記事
Mihoko Iijima · 2020年4月28日 13m read

GitLabを使用したInterSystemsソリューションの継続的デリバリー - パートIV：CDの構成

この連載記事
では、InterSystemsの技術とGitLabを使用したソフトウェア開発に向けて実現可能な複数の手法を紹介し、議論したいと思います。 次のようなトピックについて説明します。

- Git 101
- Gitフロー（開発プロセス）
- GitLabのインストール
- GitLabワークフロー
- 継続的デリバリー
- GitLabのインストールと構成
- **GitLab CI/CD**

第1回の記事
では、Gitの基本、Gitの概念を高度に理解することが現代のソフトウェア開発にとって重要である理由、Gitを使用してソフトウェアを開発する方法について説明しました。

第2回の記事
では、アイデアからユーザーフィードバックまでの完全なソフトウェアライフサイクルプロセスであるGitLabワークフローについて説明しました。

第3回の記事では、GitLabのインストールと構成ならびに利用環境のGitLabへの接続について説明しました。

この記事では、最終的にCDの構成を作成します。

計画

環境

まず、いくつかの環境とそれに対応するブランチが必要です。

環境	ブランチ	デリバリー	コミット可能な人	マージ可能な人
Test	master	自動	開発者 所有者	開発者 所有者
Preprod	preprod	自動	該当なし	所有者
Prod	prod	半自動（ボタンを押して配信）	該当なし	所有者

開発サイクル

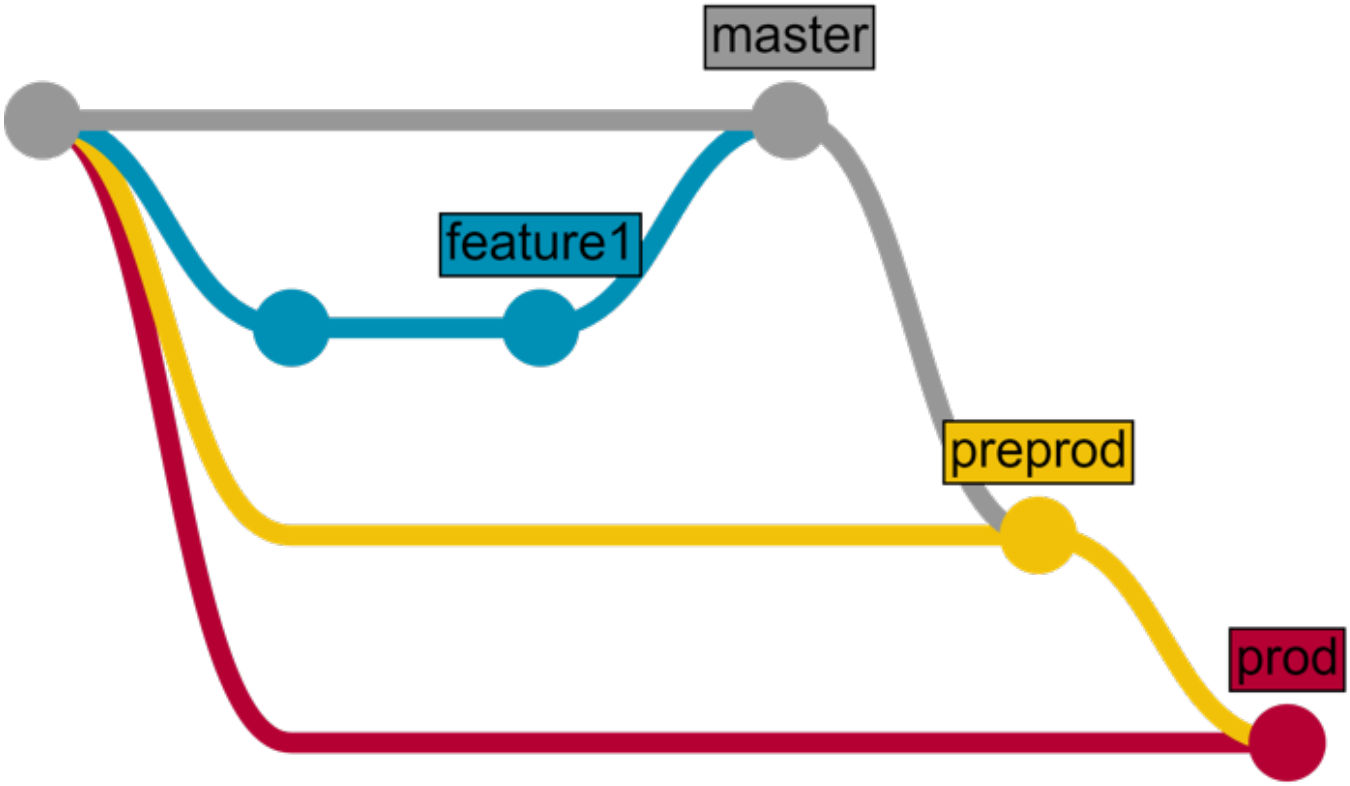
また、GitLabフローを使用して1つの新しい機能を開発し、GitLab CDを使用して配信する例を以下に記載します。

1. 機能はフィーチャーブランチで開発されます。
2. フィーチャーブランチがレビューされ、masterブランチにマージされます。
3. しばらくすると（いくつかのフィーチャーブランチがマージされた）マスターがpreprodにマージされます。
4. しばらくすると（ユーザーテストなどの後に）preprodがprodにマージされます。

その流れは次のようになります（CDのために開発する必要がある部分を筆記体で示しています）。

1. 開発とテスト
    - 開発者は新しい機能のコードを別のフィーチャーブランチにコミットします。
    - 機能が安定した後、開発者はフィーチャーブランチをmasterブランチにマージします。
    - masterブランチのコードはテスト環境に配信され、そこで読み込まれてテストされます。
2. Preprod環境への配信
    - 開発者はmasterブランチからpreprodブランチへのマージリクエストを作成します。
    - その後、リポジトリ所有者がマージリクエストを承認します。
    - preprodブランチのコードがPreprod環境に配信されます。
3. Prod環境への配信
    - 開発者はpreprodブランチからprodブランチへのマージリクエストを作成します。
    - その後、リポジトリ所有者がマージリクエストを承認します。
    - リポジトリ所有者が「デプロイ」ボタンを押します。
    - prodブランチのコードがProd環境に配信されます。

同じ内容を図に表すと次のようになります。



アプリケーション

私たちのアプリケーションは次の2つの部分で構成されています。

- InterSystemsプラットフォームで開発されたREST API
- クライアントJavaScriptウェブアプリケーション

ステージ

上記の計画から、継続的デリバリーの構成で定義する必要があるステージを決定できます。

- 読み込み - サーバーサイドのコードをInterSystems IRISにインポートします
- テスト - クライアントとサーバーのコードをテストします
- パッケージ - クライアントのコードをビルドします
- デプロイ - ウェブサーバーを使用してクライアントのコードを「公開」します

以下に .gitlab-ci.yml 構成ファイルの内容を示します。

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

スクリプト

読み込み

次に、スクリプトを定義しましょう。 スクリプトのドキュメント
はここにあります。 まずは次のような
サーバーサイドのコードを読み込むload serverスクリプトを定義しましょう。

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

ここには何が書かれているのでしょうか？

- load serverはスクリプトの名前です。
- 次に、このスクリプトの実行環境について記述します。
- only: master
  はmasterブランチへのコミットがある場合にのみこのスクリプトを実行するようGitLabに指示しています。
- tags: testはこのスクリプトをtestタグを持つランナーでのみ実行するよう指定しています。
- stage はスクリプトのステージを指定しています。
- script
  は実行するコードを定義しています。 このケー
  スでは、isc.git.GitLabクラスのloadクラスメソッドを呼び出しています。

それでは、対応するisc.git.GitLabクラスを書きましょう。 このクラスのすべてのエントリポイントは次のように

なります。

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

このメソッドは次の2つの方法で終了できます。

- 現在のプロセスを停止する。この場合はGitLabに正常終了したものとして登録されます。
- $system.Process.Terminateを呼び出す。この場合はプロセスが異常終了し、GitLabにエラーとして登録されます。

したがって、読み込み用のコードは次のようになります。

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

ここでは次の2つのユーティリティメソッドが呼び出されています。

- getExtWildcard - 関連するファイル拡張子のリストを取得します
- getDir - リポジトリのディレクトリを取得します

ディレクトリを取得するには？

GitLabは最初にスクリプトを実行するとき、多くの環境変数
を指定します。 その中にはリポジトリが複製され、ジョブが実行されるフルパスである CIPROJECTDIR があります。 この環境変数は、次のように getDir メソッドで簡単に取得できます。

```
ClassMethod getDir() [ CodeMode = expression ]
```

```
{
##class(%File).NormalizeDirectory($system.Util.GetEnviron("CI_PROJECT_DIR"))
}
```

テスト

テストスクリプトは次のとおりです。

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

何が変わったのでしょうか？ 名前とス
クリプトコードはもちろんですが、artifact
も追加されました。 アーティファクトとは、ジョブが正常に完了した後にジョブに添付されるファイルとディレクトリのリストです。 このケースでは、テストが完了した後、テスト結果にリダイレクトするHTMLページを生成し、GitLabから利用できるようにします。

ここでは読み込みステージからコピーして貼り付けられたコードがたくさんあることに注意してください。環境は同じです。環境などのスクリプト部分は個別にラベル付けしてスクリプトに添付できます。 次のようにテスト環境を定義しましょう。

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

テストスクリプトの内容は次のようになります。

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

次に、UnitTestフレームワークを使用してテストを実行しましょう。

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

この場合、テストの設定はユニットテストが保存されているリポジトリルートに対する相対パスになります。 空の場合はテストをスキップします。 writeTestHTML
メソッドは、テスト結果にリダイレクトを含むhtmlを出力するために使用されます。

```
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>
```

パッケージ

クライアントは次のようなシンプルなHTMLページです。

```
<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>
```

また、これをビルドするには ${CI_ENVIRONMENT_URL}
をその値と置き換える必要があります。 当然ながら、実際のアプリケーションではおそらくnpmが必要になるでしょう。しかし、これは単なる見本です。 スクリプトは次のとおりです。

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

デプロイ

最後に、index.htmlをウェブサーバーのルートディレクトリにコピーし、クライアントをデプロイします。

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

以上です！

複数の環境

複数の環境で同じ（同様の）スクリプトを実行する必要がある場合はどうしますか？ スクリプト部分はラベルにすることもできます。そのため、test環境とpreprod環境でコードを読み込むサンプル構成を次に示します。

```
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: irissession IRIS "##class(isc.git.GitLab).loadDiff()"
load test:
  <<: *env_test
  <<: *script_load

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

このようにして、コードのコピー＆ペーストを回避することができます。

完全なCDの構成はこちら
から取得できます。 この構成はtest環境、preprod環境、prod環境間でコードを移動するという当初の計画に従っています。

まとめ

継続的デリバリーは、必要な開発ワークフローを自動化するように構成できます。

リンク

- Hooksリポジトリ（およびサンプル構成）
- テストリポジトリ
- スクリプトのドキュメント
- 利用可能な環境変数

次の内容

次の記事では、InterSystems IRIS Dockerコンテナを活用するCD構成を作成します。

#Git #継続的デリバリー #その他

ソースURL:
https://jp.community.intersystems.com/post/gitlab%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9Fintersystems%E3%82%BD%E3%83%AA%E3%83%A5%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3%E3%81%AE%E7%B6%99%E7%B6%9A%E7%9A%84%E3%83%87%E3%83%AA%E3%83%90%E3%83%AA%E3%83%BC-%E3%83%91%E3%83%BC%E3%83%88iv%EF%BC%9Acd%E3%81%AE%E6%A7%8B%E6%88%90