Test4z Test Drive

Note: This is a 15-minute version of the full [Test4z Tutorial](). Look for the "Try it ✎" suggestions.

Like a lot of developers, you might think of testing as just more work. But with Test4z, you'll see in this exercise how it can actually speed up development! Test4z does this in two ways:

1. By reducing the need for manual testing.
2. By simplifying the setup of a valid and stable test environment using recordings, instead of the actual live environment.

Beyond the setup time savings for developers, with the benefit of automated testing, your organization saves time responding to regressions caused by less-than-complete testing. These two principles–isolation and automation–are fundamental to the Test4z ethos "the key to higher quality code is making it easier to test."

WHAT IS UNIT TESTING? HOW DOES TEST4Z WORK?

Unit testing in COBOL means validating the smallest testable parts of a program in *isolation*, which is crucial for test automation. When writing unit tests with Test4z, the developer/tester decides what's the smallest testable part and how much the unit test knows about the internal workings of the tested program.

The diagram below depicts the main actors in the Test4z runtime:
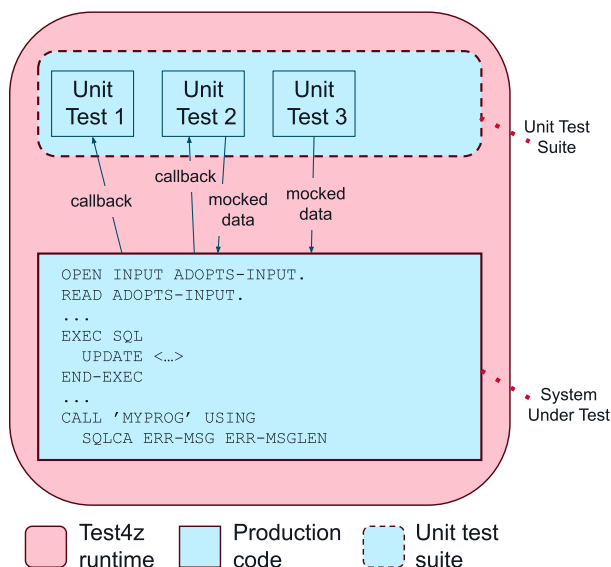
Test4z Test Drive

Your production code is the System Under Test in the lower box; it's a load module that can be used as-is, that is, there's no need to recompile it. The SUT is loaded by the Test4z runtime and then instrumented with callbacks to your unit test.

The new code you'll write is the *test suite* load module, shown in the upper dashed box, plus unit tests that execute some portion of the SUT. A test suite contains multiple unit tests defined by COBOL entry points. Once the SUT load module is ready, the Test4z test runner will execute each of the suite's unit test entry points, recording if they passed or failed.

The Test4z runtime provides an extensive API that you'll call in your unit tests. We'll review two that you'll use most often:

- **Mocks**: Simulated resources within your unit tests, replacing real dependencies with controlled versions to ensure consistent test outcomes.

- **Spies**: Code that captures interactions with resources, providing insights into how your application behaves with certain inputs and outputs.

Now that you have a better understanding of the Test4z components and how they work together, let's look at a real code example.

OVERVIEW OF TESTED PROGRAM AND UNIT TEST PROGRAM

This exercise requires two programs: The tested program ZTPDOGOS and the unit test program ZTTDOGWS.

- ZTPDOGOS - this is a simple application that reads from an input file and writes a report to an output file. The input file ADOPTS has this format:

```
01   ADOPTED-DOGS-REC.
       05 INP-DOG-BREED                PIC X(30).
       05 FILLER                       PIC X(25).
       05 INP-ADOPTED-AMOUNT           PIC 9(3).
       05 FILLER                       PIC X(22).
```

Below are example records written to the OUTREP report file:

```
BREED SHIBA                            WAS ADOPTED 008 TIMES
```

Test4z Test Drive

```
BREED SCHNAUZER                                WAS ADOPTED 000 TIMES
...
BREED OTHER                                    WAS ADOPTED 000 TIMES
```

The tested program keeps a running count of nine breeds in an internal `ACCUMULATOR` working storage variable that is used to produce the report totals:

```
01 ACCUMULATOR.
     05 BREED-ADOPTIONS PIC 9(3) OCCURS 9 TIMES VALUE 000.
```

As an example of gray box testing, your unit test program ZTTDOGWS will access the report program's working storage variable `ACCUMULATOR` at runtime to double-check its calculations.

- ZTTDOGWS - the unit test program that validates the operation of the report program.

The rest of this exercise will focus on the unit test code in ZTTDOGWS using Visual Studio Code. If you haven't started VS Code, please do so now.

HOW TO RUN A TEST4Z TEST SUITE

You can start a unit test multiple ways, such as using the Test4z command line interface (CLI) named `t4z`, a pop-up menu from the VS Code Explorer, or from the dedicated Testing view. The command line option to run the unit test program ZTTDOGWS is shown below:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

○ testdrive $ t4z test/ZTTDOGWS.cbl
```

Test4z uses Team Build to compile COBOL source found in the programs-under-test `src` folder and unit test suites in the `test` folder, execute them on the mainframe, then download the results to your VS Code workspace.

Test4z Test Drive

TEST4Z APIs AND CODE SNIPPETS

There are many Test4z APIs to build your unit test case. But don't worry about the specifics, since Test4z's Visual Studio Code extension adds code snippets. All you do is type `t4z` followed by the API name.
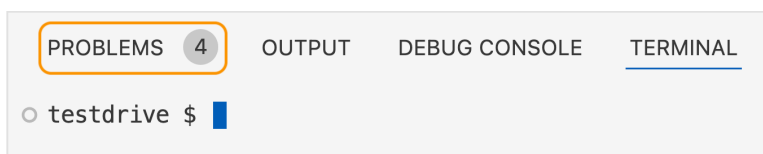
Try it ✎

Add a new Test4z API call in ZTTDOGWS (*not* ZTPDOGOS!) at the beginning of the PROCEDURE DIVISION using a code snippet. As soon as you type "t4z" in your source code followed by a few letters, a list of possible API matches is displayed.

For example, type "t4z mess" starting in column 12 and select `Message write` from the displayed list. Next, change the `messageText` to "Hello Test4z!". The final result is shown below:

```cobol
PROCEDURE DIVISION.

    move low-values to I_Message in ZWS_Message
    move 'Hello Test4z!' to messageText in ZWS_Message
    call ZTESTUT using ZWS_Message

****************************************************************
* Register test to be run (only one in this simple example).
****************************************************************
    move low-values to I_Test
    set testFunction in ZWS_Test to entry 'outrepTotalsUnitTest'
    move 'ZTTDOGWS simple totals test' to testName in ZWS_Test
    call ZTESTUT using ZWS_Test
```

Tip: Be sure that you have the [Cobol Language Support](#) extension installed. It will highlight syntax errors in the editor with "squiggles" under the problematic code.

```
PROBLEMS  4    OUTPUT    DEBUG CONSOLE    TERMINAL
○ testdrive $ ▌
```

The PROBLEMS tab summarizes the syntax errors that were found. Double-check there are no errors and then continue.

Test4z Test Drive

Try it ✎

Save your change and then use the t4z command to run the test (File > Terminal > New Terminal). By default, code coverage isn't generated, so append the `--cov` option to the `t4z` command as highlighted below:

```
PROBLEMS     OUTPUT     DEBUG CONSOLE     TERMINAL

○ testdrive $ t4z test/ZTTDOGWS.cbl --cov
```

The result in the Terminal will look something like this:

```
PROBLEMS     OUTPUT     DEBUG CONSOLE     TERMINAL     PORTS

 PASS   test/ZTTDOGWS.cbl
  ✓ ZTTDOGWS simple totals test (409 ms)

Tests Suites: 1 passed, 1 total
Tests:        1 passed, 1 total
Time:         0 s

Running tests completed successfully. For more details see files 'test-out/ZLMSG.txt' and 'test-out/SYSOUT.txt'.
Generating coverage report...

---------------|---------|----------|---------|---------|-------------------------------------------------
File           | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------|---------|----------|---------|---------|-------------------------------------------------
All files      |   84.78 |       80 |     100 |   84.26 |
 ZTPDOGOS.cbl  |   84.78 |       80 |     100 |   84.26 | ...175-176,186,195,198,201,206-207,214-215
---------------|---------|----------|---------|---------|-------------------------------------------------
```

Your "Hello Test4z!" greeting from the Test4z Message call will be written to the test-out/ZLMSG.txt file:

```
≡ ZLMSG.txt  ✕

5    ZTESRN01I CATALOG SEARCH ON DK896808.TEST4Z.**.TEST.LOAD YIELDED 001 DATASET MATCHES
6    ZTESRN02I MEMBER SEARCH ON DK896808.TEST4Z.TEST.LOAD YIELDED 001 MEMBER MATCHES
7    ZTESRN03I PROCESSING TEST ON MEMBER ZTTDOGWS
8    Hello Test4z!
```

ZLMSG also includes a summary of what unit tests were run and whether they passed or failed. The output from DISPLAY messages are stored in test-out/SYSOUT.TXT.
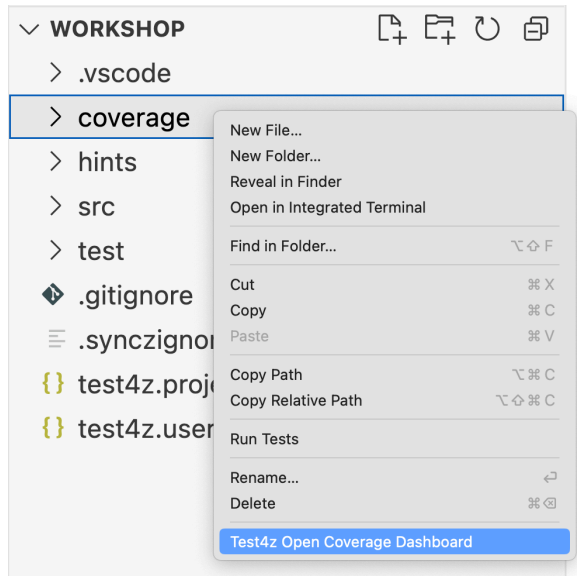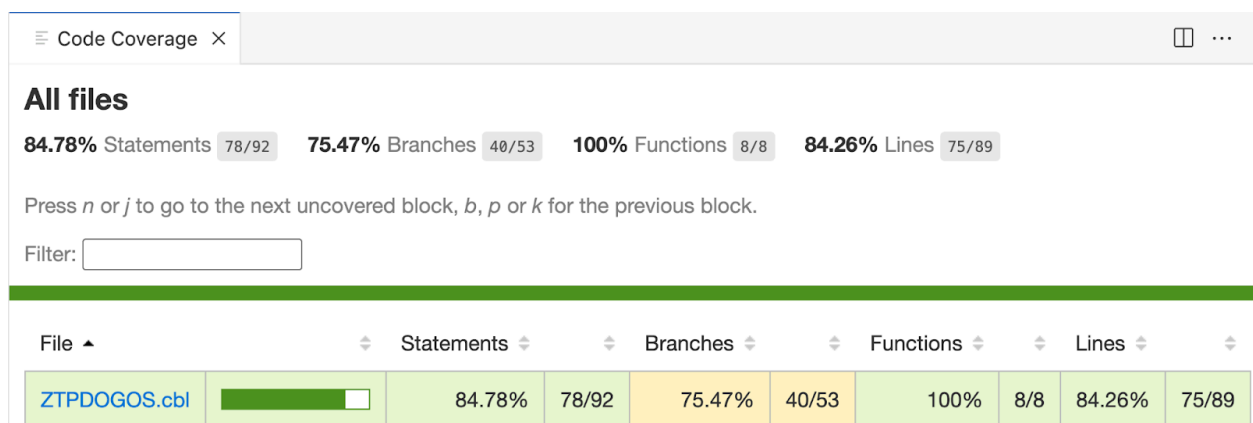
TEST4Z CODE COVERAGE

Test4z Test Drive

You generated code coverage statistics in the last section. Now let's take a quick look at them.

Try it ✎

Open the Coverage Dashboard by right clicking the coverage folder to display its pop-up menu and then select "Test4z Open Coverage Dashboard":

This view shows the percentage of statements, conditional branches like `IF` and `EVALUATE`, and functions (sections/paragraphs) that were executed:
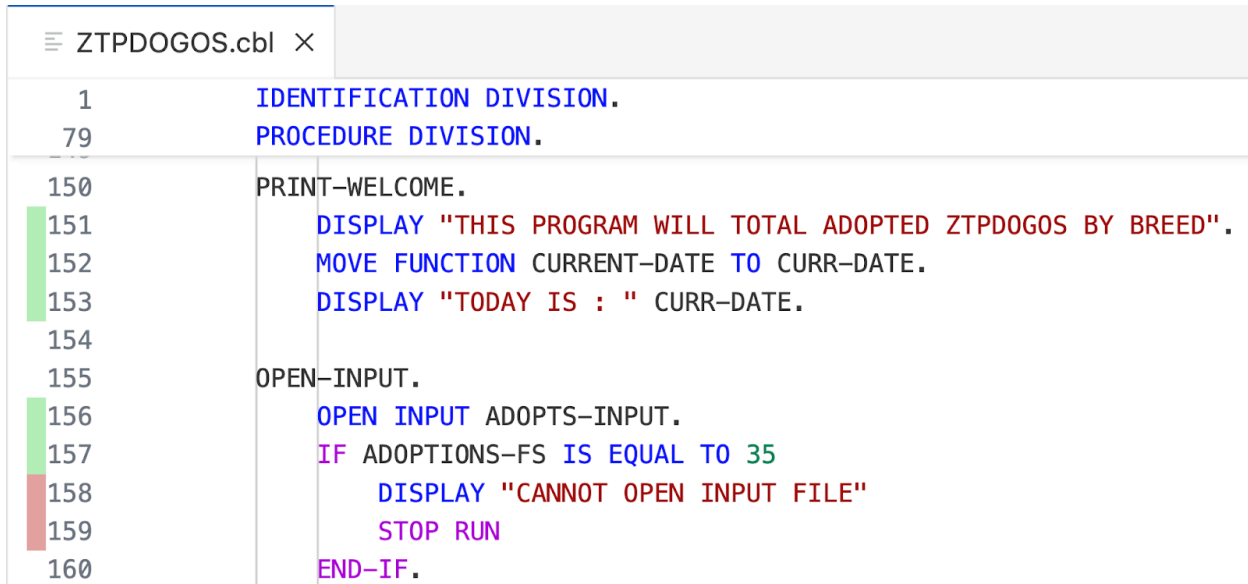

You can see more detail by clicking the source file name "ZTPDOGOS.cbl" in the dashboard.

Test4z Test Drive

The Code Coverage dashboard is a static snapshot saved as an HTML file. However, once code coverage is run, the associated source files are also annotated with green/red gutters in the VS Code editor.

Try it ✎

Open test/ZTPDOGOS.cbl as shown below:

```
≡ ZTPDOGOS.cbl ×

    1            IDENTIFICATION DIVISION.
   79            PROCEDURE DIVISION.

  150        PRINT-WELCOME.
  151            DISPLAY "THIS PROGRAM WILL TOTAL ADOPTED ZTPDOGOS BY BREED".
  152            MOVE FUNCTION CURRENT-DATE TO CURR-DATE.
  153            DISPLAY "TODAY IS : " CURR-DATE.
  154
  155        OPEN-INPUT.
  156            OPEN INPUT ADOPTS-INPUT.
  157            IF ADOPTIONS-FS IS EQUAL TO 35
  158                DISPLAY "CANNOT OPEN INPUT FILE"
  159                STOP RUN
  160            END-IF.
```

The annotations indicate which statements were executed (green) and which were not (red).

HOW UNIT TEST MOCKS SIMPLIFY ENVIRONMENT SETUP

For this exercise, the purpose of the two mocked files–ADOPTS for input and OUTREP for output–is to decouple the unit test from a live environment. When run under Test4z, the tested program will *behave* as if it's reading and writing real QSAM files, but it's actually using the Test4z data from a previously recorded live environment stored in test/ZTPDOGOS.json.

Establishing the mocked ADOPTS QSAM file starts by loading the previously recorded data:

Test4z Test Drive

```
≡ ZTTDOGWS.cbl ✕

137          * Load data from a previous "live" recording.
138
139              display 'See workshop step 1.3 (#LOADRECORDED)'
140
141              move low-values to I_LoadData
142              move 'ZTPDOGOS' to memberName in ZWS_LoadData
143              call ZTESTUT using ZWS_LoadData, loadObject in LOAD_Data
```

The `memberName` refers to the recording stored in the test/data directory; the Test4z CLI uploads the recording to the mainframe as part of running the unit test. The mock for the ADOPTS file accepts this recorded data as input in the MockQSAM API with the `loadObject` parameter:

```
≡ ZTTDOGWS.cbl ✕

145          * Initialize QSAM file access mock object for the ADOPTS DD
146          * with the load object (data) created above.
147
148              display 'See workshop step 1.4 (#MOCKADOPTS)'
149
150              move low-values to I_MockQSAM
151              move 'ADOPTS' to fileName in ZWS_MockQSAM
152              set loadObject in ZWS_MockQSAM to loadObject in LOAD_Data
153              move 80 to recordSize in ZWS_MockQSAM
154              call ZTESTUT using ZWS_MockQSAM, qsamObject in MOCK_ADOPTS.
155
156              exit.
```

Since the output file OUTREP doesn't require recorded data, creating its mock is even easier:

```
≡ ZTTDOGWS.cbl ✕

125              move low-values to I_MockQSAM
126              move 'OUTREP' to fileName in ZWS_MockQSAM
127              move 80 to recordSize in ZWS_MockQSAM
128              call ZTESTUT using ZWS_MockQSAM, qsamObject in MOCK_OUTREP.
```

Test4z Test Drive

This completes the first part of this exercise: Creating mocks for the input and output data, thereby isolating our unit test from the live environment. The next part covers the most important part of unit testing–validating the results are correct. To do that, we'll create a "spy" on the OUTREP output file.

HOW UNIT TEST SPIES SIMPLIFY VALIDATION

As the name suggests, a spy can observe the behavior of different middleware resources; in the case of this exercise, we'll create a spy on the OUTREP output file:

```
≡ ZTTDOGWS.cbl  ✕

169                move low-values to I_SpyQSAM
170                set callback in ZWS_SpyQSAM to entry 'spyCallbackOUTREP'
171                move 'OUTREP' to fileName in ZWS_SpyQSAM
172                call ZTESTUT using ZWS_SpyQSAM, qsamSpyObject in OUTREP_SPY.
```

The `callback` field specifies the unit test's entry point that will be invoked whenever an operation against the OUTREP file is requested. That gives the unit test an opportunity to validate the output records.

In the case of the OUTREP file spy, this is handled in two steps. First, during the callback to `spyCallbbackOUTREP`, it records the number of valid writes:

```
≡ ZTTDOGWS.cbl  ✕

 99                if command in ZLS_QSAM_Record = 'WRITE' and
100                      statusCode in ZLS_QSAM_Record = '00'
101
102                   display 'See workshop step 3.2 (#SPYWRITE)'
103                   add 1 to OUTREP_SPY_WRITE_COUNT
104
105                end-if
```

When the OUTREP file is closed, the spy callback's entry point is called again, giving the unit test an opportunity to validate the final count. An excerpt of this validation is shown below:

Test4z Test Drive

```
 190           validateResults.
 191
 192        * Black box test – confirm the correct number of OUTREP records.
 193
 194           display 'See workshop step 3.3 (#VALIDATEOUTREP)'
 195
 196           if OUTREP_SPY_WRITE_COUNT not = 9 then
 197             perform failOutrepWriteCount
 198           end-if
```

Try it ✎

If you have time, change the statement above to intentionally fail (e.g., delete the "`not`" in the IF statement above or change "9" to "9000"), then run the test suite again. It should fail with an assert error:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

FAIL  test/ZTTDOGWS.cbl
  × ZTTDOGWS simple totals test (103 ms)
      Assertion error: Invalid OUTREP count from ZTTDOGWS
```

Between the mocked input file and spied output file, the unit test has full visibility into the workings of the tested program. As changes are applied to the tested program, the unit test can evolve too, assuring that the tested program continues to work as expected.

WHAT'S NEXT?

This is just a glimpse of using Test4z. While the exercise might be short and simplified, the overriding Test4z principles–isolation and automation–can be applied to any code under test, reducing the need for manual testing and increasing the effectiveness of your test efforts.

If you'd like to learn more, check out the full Test4z Tutorial exercise. It includes more details about the mock, spy, and validation APIs as well as a more in-depth explanation of the unit test suite code.