Article
Eduard Lebedyuk · Mar 1, 2018 6m read

Continuous Delivery of your InterSystems solution using GitLab - Part I: Git

Everybody has a testing environment.

Some people are lucky enough to have a totally separate environment to run production in.

-- Unknown

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
- GitLab CI/CD
- CI/CD with containers

This first part deals with the cornerstone of modern software development - Git version control system and various Git flows.

Git 101

While the main topic we're going to discuss is software development in general and how GitLab can enable us in that endeavor, Git, or rather several underlying high-level concepts in the Git design that are important for the better understanding of the later concepts.

That said, **Git** is a version control system, based on these ideas (there are many more, that's the most important ones):

- **Non-linear development** means that while our software is released consequentially from version 1 to 2 to 3, under the table the move from version 1 to 2 is done in parallel - several developers develop a number of features/bug-fixes simultaneously.
- **Distributed development** means that developer is independent from one central server or other developers and can develop in his own environment easily.
- **Merging** - previous two ideas bring us to the situation where many different versions of truth exist simultaneously and we need to unite them back into one complete state.

Now, I'm not saying that Git invented these concepts. No. Rather Git made them easy and popular and that, coupled with several related innovations, i.e. infrastructure as code/containerization changed software development.

Core git terms

Repository is a project that stores data and meta-information about the data.

- "Physically" repository is a directory on a disk.
- Repository stores files and directories.
- Repository also stores a complete history of changes for each file.

Repository can be stored:

- Locally, on your own computer
- Remotely on a remote server

But there's no particular difference between local and remote repositories from the git point of view.

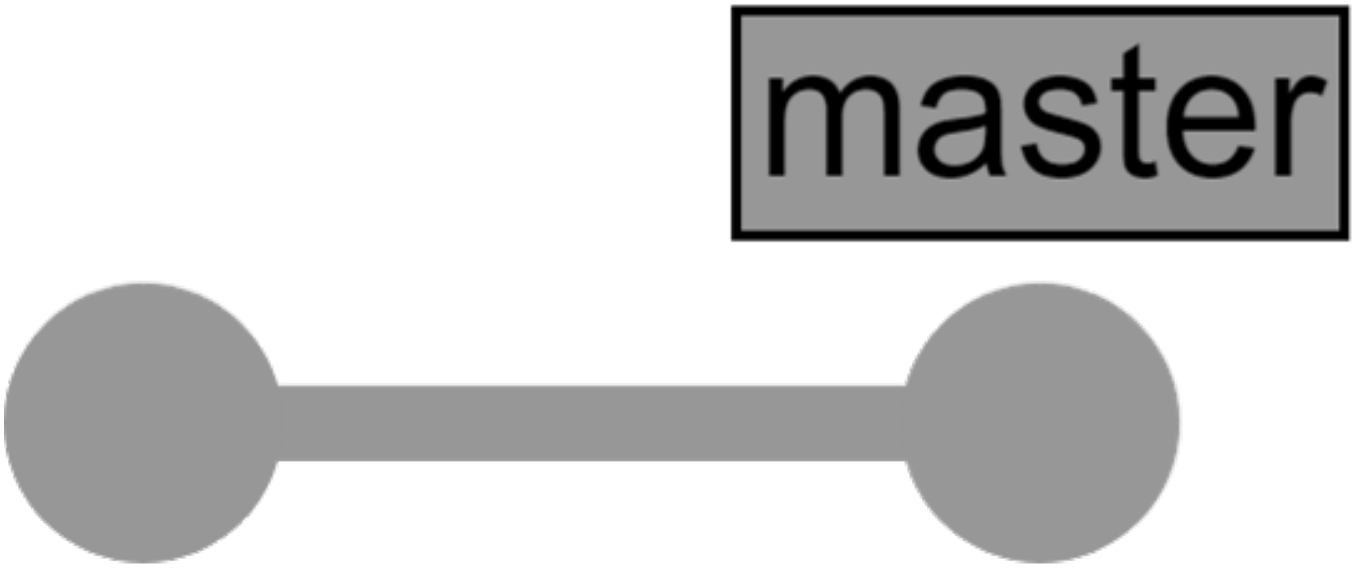
Commit is a fixed state of the repository. Obviously, if each commit stored the full state of the repository, our repository would grow very big very fast. That's why a commit stores a **diff** which is a difference between the current commit and its **parent commit**.

Different commits can have a different number of parents:

- 0 - the first commit in the repository doesn't have parents.
- 1 - business as usual - our commit changed something in the repository as it was during parent commit
- 2 - when we have two different states of the repository we can unite them into one new state. And that state and that commit would have 2 parents.
- >2 - can happen when we unite more that 2 different states of the repository into one new state. It wouldn't be particularly relevant to our discussion, but it does exist.

Now for a parent, each commit that diffs from it is called a **child commit**. Each parent commit can have any number of children commits.

Branch is a reference (or pointer) to a commit. Here's how it looks:

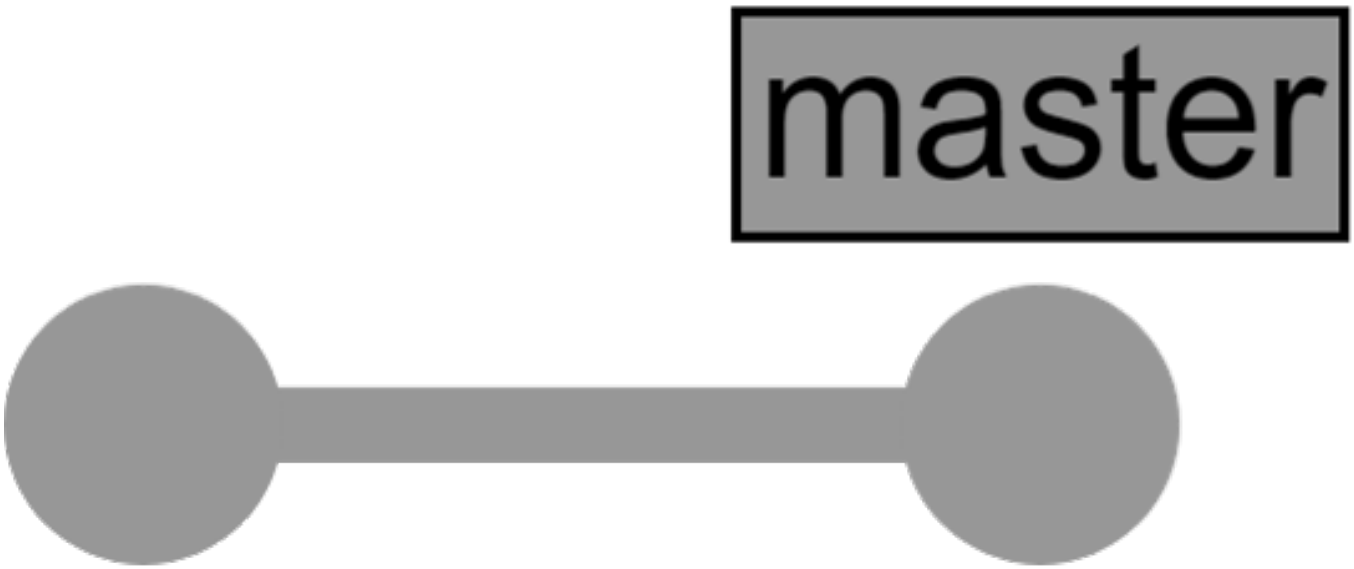


On this image can see the repository with two commits (grey circles), the second one is the head of the master branch. After we add more commits, our repository starts to look like this:

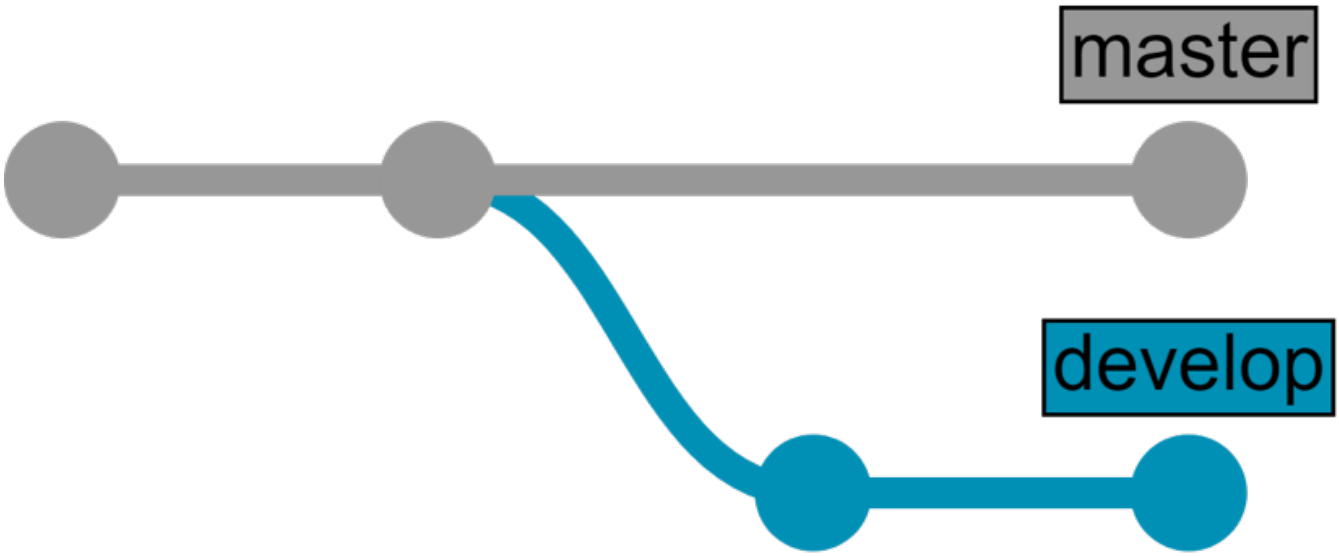
That's the most simple case. One developer works on one change at a time. However, usually, there are many developers working simultaneously on different features and we need a **commit tree** to show what's going on in our repository.

Commit tree

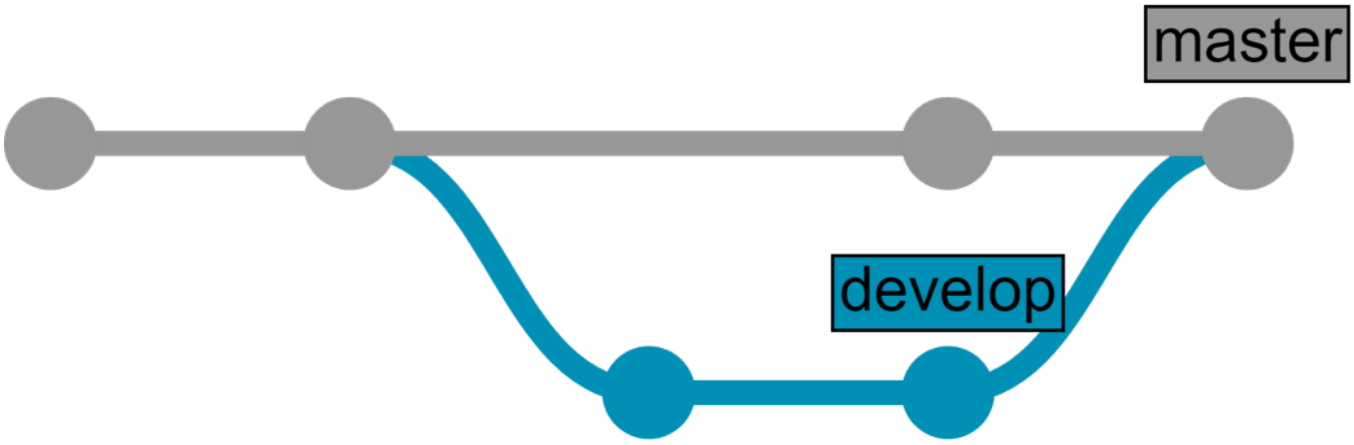
Let's begin from the same starting point. Here's the repository with two commits:



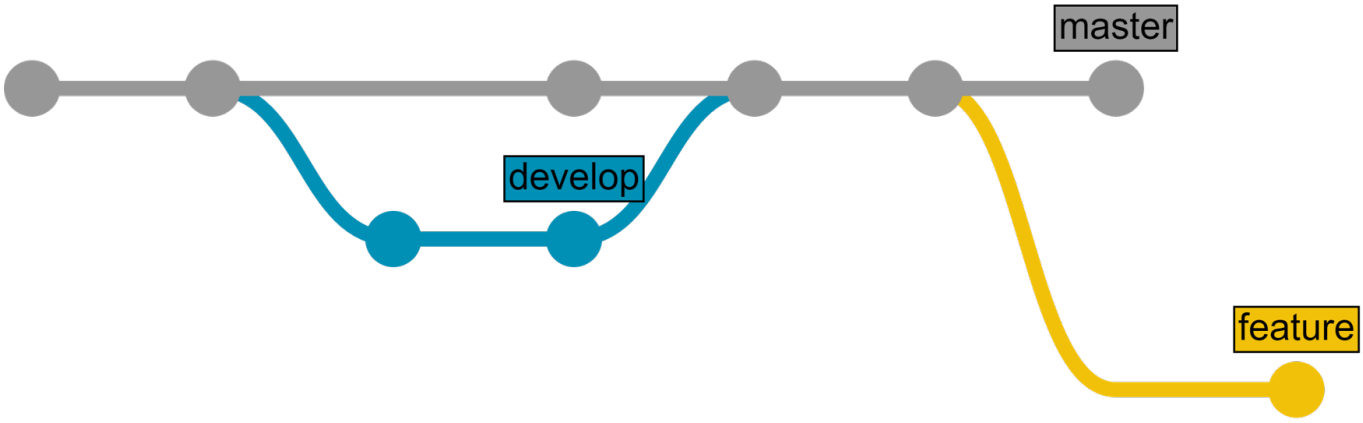
But now, two developers are working at the same time and to not interfere with each other they work in separate branches:



After a while they need to unite changes made and for that they create a **merge request** (also called **pull request**) - which is exactly what it sounds like - its a request to unite two different states of the repository (in our case we want to merge develop branch into master branch) into one new state. After it's appropriately reviewed and approved our repository looks like this:



And the development continues:



Git 101 - Summary

Main concepts:

- **Git** is a non-linear, distributed version control system.
- **Repository** stores data and meta-information about the data.
- **Commit** is a fixed state of the repository.
- **Branch** is a reference to a commit.
- **Merge request** (also called **pull request**) - is a request to unite two different states of the repository into one new state.

If you want to read more about Git, there are books available.

Git flows

Now, that the reader is familiar with basic Git terms and concepts let's talk about how development part of software life-cycle can be managed using Git. There are a number of practices (called flows) which describe development process using Git, but we going to talk about two of them:

- GitHub flow
- GitLab flow

GitHub flow

GitHub flow is as easy as it gets. Here it is:

1. Create a branch from the repository.
2. Commit your changes to your new branch
3. Send a pull request from your branch with your proposed changes to kick off a discussion.
4. Commit more changes on your branch as needed. Your pull request will update automatically.
5. Merge the pull request once the branch is ready to be merged.

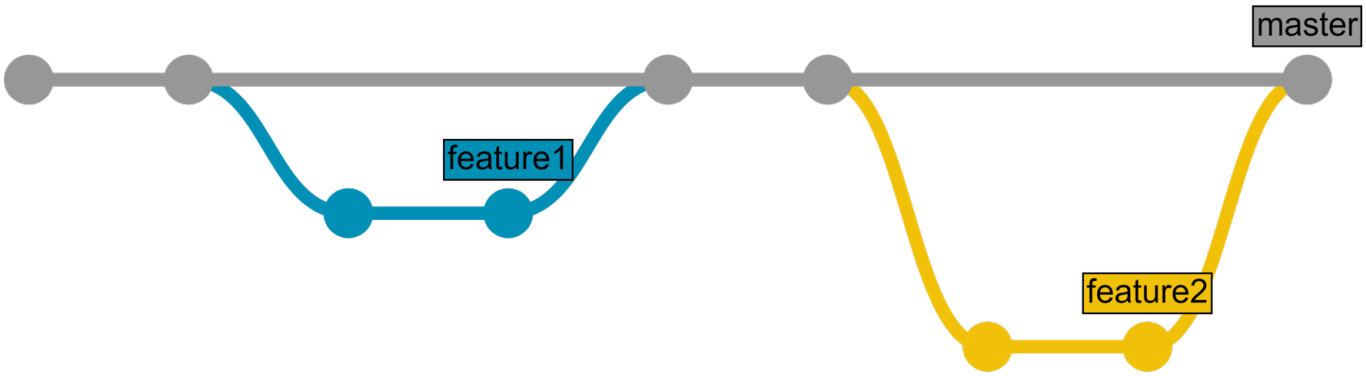
And there are several rules we need to follow:

- master branch is always deployable (and working!)
- There is no development going directly in the master branch
- Development is going on in the feature branches
- master == production* environment**
- You need to deploy to production as often as possible

* Do not confuse with "Ensemble Productions", Here "Production" means LIVE.

** Environment is a configured place where your code runs - could be a server, a VM, container even.

Here's how it looks like:



You can read more about GitHub flow here. There's also an illustrated guide.

GitHub flow is good for small projects and to try it if you're starting out with Git flows. GitHub uses it, though, so it can be viable on big projects too.

GitLab flow

If you're not ready to deploy on production right away, GitLab flow offers GitHub flow + environments. Here's how it works - you develop in feature branches, same as above, merge into master, same as above, but here's a twist: master equals only test environment. In addition to that, you have "Environment branches" which are linked to various other environments you might have.

Usually, three environments exist (you can create more if you need it):

- Test environment == master branch

- PreProduction environment == preprod branch
- Production environment == prod branch

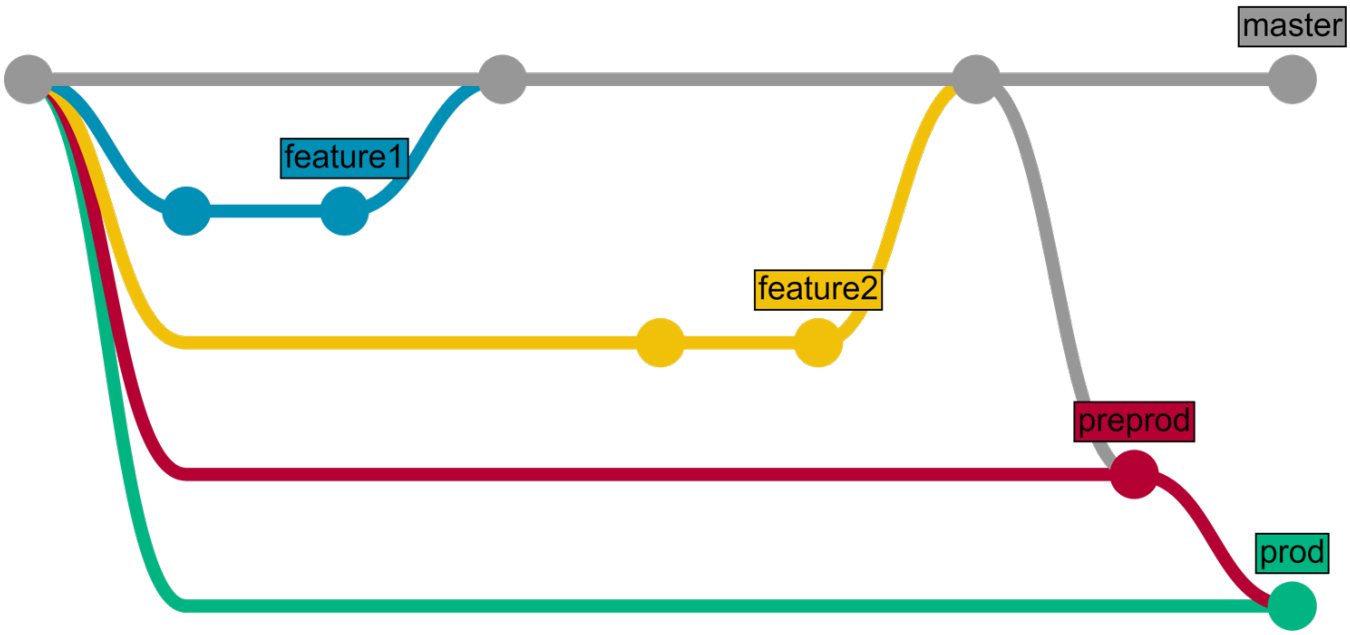
The code that arrives into one of the environment branches should be moved into the corresponding environment immediately, it can be done:

- Automatically (we'll be at it in parts 2 and 3)
- Semi-automatically (same as automatically except a button authorizing the deployment should be pressed)
- Manually

The whole process goes like this:

1. Feature is developed in feature branch.
2. Feature branch is reviewed and merged into 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
5. While we were merging and testing, several new features were developed and merged into master, so GOTO 3.

Here's how it looks like:



You can read more about GitLab flow here.

Conclusion

- Git is a non-linear, distributed version control system.
- Git flow can be used as a guideline for software development cycle, there are several that you can choose from.

Links

- Git book
- GitHub flow
- GitLab flow

- Driessen flow (more comprehensive flow, for comparison)
- Code for this article

Discussion questions

- Do you use git flow? Which one?
- How many environments do you have for an average project?

What's next

In the next part we will:

- Install GitLab.
- Talk about some recommended tweaks.
- Discuss GitLab Workflow (not to be confused with GitLab flow).

Stay tuned.

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