Artigo
Evgeny Shvarov · Out. 6, 2020 13min de leitura

Open Exchange

Dockerfile e amigos ou como executar e colaborar com Projetos ObjectScript no InterSystems IRIS.

Olá, desenvolvedores!

Muitos de vocês publicam suas bibliotecas InterSystems ObjectScript no Open Exchange e GitHub.

Mas o que você faz para facilitar o uso e a colaboração do seu projeto por desenvolvedores?

Neste artigo, quero apresentar uma maneira fácil de iniciar e contribuir com qualquer projeto ObjectScript apenas copiando um conjunto padrão de arquivos para o seu repositório.

Vamos lá!

Copie esses arquivos deste repositório para o seu repositório:

Dockerfile

docker-compose.yml

Installer.cls

iris.script

settings.json

.dockerignore
.gitattributes
.gitignore

E você agora possui uma maneira padrão de lançar e colaborar com seu projeto. Abaixo está o longo artigo sobre como e por que isso funciona.

OBS.: Neste artigo, consideraremos projetos que podem ser executados no InterSystems IRIS 2019.1 e versões mais recentes.

Escolhendo o ambiente de lançamento para projetos do InterSystems IRIS

Normalmente, queremos que um desenvolvedor teste o projeto/biblioteca e tenha certeza de que será um exercício rápido e seguro.

Na minha humilde opinião, a abordagem ideal para lançar qualquer coisa nova de forma rápida e segura é através da utilização do contêiner Docker, que dá ao desenvolvedor uma garantia de que tudo o que ele/ela inicia, importa, compila e calcula é seguro para a máquina host e de que nenhum sistema ou código será destruído ou deteriorado. Se algo der errado, basta parar e remover o contêiner. Se a aplicação ocupa uma quantidade enorme de espaço em disco, você a limpa com o contêiner e seu espaço estará de volta. Se uma aplicação deteriora a configuração do banco de dados, você exclui apenas o contêiner com configuração deteriorada. É assim, simples e seguro.

O contêiner Docker oferece segurança e padronização.

A maneira mais simples de executar o contêiner Docker do InterSystems IRIS é executar uma imagem do IRIS Community Edition:

1. Instale o Docker desktop
2. Execute no terminal do sistema operacional o seguinte:

```
docker run --rm -p 52773:52773 --init --name my-iris store/intersystems/iris-community:2020.1.0.199.0
```

3. Em seguida, abra o Portal de Administração do IRIS em seu navegador host em:

http://localhost:52773/csp/sys/UtilHome.csp

4. Ou abra uma sessão no terminal:

   docker exec -it my-iris iris session IRIS

5. Pare o contêiner IRIS quando não precisar mais dele:

   docker stop my-iris

OK! Executamos o IRIS em um contêiner docker. Mas você deseja que um desenvolvedor instale seu código no IRIS e talvez faça algumas configurações. Isso é o que discutiremos a seguir.

Importando arquivos ObjectScript

O projeto InterSystems ObjectScript mais simples pode conter um conjunto de arquivos ObjectScript como classes, rotinas, macro e globais. Verifique o artigo sobre nomenclatura e estrutura de pastas proposta.

A questão é: como importar todo esse código para um contêiner IRIS?

Aqui é o momento em que o Dockerfile nos ajuda pois podemos usá-lo para pegar o contêiner IRIS padrão, importar todo o código de um repositório para o IRIS e fazer algumas configurações com o IRIS, se necessário. Precisamos adicionar um Dockerfile no repositório.

Vamos examinar o Dockerfile do repositório de modelos ObjectScript:

```
ARG IMAGE=store/intersystems/irishealth:2019.3.0.308.0-community
ARG IMAGE=store/intersystems/iris-community:2019.3.0.309.0
ARG IMAGE=store/intersystems/iris-community:2019.4.0.379.0
ARG IMAGE=store/intersystems/iris-community:2020.1.0.199.0
FROM $IMAGE

USER root

WORKDIR /opt/irisapp
RUN chown ${ISC_PACKAGE_MGRUSER}:${ISC_PACKAGE_IRISGROUP} /opt/irisapp
```

```
USER irisowner

COPY  Installer.cls .
COPY  src src
COPY iris.script /tmp/iris.script # run iris and initial

RUN iris start IRIS \
    && iris session IRIS < /tmp/iris.script
```

As primeiras linhas ARG definem a variável $IMAGE - que usaremos então em FROM. Isso é adequado para testar/executar o código em diferentes versões do IRIS, trocando-os apenas pelo que é a última linha antes do FROM para alterar a variável $IMAGE.

Aqui temos:

```
ARG IMAGE=store/intersystems/iris-community:2020.1.0.199.0

FROM $IMAGE
```

Isso significa que estamos pegando o IRIS 2020 Community Edition versão 199.

Queremos importar o código do repositório - isso significa que precisamos copiar os arquivos de um repositório para um contêiner do docker. As linhas abaixo ajudam a fazer isso:

```
USER root

WORKDIR /opt/irisapp
RUN chown ${ISC_PACKAGE_MGRUSER}:${ISC_PACKAGE_IRISGROUP} /opt/irisapp

USER irisowner

COPY  Installer.cls .
COPY  src src
```

USER root - aqui, mudamos o usuário para root para criar uma pasta e copiar arquivos no docker.

WORKDIR /opt/irisapp - nesta linha configuramos o diretório de trabalho no qual copiaremos os arquivos.

RUN chown ${ISCPACKAGEMGRUSER}:${ISCPACKAGEIRISGROUP} /opt/irisapp - aqui, damos os direitos ao usuário e grupo irisowner que executam o IRIS.

USER irisowner - trocando usuário de root para irisowner

COPY Installer.cls . - copiando o Installer.cls para a raiz do workdir. Não esqueça aqui do ponto.

COPY src src - copia os arquivos de origem da pasta src no repo para a pasta src no workdir no docker.

No próximo bloco, executamos o script inicial, onde chamamos o instalador e o código ObjectScript:

```
COPY iris.script /tmp/iris.script # executar o iris e iniciar
RUN iris start IRIS \
    && iris session IRIS < /tmp/iris.script
```

COPY iris.script / - copiamos iris.script para o diretório raiz. Ele contém o ObjectScript que desejamos chamar para configurar o contêiner.

RUN iris start IRIS\ - inicia o IRIS

&& iris session IRIS < /tmp/iris.script - inicia o terminal IRIS e insere o ObjectScript inicial nele.

Ótimo! Temos o Dockerfile, que importa arquivos no docker. Mas nos deparamos com outros dois arquivos: installer.cls e iris.script. Vamos examiná-los.

Installer.cls

```
Class App.Installer
{

XData setup
{
<Manifest>
  <Default Name="SourceDir" Value="#{$system.Process.CurrentDirectory()}src"/>
  <Default Name="Namespace" Value="IRISAPP"/>
  <Default Name="app" Value="irisapp" />

  <Namespace Name="${Namespace}" Code="${Namespace}" Data="${Namespace}" Create="yes"
 Ensemble="no">

    <Configuration>
      <Database Name="${Namespace}" Dir="/opt/${app}/data" Create="yes" Resource="%DB
_${Namespace}"/>

      <Import File="${SourceDir}" Flags="ck" Recurse="1"/>
    </Configuration>
    <CSPApplication Url="/csp/${app}" Directory="${cspdir}${app}"  ServeFiles="1" Rec
urse="1" MatchRoles=":%DB_${Namespace}" AuthenticationMethods="32"

    />
  </Namespace>

</Manifest>
}

ClassMethod setup(ByRef pVars, pLogLevel As %Integer = 3, pInstaller As %Installer.In
staller, pLogger As %Installer.AbstractLogger) As %Status [ CodeMode = objectgenerato
r, Internal ]
{
  #; Deixe o documento XGL gerar código para este método.
  Quit ##class(%Installer.Manifest).%Generate(%compiledclass, %code, "setup")
}

}
```

Francamente, não precisamos do Installer.cls para importar arquivos. Isso pode ser feito com uma linha. Mas frequentemente, além de importar o código, precisamos configurar a aplicação CSP, introduzir configurações de segurança, criar bancos de dados e namespaces.

Neste Installer.cls, criamos um novo banco de dados, namespace com o nome IRISAPP e criamos a aplicação /csp/irisapp padrão para este namespace.

Tudo isso realizamos no elemento :

```
<Namespace Name="${Namespace}" Code="${Namespace}" Data="${Namespace}" Create="yes" E
nsemble="no">

    <Configuration>
      <Database Name="${Namespace}" Dir="/opt/${app}/data" Create="yes" Resource="%DB
_${Namespace}"/>

      <Import File="${SourceDir}" Flags="ck" Recurse="1"/>
    </Configuration>
    <CSPApplication Url="/csp/${app}" Directory="${cspdir}${app}"  ServeFiles="1" Rec
urse="1" MatchRoles=":%DB_${Namespace}" AuthenticationMethods="32"

    />
  </Namespace>
```

E importamos todos os arquivos do SourceDir com a tag Import:

```
<Import File="${SourceDir}" Flags="ck" Recurse="1"/>
```

SourceDir aqui é uma variável, que é definida para o diretório/pasta src atual:

```
<Default Name="SourceDir" Value="#{$system.Process.CurrentDirectory()}src"/>
```

Uma classe Installer.cls com essas configurações nos dá a confiança de que criamos um novo banco de dados IRISAPP limpo, no qual importamos código ObjectScript arbitrário da pasta src.

iris.script

Aqui, você é bem-vindo para fornecer qualquer código de configuração ObjectScript inicial que deseja para iniciar seu contêiner IRIS.

Ex. Aqui carregamos e executamos o installer.cls e então criamos o UserPasswords sem expiração, apenas para evitar a primeira solicitação de alteração da senha, pois não precisamos desse prompt para o desenvolvimento.

```
; run installer to create namespace
do $SYSTEM.OBJ.Load("/opt/irisapp/Installer.cls", "ck")
set sc = ##class(App.Installer).setup()  zn "%SYS"
Do ##class(Security.Users).UnExpireUserPasswords("*") ; call your initial methods her
e
halt
```

docker-compose.yml

Por que precisamos de docker-compose.yml ? Não poderíamos simplesmente construir e executar a imagem apenas com Dockerfile? Sim, poderíamos. Mas docker-compose.yml simplifica a vida.

Normalmente, docker-compose.yml é usado para iniciar várias imagens docker conectadas a uma rede.

docker-compose.yml também pode ser usado para tornar a inicialização de uma imagem docker mais fácil quando lidamos com muitos parâmetros. Você pode usá-lo para passar parâmetros para o docker, como mapeamento de portas, volumes, parâmetros de conexão VSCode.

```
version: '3.6'
services:
```

```
  iris:
    build:
      context: .
      dockerfile: Dockerfile
    restart: always
    ports:
      - 51773
      - 52773
      - 53773
    volumes:
      - ~/iris.key:/usr/irissys/mgr/iris.key
      - ./:/irisdev/app
```

Aqui, declaramos o serviço iris, que usa o arquivo docker Dockerfile e expõe as seguintes portas do IRIS: 51773, 52773, 53773. Além disso, este serviço mapeia dois volumes: iris.key do diretório inicial da máquina host para a pasta IRIS onde é esperado, e ele mapeia a pasta raiz do código-fonte para a pasta /irisdev/app.

Docker-compose nos oferece o comando mais curto e unificado para construir e executar a imagem, quaisquer que sejam os parâmetros que você configurar no docker compose.

em qualquer caso, o comando para construir e lançar a imagem é:

```
$ docker-compose up -d
```

e para abrir o terminal IRIS:

```
$ docker-compose exec iris iris session iris

Node: 05a09e256d6b, Instance: IRIS

USER>
```

Além disso, docker-compose.yml ajuda a configurar a conexão para o plugin VSCode ObjectScript.

[.vscode/settings.json]

A parte relacionada às configurações de conexão da extensão ObjectScript é esta:

```
{
    "objectscript.conn" :{
      "ns": "IRISAPP",
      "active": true,
      "docker-compose": {
        "service": "iris",
        "internalPort": 52773
      }
    }

}
```

Aqui vemos as configurações, que são diferentes das configurações padrão do plugin VSCode ObjectScript.

Aqui, dizemos que queremos nos conectar ao namespace IRISAPP (que criamos com Installer.cls):

```
"ns": "IRISAPP",
```

e há uma configuração docker-compose, que informa que, no arquivo docker-compose dentro do serviço "iris", o VSCode se conectará à porta, para a qual 52773 está mapeado:

```
"docker-compose": {
        "service": "iris",
        "internalPort": 52773
      }
```

Se verificarmos o que temos para 52773, veremos que esta é a porta mapeada não definida para 52773:

```
ports:
      - 51773
      - 52773
      - 53773
```

Isso significa que uma porta aleatória disponível em uma máquina host será obtida e o VSCode se conectará a este IRIS no docker via porta aleatória automaticamente.

Este é um recurso muito útil, pois oferece a opção de executar qualquer quantidade de imagens do docker com IRIS em portas aleatórias e ter VSCode conectado a elas automaticamente.

E quanto a outros arquivos?

Nos também temos:

.dockerignore - arquivo que você pode usar para filtrar os arquivos da máquina host que você não deseja que sejam copiados para a imagem docker que você construir. Normalmente .git e .DSStore são linhas obrigatórias.

.gitattributes - atributos para git, que unificam terminações de linha para arquivos ObjectScript em fontes. Isso é muito útil se o repositório for colaborado por proprietários de Windows e Mac/Ubuntu.

.gitignore - arquivos, os quais você não deseja que o git rastreie o histórico de alterações. Normalmente, alguns arquivos ocultos no nível do sistema operacional, como .DSStore.

Ótimo!

Como tornar seu repositório executável em docker e amigável para colaboração?

1. Clone este repositório.
2. Copie todos esses arquivos:

Dockerfile

docker-compose.yml

Installer.cls

iris.script

settings.json

.dockerignore

.gitattributes
.gitignore

para o seu repositório.

Altere esta linha no Dockerfile para corresponder ao diretório com ObjectScript no repositório que você deseja importar para o IRIS (ou não altere se estiver na pasta /src).

É isso. E todos (e você também) terão seu código importado para o IRIS em um novo namespace IRISAPP.

Como as pessoas irão iniciar o seu projeto

o algoritmo para executar qualquer projeto ObjectScript no IRIS pode ser:

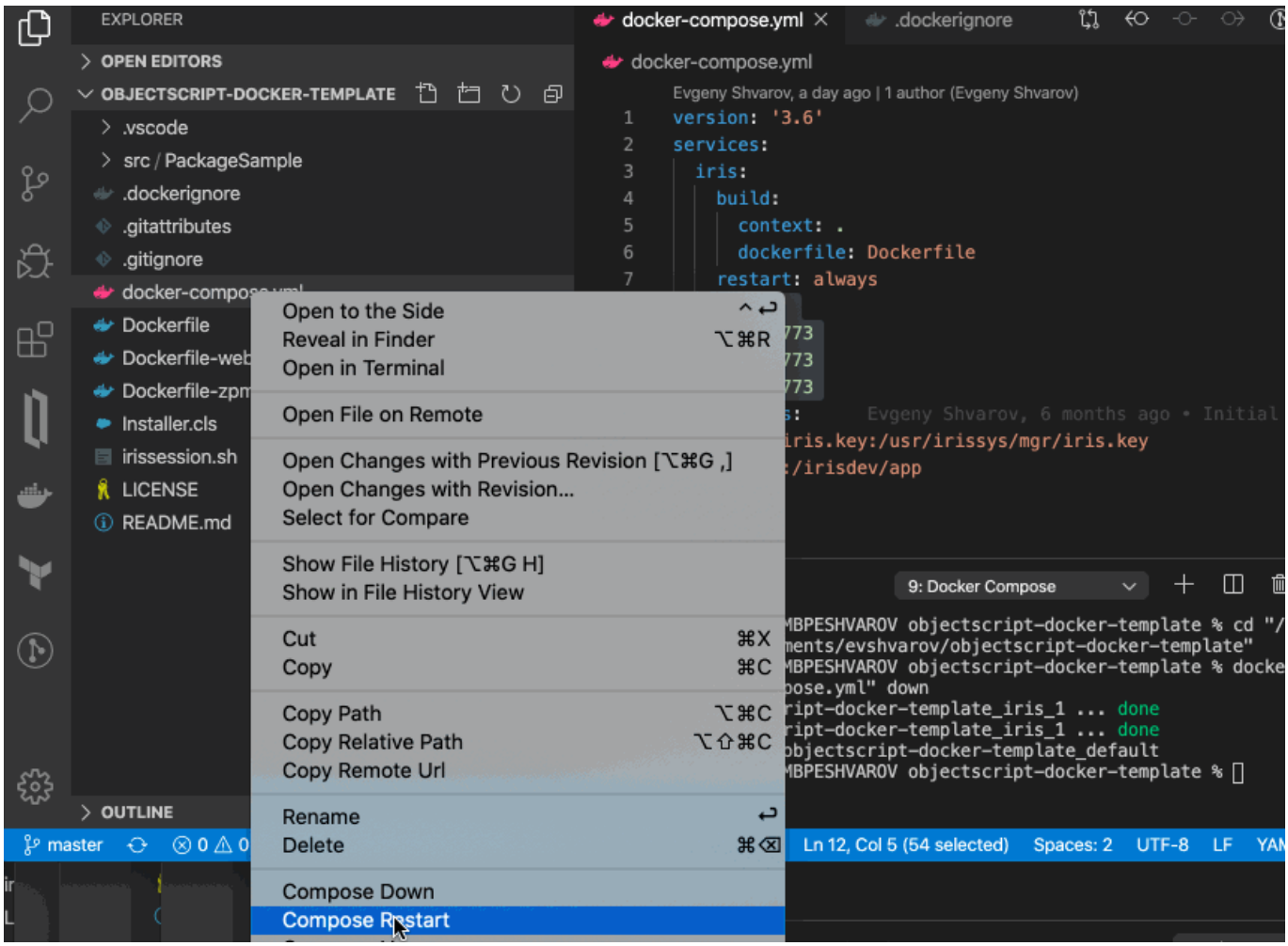
1. Clone o projeto Git localmente
2. Execute o projeto:

```
$ docker-compose up -d

$ docker-compose exec iris iris session iris

Node: 05a09e256d6b, Instance: IRIS

USER>zn "IRISAPP"
```

Como qualquer desenvolvedor pode contribuir para o seu projeto

1. Bifurque o repositório e clone o repositório git bifurcado localmente
2. Abra a pasta no VSCode (eles também precisam que as extensões Docker e ObjectScript estejam instaladas no VSCode)
3. Clique com o botão direito em docker-compose.yml->Reiniciar - VSCode ObjectScript irá conectar-se automaticamente e estará pronto para editar/compilar/depurar
4. Commit, Push e Pull as mudanças solicitadas em seu repositório

Aqui está um pequeno gif sobre como isso funciona:



É isso! Viva a programação!

#Ambiente de Desenvolvimento #Docker #Git #ObjectScript #Tutorial #InterSystems IRIS #Open Exchange
Confira o aplicativo relacionado no InterSystems Open Exchange

URL de origem:https://pt.community.intersystems.com/post/dockerfile-e-amigos-ou-como-executar-e-colaborar-com-projetos-objectscript-no-intersystems-iris