EXAMPLES

**How to Run Tests on a Schedule**

This example shows how to run test tasks on a certain schedule and process the results. You can also download this example [from GitHub](https://github.com/DevExpress/TestCafe-Scheduler).

This sample uses the [Cron for Node.JS](https://github.com/ncb000gt/node-cron) library for scheduling.

Assume that the task schedule and [TestCafe settings](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/" \l "TestCafe_Settings) are stored in the config.json file as shown below.

* {
* "cronTime" : "0 \* \* \* \* \*",
* "testcafe" : {
* "controlPanelPort" : 1337,
* "servicePort" : 1338,
* "testsDir" : "D:\\TestCafe\\my\_tests",
* "browsers" : {
* "Mozilla Firefox": {
* "path" : "C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe",
* "cmd" : "-new-window"
* }
* }
* }
* }

The **cronTime** property uses the [Cron Pattern](http://help.sap.com/saphelp_xmii120/helpdata/en/44/89a17188cc6fb5e10000000a155369/content.htm) to specify the schedule.

The following code reads the configuration file, schedules the task and logs results to a file.

* var fs = require('fs'), // Requests a reference to the Node.js file system.
* CronJob = require('cron').CronJob, // Requests a reference to the Cron module.
* TestCafe = require('testcafe').TestCafe; // Requests a reference to the TestCafe module.
* var CONFIG\_FILE\_NAME = 'config.json',
* LOG\_FILE\_NAME = 'log.txt';
* var croneJob = null,
* testCafe = null;
* getConfig(function(config) {
* // Creates a new TestCafe instance and specifies startup options for TestCafe
* // using a JSON object with the same notation as in the TestCafe configuration file.
* testCafe = new TestCafe (config.testcafe);
* // Sets a watch on changes in the configuration file.
* fs.watchFile(CONFIG\_FILE\_NAME, function() {
* restartCronJob();
* });
* // Logs the results in the console.
* console.log('\nTest will be started by pattern: ' + config.cronTime + '.');
* console.log('Test results you can find in the log.txt file.');
* runCronJob();
* });
* // Reads the configuration file and parses settings from JSON.
* function getConfig(callback) {
* fs.readFile(CONFIG\_FILE\_NAME, function(err, json) {
* if (err)
* throw err;
* callback(JSON.parse(json));
* });
* }
* function restartCronJob() {
* // Stops the Cron job.
* croneJob.stop();
* // Runs the Cron job again.
* runCronJob();
* }
* function runCronJob() {
* getConfig(function(config) {
* // Creates a new Cron job for the Node.js instance.
* croneJob = new CronJob({
* cronTime: config.cronTime, // Cron pattern.
* onTick: function() {
* runTests(); // This function fires at the specified time.
* },
* start: true
* });
* });
* }
* function runTests() {
* var runOptions = {
* workers: testCafe.listConnectedWorkers(), // Returns an array of strings identifying connected remote workers.
* browsers: testCafe.listAvailableBrowsers(), // Returns an array of strings identifying available browsers.
* emulateCursor: true // Enables emulation of cursor movements while tests are running.
* };
* // Runs tests and saves the results to the log file.
* testCafe.runTests(runOptions, function () {
* testCafe.on('taskComplete', function (report) {
* log('\n' + new Date().toString() + ':\n' + JSON.stringify(report));
* });
* });
* }
* // Saves test run results to a log file, which is created if it does not exist.
* function log(mssg) {
* fs.appendFile(LOG\_FILE\_NAME, mssg, function (err) {
* if (err)
* throw err;
* });
* }

**How to Run Tests with Arbitrary Data**

This example shows how to pass different data to a test each time it runs (not to be confused with [test cases](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Test_Parameterization) that provide a fixed set of parameters).

This sample uses the [Faker.js](https://github.com/Marak/Faker.js) library to generate random data.

The following test enters a random name and comment on the TestCafe demo page.

* "@fixture Fake.JS example";
* "@page http://testcafe.devexpress.com/Example";
* "@require ../modules/Faker.js";
* var randomName = Faker.Name.findName();
* var randomText = Faker.random.catch\_phrase\_descriptor();
* "@test"["Example with random data"] = {
* "Type name": function() {
* act.type($("#Developer\_Name"), randomName);
* },
* "Click 'I have tried TestCafe'": function() {
* var div = $(".mask").eq(6);
* act.click(div);
* },
* "Type in text area 'Please let us know what you think:'": function() {
* var textArea = $("#Developer\_Comments");
* act.type(textArea, randomText);
* }
* };

Instead of the Faker.js library, you can use any custom library to provide data according to your task.

**How to Log Results in the Console or a File**

This example shows how to log test results on a console and save them to a file.

To log results on a console, use the Node.JS [console.log](http://nodejs.org/api/console.html#console_console_log_data) method as shown in the following sample.

* testCafe.runTests(runOptions, function () {
* testCafe.on('taskComplete', function (report) {
* console.log(report);
* });
* });

To save a test report to a file, reference the [NodeJS file system](http://nodejs.org/api/fs.html) module. Use the [fs.appendFile](http://nodejs.org/api/fs.html" \l "fs_fs_appendfile_filename_data_options_callback) method to add a new entry to the log.

* var fs = require('fs');
* testCafe.runTests(runOptions, function () {
* testCafe.on('taskComplete', function (report) {
* log('\n' + new Date().toString() + ':\n' + JSON.stringify(report));
* });
* });
* function log(mssg) {
* fs.appendFile(LOG\_FILE\_NAME, mssg, function (err) {
* if (err)
* throw err;
* });
* }

**How to Control TestCafe Through the Network (RPC)**

This example shows how to create a single TestCafe instance and then interact with it programmatically on any machine in your network.

This sample uses the [TestCafe RPC](https://npmjs.org/package/testcafe-rpc) module, which implements RPC support. To install it, execute the following command.

npm install testcafe-rpc

The code snippet below illustrates how to run a server using RPC.

* var TestCafeRemote = require('testcafe-rpc');
* // Defines TestCafé options
* var opt = {
* controlPanelPort: 1337,
* servicePort: 1338,
* testsDir: 'YOUR\_TESTS\_DIR',
* browsers: {
* 'Mozilla Firefox': {
* path: "C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe",
* icon: "ff",
* cmd: "-new-window"
* }
* }
* },
* rpcPort = 1339;
* // Creates a TestCafé server instance with provided options.
* // The created instance can be accessed through RPC on port 1339.
* var testCafeServer = new TestCafeRemote.Server(opt, rpcPort);
* // The returned object exposes the standard TestCafé API, so you can use it as a regular TestCafé instance.
* testCafeServer.runTests({ browsers: testCafeServer.listAvailableBrowsers() }, function (errors, taskUid, workers) {
* // do smth...
* });

The following code shows how to run a client.

* var TestCafeRemote = require('testcafe-rpc');
* var rpcHostname = 'myhostname',
* rpcPort = 1339;
* // Connects to an existing TestCafé server (if you are running both the client and server on the same machine,
* // the 'hostname' parameter can be omitted).
* var testCafeClient = new TestCafeRemote.Client(rpcPort, rpcHostname);
* // The client can be used as a regular TestCafé instance with an exception that listAvailableBrowsers()
* // and listConnectedWorkers() are asynchronous methods.
* testCafeClient.listAvailableBrowsers(function(browsers) {
* testCafeClient.runTests({ browsers: browsers }, function (errors, taskUid, workers) {
* // do smth...
* });
* });

**How to Use TestCafe Wrappers in Tests**

[TestCafe wrappers](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts/#Wrappers) help make your test code readable and stable.

This example demonstrates how you can use the wrappers to create a test for a master-detail ASPxGridView control.

1. First, download the wrappers and define them in the modules section of the [directory configuration file](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts#Directory_Configuration_File). For detailed information on how to do this, see [TestCafe wrappers](https://testcafe.devexpress.com/Documentation/API_Reference/Test_Fixture_API/Common_Concepts/" \l "Wrappers).
2. Record a test for the [ASPxGridView: Master-Detail](https://demos.devexpress.com/ASPxGridViewDemos/MasterDetail/MasterDetail.aspx) demo page using the [TestCafe recorder](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Visual_Test_Recorder/). The recorder will generate appropriate selectors and actions.
3. Modify the test logic with the [TestCafe editor](https://testcafe.devexpress.com/Documentation/Using_TestCafe/Control_Panel/" \l "Using_TestCafe_Control_Panel_Projects_Tab_Fixtures_Test_Code_Editing). You can invoke the editor by clicking the Pencil Icon button within the Control Panel. You can also modify your test using any editors including Visual Studio, Sublime Text or even Notepad.
4. Reference the wrapper files in your test using the **@require** directive:
   1. "@require :dx";
5. Replace jQuery selectors with corresponding [ASPxGridViewWrapper functions](https://github.com/DevExpress/TestCafe-Wrappers" \l "ContentsAPI).

You can also review [this video](https://www.youtube.com/watch?v=oiIf2xigrjQ) that demonstrates how to perform all the described steps and see the test run result.

Here is a complete sample to test the ASPxGridView: Master-Detail demo page:

* "@fixture Master Detail test";
* "@page https://demos.devexpress.com/ASPxGridViewDemos/MasterDetail/MasterDetail.aspx";
* "@require :dx";
* "@test"["master-detail test"] = {
* '1.Check if a detail row visible': function() {
* var detailDataRowsCount = getGrid().getDetailRows().length;
* eq(detailDataRowsCount, 1);
* },
* '2. Expand the second row': function() {
* act.click(getGrid().getDetailButtons()[1]);
* },
* '3.Check if two detail grids are visible': function() {
* var detailDataRowsCount = getGrid().getDetailRows().length;
* eq(detailDataRowsCount, 2);
* },
* '4.Collapse the first row': function() {
* act.click(getGrid().getDetailButtons()[0]);
* },
* '5.Collapse the second row': function() {
* act.click(getGrid().getDetailButtons()[1]);
* },
* '6.Check if there is no visible detail grids': function() {
* var detailDataRowsCount = getGrid().getDetailRows().length;
* eq(detailDataRowsCount, 0);
* },
* '7.Click the "Keep a single expanded row at a time" check box': function() {
* act.click("#ContentHolder\_chkSingleExpanded\_S\_D");
* },
* '8.Expand the first row': function() {
* act.click(getGrid().getDetailButtons()[0]);
* },
* '9.Expand the second row': function() {
* act.click(getGrid().getDetailButtons()[1]);
* },
* '10.Check the "Allow single row to be expanded" property': function() {
* var detailDataRowsCount = getGrid().getDetailRows().length;
* eq(detailDataRowsCount, 1);
* }
* };
* // helpers
* var getGrid = function(){
* return dx.grid("grid");
* };