記事
Mihoko Iijima · 2020年4月27日 11m 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 II：GitLabワークフロー

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

前の記事
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。 なお、前の記事ではソフトウェア開発の実装部分を中心に取り扱っていましたが、このパートでは以下を取り扱います。

- GitLabワークフロー - 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。
- 継続的デリバリー - チームが短いサイクルでソフトウェアを作成し、ソフトウェアをいつでも確実にリリースできるようにするソフトウェアエンジニアリング手法です。 この手法はソフトウェアの構築、テスト、リリースをより速く、より頻繁に行うことを目指しています。

GitLabワークフロー

GitLabワークフロー
は、ソフトウェア開発プロセスのライフサイクル全体で実行可能な操作を論理的に並べたものです。

GitLabワークフローは、前の記事で説明したGitLabフローを考慮に入れています。 以下にそのイメージを掲載します。

1. アイデア：新しい提案はすべて、アイデアから始まります。
2. 課題：アイデアを議論するには、そのための課題を作成するのが最も効果的です。 チームとコラボレーターが課題トラッカーでアイデアの洗練と改善を支援します。
3. 計画：議論が合意に達したら、コーディングに取りかかります。 ただし、まずは課題をマイルストーンと課題ボードに割り当て、ワークフローに優先順位を付けて整理する必要があります。
4. コード：すべてを整理したら、コードを作成できるようになります。
5. コミット：ドラフトに満足したら、バージョン管理機能を使ってフィーチャーブランチにコードをコミットできるようになります。 GitLabフローについては、前の記事で詳しく説明しています。
6. テスト：GitLab CIを使用してスクリプトを実行し、アプリケーションをビルドおよびテストします。
7. レビュー：スクリプトが機能し、テストとビルドが成功すると、コードをレビューして承認する準備が整います。
8. ステージング：ここではコードをステージング環境にデプロイし、すべてが期待どおりに機能したかどうか、または調整が必要かどうかを確認します。

9. プロダクション：すべてが正常に機能するようになったら、プロダクション環境にデプロイします！
10. フィードバック：ここでは作業を振り返り、作業のどのステージで改善が必要かを確認します。



繰り返しになりますが、このプロセス自体は新しいものではなく（その点ではGitLabに固有のものではありません）、その他お好みのツールでも実現できます。

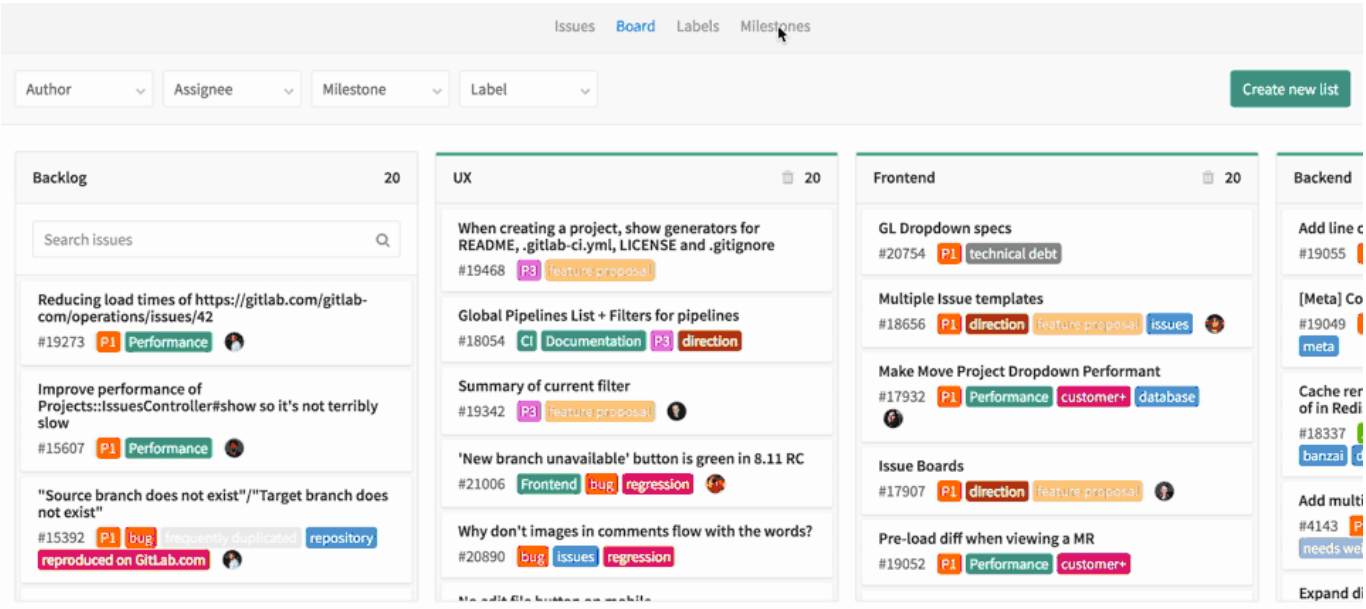
これらのステージのいくつかと、その内容について説明しましょう。 ドキュメントも入手できます。

課題と計画
GitLabワークフローの最初のステージでは、**課題**
（機能やバグ、あるいは意味的に区別されたその他の各種作業）に重点が置かれます。

課題には、次のようないくつかの目的があります。

- 実行管理：課題には期日、担当者、経過時間、見積などがあり、 問題解決の追跡に役立てることができます。
- 目標管理：課題は、バージョンが進化するにつれてソフトウェアを追跡できるマイルストーンやかんばんボードを構成しています。
- 開発：課題には関連する議論とコミットがあります。

計画ステージでは、課題を優先度、マイルストーン、かんばんボードごとにグループ化し、その概要を把握できます。



前のパートでは開発について説明しましたが、単にあなたが希望するgitフローに従ってください。 新しい機能を開発し、それをマスターにマージした後はどうなるでしょうか？

継続的デリバリー

継続的デリバリー - チームが短いサイクルでソフトウェアを作成し、ソフトウェアをいつでも確実にリリースできるようにするソフトウェアエンジニアリング手法です。 この手法はソフトウェアの構築、テスト、リリースをより速く、より頻繁に行うことを目指しています。 この手法では、実稼働中のアプリケーションに対してより多くの増分更新を行うことができ、それによって変更を配信するコスト、時間、およびリスクを削減することができます。 継続的デリバリーには、簡単で繰り返し可能な配信プロセスが重要です。

GitLabでの継続的デリバリー

GitLabでは、継続的デリバリーの構成はリポジトリごとにYAML設定ファイルとして定義されます。

- 継続的デリバリーの設定は複数の連続するステージから構成されます。
- 各ステージには並列に実行される1つ以上のスクリプトがあります。

スクリプトは1つのアクションと、それを実行するために満たすべき条件を定義します。

- 何を実行するか（OSコマンドの実行、コンテナの実行）？
- スクリプトを実行するタイミング：
- トリガーは何か（特定ブランチへのコミット）？
- 前のステージが失敗した場合にアクションを実行するかどうか？
- 手動または自動のどちらで実行するのか？
- どの環境でスクリプトを実行するのか？
- スクリプトを実行した後にどのアーティファクトを保存するのか（これらはアクセスしやすいように環境からGitLabにアップロードされます）？

環境はスクリプトを実行できる構成済みのサーバーまたはコンテナです。

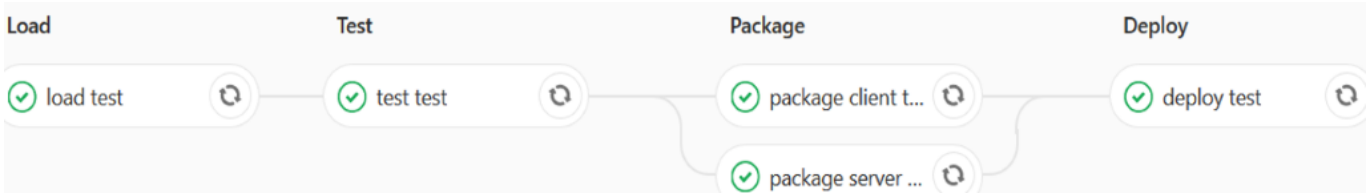
ランナー
は特定の環境でスクリプトを実行します。 これらはGitLabに接続され、必要に応じてスクリプトを実行します。

ランナーは、サーバー、コンテナ、またはローカルマシンにデプロイできます。

継続的デリバリーはどのように行われますか？

1. 新しいコミットがリポジトリにプッシュされます。
2. GitLabが継続的デリバリーの構成をチェックします。
3. 継続的デリバリーの構成にはあらゆるケースに対応したすべてのスクリプトが含まれているため、この特定のコミットに対して実行する必要があるスクリプトセットに絞り込まれます（例えばmasterブランチへのコミットの場合、masterブランチに関連するアクションのみをトリガーします）。 このセットはパイプラインと呼ばれます。
4. パイプラインはターゲット環境で実行され、実行結果はGitLabに保存および表示されます。

例えば、masterブランチへのコミット後に実行される1つのパイプラインは次のとおりです。



これは連続して実行される4つのステージで構成されています。

1. 読み込みステージではコードがサーバーに読み込まれます。
2. テストステージではユニットテストが実行されます。
3. パッケージステージでは、次の2つのスクリプトが並列実行されます。
4. クライアントのビルド
5. サーバーコードのエクスポート（主に情報提供のため）
6. デプロイステージでは、ビルドされたクライアントがウェブサーバーのディレクトリに移動されます。

ご覧のとおり、各スクリプトは順番に実行されます。デフォルトではいずれかのスクリプトが失敗した場合、後続のスクリプトは実行されません（ただし、この動作は変更できます）。



スクリプトを開くと、ログを確認して失敗の原因を特定できます。

```
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
```

コンパイルエラーが原因でスクリプトが失敗していました。

まとめ

- GitLabはソフトウェア開発のすべての主なステージをサポートしています。
- 継続的デリバリーはソフトウェアのビルド、テスト、デプロイのタスクを自動化するのに役立ちます。

リンク

- パート I：Git
- GitLabワークフローの概要
- GitLab CI/CDドキュメント
- GitLabフロー
- この記事のコード

次の内容

次の記事での実施内容：

- GitLabをインストールします。
- それをInterSystems製品がインストールされているいくつかの環境と連携させます。
- 継続的デリバリーを構成します。

継続的デリバリーの仕組みを説明しましょう。

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。 コードはこのブランチに投入され、ターゲット環境に配信されます。

環境	ブランチ	配信	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

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。

これは次のようになります。

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

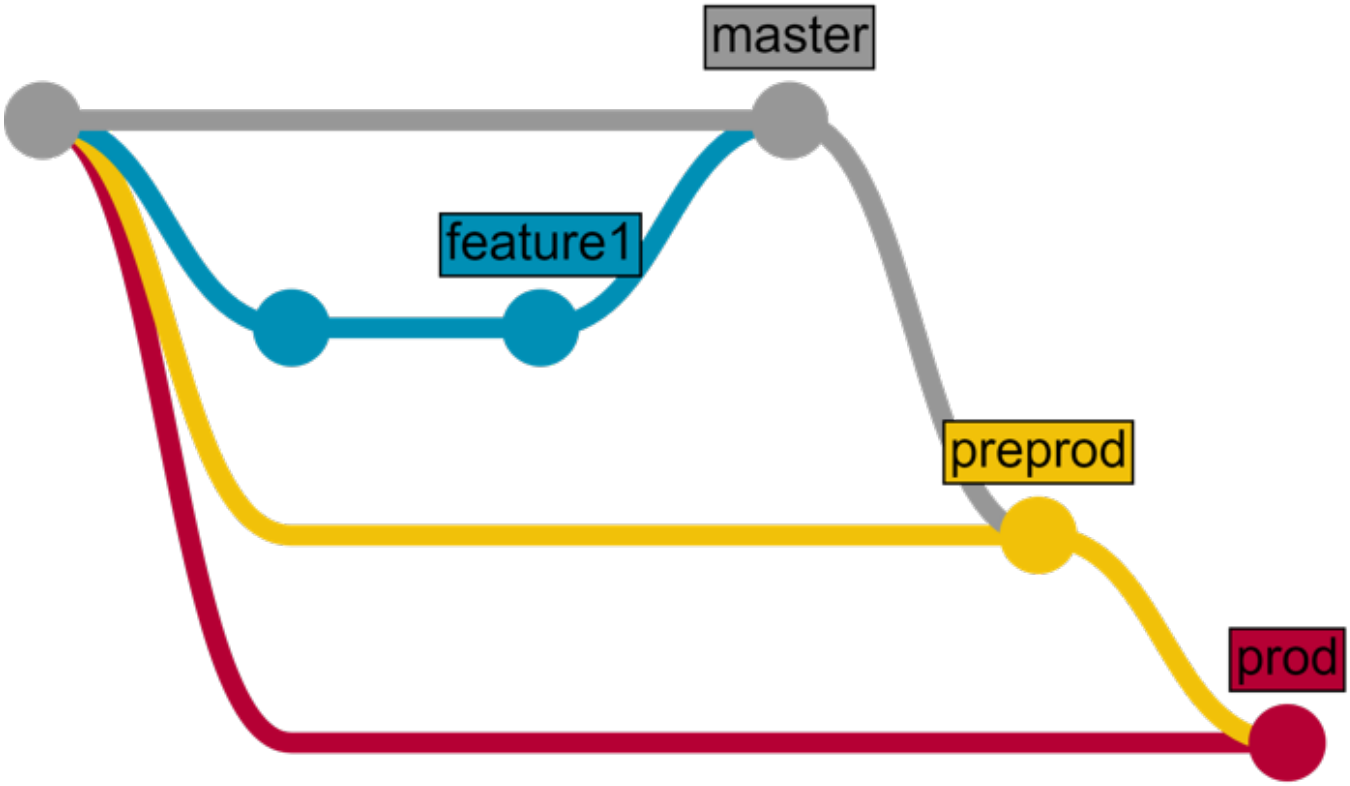
- 開発者はmasterブランチからpreprodブランチへのマージリクエストを作成します。
- その後、リポジトリ所有者がマージリクエストを承認します。
- preprodブランチのコードがPreprod環境に配信されます。

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「Deploy」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。



#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%88-ii%EF%BC%9Agitlab%E3%83%AF%E3%83%BC%E3%82%AF%E3%83%95%E3%83%AD%E3%83%BC