Article
Janne Korhonen · Jun 4, 2020 5m read

Elementary CICD solution with Studio, Git (and Azure)

A lot of developers like to work with Studio and have been looking into source code version control such as GIT or into enabling modern development workflows like CICD or DevOps processes.

This article describe an elementary solution to get you started in CICD and DevOps, even if you are not yet ready to move to Atelier or forth coming VS Code approach which enable client side source code version control.

Step 1: Setting up version control with Studio and GIT

While Atelier and VS Code enable client side source code control with ecosystem of IDE plugins Studio offers server side hooks (code that is executed on there server whenever a document is loaded or saved) and writing your own source code hooks for the source control system(s) of your choice. This was the way to implement source code version control before version 2016.2 and introduction of Atelier IDE.

Nowadays you don ' t have to write your own wrapper for server side source code version control as there are multiple available as open source distributions. For the elementary solution we ' ll be usingcache- source-control, which exports classes from studio to specified directory after every successful compilation and also imports external classes which are added to the directory (e.g. from source code version control) on specific time interval.

Cache-source-control uses the following directory structure for exports and imports:
/src
/cls

/PackageName

Classname.cls

/mac Routine.mac

/int SomeFile.int

/dfi SomeAnotherFile.dfi

/inc def.inc

First step is to setup export/import directory, e.g. by initialising or cloning a GIT repository to be used for version control or the source code. Setting up GIT repository is outside the scope of this article, but there are multiple articles on internet that guide you how to get started.

I would consider adding **" /src/cls/Util/SourceControl.cls i" .gitignore** file in the repository, so that each developers unique configurations for source code control do not get replicated through the version control mechanism.

1. To initialise cache-source-control import SourceControl.cls.xml in target namespace and configure following parameters in the **Util.SourceControl** class:
ExpMode = 2 (UDL mode, each individual class is exported as text file instead of XML package)
SourceFolder = the folder you initiated or cloned ad the root of GIT repository

RefreshTime = interval for checking for new files in the folder (in seconds)

After you have made the changes compile Util.SourceControl.

2. Open Management Portal and in **Configuration/Additional Settings/Source Control/** set Util.SourceControl as source control class for the namespace.

3. Open the terminal and run **" do ##class(Util.SourceControl).Init() "**

Now you should have Studio based source control in place. Classes get exported to specified directory after each successful compilation and new files are imported to Studio on a defined interval.

You can commit, push and pull changes from the import/export directory to central repository with your choice of GIT tools.

Step 2: Automated deployment from GIT (with Azure)

Next step is to implement an automated deployment pipeline, which takes changes from source code version control (GIT) and deploys the changes to selected IRIS (or as in our case Health Connect) environment(s).

For this you need a continuous integration tool (such as Jenkins) which launches defined tasks based on e.g. GIT push, manual approval or both. Setting up the continuous integration tool and the pipeline is out of scope for this article. I suggest to consult your tools documentation for guidance.

Here is an example of a simple Azure pipeline that uses agent on target IRIS host (" default pool ") to run the tasks (see Azure DevOps documentation for details).
trigger:
- master
pool: Default

steps:
- script: ./install.sh
displayName: 'Run a installer manifest for HealthConnect'

- script: |
echo Done
displayName: 'Echo Done! '

In our elementary solution we ' ll use Installer Manifest and %Installer utility for source code deployment and configuration management. It allows describing and configuring a specific InterSystems IRIS configuration, rather than implementing a step-by-step installation process.

Here is an example of simple Installer Manifest that loads and compiles source code for one production and namespace (HSQS):

```
Include %occInclude
Class InterSystems.Installer
{
XData HCQSInstall [ XMLNamespace = INSTALLER ]

{ <Manifest>

<Namespace Name="HCQS" Create="no" Code="HCQS" Ensemble="1" Data="HCQS"> <Import File
="/home/user/myagent/_work/1/s/HCQS/src/cls/HCQS" Flags="ck"

IgnoreErrors="1" Recurse="1"/>
```

```
<Production Name="HCQS.Routing" AutoStart="1"/>

</Namespace> </Manifest>
}

ClassMethod setup(ByRef pVars, pLogLevel As %Integer = 3, pInstaller As %Installer.In
staller,
pLogger As %Installer.AbstractLogger)
As %Status [ CodeMode = objectgenerator, Internal ]

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

%code, "HCQSInstall") }

}
```

In elementary solution we use Health Connect and RHEL on Azure. As in the Azure pipeline example above the Installer Manifest needs to be loaded and compiled in IRIS before execution. Here is an example of simple Bash shell script doing just that:

```
#!/bin/bash
printf 'SuperUser\Password\nzn "USER"\ndo $system.OBJ.Load("/home/user/myagent/
work/1/s/InterSystems/Installer.cls", "c")\ndo ##class(InterSystems.Installer).setup()\n' | irissession
HEALTHCONNECT
```

Note: in production use you should consider e.g.using operating systems based authentication. It is not a good idea to expose passwords in the sell script.

With these two steps we have the elementary CICD pipeline up and running.

#Azure #Continuous Delivery #Continuous Integration #DevOps #Git #Studio #InterSystems IRIS