TEST FIXTURE API

This section describes the API available to you when creating or editing tests in code.

For a top-level overview of the test creation process - (either using the Visual Recorder or by editing JavaScript code), please refer to the [Fixtures](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/#Using_TestCafe_Control_Panel_Projects_Tab_Fixtures) topic. When editing tests using code, you will work with **.test.js** files that use the syntax outlined below.

* '@fixture Features';
* '@page ./Features.aspx';
* '@auth JohnSmith:JSp123';
* '@require :my\_module';
* '@require ./helpers/test.js';
* '@test'['Refresh'] = {
* 'Click "Refresh" button': function() {
* this.prevImageSrc = getCaptcha().getImage().prop('src');
* act.click(getCaptcha().getRefreshButton());
* },
* 'Check if the captchaImage url has been changed': function() {
* notEq(getCaptcha().getImage().prop('src'), this.prevImageSrc);
* }
* };
* function getCaptcha() {
* return dx.captcha('Captcha');
* };

These files define named test fixtures with a specified entry page. Fixtures contain tests, which in turn are broken down into series of steps, each consisting of TestCafe Actions (no more than one action per step), TestCafe Assertions and custom JavaScript code. In addition to user actions and assertions, TestCafe provides a method to handle dialog windows.

[**Common Concepts**](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts/)

This section provides information about common concepts behind organizing your test code.

[**User Actions**](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions/)

Use the following methods to perform user actions. Remember that TestCafe has a maximum of one user action per test step.

[**Assertions**](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Assertions/)

Using TestCafe assertions, you can check whether an arbitrary expression resolves to **true** or **false**, or whether two objects are equal or different.

[**Native Dialog Handling**](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Native_Dialog_Handling/)

If a browser displays a native message box during the test, use the functions from this section to close the dialog as required, and avoid test failure.

[**Miscellaneous**](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous/)

This section lists utility data structures, keywords and selectors used by testing functions.

COMMON CONCEPTS

This section provides information about common concepts behind organizing your test code.

**Sharing Data Between Steps**

The **this** keyword can be used to share data between test steps. Note that you cannot use it to store DOM elements or jQuery objects.

To share data, declare a property and assign a value to use. You can then access this property's value in successive steps.

* this.propertyName = value;

The following example demonstrates the use of the **this** keyword in a test. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Handle Confirm'] = {
* 'Click div "Populate Form"': function () {
* var populateFormButton = $(':containsExcludeChildren(Populate Form)');
* handleConfirm('OK');
* act.click(populateFormButton);
* },
* 'Click submit button "Submit"': function () {
* this.autoGeneratedName = $('#Developer\_Name').val();
* var submitButton = $('.button.blue.fix-width-180');
* act.click(submitButton);
* },
* 'Check result': function () {
* var header = $('.article-header');
* var expectedResult = 'Thank You, ' + this.autoGeneratedName + '!';
* eq(header.html(), expectedResult);
* }
* };

**Test Run Metadata**

There will be scenarios when you may need to get a worker name from test code. For example, the way a test executes can depend on the device type. Or you can test an account by using a worker name as a username.

You can obtain a worker name from test code by using the **this.\_\_workerName** property. Names of Workers will be the same as those displayed in the WORKERS tab of the Control Panel (see the [Connecting a Remote Device](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/#Using_TestCafe_Control_Panel_Workers_Tab_Connecting_a_Remote_Device) topic).

The code above illustrates an example when a worker name is used as a username to test a login form.

* '@test'['Using a Worker Name'] = {
* 'Type in input': function () {
* var input = $('#gaia\_loginform').find('[name="Email"]');
* act.type(input, this.\_\_workerName);
* },
* 'Type in password input "Password"': function () {
* var passwordInput = $('#gaia\_loginform').find('[name="Passwd"]');
* act.type(passwordInput, 'Pass1234');
* },
* 'Click submit button "Sign in"': function () {
* var submitButton = $('#gaia\_loginform').find('[name="signIn"]');
* act.click(submitButton);
* }
* };

**Using External JS Libraries**

You can enrich TestCafe functionality and reduce repetitive code by including external JS libraries in a test. This way, the test code will become more organized and easier to read. Making changes in the external file once instead of editing each part of the duplicated code in a test will now be easier when needed.

External libraries can provide additional functionality to a test (for example, a module that helps you [run the same test with a different set of data each time](http://www.devexpress.com/Support/Center/Question/Details/T110436)) or just keep some repetitive code, which can then be used from different tests (for example, code related to authorization; refer to the [Mixins](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts" \l "Mixins)topic).

To include an individual external library in a test, use the **@require** directive.

* '@require ../modules/underscore.js';

When the underscore.js library has been included, all functions declared in that library become available for use within the test. The library's **each**function implements iteration over an array of objects.

* '@test'['Find customers'] = {
* 'Specify a salesperson': function () {
* var input = $('[name="salesperson"]');
* act.type(input, 'Peter Parker');
* },
* 'Check the customers list': function () {
* var customersTemplate = "<% \_.each(customers, function(name) { %><li><%= name %></li><% }); %>",
* expectedHTML = \_.template(customersTemplate, {
* customers: ['Moe Clark', 'John Evans', 'Larry Turner']
* });
* eq($("#custormers")[0].innerHTML, expectedHTML);
* }
* };

A set of .js files can be referenced one after another from code via the **@require** directive or can be specified as a module via the [TestCafe configuration file](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts" \l "Directory_Configuration_File), and then loaded into test code as a module.

Here is the content of the test\_config.json file.

* "modules": {
* "screenshotMaker": [
* "../modules/html2canvas.js",
* "../modules/screenshot\_maker.js"
* ]
* }

A module is loaded into test code using the **@require** directive followed by the module name after the colon.

* '@require :screenshotMaker';

In the [How to run the same test with a different set of data each time](http://www.devexpress.com/Support/Center/Question/Details/T110436) knowledge base article, an external library is loaded to a test as an individual file and serves to generate random data each time when running a test.

**Directory Configuration File**

**File name**: test\_config.json  
**Location**: directories with test fixtures (regardless of the nesting level)

Each folder with test fixtures can contain a configuration file. TestCafe automatically locates these files and applies their settings to all fixtures in the current directory. The same settings apply to all nested directories, unless they have local configuration files.

Note that settings are inherited by child directories from their parent directories. If a parent file defines a module, it will be available in all child folders. URLs in nested folders can also be specified relative to the parent value.

The following is a sample file listing that explains the supported syntax of a configuration file.

* {
* "baseUrl": "http://example.com/Products/",
* "modules": {
* "my\_module": [
* "./helpers/coordinateHelper.js",
* "./helpers/colorHelper.js"
* ]
* }
* }

**baseUrl** - Defines the URL for all test fixtures in the current folder. Specifying the base URL in the configuration file might prove useful if a folder name on the tested website changes. So, you will only need to update the path once.

**Important notes**

* In a top-level configuration file, you must specify an absolute base URL (including protocol). In configuration files of nested directories, you can specify either an absolute or relative base URL (if a parent folder contains its own configuration file).
* If you use the base URL, each fixture in the corresponding folder must define a relative path to the tested page. Otherwise, the base URL will be ignored.

Suppose that you have created a test for the *http://example.com/Products/Account/login.aspx* web page. The test is located in the Products subdirectory of the root directory. This is how you can specify the URL of the tested page.

1. In the configuration file of the root directory, specify the absolute base URL: "baseUrl": "http://example.com/".
2. In the configuration file of the subdirectory, specify the relative base URL: "baseUrl": "./Products/".
3. In the fixture containing the test, specify the relative URL: "@page ./Account/login.aspx".

**modules** - Defines modules - sets of JavaScript files that can be referenced in test fixture code.

To load a module to a fixture, use the **@require** directive as shown below. Note that individual JavaScript files will be loaded in the same order as defined in the configuration file.

* '@fixture Features';
* '@page ./Features.aspx';
* '@require :my\_module';
* '@require ./helpers/test.js';
* '@test'['Refresh'] = {
* 'Click "Refresh" button': function () {
* this.prevImageSrc = getCaptcha().getImage().prop('src');
* act.click(getCaptcha().getRefreshButton());
* },
* 'Check to see if the captchaImage url has been changed': function() {
* notEq(getCaptcha().getImage().prop('src'), this.prevImageSrc);
* }
* };

**Mixins**

As it is often the case, every single test in a group of tests has to repeat a certain routine operation. With TestCafe, you can write code for this operation once and reuse it in multiple tests.

To do this, create a mixin, which is a piece of test code meant to be reused across your test base.

To define a mixin, use the **@mixin** directive as shown in the code below.

* '@mixin'['Authentication'] = {
* 'Type in input login': function () {
* var input = $('#login\_input');
* act.type(input, 'Peter Parker');
* }
* };

You can put a mixin either to a [JS library](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Using_External_JS_Libraries) file (an existing or a new one) or to a test fixture file.

Mixins defined in a library can be used in any fixture. To reference an external library from test code, use the **@require** directive.

* '@require ./mixin.js';

Mixins embedded in a test file cannot be used outside of this file.

The placement of mixins in the fixture code is arbitrary, i.e., mixins can be placed before the tests, after or among them.

To call mixin code, put the **@mixin** directive followed by the mixin name after the test step name.

* '@test'['View profile'] = {
* 'Log In': '@mixin Authentication',
* 'Click link "My Profile"': function () {
* var link = $(':containsExcludeChildren(My Profile)');
* act.click(link);
* }
* };

Note that a call to a mixin will substitute for a test step. This means that you cannot combine a call to a mixin with some other code as a single step.

**Important note**

Mixins cannot be nested.

To share data between mixins and test steps (like between any test steps), use **this** keyword. For details, see [Sharing Data Between Steps](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Sharing_Data_Between_Steps).

The following example illustrates how to use a mixin. It shows the "Authentication" mixin that logs into an account. This mixin contains three steps: enter login, enter password and click the "Submit" button. The mixin is embedded in a test file and used as the "Log In" step in two tests.

* '@mixin'['Authentication'] = {
* 'Type in input login': function () {
* var input = $('#login\_input');
* act.type(input, 'Peter Parker');
* },
* 'Type in input password': function () {
* var input = $('#password\_input');
* act.type(input, '12345');
* },
* 'Click submit button': function () {
* var submitButton = $('#submit\_button');
* act.click(submitButton);
* }
* };
* '@test'['View profile'] = {
* 'Log In': '@mixin Authentication',
* 'Click link "My Profile"': function () {
* var link = $(':containsExcludeChildren(My Profile)');
* act.click(link);
* }
* };
* '@test'['Edit profile'] = {
* 'Log In': '@mixin Authentication',
* 'Click link "Edit Profile"': function () {
* var link = $(':containsExcludeChildren(Edit Profile)');
* act.click(link);
* }
* };

**Wrappers**

Some target web pages may contain complicated controls, making it difficult to understand and manage test code generated by [Visual Test Recorder](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Visual_Test_Recorder/). For example, the code can include very long expressions that are used to refer to web page elements. To simplify access to the elements and improve code readability, you can use TestCafe *wrappers*. These are sets of [JavaScript functions](https://github.com/DevExpress/TestCafe-Wrappers#ContentsAPI) that help you easily work with complex controls in TestCafe. You can use these functions instead of the recorded selector expressions to access the desired elements.

Another issue that the wrappers help to handle is dynamic indexes of web page elements. Sometimes the indexes can change from one test run to another. Thus, the recorded code may not work as expected. To make your test more stable, use wrapper functions.

DevExpress provides a set of wrappers for DevExpress ASP.NET controls. However, you can create wrappers for any other control on your own.

To use the wrappers in tests, follow these steps:

1. Download and unpack the [.zip archive](https://github.com/DevExpress/TestCafe-Wrappers) with TestCafe wrappers to any folder on your machine.
2. Move the wrappers folder to the directory that contains your TestCafe tests. So, the test and wrapper folder must be located at the same directory level.
3. Define the modules section in the [directory configuration file](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Directory_Configuration_File) as shown below:
   1. "modules": {
   2. "dx": [
   3. "../wrappers/dx.js",
   4. "../wrappers/web/ASPxGridViewWrapper.js"
   5. ]
   6. }

After these steps, you are ready to use wrapper functions in your tests.

For an example of how to use TestCafe wrappers in tests, see [How to Use TestCafe Wrappers in Tests](https://testcafe.devexpress.com/Documentation/Examples/#How_to_Use_TestCafe_Wrappers_in_Tests).

**Test Parameterization**

Test parameterization is a testing method utilizing parameters rather than fixed values. So you can run the same test with different data sets (test cases) used as inputs or expected outputs that are stored separately from the test steps.

In the Control Panel, parameterized tests represent a group of tests. The number of tests corresponds with the number of test conditions. You can [run parameterized tests](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/#Using_TestCafe_Control_Panel_Projects_Tab_Fixtures_Parameterized_Tests) either as a whole group or individually.

Test cases represent an array of parameters and values, and use the syntax as follows.

* {'@name': 'test\_case\_name', parameter\_name\_1: 'parameter\_value\_1', parameter\_name\_2: 'parameter\_value\_2', ...}

The **@name** parameter specifies the name of the test case. This parameter is optional. If it isn't specified, the test case will have the auto-generated name "Test case at index N" where N is the index of the test case in the array.

The **parameter*\_*name** and **parameter*\_*value** specify the name and the value of the parameter that will be used in test steps.

To specify test cases within the test code, write them after the **@testCases** directive in the beginning of the test before the test steps.

* '@test'['Data-driven testing'] = {
* '@testCases': [
* {'@name': 'Peter', login: 'Peter Parker', password: '12345', group: 'admin'},
* {'@name': 'John', login: 'John Smith', password: 'test', group: 'user'},
* {'@name': 'Olivia', login: 'Olivia Taylor', password: 'qwerty', group: 'moderator'}
* ],
* ...
* };

If there are many test cases and you wish to keep your test code organized, you can save them to a separated JSON file. It is also useful if you are going to share these data sets with other tests.

For that, create a JSON file (for example, user-credentials-cases.json file) and save test cases to it.

* [
* {"@name": "Peter", "login": "Peter Parker", "password": "12345", "group": "admin"},
* {"@name": "John", "login": "John Smith", "password": "test", "group": "user"},
* {"@name": "Olivia", "login": "Olivia Taylor", "password": "qwerty", "group": "moderator"}
* ]

To include the separated test cases in the test code, specify a relative path to that file by using the **@testCases** directive.

* '@test'['Data-driven testing'] = {
* '@testCases': './user\_credentials\_cases.json',
* ...
* };

To obtain the test case's parameters from the test steps, use the **this** keyword followed by the parameter name.

* '@test'['Data-driven testing'] = {
* ...
* 'Type in input login': function () {
* var input = $('#login\_input');
* act.type(input, this.login);
* },
* 'Type in input password': function () {
* var input = $('#password\_input');
* act.type(input, this.password);
* },
* ...
* };

The following example illustrates the test that will be run with different data parameters.

* '@test'['Data-driven testing'] = {
* '@testCases': [
* {'@name': 'Peter', login: 'Peter Parker', password: '12345', group: 'admin'},
* {'@name': 'John', login: 'John Smith', password: 'test', group: 'user'},
* {'@name': 'Olivia', login: 'Olivia Taylor', password: 'qwerty', group: 'moderator'}
* ],
* 'Type in input login': function () {
* var input = $('#login\_input');
* act.type(input, this.login);
* },
* 'Type in input password': function () {
* var input = $('#password\_input');
* act.type(input, this.password);
* },
* 'Click submit button': function () {
* var submitButton = $('#submit\_button');
* act.click(submitButton);
* },
* 'Check Result': function () {
* var header = $('#group\_header');
* eq(header.text(), this.group);
* }
* };

**Http Authentication**

TestCafe allows you to test web pages that are protected with Http Basic or Windows (NTLM) authentication.

User's login and password can be specified [within a fixture creating dialog](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/#Using_TestCafe_Control_Panel_Projects_Tab_Fixtures_Creating_a_Fixture), during [a test recording](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/#Using_TestCafe_Control_Panel_Projects_Tab_Fixtures_Test_Recording) or manually via the **@auth** directive within a fixture code. Since TestCafe supports two-way synchronization, user credentials specified by any means mentioned before will appear both in the UI and code.

To specify user credentials within the code, write the **@auth** directive followed by a colon-separated pair of login and password.

* '@auth login:password';

Once user's login and password are defined either within the UI or within the code, TestCafe won't ask for them again.

**Important note**

Note that in case of Windows authentication, TestCafe additionally requires domain and workstation (PC) names. By default, these names are automatically received from the machine where TestCafe is installed.

**IFrame Support**

TestCafe allows you to test content which is located within the same-domain and cross-domain IFrames.

To implement user actions, assertions and other scripts for elements within IFrames, use the **inIFrame** function. When you assign this function to a step, web elements selectors will be considered in the context of the specified IFrame.

* 'First step': inIFrame('#iFrame', function () {
* test step code
* })

In the example above, the first parameter '#iframe' specifies the iframe's selector ([user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target)) on the web page. The second parameter is a function that will execute user actions, assertions or other scripts within the specified iframe.

The following example illustrates how to use the **inIFrames** function.

* 'Click submit button': inIFrame('#iframe', function() {
* act.click('#submitButton');
* }),
* 'Check if error message is appeared': inIFrame('#iframe', function() {
* ok($("#errormsg").is(":visible"));
* })

USER ACTIONS

Use the following methods to perform user actions. Remember that TestCafe has a maximum of one user action per test step.

**act.click**

Clicks webpage elements.

* act.click( target [, options] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) being clicked.  
**options** - [mouse action options](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#Mouse_action_options) (optional). A set of options providing additional details about user action.

The following example shows how to pass an array of targets to the **Click** action. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Click Checkboxes and Check State'] = {
* 'Click two checkbox labels': function() {
* var labels = [ $(':containsExcludeChildren(Support for testing on remote devices)'),
* $(':containsExcludeChildren(Reusing existing JavaScript code for testing)') ];
* act.click(labels);
* },
* 'Confirm checked state': function() {
* ok($('#testing-on-remote-devices').is(':checked'));
* ok($('#re-using-existing-javascript').is(':checked'));
* }
* };

The next example uses the **options** parameter to set the caret position in the edit box after it has been clicked.

* '@test'['Click Input'] = {
* 'Type name:': function () {
* act.type(getInput(), 'Peter Parker');
* },
* 'Move caret position': function () {
* act.click(getInput(), {
* caretPos: 5
* });
* },
* 'Erase a character': function () {
* act.press('backspace');
* },
* 'Check result': function () {
* eq(getInput().val(), 'Pete Parker');
* }
* };
* function getInput() {
* return $('#Developer\_Name');
* }

**act.rclick**

Right-clicks webpage elements.

* act.rclick( target [, options] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) being right-clicked.  
**options** - [mouse action options](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#Mouse_action_options) (optional). A set of options providing additional details about user action.

Note that this action will not invoke integrated browser context menus, native editor menus, etc. Use it to perform right-clicks that are processed by webpage elements, not the browser.

The following example shows how to use the Right Click action.

* '@test'['Popup Menu'] = {
* 'Right-click on the second cell': function () {
* act.rclick(getCellByIndex(2));
* },
* 'Hover the third cell': function () {
* act.hover(getCellByIndex(3));
* }
* };

**act.dblclick**

Double-clicks webpage elements.

* act.dblclick( target [, options] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) being double-clicked.  
**options** - [mouse action options](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#Mouse_action_options) (optional). A set of options providing additional details about the user action.

Note that this action will not invoke integrated browser actions such as text selection. Use it to perform double-clicks that are processed by webpage elements, not the browser.

**act.drag**

Drags a webpage element to a new position.

* act.drag( target, destination [, options] )
* act.drag( target, dragOffsetX, dragOffsetY [, options] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) being dragged.  
**destination** - Specifies the drop location as a DOM element or jQuery object.  
**dragOffsetX, dragOffsetY** - integer. These parameters specify drop coordinates as an offset from the mouse pointer's initial position.  
**options** - [mouse action options](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#Mouse_action_options) (optional). A set of options providing additional details about user action.

Note that this action will not invoke integrated browser actions such as copying and pasting text. Use it to perform drag and drop actions that are processed by webpage elements, not the browser.

The following example demonstrates how to use this action with a jQuery UI Slider control. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Drag slider'] = {
* 'Click label "I have tried TestCafe"': function () {
* var label = $(':containsExcludeChildren(I have tried TestCafe)');
* act.click(label);
* },
* 'Check initial slider value': function () {
* eq($('#Developer\_Rating').val(), 1);
* },
* 'Drag slider handle': function () {
* var sliderHandle = $('.ui-slider-handle');
* act.drag(sliderHandle, 360, 0, { offsetX: 10, offsetY: 10 });
* },
* 'Check resulting slider value': function () {
* eq($('#Developer\_Rating').val(), 7);
* }
* };

**act.hover**

Hovers the mouse pointer over webpage elements.

* act.hover( target [, options] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) to be hovered.  
**options** - [mouse action options](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#Mouse_action_options) (optional). A set of options providing additional details about user action.

Use this action to invoke popup elements such as hint windows, popup menus or dropdown lists that appear when hovering other elements.

The following example shows how to move the mouse pointer over a combo box to display the dropdown list, then select an item and check that the combo box value has changed. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['act.hover example'] = {
* 'Hover over the combo box': function () {
* var div = $('.text-field');
* act.hover(div);
* },
* 'Select "Both"': function () {
* var div = $(':containsExcludeChildren(Both)').eq(0);
* act.click(div);
* },
* 'Check result': function () {
* var value = $('.text-field').html();
* eq(value, 'Both');
* }
* };

**act.screenshot**

Takes a screenshot of the tested page.

* act.screenshot( )

Screenshots are saved to the test report, which you can view in the [Results tab](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/#Results_Tab).

**Important note**

The method is not available on the Linux operating system.

**act.navigateTo**

Navigates to the specified url.

* act.navigateTo( url )

**url**: String (required). Specifies the url to navigate to.

The following example shows how to use **act.navigateTo** user action.

* '@test'['Navigation to the EULA page'] = {
* ...
* 'Click label "I want to..."': function () {
* var label = $(':containsExcludeChildren(I want to read the EULA)');
* act.click(label);
* },
* 'Click submit button "Submit"': function () {
* var submitButton = $('#submit-button');
* act.click(submitButton);
* },
* 'Navigate to the EULA page': function () {
* act.navigateTo('http://testcafe.devexpress.com/eula');
* }
* };

**act.press**

Presses the specified keys.

* act.press( keysCommand )

**keysCommand**: String (required). Specifies the key, combination or sequence to be pressed.

* Alphanumeric keys  
  'a', 'A', '1', ...
* Modifier keys  
  'shift', 'alt', 'ctrl', 'meta'
* Navigation and action keys  
  'backspace', 'tab', 'enter', 'capslock', 'esc', 'space', 'pageup', 'pagedown', 'end', 'home', 'left', 'right', 'down', 'ins', 'delete'
* Key combinations  
  'shift+a', 'ctrl+d', ...
* Sequential key presses (any of the above in a space-separated string)  
  'a ctrl+b'

In addition to key presses handled by webpage elements, TestCafe also allows you to execute certain key presses processed by the browser.

* 'ctrl+a', 'backspace', 'delete', 'left', 'right', 'up', 'down', 'home', 'end', 'enter', 'tab', 'shift+tab', 'shift+left', 'shift+right', 'shift+up', 'shift+down', 'shift+home', 'shift+end'

With the exception of the keys and combinations listed above, the Press action will not invoke integrated browser keystrokes.

**Important note**

For web elements that have contentEditable attribute, TestCafe supports the following key-presses:   
- 'ctrl+a',   
- 'backspace', 'delete', 'left' and 'right' (only if there is selection within the element).

The following example shows how to use the **act.press** user action. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Key Presses'] = {
* 'Type name:': function () {
* act.type(getInput(), 'Peter Parker');
* },
* 'Erase "Peter"': function () {
* act.press('home right . delete delete delete delete');
* },
* 'Check result': function () {
* eq(getInput().val(), 'P. Parker');
* }
* };
* function getInput() {
* return $('#Developer\_Name');
* }

**act.select**

Selects text within the target elements of a web page.

* act.select( target [, offset] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) whose text will be selected.  
**offset** - Integer (optional). Identifies the direction and the end position of selection. Positive **offset** value represents a forward selection (where the start position is the first symbol of the text, and the end position is specified by the **offset** value counted from the start). Negative **offset** value represents a backward selection (where the start position is the last symbol of the text, and the end position is specified by the **offset** value counted from the end). If the **offset** is not specified, the whole text will be selected within the target element.

**Important note**

For web elements that have contentEditable attribute, **offset** parameter should be specified as an ordinal number of the symbol counted from the start, including all invisible symbols.

* act.select( target [, startPos, endPos] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) whose text will be selected.  
**startPos** - Integer (optional). Specifies the start position of the selection.  
**endPos** - Integer (optional). Specifies the end position of the selection.

NOTE: **startPos** and **endPos** can't have negative values. If the **startPos** value is greater than the **endPos** value, it will be executed as a backward selection.

* act.select( target, startLine, startPos, endLine, endPos )

**act.select** user action with the following parameters is useful when it comes to implementing text selection within a textarea element.

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the webpage element(s) whose text will be selected. If the input element is specified as a target, the last two parameters will be ignored.  
**startLine** - Integer (optional). Specifies the line number at which selection will start.  
**startPos** - Integer (optional). Specifies the start position of selection within the line defined by the **startLine**.  
**endLine** - Integer (optional). Specifies the line number at which selection will end.  
**endPos** - Integer (optional). Specifies the end position of selection within the line defined by **endLine**.

NOTE: **startPos**, **endPos**, **startLine** and **endLine** can’t have negative values. If the position defined by **startLine** and **startPos** is greater than the one defined by **endLine** and **endPos**, it will execute a backward selection.

The following version of the **act.select** user action allows you to implement text selection between two specified web elements (including texts of these elements).

* act.select( target-1, target-2 )

**target-1** - Object (Dom element, jQuery object, TextNode object) (required). Identifies a webpage element where the selection starts from. The start position of selection is the first character of the element’s text.  
**target-2** - Object (Dom element, jQuery object, TextNode object) (required). Identifies a webpage element where the selection ends. The end position of selection is the last character of the element’s text.

NOTE: If the web element defined by **target-2** is located on a higher level of the page hierarchy than the one defined by **target-1**, the user action will execute a backward selection.

The following example demonstrates text selection within the input element ([TestCafe Example Page](https://testcafe.devexpress.com/Example)).

* '@test'['Select text within input'] = {
* 'Type within “Your name:" input element': function () {
* act.type(getInput(), 'Test Cafe', {
* caretPos: 0
* });
* },
* 'Select within "Your name:" input element': function () {
* act.select(getInput(), 7, 1);
* },
* 'Confirm input selection state': function () {
* var input = getInput()[0];
* eq(input.selectionStart, 1);
* eq(input.selectionEnd, 7);
* if (input.selectionDirection)
* eq(input.selectionDirection, 'backward');
* }
* };
* var getInput = function () {
* return $('#Developer\_Name');
* };

**act.type**

Types the specified text into a webpage element.

* act.type( target, text [, options] )

**target** - [user action target](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#User_action_target) (required). Identifies the element that will receive input focus.  
**text** - String (required). The text to be typed into the specified webpage element. You cannot specify an empty string.  
**options** - [typing action options](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Miscellaneous#Typing_action_options) (optional). A set of options providing additional details about user action. If this parameter is omitted, TestCafe sets the cursor to the end of the text before typing, thus preserving the text that is already in the edit box.

NOTE: Use [key press actions](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actpress) to implement text management operations such as selection or deletion.

The following example shows how to use **act.type** with and without options. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Type and Replace'] = {
* 'Type name:': function () {
* act.type($('#Developer\_Name'), 'Peter');
* },
* 'Replace with last name': function () {
* act.type($('#Developer\_Name'), 'Paker', {
* replace: true
* });
* },
* 'Update last name': function () {
* act.type($('#Developer\_Name'), 'r', {
* caretPos: 2
* });
* },
* 'Check result': function () {
* eq($('#Developer\_Name').val(), 'Parker');
* }
* };

**act.upload**

Uploads a file or a set of files.

* act.upload( target, path )

This action uploads an individual file.

**target** - String. Identifies the target element selector expression of the input field where the file is uploaded.  
**path** - String. Identifies the relative path from the folder with the test file to the uploaded file.

Use the following variant of the user action to implement the uploading of multiple files at once.

* act.upload( target, paths )

**target** - String. Identifies the target element selector expression of the input field where the files are uploaded.  
**paths** - Array of Strings. Identifies a set of relative paths from the folder with the test file to the uploaded files.

If an error occurs during the uploading of the saved file(s), the test will fail.

If you need to imitate the browser's behaviour when a user clicks the **Cancel** button, use the following version of the user action.

* act.upload( target )

**target** - String. Identifies the target element selector expression of the input field where the files are uploaded.

The following example illustrates how to use the **act.upload** user action.

* '@test'['Uploading'] = {
* 'Click file button': function() {
* act.click('#upload\_input');
* },
* 'Upload "1.jpg", "2.jpg", "3.jpg" files': function() {
* act.upload('#upload\_input', [ './uploads/1.jpg', './uploads/2.jpg', './uploads/3.jpg' ]);
* },
* 'Click div': function() {
* act.click('#upload\_button');
* }
* };

**act.wait**

Pauses the test for a specified period of time.

* act.wait( milliseconds [, resumePredicate] )

**milliseconds**: Integer (required). The pause interval in milliseconds.  
**resumePredicate**: Function (optional). The action calls this function repeatedly, and resumes the test if the return value is **true**. Use [this](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Sharing_Data_Between_Steps) keyword within the predicate to use the shared data of the step.

The following example uses the **act.wait** user action to pause the test while animation is running. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Wait Example'] = {
* 'Initiate animation and wait': function () {
* $('.article-header').animate({ opacity: 0 }, 1000);
* act.wait(10000, isTransparent);
* },
* 'Type': function () {
* var input = $('#Developer\_Name');
* act.type(input, 'The wait is over!');
* }
* };
* function isTransparent() {
* return $('.article-header').css('opacity') == 0;
* }

**act.waitFor**

Allows you to pause test execution until a specified event occurs.

The **waitFor** action enables you to execute an asynchronous function while pausing a test run until it completes.

* act.waitFor( event [, timeout] )

**event**: Function (required). Represents an asynchronous function that receives a callback argument. Use [this](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Sharing_Data_Between_Steps) keyword within the function to use the shared data of the step.  
**timeout**: Integer (optional). The pause interval in milliseconds. If omitted, execution is paused for **10000** milliseconds.

Note that if the specified asynchronous event does not occur, the test will fail. Use the [act.wait](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions" \l "actwait) user action to enable a simple waiting mechanism without test failure.

The following versions of the **waitFor** action allow you to pause tests until a specific element (or multiple elements) appears on the page.

* act.waitFor( selector [, timeout] )

**selector**: String (required). The CSS selector that selects elements to wait for.  
**timeout**: Integer (optional). The pause interval in milliseconds. If omitted, execution is paused for **10000** milliseconds.

* act.waitFor( selectors [, timeout] )

**selectors**: Array of strings (required). An array of CSS selectors that select elements to wait for.  
**timeout**: Integer (optional). The pause interval in milliseconds. If omitted, execution is paused for **10000** milliseconds.

The following example uses the **act.waitFor** action to pause the test until the asynchronous function completes an HTTP request and obtains the status code.

* '@test'['waitFor example'] = {
* 'wait': function () {
* act.waitFor(function (callback) {
* $.get('http://some.address', function (jqxhr){
* eq(jqxhr.statusCode, 200);
* callback();
* });
* }, 500);
* }
* };

The following example demonstrates how to use the **act.waitFor** action to pause the test until the animated element is loaded.

* '@test'['Waiting for the animated element to load'] = {
* '1.Click the Start button to view the animated element': function() {
* act.click('.start-button');
* },
* '2.Wait for the animation to finish': function () {
* act.waitFor('#animated-element', 5000);
* },
* '3.Perform some actions with the animated element': function() {
* eq($('#animated-element').text(), 'The expected text');
* }
* };

# ASSERTIONS

Using TestCafe assertions, you can check whether an arbitrary expression resolves to **true** or **false**, or whether two objects are equal or different.

#### ok

Checks if the specified expression is true.

* ok( expression [, message] )

**expression** - Object (required). Use this parameter to specify an expression to be tested. If the expression resolves to true, the assertion passes. Otherwise, the test fails.  
**message** - String (optional). This text identifies the assertion in the error log if the expression resolves to false and the test fails.

The following code snippet demonstrates various expressions with the corresponding assertion results.

* ok(true, "true succeeds");
* ok(false, "false fails");
* ok("non-empty", "non-empty string succeeds");
* ok("", "empty string fails");
* ok(0, "0 fails");
* ok(NaN, "NaN fails");
* ok(null, "null fails");
* ok(undefined, "undefined fails");

The following example checks if an editor is initially disabled and becomes enabled after a check box is clicked. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Check Enabled State'] = {
* 'Check disabled': function () {
* ok($('#Developer\_Comments').is(':disabled'));
* },
* 'Click label "I have tried TestCafe"': function () {
* var label = $(':containsExcludeChildren(I have tried TestCafe)');
* act.click(label);
* },
* 'Check enabled': function () {
* notOk($('#Developer\_Comments').is(':disabled'));
* }
* };

#### notOk

Checks if the specified expression is false.

* notOk( expression [, message] )

**expression** - Object (required). Use this parameter to specify an expression to be tested. If the expression resolves to false, the assertion passes. Otherwise, the test fails.  
**message** - String (optional). This text identifies the assertion in the error log if the expression resolves to true and the test fails.

The following code snippet demonstrates various expressions with the corresponding assertion results.

* notOk(false, "false succeeds");
* notOk(true, "true fails");
* notOk("", "empty string succeeds");
* notOk("non-empty", "non-empty string fails");
* notOk(0, "0 succeeds");
* notOk(NaN, "NaN succeeds");
* notOk(null, "null succeeds");
* notOk(undefined, "undefined succeeds");

The following example checks if an editor is initially disabled and becomes enabled after a check box is clicked. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Check Enabled State'] = {
* 'Check disabled': function () {
* ok($('#Developer\_Comments').is(':disabled'));
* },
* 'Click label "I have tried TestCafe"': function () {
* var label = $(':containsExcludeChildren(I have tried TestCafe)');
* act.click(label);
* },
* 'Check enabled': function () {
* notOk($('#Developer\_Comments').is(':disabled'));
* }
* };

#### eq

Checks whether or not two objects are equal. This assertion uses a deep recursive comparison that works on primitive types, arrays and objects.

* eq( actual, expected [, message] )

**actual, expected** - Object (required). The objects that are being compared to each other. If the specified objects are different, the assertion fails.  
**message** - String (optional). This text identifies the assertion in the error log if the objects are different and the test fails.

The following example checks to see if a survey's 'Thank You' page displays the proper message. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['eq() Example'] = {
* 'Type name': function () {
* act.type(getInput(), 'Peter Parker');
* },
* 'Click "Submit"': function () {
* var submitButton = $('.button.blue.fix-width-180');
* act.click(submitButton);
* },
* 'Check message': function () {
* var header = $('.article-header');
* eq(header.html(), 'Thank You, Peter Parker!');
* }
* };

#### notEq

Checks whether or not two objects are different. This assertion uses a deep recursive comparison that works on primitive types, arrays and objects.

* notEq( actual, unexpected [, message] )

**actual, unexpected** - Object (required). The objects that are being compared to each other. If the specified objects are the same, the assertion fails.  
**message** - String (optional). This text identifies the assertion in the error log if the objects are identical and the test fails.

NATIVE DIALOG HANDLING

If a browser displays a native message box during the test, use the functions from this section to close the dialog as required, and avoid test failure.

**handleAlert**

Closes an alert dialog.

* handleAlert();

This function closes the dialogs invoked via the standard JavaScript **window.alert()** method. Note that the **handleAlert** call should precede the action that invokes the dialog.

* '@test'['handleAlert'] = {
* 'Click submit button "Try it"': function () {
* var submitButton = $(':containsExcludeChildren(Try it)', $("#iframeResult").contents());
* handleAlert();
* act.click(submitButton);
* }
* };

**handleBeforeUnload**

Closes a dialog that is shown when a window is about to be unloaded.

* handleBeforeUnload();

This function closes a dialog invoked via the standard JavaScript **window.onbeforeunload()** function. Note that the **handleBeforeUnload** call should precede the action that invokes the dialog.

* '@test'['handleBeforeUnload'] = {
* 'Click link "Link"': function () {
* var link = $(':containsExcludeChildren(LINK)');
* handleBeforeUnload();
* act.click(link);
* }
* };

**handleConfirm**

Closes a confirmation dialog.

* handleConfirm( result );

**result**: String (required). Specifies one of the following dialog result values.

* 'OK' or **true**: Closes the dialog and confirms the action, thus emulating the **OK** button.
* 'Cancel' or **false**: Closes the dialog and prohibits the action, thus emulating the **Cancel** button.

This function handles dialogs invoked via the standard JavaScript **window.confirm()** method. Note that the **handleConfirm** call should precede the action that invokes the dialog.

* '@test'['Handle Confirm'] = {
* 'Click div "Populate Form"': function () {
* var populateFormButton = $(':containsExcludeChildren(Populate Form)');
* handleConfirm('OK');
* act.click(populateFormButton);
* }
* };

**handlePrompt**

Closes a dialog that prompts user input.

* handlePrompt( input );

**input**: String (required). Specify the string to be entered before closing the dialog by clicking **OK**. Pass **null** to close the dialog by clicking **Cancel**.

This function handles dialogs invoked via the standard JavaScript **window.prompt()** method. Note that the **handlePrompt** call should precede the action that invokes a dialog.

* '@test'['handlePrompt'] = {
* 'Click submit button "Try it"': function () {
* var submitButton = $(':containsExcludeChildren(Try it)', $("#iframeResult").contents());
* handlePrompt("Peter");
* act.click(submitButton);
* }
* };

MISCELLANEOUS

This section lists utility data structures, keywords and selectors used by testing functions.

**User action target**

Identifies one or more webpage elements; and accepts the following object types.

* DOM Element
* jQuery object (with one or more elements)
* CSS selector (String)
* Function that returns any of the above
* Array of any of the above

You can also use web elements that have contentEditable attributes as a target element for TestCafe user actions.

Used in the following functions: [act.click](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actclick), [act.rclick](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actrclick), [act.dblclick](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actdblclick), [act.drag](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actdrag), [act.hover](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#acthover), [act.type](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#acttype)

**Mouse action options**

Provides additional information about the mouse action.

* {
* ctrl: false | true,
* alt: false | true,
* shift: false | true,
* meta: false | true,
* offsetX: Integer,
* offsetY: Integer,
* caretPos: Integer
* }

**ctrl, alt, shift, meta** - Boolean (optional). Indicates which modifier keys are to be pressed during the mouse action. By default, none will be pressed.  
**offsetX, offsetY** - Integer (optional). Mouse pointer coordinates set relative to the top-left corner of the target element. If not specified, TestCafe will move the mouse pointer to the center of the target element.  
**caretPos** - Integer (optional). Specifies the target caret position if the action is performed on a text input field. If not specified, the caret will be set to the end of the content of the input field.

Used in the following functions: [act.click](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actclick), [act.rclick](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actrclick), [act.dblclick](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actdblclick), [act.drag](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#actdrag), [act.hover](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions#acthover)

**Typing action options**

Specifies the options that provide additional information about a typing operation.

* {
* replace: false | true,
* ctrl: false | true,
* alt: false | true,
* shift: false | true,
* meta: false | true,
* offsetX: Integer,
* offsetY: Integer,
* caretPos: Integer
* }

**replace**: Boolean (optional). **true** to remove the current text in the target element, and **false** to leave the text as is. The default is **false**.  
**ctrl, alt, shift, meta**: Boolean (optional). Indicates which modifier keys are to be pressed during the mouse action. By default, none will be pressed.  
**offsetX, offsetY**: Integer (optional). Specifies the mouse pointer coordinates relative to the target element's top-left corner. The element will be clicked in this position to receive input focus. If these parameters are not specified, TestCafe will move the mouse pointer to the center of the target element.  
**caretPos**: Integer (optional). Specifies where to set the caret before typing. If not specified, the caret will be set to the end of content of the input field.

Used in functions: [act.type](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/User_Actions" \l "acttype)

**containsExcludeChildren() Selector**

Selects all elements that contain the specified text. Text within child nodes is ignored.

* $(':containsExcludeChildren(text)')

The following example shows how to use this selector to identify checkboxes. ([TestCafe Example Page](https://testcafe.devexpress.com/Example))

* '@test'['Click Checkboxes and Check State'] = {
* 'Click two checkbox labels': function () {
* var labels = [$(':containsExcludeChildren(Support for testing on remote devices)'),
* $(':containsExcludeChildren(Reusing existing JavaScript code for testing)')
* ];
* act.click(labels);
* },
* 'Confirm checked state': function () {
* ok($('#testing-on-remote-devices').is(':checked'));
* ok($('#re-using-existing-javascript').is(':checked'));
* }
* };

**Important note**

When TestCafe searches for the specified text content, it clears every string it encounters of all characters that are not numbers, latin or cyrillic letters before comparing it to the provided string.

**attrRegExp() Selector**

Selects elements by checking whether the value of an attribute matches the specified regular expression.

* $(':attrRegExp(attributeName:/regExp/)')