Article
Eduard Lebedyuk · Mar 7, 2018 7m read

Continuous Delivery of your InterSystems solution using GitLab - Part II: GitLab workflow

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

In the previous 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. Still, our focus was on the implementation part of software development, but this part presents:

- **GitLab Workflow** - a complete software life cycle process - from idea to user feedback
- **Continuous Delivery** - software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently.

GitLab Workflow

GitLab Workflow is a logical sequence of possible actions to be taken during the entire lifecycle of the software development process.

GitLab Workflow takes into account the GitLab Flow, which we discussed in a previous article. Here's how it looks like:

1. Idea: every new proposal starts with an idea.
2. Issue: the most effective way to discuss an idea is creating an issue for it. Your team and your collaborators can help you to polish and improve it in the issue tracker.
3. Plan: once the discussion comes to an agreement, it's time to code. But first, we need to prioritize and organize our workflow by assigning issues to milestones and issue board.
4. Code: now we're ready to write our code, once we have everything organized.
5. Commit: once we're happy with our draft, we can commit our code to a feature-branch with version control. GitLab flow was explained in detail in the previous article.
6. Test: run our scripts using GitLab CI, to build and test our application.
7. Review: once our script works and our tests and builds succeeds, we are ready to get our code reviewed and approved.
8. Staging: now it's time to deploy our code to a staging environment to check if everything worked as we were expecting or if we still need adjustments.
9. Production: when we have everything working as it should, it's time to deploy to our production environment!
10. Feedback: now it's time to look back and check what stage of our work needs improvement.

Again, the process itself is not new (or unique to GitLab for that matter) and can be achieved with other tools of choice.

Let's discuss several of these stages and what they entail. There is also documentation available.

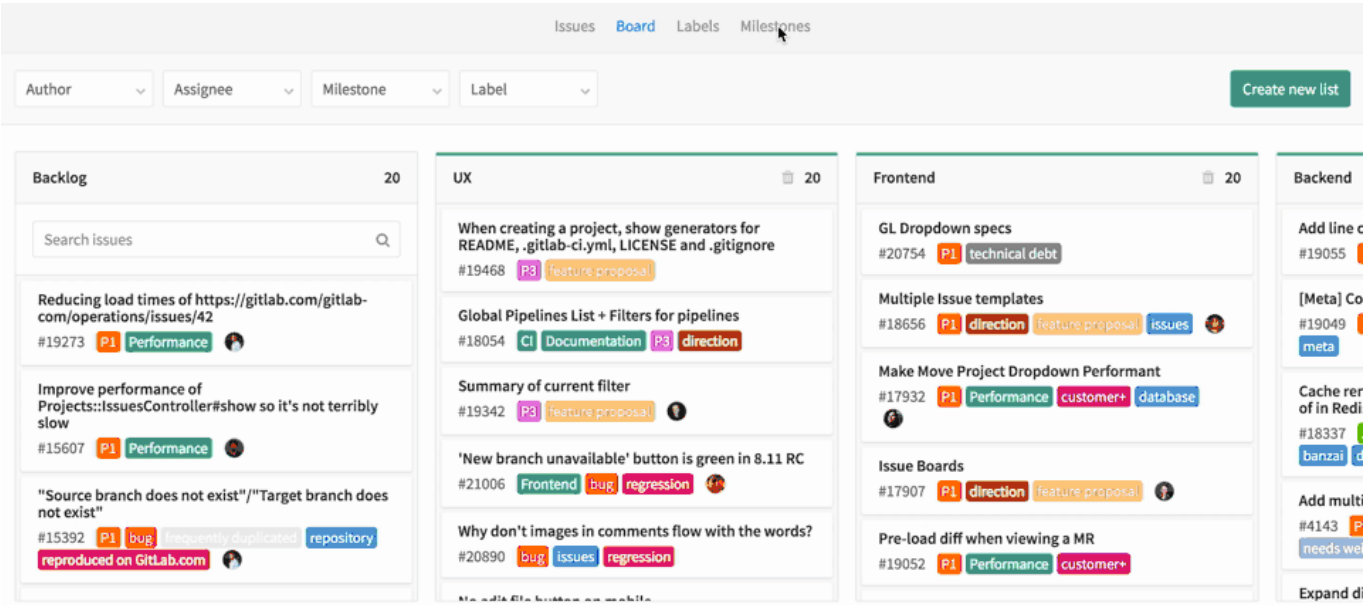
Issue and Plan

The beginning stages of GitLab workflow are centered on an **issue** - a feature or a bug or some other kind of semantically separate piece of work.

The issue has several purposes such as:

- Management: an issue has a due date, an assigned person, due date, time spent and estimates, etc. to help track with issue resolving.
- Administrative: an issue is a part of a milestone, kanban board that allows us to track our software as it progresses from version to version.
- Development: an issue has a discussion and commits associated with it.

Planning stage allows us to group issues by their priority, milestone, kanban board and have an overview for that.



Development was discussed in the previous part, just follow any git flow you wish. After we developed our new feature and merged it into master - what happens next?

Continuous Delivery

Continuous Delivery is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time. It aims at building, testing, and releasing software faster and more frequently. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for continuous delivery.

Continuous Delivery in GitLab

In GitLab continuous delivery configuration is defined on a per-repository basis as a YAML config file.

- Continuous delivery configuration is a series of consecutive **stages**.
- Each stage has one or several **scripts** that are executed in parallel.

Script defines one action and what conditions should be met to execute it:

- What to do (run OS command, run a container)?
- When to run the script:
    - What triggers it (commit to a specific branch)?
    - Do we run it if previous stages failed?
- Run manually or automatically?
- In what environment to run the script?
- What artifacts to save after executing the scripts (they are uploaded from the environment into GitLab for easier access)?

Environment - is a configured server or container in which you can run your scripts.

Runners execute scripts in specific environments. They are connected to your GitLab and execute scripts as required.

Runner can be deployed on a server, a container or even your local machine.

How does Continuous Delivery happen?

1. New commit is pushed into the repository.
2. GitLab checks Continuous Delivery configuration.
3. Continuous Delivery configuration contains all possible scripts for all cases so they are filtered to a set of scripts that should be run for this specific commit (for example a commit to master branch triggers only actions related to a master branch). This set is called a **pipeline**.
4. Pipeline is executed in a target environment, the results of the execution are saved and displayed in GitLab.

For example, here's one pipeline executed after a commit to a master branch: 

It consists of four stages, executed consecutively

1. Load stage loads code into a server
2. Test stage runs unit tests
3. Package stage consists of two scripts run in parallel:
    - Build client
    - Export server code (for information purposes mainly)
4. Deploy stage moves built client into the web-server directory.

As we can see, each script has run successfully, if one of the scripts failed by default later scripts would not be run (but we can change this behavior):

If we open the script we can see the log and determine why it failed:

```
Running with gitlab-runner 10.4.0 (857480b6)
 on test runner (ab34a8c5)
Using Shell executor...
Running on gitlab-test...
Fetching changes...
Removing diff.xml
Removing full.xml
Removing index.html
Removing tests.html
HEAD is now at a5bf3e8 Merge branch '4-versiya-1-0' into 'master'
From http://gitlab.eduard.win/test/testProject
 * [new branch] 5-versiya-1-1 -> origin/5-versiya-1-1
 a5bf3e8..442a4db master -> origin/master
 d28295a..42a10aa preprod -> origin/preprod
 3ac4b21..7edf7f4 prod -> origin/prod
Checking out 442a4db1 as master...
Skipping Git submodules setup
$ csession ensemble "##class(isc.git.GitLab).loadDiff()"

[2018-03-06 13:58:19.188] Importing dir /home/gitlab-
runner/builds/ab34a8c5/0/test/testProject/

[2018-03-06 13:58:19.188] Loading diff between a5bf3e8596d842c5cc3da7819409ed81e62c31
e3 and 442a4db170aa58f2129e5889a4bb79261aa0cad0

[2018-03-06 13:58:19.192] Variable modified
var=$lb("MyApp/Info.cls")

Load started on 03/06/2018 13:58:19
Loading file /home/gitlab-
runner/builds/ab34a8c5/0/test/testProject/MyApp/Info.cls as udl
Load finished successfully.

[2018-03-06 13:58:19.241] Variable items
var="MyApp.Info.cls"
var("MyApp.Info.cls")=""

Compilation started on 03/06/2018 13:58:19 with qualifiers 'cuk /checkuptodate=expand
edonly'
Compiling class MyApp.Info
Compiling routine MyApp.Info.1
ERROR: MyApp.Info.cls(version+2) #1003: Expected space : '}' : Offset:14 [zversion+1^
MyApp.Info.1]
 TEXT:  quit, "1.0" }
Detected 1 errors during compilation in 0.010s.

[2018-03-06 13:58:19.252] ERROR #5475: Error compiling routine: MyApp.Info.1. Errors:
 ERROR: MyApp.Info.cls(version+2) #1003: Expected space : '}' : Offset:14 [zversion+1
^MyApp.Info.1]
 > ERROR #5030: An error occurred while compiling class 'MyApp.Info'
ERROR: Job failed: exit status 1
```

Compilation error caused our script to fail.

Conclusion

- GitLab supports all main stages of software development.
- Continuous delivery can help you to automate tasks of building, testing and deploying your software.

Links

- Part I: Git
- Introduction to GitLab workflow
- GitLab CI/CD documentation
- GitLab flow
- Code for this article

What's next?

In the next article, we'll:

- Install GitLab.
- Connect it to several environments with InterSystems products installed.
- Write a Continuous Delivery configuration.

Let's discuss how our Continuous Delivery should work.

First of all, we need several environments and branches that correspond to them. Code goes into this branch and delivered to the target environment:

Environment	Branch	Delivery	Who can commit	Who can merge
Test	master	Automatic	Developers Owners	Developers Owners
Preprod	preprod	Automatic	No one	Owners
Prod	prod	Semiautomatic (press button to deliver)	No one	Owners

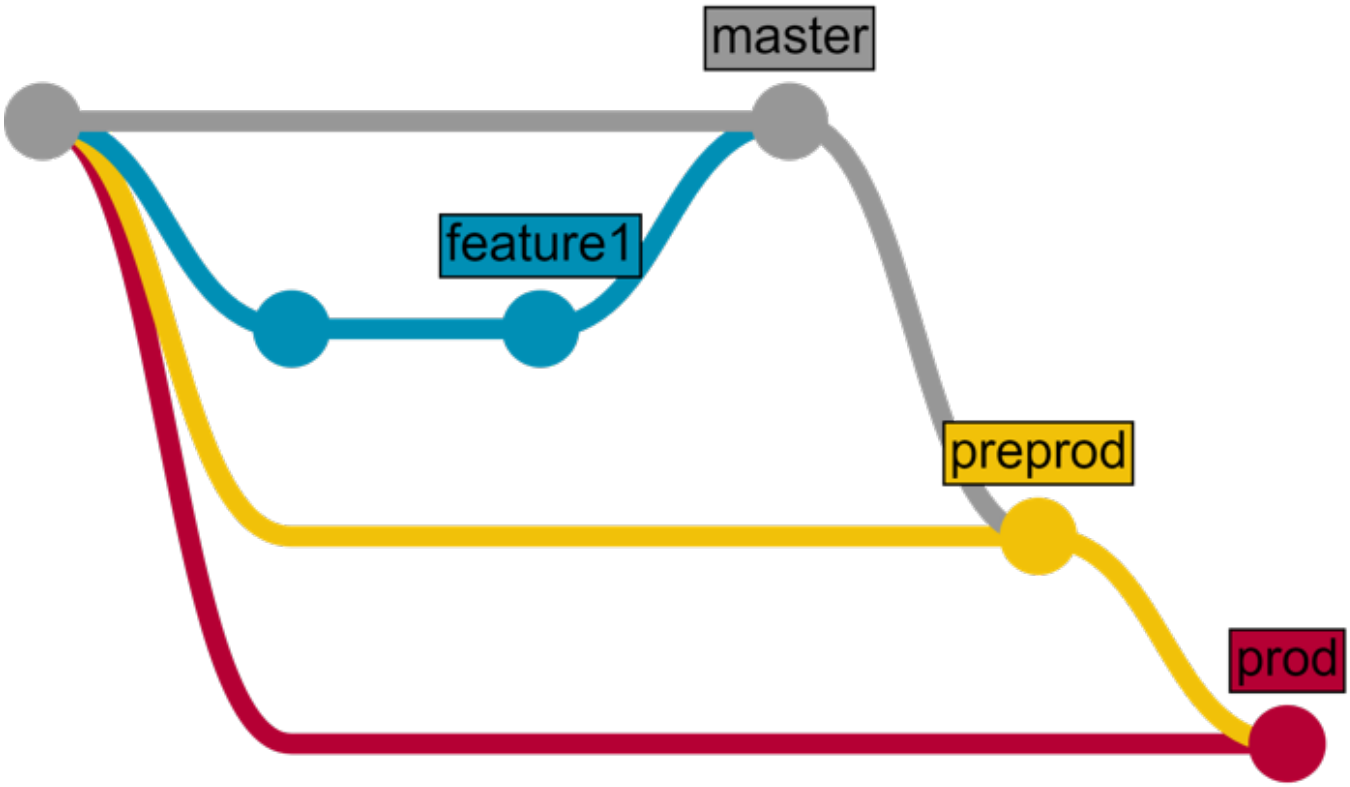
And as an example, we'll develop one new feature using GitLab flow and deliver it using GitLab CD.

1. Feature is developed in a feature branch.
2. Feature branch is reviewed and merged into the master branch.
3. After a while (several features merged) master is merged into preprod
4. After a while (user testing, etc.) preprod is merged into prod

Here's how it would look like:

1. Development and testing
    - Developer commits the code for the new feature into a separate feature branch
    - After feature becomes stable, the developer merges our feature branch into the master branch
    - Code from the master branch is delivered to the Test environment, where it's loaded and tested
2. Delivery to the Preprod environment
    - Developer creates merge request from master branch into the preprod branch
    - Repository Owner after some time approves merge request
    - Code from the preprod branch is delivered to the Preprod environment
3. Delivery to the Prod environment
    - Developer creates merge request from preprod branch into the prod branch
    - Repository Owner after some time approves merge request
    - Repository owner presses "Deploy" button
    - Code from prod branch is delivered to the Prod environment

Or the same but in a graphic form:



#Beginner #Best Practices #Change Management #Containerization #Continuous Integration #Deployment #Docker #Git #System Administration #Caché

Source URL:https://community.intersystems.com/post/continuous-delivery-your-intersystems-solution-using-gitlab-part-ii-gitlab-workflow