Artículo
Jose-Tomas Salvador · 22 sep, 2021 Lectura de 13 min

Open Exchange

Dockerfile y amigos. O cómo ejecutar y colaborar en los proyectos de ObjectScript en InterSystems IRIS

¡Hola desarrolladores!

Muchos de vosotros publicáis vuestras bibliotecas de InterSystems ObjectScript en Open Exchange y Github.

Pero, ¿qué puedes hacer para facilitar a los desarrolladores el uso y la colaboración en tu proyecto?

En este artículo, quiero presentar una forma sencilla de poner en marcha y contribuir en cualquier proyecto ObjectScript con solo copiar un conjunto estándar de archivos en tu repositorio.

¡Vamos!

TLDR: copia estos archivos desde el repositorio a tu repositorio:

Dockerfile

docker-compose.yml

Installer.cls

iris.script

settings.json

.dockerignore
.gitattributes
.gitignore

Y obtienes la forma estándar de poner en marcha y colaborar en tu proyecto. A continuación, se explica detalladamente cómo y por qué funciona esto así.

Nota: En este artículo, consideraremos los proyectos que se pueden ejecutar en InterSystems IRIS 2019.1 y versiones posteriores.

Cómo elegir el entorno de lanzamiento de los proyectos con InterSystems IRIS

Normalmente, queremos que un desarrollador pruebe el proyecto/biblioteca y esté seguro de que será un ejercicio rápido y seguro.

En mi modesta opinión, el enfoque ideal para poner en marcha cualquier cosa nueva de forma rápida y segura es el contenedor Docker, que ofrece al desarrollador la garantía de que todo lo que inicie, importe, compile y calcule es seguro para el servidor y ningún sistema o código será destruido o estropeado. Si algo sale mal, solo hay que parar y retirar el contenedor. Si la aplicación ocupa una enorme cantidad de espacio en el disco, la retiras junto con el contenedor y vuelves a tener espacio. Si una aplicación estropea la configuración de la base de datos, solo tienes que borrar el contenedor con la configuración estropeada. Así de simple y seguro.

El contenedor Docker ofrece seguridad y estandarización.

La forma más sencilla de ejecutar un contenedor Docker común de InterSystems IRIS es ejecutar una imagen de la Edición *Community* de IRIS:

1. Instala Docker desktop
2. Ejecuta lo siguiente en el terminal del sistema operativo:

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

3. A continuación, abre el Portal de Administración en tu navegador del servidor en:

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

4. O abre un terminal en IRIS:

   docker exec -it my-iris iris session IRIS

5. Deja de usar el contenedor IRIS cuando no lo necesites:

   docker stop my-iris

¡Muy bien! Ejecutamos IRIS en un contenedor Docker. Pero quieres que un desarrollador instale tu código en IRIS y tal vez haga algunos ajustes. Esto es lo que discutiremos a continuación.

Importación de archivos ObjectScript

El proyecto más sencillo de InterSystems ObjectScript puede contener un conjunto de archivos ObjectScript como clases, rutinas, macros y globales. Consulta el artículo sobre la convención de nombres y la estructura de carpetas propuesta.

La pregunta es: ¿cómo importar todo este código a un contenedor de IRIS?

Este es el momento en el que nos ayuda Dockerfile y podemos utilizarlo para coger el contenedor común de IRIS e importar todo el código de un repositorio a IRIS, y hacer algunos ajustes con IRIS si es necesario. Necesitamos añadir un Dockerfile en el repositorio.

Vamos a examinar el Dockerfile del repositorio ObjectScript template:

```
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 &lt; /tmp/iris.script
```

Las primeras líneas ARG establecen la variable $IMAGE, que después utilizaremos en FROM. Esto es adecuado para probar/ejecutar el código en diferentes versiones de IRIS cambiándolas solo por la última línea antes de FROM, para cambiar la variable $IMAGE.

Aquí tenemos:

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

FROM $IMAGE
```

Esto significa que estamos tomando la versión 199 de la Edición *Community* de IRIS 2020.

Queremos importar el código desde el repositorio. Eso significa que necesitamos copiar los archivos de un repositorio a un contenedor Docker. Las siguientes líneas ayudan a hacer eso:

```
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: aquí cambiamos de usuario a root para crear una carpeta y copiar archivos en Docker.

WORKDIR /opt/irisapp: en esta línea configuramos el workdir en el que copiaremos los archivos.

RUN chown ${ISC_PACKAGE_MGRUSER}:${ISC_PACKAGE_IRISGROUP} /opt/irisapp : aquí le damos los derechos al usuario irisowner y al grupo que ejecuta IRIS.

USER irisowner: cambia de usuario de root a irisowner

COPY Installer.cls . - copia Installer.cls a una raíz de workdir. No te olvides el punto.

COPY src src : copia archivos fuente desde la carpeta src del repositorio a la carpeta src de workdir en el Docker.

En el siguiente bloque ejecutamos el *script* inicial, donde llamamos al instalador y al código ObjectScript:

```
COPY iris.script /tmp/iris.script # run iris and initial
RUN iris start IRIS \
    && iris session IRIS &lt; /tmp/iris.script
```

COPY iris.script /: copiamos iris.script en el directorio raíz. Contiene el ObjectScript que queremos llamar para configurar el contenedor.

RUN iris start IRIS \ - inicia IRIS

&& iris session IRIS < /tmp/iris.script - inicia el terminal IRIS e introduce el ObjectScript inicial dentro de él.

¡Bien! Tenemos el Dockerfile, que importa archivos en Docker. Pero nos encontramos con otros dos archivos: installer.cls e iris.script. Vamos a examinarlos.

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 ]
{
  #; Let XGL document generate code for this method.
  Quit ##class(%Installer.Manifest).%Generate(%compiledclass, %code, "setup")
}

}
```

Sinceramente, no necesitamos Installer.cls para importar archivos. Esto podría hacerse con una sola línea. Pero a menudo, además de importar el código, es necesario configurar la aplicación CSP, introducir la configuración de seguridad, crear bases de datos y namespaces.

En este Installer.cls creamos una nueva base de datos y un namespace con el nombre IRISAPP, y creamos la aplicación /csp/irisapp predeterminada para este namespace.

Todo esto lo realizamos en el elemento <Namespace>:

E importamos todos los archivos desde SourceDir con la etiqueta Import:

SourceDir aquí es una variable, que se establece en el directorio actual/src:

Cuando Installer.cls cuenta con esta configuración nos da la confianza de que creamos una nueva base de datos IRISAPP limpia en la que importamos el código ObjectScript que queramos desde la carpeta src.

iris.script

Aquí puedes proporcionar cualquier código de configuración inicial de ObjectScript que quieras para iniciar tu contenedor IRIS.

Por ejemplo, aquí cargamos y ejecutamos installer.cls y después hacemos que las password de usuario no expiren solo para evitar la primera solicitud de cambio de contraseña porque no necesitamos esta línea de comandos para el desarrollo.

```
; 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 qué necesitamos docker-compose.yml? ¿No podríamos simplemente construir y ejecutar la imagen solo con Dockerfile? Sí, podríamos. Pero docker-compose.yml simplifica la vida.

Normalmente, docker-compose.yml se utiliza para lanzar varias imágenes Docker conectadas a una red.

docker-compose.yml también se podría utilizar para hacer más fácil el lanzamiento de una imagen Docker cuando tenemos una gran cantidad de parámetros. Se puede utilizar para pasar parámetros a Docker, como el mapeo de puertos, volúmenes, o parámetros de conexión 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
```

Aquí declaramos el servicio de iris, que utiliza el archivo Dockerfile y expone los siguientes puertos de IRIS: 51773, 52773, 53773. También este servicio mapea dos volúmenes: iris.key desde el directorio principal del equipo servidor a la carpeta IRIS donde se espera y mapea la carpeta raíz del código fuente a la carpeta /irisdev/app.

Docker-compose nos da el comando más corto y unificado para construir y ejecutar la imagen, independientemente de los parámetros que se hayan configurado en docker compose.

en cualquier caso, el comando para construir y presentar la imagen es:

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

y para abrir el terminal de IRIS:

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

Node: 05a09e256d6b, Instance: IRIS

USER>
```

Además, docker-compose.yml ayuda a configurar la conexión para el *plugin* VSCode-ObjectScript.

.vscode/settings.json

La parte que se refiere a la configuración de la conexión del complemento de ObjectScript es esta:

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

}
```

Es aquí donde vemos la configuración, que es diferente de la configuración predeterminada del *plugin* VSCode-ObjectScript.

Aquí podemos decir que queremos conectarnos al namespace IRISAPP (que creamos con Installer.cls):

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

y hay una configuración para docker-compose, que dice que en el archivo docker-compose dentro del servicio "iris", VSCode se conectará al puerto 52773 que se mapea a:

```
"docker-compose": {
```

```
        "service": "iris",
        "internalPort": 52773
      }
```

Si revisamos lo que tenemos para 52773 vemos que el puerto mapeado no está definido para 52773:

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

Esto significa que se tomará un puerto aleatorio disponible en un equipo servidor, y VSCode se conectará automáticamente a este IRIS en Docker por medio del puerto aleatorio.

Esta es una función muy útil, porque te da la opción de ejecutar cualquier cantidad de imágenes Docker con IRIS en puertos aleatorios y tener VSCode conectado a ellos de forma automática.

¿Qué sucede con los otros archivos?

También tenemos:

.dockerignore: archivo que se puede utilizar para filtrar los archivos del equipo servidor que no quieres que se copien en la imagen Docker que creaste. Normalmente . git y . DSStore son líneas obligatorias.

.gitattributes: atributos para git, que unifican los finales de línea de los archivos ObjectScript en las fuentes. Esto es muy útil si en el repositorio colaboran usuarios de Windows y Mac/Ubuntu.

.gitignore: archivos a los que no quieres que git siga el historial de cambios. Por lo general, son algunos archivos ocultos a nivel de sistema operativo, como .DSStore.

¡Bien!

¿Cómo hacer que tu repositorio sea ejecutable por Docker y fácil para colaborar?

1. Clona este repositorio.
2. Copia todos estos archivos:

Dockerfile

docker-compose.yml

Installer.cls

iris.script

settings.json

.dockerignore
.gitattributes
.gitignore

a tu repositorio.

Cambia esta línea en Dockerfile para que coincida con el directorio que cuenta con ObjectScript en el repositorio que quieres importar en IRIS (o no lo cambies si lo tienes en la carpeta /src).

Eso es todo. Todos (y tú también) tendrán tu código importado en IRIS en un nuevo namespace de IRISAPP.

Cómo lanzará la gente tu proyecto

La secuencia para ejecutar cualquier proyecto ObjectScript en IRIS sería:

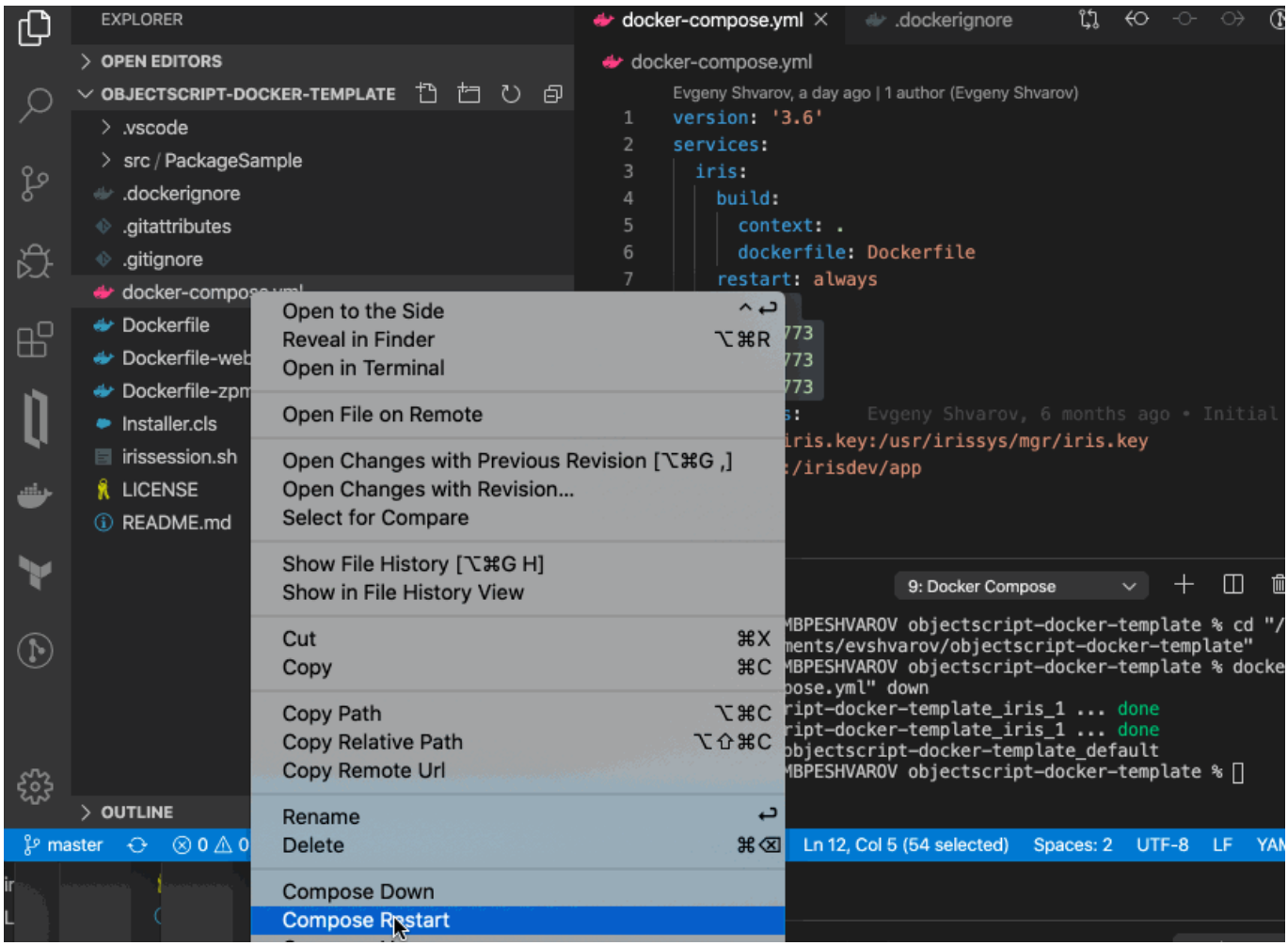
1. Clonar el proyecto localmente con git clone
2. Ejecutar el proyecto:

```
$ docker-compose up -d

$ docker-compose exec iris iris session iris

Node: 05a09e256d6b, Instance: IRIS

USER>zn "IRISAPP"
```

¿Cómo contribuirán los desarrolladores a tu proyecto?

1. Crea un fork del repositorio y clona el fork en local
2. Abre la carpeta en VSCode (también se necesita que las extensiones Docker y ObjectScript estén instaladas en VSCode)
3. Haz clic derecho en docker-compose.yml->Reinicia - VSCode ObjectScript se conectará automáticamente y estará listo para editar, compilar o depurar
4. Confirma, envía y extrae los cambios de solicitud a tu repositorio

Aquí os muestro en un breve gif cómo funciona esto:



¡Y esto es todo! Happy coding!

#Docker #Entorno de desarrollo #Git #ObjectScript #Tutorial #InterSystems IRIS #Open Exchange
Ir a la aplicación en InterSystems Open Exchange

URL de fuente:https://es.community.intersystems.com/post/dockerfile-y-amigos-o-c%C3%B3mo-ejecutar-y-colaborar-en-los-proyectos-de-objectscript-en