記事
Toshihiko Minamoto · 2022年5月17日 9m read

ObjectScript Package Manager、GitHub Actions、および Docker による継続的インテグレーション

はじめに

前の記事では、ObjectScript Package Manager
を使用してユニットテストを実行するためのパターンについて説明しました。
この記事では、さらに一歩踏み込み、GitHub Actions を使用してテストの実行とレポート作成を行います。 私の Open Exchange プロジェクトの 1 つである AppS.REST に CI
を実行するのが、やる気の出るユースケースでしょう（この導入編の記事は、こちらにあります）。
この記事のスニペットが使用されている完全な実装は、GitHub でご覧ください。ObjectScript Package Manager を使って他のプロジェクトで CI を実行するためのテンプレートとして簡単に利用できます。

紹介する実装の機能は以下のとおりです。

- ObjectScript パッケージの構築とテスト
- codecov.io によるテストカバレッジ測定のレポート（TestCoverage パッケージを使用）
- テスト結果に関するレポートのビルドアーティファクトとしてのアップロード

ビルド環境

GitHub Actions
に関する完全なドキュメントはこちら
にあります。この記事の目的に準じ、この例で紹介される側面だけを詳しく確認します。

GitHub Actions のワークフローは、構成可能な一連のイベントによってトリガーされ、順番に、または並行して実行できる多数のジョブで構成されています。 それぞれのジョブには一連のステップがあります。このサンプルアクションのステップは、少し後の方で詳しく説明します。 これらのステップは、GitHub
で提供されているアクションへの参照で構成されているか、単にシェルコマンドである場合があります。
この例での最初のボイラープレートのスニペットは、以下のようになります。

```
# 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...
```

この例では、多数の環境変数が使用されています。 この例を ObjectScript Package Manager を使用して他のパッケージに適用する場合、ほとんどの変数を変更する必要はありませんが、一部には変更が必要です。

```
    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
```

これを独自のパッケージに適応させるには、独自のパッケージ名と優先するコンテナイメージをドロップしてください（zpm を含める必要があります。https://hub.docker.com/r/intersystemsdc/iris-community をご覧ください）。また、ユニットテストパッケージが独自のパッケージの規則に一致するように変更することもお勧めします（ユニットテストを実行する前に、読み込みとコンパイルを行ってして依存関係の読み込み/コンパイルを処理する必要がある場合。私自身、このパッケージではユニットテストに特有の奇妙な問題に遭遇したためですが、他のケースでは関連性がないかもしれません）。

インスタンス名と testreports ディレクトリは、他の使用では変更する必要はありませんし、testflags には有効なデフォルトセットが備わっています。これらは、ユニットテストが失敗すると、ビルドを失敗としてフラグする機能をサポートしており、jUnit 形式のテスト結果とコードカバレッジレポートのエクスポートも処理できます。

ビルドのステップ

GitHub リポジトリの確認

この例では、テストされているリポジトリと、Forgery のフォーク（ユニットテストで必要であるため）の2つのリポジトリを確認する必要があります。

```
    # 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
```

$GITHUBWORKSPACE は非常に重要な環境変数で、このすべてが実行するルートディレクトリを表します。 権限の観点では、そのディレクトリ内ではほぼあらゆる操作を実行できますが、他の場所では問題に遭遇するかもしれません。

InterSystems IRIS コンテナの実行

最終的にテスト結果レポートを配置するディレクトリをセットアップしたら、ビルドの InterSystems IRIS Community Edition（+ZPM）コンテナを実行します。

```
    - 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 &lt; wait; d
o sleep 1; done
```

コンテナに共有されている GitHub
ワークスペース（コードを読み込めるようにするため。ここにはテストカバレッジ情報もレポートします）と jUnit テスト結果を配置する別のディレクトリの 2 つのボリュームがあります。

「docker run」が終了しても、インスタンスが完全に開始され、コマンドを処理する準備が整っているわけではありません。 インスタンスの準備が整うまで、iris セッションを通じて「halt」コマンドを実行し続けます。これは失敗となりますが、（最終的に）成功になるまで、1 秒に 1
回ずつ試行し続けます。成功になれば、インスタンスの準備は完了です。

テスト関連ライブラリのインストール

このユースケースでは、他に TestCoverage と Forgery という 2 つのライブラリを使用してテストします。
TestCoverage は、Community Package Manager を介して直接インストールできますが、Forgery
については、（現時点では）zpm「load」を介して読み込む必要があります。いずれのアプローチも有効です。

```
    - name: Install TestCoverage
      run: |
        echo "zpm \"install testcoverage\":1:1" > install-testcoverage
        docker exec --interactive $instance iris session $instance -B &lt; 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 &lt; load-
forgery
```

一般的には、ファイルにコマンドを記述して空、IRIS セッションで実行します。 ZPM コマンドに追加されている「:1:1」は、エラーが発生したらエラーコードとともにプロセスを終了し、エラーが発生しない場合は最後に停止することを示しています。つまり、エラーが発生した場合は、失敗したビルドステップとして報告されるため、ファイルの最後に「halt」コマンドを追加する必要はありません。

パッケージの構築とテスト

最後に、パッケージのテストを実際に構築して実行しましょう。 これはいたって単純です。始めの方で定義した $buildflags/$testflags 環境変数が使用されていることに注目してください。

```
    # 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 &lt; build && d
ocker exec --interactive $instance iris session $instance -B &lt; test && bash &lt;(c
url -s https://codecov.io/bash)
```

これは、見たことのあるパターンに則っています。ファイルにコマンドを書き出してから、そのファイルを iris セッションの入力として使用しています。

最後の行の最後の部分は、コードカバレッジの結果を codecov.io にアップロードしています。 とても簡単です！

ユニットテスト結果のアップロード

ユニットテストが失敗したとしましょう。 ビルドログに戻って、どこで間違ってしまったのかを見つけ出すのは本当に面倒な作業ですが、有用なコンテキストが得られるかもしれません。 作業を楽にするために、jUnit 形式の結果をアップロードできるだけでなく、サードパーティのプログラムを実行して、見栄えの良い HTML レポートに変換することも可能です。

```
    # 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 }}
```

このほとんどは、https://github.com/AutoModality/action-xunit-viewer の Readme から拝借したものです。

最終結果

このワークフローの結果は、以下でご覧いただけます。

intersystems/apps-rest での CI ジョブのログ（ビルドアーティファクトを含む）：
https://github.com/intersystems/apps-rest/actions?query=workflow%3ACI
テストカバレッジレポート: https://codecov.io/gh/intersystems/apps-rest

ご質問がございましたら、お知らせください！

#Code Snippet #Docker #GitHub #InterSystems Package Manager (IPM) #テスト #継続的インテグレーション #InterSystems IRIS #InterSystems IRIS for Health #Open Exchange

---

ソースURL:https://jp.community.intersystems.com/post/objectscript-package-manager%E3%80%81github-actions%E3%80%81%E3%81%8A%E3%82%88%E3%81%B3-docker-%E3%81%AB%E3%82%88%E3%82%8B%E7%B6%99%E7%B6%9A%E7%9A%84%E3%82%A4%E3%83%B3%E3%83%86%E3%82%B0%E3%83%AC%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3