Article
Eduard Lebedyuk · Apr 6, 2018 5m read

Continuous Delivery of your InterSystems solution using GitLab - Part VI: Containers infrastructure

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**
- GitLab CI/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.

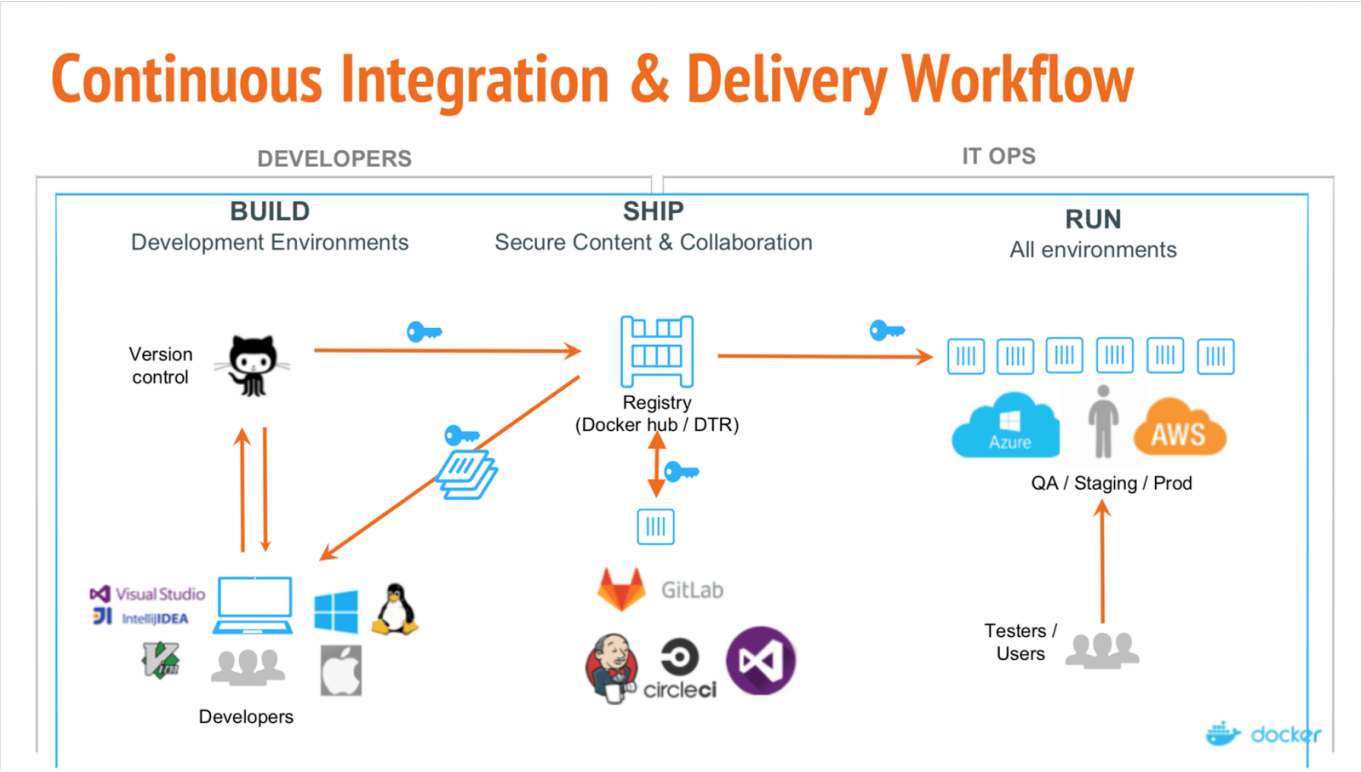
In the third article, we covered GitLab installation and configuration and connecting your environments to GitLab

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 this article let's discuss main components you'll need to run a continuous delivery pipeline with containers and how they all work together.

Our configuration would look like this:



Here we can see the separation of the three main stages:

- Build
- Ship
- Run

Build

In the previous parts, the build was often incremental - we calculated the difference between current environment and current codebase and modified our environment to correspond to the codebase. With containers, each build is a full build. The result of a build is an image that could be run anywhere with to dependencies.

Ship

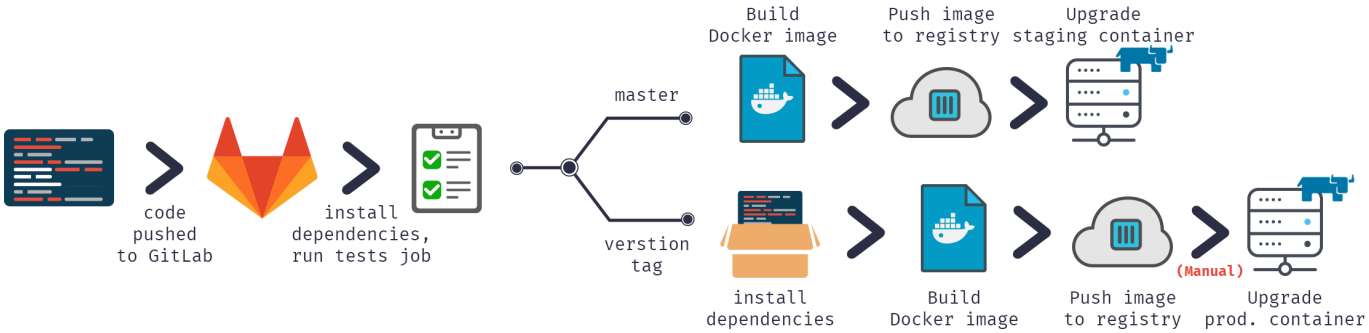
After our image is built and passed the tests it is uploaded into the registry - specialized server to host docker images. There it can replace the previous image with the same tag. For example, due to new commit to the master branch we built the new image (project/version:master) and if tests passed we can replace the image in the registry with the new one under the same tag, so everyone who pulls project/version:master would get a new version.

Run

Finally, our images are deployed. CI solution such as GitLab can control that or a specialized orchestrator, but the point is the same - some images are executed, periodically checked for health and updated if a new image version becomes available.

Check out docker webinar explaining these different stages.

Alternatively, from the commit point of view:



In our 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

To do that we'll need:

- Docker
- Docker registry
- Registered domain (optional but preferable)
- GUI tools (optional)

Docker

First of all, we need to run docker somewhere. I'd recommend starting with one server with more mainstream Linux flavor like Ubuntu, RHEL or Suse. Don't use cloud-oriented distributions like CoreOS, RancherOS etc. - they are not really aimed at the beginners. Don't forget to switch storage driver to devicemapper.

If we're talking about big deployments then using container orchestration tools like Kubernetes, Rancher or Swarm can automate most tasks but we're not going to discuss them (at least in this part).

Docker registry

That's a first container we need to run, and it is a stateless, scalable server-side application that stores and lets you distribute Docker images.
You should use the Registry if you want to:

- tightly control where your images are being stored
- fully own your images distribution pipeline
- integrate image storage and distribution tightly into your in-house development workflow

Here's registry documentation.

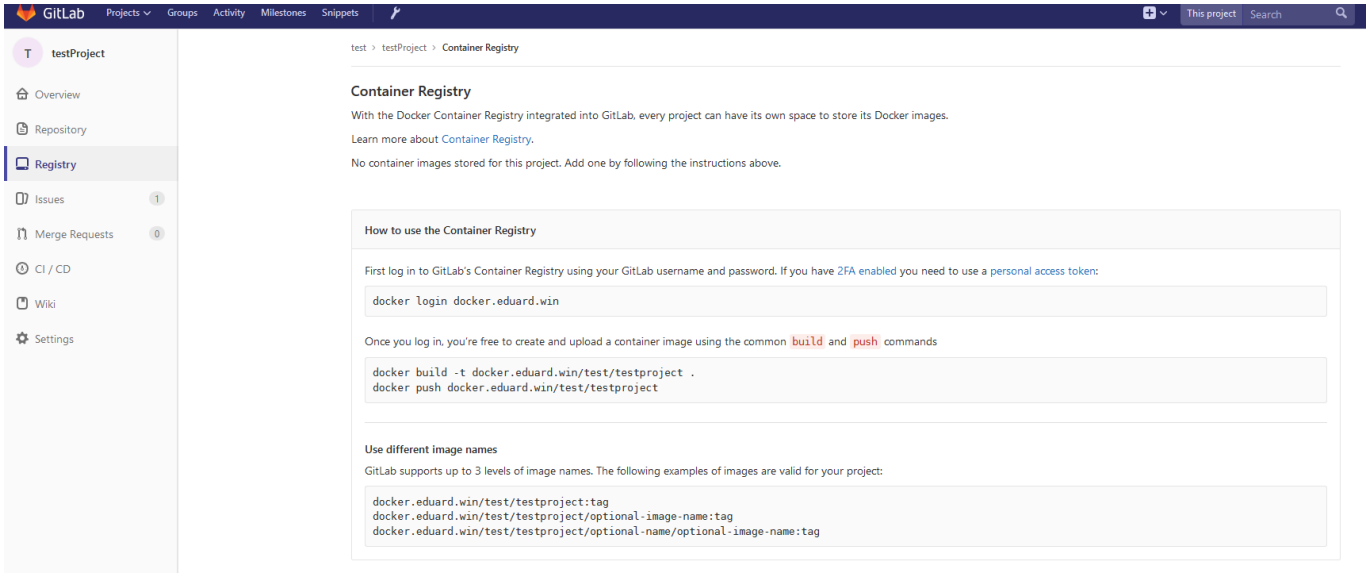
Connecting registry and GitLab

> Note: GitLab includes in-built registry. You can run it instead of external registry. Read GitLab docs linked in this paragraph.

To connect your registry to GitLab, you'll need to run your registry with HTTPS support - I use Let's Encrypt to get certificates, and I followed this Gist to get certificates and passed them into a container. After making sure that the registry is available over HTTPS (you can check from browser) follow these instructions on connecting registry to GitLab. These instructions differ based on what you need and your GitLab installation, in my case configuration was adding registry certificate and key (properly named and with correct permissions) to /etc/gitlab/ssl and these lines to /etc/gitlab/gitlab.rb:

```
registry_external_url 'https://docker.domain.com'
gitlab_rails['registry_api_url'] = "https://docker.domain.com"
```

And after reconfiguring GitLab I could see a new Registry tab where we're provided with the information on how to properly tag newly built images, so they would appear here.



Domain

In our Continuous Delivery configuration, we would automatically build an image per branch and if the image passes tests then we'd publish it in the registry and run it automatically, so our application would be automatically available in all "states", for example, we can access:

- Several feature branches at `<featureName>.docker.domain.com`
- Test version at `master.docker.domain.com`
- Preprod version at `preprod.docker.domain.com`
- Prod version at `prod.docker.domain.com`

To do that we need a domain name and add a wildcard DNS record that points `*.docker.domain.com` to the IP address of `docker.domain.com`. Other option would be to use different ports.

Nginx proxy

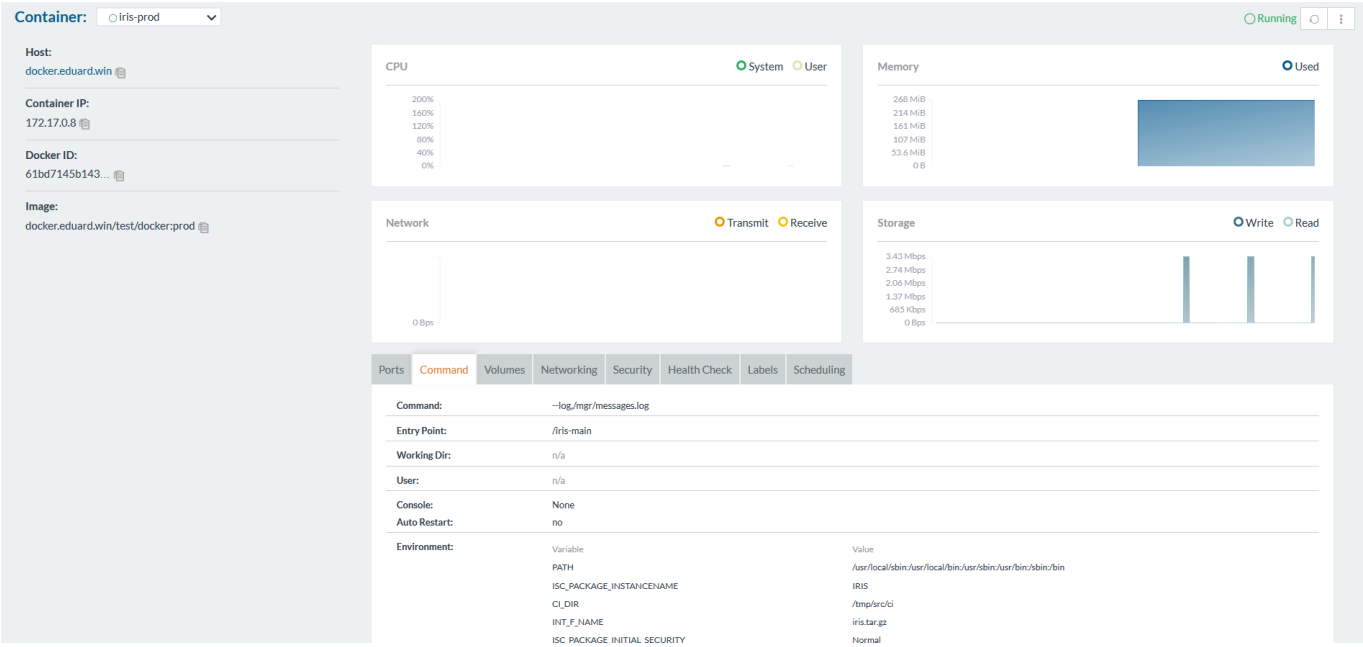
As we have several feature branches we need to redirect subdomains automatically to the correct container. To do that we can use Nginx as a reverse proxy. Here's a guide.

GUI tools

To start working with containers you can use either command line or one of the GUI interfaces. There are many available, for example:

- Rancher
- MicroBadger
- Portainer
- Simple Docker UI
- ...

They allow you to create containers and manage them from the GUI instead of CLI. Here's how Rancher looks like:



GitLab runner

Same as before to execute scripts on other servers we'll need to install GitLab runner. I discussed that in the third article.

Note that you'll need to use 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.

Conclusion

It's easy to start running containers and there are many tools to choose from.

Continuous Delivery using containers differs from the usual Continuous Delivery configuration in several ways:

- Dependencies are satisfied at build time and after image is built you don't need to think about dependencies.
- Reproducibility - you can easily reproduce any existing environment by running the same container locally.
- Speed - as containers do not have anything except what you explicitly added, they can be built faster more importantly they are built once and used whenever required.
- Efficiency - same as above containers produce less overhead than, for example, VMs.
- Scalability - with orchestration tools you can automatically scale your application to the workload and consume only resources you need right now.

What's next

In the next article, we'll talk create a CD configuration that leverages InterSystems IRIS Docker container.

#Change Management #Containerization #Continuous Delivery #Continuous Integration #Docker #Caché