There are many ways to test an application. In Odoo, we have three kinds of tests

Python unit tests (see Testing Python code): useful for testing model business logic

JS unit tests (see Testing JS code): useful to test the javascript code in isolation

Tours (see Integration Testing): tours simulate a real situation. They ensures that the python and the javascript parts properly talk to each other.

Testing Python code

Odoo provides support for testing modules using unittest.
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

The test runner will simply run any test case, as described in the official unittest documentation (https://docs.python.org/3/library/unittest.html), but Odoo provides a number of utilities and helpers related to testing Odoo content (modules, mainly):

***class* odoo.tests.common.TransactionCase(*methodName='runTest'*)**
(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L584)

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*)_(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L338)

Returns a record object for the provided external identifier (../glossary.html#term-external-identifier)

Parameters:

xid – fully-qualified external identifier (../glossary.html#term-external-identifier), in the form ***module.identifier***

Raise:

ValueError if not found

Returns:

BaseModel (orm.html#odoo.models.BaseModel)

ref(*xid*)_(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L327)

Returns database ID for the provided external identifier (../glossary.html#term-external-identifier), shortcut for `get_object_reference`

Parameters:

xid – fully-qualified external identifier (../glossary.html#term-external-identifier), in the form ***module.identifier***

Raise:

ValueError if not found

Returns:

registered id

***class* odoo.tests.common.SingleTransactionCase(*methodName='runTest'*)**
(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L617)

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*)_(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L338)

Returns a record object for the provided external identifier (../glossary.html#term-external-identifier)

Parameters:

xid – fully-qualified external identifier (../glossary.html#term-external-identifier), in the form ***module.identifier***

Raise:

ValueError if not found

Returns:

BaseModel (orm.html#odoo.models.BaseModel)

ref(*xid*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L327)

Returns database ID for the provided external identifier (../glossary.html#term-external-identifier), shortcut for **get_object_reference**

Parameters:

xid – fully-qualified external identifier (../glossary.html#term-external-identifier), in the form ***module.identifier***

Raise:

ValueError if not found

Returns:

registered id

***class* odoo.tests.common.SavepointCase(*methodName='runTest'*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L642)**

Similar to **SingleTransactionCase** in that all test methods are run in a single transaction *but* each test case is run inside a rollbacked savepoint (sub-transaction).

Useful for test cases containing fast tests but with significant database setup common to all cases (complex in-db test data): **setUpClass()** can be used to generate db test data once, then all test cases use the same data without influencing one another but without having to recreate the test data either.

***class* odoo.tests.common.HttpCase(*methodName='runTest'*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L1442)**

Transactional HTTP TestCase with url_open and Chrome headless helpers.

browse_ref(*xid*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L338)

Returns a record object for the provided external identifier (../glossary.html#term-external-identifier)

Parameters:

xid – fully-qualified external identifier (../glossary.html#term-external-identifier), in the form ***module.identifier***

Raise:

ValueError if not found

Returns:

BaseModel (orm.html#odoo.models.BaseModel)

browser_js(*url_path*, *code*, *ready=''*, *login=None*, *timeout=60*, ***kw*)
(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L1365)

Test js code running in the browser - optionnally log as 'login' - load page given by url_path - wait for ready object to be available - eval(code) inside the page

To signal success test do: console.log('test successful') To signal test failure raise an exception or call console.error

ref(*xid*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L327)

Returns database ID for the provided external identifier (../glossary.html#term-external-identifier), shortcut for **get_object_reference**

Parameters:

xid – fully-qualified external identifier (../glossary.html#term-external-identifier), in the form ***module.identifier***

Raise:

ValueError if not found

Returns:

registered id

odoo.tests.common.tagged(tags*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2465)**

A decorator to tag BaseCase objects Tags are stored in a set that can be accessed from a 'test_tags' attribute A tag prefixed by '-' will remove the tag e.g. to remove the 'standard' tag By default, all Test classes from odoo.tests.common have a test_tags attribute that defaults to 'standard' and also the module technical name When using class inheritance, the tags are NOT inherited.

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:

```
# coding: utf-8
from odoo.tests import HttpCase, tagged

# This test should only be executed after all modules have been installed.
@tagged('-at_install', 'post_install')
class WebsiteVisitorTests(HttpCase):
  def test_create_visitor_on_tracked_page(self):
      Page = self.env['website.page']
```

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...
```

> Test methods must start with **test_**

class **odoo.tests.common.Form(*recordp*, *view=None*)** **(https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L1534)**

Server-side form view implementation (partial)

Implements much of the "form view" manipulation flow, such that server-side tests can more properly reflect the behaviour which would be observed when manipulating the interface:

- call default_get and the relevant onchanges on "creation"
- call the relevant onchanges on setting fields
- properly handle defaults & onchanges around x2many fields

Saving the form returns the created record if in creation mode.

Regular fields can just be assigned directly to the form, for **Many2one** (orm.html#odoo.fields.Many2one) fields assign a singleton recordset:

```
# empty recordset => creation mode
f = Form(self.env['sale.order'])
f.partner_id = a_partner
so = f.save()
```

When editing a record, using the form as a context manager to automatically save it at the end of the scope:

```
with Form(so) as f2:
    f2.payment_term_id = env.ref('account.account_payment_term_15days')
    # f2 is saved here
```

For **Many2many** (orm.html#odoo.fields.Many2many) fields, the field itself is a **M2MProxy** and can be altered by adding or removing records:

```
with Form(user) as u:
    u.groups_id.add(env.ref('account.group_account_manager'))
    u.groups_id.remove(id=env.ref('base.group_portal').id)
```

Finally **One2many** (orm.html#odoo.fields.One2many) are reified as **O2MProxy** .

Because the **One2many** (orm.html#odoo.fields.One2many) only exists through its parent, it is manipulated more directly by creating "sub-forms" with the **new()** and **edit()** methods. These would normally be used as context managers since they get saved in the parent record:

```
with Form(so) as f3:
    # add support
    with f3.order_line.new() as line:
        line.product_id = env.ref('product.product_product_2')
    # add a computer
    with f3.order_line.new() as line:
        line.product_id = env.ref('product.product_product_3')
    # we actually want 5 computers
    with f3.order_line.edit(1) as line:
        line.product_uom_qty = 5
    # remove support
    f3.order_line.remove(index=0)
    # SO is saved here
```

Parameters:

recordp (**odoo.models.Model** (orm.html#odoo.models.Model)) – empty or singleton recordset. An empty recordset will put the view in “creation” mode and trigger calls to default_get and on-load onchanges, a singleton will put it in “edit” mode and only load the view’s data.

view (**int | str | odoo.model.Model**) – the id, xmlid or actual view object to use for onchanges and view constraints. If none is provided, simply loads the default view for the model.

New in version 12.0.

save() (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L1866)

Saves the form, returns the created record if applicable

does not save **readonly** fields

does not save unmodified fields (during edition) — any assignment or onchange return marks the field as modified, even if set to its current value

Raises:

AssertionError (https://docs.python.org/3/library/exceptions.html#AssertionError) – if the form has any unfilled required field

class **odoo.tests.common.M2MProxy** (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2342)

Behaves as a **Sequence** of recordsets, can be indexed or sliced to get actual underlying recordsets.

add(*record*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2371)

Adds **record** to the field, the record must already exist.

The addition will only be finalized when the parent record is saved.

clear() (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2408)

Removes all existing records in the m2m

remove(*id=None*, *index=None*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2391)

Removes a record at a certain index or with a provided id from the field.

class **odoo.tests.common.O2MProxy** (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2240)

edit(*index*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2307)

Returns a **Form** to edit the pre-existing **One2many** (orm.html#odoo.fields.One2many) record.

The form is created from the list view if editable, or the field’s form view otherwise.

Raises:

AssertionError (https://docs.python.org/3/library/exceptions.html#AssertionError) – if the field is not editable

new() (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2295)

Returns a **Form** for a new **One2many** (orm.html#odoo.fields.One2many) record, properly initialised.

The form is created from the list view if editable, or the field’s form view otherwise.

Raises:

AssertionError (https://docs.python.org/3/library/exceptions.html#AssertionError) – if the field is not editable

remove(*index*) (https://github.com/odoo/odoo/blob/14.0/odoo/tests/common.py#L2319)

Removes the record at **index** from the parent form.

Raises:

AssertionError (https://docs.python.org/3/library/exceptions.html#AssertionError) – if the field is not editable

Running tests

Tests are automatically run when installing or updating modules if **--test-enable** (cmdline.html#cmdoption-odoo-bin-test-enable) was enabled when starting the Odoo server.

Test selection

In Odoo, Python tests can be tagged to facilitate the test selection when running tests.

Subclasses of `odoo.tests.common.BaseCase` (usually through `TransactionCase`, `SavepointCase` or `HttpCase`) are automatically tagged with `standard`, `at_install` and their source module's name by default.

Invocation

`--test-tags` (cmdline.html#cmdoption-odoo-bin-test-tags) can be used to select/filter tests to run on the command-line.

This option defaults to `+standard` meaning tests tagged `standard` (explicitly or implicitly) will be run by default when starting Odoo with `--test-enable` (cmdline.html#cmdoption-odoo-bin-test-enable).

When writing tests, the `tagged()` decorator can be used on **test classes** to add or remove tags.

The decorator's arguments are tag names, as strings.

> **⚠ Danger**
> `tagged()` is a class decorator, it has no effect on functions or methods

Tags can be prefixed with the minus (`-`) sign, to *remove* them instead of add or select them e.g. if you don't want your test to be executed by default you can remove the `standard` tag:

```
from odoo.tests import TransactionCase, tagged

@tagged('-standard', 'nice')
class NiceTest(TransactionCase):
    ...
```

This test will not be selected by default, to run it the relevant tag will have to be selected explicitely:

```
$ odoo-bin --test-enable --test-tags nice
```

Note that only the tests tagged `nice` are going to be executed. To run *both* `nice` and `standard` tests, provide multiple values to `--test-tags` (cmdline.html#cmdoption-odoo-bin-test-tags): on the command-line, values are *additive* (you're selecting all tests with *any* of the specified tags)

```
$ odoo-bin --test-enable --test-tags nice,standard
```

The config switch parameter also accepts the `+` and `-` prefixes. The `+` prefix is implied and therefore, totaly optional. The `-` (minus) prefix is made to deselect tests tagged with the prefixed tags, even if they are selected by other specified tags e.g. if there are `standard` tests which are also tagged as `slow` you can run all standard tests *except* the slow ones:

```
$ odoo-bin --test-enable --test-tags 'standard,-slow'
```

When you write a test that does not inherit from the `BaseCase`, this test will not have the default tags, you have to add them explicitely to have the test included in the default test suite. This is a common issue when using a simple `unittest.TestCase` as they're not going to get run:

```
import unittest
from odoo.tests import tagged

@tagged('standard', 'at_install')
class SmallTest(unittest.TestCase):
    ...
```

Special tags

`standard`: All Odoo tests that inherit from `BaseCase` are implicitely tagged standard. `--test-tags` (cmdline.html#cmdoption-odoo-bin-test-tags) also defaults to `standard`.

That means untagged test will be executed by default when tests are enabled.

`at_install`: Means that the test will be executed right after the module installation and before other modules are installed. This is a default implicit tag.

`post_install`: Means that the test will be executed after all the modules are installed. This is what you want for HttpCase tests most of the time.

Note that this is *not exclusive* with `at_install`, however since you will generally not want both `post_install` is usually paired with `-at_install` when tagging a test class.

module_name: Odoo tests classes extending `BaseCase` are implicitely tagged with the technical name of their module. This allows easily selecting or excluding specific modules when testing e.g. if you want to only run tests from `stock_account`:

```
$ odoo-bin --test-enable --test-tags stock_account
```

Examples

> **⚠ Important**
> Tests will be executed only in the installed or updated modules. So modules have to be selected with the **`-u`** **(cmdline.html#cmdoption-odoo-bin-u)** or **`-i`** **(cmdline.html#cmdoption-odoo-bin-i)** switches. For simplicity, those switches are not specified in the examples below.

Run only the tests from the sale module:

```
$ odoo-bin --test-enable --test-tags sale
```

Run the tests from the sale module but not the ones tagged as slow:

```
$ odoo-bin --test-enable --test-tags 'sale,-slow'
```

Run only the tests from stock or tagged as slow:

```
$ odoo-bin --test-enable --test-tags '-standard, slow, stock'
```

> **`-standard`** is implicit (not required), and present for clarity

Testing JS code

Testing a complex system is an important safeguard to prevent regressions and to guarantee that some basic functionality still works. Since Odoo has a non trivial codebase in Javascript, it is necessary to test it. In this section, we will discuss the practice of testing JS code in isolation: these tests stay in the browser, and are not supposed to reach the server.

Qunit test suite

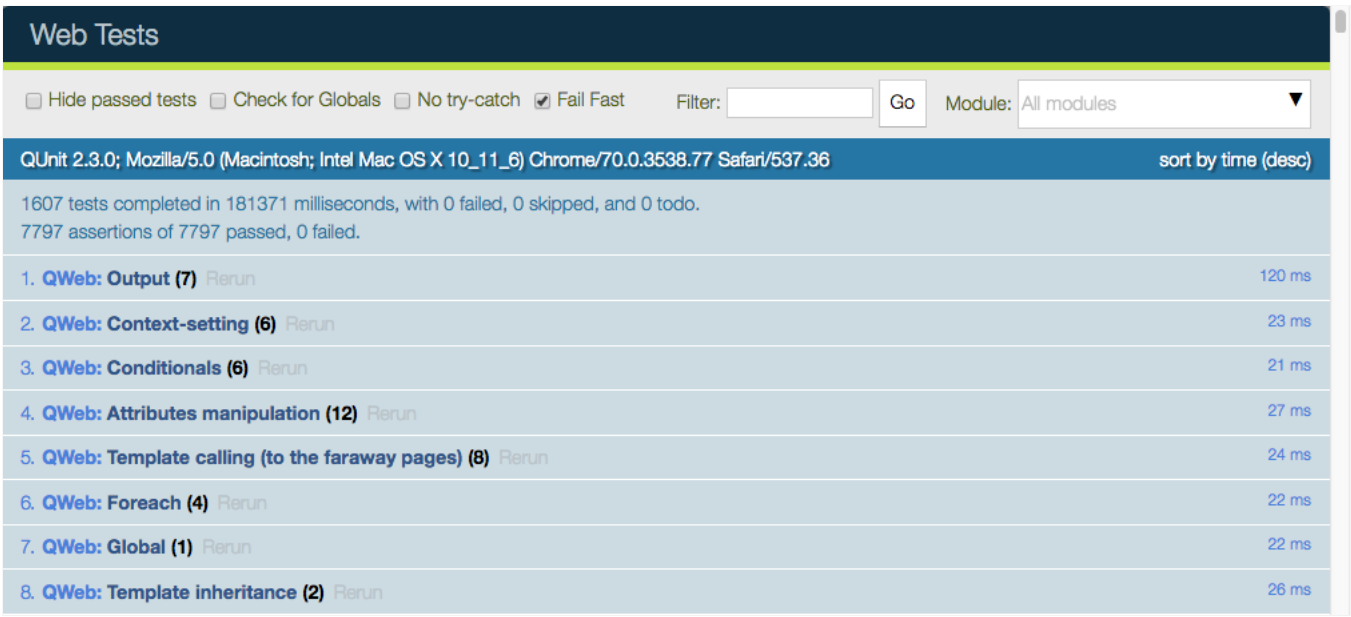
The Odoo framework uses the QUnit (https://qunitjs.com/) library testing framework as a test runner. QUnit defines the concepts of *tests* and *modules* (a set of related tests), and gives us a web based interface to execute the tests.
For example, here is what a pyUtils test could look like:

```
QUnit.module('py_utils');

QUnit.test('simple arithmetic', function (assert) {
    assert.expect(2);

    var result = pyUtils.py_eval("1 + 2");
    assert.strictEqual(result, 3, "should properly evaluate sum");
    result = pyUtils.py_eval("42 % 5");
    assert.strictEqual(result, 2, "should properly evaluate modulo operator");
});
```

The main way to run the test suite is to have a running Odoo server, then navigate a web browser to **`/web/tests`** . The test suite will then be executed by the web browser Javascript engine.



The web UI has many useful features: it can run only some submodules, or filter tests that match a string. It can show every assertions, failed or passed, rerun specific tests, ...

> **⚠ Warning**
> While the test suite is running, make sure that:
>
> your browser window is focused,

> it is not zoomed in/out. It needs to have exactly 100% zoom level.
>
> If this is not the case, some tests will fail, without a proper explanation.

Testing Infrastructure

Here is a high level overview of the most important parts of the testing infrastructure:

there is an asset bundle named web.qunit_suite (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/views/webclient_templates.xml#L660). This bundle contains the main code (assets common + assets backend), some libraries, the QUnit test runner and the test bundles listed below.

a bundle named web.tests_assets (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/views/webclient_templates.xml#L594) includes most of the assets and utils required by the test suite: custom QUnit asserts, test helpers, lazy loaded assets, etc.

another asset bundle, web.qunit_suite_tests (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/views/webclient_templates.xml#L680), contains all the test scripts. This is typically where the test files are added to the suite.

there is a controller (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/controllers/main.py#L637) in web, mapped to the route */web/tests*. This controller simply renders the *web.qunit_suite* template.

to execute the tests, one can simply point its browser to the route */web/tests*. In that case, the browser will download all assets, and QUnit will take over.

there is some code in qunit_config.js (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/static/tests/helpers/qunit_config.js#L49) which logs in the console some information when a test passes or fails.

we want the runbot to also run these tests, so there is a test (in test_js.py (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/tests/test_js.py#L13)) which simply spawns a browser and points it to the *web/tests* url. Note that the browser_js method spawns a Chrome headless instance.

Modularity and testing

With the way Odoo is designed, any addon can modify the behaviour of other parts of the system. For example, the *voip* addon can modify the *FieldPhone* widget to use extra features. This is not really good from the perspective of the testing system, since this means that a test in the addon web will fail whenever the voip addon is installed (note that the runbot runs the tests with all addons installed).
At the same time, our testing sytem is good, because it can detect whenever another module breaks some core functionality. There is no complete solution to this issue. For now, we solve this on a case by case basis.
Usually, it is not a good idea to modify some other behaviour. For our voip example, it is certainly cleaner to add a new *FieldVOIPPhone* widget and modify the few views that needs it. This way, the *FieldPhone* widget is not impacted, and both can be tested.

Adding a new test case

Let us assume that we are maintaining an addon *my_addon*, and that we want to add a test for some javascript code (for example, some utility function myFunction, located in *my_addon.utils*). The process to add a new test case is the following:

1. create a new file *my_addon/static/tests/utils_tests.js*. This file contains the basic code to add a QUnit module *my_addon > utils*.

```
odoo.define('my_addon.utils_tests', function (require) {
"use strict";

var utils = require('my_addon.utils');

QUnit.module('my_addon', {}, function () {

    QUnit.module('utils');

});
});
```

2. In *my_addon/assets.xml*, add the file to the main test assets:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>
    <template id="qunit_suite_tests" name="my addon tests" inherit_id="web.qunit_suite_tests">
        <xpath expr="//script[last()]" position="after">
            <script type="text/javascript" src="/my_addon/static/tests/utils_tests.js"/>
        </xpath>
    </template>
</odoo>
```

3. Restart the server and update *my_addon*, or do it from the interface (to make sure the new test file is loaded)
4. Add a test case after the definition of the *utils* sub test suite:

```
QUnit.test("some test case that we want to test", function (assert) {
    assert.expect(1);

    var result = utils.myFunction(someArgument);
    assert.strictEqual(result, expectedResult);
});
```

5. Visit */web/tests/* to make sure the test is executed

Helper functions and specialized assertions

Without help, it is quite difficult to test some parts of Odoo. In particular, views are tricky, because they communicate with the server and may perform many rpcs, which needs to be mocked. This is why we developed some specialized helper functions, located in test_utils.js (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/static/tests/helpers/test_utils.js).

Mock test functions: these functions help setting up a test environment. The most important use case is mocking the answers given by the Odoo server. These functions use a mock server (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/static/tests/helpers/mock_server.js). This is a javascript class that simulates answers to the most common model methods: read, search_read, nameget, ...

DOM helpers: useful to simulate events/actions on some specific target. For example, testUtils.dom.click performs a click on a target. Note that it is safer than doing it manually, because it also checks that the target exists, and is visible.

create helpers: they are probably the most important functions exported by test_utils.js (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/static/tests/helpers/test_utils.js). These helpers are useful to create a widget, with a mock environment, and a lot of small detail to simulate as much as possible the real conditions. The most important is certainly createView (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/static/tests/helpers/test_utils_create.js#L2

qunit assertions (https://github.com/odoo/odoo/blob/51ee0c3cb59810449a60dae0b086b49b1ed6f946/addons/web/static/tests/helpers/qunit_asserts.js): QUnit can be extended with specialized assertions. For Odoo, we frequently test some DOM properties. This is why we made some assertions to help with that. For example, the *containsOnce* assertion takes a widget/jQuery/HtmlElement and a selector, then checks if the target contains exactly one match for the css selector.

For example, with these helpers, here is what a simple form test could look like:

```
QUnit.test('simple group rendering', function (assert) {
    assert.expect(1);

    var form = testUtils.createView({
        View: FormView,
        model: 'partner',
        data: this.data,
        arch: '<form string="Partners">' +
                '<group>' +
                    '<field name="foo"/>' +
                '</group>' +
            '</form>',
        res_id: 1,
    });

    assert.containsOnce(form, 'table.o_inner_group');

    form.destroy();
});
```

Notice the use of the testUtils.createView helper and of the containsOnce assertion. Also, the form controller was properly destroyed at the end of the test.

Best Practices

In no particular order:

- all test files should be added in *some_addon/static/tests/*
- for bug fixes, make sure that the test fails without the bug fix, and passes with it. This ensures that it actually works.
- try to have the minimal amount of code necessary for the test to work.
- usually, two small tests are better than one large test. A smaller test is easier to understand and to fix.
- always cleanup after a test. For example, if your test instantiates a widget, it should destroy it at the end.
- no need to have full and complete code coverage. But adding a few tests helps a lot: it makes sure that your code is not completely broken, and whenever a bug is fixed, it is really much easier to add a test to an existing test suite.
- if you want to check some negative assertion (for example, that a HtmlElement does not have a specific css class), then try to add the positive assertion in the same test (for example, by doing an action that changes the state). This will help avoid the test to become dead in the future (for example, if the css class is changed).

Tips

- running only one test: you can (temporarily!) change the *QUnit.test(...)* definition into *QUnit.only(...)*. This is useful to make sure that QUnit only runs this specific test.
- debug flag: most create utility functions have a debug mode (activated by the debug: true parameter). In that case, the target widget will be put in the DOM instead of the hidden qunit specific fixture, and more information will be logged. For example, all mocked network communications will be available in the console.
- when working on a failing test, it is common to add the debug flag, then comment the end of the test (in particular, the destroy call). With this, it is possible to see the state of the widget directly, and even better, to manipulate the widget by clicking/interacting with it.

Integration Testing

Testing Python code and JS code separately is very useful, but it does not prove that the web client and the server work together. In order to do that, we can write another kind of test: tours. A tour is a mini scenario of some interesting business flow. It explains a sequence of steps that should be followed. The test runner will then create a Chrome headless browser, point it to the proper url and simulate the click and inputs, according to the scenario.

Screenshots and screencasts during browser_js tests

When running tests that use HttpCase.browser_js from the command line, the Chrome browser is used in headless mode. By default, if a test fails, a PNG screenshot is taken at the moment of the failure and written in

```
'/tmp/odoo_tests/{db_name}/screenshots/'
```

Two new command line arguments were added since Odoo 13.0 to control this behavior: **`--screenshots`** (cmdline.html#cmdoption-odoo-bin-screenshots) and **`--screencasts`** (cmdline.html#cmdoption-odoo-bin-screencasts)

Performance Testing

Query counts

One of the ways to test performance is to measure database queries. Manually, this can be tested with the **`--log-sql`** CLI parameter. If you want to establish the maximum number of queries for an operation, you can use the **`assertQueryCount()`** method, integrated in Odoo test classes.

```
with self.assertQueryCount(11):
    do_something()
```

Database population

Odoo CLI offers a database population (cmdline.html#reference-cmdline-populate) feature.

```
odoo-bin populate
```

Instead of the tedious manual, or programmatic, specification of test data, one can use this feature to fill a database on demand with the desired number of test data. This can be used to detect diverse bugs or performance issues in tested flows.
To specify this feature for a given model, the following methods and attributes can be defined.

Model._populate_sizes

Return a dict mapping symbolic sizes (**'small'** , **'medium'** , **'large'**) to integers, giving the minimal number of records that **_populate()** should create.

The default population sizes are:

- **small** : 10
- **medium** : 100
- **large** : 1000

Model._populate_dependencies

Return the list of models which have to be populated before the current one.

Return type:

list (https://docs.python.org/3/library/stdtypes.html#list)

Model._populate(*size*) (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L6324)

Create records to populate this model.

Parameters:

size (**str** (https://docs.python.org/3/library/stdtypes.html#str)) – symbolic size for the number of records: **'small'** , **'medium'** or **'large'**

Model._populate_factories() (https://github.com/odoo/odoo/blob/14.0/odoo/models.py#L6272)

Generates a factory for the different fields of the model.

factory is a generator of values (dict of field values).

Factory skeleton:

```
def generator(iterator, field_name, model_name):
    for counter, values in enumerate(iterator):
        # values.update(dict())
        yield values
```

See **odoo.tools.populate** for population tools and applications.

Returns:

list of pairs(field_name, factory) where **factory** is a generator function.

Return type:

list (https://docs.python.org/3/library/stdtypes.html#list)(tuple (https://docs.python.org/3/library/stdtypes.html#tuple)(str (https://docs.python.org/3/library/stdtypes.html#str), generator))

> It is the responsibility of the generator to handle the field_name correctly. The generator could generate values for multiple fields together. In this case, the field_name should be more a "field_group" (should be begin by a "_"), covering the different fields updated by the generator (e.g. "_address" for a generator updating multiple address fields).

> You have to define at least **_populate()** or **_populate_factories()** on the model to enable database population.

Example model

```
from odoo.tools import populate

class CustomModel(models.Model)
    _inherit = "custom.some_model"
    _populate_sizes = {"small": 100, "medium": 2000, "large": 10000}
    _populate_dependencies = ["custom.some_other_model"]

    def _populate_factories(self):
        # Record ids of previously populated models are accessible in the registry
        some_other_ids = self.env.registry.populated_models["custom.some_other_model"]

        def get_some_field(values=None, random=None, **kwargs):
            """ Choose a value for some_field depending on other fields values.

                :param dict values:
                :param random: seeded :class:`random.Random` object
            """
            field_1 = values['field_1']
            if field_1 in [value2, value3]:
                return random.choice(some_field_values)
            return False

        return [
            ("field_1", populate.randomize([value1, value2, value3])),
            ("field_2", populate.randomize([value_a, value_b], [0.5, 0.5])),
            ("some_other_id", populate.randomize(some_other_ids)),
            ("some_field", populate.compute(get_some_field, seed="some_field")),
            ('active', populate.cartesian([True, False])),
        ]

    def _populate(self, size):
        records = super()._populate(size)

        # If you want to update the generated records
        # E.g setting the parent-child relationships
        records.do_something()

        return records
```

Population tools

Multiple population tools are available to easily create the needed data generators.

odoo.tools.populate.cartesian(*vals*, *weights=None*, *seed=False*, *formatter=<function format_str>*, *then=None*) **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/populate.py#L56)**

Return a factory for an iterator of values dicts that combines all **vals** for the field with the other field values in input.

Parameters:

vals (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list in which a value will be chosen, depending on **weights**

weights (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list of probabilistic weights

seed – optional initialization of the random number generator

formatter (**function**) – (val, counter, values) –> formatted_value

then (**function**) – if defined, factory used when vals has been consumed.

Returns:

function of the form (iterator, field_name, model_name) -> values

Return type:

function (iterator, str (https://docs.python.org/3/library/stdtypes.html#str), str (https://docs.python.org/3/library/stdtypes.html#str)) -> dict

odoo.tools.populate.compute(*function*, *seed=None*) **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/populate.py#L121)**

Return a factory for an iterator of values dicts that computes the field value as **function(values, counter, random)**, where **values** is the other field values, **counter** is an integer, and **random** is a pseudo-random number generator.

Parameters:

function (**function**) – (values, counter, random) –> field_values

seed – optional initialization of the random number generator

Returns:

function of the form (iterator, field_name, model_name) -> values

Return type:

function (iterator, str (https://docs.python.org/3/library/stdtypes.html#str), str (https://docs.python.org/3/library/stdtypes.html#str)) -> dict

odoo.tools.populate.constant(*val*, *formatter=<function format_str>*) **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/populate.py#L107)**

Return a factory for an iterator of values dicts that sets the field to the given value in each input dict.

Returns:

function of the form (iterator, field_name, model_name) -> values

Return type:

function (iterator, str (https://docs.python.org/3/library/stdtypes.html#str), str (https://docs.python.org/3/library/stdtypes.html#str)) -> dict

odoo.tools.populate.iterate(*vals*, *weights=None*, *seed=False*, *formatter=<function format_str>*, *then=None*) **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/populate.py#L81)**

Return a factory for an iterator of values dicts that picks a value among **vals** for each input. Once all **vals** have been used once, resume as **then** or as a **randomize** generator.

Parameters:

vals (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list in which a value will be chosen, depending on **weights**

weights (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list of probabilistic weights

seed – optional initialization of the random number generator

formatter (**function**) – (val, counter, values) –> formatted_value

then (**function**) – if defined, factory used when vals has been consumed.

Returns:

function of the form (iterator, field_name, model_name) -> values

Return type:

function (iterator, str (https://docs.python.org/3/library/stdtypes.html#str), str (https://docs.python.org/3/library/stdtypes.html#str)) -> dict

odoo.tools.populate.randint(*a*, *b*, *seed=None*) **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/populate.py#L139)**

Return a factory for an iterator of values dicts that sets the field to the random integer between a and b included in each input dict.

Parameters:

a (**int** (https://docs.python.org/3/library/functions.html#int)) – minimal random value

b (**int** (https://docs.python.org/3/library/functions.html#int)) – maximal random value

Returns:

function of the form (iterator, field_name, model_name) -> values

Return type:

function (iterator, str (https://docs.python.org/3/library/stdtypes.html#str), str (https://docs.python.org/3/library/stdtypes.html#str)) -> dict

odoo.tools.populate.randomize(*vals*, *weights=None*, *seed=False*, *formatter=<function format_str>*, *counter_offset=0*) **(https://github.com/odoo/odoo/blob/14.0/odoo/tools/populate.py#L35)**

Return a factory for an iterator of values dicts with pseudo-randomly chosen values (among **vals**) for a field.

Parameters:

vals (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list in which a value will be chosen, depending on **weights**

weights (**list** (https://docs.python.org/3/library/stdtypes.html#list)) – list of probabilistic weights

seed – optional initialization of the random number generator

formatter (**function**) – (val, counter, values) –> formatted_value

counter_offset (**int** (https://docs.python.org/3/library/functions.html#int)) –

Returns:

function of the form (iterator, field_name, model_name) -> values

Return type:

function (iterator, str (https://docs.python.org/3/library/stdtypes.html#str), str (https://docs.python.org/3/library/stdtypes.html#str)) -> dict