**Testing in Odoo**

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

* Python unit tests (see [Testing Python code](https://www.odoo.com/documentation/14.0/reference/testing.html#testing-python-code)): useful for testing model business logic
* JS unit tests (see [Testing JS code](https://www.odoo.com/documentation/14.0/reference/testing.html#testing-js-code)): useful to test the javascript code in isolation
* Tours (see [Integration Testing](https://www.odoo.com/documentation/14.0/reference/testing.html#integration-testing)): tours simulate a real situation. They ensures that the python and the javascript parts properly talk to each other.

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");**

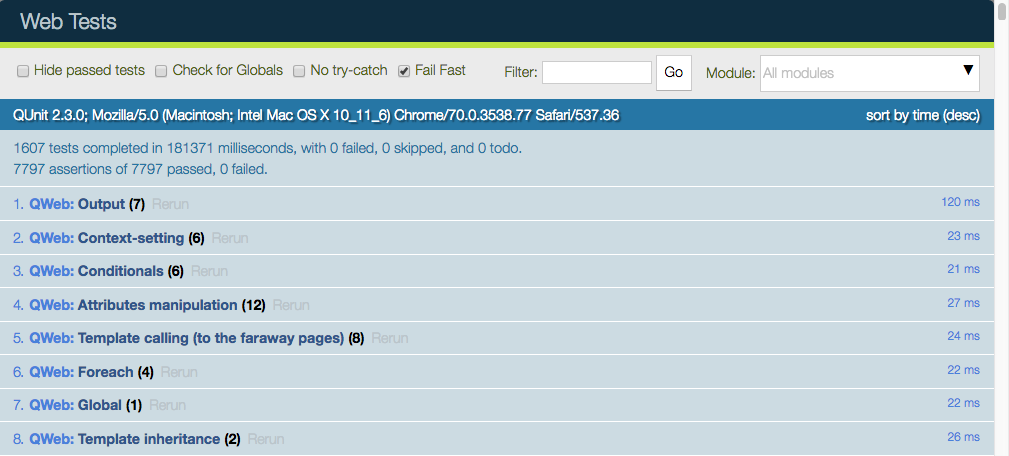
**assert.strictEqual(result, 3, "should properly evaluate sum");**

**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" \l "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" \l "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" \l "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*.
2. ***odoo.define('my\_addon.utils\_tests', function (require) {***
3. ***"use strict";***
4. ***var utils = require('my\_addon.utils');***
5. ***QUnit.module('my\_addon', {}, function () {***
6. ***QUnit.module('utils');***
7. ***});***
8. ***});***
9. In *my\_addon/assets.xml*, add the file to the main test assets:
10. ***<?xml version="1.0" encoding="utf-8"?>***
11. ***<odoo>***
12. ***<template id="qunit\_suite\_tests" name="my addon tests" inherit\_id="web.qunit\_suite\_tests">***
13. ***<xpath expr="//script[last()]" position="after">***
14. ***<script type="text/javascript" src="/my\_addon/static/tests/utils\_tests.js"/>***
15. ***</xpath>***
16. ***</template>***
17. ***</odoo>***
18. Restart the server and update *my\_addon*, or do it from the interface (to make sure the new test file is loaded)
19. Add a test case after the definition of the *utils* sub test suite:
20. ***QUnit.test("some test case that we want to test", function (assert) {***
21. ***assert.expect(1);***
22. ***var result = utils.myFunction(someArgument);***
23. ***assert.strictEqual(result, expectedResult);***
24. ***});***
25. 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" \l "L267).
* [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.

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**](https://www.odoo.com/documentation/14.0/reference/cmdline.html#cmdoption-odoo-bin-screenshots) and [**--screencasts**](https://www.odoo.com/documentation/14.0/reference/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](https://www.odoo.com/documentation/14.0/reference/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()**](https://www.odoo.com/documentation/14.0/reference/testing.html#odoo.models.Model._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*)**

Create records to populate this model.

**Parameters**

**size** (**[str](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "(in Python v3.9))**) – symbolic size for the number of records: **'small'**, **'medium'** or **'large'**

**Model.\_populate\_factories()**

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](https://www.odoo.com/documentation/14.0/reference/testing.html" \l "module-odoo.tools.populate" \o "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" \l "str" \o "(in Python v3.9)), 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()**](https://www.odoo.com/documentation/14.0/reference/testing.html#odoo.models.Model._populate) or [**\_populate\_factories()**](https://www.odoo.com/documentation/14.0/reference/testing.html#odoo.models.Model._populate_factories) on the model to enable database population.

**Travis CI Tool Used In Odoo**

* [WHAT IS CONTINUOUS INTEGRATION (CI)?](https://docs.travis-ci.com/user/for-beginners/#what-is-continuous-integration-ci)
* [CI BUILDS AND AUTOMATION: BUILDING, TESTING, DEPLOYING](https://docs.travis-ci.com/user/for-beginners/#ci-builds-and-automation-building-testing-deploying)
* [BUILDS, STAGES, JOBS AND PHASES](https://docs.travis-ci.com/user/for-beginners/#builds-stages-jobs-and-phases)
* [BREAKING THE BUILD](https://docs.travis-ci.com/user/for-beginners/#breaking-the-build)
* [INFRASTRUCTURE AND ENVIRONMENT NOTES](https://docs.travis-ci.com/user/for-beginners/#infrastructure-and-environment-notes)

Welcome to Travis CI! This page provides some contexts and terminologies used throughout the platform and documentation, which might be helpful, if you are new here or new to Continuous Integration (CI).

## What Is Continuous Integration (CI)? [#](https://docs.travis-ci.com/user/for-beginners/#what-is-continuous-integration-ci)

Continuous Integration is the practice of merging in small code changes frequently - rather than merging in a large change at the end of a development cycle. The goal is to build healthier software by developing and testing in smaller increments. This is where Travis CI comes in.

As a continuous integration platform, Travis CI supports your development process by automatically building and testing code changes, providing immediate feedback on the success of the change. Travis CI can also automate other parts of your development process by managing deployments and notifications.

## CI Builds and Automation: Building, Testing, Deploying [#](https://docs.travis-ci.com/user/for-beginners/#ci-builds-and-automation-building-testing-deploying)

When you run a build, Travis CI clones your GitHub repository into a brand-new virtual environment, and carries out a series of tasks to build and test your code. If one or more of those tasks fail, the build is considered [broken](https://docs.travis-ci.com/user/for-beginners/#breaking-the-build). If none of the tasks fail, the build is considered [passed](https://docs.travis-ci.com/user/for-beginners/#breaking-the-build) and Travis CI can deploy your code to a web server or application host.

CI builds can also automate other parts of your delivery workflow. This means you can have jobs depend on each other with [Build Stages](https://docs.travis-ci.com/user/build-stages/), set up [notifications](https://docs.travis-ci.com/user/notifications/), prepare [deployments](https://docs.travis-ci.com/user/deployment/) after builds and many other tasks.

## Builds, Stages, Jobs and Phases [#](https://docs.travis-ci.com/user/for-beginners/#builds-stages-jobs-and-phases)

In the Travis CI documentation, some common words have specific meanings:

* build - a group of jobs that run in sequence. For example, a build might have two jobs, each of which tests a project with a different version of a programming language. A build finishes when all of its jobs are finished.
* stage - a group of jobs that run in parallel as part of a sequential build process composed of multiple [stages](https://docs.travis-ci.com/user/build-stages/).
* job - an automated process that clones your repository into a virtual environment and then carries out a series of phases such as compiling your code, running tests, etc. A job fails, if the return code of the script phase is non-zero.
* phase - the [sequential steps](https://docs.travis-ci.com/user/job-lifecycle/) of a job. For example, the install phase, comes before the script phase, which comes before the optional deploy phase.

## Breaking the Build [#](https://docs.travis-ci.com/user/for-beginners/#breaking-the-build)

The build is considered broken, when one or more of its jobs complete with a state that is not passed:

* errored - a command in the before\_install, install, or before\_script phase returned a non-zero exit code. The job stops immediately.
* failed - a command in the script phase returned a non-zero exit code. The job continues to run until it completes.
* canceled - a user cancels the job before it completes.

Our [Common Builds Problems](https://docs.travis-ci.com/user/common-build-problems/) page is a good place to start troubleshooting why your build is broken.

## Infrastructure and Environment Notes [#](https://docs.travis-ci.com/user/for-beginners/#infrastructure-and-environment-notes)

Travis CI offers a few different infrastructure environments, so you can select the setup that suits your project best:

* Ubuntu Linux - these Linux Ubuntu environments run inside full virtual machines, provide plenty of computational resources, and support the use of sudo, setuid, and setgid. Check out more information on the [Ubuntu Linux Build Environment](https://docs.travis-ci.com/user/reference/linux/).
* macOS - uses one of several versions of the macOS operating system. This environment is useful for building projects that require the macOS software, such as projects written in Swift. It is not a requirement to use the macOS environment, if you develop on a macOS machine. Here you can find more details on the [macOS Build Environment](https://docs.travis-ci.com/user/reference/osx/).
* Windows - currently Windows Server version 1803 is supported. If you want to know more about it, see the [Windows Build Environment](https://docs.travis-ci.com/user/reference/windows/).

**References**

1. <https://www.odoo.com/documentation/14.0/reference/testing.html>

2. <https://docs.travis-ci.com/user/for-beginners/>