

How to Optimize Data-driven Automation with Test Case Designer



What you'll learn

- Why data-driven automation can be a good choice
- Benefits of data-driven automation with Test Case Designer
- Prerequisites
 - Creation of compatible scripts in Playwright and Cypress
- How to do Validation of data-driven Xray reports in Jira
 - Common steps
 - Integrate Test Case Designer with Playwright
 - Build a TCD model and export scenarios
 - Create a script in Playwright
 - Import results to Xray
 - Integrate Test Case Designer with Cypress
 - Build a TCD model and export scenarios
 - Create a script in Cypress
 - Import results to Xray

Overview

• Tips

Why data-driven automation can be a good choice



Definition

Data-driven testing is the practice of keeping the execution script separate from the test data. In other words, you will have a single automation script that will reference a data table containing multiple rows with scenarios.

When we talk about prioritizing tests for automation, it often comes down to high repetitiveness of actions and high number of iterations. Data-driven testing fits that pattern well and helps with some of the adoption barriers:

1. It removes the need to create an individual script for each test case, which is typically a time-consuming process. But often more importantly, it simplifies and accelerates maintenance because most changes to the test script do not affect the test data, and vice versa. Scaling execution up or down is also more flexible.



As an added bonus, if you need to switch frameworks/languages, your data sources would still be valid, and not as many scripts would need to be replaced.

2. It can make collaboration easier since reviewing a data table does not require special technical skills. That, in turn, accelerates the overall process (since any subject matter expert can add iterations) and increases test data accuracy. The reporting can be set up to clearly reflect the status of each iteration, just as it would for separate scripts.

Of course, data-driven automation is not the universal solution, a few of its limitations are:

1. Skill level requirements - if each iteration has enough "uniqueness" about its actions & validations or complexity of the expected result, trying to fit numerous iterations into 1 script will result in the overload of conditional statements. That would require a lot more automation experience.
2. Shifting some of the problems to a different medium - manual creation & maintenance of data sources can be only marginally less cumbersome than those of individual scripts. Data relevancy and accuracy can still be large concerns.

For the first limitation, the collaborative analysis of the draft data tables should validate the pattern early on and properly choose the targets for data-driven automation, without excessively complicating the overall implementation. For the avoidance of doubt, you will likely always have a mix of automation approaches, where data-driven testing is one part.

For the second one, we can find a better alternative to manual data source management. For example, Xray Enterprise Test Case Designer (TCD).

Why data-driven automation with Test Case Designer can be an even better choice

- Faster data source creation - designing the table by hand can't compete with the algorithmic generation when it comes to speed, especially for complex systems.
- More efficient maintenance - changes at the model level are automatically applied to all affected test cases.
- Minimized redundancy & maximized thoroughness - the TCD algorithm makes sure the data scenarios are as varied as possible while providing flexible coverage goals. That often leads to earlier detection of defects.
- Diverse collaboration artifacts - in addition to the data table, you can leverage auto-generated visualizations like mind map and coverage matrix to make sure all stakeholders are on the same page about the testing scope.



If at any point you need to switch to the “traditional” script-based approach, you can easily do it by leveraging the data-driven BDD/Robot Framework templates under Scripts. Create 1 Scenario/Test Case block inside TCD, export or sync it to as many issues as you have iterations. You don't need to rebuild your models, and you still benefit from all 4 points above.

So let's dive into how such optimized data-driven automation can be accomplished.

Prerequisites

For Playwright, in order to run this tutorial you need to have [Nodejs](#) and [install the playwright-test runner](#).

For Cypress, you need to have [Cypress installed](#).

How to do it

Common steps

We will discuss the implementation in 2 frameworks, with different complexity of the target apps. At a high level, steps are similar for each example:

1. Create the setup in Jira with the requirement/story and the Xray Test Plan.
2. Create a model for the target system in TCD.
 - a. can add the model URL to the requirement story.
3. Export the data table as CSV.
4. Create the data-driven execution script in the selected framework/language.
5. Run that code and generate the report format that is supported by Xray.
 - a. optionally, configure the level of evidence in the report and add links to the requirement and/or test plan from step 1.
6. Import the report file via REST API (directly or through a CI/CD tool).
7. Enjoy full visibility of the results in Xray.

Integrate Test Case Designer with Playwright



For Playwright versions 1.34 and later, if you want to attach screenshots, you will need to additionally [install this Playwright reporter](#) and change the config file or the command line statement accordingly.

The scope of this tutorial is:

- passing different combinations of data to the calculator app at <https://treasurydirect.gov/BC/SBCGrw>

Growth Calculator

Feel free to change the default values below. Then, click the "calculate" button to see how your savings add up! For more information, click the instructions link on this page.

This calculator is for estimation purposes only.

GROWTH CALCULATOR

Initial Investment Amount:

\$1,000

Expected Interest Rate:

2.00%

Periodic Investment:

\$200.00

How Often:

Monthly

Years Invested:

25

Your Federal Tax Rate:

28%

Help

CALCULATE

- validating that the evaluation results are returned successfully (by checking the presence of the title on the page, since we don't know the actual formula behind the app)

Build a TCD model and export scenarios

The model in TCD looks like this (a fairly straightforward approach of a parameter per UI field):

InitialInvestment (\$)	100 - 5000	5001 - 15000	15001 - 50000	100000
Inflate (%)	min	med	max	
PeriodicInv (\$)	\$0.00	small	medium	large
Frequency (%)	Weekly	Bi-Weekly	Monthly	Annually
YearsInvested (Y)	min	med	max	
FedTaxRate (%)	min	med	max	

Note that:

- Shorter parameter names without spaces or special characters work better.
- The spelling of value names should match the syntax expected by the automation script. You can share your draft TCD model with the engineers to verify that.
- For data table and Scripts->Automate exports, only value expansion text will be populated, so we can still benefit from the equivalence class approach for, e.g., IntRate or FedTaxRate, without worrying about the value name syntax.

After generating Scenarios, we can export the CSV data table from the same screen:

Parameters

5 scenarios and 23 1-way interactions

Freeze these Scenarios

Save as...

1 100 1.00% - min \$0.00 Weekly 1 - min 0% - min

2 5001 6.00% - med \$17.50 - small Bi-Weekly 25 - max 24.5% - med

3 15001 7.00% - max \$100.00 - medium Monthly 3 - med 50.1% - max

4 100000 2.00% - med \$1,000.00 - large Annually 2 - min 0% - min

5 5000 8.00% - max \$10,000.00 - max Semi-Annually 2 - min 50.1% - max

Save as...

CSV

Generate Data Table

Generate Data Table (JSON)

XML

HTML

As the automation is being developed, it may make sense to leverage the "1-way interactions" goal for the TCD algorithm. It exposes the execution to each possible value with the smallest number of iterations, which helps with the feedback speed.

You can [import the model](#) and try this process yourselves:



TCD_Demo_-_Gov...ent-1-way.xlsx

Create a script in Playwright

On the Playwright side, we first adjust the configuration file to support Xray options with the appropriate Junit report formatting ("const xrayOptions" and "reporter" sections):

playwright.config.ts

```
import { defineConfig } from '@playwright/test';

/* JUnit reporter config for Xray https://playwright.dev/docs/test-reporters#junit-reporter */
const xrayOptions = {
  embedAnnotationsAsProperties: true,
  textContentAnnotations: ['test_description'],
  embedAttachmentsAsProperty: 'testrun_evidence',
  outputFile: './TCDDemoReport.xml'
};

export default defineConfig({
  reporter: [
    ['line'],
    ['junit', xrayOptions]
  ],
  testDir: '.',
  outputDir: './test-results',
  timeout: 50 * 1000,
  expect: {
    timeout: 10000
  },
  fullyParallel: false,
  forbidOnly: !!process.env.CI,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? 1 : undefined,
  use: {
    actionTimeout: 0,
    headless: false,
    viewport: { width: 1280, height: 720 },
    ignoreHTTPSErrors: true,
    video: 'on-first-retry',
    screenshot: 'on-first-retry',
    trace: 'on-first-retry',
  },
  projects: [
    {
      name: 'chromium',
      use: { ...devices['Desktop Chrome'] },
    },
  ],
});
```

Then implement the driver script:

TCDDemo.spec.ts

```
import { test, expect } from '@playwright/test';
import fs from 'fs';
import path from 'path';
import { parse } from 'csv-parse/sync';

const records = parse(fs.readFileSync(path.join(__dirname, 'TCDDemo.csv')), {
  columns: true,
  skip_empty_lines: true
});

for (const record of records) {
  test(`${record.Summary}`, async ({ page }, testInfo) => {
    testInfo.annotations.push({ type: 'requirements', description:
'TAC-117' });
    testInfo.annotations.push({ type: 'test_description', description:
`Confirm the calculation is successful for Initial Amount = ${record.
InitialInvAmount}, IntRate = ${record.IntRate}, Periodic Investment =
${record.PeriodicInv},
    Frequency = ${record.Frequency}, Years Invested = ${record.
YearsInvested}, Federal Tax Rate = ${record.FedTaxRate} `});

    console.log(record.Summary, record.InitialInvAmount, record.
IntRate, record.PeriodicInv, record.Frequency, record.YearsInvested,
record.FedTaxRate);

    await page.goto('https://treasurydirect.gov/BC/SBCGrw');
    await expect(page.getByRole('heading', { name: 'Growth Calculator'
})).toBeVisible();

    await page.locator('input[name="InitInvest"]').fill(record.
InitialInvAmount);
    await page.locator('select[name="IntRate"]').selectOption({label:
record.IntRate});
    await page.locator('select[name="PeriodicPmt"]').selectOption
({label:record.PeriodicInv});
    await page.locator('select[name="PmtFrequency"]').selectOption
({label:record.Frequency});
    await page.locator('input[name="Years"]').fill(record.
YearsInvested);
    await page.locator('input[name="FedTax"]').fill(record.FedTaxRate);

    const completeinputs = await page.screenshot();
    await page.getByRole('button', { name: 'CALCULATE' }).click();
    await expect(page.getByRole('heading', { name: 'Growth Calculator
Results' })).toBeVisible();
    const calcresults = await page.screenshot();
    await page.getByRole('button', { name: 'Back to Growth Calculator'
}).click();

    await testInfo.attach('completeinputs', { body: completeinputs,
contentType: 'image/png' });
    await testInfo.attach('calcresults', { body: calcresults,
contentType: 'image/png' });

  });
};
```

The core detail about this approach is, using the extra libraries, we first read the csv from TCD, then iterate through it with the “for” loop:

CSV Handling

```
...
import fs from 'fs';
import path from 'path';
import { parse } from 'csv-parse/sync';

const records = parse(fs.readFileSync(path.join(__dirname, 'TCDDemo.csv')), {
  columns: true,
  skip_empty_lines: true
});

for (const record of records) {
  ...
};
```

The parameters (aka column names from the CSV, e.g. `record.InitialInvAmount`) are passed in typical Playwright methods (like `fill` or `selectOption`) instead of hardcoding text.

Parameterized steps

```
...
    await page.locator('input[name="InitInvest"]').fill(record.InitialInvAmount);
    await page.locator('select[name="IntRate"]').selectOption({label: record.IntRate});
    await page.locator('select[name="PeriodicPmt"]').selectOption({label: record.PeriodicInv});
    await page.locator('select[name="PmtFrequency"]').selectOption({label: record.Frequency});
    await page.locator('input[name="Years"]').fill(record.YearsInvested);
    await page.locator('input[name="FedTax"]').fill(record.FedTaxRate);
...

```

To improve the clarity and traceability, we can add a link to the requirement story, parameterized name & description, and custom screenshots.

Customization

```
...
  test(`${record.Summary}`, async ({ page }, testInfo) => {
    testInfo.annotations.push({ type: 'requirements', description:
'TAC-117' });
    testInfo.annotations.push({ type: 'test_description', description:
`Confirm the calculation is successful for Initial Amount = ${record.
InitialInvAmount}, IntRate = ${record.IntRate}, Periodic Investment =
${record.PeriodicInv},
    Frequency = ${record.Frequency}, Years Invested = ${record.
YearsInvested}, Federal Tax Rate = ${record.FedTaxRate} `});
  });
...

  const completeinputs = await page.screenshot();
  const calcresults = await page.screenshot();
...
  await testInfo.attach('completeinputs', { body: completeinputs,
contentType: 'image/png' });
  await testInfo.attach('calcresults', { body: calcresults,
contentType: 'image/png' });
}
```

Import results to Xray

Once the script [completes the execution](#), we can import the results via, e.g., curl (parts in {} should be customized based on your details):

Command Line Import

```
curl -H "Content-Type: multipart/form-data" -u {username}:{password} -F
"file=@{ReportName}.xml" "https://{JiraURL}/rest/raven/1.0/import/execution
/junit?projectKey={ProjectKey}&testPlanKey={TestPlanIssueKey}"
```



CI/CD

For the CI/CD process, please refer to this collection of tutorials - [DC/Server](#) ; [Cloud](#)

As a result, you should see five Test issues (linked to the requirement story and to the Test Plan issue) and a Test Execution issue (with the screenshots under Execution details of each run):

Overall Execution Status View on Board

5 PASS

Total Tests: 5

15 Filters

Apply Rank

Show 10 entries Columns

Rank	Key	Summary	Test Type	#Req	#Def	Test Sets	Assignee	Dataset	Status	
5	TAC-145	Calculation_5,Semi-Annually	Generic	1	0		Ivan Filippov		PASS	▶
4	TAC-144	Calculation_4,Annually	Generic	1	0		Ivan Filippov		PASS	▶
3	TAC-143	Calculation_3,Monthly	Generic	1	0		Ivan Filippov		PASS	▶
2	TAC-142	Calculation_2,Bi-Weekly	Generic	1	0		Ivan Filippov		PASS	▶
1	TAC-141	Calculation_1,Weekly	Generic	1	0		Ivan Filippov		PASS	▶

Integrate Test Case Designer with Cypress

The scope of this tutorial is:

- providing different combinations of applicant data to the web mortgage quoting system at <https://www.wellsfargo.com/mortgage/calculators/>.

- The flow starts with selecting “Buy a home” as the purpose, then entering location, amount, and credit rating details. After the initial results, you can adjust the term.

Mortgage Calculator

Mortgage interest rates can vary based on your circumstances. Use our mortgage rate calculator to get customized rates and monthly mortgage payments.

Find your estimated rate

What's the purpose of your loan?

Buy a home

- checking the app's ability to return the loan evaluation results successfully by asserting the page title (without looking at the specific numbers).

Build a TCD model and export scenarios

The model in TCD looks like this:

Purpose (1)	Buy a home			
Location (2)	Lower rates	Higher rates		
PurchasePrice (5)	10000	15000 - 100000	110000 - 500000	750000 - 5000000
DownpaymentPercentage (5)	below 3%	3	4-19	20
NumberOfPeople (3)	1toBorrowers	creditBorrower	badBorrowers	
creditRating (6)	excellentCredit	goodCredit	fairCredit	poorCredit
Term (6)	0	1	2	3

Note that:

- It's not wrong to include 1-value parameters for test data rationale (like “Purpose”) if you feel like it's better than hardcoding them in the script (often because you expect that parameter to grow over time).
- For some web apps, selection by labels can be too complicated, so you may have to switch to indices (e.g. “Term” values). Refer to the feedback from your automation team.
- This example showcases how to deal with conditional steps.

“If the DownpaymentPercentage is 20 or higher, then an additional question about the number of borrowers doesn't appear”. On the TCD side, you can have:

- either a “special” value (like “NABorrowers” for “NumberOfPeople”) with the corresponding set of constraints

```

3 DownpaymentPercentage [ below 3% ] → PurchasePrice [ 10000 ]
4 DownpaymentPercentage [ below 3% ] ≠ NumberOfPeople [ NABorrowers ]
5 DownpaymentPercentage [ 3 ] ≠ NumberOfPeople [ NABorrowers ]
6 DownpaymentPercentage [ 4-19 ] ≠ NumberOfPeople [ NABorrowers ]
7 DownpaymentPercentage [ 20 ] → NumberOfPeople [ NABorrowers ]
8 DownpaymentPercentage [ 21 - 50 ] → NumberOfPeople [ NABorrowers ]

```

- or a “skip” constraint (DownpaymentPercentage[20,21 - 50] >> NumberOfPeople)

The choice will determine which trigger will be assigned to the automation step that we will show later.

You can [import the model](#) and try this process yourselves:



TCD_Demo_-_Fina...scenarios.xlsx

Create a script in Cypress

On the Cypress side, you will also first need to adjust the package and the config files to support csv handling with 3 imports:

Package.json

```
{
  "type": "module",
  "devDependencies": {
    "cypress": "^12.9.0",
    "cypress-if": "^1.10.4",
    "mocha": "^10.2.0",
    "mochawesome": "^7.1.3",
    "mochawesome-merge": "^4.3.0",
    "mochawesome-report-generator": "^6.2.0"
  },
  "dependencies": {
    "cypress-mochawesome-reporter": "^3.3.0",
    "neat-csv": "^7.0.0",
    "node-fetch": "^2.6.1"
  }
}
```

cypress.config.js

```
import { defineConfig } from 'cypress'

import fs from 'fs'
import path from 'path'
import neatCSV from 'neat-csv'

export default defineConfig({
  e2e: {
    async setupNodeEvents (on, config) {
      const filename = './cypress/fixtures/TCDDemo.csv'
      console.log('loading file', filename)
      const text = fs.readFileSync(filename, 'utf8')
      const csv = await neatCSV(text)
      console.log('loaded the quote data')
      console.log(csv)

      config.env.TCDDemo = csv
      return config
    },
  },
})
```

The optimized data table can go into cypress/fixtures while the execution script in cypress/e2e would look like this:

TCDDemoScript.cy.js

```
const tcdloaddata = Cypress.env('TCDDemo')
describe('Quote a mortgage', () => {
  tcdloaddata.forEach((l) => {
    it(`With ${l.PurchasePrice} purchase price and ${l.
DownpaymentPercentage} downpayment for ${l.NumberofPeople}`, () => {
      cy.visit('https://www.wellsfargo.com
/mortgage/calculators/')

      cy.contains(l.Purpose).click()
      cy.get('#label-locationField').click()
      cy.get('#locationField').type(l.Location)
      cy.contains('Next').click()
      cy.get('#label-purchasePriceAmount').

click()

      cy.get('#purchasePriceAmount').type(l.
PurchasePrice)
      // the location suggestions window sometimes persists through all the
subsequent steps, so added {force: true} to ignore it
      cy.contains('Next').click({force: true})
      cy.get('#downPaymentPercent').clear()
      cy.get('#downPaymentPercent').type(l.
DownpaymentPercentage)

      cy.contains('Next').click({force: true})

      cy.get("body").then($body => {
        if ($body.find(`#${l.
NumberofPeople}`).length > 0) {
          //evaluates as true if button
exists at all

          cy.get(`#${l.
NumberofPeople}`).then($btn => {
            if ($btn.is(':
visible')){
              //you get here
              cy.wrap($btn).click
({force: true})
            }
          })
        }
      })
    })
  })
})
```

```

                                } else {
                                //you get here

                                cy.log('Question
                                about borrowers did not appear')

                                }
                                });
                                else {
                                //you get here if

                                cy.log('Question
                                about borrowers did not exist on the page')

                                assert.isOk

                                }

                                })

                                cy.get(`#${l.creditRating}`).click({force:
true})

                                // form
                                cy.get('#label-aboutYourselfFirstName').
click()

                                cy.get('#aboutYourselfFirstName').type('I')
                                cy.get('#label-aboutYourselfLastName').
click()

                                cy.get('#aboutYourselfLastName').type('F')
                                cy.get('#label-aboutYourselfEmail').click()
                                cy.get('#aboutYourselfEmail').type

('test@test.org')

                                cy.get('#label-aboutYourselfPhone').click()
                                cy.get('#aboutYourselfPhone').type

('1111111111')

                                cy.get('#label-aboutYourselfZipCode').
click()

                                cy.get('#aboutYourselfZipCode').type

('28262')

                                cy.get('#continue-view-rates').click

                                cy.get('#consent-submit-click').click

                                cy.get('#RST_Results_LoanTerm').select(1.
Term)

                                cy.get('#skip').should('have.text',
'\n\t\t\t\tCustomized Mortgage Rate Results\n\t\t\t\t')

                                })
                                })
                                })

```

Similarly to the Playwright example, we will first read the CSV file, then iterate the script through it with the “forEach” loop.

TCDDemoScript.cy.js

```

const tcdloandata = Cypress.env('TCDDemo')
...
tcdloandata.forEach((l) => {
...
})

```

Keep in mind that the final Test Issue naming convention in Xray will concatenate these lines:

TCDDemoScript.cy.js

```
describe('Quote a mortgage', () => {
  ...
  it('With ${l.PurchasePrice} purchase price and ${l.
DownpaymentPercentage} downpayment for ${l.NumberofPeople}', () => {
    ...
  })
})
```

So you need to split the desired summary accordingly while parameterizing the second line.

The code below is responsible for the conditional logic we mentioned earlier around Downpayment Percentage and Number of Borrowers.

TCDDemoScript.cy.js

```
...
      cy.get("body").then($body => {
        if ($body.find(`#${l.
NumberofPeople}`).length > 0) {
          cy.get(`#${l.
NumberofPeople}`).then($btn => {
            if ($btn.is(':
visible')){
              cy.wrap($btn).click
            } else {
              cy.log('Question
about borrowers did not appear')
            }
          })
        } else {
          cy.log('Question
about borrowers did not exist on the page')
          assert.isOk
        }
      })
    ...
  })
}
```

It searches for the dynamically defined element (``#${l.NumberofPeople}``), checks whether it's present and visible, and, if yes, clicks it, otherwise, logs a message. In this implementation of the TCD model, `'#NABorrowers'` element would never be found, so the automation skips the step whenever that value comes from the CSV.



- If you know the exact behavior of the element (visibility vs presence), that conditional logic can be simplified.
- If the "skip" constraint route is taken, the same code could work. Or you may need to adjust the dynamic element syntax if `'#_'` is treated as undefined/syntax error.

Import results to Xray

The results import process would be identical to the Playwright one. You will notice that 2 tests are failing as the results are not returned:

Overall Execution Status

3 PASS
 2 FAIL

Total Tests: 5

Filter(s)

Apply Rank

Show 10 entries Columns

Rank	Key	Summary	Test Type	PRQ	PRQ2	Test Sets	Assignee	Dataset	Status
5	TAC-225	Quote a mortgage With 4000000 purchase price and 0 downpayment for twoborrowers	Generic	0	0		Ivan Filippov		FAIL
4	TAC-234	Quote a mortgage With 700000 purchase price and 21 downpayment for twoborrowers	Generic	0	0		Ivan Filippov		FAIL
3	TAC-223	Quote a mortgage With 110000 purchase price and 20 downpayment for twoborrowers	Generic	0	0		Ivan Filippov		PASS
2	TAC-222	Quote a mortgage With 10000 purchase price and 3 downpayment for twoborrowers	Generic	0	0		Ivan Filippov		PASS
1	TAC-221	Quote a mortgage With 10000 purchase price and 1 downpayment for oneborrower	Generic	0	0		Ivan Filippov		PASS

This is a decent example of the execution feedback loop for combinatorial test design: we will often exercise the combinations that haven't been explicitly mentioned in requirements. Since all individual values are within valid ranges (based on UI tooltips), we would review the scenario as a whole with the stakeholders and confirm the expected behavior. (You can notice that both failures share at least 1 similarity: the purchase price is close to the upper limit.)

If the lack of results is valid, we can make the assertion conditional on data in each iteration or adjust the values/constraints in the TCD model to focus more on the combinations that return rates.

Tips

- Generating CSV from the Scenarios screen in TCD results in only value expansion names being included, where applicable. The CSV option on the Export screen is geared more towards manual execution and will produce "Value name - Value expansion name" syntax.
- To create script steps in Playwright, if you are not yet familiar with the syntax, you can use the [CodeGen feature](#).
- The csv processing for Cypress recommends [neat-csv package](#). Depending on the version, you need to switch the Cypress config and package files between CommonJS and ES Module scripting. Our example uses ES Module. Or you can try an alternative package.

References

- Testing web applications using Playwright
- <https://playwright.dev/docs/test-reporters#junit-reporter>
- <https://playwright.dev/docs/test-parameterize#create-tests-via-a-csv-file>
- Testing web applications using Cypress
- <https://docs.cypress.io/examples/recipes>
- <https://learn.cypress.io/advanced-cypress-concepts/using-data-to-build-dynamic-tests>
- https://github.com/cypress-io/cypress-example-recipes/tree/master/examples/fundamentals__dynamic-tests-from-csv