*collections.Mapping*) – cookies to set on the client

not_found(*description=None*)
Shortcut for a HTTP 404 (Not Found) response

render(*template*, *qcontext=None*, *lazy=True*, ***kw*)
Lazy render of a QWeb template.

The actual rendering of the given template will occur at then end of the dispatching. Meanwhile, the template and/or qcontext can be altered or even replaced by a static response.

Parameters

- **template** (*basestring*) – template to render
- **qcontext** (*dict*) – Rendering context to use

- **lazy** (*bool*) – whether the template rendering should be deferred until the last possible moment
- **kw** – forwarded to werkzeug's Response object

class openerp.http.**JsonRequest**(**args*)

Request handler for JSON-RPC 2 over HTTP

- `method` is ignored
- `params` must be a JSON object (not an array) and is passed as keyword arguments to the handler method
- the handler method's result is returned as JSON-RPC `result` and wrapped in the JSON-RPC Response

Sucessful request:

```
--> {"jsonrpc": "2.0",
     "method": "call",
     "params": {"context": {},
                "arg1": "val1" },
     "id": null}

<-- {"jsonrpc": "2.0",
     "result": { "res1": "val1" },
     "id": null}
```

Request producing a error:

```
--> {"jsonrpc": "2.0",
     "method": "call",
     "params": {"context": {},
                "arg1": "val1" },
     "id": null}

<-- {"jsonrpc": "2.0",
     "error": {"code": 1,
               "message": "End user error message.",
               "data": {"code": "codestring",
                        "debug": "traceback" } },
     "id": null}
```

4.9.3 Response

class openerp.http.**Response**(**args*, ***kw*)

Response object passed through controller route chain.

In addition to the `werkzeug.wrappers.Response` parameters, this class's constructor can take the following additional parameters for QWeb Lazy Rendering.

Parameters

- **template** (*basestring*) – template to render
- **qcontext** (*dict*) – Rendering context to use
- **uid** (*int*) – User id to use for the ir.ui.view render call, `None` to use the request's user (the default)

these attributes are available as parameters on the Response object and can be altered at any time before rendering

Also exposes all the attributes and methods of `werkzeug.wrappers.Response`.

render()

Renders the Response's template, returns the result

flatten()
Forces the rendering of the response's template, sets the result as response body and unsets `template`

4.9.4 Controllers

Controllers need to provide extensibility, much like *Model*, but can't use the same mechanism as the pre-requisites (a database with loaded modules) may not be available yet (e.g. no database created, or no database selected).

Controllers thus provide their own extension mechanism, separate from that of models:

Controllers are created by inheriting from

class openerp.http.**Controller**

and defining methods decorated with *route()*:

```
class MyController(openerp.http.Controller):
    @route('/some_url', auth='public')
    def handler(self):
        return stuff()
```

To *override* a controller, inherit from its class and override relevant methods, re-exposing them if necessary:

```
class Extension(MyController):
    @route()
    def handler(self):
        do_before()
        return super(Extension, self).handler()
```

- decorating with *route()* is necessary to keep the method (and route) visible: if the method is redefined without decorating, it will be "unpublished"
- the decorators of all methods are combined, if the overriding method's decorator has no argument all previous ones will be kept, any provided argument will override previously defined ones e.g.:

```
    class Restrict(MyController):
        @route(auth='user')
        def handler(self):
            return super(Restrict, self).handler()
```

will change `/some_url` from public authentication to user (requiring a log-in)

4.10 QWeb

QWeb is the primary templating engine used by Odoo[6]. It is an XML templating engine[7] and used mostly to generate HTML fragments and pages.

Template directives are specified as XML attributes prefixed with `t-`, for instance `t-if` for *conditionals*, with elements and other attributes being rendered directly.

To avoid element rendering, a placeholder element `<t>` is also available, which executes its directive but doesn't generate any output in and of itself:

```
<t t-if="condition">
    <p>Test</p>
</t>
```

will result in:

[6] although it uses a few others, either for historical reasons or because they remain better fits for the use case. Odoo 8.0 still depends on Jinja and Mako.

[7] it is similar in that to Genshi, although it does not use (and has no support for) XML namespaces

```
<p>Test</p>
```

if `condition` is true, but:

```
<div t-if="condition">
    <p>Test</p>
</div>
```

will result in:

```
<div>
    <p>Test</p>
</div>
```

4.10.1 data output

QWeb has a primary output directive which automatically HTML-escape its content limiting XSS risks when displaying user-provided content: `esc`.

`esc` takes an expression, evaluates it and prints the content:

```
<p><t t-esc="value"/></p>
```

rendered with the value `value` set to `42` yields:

```
<p>42</p>
```

There is one other output directive `raw` which behaves the same as respectively `esc` but *does not HTML-escape its output*. It can be useful to display separately constructed markup (e.g. from functions) or already sanitized user-provided markup.

4.10.2 conditionals

QWeb has a conditional directive `if`, which evaluates an expression given as attribute value:

```
<div>
    <t t-if="condition">
        <p>ok</p>
    </t>
</div>
```

The element is rendered if the condition is true:

```
<div>
    <p>ok</p>
</div>
```

but if the condition is false it is removed from the result:

```
<div>
</div>
```

The conditional rendering applies to the bearer of the directive, which does not have to be `<t>`:

```
<div>
    <p t-if="condition">ok</p>
</div>
```

will give the same results as the previous example.

4.10.3 loops

QWeb has an iteration directive `foreach` which take an expression returning the collection to iterate on, and a second parameter `t-as` providing the name to use for the "current item" of the iteration:

```
<t t-foreach="[1, 2, 3]" t-as="i">
    <p><t t-esc="i"/></p>
</t>
```

will be rendered as:

```
<p>1</p>
<p>2</p>
<p>3</p>
```

Like conditions, `foreach` applies to the element bearing the directive's attribute, and

```
<p t-foreach="[1, 2, 3]" t-as="i">
    <t t-esc="i"/>
</p>
```

is equivalent to the previous example.

`foreach` can iterate on an array (the current item will be the current value), a mapping (the current item will be the current key) or an integer (equivalent to iterating on an array between 0 inclusive and the provided integer exclusive).

In addition to the name passed via `t-as`, `foreach` provides a few other variables for various data points:

Warning: `$as` will be replaced by the name passed to `t-as`

***$as*_all** the object being iterated over

***$as*_value** the current iteration value, identical to `$as` for lists and integers, but for mappings it provides the value (where `$as` provides the key)

***$as*_index** the current iteration index (the first item of the iteration has index 0)

***$as*_size** the size of the collection if it is available

***$as*_first** whether the current item is the first of the iteration (equivalent to *$as*_index == 0)

***$as*_last** whether the current item is the last of the iteration (equivalent to *$as*_index + 1 == *$as*_size), requires the iteratee's size be available

***$as*_parity** either "even" or "odd", the parity of the current iteration round

***$as*_even** a boolean flag indicating that the current iteration round is on an even index

***$as*_odd** a boolean flag indicating that the current iteration round is on an odd index

These extra variables provided and all new variables created into the `foreach` are only available in the scope of the''foreach''. If the variable exists outside the context of the `foreach`, the value is copied at the end of the foreach into the global context.

```
<t t-set="existing_variable" t-value="False"/>
<!-- existing_variable now False -->

<p t-foreach="[1, 2, 3]" t-as="i">
    <t t-set="existing_variable" t-value="True"/>
    <t t-set="new_variable" t-value="True"/>
    <!-- existing_variable and new_variable now True -->
</p>

<!-- existing_variable always True -->
<!-- new_variable undefined -->
```

4.10.4 attributes

QWeb can compute attributes on-the-fly and set the result of the computation on the output node. This is done via the `t-att` (attribute) directive which exists in 3 different forms:

`t-att-$name` an attribute called `$name` is created, the attribute value is evaluated and the result is set as the attribute's value:

```
<div t-att-a="42"/>
```

will be rendered as:

```
<div a="42"></div>
```

`t-attf-$name` same as previous, but the parameter is a format string instead of just an expression, often useful to mix literal and non-literal string (e.g. classes):

```
<t t-foreach="[1, 2, 3]" t-as="item">
    <li t-attf-class="row {{ item_parity }}"><t t-esc="item"/></li>
</t>
```

will be rendered as:

```
<li class="row even">1</li>
<li class="row odd">2</li>
<li class="row even">3</li>
```

`t-att=mapping` if the parameter is a mapping, each (key, value) pair generates a new attribute and its value:

```
<div t-att="{'a': 1, 'b': 2}"/>
```

will be rendered as:

```
<div a="1" b="2"></div>
```

`t-att=pair` if the parameter is a pair (tuple or array of 2 element), the first item of the pair is the name of the attribute and the second item is the value:

```
<div t-att="['a', 'b']"/>
```

will be rendered as:

```
<div a="b"></div>
```

4.10.5 setting variables

QWeb allows creating variables from within the template, to memoize a computation (to use it multiple times), give a piece of data a clearer name, ...

This is done via the `set` directive, which takes the name of the variable to create. The value to set can be provided in two ways:

- a `t-value` attribute containing an expression, and the result of its evaluation will be set:

```
<t t-set="foo" t-value="2 + 1"/>
<t t-esc="foo"/>
```

will print `3`

- if there is no `t-value` attribute, the node's body is rendered and set as the variable's value:

```
<t t-set="foo">
    <li>ok</li>
</t>
<t t-esc="foo"/>
```

will generate `<li>ok</li>` (the content is escaped as we used the `esc` directive)

Note: using the result of this operation is a significant use-case for the `raw` directive.

4.10.6 calling sub-templates

QWeb templates can be used for top-level rendering, but they can also be used from within another template (to avoid duplication or give names to parts of templates) using the `t-call` directive:

```
<t t-call="other-template"/>
```

This calls the named template with the execution context of the parent, if `other_template` is defined as:

```
<p><t t-value="var"/></p>
```

the call above will be rendered as `<p/>` (no content), but:

```
<t t-set="var" t-value="1"/>
<t t-call="other-template"/>
```

will be rendered as `<p>1</p>`.

However this has the problem of being visible from outside the `t-call`. Alternatively, content set in the body of the `call` directive will be evaluated *before* calling the sub-template, and can alter a local context:

```
<t t-call="other-template">
    <t t-set="var" t-value="1"/>
</t>
<!-- "var" does not exist here -->
```

The body of the `call` directive can be arbitrarily complex (not just `set` directives), and its rendered form will be available within the called template as a magical `0` variable:

```
<div>
    This template was called with content:
    <t t-raw="0"/>
</div>
```

being called thus:

```
<t t-call="other-template">
    <em>content</em>
</t>
```

will result in:

```
<div>
    This template was called with content:
    <em>content</em>
</div>
```

4.10.7 Python

Exclusive directives

asset bundles

"smart records" fields formatting

The `t-field` directive can only be used when performing field access (`a.b`) on a "smart" record (result of the `browse` method). It is able to automatically format based on field type, and is integrated in the website's rich text

edition.

`t-field-options` can be used to customize fields, the most common option is `widget`, other options are field- or widget-dependent.

debugging

`t-debug` invokes a debugger using PDB's `set_trace` API. The parameter should be the name of a module, on which a `set_trace` method is called:

```
<t t-debug="pdb"/>
```

is equivalent to `importlib.import_module("pdb").set_trace()`

Helpers

Request-based

Most Python-side uses of QWeb are in controllers (and during HTTP requests), in which case templates stored in the database (as *views*) can be trivially rendered by calling *openerp.http.HttpRequest.render()*:

```
response = http.request.render('my-template', {
    'context_value': 42
})
```

This automatically creates a *Response* object which can be returned from the controller (or further customized to suit).

View-based

At a deeper level than the previous helper is the `render` method on `ir.ui.view`:

`render`(*cr, uid, id[, values][, engine='ir.qweb][, context]*)

Renders a QWeb view/template by database id or external id. Templates are automatically loaded from `ir.ui.view` records.

Sets up a number of default values in the rendering context:

`request` the current *WebRequest* object, if any

`debug` whether the current request (if any) is in `debug` mode

`quote_plus` url-encoding utility function

`json` the corresponding standard library module

`time` the corresponding standard library module

`datetime` the corresponding standard library module

relativedelta see module

`keep_query` the `keep_query` helper function

Parameters

- **`values`** – context values to pass to QWeb for rendering
- **`engine`** (*str*) – name of the Odoo model to use for rendering, can be used to expand or customize QWeb locally (by creating a "new" qweb based on `ir.qweb` with alterations)

API

It is also possible to use the `ir.qweb` model directly (and extend it, and inherit from it):

class `openerp.addons.base.ir.ir_qweb.`**QWeb**(*pool*, *cr*)
Base QWeb rendering engine

- to customize `t-field` rendering, subclass `ir.qweb.field` and create new models called `ir.qweb.field.`*`widget`*
- alternatively, override *`get_converter_for()`* and return an arbitrary model to use as field converter

Beware that if you need extensions or alterations which could be incompatible with other subsystems, you should create a local object inheriting from `ir.qweb` and customize that.

add_template(*qwebcontext*, *name*, *node*)
Add a parsed template in the context. Used to preprocess templates.

get_converter_for(*field_type*)
returns a *`Model`* used to render a `t-field`.

By default, tries to get the model named `ir.qweb.field.`*`field_type`*, falling back on `ir.qweb.field`.

Parameters **field_type** (*str*) – type or widget of field to render

get_template(*name*, *qwebcontext*)
Tries to fetch the template `name`, either gets it from the context's template cache or loads one with the context's loader (if any).

Raises **QWebTemplateNotFound** if the template can not be found or loaded

get_widget_for(*widget*)
returns a *`Model`* used to render a `t-esc`

Parameters **widget** (*str*) – name of the widget to use, or `None`

load_document(*document*, *res_id*, *qwebcontext*)
Loads an XML document and installs any contained template in the engine

prefixed_methods(*prefix*)
Extracts all methods prefixed by `prefix`, and returns a mapping of (t-name, method) where the t-name is the method name with prefix removed and underscore converted to dashes

Parameters **prefix** (*str*) –

Returns dict

render(*cr*, *uid*, *id_or_xml_id*, *qwebcontext=None*, *loader=None*, *context=None*)
Renders the template specified by the provided template name

Parameters

- **qwebcontext** (dict or `QWebContext` instance) – context for rendering the template
- **loader** – if `qwebcontext` is a dict, loader set into the context instantiated for rendering

render_tag_call_assets(*element*, *template_attributes*, *generated_attributes*, *qwebcontext*)
This special 't-call' tag can be used in order to aggregate/minify javascript and css assets

render_tag_field(*element*, *template_attributes*, *generated_attributes*, *qwebcontext*)
eg: +1 555 555 8069

class openerp.addons.base.ir.ir_qweb.**FieldConverter**(*pool*, *cr*)
Used to convert a t-field specification into an output HTML field.

to_html() is the entry point of this conversion from QWeb, it:

- converts the record value to html using *record_to_html()*
- generates the metadata attributes (data-oe-) to set on the root result node
- generates the root result node itself through *render_element()*

attributes(*cr*, *uid*, *field_name*, *record*, *options*, *source_element*, *g_att*, *t_att*, *qweb_context*, *context=None*)
Generates the metadata attributes (prefixed by data-oe- for the root node of the field conversion. Attribute values are escaped by the parent.

The default attributes are:

- model, the name of the record's model
- id the id of the record to which the field belongs
- field the name of the converted field
- type the logical field type (widget, may not match the field's type, may not be any Field subclass name)
- translate, a boolean flag (0 or 1) denoting whether the field is translatable
- expression, the original expression

Returns iterable of (attribute name, attribute value) pairs.

record_to_html(*cr*, *uid*, *field_name*, *record*, *options=None*, *context=None*)
Converts the specified field of the browse_record record to HTML

render_element(*cr*, *uid*, *source_element*, *t_att*, *g_att*, *qweb_context*, *content*)
Final rendering hook, by default just calls ir.qweb's render_element

to_html(*cr*, *uid*, *field_name*, *record*, *options*, *source_element*, *t_att*, *g_att*, *qweb_context*, *context=None*)
Converts a t-field to its HTML output. A t-field may be extended by a t-field-options, which is a JSON-serialized mapping of configuration values.

A default configuration key is widget which can override the field's own _type.

user_lang(*cr*, *uid*, *context*)
Fetches the res.lang object corresponding to the language code stored in the user's context. Fallbacks to en_US if no lang is present in the context *or the language code is not valid*.

Returns res.lang browse_record

value_to_html(*cr*, *uid*, *value*, *field*, *options=None*, *context=None*)
Converts a single value to its HTML version/output

4.10.8 Javascript

Exclusive directives

defining templates

The t-name directive can only be placed at the top-level of a template file (direct children to the document root):

```
<templates>
    <t t-name="template-name">
        <!-- template code -->
```

```
    </t>
</templates>
```

It takes no other parameter, but can be used with a `<t>` element or any other. With a `<t>` element, the `<t>` should have a single child.

The template name is an arbitrary string, although when multiple templates are related (e.g. called sub-templates) it is customary to use dot-separated names to indicate hierarchical relationships.

template inheritance

Template inheritance is used to alter existing templates in-place, e.g. to add information to templates created by an other modules.

Template inheritance is performed via the `t-extend` directive which takes the name of the template to alter as parameter.

The alteration is then performed with any number of `t-jquery` sub-directives:

```
<t t-extend="base.template">
    <t t-jquery="ul" t-operation="append">
        <li>new element</li>
    </t>
</t>
```

The `t-jquery` directives takes a CSS selector. This selector is used on the extended template to select *context nodes* to which the specified `t-operation` is applied:

append the node's body is appended at the end of the context node (after the context node's last child)

prepend the node's body is prepended to the context node (inserted before the context node's first child)

before the node's body is inserted right before the context node

after the node's body is inserted right after the context node

inner the node's body replaces the context node's children

replace the node's body is used to replace the context node itself

No operation if no `t-operation` is specified, the template body is interpreted as javascript code and executed with the context node as `this`

> **Warning:** while much more powerful than other operations, this mode is also much harder to debug and maintain, it is recommended to avoid it

debugging

The javascript QWeb implementation provides a few debugging hooks:

t-log takes an expression parameter, evaluates the expression during rendering and logs its result with `console.log`:

```
    <t t-set="foo" t-value="42"/>
    <t t-log="foo"/>
```

will print `42` to the console

t-debug triggers a debugger breakpoint during template rendering:

```
    <t t-if="a_test">
        <t t-debug="">
    </t>
```

will stop execution if debugging is active (exact condition depend on the browser and its development tools)

t-js the node's body is javascript code executed during template rendering. Takes a `context` parameter, which is the name under which the rendering context will be available in the `t-js`'s body:

```
<t t-set="foo" t-value="42"/>
<t t-js="ctx">
    console.log("Foo is", ctx.foo);
</t>
```

Helpers

openerp.**qweb**
An instance of *QWeb2.Engine()* with all module-defined template files loaded, and references to standard helper objects _ (underscore), _t (translation function) and JSON.

openerp.qweb.render can be used to easily render basic module templates

API

class QWeb2.**Engine**()
The QWeb "renderer", handles most of QWeb's logic (loading, parsing, compiling and rendering templates).

OpenERP Web instantiates one for the user, and sets it to `instance.web.qweb`. It also loads all the template files of the various modules into that QWeb instance.

A *QWeb2.Engine()* also serves as a "template namespace".

QWeb2.Engine.**render**(*template*[, *context*])
Renders a previously loaded template to a String, using `context` (if provided) to find the variables accessed during template rendering (e.g. strings to display).

Arguments

- **template** (*String*) – the name of the template to render
- **context** (*Object*) – the basic namespace to use for template rendering

Returns String

The engine exposes an other method which may be useful in some cases (e.g. if you need a separate template namespace with, in OpenERP Web, Kanban views get their own *QWeb2.Engine()* instance so their templates don't collide with more general "module" templates):

QWeb2.Engine.**add_template**(*templates*)
Loads a template file (a collection of templates) in the QWeb instance. The templates can be specified as:

An XML string QWeb will attempt to parse it to an XML document then load it.

A URL QWeb will attempt to download the URL content, then load the resulting XML string.

A `Document` or `Node` QWeb will traverse the first level of the document (the child nodes of the provided root) and load any named template or template override.

A *QWeb2.Engine()* also exposes various attributes for behavior customization:

QWeb2.Engine.**prefix**
Prefix used to recognize directives during parsing. A string. By default, `t`.

QWeb2.Engine.**debug**
Boolean flag putting the engine in "debug mode". Normally, QWeb intercepts any error raised during template execution. In debug mode, it leaves all exceptions go through without intercepting them.

`QWeb2.Engine.`**`jQuery`**
The jQuery instance used during template inheritance processing. Defaults to `window.jQuery`.

`QWeb2.Engine.`**`preprocess_node`**
A `Function`. If present, called before compiling each DOM node to template code. In OpenERP Web, this is used to automatically translate text content and some attributes in templates. Defaults to `null`.

4.11 Javascript

4.11.1 Widgets

class `openerp.`**`Widget`**`()`
The base class for all visual components. It corresponds to an MVC view, and provides a number of service to simplify handling of a section of a page:

- Handles parent/child relationships between widgets
- Provides extensive lifecycle management with safety features (e.g. automatically destroying children widgets during the destruction of a parent)
- Automatic rendering with *qweb*
- Backbone-compatible shortcuts

DOM Root

A *`Widget()`* is responsible for a section of the page materialized by the DOM root of the widget.

A widget's DOM root is available via two attributes:

`openerp.Widget.`**`el`**
raw DOM element set as root to the widget

`openerp.Widget.`**`$el`**
jQuery wrapper around *`el`*

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

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

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

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

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

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

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

The DOM root can also be defined programmatically by overridding

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

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

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

Using a widget

A widget's lifecycle has 3 main phases:

- creation and initialization of the widget instance

  openerp.Widget.**init**(*parent*)
  initialization method of widgets, synchronous, 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.

- DOM injection and startup, this is done by calling one of:

  openerp.Widget.**appendTo**(*element*)
  Renders the widget and inserts it as the last child of the target, uses .appendTo()

  openerp.Widget.**prependTo**(*element*)
  Renders the widget and inserts it as the first child of the target, uses .prependTo()

  openerp.Widget.**insertAfter**(*element*)
  Renders the widget and inserts it as the preceding sibling of the target, uses .insertAfter()

  openerp.Widget.**insertBefore**(*element*)
  Renders the widget and inserts it as the following sibling of the target, uses .insertBefore()

  All of these methods accept whatever the corresponding jQuery method accepts (CSS selectors, DOM nodes or jQuery objects). They all return a deferred 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

    openerp.Widget.**start**()
    asynchronous startup of the widget once it's been injected in the DOM, generally used to perform asynchronous RPC calls to fetch whatever remote data is necessary for the widget to do its work.

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

    A widget is *not guaranteed* to work correctly until its *start()* method has finished executing. The widget's parent/creator must wait for a widget to be fully started before interacting with it
    Returns deferred object

- widget destruction and cleanup

  openerp.Widget.**destroy**()
  destroys the widget's children, unbinds its events and removes its root from the DOM. Automatically called when the widget's parent is destroyed, must be called explicitly if the widget has no parents or if it is removed but its parent remains.

A widget being destroyed is automatically unlinked from its parent.

Related to widget destruction is an important utility method:

openerp.Widget.**alive**(*deferred*[, *reject=false*])
A significant issue with RPC and destruction is that an RPC call may take a long time to execute and return while a widget is being destroyed or after it has been destroyed, trying to execute its operations on a widget in a broken/invalid state.

This is a frequent source of errors or strange behaviors.

alive() can be used to wrap an RPC call, ensuring that whatever operations should be executed when the call ends are only executed if the widget is still alive:

```
this.alive(this.model.query().all()).then(function (records) {
    // would break if executed after the widget is destroyed, wrapping
    // rpc in alive() prevents execution
    _.each(records, function (record) {
        self.$el.append(self.format(record));
    });
});
```

Arguments

- **deferred** – a deferred object to wrap
- **reject** – by default, if the RPC call returns after the widget has been destroyed the returned deferred is left in limbo (neither resolved nor rejected). If `reject` is set to `true`, the deferred will be rejected instead.

Returns deferred object

openerp.Widget.**isDestroyed**()

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

Accessing DOM content

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

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

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

is functionally identical to:

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

Arguments

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

Returns jQuery object

Note: this helper method is similar to `Backbone.View.$`

Resetting the DOM root

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

Arguments

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

Note: should be mostly compatible with Backbone's setElement

DOM events handling

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

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

openerp.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[8]. If the selector is left out (only an event name is specified), the event will be set directly on the widget's DOM root.

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

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

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

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

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

Subclassing Widget

Widget() is subclassed in the standard manner (via the `extend()` method), and provides a number of abstract properties and concrete methods (which you may or may not want to override). Creating a subclass looks like this:

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

```
var MyWidget = openerp.Widget.extend({
    // QWeb template to use when rendering the object
    template: "MyQWebTemplate",
    events: {
        // events binding example
        'click .my-button': 'handle_click',
    },

    init: function(parent) {
        this._super(parent);
        // insert code to execute before rendering, for object
        // initialization
    },
    start: function() {
        var sup = this._super();
        // post-rendering initialization code, at this point

        // allows multiplexing deferred objects
        return $.when(
            // propagate asynchronous signal from parent class
            sup,
            // return own's asynchronous signal
            this.rpc(/* ... */))
    }
});
```

The new class can then be used in the following manner:

```
// Create the instance
var my_widget = new MyWidget(this);
// Render and insert into DOM
my_widget.appendTo(".some-div");
```

After these two lines have executed (and any promise returned by *appendTo()* has been resolved if needed), the widget is ready to be used.

Note: the insertion methods will start the widget themselves, and will return the result of *start()*.

If for some reason you do not want to call these methods, you will have to first call `render()` on the widget, then insert it into your DOM and start it.

If the widget is not needed anymore (because it's transient), simply terminate it:

```
my_widget.destroy();
```

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

Development 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`
- 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.

4.11.2 RPC

To display and interact with data, calls to the Odoo server are necessary. This is performed using RPC <REMOTE PROCEDURE CALL>.

Odoo Web provides two primary APIs to handle this: a low-level JSON-RPC based API communicating with the Python section of Odoo Web (and of your module, if you have a Python part) and a high-level API above that allowing your code to talk directly to high-level Odoo models.

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

High-level API: calling into Odoo models

Access to Odoo object methods (made available through XML-RPC from the server) is done via *openerp.Model()*. It maps onto the Odoo server objects via two primary methods, *call()* and *query()*.

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

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

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

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

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

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

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

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

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

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

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

Arguments

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

Return type Deferred<>

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

Arguments

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

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

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

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

Return type Deferred<Array<>>

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

Return type Deferred<Object | null>

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

Return type Deferred<Number>

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

Arguments

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

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

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

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

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

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

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

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

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

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

Aggregation (grouping)

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

Odoo Web eschews direct calls to read_group() in favor of calling a method of *Query()*, much in the way it is one in SQLAlchemy [9]:

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

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

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

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

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

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

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

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

class openerp.web.**QueryGroup**()

[9] with a small twist: sqlalchemy.orm.query.Query.group_by() is not terminal, it returns a query which can still be altered.

openerp.web.QueryGroup.**get**(*key*)
returns the group's attribute `key`. Known attributes are:

grouped_on which grouping field resulted from this group

value `grouped_on`'s value for this group

length the number of records in the group

aggregates a {field: value} mapping of aggregations for the group

openerp.web.QueryGroup.**query**([*fields...*])
equivalent to `openerp.web.Model.query()` but pre-filtered to only include the records within this group. Returns a *Query()* which can be further manipulated as usual.

openerp.web.QueryGroup.**subgroups**()
returns a deferred to an array of *QueryGroup()* below this one

Low-level API: RPC calls to Python side

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

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

This method simply takes an absolute path (the absolute URL of the JSON *route* to call) and a mapping of attributes to values (passed as keyword arguments to the Python method). This function fetches the return value of the Python methods, converted to JSON.

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

```
openerp.session.rpc('/web/dataset/resequence', {
    model: some_model,
    ids: array_of_ids,
    offset: 42
}).then(function (result) {
    // resequence didn't error out
}, function () {
    // an error occured during during call
});
```

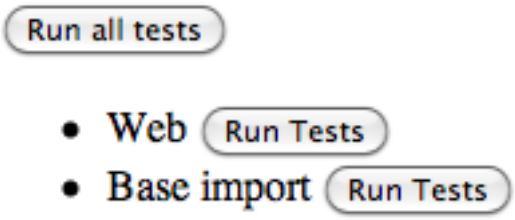
4.11.3 Web Client

4.11.4 Testing in Odoo Web Client

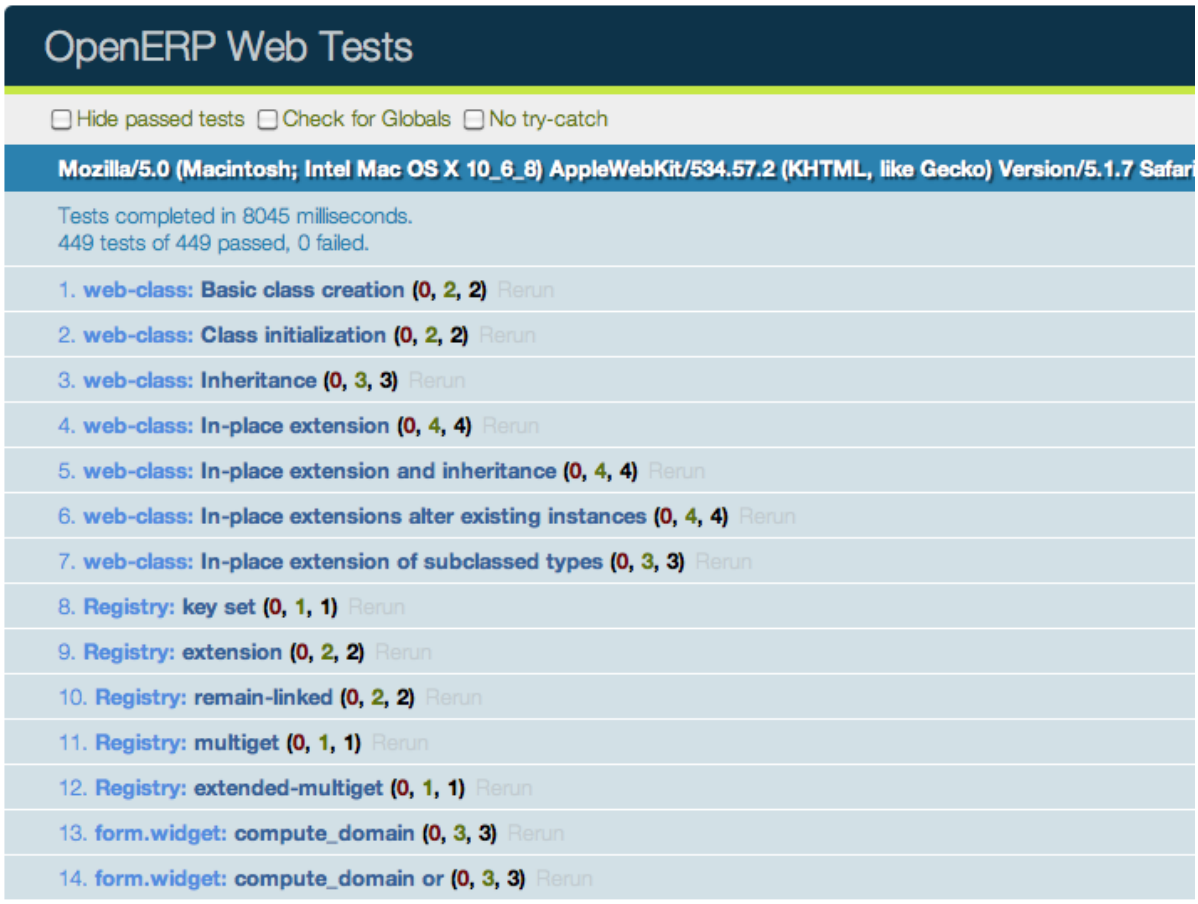
Javascript Unit Testing

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

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



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



Writing a test case

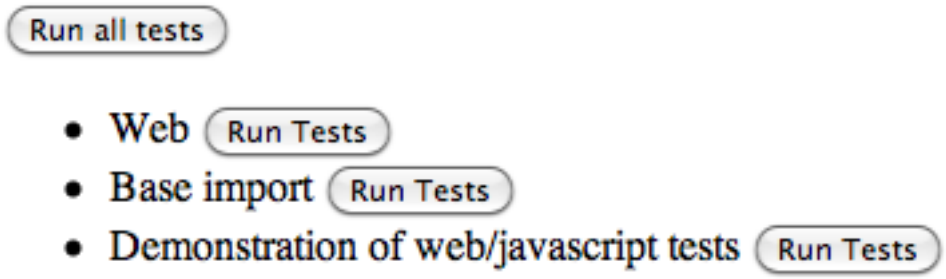
The first step is to list the test file(s). This is done through the `test` key of the Odoo manifest, by adding javascript files to it:

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

and to create the corresponding test file(s)

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

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



The next step is to create a test case:

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

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

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

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



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



Assertions

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

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

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

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

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

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

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

Arguments

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

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

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

Getting an Odoo instance

The Odoo instance is the base through which most Odoo Web modules behaviors (functions, objects, ...) are accessed. As a result, the test framework automatically builds one, and loads the module being tested and all of

its dependencies inside it. This new instance is provided as the first positional parameter to your test cases. Let's observe by adding javascript code (not test code) to the test module:

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

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

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

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

DOM Scratchpad

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

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

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

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

Loading templates

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

test case function.

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

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

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

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

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

Asynchronous cases

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

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

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

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

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

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

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

RPC

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

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

To enable mock RPC, set the *rpc option* to mock. This will add a third parameter to the test case callback:

mock (*rpc_spec*, *handler*)

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

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

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

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

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

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

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

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

    // widget needs that or it blows up
```

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

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

Testing API

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

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

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

Arguments

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

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

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

Arguments

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

class TestOptions()

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

TestOptions.**asserts**

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

TestOptions.**setup**

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

TestOptions.**teardown**

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

TestOptions.**fail_on_rejection**

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

```
        // test/demo.js
        test('unfail rejection', {
            asserts: 1,
            fail_on_rejection: false
        }, function () {
```

```
            var d = $.Deferred();
            setTimeout(function () {
                ok(true);
                d.reject();
            }, 100);
            return d;
        });
```

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

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

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

- `oe_all_dependencies` is an `Array` of all modules with a web component, ordered by dependency (for a module `A` with dependencies `A'`, any module of `A'` must come before `A` in the array)

Running through Python

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

1. Install unittest2 in your Python environment. Both can trivially be installed via pip or easy_install.
2. Install PhantomJS. It is a headless browser which allows automating running and testing web pages. QUnit-Suite uses it to actually run the qunit test suite.

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

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

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

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

   ---
3. Install a new database with all relevant modules (all modules with a web component at least), then restart the server

   ---

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

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

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

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

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

[snip ~150 lines]

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

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

OK
```

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

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

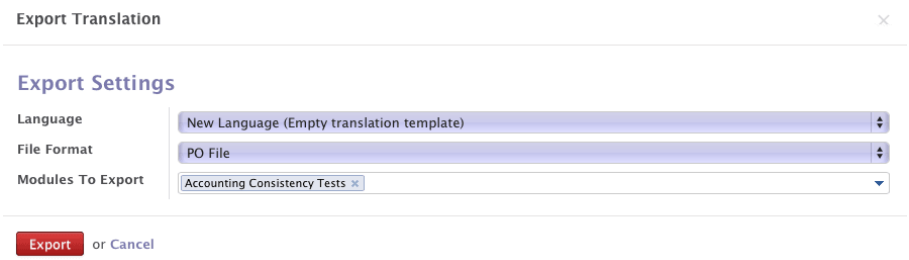
4.12 Translating Modules

4.12.1 Exporting translatable term

A number of terms in your modules are "implicitly translatable" as a result, even if you haven't done any specific work towards translation you can export your module's translatable terms and may find content to work with.

Translations export is performed via the administration interface by logging into the backend interface and opening *Settings* → *Translations* → *Import / Export* → *Export Translations*

- leave the language to the default (new language/empty template)
- select the PO File format
- select your module
- click *Export* and download the file



This gives you a file called `yourmodule.pot` which should be moved to the `yourmodule/i18n/` directory. The file is a *PO Template* which simply lists translatable strings and from which actual translations (PO files) can

be created. PO files can be created using msginit, with a dedicated translation tool like POEdit or by simply copying the template to a new file called *language*.po. Translation files should be put in *yourmodule*/i18n/, next to *yourmodule*.pot, and will be automatically loaded by Odoo when the corresponding language is installed (via *Settings* → *Translations* → *Load a Translation*)

Note: translations for all loaded languages are also installed or updated when installing or updating a module

4.12.2 Implicit exports

Odoo automatically exports translatable strings from "data"-type content:

- in non-QWeb views, all text nodes are exported as well as the content of the `string`, `help`, `sum`, `confirm` and `placeholder` attributes
- QWeb templates (both server-side and client-side), all text nodes are exported except inside `t-translation="off"` blocks, the content of the `title`, `alt`, `label` and `placeholder` attributes are also exported
- for *Field*, unless their model is marked with `_translate = False`:
  - their `string` and `help` attributes are exported
  - if `selection` is present and a list (or tuple), it's exported
  - if their `translate` attribute is set to `True`, all of their existing values (across all records) are exported
- help/error messages of *_constraints* and *_sql_constraints* are exported

4.12.3 Explicit exports

When it comes to more "imperative" situations in Python code or Javascript code, Odoo cannot automatically export translatable terms so they must be marked explicitly for export. This is done by wrapping a literal string in a function call.

In Python, the wrapping function is `openerp._()`:

```
title = _("Bank Accounts")
```

In JavaScript, the wrapping function is generally `openerp.web._t()`:

```
title = _t("Bank Accounts")
```

Warning: Only literal strings can be marked for exports, not expressions or variables. For situations where strings are formatted, this means the format string must be marked, not the formatted string:

```
# bad, the extract may work but it will not translate the text correctly
_("Scheduled meeting with %s" % invitee.name)

# good
_("Scheduled meeting with %s") % invitee.name
```

4.13 QWeb Reports

Reports are written in HTML/QWeb, like all regular views in Odoo. You can use the usual *QWeb control flow tools*. The PDF rendering itself is performed by wkhtmltopdf.

If you want to create a report on a certain model, you will need to define this *Report* and the *Report template* it will use. If you wish, you can also specify a specific *Paper Format* for this report. Finally, if you need access to

more than your model, you can define a *Custom Reports* class that gives you access to more models and records in the template.

4.13.1 Report

Every report must be declared by a *report action*.

For simplicity, a shortcut `<report>` element is available to define a report, rather than have to set up *the action* and its surroundings manually. That `<report>` can take the following attributes:

`id` the generated record's external id

`name` (mandatory) only useful as a mnemonic/description of the report when looking for one in a list of some sort

`model` (mandatory) the model your report will be about

`report_type` (mandatory) either `qweb-pdf` for PDF reports or `qweb-html` for HTML

`report_name` the name of your report (which will be the name of the PDF output)

`groups` *Many2many* field to the groups allowed to view/use the current report

`attachment_use` if set to True, the report will be stored as an attachment of the record using the name generated by the `attachment` expression; you can use this if you need your report to be generated only once (for legal reasons, for example)

`attachment` python expression that defines the name of the report; the record is acessible as the variable `object`

Warning: The paper format cannot currently be declared via the `<report>` shortcut, it must be added afterwards using a `<record>` extension on the report action itself:

```
<record id="<report_id>" model="ir.actions.report.xml">
    <field name="paperformat_id" ref="<paperformat>"/>
</record>
```

Example:

```
<report
    id="account_invoices"
    model="account.invoice"
    string="Invoices"
    report_type="qweb-pdf"
    name="account.report_invoice"
    file="account.report_invoice"
    attachment_use="True"
    attachment="(object.state in ('open','paid')) and
        ('INV'+(object.number or '').replace('/','')+'.pdf')"
/>
```

4.13.2 Report template

Minimal viable template

A minimal template would look like:

```
<template id="report_invoice">
    <t t-call="report.html_container">
        <t t-foreach="docs" t-as="o">
            <t t-call="report.external_layout">
                <div class="page">
```

```
                    <h2>Report title</h2>
                    <p>This object's name is <span t-field="o.name"/></p>
                </div>
            </t>
        </t>
    </t>
</template>
```

Calling `external_layout` will add the default header and footer on your report. The PDF body will be the content inside the `<div class="page">`. The template's `id` must be the name specified in the report declaration; for example `account.report_invoice` for the above report. Since this is a QWeb template, you can access all the fields of the `docs` objects received by the template.

There are some specific variables accessible in reports, mainly:

`docs` records for the current report

`doc_ids` list of ids for the `docs` records

`doc_model` model for the `docs` records

`time` a reference to `python:time` from the Python standard library

`translate_doc` a function to translate a part of a report. It must be used as follow:

```
    <t t-foreach="doc_ids" t-as="doc_id">
      <t t-raw="translate_doc(doc_id, doc_model, 'partner_id.lang', account.report_invoice_docume
    </t>
```

`user` `res.user` record for the user printing the report

`res_company` record for the current `user`'s company

If you wish to access other records/models in the template, you will need *a custom report*.

Translatable Templates

If you wish to translate reports (to the language of a partner, for example), you need to define two templates:

- The main report template
- The translatable document

You can then call translate_doc from your main template to obtain the translated document. If you wish to see the details of the translation in the backend, you can go to *Settings* → *Reports* → *Report* → *<report_name>* → *Search associated QWeb views* → *<translatable_document>* → *Associated translations*.

For example, let's look at the Sale Order report from the Sale module:

```
<!-- Main template -->
<template id="sale.report_saleorder">
    <t t-call="report.html_container">
        <t t-foreach="doc_ids" t-as="doc_id">
            <t t-raw="translate_doc(doc_id, doc_model, 'partner_id.lang', 'sale.report_saleorder_
        </t>
    </t>
</template>

<!-- Translatable template -->
<template id="report_saleorder_document">
    <t t-call="report.external_layout">
        <div class="page">
            <div class="oe_structure"/>
            <div class="row">
                <div class="col-xs-6">
                    <strong t-if="o.partner_shipping_id == o.partner_invoice_id">Invoice and shipp
```

```
                    <strong t-if="o.partner_shipping_id != o.partner_invoice_id">Invoice address:
                    <div t-field="o.partner_invoice_id" t-field-options="{&quot;no_marker&quot;: t
                <...>
            <div class="oe_structure"/>
        </div>
    </t>
</template>
```

The main template calls translate_doc with `partner_id.lang` as a parameter, which means it uses *a custom report model* to access a `res.partner` record.

Barcodes

Barcodes are images returned by a controller and can easily be embedded in reports thanks to the QWeb syntax:

```
<img t-att-src="'/report/barcode/QR/%s' % 'My text in qr code'"/>
```

More parameters can be passed as a query string

```
<img t-att-src="'/report/barcode/?
    type=%s&value=%s&width=%s&height=%s'%('QR', 'text', 200, 200)"/>
```

Useful Remarks

- Twitter Bootstrap and FontAwesome classes can be used in your report template
- Local CSS can be put directly in the template
- Global CSS can be inserted in the main report layout by inheriting its template and inserting your CSS:

```
    <template id="report_saleorder_style" inherit_id="report.style">
      <xpath expr=".">
        <t>
          .example-css-class {
            background-color: red;
          }
        </t>
      </xpath>
    </template>
```

- If it appears that your PDF report is missing the styles, please check *these instructions*.

4.13.3 Paper Format

Paper formats are records of `report.paperformat` and can contain the following attributes:

`name` (mandatory) only useful as a mnemonic/description of the report when looking for one in a list of some sort

`description` a small description of your format

`format` either a predefined format (A0 to A9, B0 to B10, Legal, Letter, Tabloid,...) or `custom`; A4 by default. You cannot use a non-custom format if you define the page dimensions.

`dpi` output DPI; 90 by default

`margin_top`, `margin_bottom`, `margin_left`, `margin_right` margin sizes in mm

`page_height`, `page_width` page dimensions in mm

`orientation` Landscape or Portrait

`header_line` boolean to display a header line

header_spacing header spacing in mm

Example:

```
<record id="paperformat_frenchcheck" model="report.paperformat">
    <field name="name">French Bank Check</field>
    <field name="default" eval="True"/>
    <field name="format">custom</field>
    <field name="page_height">80</field>
    <field name="page_width">175</field>
    <field name="orientation">Portrait</field>
    <field name="margin_top">3</field>
    <field name="margin_bottom">3</field>
    <field name="margin_left">3</field>
    <field name="margin_right">3</field>
    <field name="header_line" eval="False"/>
    <field name="header_spacing">3</field>
    <field name="dpi">80</field>
</record>
```

4.13.4 Custom Reports

The report model has a default `get_html` function that looks for a model named `report.module.report_name`. If it exists, it will use it to call the QWeb engine; otherwise a generic function will be used. If you wish to customize your reports by including more things in the template (like records of others models, for example), you can define this model, overwrite the function `render_html` and pass objects in the `docargs` dictionnary:

```
from openerp import api, models

class ParticularReport(models.AbstractModel):
    _name = 'report.module.report_name'
    @api.multi
    def render_html(self, data=None):
        report_obj = self.env['report']
        report = report_obj._get_report_from_name('module.report_name')
        docargs = {
            'doc_ids': self._ids,
            'doc_model': report.model,
            'docs': self,
        }
        return report_obj.render('module.report_name', docargs)
```

4.13.5 Reports are web pages

Reports are dynamically generated by the report module and can be accessed directly via URL:

For example, you can access a Sale Order report in html mode by going to http://<server-address>/report/html/sale.report_saleorder/38

Or you can access the pdf version at http://<server-address>/report/pdf/sale.report_saleorder/38

4.14 Workflows

In Odoo, a workflow is a technical artefact to manage a set of "things to do" associated to the records of a model. The workflow provides a higher-level way to organize tasks to perform with or on a record.

More specifically, a workflow is a directed graph where the nodes are called "activities" and the arcs are called "transitions".

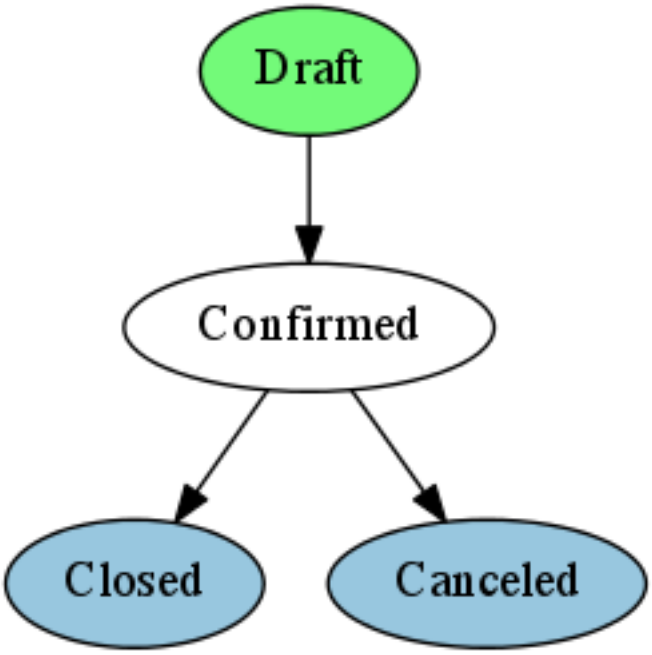
- Activities define work that should be done within the Odoo server, such as changing the state of some records, or sending emails.
- Transitions control how the workflow progresses from activity to activity.

In the definition of a workflow, one can attach conditions, signals, and triggers to transitions, so that the behavior of the workflow depends on user actions (such as clicking on a button), changes to records, or arbitrary Python code.

All in all, Odoo's workflow system provides:

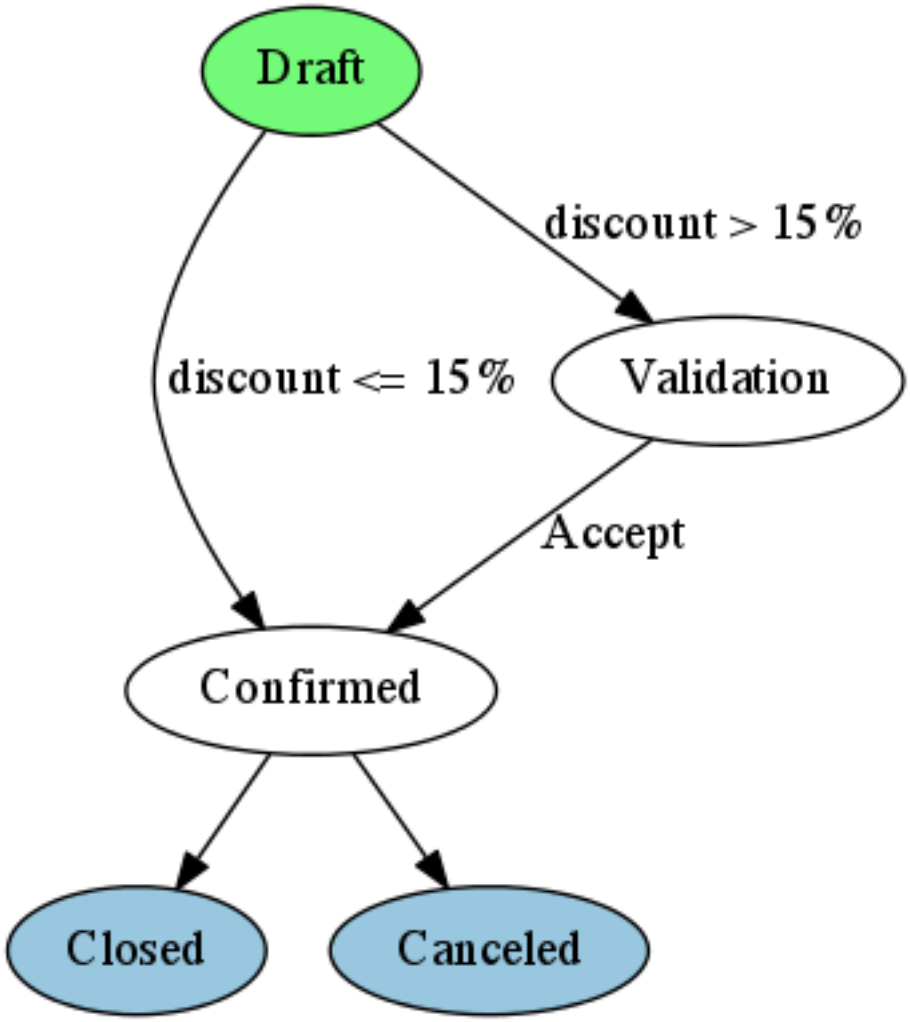
- a description of the evolution of a record (document) over time
- automatic actions based on various and flexible conditions
- management of company roles and validation steps
- management of interactions between objects
- a visual representation of document flows through their lifecycle

For instance, a basic order could have the following flow:



Orders start in the *Draft* state, can be *Confirmed* by a user, and then either shipped (*Closed*) or *Canceled*.

A company using Odoo may want to add discount support to orders, where sales staff has discretionary discounting powers up to 15%, but manager validation is required for discounts beyond 15%. The workflow can be altered online to add the relevant steps without editing Python or XML files:



Because Activities can perform arbitrary actions, the *Validation* can automatically send a validation request to the relevant employee.

Note: the order view needs to be modified to add an *Accept Discount* button for managers

4.14.1 Basics

Defining a workflow with data files is straightforward: a record "workflow" is given together with records for the activities and the transitions. For instance, here is a simple sequence of two activities defined in XML

```
<record id="test_workflow" model="workflow">
    <field name="name">test.workflow</field>
    <field name="osv">test.workflow.model</field>
    <field name="on_create">True</field>
</record>

<record id="activity_a" model="workflow.activity">
    <field name="wkf_id" ref="test_workflow"/>
    <field name="flow_start">True</field>
    <field name="name">a</field>
    <field name="kind">function</field>
    <field name="action">print_a()</field>
</record>
<record id="activity_b" model="workflow.activity">
    <field name="wkf_id" ref="test_workflow"/>
```

```
    <field name="flow_stop">True</field>
    <field name="name">b</field>
    <field name="kind">function</field>
    <field name="action">print_b()</field>
</record>

<record id="trans_a_b" model="workflow.transition">
    <field name="act_from" ref="activity_a"/>
    <field name="act_to" ref="activity_b"/>
</record>
```

A worfklow is always defined with respect to a particular model (the model is given by the attribute `osv` on the model `workflow`). Methods specified in the activities or transitions will be called on that model.

In the example code above, a workflow called "test_workflow" is created. It is made up of two activies, named "a" and "b", and one transition, going from "a" to "b".

The first activity has its attribute `flow_start` set to `True` so that Odoo knows where to start the workflow traversal after it is instanciated. Because `on_create` is set to True on the workflow record, the workflow is instanciated for each newly created record. (Otherwise, the workflow should be instanciated by other means, such as from some module Python code.)

When the workflow is instanciated, it begins with activity "a". That activity is of kind `function`, which means that the action `print_a()` is a method call on the model `test.workflow` (the usual `cr, uid, ids, context` arguments are passed for you).

The transition between "a" and "b" does not specify any condition. This means that the workflow instance immediately goes from "a" to "b" after "a" has been processed, and thus also processes activity "b".

4.14.2 Activities

While the transitions can be seen as the control structures of the workflows, activities are where everything happens, from changing record states to sending email.

Different kinds of activities exist: `Dummy`, `Function`, `Subflow`, and `Stop all`, each doing different things when the activity is processed. In addition to their kind, activies have other properties, detailed in the next sections.

Flow start and flow stop

The attribute `flow_start` is a boolean value specifying whether the activity is processed when the workflow is instanciated. Multiple activities can have their attribute `flow_start` set to `True`. When instanciating a workflow for a record, Odoo simply processes all of them, and evaluate all their outgoing transitions afterwards.

The attribute `flow_stop` is a boolean value specifying whether the activity stops the workflow instance. A workflow instance is considered completed when all its activities with the attribute `flow_stop` set to `True` are completed.

It is important for Odoo to know when a workflow instance is completed. A workflow can have an activity that is actually another workflow (called a subflow); that activity is completed when the subflow is completed.

Subflow

An activity can embed a complete workflow, called a subflow (the embedding workflow is called the parent workflow). The workflow to instanciate is specified by attribute `subflow_id`.

Note: In the GUI, that attribute can not be set unless the kind of the activity is `Subflow`.

The activity is considered completed (and its outgoing transitions ready to be evaluated) when the subflow is completed (see attribute `flow_stop` above).

Sending a signal from a subflow

When a workflow is embedded in an activity (as a subflow) of a workflow, the sublow can send a signal from its own activities to the parent workflow by giving a signal name in the attribute `signal_send`. Odoo processes those activities by sending the value of `signal_send` prefixed by "subflow." to the parent workflow instance.

In other words, it is possible to react and get transitions in the parent workflow as activities are executed in the sublow.

Server actions

An activity can run a "Server Action" by specifying its ID in the attribute `action_id`.

Python action

An activity can execute some Python code, given by the attribute `action`. The evaluation environment is the same as the one explained in the section *Conditions*.

Split mode

After an activity has been processed, Odoo evaluates its transition to reach the next activity in the flow.

However if an activity has more than one transition, Odoo must decide which activity or activities to follow.



This choice is controlled by the `split_mode` attribute:

`XOR` (default) By default, Odoo will use the first transition (in `sequence` order) whose condition is satisfied. All other transitions are ignored.

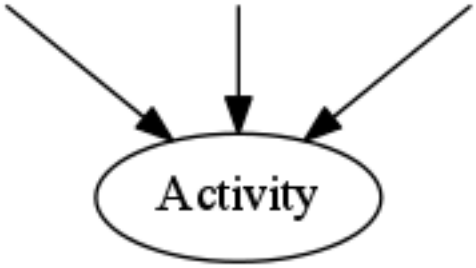
`OR` In `OR` mode, all transitions with a satisfied condition are traversed simultanously. Transitions not yet valid will be ignored, even if they become valid later.

`AND` In `AND` mode, Odoo will wait until *all* transitions are satisfied, and will traverse all of them (much like the `OR` mode).

Both `OR` and `AND` mode will lead to activities being active in the same workflow.

Join mode

Just like outgoing transition conditions can be combined together to decide whether they can be traversed or not, incoming transitions can be combined together to decide if and when an activity may be processed.



The `join_mode` attribute controls that behavior:

`XOR` (default) Any incoming transition enables the activity and starts its processing.

`AND` The activity is enabled and processed only once *all* incoming transitions have been traversed.

Kinds

An activity's kind defines the type of work an activity can perform.

Dummy (`dummy`, default) Do nothing at all, or call a server action. Often used as dispatch or gather "hubs" for transitions.

Function (`function`) Run some python code, execute a server action.

Stop all (`stopall`) Completely stops the workflow instance and marks it as completed.

Subflow (`subflow`) Starts executing an other workflow, once that workflow is completed the activity is done processing.

By default, the subflow is instanciated for the same record as the parent workflow. It is possible to change that behavior by providing Python code that returns a record ID (of the same data model as the subflow). The embedded subflow instance is then the one of the given record.

4.14.3 Transitions

Transitions provide the control structures to orchestrate a workflow. When an activity is completed, the workflow engine tries to get across transitions departing from the completed activity, towards the next activities. In their simplest form (as in the example above), they link activities sequentially: activities are processed as soon as the activities preceding them are completed.

Instead of running all activities in one fell swoop, it is also possible to wait on transitions, going through them only when some criteria are met. The criteria are the conditions, the signals, and the triggers. They are detailed in the following sections.

Conditions

When an activity has been completed, its outgoing transitions are inspected to determine whether it is possible for the workflow instance to proceed through them and reach the next activities. When only a condition is defined (i.e., no signal or trigger is defined), the condition is evaluated by Odoo, and if it evaluates to `True`, the workflow instance progresses through the transition. If the condition is not met, it will be reevaluated every time the associated record is modified, or by an explicit method call to do it.

By default, the attribute `condition` (i.e., the expression to be evaluated) is just "True", which trivially evaluates to `True`. Note that the condition may be several lines long; in that case, the value of the last one determines whether the transition can be taken.

In the condition evaluation environment, several symbols are conveniently defined (in addition to the Odoo `safe_eval` environment):

- all the model column names, and
- all the browse record's attributes.

Signals

In addition to a condition, a transition can specify a signal name. When such a signal name is present, the transition is not taken directly, even if the condition evaluates to `True`. Instead the transition blocks, waiting to be woken up.

In order to wake up a transition with a defined signal name, the signal must be sent to the workflow instance. A common way to send a signal is to use a button in the user interface, using the element `<button/>` with the signal name as the attribute `name` of the button. Once the button is clicked, the signal is sent to the workflow instance of the current record.

Note: The condition is still evaluated when the signal is sent to the workflow instance.

Triggers

With conditions that evaluate to `False`, transitions are not taken (and thus the activity it leads to is not processed immediately). Still, the workflow instance can get new chances to progress across that transition by providing so-called triggers. The idea is that when the condition is not satisfied, triggers are recorded in database. Later, it is possible to wake up specifically the workflow instances that installed those triggers, offering them to reevaluate their transition conditions. This mechanism makes it cheaper to wake up workflow instances by targetting just a few of them (those that have installed the triggers) instead of all of them.

Triggers are recorded in database as record IDs (together with the model name) and refer to the workflow instance waiting for those records. The transition definition provides a model name (attribute `trigger_model`) and a Python expression (attribute `trigger_expression`) that evaluates to a list of record IDs in the given model. Any of those records can wake up the workflow instance they are associated with.

Note: triggers are not re-installed whenever the transition is re-tried.

4.15 Odoo Guidelines

This page introduce the new Odoo Coding Guidelines. These guidelines aim to improve the quality of the code (better readability of source, ...) and Odoo Apps. Indeed, proper code ought ease maintenance, aid debugging, lower complexity and promote reliability.

4.15.1 Module structure

Directories

A module is organised in a few directory :

- *data/* : demo and data xml
- *models/* : models definition
- *controllers/* : contains controllers (http routes).
- *views/* : contains the views and templates
- *static/* : contains the web assets, separated into *css/, js/, img/, lib/, ...*

File naming

For *views* declarations, split backend views from (frontend) templates in 2 differents files.

For *models*, split the business logic by sets of models, in each set select a main model, this model gives its name to the set. If there is only one model, its name is the same as the module name. For each set named <main_model> the following files may be created:

- `models/<main_model>.py`
- `models/<inherited_main_model>.py`
- `views/<main_model>_templates.xml`
- `views/<main_model>_views.xml`

For instance, *sale* module introduces `sale_order` and `sale_order_line` where `sale_order` is dominant. So the `<main_model>` files will be named `models/sale_order.py` and `views/sale_order_views.py`.

For *data*, split them by purpose : demo or data. The filename will be the main_model name, suffixed by *_demo.xml* or *_data.xml*.

For *controller*, the only file should be named *main.py*.

For *static files*, the name pattern is *<module_name>.ext* (i.e. : static/js/im_chat.js, static/css/im_chat.css, static/xml/im_chat.xml, ...). Don't link data (image, libraries) outside Odoo : don't use an url to an image but copy it in our codebase instead.

The complete tree should look like

```
addons/<my_module_name>/
|-- __init__.py
|-- __openerp__.py
|-- controllers/
|   |-- __init__.py
|   `-- main.py
|-- data/
|   |-- <main_model>_data.xml
|   `-- <inherited_main_model>_demo.xml
|-- models/
|   |-- __init__.py
|   |-- <main_model>.py
|   `-- <inherited_main_model>.py
|-- security/
|   |-- ir.model.access.csv
|   `-- <main_model>_security.xml
|-- static/
|   |-- img/
|   |   |-- my_little_kitten.png
|   |   `-- troll.jpg
|   |-- lib/
|   |   `-- external_lib/
|   `-- src/
|       |-- js/
|       |   `-- <my_module_name>.js
|       |-- css/
|       |   `-- <my_module_name>.css
|       |-- less/
|       |   `-- <my_module_name>.less
|       `-- xml/
|           `-- <my_module_name>.xml
`-- views/
    |-- <main_model>_templates.xml
    |-- <main_model>_views.xml
```

```
    |-- <inherited_main_model>_templates.xml
    `-- <inherited_main_model>_views.xml
```

Note: File names should only contain `[a-z0-9_]` (lowercase alphanumerics and _)

Warning: Use correct file permissions : folder 755 and file 644.

4.15.2 XML files

Format

When declaring a record in XML,

- Place `id` attribute before `model`
- For field declaration, `name` attribute is first. Then place the *value* either in the `field` tag, either in the `eval` attribute, and finally other attributes (widget, options, ...) ordered by importance.
- Try to group the record by model. In case of dependencies between action/menu/views, the convention may not be applicable.
- Use naming convention defined at the next point
- The tag *<data>* is only used to set not-updatable data with `noupdate=1`

```
<record id="view_id" model="ir.ui.view">
    <field name="name">view.name</field>
    <field name="model">object_name</field>
    <field name="priority" eval="16"/>
    <field name="arch" type="xml">
        <tree>
            <field name="my_field_1"/>
            <field name="my_field_2" string="My Label" widget="statusbar" statusbar_visible="draft
        </tree>
    </field>
</record>
```

Naming xml_id

Security, View and Action

Use the following pattern :

- For a menu: *<model_name>*`_menu`
- For a view: *<model_name>*`_view_`*<view_type>*, where *view_type* is `kanban`, `form`, `tree`, `search`, ...
- For an action: the main action respects *<model_name>*`_action`. Others are suffixed with `_`*<detail>*, where *detail* is a lowercase string briefly explaining the action. This is used only if multiple actions are declared for the model.
- For a group: *<model_name>*`_group_`*<group_name>* where *group_name* is the name of the group, generally 'user', 'manager', ...
- For a rule: *<model_name>*`_rule_`*<concerned_group>* where *concerned_group* is the short name of the concerned group ('user' for the 'model_name_group_user', 'public' for public user, 'company' for multi-company rules, ...).

```
<!-- views and menus -->
<record id="model_name_menu" model="ir.ui.menu">
    ...
</record>

<record id="model_name_view_form" model="ir.ui.view">
    ...
</record>

<record id="model_name_view_kanban" model="ir.ui.view">
    ...
</record>

<!-- actions -->
<record id="model_name_action" model="ir.actions.act_window">
    ...
</record>

<record id="model_name_action_child_list" model="ir.actions.act_window">
    ...
</record>

<!-- security -->
<record id="model_name_group_user" model="res.groups">
    ...
</record>

<record id="model_name_rule_public" model="ir.rule">
    ...
</record>

<record id="model_name_rule_company" model="ir.rule">
    ...
</record>
```

Note: View names use dot notation `my.model.view_type` or `my.model.view_type.inherit` instead of *"This is the form view of My Model"*.

Inherited XML

The naming pattern of inherited view is *<base_view>*`_inherit_`*<current_module_name>*. A module may only extend a view once. Suffix the orginal name with `_inherit_`*<current_module_name>* where *current_module_name* is the technical name of the module extending the view.

```
<record id="inherited_model_view_form_inherit_my_module" model="ir.ui.view">
    ...
</record>
```

4.15.3 Python

PEP8 options

Using a linter can help show syntax and semantic warnings or errors. Odoo source code tries to respect Python standard, but some of them can be ignored.

- E501: line too long
- E301: expected 1 blank line, found 0
- E302: expected 2 blank lines, found 1

- E126: continuation line over-indented for hanging indent
- E123: closing bracket does not match indentation of opening bracket's line
- E127: continuation line over-indented for visual indent
- E128: continuation line under-indented for visual indent
- E265: block comment should start with '# '

Imports

The imports are ordered as

1. External libraries (one per line sorted and split in python stdlib)
2. Imports of `openerp`
3. Imports from Odoo modules (rarely, and only if necessary)

Inside these 3 groups, the imported lines are alphabetically sorted.

```
# 1 : imports of python lib
import base64
import re
import time
# 2 :  imports of openerp
import openerp
from openerp import api, fields, models # alphabetically ordered
from openerp.tools.safe_eval import safe_eval as eval
from openerp.tools.translate import _
# 3 :  imports from odoo modules
from openerp.addons.website.models.website import slug
from openerp.addons.web.controllers.main import login_redirect
```

Idioms

- Prefer `%` over `.format()`, prefer `%(varname)` instead of position (This is better for translation)
- Try to avoid generators and decorators
- Always favor *Readability* over *conciseness* or using the language features or idioms.
- Use list comprehension, dict comprehension, and basic manipulation using `map`, `filter`, `sum`, ... They make the code easier to read.
- The same applies for recordset methods : use `filtered`, `mapped`, `sorted`, ...
- Each python file should have `# -*- coding: utf-8 -*-` as first line
- Use the `UserError` defined in `openerp.exceptions` instead of overriding `Warning`, or find a more appropriate exception in *exceptions.py*
- Document your code (docstring on methods, simple comments for the tricky part of the code)
- Use meaningful variable/class/method names

Symbols

- Odoo Python Class : use camelcase for code in api v8, underscore lowercase notation for old api.

```
class AccountInvoice(models.Model):
    ...

class account_invoice(osv.osv):
    ...
```

- **Variable name :**
  - use camelcase for model variable
  - use underscore lowercase notation for common variable.
  - since new API works with record or recordset instead of id list, don't suffix variable name with *_id* or *_ids* if they not contain id or list of id.

```
ResPartner = self.env['res.partner']
partners = ResPartner.browse(ids)
partner_id = partners[0].id
```

- `One2Many` and `Many2Many` fields should always have *_ids* as suffix (example: sale_order_line_ids)
- `Many2One` fields should have *_id* as suffix (example : partner_id, user_id, ...)
- **Method conventions**
  - Compute Field : the compute method pattern is *_compute_<field_name>*
  - Search method : the search method pattern is *_search_<field_name>*
  - Default method : the default method pattern is *_default_<field_name>*
  - Onchange method : the onchange method pattern is *_onchange_<field_name>*
  - Constraint method : the constraint method pattern is *_check_<constraint_name>*
  - Action method : an object action method is prefix with *action_*. Its decorator is `@api.multi`, but since it use only one record, add `self.ensure_one()` at the beginning of the method.
- **In a Model attribute order should be**
  1. Private attributes (`_name`, `_description`, `_inherit`, ...)
  2. Default method and `_default_get`
  3. Field declarations
  4. Compute and search methods in the same order as field declaration
  5. Constrains methods (`@api.constrains`) and onchange methods (`@api.onchange`)
  6. CRUD methods (ORM overrides)
  7. Action methods
  8. And finally, other business methods.

```
class Event(models.Model):
    # Private attributes
    _name = 'event.event'
    _description = 'Event'

    # Default methods
    def _default_name(self):
        ...

    # Fields declaration
    name = fields.Char(string='Name', default=_default_name)
    seats_reserved = fields.Integer(oldname='register_current', string='Reserved Seats',
        store=True, readonly=True, compute='_compute_seats')
    seats_available = fields.Integer(oldname='register_avail', string='Available Seats',
        store=True, readonly=True, compute='_compute_seats')
    price = fields.Integer(string='Price')

    # compute and search fields, in the same order that fields declaration
    @api.multi
    @api.depends('seats_max', 'registration_ids.state', 'registration_ids.nb_register')
    def _compute_seats(self):
```

```
        ...

    # Constraints and onchanges
    @api.constrains('seats_max', 'seats_available')
    def _check_seats_limit(self):
        ...

    @api.onchange('date_begin')
    def _onchange_date_begin(self):
        ...

    # CRUD methods
    def create(self):
        ...

    # Action methods
    @api.multi
    def action_validate(self):
        self.ensure_one()
        ...

    # Business methods
    def mail_user_confirm(self):
        ...
```

4.15.4 Javascript and CSS

For javascript :

- `use strict;` is recommended for all javascript files
- Use a linter (jshint, ...)
- Never add minified Javascript Libraries
- Use camelcase for class declaration

For CSS :

- Prefix all your class with *o_<module_name>* where *module_name* is the technical name of the module ('sale', 'im_chat', ...) or the main route reserved by the module (for website module mainly, i.e. : 'o_forum' for website_forum module). The only exception for this rule is the webclient : it simply use *o_* prefix.
- Avoid using id
- Use bootstrap native class
- Use underscore lowercase notation to name class

4.15.5 Git

Commit message

Prefix your commit with

- **[IMP]** for improvements
- **[FIX]** for bug fixes
- **[REF]** for refactoring
- **[ADD]** for adding new resources
- **[REM]** for removing of resources

- **[MERGE]** for merge commits (only for forward/back-port)
- **[CLA]** for signing the Odoo Individual Contributor License

Then, in the message itself, specify the part of the code impacted by your changes (module name, lib, transversal object, ...) and a description of the changes.

- Always include a meaningful commit message: it should be self explanatory (long enough) including the name of the module that has been changed and the reason behind the change. Do not use single words like "bugfix" or "improvements".
- Avoid commits which simultaneously impact multiple modules. Try to split into different commits where impacted modules are different (It will be helpful if we need to revert a module separately).

```
[FIX] website, website_mail: remove unused alert div, fixes look of input-group-btn

Bootstrap's CSS depends on the input-group-btn
element being the first/last child of its parent.
This was not the case because of the invisible
and useless alert.

[IMP] fields: reduce memory footprint of list/set field attributes

[REF] web: add module system to the web client

This commit introduces a new module system for the javascript code.
Instead of using global ...
```

Note: Use the long description to explain the *why* not the *what*, the *what* can be seen in the diff

4.16 Upgrade API

4.16.1 Introduction

This document describes the API used to upgrade an Odoo database to a higher version.

It allows a database to be upgraded without ressorting to the html form at https://upgrade.odoo.com Although the database will follow the same process described on that form.

The required steps are:

- creating a request
- uploading a database dump
- running the upgrade process
- obtaining the status of the database request
- downloading the upgraded database dump

4.16.2 The methods

Creating a database upgrade request

This action creates a database request with the following information:

- your contract reference
- your email address
- the target version (the Odoo version you want to upgrade to)

- the purpose of your request (test or production)
- the database dump name (required but purely informative)
- optionally the server timezone (for Odoo source version < 6.1)

The `create` method

`https://upgrade.odoo.com/database/v1/create`
Creates a database upgrade request

Parameters

- **`contract`** (*str*) – (required) your enterprise contract reference
- **`email`** (*str*) – (required) your email address
- **`target`** (*str*) – (required) the Odoo version you want to upgrade to. Valid choices: 6.0, 6.1, 7.0, 8.0
- **`aim`** (*str*) – (required) the purpose of your upgrade database request. Valid choices: test, production.
- **`filename`** (*str*) – (required) a purely informative name for you database dump file
- **`timezone`** (*str*) – (optional) the timezone used by your server. Only for Odoo source version < 6.1

Returns request result

Return type json dictionary

The *create* method returns a json dictionary containing the following keys:

`failures` The list of errors.

A list of dictionaries, each dictionary giving information about one particular error. Each dictionary can contain various keys depending of the type of error but you will always get the *reason* and the *message* keys:

- `reason`: the error type
- `message`: a human friendly message

Some possible keys:

- `code`: a faulty value
- `value`: a faulty value
- `expected`: a list of valid values

See a sample output aside.

`request` If the *create* method is successful, the value associated to the *request* key will be a dictionary containing various information about the created request:

The most important keys are:

- `id`: the request id
- `key`: your private key for this request

These 2 values will be requested by the other methods (upload, process and status)

The other keys will be explained in the section describing the *status method*.

Sample script Here are 2 examples of database upgrade request creation using:

- one in the python programming language using the pycurl library
- one in the bash programming language using curl (tool for transfering data using http) and jq (JSON processor):

Uploading your database dump

This action upload your database dump.

The `upload` method

`https://upgrade.odoo.com/database/v1/upload`
Uploads a database dump

Parameters

- **`key`** (*str*) – (required) your private key
- **`request`** (*str*) – (required) your request id

Returns request result

Return type json dictionary

The request id and the private key are obtained using the *create method*

The result is a json dictionary containing the list of `failures`, which should be empty if everything went fine.

Asking to process your request

This action ask the Upgrade Platform to process your database dump.

The `process` method

`https://upgrade.odoo.com/database/v1/process`
Process a database dump

Parameters

- **`key`** (*str*) – (required) your private key
- **`request`** (*str*) – (required) your request id

Returns request result

Return type json dictionary

The request id and the private key are obtained using the *create method*

The result is a json dictionary containing the list of `failures`, which should be empty if everything went fine.

Obtaining your request status

This action ask the status of your database upgrade request.

The `status` method

`https://upgrade.odoo.com/database/v1/status`
Ask the status of a database upgrade request

Parameters

- **`key`** (*str*) – (required) your private key
- **`request`** (*str*) – (required) your request id

Returns request result

Return type json dictionary

The request id and the private key are obtained using the *create method*

The result is a json dictionary containing various information about your database upgrade request.

Sample output

The `request` key contains various useful information about your request:

`id` the request id

`key` your private key

`email` the email address you supplied when creating the request

`target` the target Odoo version you supplied when creating the request

`aim` the purpose (test, production) of your database upgrade request you supplied when creating the request

`filename` the filename you supplied when creating the request

`timezone` the timezone you supplied when creating the request

`state` the state of your request

`issue_stage` the stage of the issue we have create on Odoo main server

`issue` the id of the issue we have create on Odoo main server

`status_url` the URL to access your database upgrade request html page

`notes_url` the URL to get the notes about your database upgrade

`original_sql_url` the URL used to get your uploaded (not upgraded) database as an SQL stream

`original_dump_url` the URL used to get your uploaded (not upgraded) database as an archive file

`upgraded_sql_url` the URL used to get your upgraded database as an SQL stream

`upgraded_dump_url` the URL used to get your upgraded database as an archive file

`modules_url` the URL used to get your custom modules

`filesize` the size of your uploaded database file

`database_uuid` the Unique ID of your database

`created_at` the date when you created the request

`estimated_time` an estimation of the time it takes to upgrade your database

`processed_at` time when your database upgrade was started

`elapsed` the time it takes to upgrade your database

`filestore` your attachments were converted to the filestore

`customer_message` an important message related to your request

database_version the guessed Odoo version of your uploaded (not upgraded) database

postgresql the guessed Postgresql version of your uploaded (not upgraded) database

compressions the compression methods used by your uploaded database

CHAPTER

FIVE

BUSINESS MEMENTOES

O

openerp.addons.base.ir.ir_qweb, 173
openerp.api, 117