記事
Mihoko Iijima · 2020年7月6日 12m 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VIII：ICM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いくつかの可能性のあるアプローチをを紹介し、議論したいと思います。 今回は以下のようなトピックを取り上げます。

- 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
- **ICMを使用したCD**

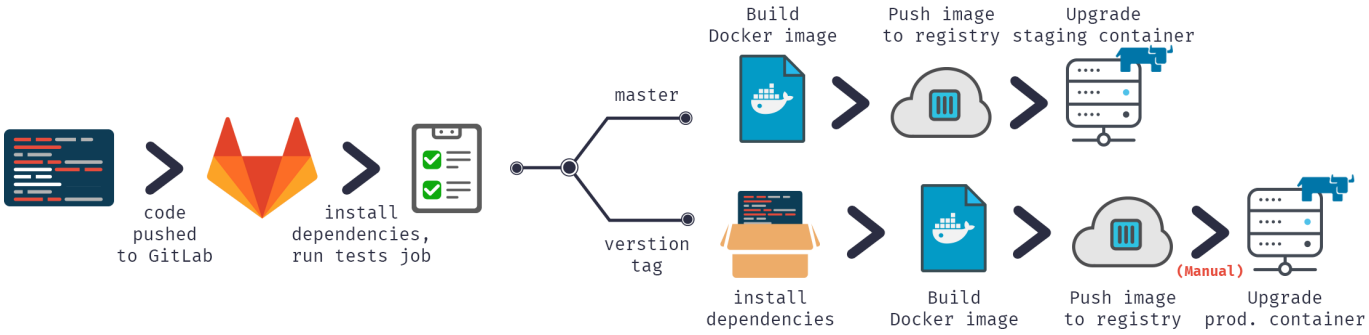
この記事では、InterSystems Cloud Managerを使用して継続的デリバリーを構築ドします。 ICMは、InterSystems IRISをベースとしたアプリケーション用のクラウドプロビジョニングおよびデプロイメントソリューションです。これにより、 必要なデプロイ構成を定義することができ、ICMが自動的にプロビジョニングします。
詳細については、First Look: ICM をご覧ください。

ワークフロー

継続的デリバリーの構成では、次のようになります。

- 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
- Dockerレジストリにイメージを公開する
- テストサーバーでテストする
- テストに合格した場合は、本番サーバーにデプロイする

または図解では以下の通りです：



ご覧のとおり、Dockerコンテナを手動で管理する代わりにICMを使用すること以外は、ほとんど同じです。

ICMの構成

コンテナのアップグレードを開始する前に、コンテナをプロビジョニングする必要があります。 そのためには、私たちのアーキテクチャを説明する defaults.json と definitions.json を定義する必要があります。

ここでは、LIVEサーバ用に2つのファイルを提供します。TESTサーバーの 定義 は同じで、 デフォルト も Tag と SystemMode の値以外は同じです。

defaults.json :

```
{
  "Provider": "GCP",
  "Label": "gsdemo2",
  "Tag": "LIVE",
  "SystemMode": "LIVE",
  "DataVolumeSize": "10",
  "SSHUser": "sample",
  "SSHPublicKey": "/icmdata/ssh/insecure.pub",
  "SSHPrivateKey": "/icmdata/ssh/insecure",
  "DockerImage": "eduard93/icmdemo:master",
  "DockerUsername": "eduard93",
  "DockerPassword": "...",
  "TLSKeyDir": "/icmdata/tls",
  "Credentials": "/icmdata/gcp.json",
  "Project": "elebedyu-test",
  "MachineType": "n1-standard-1",
  "Region": "us-east1",
  "Zone": "us-east1-b",
  "Image": "rhel-cloud/rhel-7-v20170719",
  "ISCPassword": "SYS",
  "Mirror": "false"
}
```

definitions.json

```
[
  {
  "Role": "DM",
  "Count": "1",
  "ISCLicense": "/icmdata/iris.key"
  }
]
```

ICMコンテナ内で/icmdata フォルダーがホストからマウントされ、

- TESTサーバー定義が /icmdata/test フォルダーに配置されます
- LIVEサーバー定義は /icmdata/live フォルダーに配置されます

必要なすべてのキーを取得した後で：
keygenSSH.sh /icmdata/ssh
keygenTLS.sh /icmdata/tls

そして、必要なファイルを/icmdata に配置します。

- iris.key
- gcp.json（Google Cloud Platformへのデプロイ用）

ICMを呼び出してインスタンスをプロビジョニングします。
cd /icmdata/test
icm provision
icm run
cd /icmdata/live
icm provision
icm run

これにより、1台のTESTサーバーと1台のLIVEサーバーをプロビジョニングし、

それぞれにスタンドアロンのInterSystems IRISインスタンスがプロビジョニングされます。

より詳細な説明については、First Look: ICM を参照してください。

ビルド

まず、イメージをビルドする必要があります。

コードはいつものようにリポジトリに保存され、CDの設定は 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）いくつかのサーバー固有のファイルをビルドサーバーに保存します。

iris.key

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可能性があります。

loadciicm.script

初期スクリプトは：

- インストーラをロード
- インストーラがアプリケーションの初期化を行う
- コードをロードする

```
set dir = ##class(%File).NormalizeDirectory($system.Util.GetEnviron("CIPROJECTDIR"))
do ##class(%SYSTEM.OBJ).Load(dir _"Installer/Global.cls","cdk")
do ##class(Installer.Global).init()
halt
```

最初の行は意図的に空欄にしてあります。

これまでの例とは異なる点がいくつかあります。 1つ目は、ICMはGitLabの代わりにコンテナと直接やり取りするため、OS認証を有効にしていない点です。 2つ目は、インストーラマニフェストを使ってアプリケーションを初期化し、初期化のさまざま
な方法を示している点です。 インストーラについて詳細はこの記事のインストーラの詳細をご参照ください。
最後に、イメージをDocher Hubでプライベートリポジトとして公開します。

Installer/Global.cls

インストーラマニフェストは以下の通りです。

```
<Manifest>
  <Log Text="Creating namespace ${Namespace}" Level="0"/>
  <Namespace Name="${Namespace}" Create="yes" Code="${Namespace}" Ensemble="" Data="IRISTEMP">
  <Configuration>
  <Database Name="${Namespace}" Dir="${MGRDIR}/${Namespace}" Create="yes" MountRequired="true"
Resource="%DB${Namespace}" PublicPermissions="RW" MountAtStartup="true"/>
  </Configuration>
  <Import File="${Dir}MyApp" Recurse="1" Flags="cdk" IgnoreErrors="1" />
  </Namespace>

  <Log Text="Mapping to USER" Level="0"/>
  <Namespace Name="USER" Create="no" Code="USER" Data="USER" Ensemble="0">
  <Configuration>
  <Log Text="Mapping MyApp package to USER namespace" Level="0"/>
  <ClassMapping From="${Namespace}" Package="MyApp"/>
  </Configuration>

  <CSPApplication Url="/" Directory="${Dir}client" AuthenticationMethods="64" IsNamespaceDefault="false"
Grant="%ALL" />
  <CSPApplication Url="/myApp" Directory="${Dir}" AuthenticationMethods="64" IsNamespaceDefault="false"
Grant="%ALL" />
  </Namespace>
</Manifest>
```

そして、以下の変更が実装されています。

1. アプリケーションのネームスペースを作成します。
2. アプリケーションコードデータベースを作成します（データはUSERデータベースに格納されます）。
3. コードをアプリケーションコードデータベースにロードします。
4. MyAppパッケージをUSERネームスペースにマップします。
5. HTML用とREST用の2つのWebアプリケーションを作成します。

gitlab-ci.yml

次は、継続的デリバリーの構成についてです：

```
build image:
  stage: build
  tags:
  - master
  script:
  - cp -r /InterSystems/mount ci
```

```
 - cd ci
 - echo 'SuperUser' ¦ cat - pwd.txt loadciicm.script > temp.txt
 - mv temp.txt loadci.script
 - cd ..
 - docker build --build-arg CIPROJECTDIR=$CIPROJECTDIR -t
eduard93/icmdemo:$CICOMMITREFNAME .
```

ここでは何が起きているのでしょうか？

まず、docker build
はベースとなるビルドディレクトリのサブディレクトリににしかアクセスできないので秘密」ディレクトリ (iris.key、pwd.txt、およびloadciicm.script があるディレクトリをクローンしたリポジトリをコピーする必要があります。

次の最初のターミナルアクセスにはユーザとパスの入力が必要なので、loadci.script に追加します（loadci.script の先頭行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正しくタグを付けします。

```
$ eduard93/icmdemo:$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にする必要があることに注意してください。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 Dockerfile を使って行います。

```
FROM intersystems/iris:2018.1.1-released
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（と「秘密」ディレクトリ）をコピーします。

次に、ライセンスキーを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が実行されます。

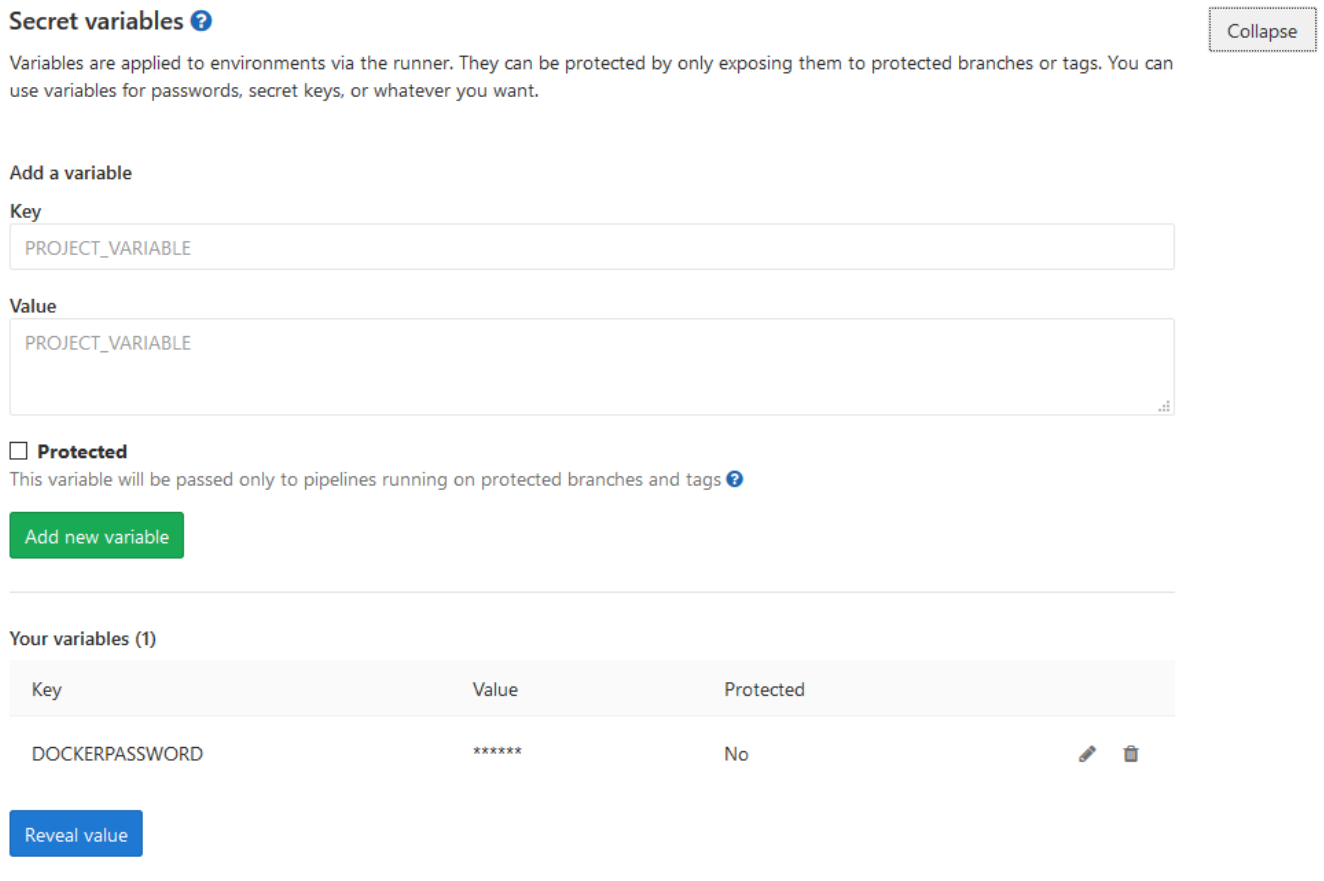
最後に、IRISインスタンスを停止します。

ここで注意したいのは、Docker executorではなく GitLabShell executor を使用していることです。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コンテナで構築していて、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を使ってDockerコマンドを実行しています。

公開

Docker Hubでイメージを公開しましょう
```
publish image:
  stage: publish
  tags:
  - master
  script:
  - docker login -u eduard93 -p ${DOCKERPASSWORD}
  - docker push eduard93/icmdemo:$CICOMMITREFNAME
```

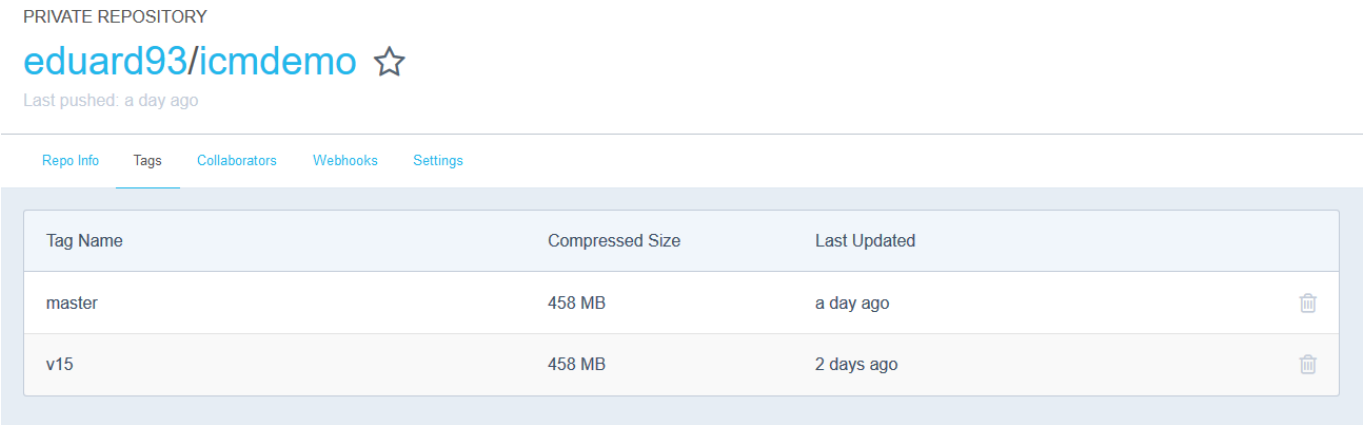
変数${DOCKERPASSWORD} に注意してください。これはGitLab 秘密変数です。 GitLab>プロジェクト>設定> CI / CD>変数でそれらを追加できます。



ジョブログには、パスワードの値は含まれません。
```
Running with gitlab-runner 10.6.0 (a3543a27)
  on icm 82634fd1
Using Shell executor...
Running on docker...
Fetching changes...
Removing ci/
```

```
HEAD is now at 8e24591 Add deploy to LIVE
Checking out 8e245910 as master...
Skipping Git submodules setup
$ docker login -u eduard93 -p ${DOCKERPASSWORD}
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Login Succeeded
$ docker push eduard93/icmdemo:$CICOMMITREFNAME
The push refers to repository [docker.io/eduard93/icmdemo]
master: digest: sha256:d1612811c11154e77c84f0c08a564a3edeb7ddbbd9b7acb80754fda97f95d101 size: 2620
Job succeeded
```

そして、Docker Hubで新しいイメージを確認できます。



実行

イメージが作成されました。次に、テストサーバーで実行してみましょう。 スクリプトは以下とおりです。

```
run image:
  stage: run
  environment:
  name: $CICOMMITREFNAME
  tags:
  - master
  script:
  - docker exec icm sh -c "cd /icmdata/test && icm upgrade -image eduard93/icmdemo:$CICOMMITREFNAME"
```

ICMでは、1つのコマンド（ icm upgrade ）を実行するだけで、既存のデプロイメントをアップグレードできます。このコマンドは icm コンテナ内で指定されたコマンドを実行する "docker exec icm sh -c "を実行して呼び出します。 最初に、ICMデプロイメント定義がTESTサーバー用に定義されている /icmdata/test にモードを切り替えます。
その後、 icmアップグレード を呼び出して、既存のコンテナを新しいコンテナに置き換えます。

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:
  - master
  script:
```

```
  - docker exec icm sh -c "cd /icmdata/test && icm session -namespace USER -command 'do
\$classmethod(\"%UnitTest.Manager\",\"RunTest\",\"MyApp/Tests\",\"/nodelete\")' | tee /dev/stderr | grep 'All
PASSED' && exit 0 || exit 1"
```

繰り返しになりますが、ここでも、icmコンテナ内で1つのコマンドを実行しています。
icmセッションは、デプロイされたノードでコマンドを実行します。 コマンドはユニットテストを実行します。
その後、すべての出力を画面にパイプし、ユニットテストの結果を見つけるために grep
にもパイプして、正常かエラーでプロセスを終了します。

デプロイ

本番サーバーへのデプロイは、LIVEデプロイメント定義用の別のディレクトリを除いては、テストのデプロイとまったく同じです。 テストが失敗した場合、このステージは実行されません。

```
deploy image:
deploy image:
  stage: deploy
  environment:
  name: $CICOMMITREFNAME
  tags:
  - master
  script:
  - docker exec icm sh -c "cd /icmdata/live && icm upgrade -image eduard93/icmdemo:$CICOMMITREFNAME"
```

まとめ

ICMは、クラウドインフラストラクチャをプロビジョニングとサービスをデプロイするための

シンプルで直感的な方法を提供します。これにより、大規模な開発や再構築を行わなくても、クラウドにアクセスできるようになります。 コードとしてのインフラストラクチャ（IaC）およびコンテナ化されたデプロイメントの利点により、InterSystems IRISベースのアプリケーションをGoogle、Amazon、Azureなどのパブリッククラウドプラットフォーム、またはプライベートのVMware vSphereクラウドに簡単にデプロイできます。 必要なものを定義し、いくつかのコマンドを発行すれば、後はICMが行います。

すでにクラウドインフラストラクチャ、コンテナ、またはその両方を使用している場合でも、ICMを使用することでアプリケーションのプロビジョニングとデプロイに必要な時間と労力を大幅に削減します。

リンク

- 記事のコード
- テストプロジェクト
- ICMドキュメント
- First Look: ICM

#AWS #Azure #DevOps #Docker #GCP #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%88viii%EF%BC%9Aicm%E3%82%92%E4%BD%BF%E7%

94%A8%E3%81%97%E3%81%9Fcd