
Article

[Eduard Lebedyuk](#) · May 10, 2018 9m read

Continuous Delivery of your InterSystems solution using GitLab - Part VII: CD using containers

In this series of articles, I'd like to present and discuss several possible approaches toward software development with InterSystems technologies and GitLab. I will cover such topics as:

- Git 101
- Git flow (development process)
- GitLab installation
- GitLab Workflow
- Continuous Delivery
- GitLab installation and configuration
- GitLab CI/CD
- Why containers?
- Containers infrastructure
- CD using containers

In the [first article](#), we covered Git basics, why a high-level understanding of Git concepts is important for modern software development, and how Git can be used to develop software.

In the [second article](#), we covered GitLab Workflow - a complete software life cycle process and Continuous Delivery.

In the [third article](#), we covered GitLab installation and configuration and connecting your environments to GitLab

In the [fourth article](#), we wrote a CD configuration.

In the [fifth article](#), we talked about containers and how (and why) they can be used.

In the [sixth article](#) let's discuss main components you'll need to run a continuous delivery pipeline with containers and how they all work together.

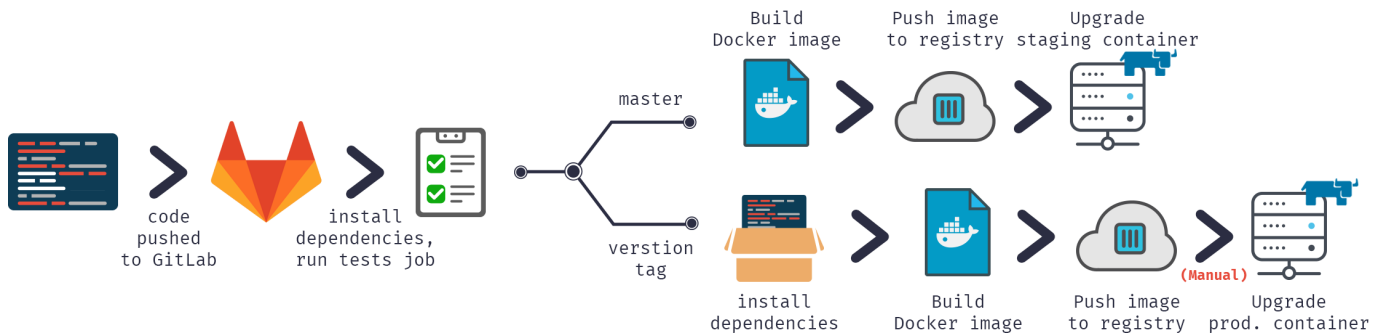
In this article, we'll build Continuous Delivery configuration discussed in the previous articles.

Workflow

In our Continuous Delivery configuration we would:

- Push code into GitLab repository
- Build docker image
- Test it
- Publish image to our docker registry
- Swap old container with the new version from the registry

Or in graphical format:



Let's start.

Build

First, we need to build our image.

Our code would be, as usual, stored in the repository, CD configuration in `gitlab-ci.yml` but in addition (to increase security) we would store several server-specific files on a build server.

`GitLab.xml`

Contains CD hooks code. It was developed in the [previous article](#) and available on [GitHub](#). This is a small library to load code, run various hooks and test code. As a preferable alternative, you can use [git submodules](#) to include this project or something similar into your repository. Submodules are better because it's easier to keep them up to date. One other alternative would be tagging releases on GitLab and loading them with [ADD](#) command.

`iris.key`

License key. Alternatively, it can be downloaded during container build rather than stored on a server. It's rather insecure to store in the repository.

`pwd.txt`

File containing default password. Again, it's rather insecure to store it in the repository. Also, if you're hosting prod environment on a separate server it could have a different default password.

`loadci.script`

Initial script, it:

- Enables OS authentication
- Loads `GitLab.xml`
- Initializes GitLab utility settings
- Loads the code

```
set sc = ##Class(Security.System).Get("SYSTEM",.Properties)
write:('sc) $System.Status.GetErrorText(sc)
set AuthEnabled = Properties("AuthEnabled")
set AuthEnabled = $zb(+AuthEnabled,16,7)
set Properties("AuthEnabled") = AuthEnabled
set sc = ##Class(Security.System).Modify("SYSTEM",.Properties)
write:('sc) $System.Status.GetErrorText(sc)
zn "USER"
do ##class(%SYSTEM.OBJ).Load(##class(%File).ManagerDirectory() _ "GitLab.xml", "cdk")
do ##class(isc.git.Settings).setSetting("hooks", "MyApp/Hooks/")
```

```
do ##class(isc.git.Settings).setSetting("tests", "MyApp/Tests/")
do ##class(isc.git.GitLab).load()
halt
```

Note that the first line is intentionally left empty.

As some settings can be server-specific, it's stored not in the repository, but rather separately. If this initial hook is always the same, you can just store it in the repository.

gitlab-ci.yml

Now, to Continuous Delivery configuration:

```
build image:
  stage: build
  tags:
    - test
  script:
    - cp -r /InterSystems/mount ci
    - cd ci
    - echo 'SuperUser' | cat - pwd.txt load_ci.script > temp.txt
    - mv temp.txt load_ci.script
    - cd ..
    - docker build --build-arg CI_PROJECT_DIR=$CI_PROJECT_DIR -t docker.domain.com/test/docker:$CI_COMMIT_REF_NAME .
```

What is going on here?

First of all, as [docker build](#) can access only subdirectories of a base build directory - in our case repository root, we need to copy our "secret" directory (the one with GitLab.xml, iris.key, pwd.txt and load_ci.script) into the cloned repository.

Next, first terminal access requires a user/pass so we'd add them to load_ci.script (that's what empty line at the beginning of load_ci.script is for btw).

Finally, we build docker image and tag it appropriately:

```
docker.domain.com/test/docker:$CI_COMMIT_REF_NAME
```

where `$CI_COMMIT_REF_NAME` is the name of a current branch. Note that the first part of the image tag should be named same as project name in GitLab, so it could be seen in GitLab Registry tab (instructions on tagging are available in Registry tab).

Dockerfile

Building docker image is done using [Dockerfile](#), here it is:

```
FROM docker.intersystems.com/intersystems/iris:2018.1.1.611.0

ENV SRC_DIR=/tmp/src
ENV CI_DIR=$SRC_DIR/ci
ENV CI_PROJECT_DIR=$SRC_DIR

COPY ./ $SRC_DIR

RUN cp $CI_DIR/iris.key $ISC_PACKAGE_INSTALLDIR/mgr/ \
  && cp $CI_DIR/GitLab.xml $ISC_PACKAGE_INSTALLDIR/mgr/ \
```

```
&& $ISC_PACKAGE_INSTALLDIR/dev/Cloud/ICM/changePassword.sh $CI_DIR/pwd.txt \  
&& iris start $ISC_PACKAGE_INSTANCENAME \  
&& irissession $ISC_PACKAGE_INSTANCENAME -U%SYS < $CI_DIR/load_ci.script \  
&& iris stop $ISC_PACKAGE_INSTANCENAME quietly
```

We start from the basic iris container.

First of all, we copy our repository (and "secret" directory) inside the container.

Next, we copy license key and GitLab.xml to mgr directory.

Then we change the password to the value from pwd.txt. Note that pwd.txt is deleted in this operation.

After that, the instance is started and loadci.script is executed.

Finally, iris instance is stopped.

Here's the job log (partial, skipped load/compilation logs):

```
Running with gitlab-runner 10.6.0 (a3543a27)  
  on docker 7b21e0c4  
Using Shell executor...  
Running on docker...  
Fetching changes...  
Removing ci/  
Removing temp.txt  
HEAD is now at 5ef9904 Build load_ci.script  
From http://gitlab.eduard.win/test/docker  
   5ef9904..9753a8d  master    -> origin/master  
Checking out 9753a8db as master...  
Skipping Git submodules setup  
$ cp -r /InterSystems/mount ci  
$ cd ci  
$ echo 'SuperUser' | cat - pwd.txt load_ci.script > temp.txt  
$ mv temp.txt load_ci.script  
$ cd ..  
$ docker build --build-arg CI_PROJECT_DIR=$CI_PROJECT_DIR -t docker.eduard.win/test/d  
ocker:$CI_COMMIT_REF_NAME .  
Sending build context to Docker daemon  401.4kB
```

```
Step 1/6 : FROM docker.intersystems.com/intersystems/iris:2018.1.1.611.0  
----> cd2e53e7f850  
Step 2/6 : ENV SRC_DIR=/tmp/src  
----> Using cache  
----> 68balcb00aff  
Step 3/6 : ENV CI_DIR=$SRC_DIR/ci  
----> Using cache  
----> 6784c34a9ee6  
Step 4/6 : ENV CI_PROJECT_DIR=$SRC_DIR  
----> Using cache  
----> 3757fa88a28a  
Step 5/6 : COPY ./ $SRC_DIR  
----> 5515e13741b0  
Step 6/6 : RUN cp $CI_DIR/iris.key $ISC_PACKAGE_INSTALLDIR/mgr/ && cp $CI_DIR/GitLab  
.xml $ISC_PACKAGE_INSTALLDIR/mgr/ && $ISC_PACKAGE_INSTALLDIR/dev/Cloud/ICM/changePas  
sword.sh $CI_DIR/pwd.txt && iris start $ISC_PACKAGE_INSTANCENAME && irissession $IS  
C_PACKAGE_INSTANCENAME -U%SYS < $CI_DIR/load_ci.script && iris stop $ISC_PACKAGE_INS  
TANCENAME quietly
```

```
---> Running in 86526183cf7c
.
Waited 1 seconds for InterSystems IRIS to start
This copy of InterSystems IRIS has been licensed for use exclusively by:
ISC Internal Container Sharding
Copyright (c) 1986-2018 by InterSystems Corporation
Any other use is a violation of your license agreement

%SYS>
1

%SYS>
Using 'iris.cpf' configuration file

This copy of InterSystems IRIS has been licensed for use exclusively by:
ISC Internal Container Sharding
Copyright (c) 1986-2018 by InterSystems Corporation
Any other use is a violation of your license agreement

1 alert(s) during startup. See messages.log for details.
Starting IRIS

Node: 39702b122ab6, Instance: IRIS

Username:
Password:

Load started on 04/06/2018 17:38:21
Loading file /usr/irissys/mgr/GitLab.xml as xml
Load finished successfully.

USER>

USER>

[2018-04-06 17:38:22.017] Running init hooks: before

[2018-04-06 17:38:22.017] Importing hooks dir /tmp/src/MyApp/Hooks/

[2018-04-06 17:38:22.374] Executing hook class: MyApp.Hooks.Global

[2018-04-06 17:38:22.375] Executing hook class: MyApp.Hooks.Local

[2018-04-06 17:38:22.375] Importing dir /tmp/src/

Loading file /tmp/src/MyApp/Tests/TestSuite.cls as udl

Compilation started on 04/06/2018 17:38:22 with qualifiers 'c'
Compilation finished successfully in 0.194s.

Load finished successfully.

[2018-04-06 17:38:22.876] Running init hooks: after

[2018-04-06 17:38:22.878] Executing hook class: MyApp.Hooks.Local

[2018-04-06 17:38:22.921] Executing hook class: MyApp.Hooks.Global
Removing intermediate container 39702b122ab6
---> dea6b2123165
```

```
[Warning] One or more build-args [CI_PROJECT_DIR] were not consumed
Successfully built dea6b2123165
Successfully tagged docker.domain.com/test/docker:master
Job succeeded
```

Note that I'm using [GitLab Shell executor](#) and not Docker executor. Docker executor is used when you need something from inside of the image, for example, you're building an Android application in java container and you only need an apk. In our case, we need a whole container and for that, we need Shell executor. So we're running Docker commands via GitLab Shell executor.

Run

We have our image, next let's run it. In the case of feature branches, we can just destroy old container and start the new one. In the case of the environment, we can run a temporary container and replace environment container in case tests succeed (that is left as an exercise to the reader).

Here is the script.

```
destroy old:
  stage: destroy
  tags:
    - test
  script:
    - docker stop iris-$CI_COMMIT_REF_NAME || true
    - docker rm -f iris-$CI_COMMIT_REF_NAME || true
```

This script destroys currently running container and always succeeds (by default docker fails if it tries to stop/remove non-existing container).

Next, we start the new image and register it as an environment. [Nginx container](#) automatically proxies requests using VIRTUAL_HOST environment variable and expose directive (to know which port to proxy).

```
run image:
  stage: run
  environment:
    name: $CI_COMMIT_REF_NAME
    url: http://$CI_COMMIT_REF_SLUG.docker.domain.com/index.html
  tags:
    - test
  script:
    - docker run -d
      --expose 52773
      --env VIRTUAL_HOST=$CI_COMMIT_REF_SLUG.docker.eduard.win
      --name iris-$CI_COMMIT_REF_NAME
      docker.domain.com/test/docker:$CI_COMMIT_REF_NAME
      --log $ISC_PACKAGE_INSTALLDIR/mgr/messages.log
```

Tests

Let's run some tests.

```
test image:
  stage: test
  tags:
    - test
  script:
    - docker exec iris- $\$CI\_COMMIT\_REF\_NAME$  irissession iris -U USER "##class(isc.git.GitLab).test()"
```

Publish

Finally, let's publish our image in the registry

```
publish image:
  stage: publish
  tags:
    - test
  script:
    - docker login docker.domain.com -u dev -p 123
    - docker push docker.domain.com/test/docker: $\$CI\_COMMIT\_REF\_NAME$ 
```


User/pass could be passed using [GitLab secret variables](#).









Now we can see the image in GitLab:

Container Registry

With the Docker Container Registry integrated into GitLab, every project can have its own space to store its Docker images.















Learn more about [Container Registry](#).

[test/docker](#) 

Tag	Tag ID	Size	Created	
master 	d7d13903f	435.96 MiB	32 minutes ago	
1-version-1-4 	afb1ff4b7	435.96 MiB	2 days ago	
preprod 	01882e781	435.96 MiB	2 days ago	
prod 	693c8c3f3	435.96 MiB	2 days ago	

And other developers can pull it from the registry. On environments tab all our environments are available for easy browsing:

Available **4** Stopped **0** New environment

Environment	Deployment	Job	Commit	Updated	
1-version-1-4	#1 by 	run image #652	 85ed9c67 Version 1.4, CI fix	4 days ago	 Re-deploy
master	#8 by 	run image #693	 85da69fd Remove mkdir	2 days ago	 Re-deploy
preprod	#3 by 	run image #663	 ed8fb741  Merge branch 'master' into 'preprod'	4 days ago	 Re-deploy
prod	#4 by 	run image #668	 e3260de2  Merge branch 'preprod' into 'prod'	4 days ago	 Re-deploy

Conclusion

In this series of articles, I covered general approaches to the Continuous Delivery. It is an extremely broad topic and this series of articles should be seen more as a collection of recipes rather than something definitive. If you want to automate building, testing and delivery of your application Continuous Delivery in general and GitLab in particular is the way to go. Continuous Delivery and containers allows you to customize your workflow as you need it.

Links

- [Code for the article](#)
- [Test project](#)
- [Complete CD configuration](#)

What's next

That's it. I hope I covered the basics of continuous delivery and containers.

There's a bunch of topics I did not talked about (so maybe later), especially towards the containers:

- Data could be persisted outside of container, here's the [documentation](#) on that
- Orchestration platforms like [kubernetes](#)
- [InterSystems Cloud Manager](#)
- Environment management - creating temporary environments for testing, removing old environments after feature branch merging
- [Docker compose](#) for multi-container deployments
- Decreasing docker image size and build times
- ...

[#Best Practices](#) [#Change Management](#) [#Containerization](#) [#Continuous Delivery](#) [#Continuous Integration](#) [#Git](#) [#Cache](#)

Source

URL: <https://community.intersystems.com/post/continuous-delivery-your-intersystems-solution-using-gitlab-part-vii-cd-using-containers>