Article
Evgeny Shvarov · Feb 24, 2020 10m read

Open Exchange

Dockerfile and Friends or How to Run and Collaborate to ObjectScript Projects on InterSystems IRIS

Hi Developers!

Many of you publish your InterSystems ObjectScript libraries on Open Exchange and Github.

But what do you do to ease the usage and collaboration to your project for developers?

In this article, I want to introduce the way how to introduce an easy way to launch and contribute to any ObjectScript project just by copying a standard set of files to your repository.

Let's go!

TLDR - copy these files from the repository into your repository:

Dockerfile

docker-compose.yml

Installer.cls

iris.script

settings.json

.dockerignore
.gitattributes
.gitignore

And you get the standard way to launch and collaborate to your project. Below is the long article on how and why this works.

NB: In this article, we will consider projects which are runnable on InterSystems IRIS 2019.1 and newer.

Choosing the launch environment for InterSystems IRIS projects

Usually, we want a developer to try the project/library and be sure that this will be fast and safe exercise.

IMHO the ideal approach to launch anything new fast and safe is the Docker container which gives a developer a guarantee that anything he/she launches, imports, compiles and calculates is safe for the host machine and no system or code would be destroyed or spoiled. If something goes wrong you just stop and remove the container. If the application takes an enormous amount of disk space - you wipe out it with the container and your space is back. If an application spoils the database configuration - you just delete the container with spoiled configuration. Simple and safe like that.

Docker container gives you safety and standardization.

The simplest way to run vanilla InterSystems IRIS Docker container is to run an IRIS Community Edition image:

1. Install Docker desktop

2. Run in OS terminal the following:

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

3. Then open Management portal in your host browser on:

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

4. Or open a terminal to IRIS:

```
docker exec -it my-iris iris session IRIS
```

5. Stop IRIS container when you don't need it:

```
docker stop my-iris
```

OK! We run IRIS in a docker container. But you want a developer to install your code into IRIS and maybe make some settings. This is what we will discuss below.

Importing ObjectScript files

The simplest InterSystems ObjectScript project can contain a set of ObjectScript files like classes, routines, macro, and globals. Check the article on the naming convention and proposed folder structure.

The question is how to import all this code into an IRIS container?

Here is the momennt where Dockerfile helps us which we can use to take the vanilla IRIS container and import all the code from a repository to IRIS and do some settings with IRIS if we need. We need to add a Dockerfile in the repo.

Let's examine the Dockerfile from ObjectScript template repo:

```
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
```

First ARG lines set the $IMAGE variable - which we will use then in FROM. This is suitable to test/run the code in different IRIS versions switching them just by what is the last line before FROM to change the $IMAGE variable.

Here we have:

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

FROM $IMAGE
```

This means that we are taking IRIS 2020 Community Edition build 199.

We want to import the code from the repository - that means we need to copy the files from a repository into a docker container. The lines below help to do that:

```
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 - here we switch user to a root to create a folder and copy files in docker.

WORKDIR /opt/irisapp - in this line we setup the workdir in which we will copy files.

RUN chown ${ISC_PACKAGE_MGRUSER}:${ISC_PACKAGE_IRISGROUP} /opt/irisapp - here we give the rights to irisowner user and group which are run IRIS.

USER irisowner - switching user from root to irisowner

COPY Installer.cls . - coping Installer.cls to a root of workdir. Don't miss the dot, here.

COPY src src - copy source files from src folder in the repo to src folder in workdir in the docker.

In the next block we run the initial script, where we call installer and ObjectScript code:

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

COPY iris.script / - we copy iris.script into the root directory. It contains ObjectScript we want to call to setup the container.

RUN iris start IRIS\ - start IRIS

&& iris session IRIS < /tmp/iris.script - start IRIS terminal and input the initial ObjectScript to it.

Fine! We have the Dockerfile, which imports files in docker. But we faced two other files: installer.cls and iris.script Let's examine it.

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")
}

}
```

Frankly, we do not need Installer.cls to import files. This could be done with one line. But often besides importing code we need to setup the CSP app, introduce security settings, create databases and namespaces.

In this Installer.cls we create a new database and namespace with the name IRISAPP and create the default /csp/irisapp application for this namespace.

All this we perform in <Namespace> element:

```
<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>
```

And we import files all the files from SourceDir with Import tag:

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

SourceDir here is a variable, which is set to the current directory/src folder:

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

Installer.cls with these settings gives us confidence, that we create a clear new database IRISAPP in which we import arbitrary ObjectScript code from src folder.

iris.script

Here you are welcome to provide any initial ObjectScript setup code you want to start your IRIS container.

E.g. here we load and run installer.cls and then we make UserPasswords forever just to avoid the first request to change the password cause we don't need this prompt for development.

```
; 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

Why do we need docker-compose.yml - couldn't we just build and run the image just with Dockerfile? Yes, we could. But docker-compose.yml simplifies the life.

Usually, docker-compose.yml is used to launch several docker images connected to one network.

docker-compose.yml could be used to also make launches of one docker image easier when we deal with a lot of parameters. You can use it to pass parameters to docker, such as ports mapping, volumes, VSCode connection parameters.

```
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
```

Here we declare service iris, which uses docker file Dockerfile and which exposes the following ports of IRIS: 51773, 52773, 53773. Also this service maps two volumes: iris.key from home directory of host machine to IRIS folder where it is expected and it maps the root folder of source code to /irisdev/app folder.

Docker-compose gives us the shorter and unified command to build and run the image whatever parameters you setup in docker compose.

in any case, the command to build and launch the image is:

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

and to open IRIS terminal:

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

Node: 05a09e256d6b, Instance: IRIS

USER>
```

Also, docker-compose.yml helps to set up the connection for VSCode ObjectScript plugin.

.vscode/settings.json

The part, which relates to ObjectScript addon connection settings is this:

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

}
```

Here we see the settings, which are different from default settings of VSCode ObjectScript plugin.

Here we say, that we want to connect to IRISAPP namespace (which we create with Installer.cls):

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

and there is a docker-compose setting, which tells, that in docker-compose file inside service "iris" VSCode will connect to the port, which 52773 is mapped to:

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

If we check, what we have for 52773 we see that this is the mapped port is not defined for 52773:

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

This means that a random available on a host machine port will be taken and VSCode will connect to this IRIS on docker via random port automatically.

This is a very handy feature, cause it gives you the option to run any amount of docker images with IRIS on random ports and having VSCode connected to them automatically.

What about other files?

We also have:

.dockerignore - file which you can use to filter host machine files you don't want to be copied into docker image you build. Usually .git and .DSStore are mandatory lines.

.gitattributes - attributes for git, which unify line endings for ObjectScript files in sources. This is very useful if the repo is collaborated by Windows and Mac/Ubuntu owners.

.gitignore - files, which you don't want git to track the changes history for. Typically some hidden OS level files, like .DSStore.

Fine!

How to make your repository docker-runnable and collaboration friendly?

1. Clone this repository.

2. Copy all this files:

Dockerfile

docker-compose.yml

Installer.cls

iris.script

settings.json

.dockerignore
.gitattributes
.gitignore

to your repository.

Change this line in Dockerfile to match the directory with ObjectScript in the repo you want to import into IRIS (or don't change if you have it in /src folder).

That's it. And everyone (and you too) will have your code imported into IRIS in a new IRISAPP namespace.

How will people launch your project

the algorithm to execute any ObjectScript project in IRIS would be:

1. Git clone the project locally

2. Run the project:

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

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

Node: 05a09e256d6b, Instance: IRIS

USER>zn "IRISAPP"
```

How would any the developer contribute to your project

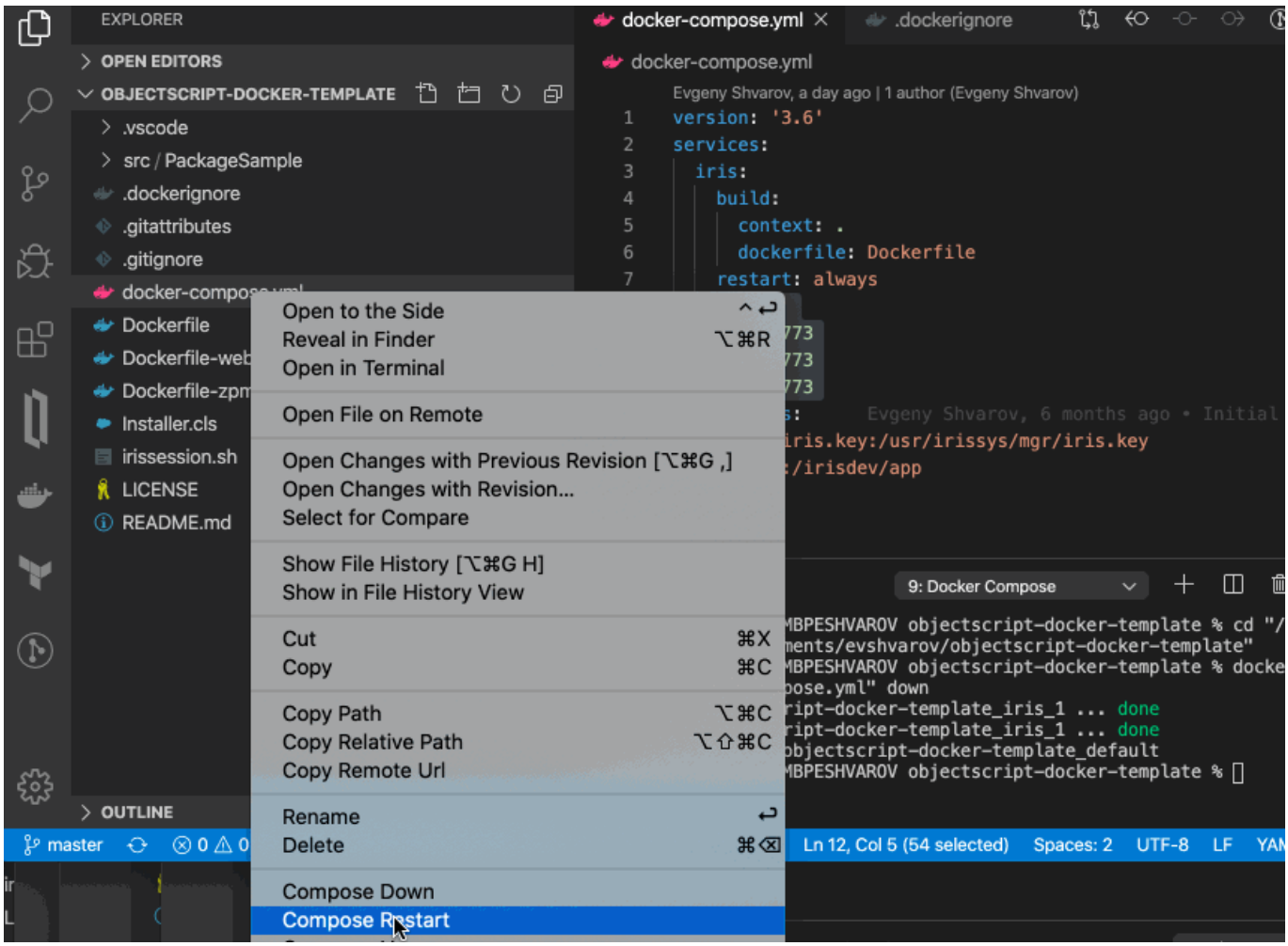
1. Fork the repository and git clone the forked repo locally

2. Open the folder in VSCode (they also need Docker and ObjectScript extensions are installed in VSCode)

3. Right-click on docker-compose.yml->Restart - VSCode ObjectScript will automatically connect and be ready to edit/compile/debug

4. Commit, Push and Pull request changes to your repository

Here is the short gif on how this works:



That's it! Happy coding!

#Best Practices #Development Environment #Docker #Git #ObjectScript #Tutorial #InterSystems IRIS #Open Exchange
Check the related application on InterSystems Open Exchange

Source URL:https://community.intersystems.com/post/dockerfile-and-friends-or-how-run-and-collaborate-objectscript-projects-intersystems-iris