Artigo
Danusa Calixto · Dez. 22, 2022 10min de leitura

Entrega contínua de sua solução InterSystems usando GitLab – Parte IV: Configuração da CD

Nesta série de artigos, quero apresentar e discutir várias abordagens possíveis para o desenvolvimento de software com tecnologias da InterSystems e do GitLab. Vou cobrir tópicos como:

- Git básico
- Fluxo Git (processo de desenvolvimento)
- Instalação do GitLab
- Fluxo de trabalho do GitLab
- Entrega contínua
- Instalação e configuração do GitLab
- **CI/CD do GitLab**

No primeiro artigo, abordamos os fundamentos do Git, por que um entendimento de alto nível dos conceitos do Git é importante para o desenvolvimento de software moderno e como o Git pode ser usado para desenvolver software.

No segundo artigo, abordamos o fluxo de trabalho do GitLab: um processo inteiro do ciclo de vida do software e a entrega contínua.

No terceiro artigo, abordamos a instalação e configuração do GitLab e a conexão dos seus ambientes a ele

Neste artigo, finalmente, vamos escrever uma configuração de CD.

Plano

Ambientes

Em primeiro lugar, precisamos de vários ambientes e branches que correspondam a eles:

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

Ciclo de desenvolvimento

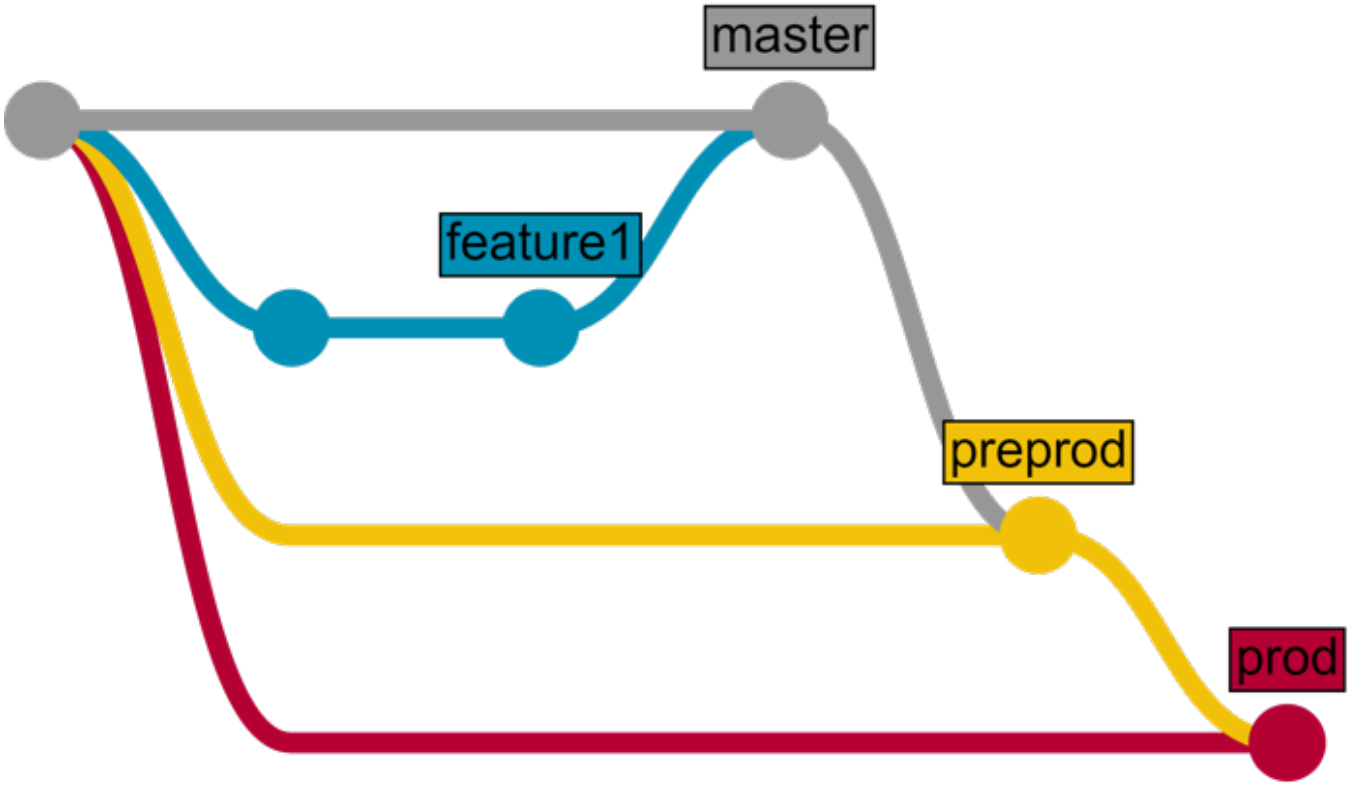
E, como exemplo, desenvolveremos um novo recurso usando o fluxo do GitLab e o entregaremos usando a CD do GitLab.

1. O recurso é desenvolvido em um branch de recursos.
2. O branch de recurso é revisado e mesclado no master branch.
3. Depois de um tempo (vários recursos mesclados), o master é mesclado com o preprod
4. Depois de um tempo (teste do usuário, etc.), o preprod é mesclado com o prod

Veja como isso ficaria (marquei as partes que precisamos desenvolver para o CD em itálico):

1. Desenvolvimento e teste
    - O desenvolvedor envia o código para o novo recurso em um branch de recursos separado
    - Depois que o recurso se torna estável, o desenvolvedor mescla nosso branch de recursos no master branch
    - *O código do branch master é entregue ao ambiente de teste, onde é carregado e testado*
2. Entrega para o ambiente de pré-produção
    - O desenvolvedor cria a solicitação de mesclagem do branch master para o branch de pré-produção
    - Depois de algum tempo, o proprietário do repositório aprova a solicitação de mesclagem
    - *O código do branch de pré-produção é entregue ao ambiente de pré-produção*
3. Entrega para o ambiente de produção
    - O desenvolvedor cria a solicitação de mesclagem do branch de pré-produção para o branch de produção
    - Depois de algum tempo, o proprietário do repositório aprova a solicitação de mesclagem
    - O proprietário do repositório aperta o botão "Implantar"
    - *O código do branch de produção é entregue ao ambiente de produção*

Ou o mesmo, mas em formato de gráfico:



Aplicativo

Nosso aplicativo consiste em duas partes:

- API REST desenvolvida na plataforma InterSystems
- Web application de JavaScript cliente

Estágios

Com o plano acima, podemos determinar as etapas que precisamos definir na nossa configuração de entrega

contínua:

- Carregamento — para importar o código do lado do servidor para o InterSystems IRIS
- Teste — para testar o código do servidor e cliente
- Pacote — para criar o código do cliente
- Implantação — para "publicar" o código do cliente usando o servidor web

Veja como isso fica no arquivo de configuração gitlab-ci.yml:

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

Scripts

Carregamento

Em seguida, vamos definir os scripts. Documentos de scripts. Primeiro, vamos definir um script load server que carrega o código do lado do servidor:

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

O que acontece aqui?

- `load server` é o nome de um script
- em seguida, descrevemos o ambiente em que esse script é executado
- `only: master` — informa ao GitLab que esse script só deve ser executado quando houver um commit para o master branch
- `tags: test` especifica que esse script só deve ser executado em um runner com a tag `test`
- `stage` especifica o estágio para um script
- `script` define o código para executar. No nosso caso, chamamos o classmethod `load` da classe `isc.git.GitLab`

  Observação importante

  Para InterSystems IRIS, troque *csession* por *iris session*.

  Para Windows, use: *irisdb -s ../mgr -U TEST "##class(isc.git.GitLab).load()*

Agora, vamos escrever a classe `isc.git.GitLab` correspondente. Todos os pontos de entrada nessa classe ficam desta forma:

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

Observe que esse método pode terminar de duas maneiras:

- interrompendo o processo atual — que é registrado no GitLab como uma conclusão bem-sucedida
- chamando $system.Process.Terminate — que termina o processo de maneira anormal e o GitLab registra isso como um erro

Dito isso, aqui está nosso código de carregamento:

```
/// 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)
    }
}
```

Dois métodos de utilitários são chamados:

- getExtWildcard — para obter uma lista das extensões de arquivo relevantes
- getDir — para obter o diretório do repositório

Como podemos obter o diretório?

Quando o GitLab executa um script, primeiro, ele especifica várias variáveis de ambiente. Uma delas é a `CI_PROJECT_DIR` — o caminho completo onde o repositório é clonado e onde o job é executado. Ele pode ser obtido facilmente no nosso método `getDir` :

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

Testes

Aqui está o script de teste:

```
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
```

O que mudou? O nome e o código do script, é claro, mas o `artefato` também foi adicionado. Um artefato é uma lista de arquivos e diretórios que são anexados a um job depois que ele é concluído com sucesso. No nosso caso, depois que os testes forem concluídos, podemos gerar a página HTML redirecionando para os resultados dos testes e disponibilizá-la a partir do GitLab.

Observe que há bastante copiar e colar do estágio de carregamento — o ambiente é o mesmo, partes do script, como ambientes, podem ser rotuladas separadamente e anexadas a um script. Vamos definir o ambiente de teste:

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

Agora, nosso script de teste fica assim:

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

Em seguida, vamos executar os testes usando o framework UnitTest.

```
/// 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)
    }
}
```

A definição do teste, nesse caso, é um caminho relativo à raiz do repositório onde os testes de unidade são armazenados. Se estiver vazio, pulamos testes. O método `writeTestHTML` é usado para gerar o html com um redirecionamento para os resultados dos testes:

```
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>
}
```

Pacote

Nosso cliente é uma página HTML simples:

```
<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>
```

E, para criá-la, precisamos substituir `${CI_ENVIRONMENT_URL}` pelo seu valor. Claro, um aplicativo real provavelmente exigiria npm, mas esse é apenas um exemplo. Aqui está o script:

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

Implantação

Por fim, implantamos nosso cliente ao copiar index.html para o diretório raiz do servidor web.

```
deploy client:
  <<: *env_test
  stage: deploy
```

```
  script: cp -f index.html /var/www/html/index.html
```

É isso!

Vários ambientes

O que fazer se você precisar executar o mesmo script (semelhante) em vários ambientes? Partes do script também podem ser rótulos, então aqui está uma configuração de exemplo que carrega o código em ambientes de teste e pré-produção:

```
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
```

Assim, podemos fugir de copiar e colar o código.

Veja a configuração de CD completa aqui. Ela segue o plano original de mover código entre os ambientes de teste, pré-produção e produção.

Conclusão

A entrega contínua pode ser configurada para automatizar qualquer fluxo de trabalho de desenvolvimento necessário.

Links

- Repositório de hooks (e configuração de exemplo)
- Repositório de teste
- Documentos de scripts
- Variáveis de ambiente disponíveis

O que vem a seguir

No próximo artigo, vamos criar a configuração de CD que usa o contêiner Docker do InterSystems IRIS.

#Administração do Sistema #Gestão da Mudança #Git #Implantação #Iniciante #Integração Contínua #Caché

URL de origem:https://pt.community.intersystems.com/post/entrega-cont%C3%ADnua-de-sua-solu%C3%A7%C3%A3o-intersystems-usando-gitlab-%E2%80%93-parte-iv-configura%C3%A7%C3%A3o-da-cd