Continuous Integration with the ObjectScript Package Manager, GitHub Actions, and Docker

Article 

Timothy Leavitt · Aug 27, 2020 7m read

Continuous Integration with the ObjectScript Package Manager, GitHub Actions, and Docker

Introduction

In a previous article, I discussed patterns for running unit tests via the ObjectScript package manager. This article goes a step further, using GitHub actions to drive test execution and reporting. The motivating use case is running CI for one of my Open Exchange projects, AppS.REST (see the introductory article for it here). You can see the full implementation from which the snippets in this article were taken on GitHub; it could easily serve as a template for running CI for other projects using the ObjectScript package manager.

Features demonstrated implementation include:

- Building and testing an ObjectScript package
- Reporting test coverage measurement (using the TestCoverage package) via codecov.io
- Uploading a report on test results as a build artifact

The Build Environment

There's comprehensive documentation on GitHub actions here; for purposes of this article, we'll just explore the aspects demonstrated in this example.

A workflow in GitHub actions is triggered by a configurable set of events, and consists of a number of jobs that can run sequentially or in parallel. Each job has a set of steps - we'll go into the details of the steps for our example action in a bit. These steps consist of references to actions available on GitHub, or may just be shell commands. A snippet of the initial boilerplate in our example looks like:

```
# Continuous integration workflow
name: CI

# Controls when the action will run. Triggers the workflow on push or pull request
# events in all branches
on: [push, pull_request]

# A workflow run is made up of one or more jobs that can run sequentially or in paral
lel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest
```

```
    env:
      # Environment variables usable throughout the "build" job, e.g. in OS-
level commands
      package: apps.rest
      container_image: intersystemsdc/iris-community:2019.4.0.383.0-zpm
      # More of these will be discussed later...

    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
      # These will be shown later...
```

For this example, there are a number of environment variables in use. To apply this example to other packages using the ObjectScript Package Manager, many of these wouldn't need to change at all, though some would.

```
    env:
      # ** FOR GENERAL USE, LIKELY NEED TO CHANGE: **
      package: apps.rest
      container_image: intersystemsdc/iris-community:2019.4.0.383.0-zpm

      # ** FOR GENERAL USE, MAY NEED TO CHANGE: **
      build_flags: -dev -verbose # Load in -dev mode to get unit test code preloaded
      test_package: UnitTest

      # ** FOR GENERAL USE, SHOULD NOT NEED TO CHANGE: **
      instance: iris
      # Note: test_reports value is duplicated in test_flags environment variable
      test_reports: test-reports
      test_flags: >-
       -verbose -DUnitTest.ManagerClass=TestCoverage.Manager -DUnitTest.JUnitOutput=/
test-reports/junit.xml
       -DUnitTest.FailuresAreFatal=1 -DUnitTest.Manager=TestCoverage.Manager
       -DUnitTest.UserParam.CoverageReportClass=TestCoverage.Report.Cobertura.ReportG
enerator
       -DUnitTest.UserParam.CoverageReportFile=/source/coverage.xml
```

If you want to adapt this to your own package, just drop in your own package name and preferred container image (must include zpm - see https://hub.docker.com/r/intersystemsdc/iris-community). You might also want to change the unit test package to match your own package's convention (if you need to load and compile unit tests before running them to deal with any load/compile dependencies; I had some weird issues specific to the unit tests for this package, so it might not even be relevant in other cases).

The instance name and test_reports directory shouldn't need to be modified for other use, and the test_flags provide a good set of defaults - these support having unit test failures flag the build as failing, and also handle export of jUnit-formatted test results and a code coverage report.

Build Steps

Checking out GitHub Repositories

In our motivating example, two repositories need to be checked out - the one being tested, and also my fork of Forgery (because the unit tests need it).

```
    # Checks out this repository under $GITHUB_WORKSPACE, so your job can access it
    - uses: actions/checkout@v2

    # Also need to check out timleavitt/forgery until the official version installabl
```

```
e via ZPM
    - uses: actions/checkout@v2
      with:
        repository: timleavitt/forgery
        path: forgery
```

$GITHUB_WORKSPACE is a very important environment variable, representing the root directory where all of this runs. From a permissions perspective, you can do pretty much whatever you want within that directory; elsewhere, you may run in to issues.

Running the InterSystems IRIS Container

After setting up a directory where we'll end up putting our test result reports, we'll run the InterSystems IRIS Community Edition (+ZPM) container for our build.

```
    - name: Run Container
      run: |
        # Create test_reports directory to share test results before running containe
r
        mkdir $test_reports
        chmod 777 $test_reports
        # Run InterSystems IRIS instance
        docker pull $container_image
        docker run -d -h $instance --name $instance -v $GITHUB_WORKSPACE:/source -v $
GITHUB_WORKSPACE/$test_reports:/$test_reports --init $container_image
        echo halt > wait
        # Wait for instance to be ready
        until docker exec --interactive $instance iris session $instance < wait; do s
leep 1; done
```

There are two volumes shared with the container - the GitHub workspace (so that the code can be loaded; we'll also report test coverage info back to there), and a separate directory where we'll put the jUnit test results.

After "docker run" finishes, that doesn't mean the instance is fully started and ready to command yet. To wait for the instance to be ready, we'll keep trying to run a "halt" command via iris session; this will fail and continue trying once per second until it (eventually) succeeds, indicating that the instance is ready.

Installing test-related libraries

For our motivating use case, we'll be using two other libraries for testing - TestCoverage and Forgery. TestCoverage can be installed directly via the Community Package Manager; Forgery (currently) needs to be loaded via zpm "load"; but both approaches are valid.

```
    - name: Install TestCoverage
      run: |
        echo "zpm \"install testcoverage\":1:1" > install-testcoverage
        docker exec --interactive $instance iris session $instance -B < install-
testcoverage
        # Workaround for permissions issues in TestCoverage (creating directory for s
ource export)
        chmod 777 $GITHUB_WORKSPACE

    - name: Install Forgery
      run: |
        echo "zpm \"load /source/forgery\":1:1" > load-forgery
        docker exec --interactive $instance iris session $instance -B < load-forgery
```

The general approach is to write out commands to a file, then run then in IRIS session. The extra ":1:1" in the ZPM commands indicates that the command should exit the process with an error code if an error occurs, and halt at the end if no errors occur; this means that if an error occurs, it will be reported as a failed build step, and we don't need to add a "halt" command at the end of each file.

Building and Testing the Package

Finally, we can actually build and run tests for our package. This is pretty simple - note use of the $build_flags/$test_flags environment variables we defined earlier.

```
    # Runs a set of commands using the runners shell
    - name: Build and Test
      run: |
        # Run build
        echo "zpm \"load /source $build_flags\":1:1" > build
        # Test package is compiled first as a workaround for some dependency issues.
        echo "do \$System.OBJ.CompilePackage(\"$test_package\",\"ckd\") " > test
        # Run tests
        echo "zpm \"$package test -only $test_flags\":1:1" >> test
        docker exec --interactive $instance iris session $instance -B < build && dock
er exec --interactive $instance iris session $instance -B < test && bash <(curl -s ht
tps://codecov.io/bash)
```

This follows the same pattern we've seen, writing out commands to a file then using that file as input to iris session.

The last part of the last line uploads code coverage results to codecov.io. Super easy!

Uploading Unit Test Results

Suppose a unit test fails. It'd be really annoying to have to go back through the build log to find out what went wrong, though this may still provide useful context. To make life easier, we can upload our jUnit-formatted results and even run a third-party program to turn them into a pretty HTML report.

```
    # Generate and Upload HTML xUnit report
    - name: XUnit Viewer
      id: xunit-viewer
      uses: AutoModality/action-xunit-viewer@v1
      if: always()
      with:
        # With -DUnitTest.FailuresAreFatal=1 a failed unit test will fail the build b
efore this point.
        # This action would otherwise misinterpret our xUnit style output and fail th
e build even if
        # all tests passed.
        fail: false
    - name: Attach the report
      uses: actions/upload-artifact@v1
      if: always()
      with:
        name: ${{ steps.xunit-viewer.outputs.report-name }}
        path: ${{ steps.xunit-viewer.outputs.report-dir }}
```

This is mostly taken from the readme at https://github.com/AutoModality/action-xunit-viewer.

The End Result

If you want to see the results of this workflow, check out:

Logs for the CI job on intersystems/apps-rest (including build artifacts): https://github.com/intersystems/apps-rest/actions?query=workflow%3ACI
Test coverage reports: https://codecov.io/gh/intersystems/apps-rest

Please let me know if you have any questions!

#Code Snippet #Continuous Integration #Docker #GitHub #ObjectScript Package Manager (ZPM) #Testing #InterSystems IRIS #InterSystems IRIS for Health #Open Exchange
20 1 2 0 178

Related posts

- Unit Tests and Test Coverage in the ObjectScript Package Manager
- Continuous Integration with the ObjectScript Package Manager, GitHub Actions, and Docker

Source URL: https://community.intersystems.com/post/continuous-integration-objectscript-package-manager-github-actions-and-docker