記事
Mihoko Iijima · 2020年7月6日 11m read

GitLabを使用したInterSystemsソリューションの継続的デリバリー - パートVII：コンテナを使用したCD

この連載記事
では、InterSystemsの技術とGitLabを使用したソフトウェア開発に向けて実現可能な複数の手法を紹介し、議論したいと思います。 次のようなトピックについて説明します。

- Git 101
- Gitフロー（開発プロセス）
- GitLabのインストール
- GitLabワークフロー
- 継続的デリバリー
- GitLabのインストールと構成
- GitLab CI/CD
- コンテナを使用する理由
- コンテナインフラストラクチャ
- **コンテナを使用したCD**

第1回の記事
では、Gitの基本、Gitの概念を高度に理解することが現代のソフトウェア開発にとって重要である理由、Gitを使用してソフトウェアを開発する方法について説明しています。

第2回の記事
では、ソフトウェアのライフサイクルの完全なプロセスであるGitLabワークフローについて説明しています。

第3回の記事では、GitLabのインストールと構成ならびに利用環境のGitLabへの接続について説明しています。

第4回の記事では、CDの構成を説明しています。

第5回の記事では、コンテナとその使用方法（および使用する理由）について説明しています。

第6回の記事
では、コンテナを使用して継続的デリバリーのパイプラインを実行する必要がある主なコンポーネントと、それらすべての連携の仕組みについて説明しています。

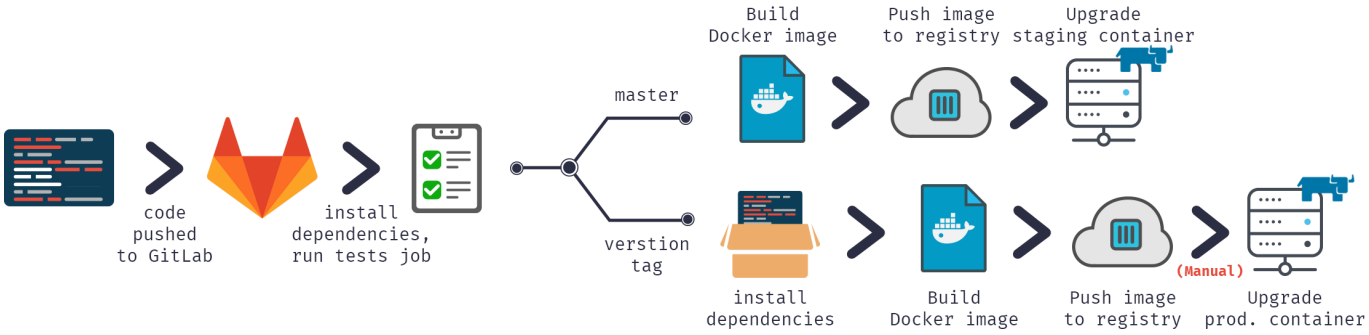
この記事では、これまでの記事で説明した継続的デリバリーの構成を構築します。

ワークフロー

継続的デリバリーの構成では、次のことを行います。

- GitLabリポジトリにコードを Push する
- Dockerイメージをビルドする
- テストする
- Dockerレジストリにイメージを公開する
- 古いコンテナをレジストリの新しいバージョンと入れ替える

図解では以下の通りです：



詳細は以下の通りです。

ビルド

まず、Dockerイメージをビルドする必要があります。

インターシステムズの製品内で使用するコードはいつものようにリポジトリに保存され、CD設定は gitlab-ci.yml に保存されますが、それに加えて（セキュリティを高めるために）ビルドサーバーにサーバー固有のファイルをいくつか保存します。

GitLab.xml

このファイルには、CDフックコードが含まれています
。 前の記事で開発したもので、 GitHubで公開されています。 これはコードをロードしたり、さまざまなフックを実行したり、コードをテストするための小さなライブラリです。
好ましい代替策として、 gitサブモジュール
を使用して、このプロジェクトまたは類似のものをリポジトリに含めることができます。
サブモジュールの方が最新の状態に保つことが簡単なので良いでしょう。
もう1つの方法としては、GitLabでリリースにタグを付け、 ADD コマンドで読み込む方法があります。

iris.key

IRISのライセンスキーです。
サーバーに保存するのではなく、コンテナのビルド中にダウンロードすることもできます。
リポジトリに保存するのはあまり安全ではありません。

pwd.txt

デフォルトのパスワードを含むファイル。 繰り返しますが、リポジトリに保存するのは安全ではありません。
また、別のサーバーでprod環境をホストしている場合は、デフォルトのパスワードが異なる場合があります。

loadci.script

初期スクリプトは：

- OS認証を有効にする
- GitLab.xmlをロードする
- GitLabユーティリティ設定を初期化する
- コードをロードする

```
set sc = ##Class(Security.System).Get("SYSTEM",.Properties)
write:('sc) $System.Status.GetErrorText(sc)
set AutheEnabled = Properties("AutheEnabled")
set AutheEnabled = $zb(+AutheEnabled,16,7)
```

```
set Properties("AutheEnabled") = AutheEnabled
set sc = ##Class(Security.System).Modify("SYSTEM",.Properties)
write:('sc) $System.Status.GetErrorText(sc)
zn "USER"
do ##class(%SYSTEM.OBJ).Load(##class(%File).ManagerDirectory() _ "GitLab.xml","cdk")
do ##class(isc.git.Settings).setSetting("hooks", "MyApp/Hooks/")
do ##class(isc.git.Settings).setSetting("tests", "MyApp/Tests/")
do ##class(isc.git.GitLab).load()
halt
```

1行目は意図的に空欄にしてあることにご注意ください。

設定によってはサーバー固有のものもあるので、リポジトリではなく個別に保存しています。
この初期フックが常に同じである場合は、リポジトリに保存すればよいだけです。

gitlab-ci.yml

次に、継続的デリバリーの構成のを参照します：

```
build image:
  stage: build
  tags:
  - test
  script:
  - cp -r /InterSystems/mount ci
  - cd ci
  - echo 'SuperUser' ¦ cat - pwd.txt loadci.script > temp.txt
  - mv temp.txt loadci.script
  - cd ..
  - docker build --build-arg CIPROJECTDIR=$CIPROJECTDIR -t
docker.domain.com/test/docker:$CICOMMITREFNAME .
```

ここでは何が起きているのでしょうか？

まず、最初にdocker build
はベースのビルドディレクトリのサブディレクトリにしかアクセスできないので（この場合はリポジトリルートにのみ）、「秘密の」ディレクトリ (GitLab.xml、iris.key、pwd.txt および loadci.scriptがあるもの）をクローンしたリポジトリにコピーする必要があります。

次に、最初のターミナルアクセスにはユーザーとパスが必要なので、loadci.script に追加します（ loadci.script の先頭の空行はそのためのものです）。

最後に、dockerイメージをビルドして次のように適切なタグを付けます。

```
  docker.domain.com/test/docker：$CICOMMITREFNAME
```

ここで $CICOMMITREFNAME は、現在のブランチの名前です。 イメージタグの最初の部分は、GitLabのプロジェクト名と同じ名前でなければならないので、GitLabのレジストリタブで確認することができます（タグ付けの方法はレジストリタブで確認することができます）。

Dockerfile

Dockerイメージのビルドは、次のようにDockerfile を使用します。
FROM docker.intersystems.com/intersystems/iris:2018.1.1.611.0

```
ENV SRCDIR=/tmp/src
ENV CIDIR=$SRCDIR/ci
ENV CIPROJECTDIR=$SRCDIR

COPY ./ $SRCDIR

RUN cp $CIDIR/iris.key $ISCPACKAGEINSTALLDIR/mgr/ \
 && cp $CIDIR/GitLab.xml $ISCPACKAGEINSTALLDIR/mgr/ \
 && $ISCPACKAGEINSTALLDIR/dev/Cloud/ICM/changePassword.sh $CIDIR/pwd.txt \
 && iris start $ISCPACKAGEINSTANCENAME \
 && irissession $ISCPACKAGEINSTANCENAME -U%SYS < $CIDIR/loadci.script \
 && iris stop $ISCPACKAGEINSTANCENAME quietly
```

基本的なIRISコンテナから開始します。

まず、コンテナ内にリポジトリ（および「秘密の」ディレクトリ）をコピーします。

次に、ライセンスキーと　GitLab.xml を mgr ディレクトリにコピーします。

次に、パスワードを pwd.txt の値に変更します。この操作では、pwd.txt が削除されることに注意してください。

その後、インスタンスが起動され、loadci.script を実行します。

最後に、IRISインスタンスを停止します。

これがジョブログです（部分的に、ロード/コンパイルをスキップしたログ）：

```
Running with gitlab-runner 10.6.0 (a3543a27)
  on docker 7b21e0c4
Using Shell executor...
Running on docker...
Fetching changes...
Removing ci/
Removing temp.txt
HEAD is now at 5ef9904 Build loadci.script
From http://gitlab.eduard.win/test/docker
  5ef9904..9753a8d master -> origin/master
Checking out 9753a8db as master...
Skipping Git submodules setup
$ cp -r /InterSystems/mount ci
$ cd ci
$ echo 'SuperUser' | cat - pwd.txt loadci.script > temp.txt
$ mv temp.txt loadci.script
$ cd ..
$ docker build --build-arg CIPROJECTDIR=$CIPROJECTDIR -t
docker.eduard.win/test/docker:$CICOMMITREFNAME .
Sending build context to Docker daemon 401.4kB
Step 1/6 : FROM docker.intersystems.com/intersystems/iris:2018.1.1.611.0
 ---> cd2e53e7f850
Step 2/6 : ENV SRCDIR=/tmp/src
 ---> Using cache
 ---> 68ba1cb00aff
Step 3/6 : ENV CIDIR=$SRCDIR/ci
 ---> Using cache
 ---> 6784c34a9ee6
Step 4/6 : ENV CIPROJECTDIR=$SRCDIR
 ---> Using cache
```

```
 ---> 3757fa88a28a
Step 5/6 : COPY ./ $SRCDIR
 ---> 5515e13741b0
Step 6/6 : RUN cp $CIDIR/iris.key $ISCPACKAGEINSTALLDIR/mgr/ && cp $CIDIR/GitLab.xml
$ISCPACKAGEINSTALLDIR/mgr/ && $ISCPACKAGEINSTALLDIR/dev/Cloud/ICM/changePassword.sh
$CIDIR/pwd.txt && iris start $ISCPACKAGEINSTANCENAME && irissession
$ISCPACKAGEINSTANCENAME -U%SYS < $CIDIR/loadci.script && iris stop
$ISCPACKAGEINSTANCENAME quietly
 ---> Running in 86526183cf7c
.
Waited 1 seconds for InterSystems IRIS to start
This copy of InterSystems IRIS has been licensed for use exclusively by:
ISC Internal Container Sharding
Copyright (c) 1986-2018 by InterSystems Corporation
Any other use is a violation of your license agreement

%SYS>
1

%SYS>
Using 'iris.cpf' configuration file

This copy of InterSystems IRIS has been licensed for use exclusively by:
ISC Internal Container Sharding
Copyright (c) 1986-2018 by InterSystems Corporation
Any other use is a violation of your license agreement

1 alert(s) during startup. See messages.log for details.
Starting IRIS

Node: 39702b122ab6, Instance: IRIS

Username:
Password:

Load started on 04/06/2018 17:38:21
Loading file /usr/irissys/mgr/GitLab.xml as xml
Load finished successfully.

USER>

USER>

[2018-04-06 17:38:22.017] Running init hooks: before

[2018-04-06 17:38:22.017] Importing hooks dir /tmp/src/MyApp/Hooks/

[2018-04-06 17:38:22.374] Executing hook class: MyApp.Hooks.Global

[2018-04-06 17:38:22.375] Executing hook class: MyApp.Hooks.Local

[2018-04-06 17:38:22.375] Importing dir /tmp/src/

Loading file /tmp/src/MyApp/Tests/TestSuite.cls as udl

Compilation started on 04/06/2018 17:38:22 with qualifiers 'c'
Compilation finished successfully in 0.194s.

Load finished successfully.
```

[2018-04-06 17:38:22.876] Running init hooks: after

[2018-04-06 17:38:22.878] Executing hook class: MyApp.Hooks.Local

[2018-04-06 17:38:22.921] Executing hook class: MyApp.Hooks.Global
Removing intermediate container 39702b122ab6
 ---> dea6b2123165
[Warning] One or more build-args [CIPROJECTDIR] were not consumed
Successfully built dea6b2123165
Successfully tagged docker.domain.com/test/docker:master
Job succeeded

Docker executorではなく GitLabShell executor を使用していることに注意してください。 Docker executorはイメージ内の何かが必要な場合に使用します。

例えば、Androidアプリケーションをjavaコンテナでビルドしていて、apkだけが必要な場合などです。 この場合はコンテナ全体が必要になり、そのためにShell executorが必要になります。 そのため、GitLab Shell executorを使用してDockerコマンドを実行しています。

Run

イメージができたので、次は実行してみましょう。フィーチャーブランチの場合、古いコンテナを破棄して新しいコンテナを開始すればよいだけです。 この環境の場合、一時的なコンテナを実行し、テストが成功した場合に環境コンテナを置き換えることができます（これは読者の方にお任せします)。

スクリプトは以下の通りです。
```
destroy old:
  stage: destroy
  tags:
  - test
  script:
  - docker stop iris-$CICOMMITREFNAME ¦¦ true
  - docker rm -f iris-$CICOMMITREFNAME ¦¦ true
```

このスクリプトは現在実行中のコンテナーを破棄し、常に成功します（デフォルトでは、Dockerが存在しないコンテナーを停止または削除しようとすると、失敗します）。

次に、新しいイメージを開始して環境として登録します。 Nginxコンテナは、
環境変数 VIRTUALHOST と（プロキシするポートを知るため）expose命令使用してリクエストを自動的にプロキシします。
```
run image:
  stage: run
  environment:
  name: $CICOMMITREFNAME
  url: http://$CICOMMITREFSLUG. docker.domain.com/index.html
  tags:
  - test
  script:
  - docker run -d
  --expose 52773
```

```
  --env VIRTUALHOST=$CICOMMITREFSLUG.docker.eduard.win
  --name iris-$CICOMMITREFNAME
  docker.domain.com/test/docker:$CICOMMITREFNAME
  --log $ISCPACKAGEINSTALLDIR/mgr/messages.log
```

テスト

テストをいくつか実行してみましょう。

```
test image:
  stage: test
  tags:
  - test
  script:
  - docker exec iris-$CICOMMITREFNAME irissession iris -U USER "##class(isc.git.GitLab).test()"
```

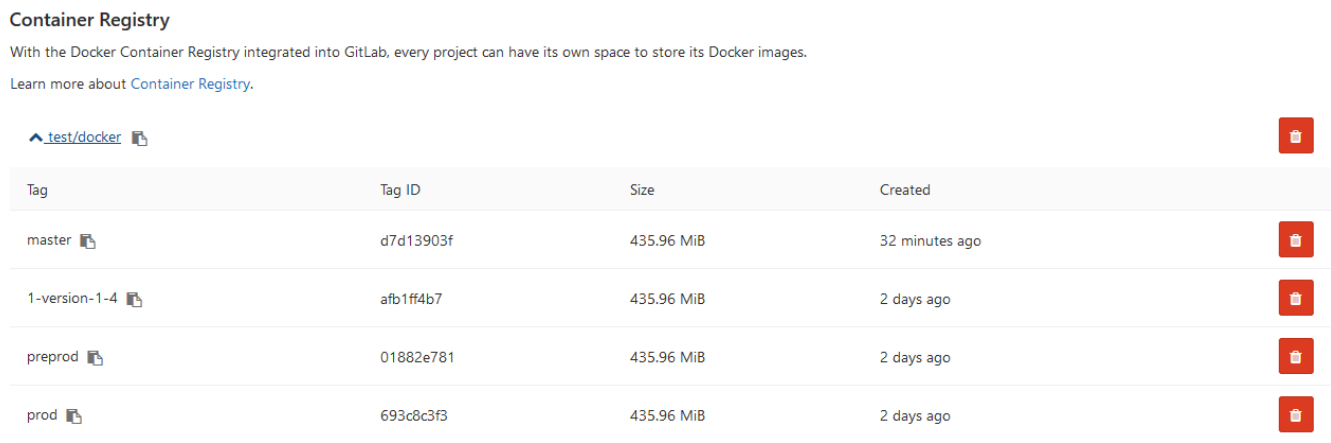
公開

最後に、レジストリにイメージを公開しましょう。

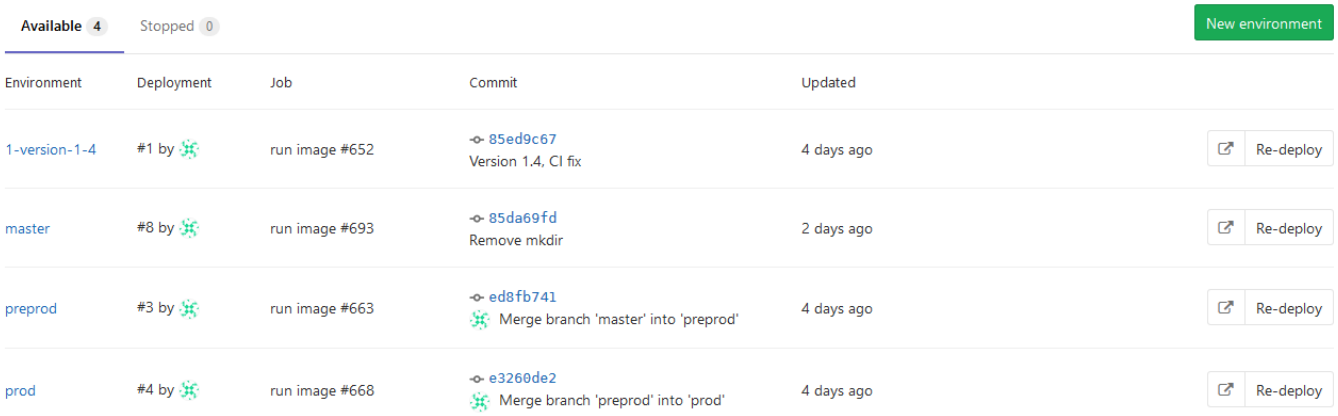
```
publish image:
  stage: publish
  tags:
    - test
  script:
    - docker login docker.domain.com -u dev -p 123
    - docker push docker.domain.com/test/docker:$CI_COMMIT_REF_NAME
```

ユーザとパスワードはGitLabの秘密の変数を使って渡すことができます。

これで、GitLabでイメージを確認できます。



そして、他の開発者がレジストリからそれをpullすることができます。 Environmentタブでは、すべての環境を簡単に参照できます。



まとめ

この連載記事では、継続的デリバリーの一般的な手法について説明しました。 これは非常に広範なトピックであり、この連載記事の内容は完成されたものではなく、レシピを集めたものとして考えてください。 アプリケーションのビルド、テスト、デリバリーを自動化したいのであれば、継続的デリバリー全般、特にGitLabが最適です。継続的デリバリーとコンテナを使うことで、必要に応じてワークフローをカスタマイズできます。

リンク

- 記事のコード
- テストプロジェクト
- CDの構成を完成する

次の内容

以上です。 継続的デリバリーとコンテナの基本をすべて網羅できたと思います。

特にコンテナーに関しては、話さなかったトピックがたくさんあります（多分、また後日お話します）。

- コンテナの外にデータを保存することができます。これにに関するドキュメントがあります
- kubernetesなどのオーケストレーション・プラットフォーム
- InterSystems Cloud Manager
- 環境管理－テスト用の一時的な環境の作成、フィーチャーブランチ合併後の旧環境の削除
- マルチコンテナ展開用の Docker compose
- Dockerイメージのサイズとビルド時間の削減

#DevOps #Docker #Git #GitHub #継続的デリバリー #InterSystems IRIS #InterSystems IRIS for Health

ソースURL:
https://jp.community.intersystems.com/post/gitlab%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9Fintersystems%E3%82%BD%E3%83%AA%E3%83%A5%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3%E3%81%AE%E7%B6%99%E7%B6%9A%E7%9A%84%E3%83%87%E3%83%AA%E3%83%90%E3%83%AA%E3%83%BC-%E3%83%91%E3%83%BC%E3%83%88vii%EF%BC%9A%E3%82%B3%E3%83%B3%E3%83%86%E3%83%8A%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9Fcd