Article
Mihoko Iijima · Jul 6, 2020 8m 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IX： コンテナアーキテクチャ

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紹介し、説明したいと思います。以下のようなトピックについて取り上げます。

- 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
- **コンテナアーキテクチャ**

この記事では、独自のコンテナを構築してデプロイ展開する方法について説明します。

Durable %SYS

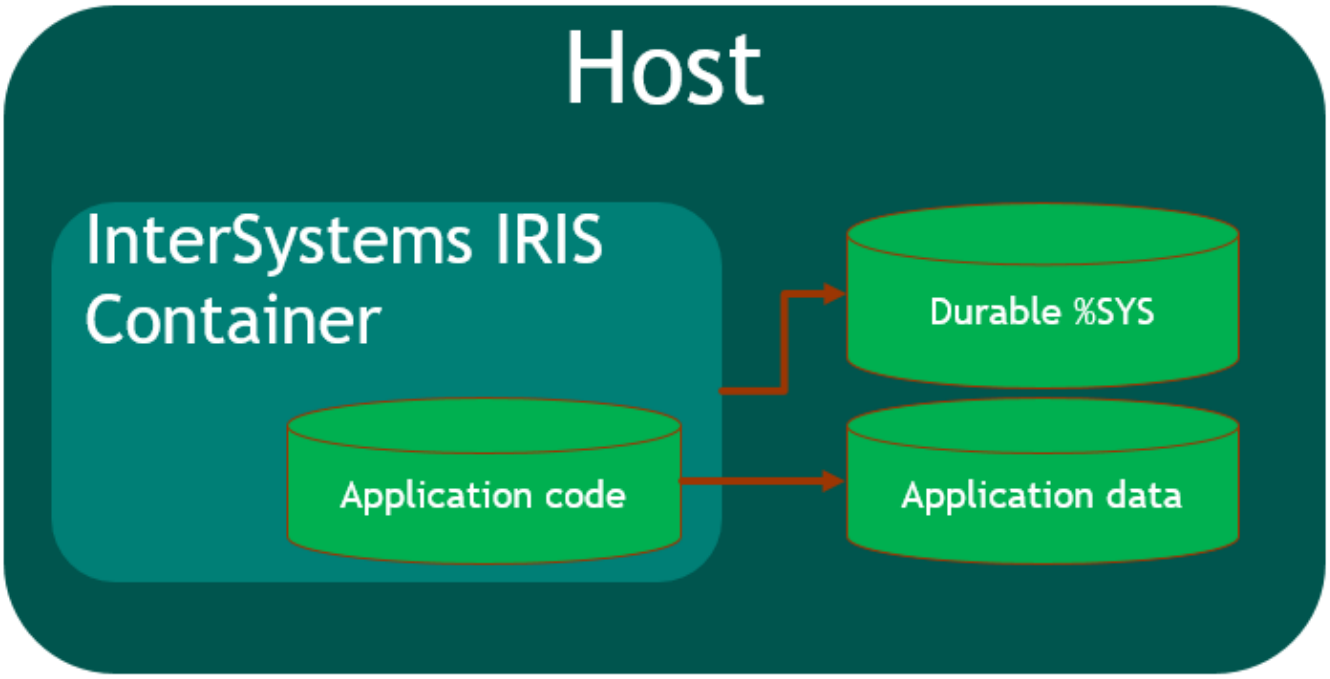
コンテナは一時的なものなので、アプリケーションデータを保存するべきではありません。Durable %SYS の機能により設定、構成、 %SYSデータなどをホストボリュームに保存することができます。

- iris.cpfファイル
- /cspディレクトリ：Webゲートウェイの構成とログファイルが含まれています。
- /httpd/httpd.confファイル：インスタンスのプライベートWebサーバーの構成ファイル。
- /mgrディレクトリ：
    - IRIS.DATおよびiris.lckファイルとストリームディレクトリで構成されるRISSYS システム・データベース、および IRISTEMP、IRISAUDIT、IRIS、USER システム・データベースを含む IRISTEMP、IRISAUDIT、IRIS、USER ディレクトリ。
    - ライトイメージジャーナルファイル：IRIS.WIJ
    - ジャーナルファイルが格納される /jounal directory
    - 一時ファイル用の /temp ディレクトリ
    - messages.log、journal.log、およびSystemMonitor.logを含むログファイル。

コンテナアーキテクチャ

一方、必要に応じてアップグレードできるように、コンテナ内にアプリケーションコードを保存しておく必要があります。

これにより、次のアーキテクチャが実現します。



ビルド時にこれを実現するには、最低限、追加のデータベース（アプリケーション・コードを格納するための）を作成し、それをアプリケーションのネームスペースにマッピングする必要があります。 この例では、アプリケーションデータがすでに存在して耐久性もあるので、USERネームスペースを利用して保存します。

インストーラ

上記に基づいて、インストーラは次のことをする必要があります：

- APPネームスペース/データベースを作成する
- APPネームスペースにコードをロードする
- アプリケーションクラスをUSER ネームスペースにマッピングする
- 他のすべてのインストールを実行する（この場合、CSP WebアプリとRESTアプリを作成しました）

```
Class MyApp.Hooks.Local
{
Parameter Namespace = "APP";

/// See generated code in zsetup+1^MyApp.Hooks.Local.1
XData Install [ XMLNamespace = INSTALLER ]
{
<Manifest>

<Log Text="Creating namespace ${Namespace}" Level="0"/>
<Namespace Name="${Namespace}" Create="yes" Code="${Namespace}" Ensemble="" Data="IRISTEMP">
<Configuration>
<Database Name="${Namespace}" Dir="/usr/irissys/mgr/${Namespace}" Create="yes" MountRequired="true"
Resource="%DB${Namespace}" PublicPermissions="RW" MountAtStartup="true"/>
</Configuration>

<Import File="${Dir}Form" Recurse="1" Flags="cdk" IgnoreErrors="1" />
</Namespace>
<Log Text="End Creating namespace ${Namespace}" Level="0"/>


<Log Text="Mapping to USER" Level="0"/>
```

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

<CSPApplication Url="/" Directory="${Dir}client" AuthenticationMethods="64" IsNamespaceDefault="false"
Grant="%ALL" Recurse="1" />
</Namespace>

</Manifest>
}

/// This is a method generator whose code is generated by XGL.
/// Main setup method
/// set vars("Namespace")="TEMP3"
/// do ##class(MyApp.Hooks.Global).setup(.vars)
ClassMethod setup(ByRef pVars, pLogLevel As %Integer = 0, pInstaller As %Installer.Installer) As %Status [
CodeMode = objectgenerator, Internal ]
{
  Quit ##class(%Installer.Manifest).%Generate(%compiledclass, %code, "Install")
}

/// Entry point
ClassMethod onAfter() As %Status
{
  try {
  write "START INSTALLER",!
  set vars("Namespace") = ..#Namespace
  set vars("Dir") = ..getDir()
  set sc = ..setup(.vars)
  write !,$System.Status.GetErrorText(sc),!

  set sc = ..createWebApp()
  } catch ex {
  set sc = ex.AsStatus()
  write !,$System.Status.GetErrorText(sc),!
  }
  quit sc
}

/// Modify web app REST
ClassMethod createWebApp(appName As %String = "/forms") As %Status
{
  set:$e(appName)'="/" appName = "/" _appName
  #dim sc As %Status = $$$OK
  new $namespace
  set $namespace = "%SYS"
  if '##class(Security.Applications).Exists(appName) {
  set props("AutheEnabled") = $$$AutheUnauthenticated
  set props("NameSpace") = "USER"
  set props("IsNameSpaceDefault") = $$$NO
  set props("DispatchClass") = "Form.REST.Main"
  set props("MatchRoles")=":" _$$$AllRoleName
  set sc = ##class(Security.Applications).Create(appName, .props)
  }
  quit sc
}
```

```
ClassMethod getDir() [ CodeMode = expression ]
{
##class(%File).NormalizeDirectory($system.Util.GetEnviron("CIPROJECTDIR"))
}

}
```

耐久性のないデータベースを作成するには、/usr/irissys/mgrのサブディレクトリを使用しますがこれは永続的ではありません。

##class(%File).ManagerDirectory()への呼び出しは、内部コンテナディレクトリへのパスではなく、耐久性のあるディレクトリのパスを返すことに注意してください。

継続的デリバリーの構成

パートVIIをご参照ください。これらの2行（太字）を既存の構成に追加すればよいだけです。

```
run image:
  stage: run
  environment:
  name: $CICOMMITREFNAME
  url: http://$CICOMMITREFSLUG.docker.eduard.win/index.html
  tags:
  - test
  script:
  - docker run -d
  --expose 52773
  --volume /InterSystems/durable/$CICOMMITREFSLUG:/data
  --env ISCDATADIRECTORY=/data/sys
  --env VIRTUALHOST=$CICOMMITREFSLUG.docker.eduard.win
  --name iris-$CICOMMITREFNAME
  docker.eduard.win/test/docker:$CICOMMITREFNAME
  --log $ISCPACKAGEINSTALLDIR/mgr/messages.log
```

volume引数は、ホストディレクトリをコンテナにマウントし、 ISCDATADIRECTORY 変数は、どのディレクトリを使用するかをInterSystems IRISに示します。ドキュメントを引用します。

これらのオプションを使用して InterSystems IRISコンテナを実行すると、次のことが起こります。

- 指定された外部ボリュームがマウントされています。
- 環境変数 ISCDATADIRECTORYで指定された耐久性のある%SYSディレクトリ（前に挙げた例のiconfig /）がすでに存在し、その中にdurable %SYSデータが含まれている場合、インスタンスのすべての内部ポインターがそのディレクトリにリセットされ、インスタンスはディレクトリに含まれるデータを使用します。
- 環境変数 ISCDATADIRECTORYで指定された耐久性のある%SYSディレクトリはすでに存在しているが、耐久性のある%SYSデータが含まれていない場合、データはコピーされず、インスタンスはコンテナ内のインストールツリーのデータを使用して実行されます。つまり、インスタンス固有のデータには持続性がありません。 このため、コンテナを実行する前に、スクリプトにこの条件のチェックを含めることをお勧めします。
- ISCDATADIRECTORYで指定された耐久性のある%SYSディレクトリが存在しない場合：
    - 指定された耐久性のある%SYSディレクトリが作成されます。
    - [耐久性のある%SYS ディレクトリ]のコンテンツにリストされているディレクトリとファイルは、

インストールされた場所から耐久性のある%SYS
ディレクトリにコピーされます(オリジナルはそのまま同じ場所に残ります）。
- インスタンスのすべての内部ポインタは耐久性のある%SYSディレクトリにリセットされ、インスタンスはそこに含まれるデータを使用します。

更新

アプリケーションが進化して新しいバージョン（コンテナ）がリリースされたときに、コードをいくつか実行する必要があるかもしれません。 コンパイル前後のフック、スキーマの移行、単体テストなどが考えられますが、結局のところ、任意のコードを実行する必要があるということです。
そのため、アプリケーションを管理するフレームワークが必要です。 以前の記事 の中で、このフレームワークの基本構造を概説しましたが、もちろん、特定のアプリケーション要件に合わせてかなり拡張することができます。

まとめ

コンテナ化されたアプリケーションの作成にはいくつかの考慮事項がありますが、InterSystems IRIS は、このプロセスを容易にするためのいくつかの機能を提供しています。

リンク

- 索引
- 記事のコード
- テストプロジェクト
- CDの構成を完成する
- Durable %SYSでCDの構成を完了する

#Docker #Git #Containerization #Deployment #Continuous Delivery #InterSystems IRIS #InterSystems IRIS for Health