Article
Eduard Lebedyuk · Mar 20, 2018 8m read

# Continuous Delivery of your InterSystems solution using GitLab - Part IV: CD configuration

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 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

I this article we'll finally write a CD configuration.

## Plan

### Environments

First of all, we need several environments and branches that correspond to them:

| 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 |

### Development cycle

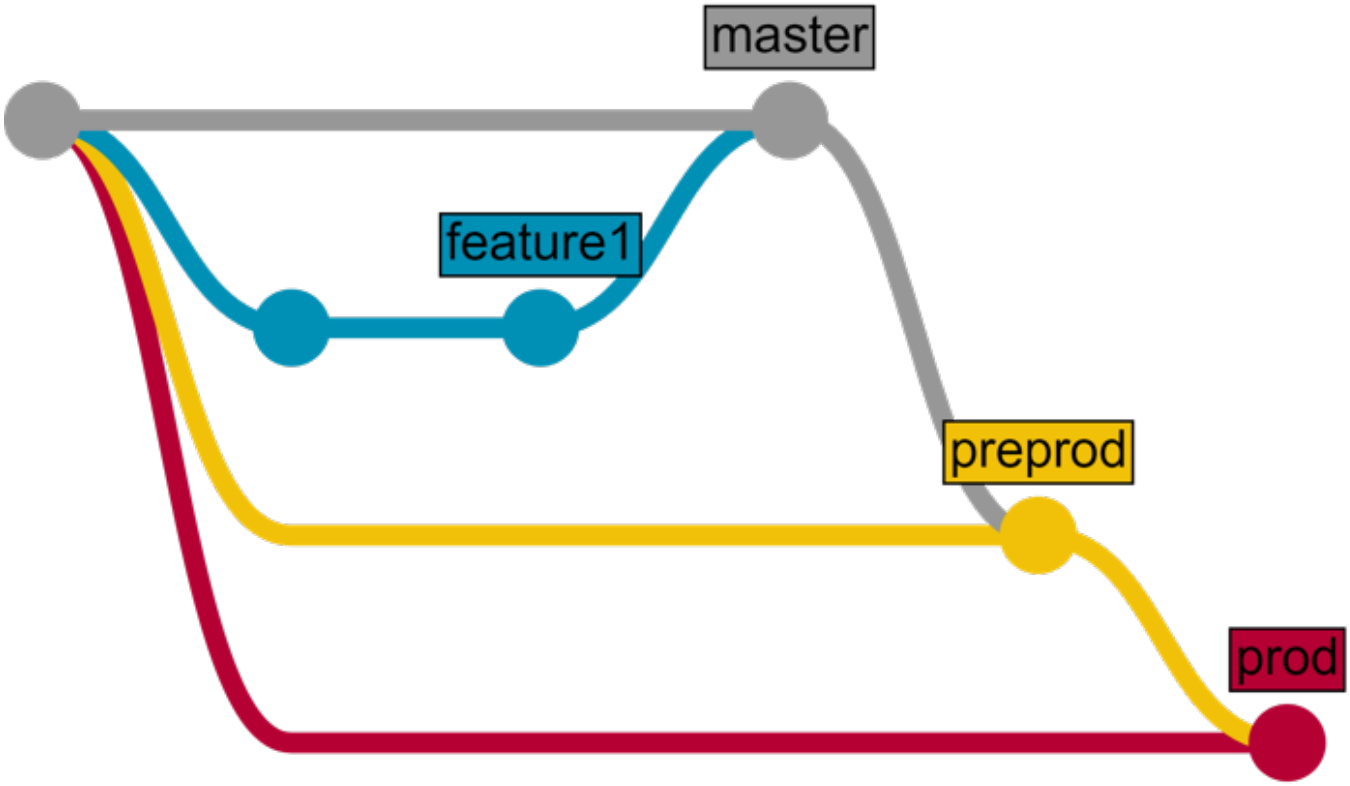
And as an example, we'll develop one new feature using GitLab flow and deliver it using GitLab CD.

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 (I have marked the parts that we need to develop for CD in cursive):

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:




### Application

Our application consists of two parts:

- REST API developed on InterSystems platform
- Client JavaScript web application

## Stages

From the plan above we can determine the stages that we need to define in our Continuous Delivery configuration:

- Load - to import server-side code into InterSystems IRIS
- Test - to test client and server code
- Package - to build client code
- Deploy - to "publish" client code using web server

Here's how it looks like in .gitlab-ci.yml configuration file:

```
stages:
  - load
  - test
  - package
  - deploy
```

## Scripts

### Load

Next, let's define the scripts. Scripts docs. First, lets define a script load server that loads server-side code:

```
load server:
  environment:
    name: test
    url: http://test.hostname.com
  only:
    - master
  tags:
    - test
  stage: load
  script: csession IRIS "##class(isc.git.GitLab).load()"
```

What happens here?

- `load server` is a script name
- next, we describe the environment where this script runs
- `only: master` - tells GitLab that this script should be run only when there's a commit to master branch
- `tags: test` specifies that this script should only run on a runner which has `test` tag
- `stage` specifies stage for a script
- `script` defines code to execute. In our case, we call classmethod `load` from `isc.git.GitLab` class

Now let's write the corresponding `isc.git.GitLab` class. All entry points in this class look like this:

```
ClassMethod method()
{
    try {
        // code
        halt
    } catch ex {
        write !,$System.Status.GetErrorText(ex.AsStatus()),!
        do $system.Process.Terminate(, 1)
    }
}
```

Note that this method can end in two ways:

- by halting current process - that registers in GitLab as a successful completion
- by calling $system.Process.Terminate - which terminates the process abnormally and GitLab registers this as an error

That said, here's our load code:

```
/// Do a full load
/// do ##class(isc.git.GitLab).load()
ClassMethod load()
{
    try {
        set dir = ..getDir()
        do ..log("Importing dir " _ dir)
        do $system.OBJ.ImportDir(dir, ..getExtWildcard(), "c", .errors, 1)
        throw:$get(errors,0)'=0 ##class(%Exception.General).%New("Load error")

        halt
    } catch ex {
        write !,$System.Status.GetErrorText(ex.AsStatus()),!
        do $system.Process.Terminate(, 1)
    }
}
```

Two utility methods are called:

- getExtWildcard - to get a list of relevant file extensions
- getDir - to get repository directory

How can we get the directory?

When GitLab executes a script first it specifies a lot of environment variables. One of them is `CI_PROJECT_DIR` - The full path where the repository is cloned and where the job is run. We can easily get it in our `getDir` method:

```
ClassMethod getDir() [ CodeMode = expression ]
{
##class(%File).NormalizeDirectory($system.Util.GetEnviron("CI_PROJECT_DIR"))
}
```

### Tests

Here's test script:

```
load test:
  environment:
    name: test
    url: http://test.hostname.com
  only:
    - master
  tags:
    - test
  stage: test
  script: csession IRIS "##class(isc.git.GitLab).test()"
  artifacts:
    paths:
```

```
      - tests.html
```

What changed? Name and script code of course, but `artifact` also was added. An artifact is a list of files and directories which are attached to a job after it completes successfully. In our case after the tests are completed, we can generate HTML page redirecting to the test results and make it available from GitLab.

Note that there's a lot of copy-paste from the load stage - environment is the same, script parts, such as environments can be labeled separately and attached to a script. Let's define test environment:

```
.env_test: &env_test
  environment:
    name: test
    url: http://test.hostname.com
  only:
    - master
  tags:
    - test
```

Now our test script looks like this:

```
load test:
  <<: *env_test
  script: csession IRIS "##class(isc.git.GitLab).test()"
  artifacts:
    paths:
      - tests.html
```

Next, let's execute the tests using UnitTest framework.

```
/// do ##class(isc.git.GitLab).test()
ClassMethod test()
{
    try {
        set tests = ##class(isc.git.Settings).getSetting("tests")
        if (tests'="") {
            set dir = ..getDir()
            set ^UnitTestRoot = dir

            $$$TOE(sc, ##class(%UnitTest.Manager).RunTest(tests, "/nodelete"))
            $$$TOE(sc, ..writeTestHTML())
            throw:'..isLastTestOk() ##class(%Exception.General).%New("Tests error")
        }
        halt
    } catch ex {
        do ..logException(ex)
        do $system.Process.Terminate(, 1)
    }
}
```

Tests setting, in this case, is a path relative to repository root where unit tests are stored. If It's empty then we skip tests. `writeTestHTML` method is used to output html with a redirect to test results:

```
ClassMethod writeTestHTML()
```

```
{
    set text = ##class(%Dictionary.XDataDefinition).IDKEYOpen($classname(), "html").D
ata.Read()
    set text = $replace(text, "!!!", ..getURL())

    set file = ##class(%Stream.FileCharacter).%New()
    set name = ..getDir() _  "tests.html"
    do file.LinkToFile(name)
    do file.Write(text)
    quit file.%Save()
}

ClassMethod getURL()
{
    set url = ##class(isc.git.Settings).getSetting("url")
    set url = url _ $system.CSP.GetDefaultApp("%SYS")
    set url = url_"/%25UnitTest.Portal.Indices.cls?Index="_ $g(^UnitTest.Result, 1) _
 "&$NAMESPACE=" _ $zconvert($namespace,"O","URL")
    quit url
}

ClassMethod isLastTestOk() As %Boolean
{
    set in = ##class(%UnitTest.Result.TestInstance).%OpenId(^UnitTest.Result)
    for i=1:1:in.TestSuites.Count() {
        #dim suite As %UnitTest.Result.TestSuite
        set suite = in.TestSuites.GetAt(i)
        return:suite.Status=0 $$$NO
    }
    quit $$$YES
}

XData html
{
<html lang="en-US">
<head>
<meta charset="UTF-8"/>
<meta http-equiv="refresh" content="0; url=!!!"/>
<script type="text/javascript">
window.location.href = "!!!"
</script>
</head>
<body>
If you are not redirected automatically, follow this <a href='!!!'>link to tests</a>.
</body>
</html>
}
```

### Package

Our client is a simple HTML page:

```
<html>
<head>
<script type="text/javascript">
function initializePage() {
  var xhr = new XMLHttpRequest();
```

```
  var url = "${CI_ENVIRONMENT_URL}:57772/MyApp/version";
  xhr.open("GET", url, true);
  xhr.send();
  xhr.onloadend = function (data) {
    document.getElementById("version").innerHTML = "Version: " + this.response;
  };

  var xhr = new XMLHttpRequest();
  var url = "${CI_ENVIRONMENT_URL}:57772/MyApp/author";
  xhr.open("GET", url, true);
  xhr.send();
  xhr.onloadend = function (data) {
    document.getElementById("author").innerHTML = "Author: " + this.response;
  };
}
</script>
</head>
<body  onload="initializePage()">
<div id = "version"></div>
<div id = "author"></div>
</body>
</html>
```

And to build it we need to replace `${CI_ENVIRONMENT_URL}` with its value. Of course, real-world application would probably require npm, but it's just an example. Here's the script:

```
package client:
  <<: *env_test
  stage: package
  script: envsubst < client/index.html > index.html
  artifacts:
    paths:
      - index.html
```

#### Deploy

And finally, we deploy our client by copying index.html into webserver root directory.

```
deploy client:
  <<: *env_test
  stage: deploy
  script: cp -f index.html /var/www/html/index.html
```

That's it!

## Several environments

What to do if you need to execute the same (similar) script in several environments? Script parts can also be labels, so here's a sample configuration that loads code in test and preprod environments:

```
stages:
  - load
  - test
```

```
.env_test: &env_test
  environment:
    name: test
    url: http://test.hostname.com
  only:
    - master
  tags:
    - test

.env_preprod: &env_preprod
  environment:
    name: preprod
    url: http://preprod.hostname.com
  only:
    - preprod
  tags:
    - preprod

.script_load: &script_load
  stage: load
  script: csession IRIS "##class(isc.git.GitLab).loadDiff()"

load test:
  <<: *env_test
  <<: *script_load

load preprod:
  <<: *env_preprod
  <<: *script_load
```

This way we can escape copy-pasting the code.

Complete CD configuration is available here. It follows the original plan of moving code between test, preprod and prod environments.

## Conclusion

Continuous Delivery can be configured to automate any required development workflow.

## Links

- Hooks repository (and sample configuration)
- Test repository
- Scripts docs
- Available environment variables

## What's next

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

#Beginner #Change Management #Continuous Integration #Deployment #Git #System Administration #Caché

**Source URL:** https://community.intersystems.com/post/continuous-delivery-your-intersystems-solution-using-gitlab-part-iv-cd-configuration