SI現場のテスト自動化への挑戦
～フルコンテナ構成のCI/CD環境～

2019年5月18日

JJUG CCC 2019 Spring

#ccc_m4

川沼 大輝

日本アイ・ビー・エム株式会社

横山 夏実

日本アイ・ビー・エム
システムズ・エンジニアリング株式会社

自己紹介

川沼 大輝

日本アイ・ビー・エム株式会社

クラウドデリバリー部門に所属し、インフラからアプリまで幅広く担当

Java/C#/IBM Cloud/AWS

横山 夏実

日本アイ・ビー・エム

システムズ・エンジニアリング株式会社

開発プロセスの改善、開発環境の構築運用に関わり10数年

Docker/Kubernetes

免責事項

本発表は私自身の見解であり、
所属組織の立場、戦略、意見を代表するものではありません

アジェンダ

- CI/CDを適用したプロジェクトについて
- Javaのテスト実装について
- フルコンテナ構成のCI/CD環境について
- まとめ

CI/CDを適用したプロジェクトについて

CI/CDを適用したプロジェクトについて

プロジェクト概要

- 既存システムを統合し、新たなサービス基盤を構築するプロジェクト
- APIにはMSA(Microservice Architecture)を採用
  - ドメイン単位でCI/CDができる開発環境が必要
- 開発手法はWF型
  - 想定される仕様変更に耐えられる品質ゲートを設けたい

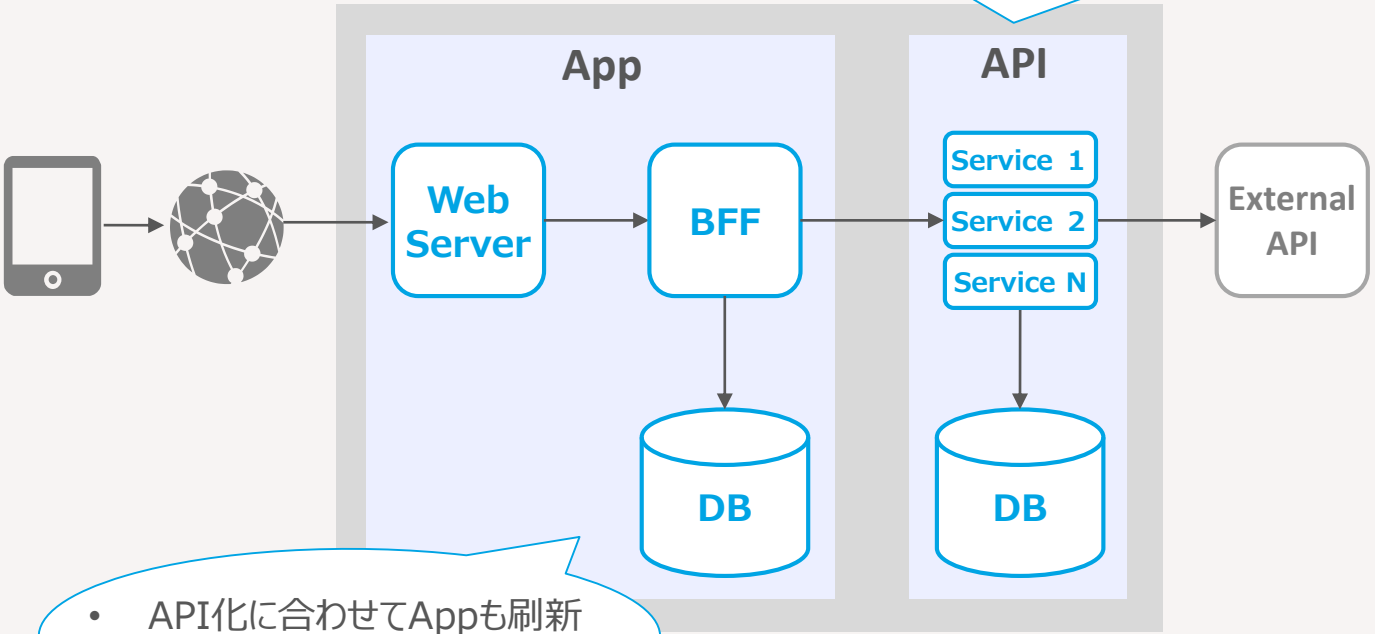
お客様

- CI/CDの必要性を認識されていて、取り組みに積極的
- 他のプロジェクトに横展開できるリファレンスケースを作りたい

CI/CDを適用したプロジェクトについて

システムアーキテクチャ

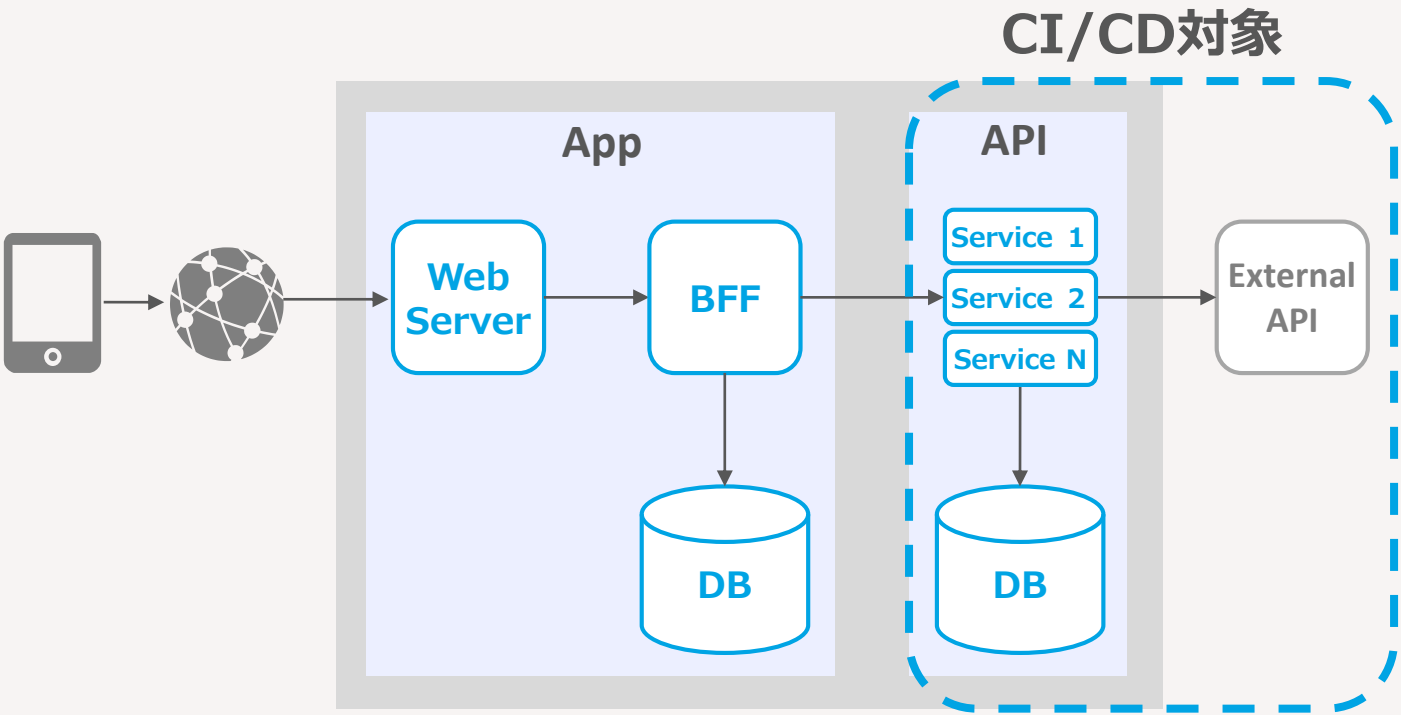
- 既存システムを統合/API化
- APIアーキテクチャにはMSAを採用



- API化に合わせてAppも刷新
- Appアーキテクチャはモノリス

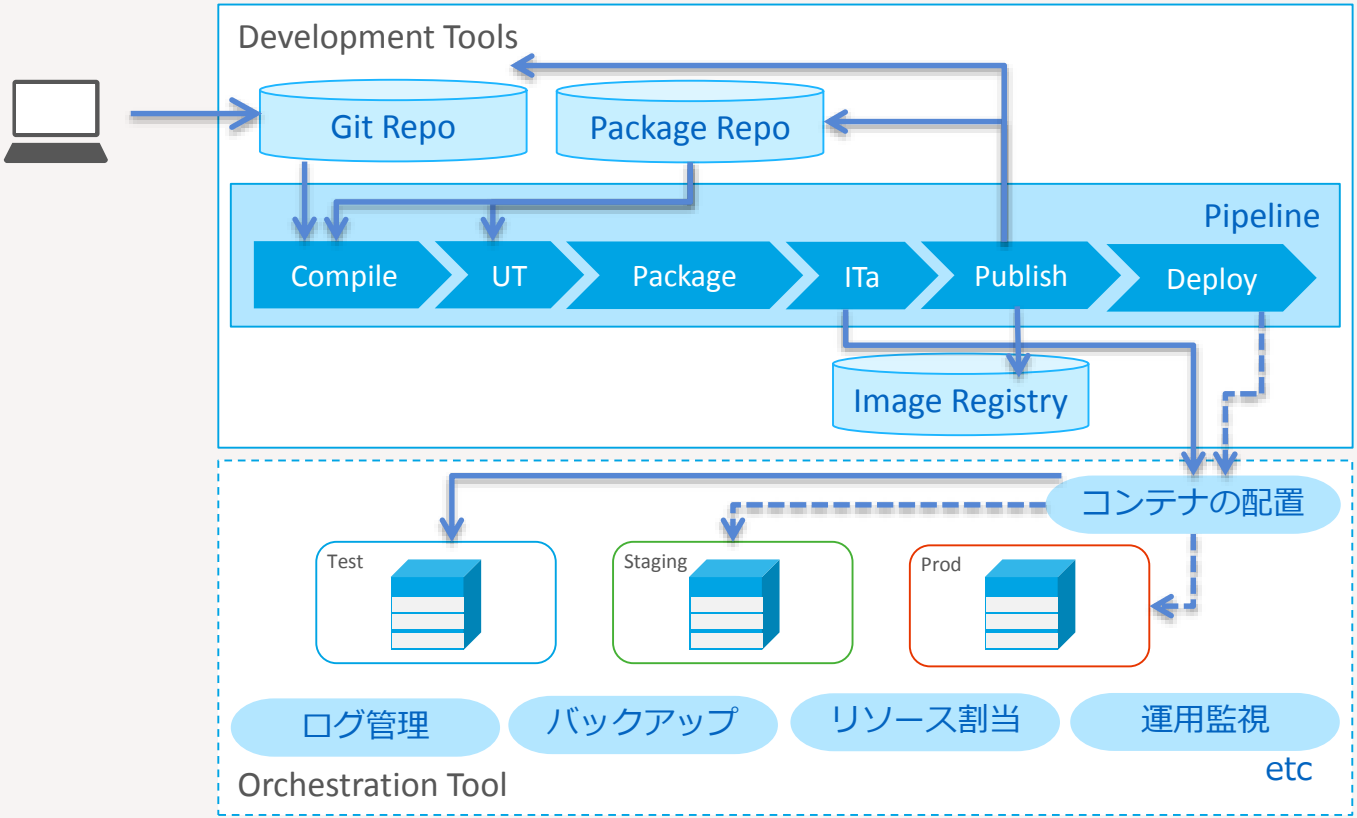
CI/CDを適用したプロジェクトについて

システムアーキテクチャ



CI/CDを適用したプロジェクトについて

以下のようなCI/CD環境を構想



CI/CDを適用したプロジェクトについて

テスト自動化を考える上での二つの観点

- テスト実装そのもの
  - 業務や業務データに強く依存し、どんなテストをすれば良いのかを一般化することは難しい
  - テスト実装コストと品質の兼ね合いが一つのポイント

➡ 今回のプロジェクトの観点を短めに説明する（当然唯一解ではない）

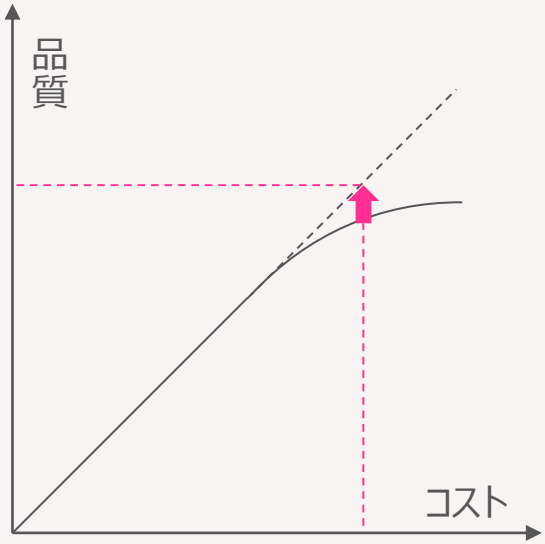
- テストを動かす環境
  - テスト実装に関わらず、ある程度一般化することが可能
  - ポータビリティーとスケーラビリティーがポイント

➡ 幅広く適用できる構成を詳しく説明する

Javaの
テスト実装について

Javaのテスト実装について

3つのこだわりポイント



- 実装コストと品質が比例しなくなる
  損益分岐点のようなものが存在する
- 担保しなくてはいけない品質は満たしつつ、
  妥当な実装コストを見出したい

1. テスト実施観点の住み分け
2. 実装方法ごとにテストのレベル感を変える
3. テストデータは全て構造化されたフォーマットを使用する

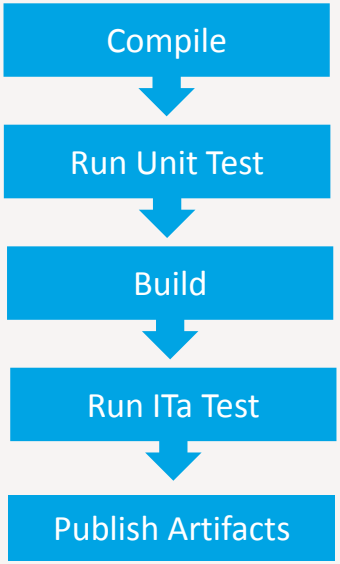
Javaのテスト実装について

1. テスト実施観点の住み分け

		対象	観点	内容	CIでの実施
Test	静的解析	• 全コード	静的コード解析	• Checkstyle • SpotBugs	**必須**
	UT	• Controller • Service • Repository	ホワイトボックス	• 各レイヤーごとの単体テスト • ホワイトボックスの観点 • C1 100%を目指す	**必須**
	ITa	• Functional	ブラックボックス	• 一気通貫の機能テスト • ブラックボックスの観点 • パラメータ網羅を目指す	**部分的**

Javaのテスト実装について

1. テスト実施観点の住み分け

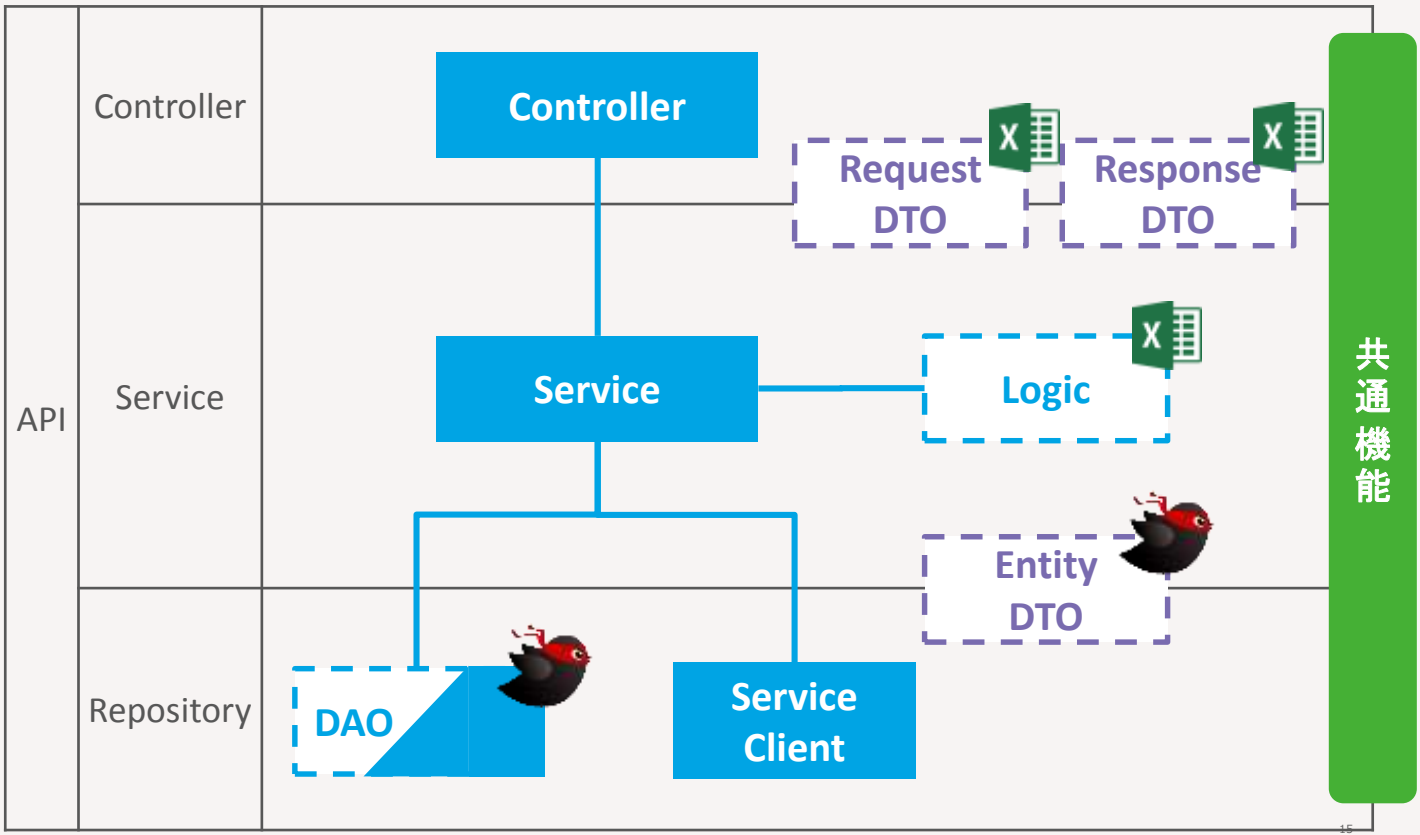


- Checkstyle
- SpotBugs
- JUnit
- Jacoco

- **ITaの正常系のみ実施**
  - スクリプトのメンテナンスが追いつかなくなり自動化が回らなくなることも考慮
  - シンプルな正常系のみをCI上では自動実行

Javaのテスト実装について

2. 実装方法ごとにテストのレベル感を変える



Javaのテスト実装について

2. 実装方法ごとにテストのレベル感を変える

自動生成クラス	Logic DAO	• 正常系1ケースのみ実装する • カバレッジ網羅が目的 • (UTが書いてある安心感)
カスタム実装クラス	Controller Service Service Client	• 正常系と異常系を網羅的に実装する • C1 100%を目標にする • C2 は基本的に考慮しない

Javaのテスト実装について

3. テストデータには全て構造化されたフォーマットを使用

API	Controller	• Request → JSON • Response → JSON • Query param → TXT	構造化されたテストデータ ○ Reviewerが データを把握しやすい ○ テストコードと テストデータを切り離せる ○ 変更箇所が明確になる
	Service	• Request → JSON • DTO → JSON • Response → JSON	
	Repository	• DTO → JSON • DB Data → CSV	

Javaのテスト実装について

テストコード（Controller）

```
@Test
public void ID001_selectPets_正常系パラメータ指定あり() throws Exception {

   List<PetResponse> actualResponse = TestDataLoader.loadJsonAsObject(
      RESOURCE_PATH + "/case001/response.json", new TypeReference<List<PetResponse>>() {});

   when(petService.selectAll(any(PetRequestQuery.class), anyList(), anyList(),
      any(RowBounds.class))).thenReturn(actualResponse);

   String query = TestDataLoader.loadJsonAsString(RESOURCE_PATH + "/case001/query.txt");
   String actualJson = TestDataLoader.loadJsonAsString(RESOURCE_PATH + "/case001/response.json");

   this.mockMvc.perform(get(
      "/api/v1/customers/pets?" + query))
      .andExpect(status().isOk())
      .andExpect(content().contentType(MediaType.APPLICATION_JSON_UTF8))
      .andExpect(content().json(actualJson));

   verify(petService, times(1)).selectAll(queryCaptor.capture(), fieldsCaptor.capture(),
      sortsCaptor.capture(), rowBoundsCaptor.capture());
   assertEquals(Arrays.asList(1, 2), queryCaptor.getValue().getSeq());
   assertEquals(Arrays.asList("ぽち", "たま"), queryCaptor.getValue().getName());
   assertEquals(Arrays.asList(1, 2), queryCaptor.getValue().getPetTypeId());
}
```

Javaのテスト実装について

テストコード（Repository）

```
@Test
public void ID005_insertSelective_正常系() throws Exception {

    databaseSetup();

    PetDto dto = TestDataLoader.loadJsonAsObject(RESOURCE_PATH + "/case005/dto.json", PetDto.class);

    int insertedCount = petDao.insertSelective(dto);
    assertEquals(1, insertedCount);

    IDataSet expectedDataset = new CsvDataSet(new File(FILE_PATH + "/case005"));
    ITable expectedTable = expectedDataset.getTable("pet");

    IDataSet databaseDataSet = databaseConnection.createDataSet();
    ITable actualTable = databaseDataSet.getTable("pet");

    Assertion.assertEquals(expectedTable, actualTable);

    databseClear();
}
```

フルコンテナ構成の
CI/CD環境について

フルコンテナ構成のCI/CD環境について

【再掲】

プロジェクト概要

- 既存システムを統合し、新たなサービス基盤を構築するプロジェクト
- APIにはMSA(Microservice Architecture)を採用
  - ドメイン単位でCI/CDができる開発環境が必要
- 開発手法はWF型
  - **想定される仕様変更に耐えられる品質ゲートを設けたい**

お客様

- CI/CDの必要性を認識されていて、取り組みに積極的
- **他のプロジェクトに横展開できるリファレンスケースを作りたい**

フルコンテナ構成のCI/CD環境について

CI/CD環境を構築する上で実現したいこと

- 実現したいこと
  1. 開発中のバグを早期に発見し、品質を担保する品質ゲートの構築

  2. 横展開できるリファレンスケース
     - 他のプロジェクトにも適用可能なポータビリティー
     - 大規模のプロジェクトにも適用可能なスケーラビリティー

フルコンテナ構成のCI/CD環境について

1. 開発中のバグを早期発見し、品質を担保する品質ゲートの構築

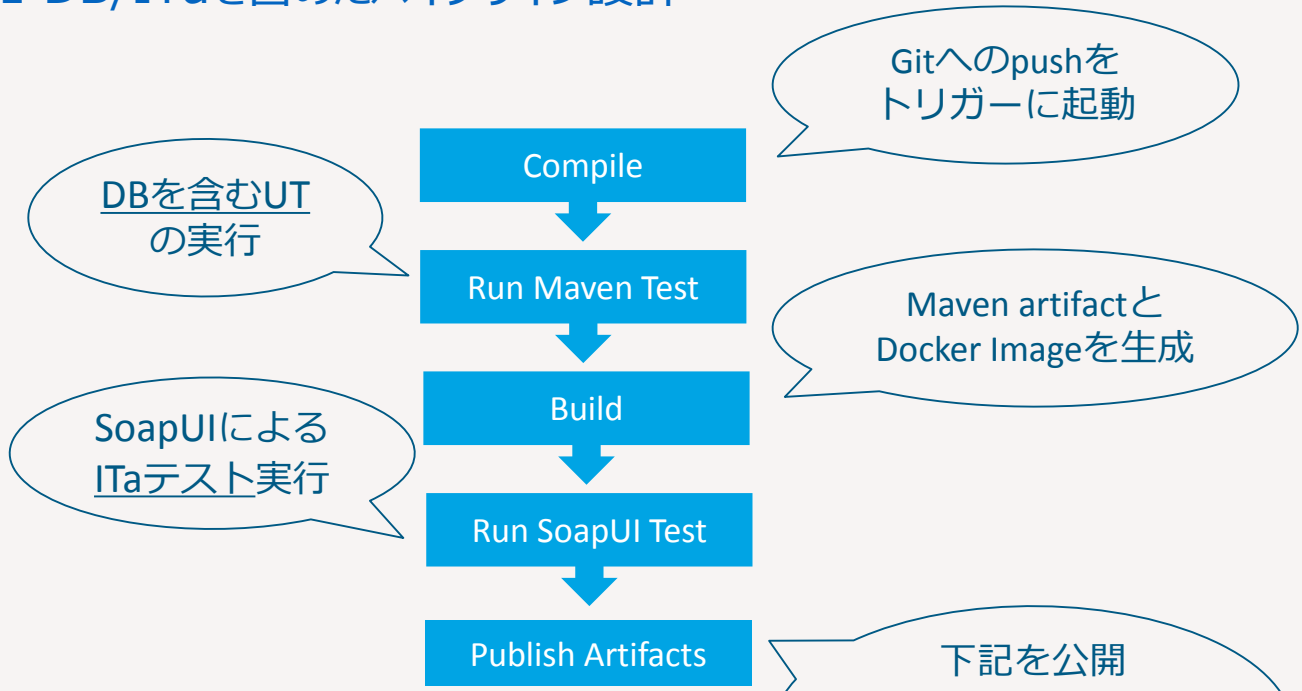
目標:
正常なコードおよび実行可能なモジュールが常に用意できていること

1.1 実行可能であることが担保できるパイプラインを設定する
➡ DBを含むテスト、ITa相当のテストもCIの対象とする

1.2 バグが混在するコードはmasterにマージさせない
➡ featureブランチもCIの対象にする

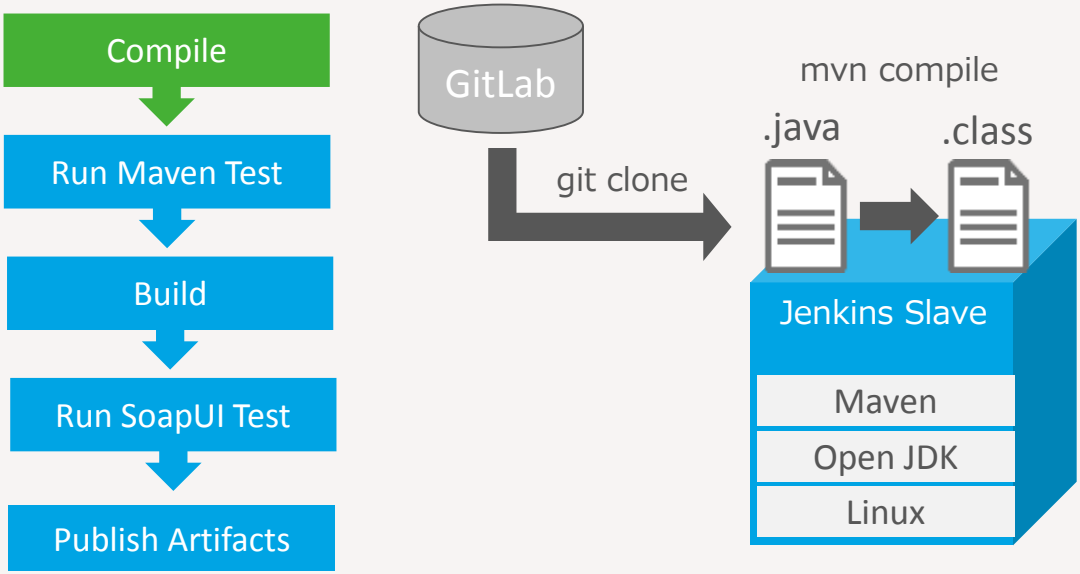
フルコンテナ構成のCI/CD環境について

1.1 DB/ITaを含めたパイプライン設計



フルコンテナ構成のCI/CD環境について

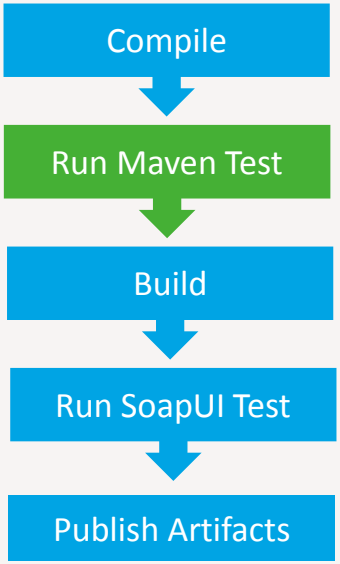
1.1 DB/ITaを含めたパイプライン設計

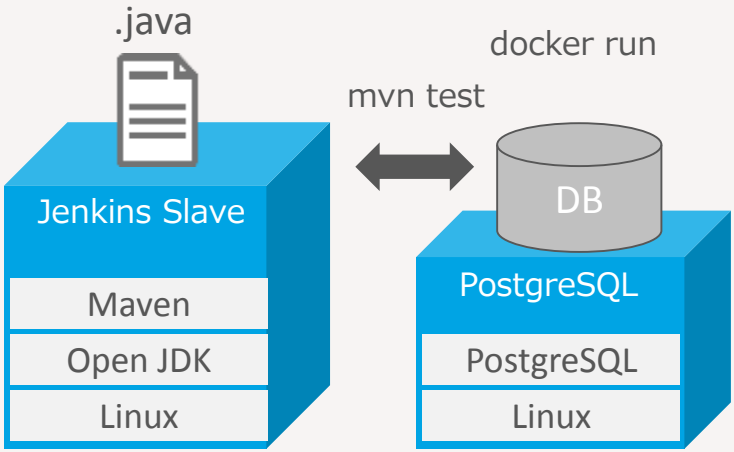


Compileが通ることを確認

フルコンテナ構成のCI/CD環境について

1.1 DB/ITaを含めたパイプライン設計

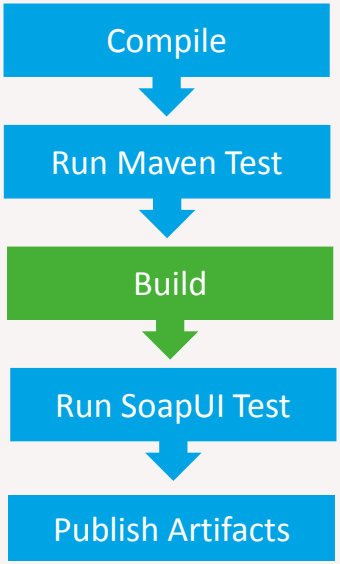




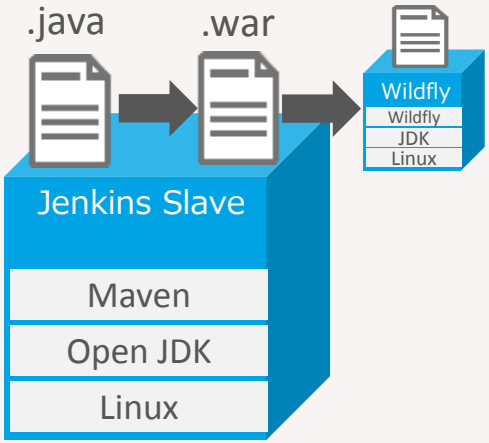
mvn test が通ることの確認

フルコンテナ構成のCI/CD環境について

1.1 DB/ITaを含めたパイプライン設計



mvn clean package –DskipTests=true

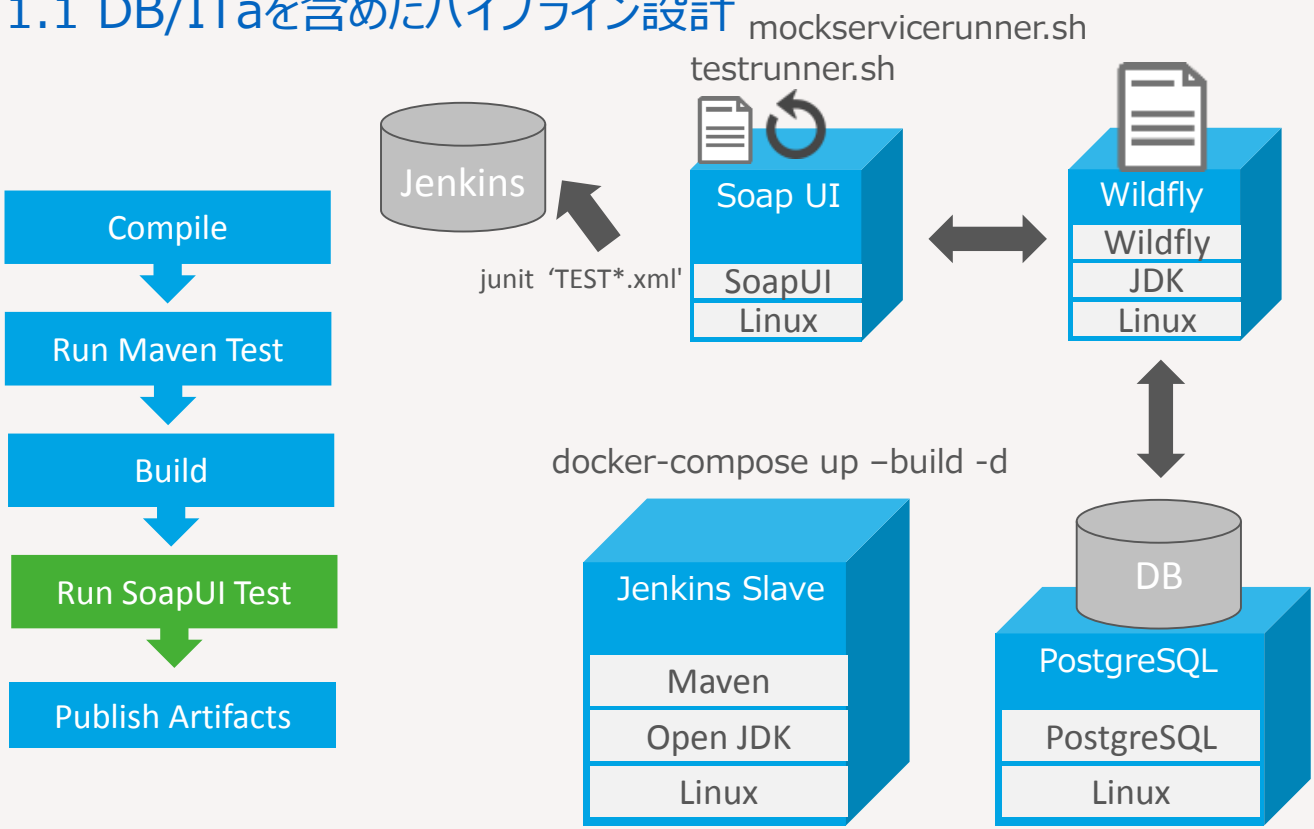


docker build

.war, Dockerイメージ を生成

フルコンテナ構成のCI/CD環境について

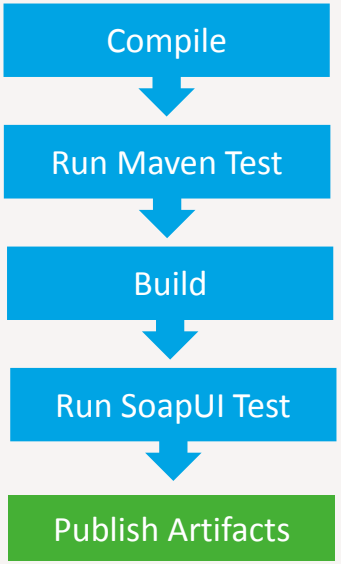
1.1 DB/ITaを含めたパイプライン設計

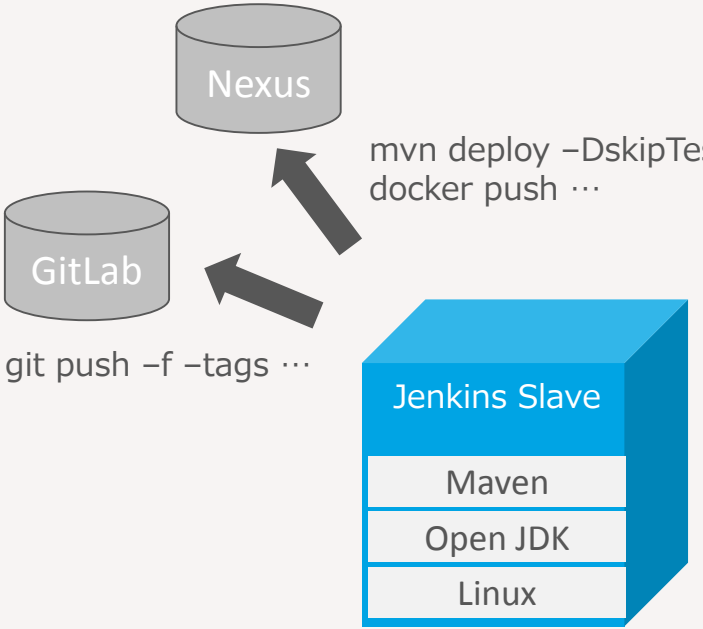


ITaテストが通ることを確認

フルコンテナ構成のCI/CD環境について

1.1 DB/ITaを含めたパイプライン設計

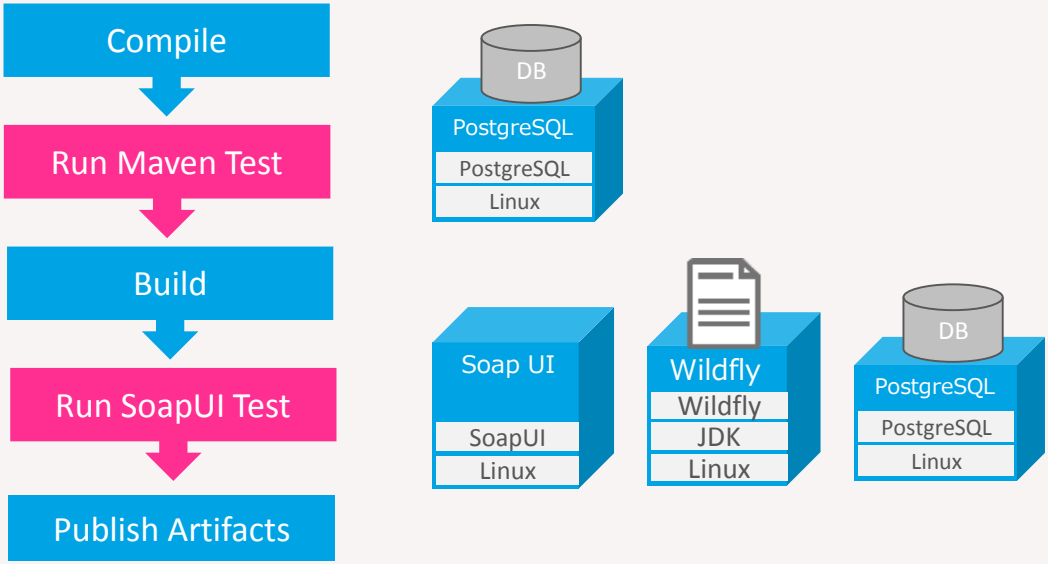




ソースコードのタグをGitLabに格納
テスト済みのモジュールとDockerイメージをNexusに格納

フルコンテナ構成のCI/CD環境について

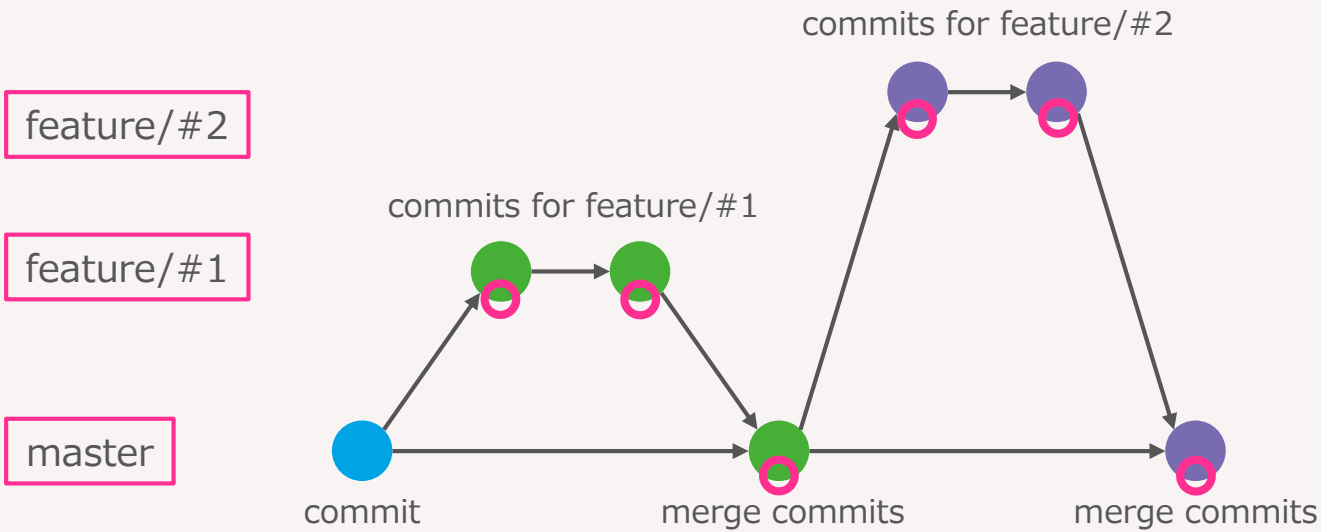
1.1 DB/ITaを含めたパイプライン設計



DBテスト、ITa相当のテストをCIに含むことで
品質ゲートのクオリティが向上した

フルコンテナ構成のCI/CD環境について

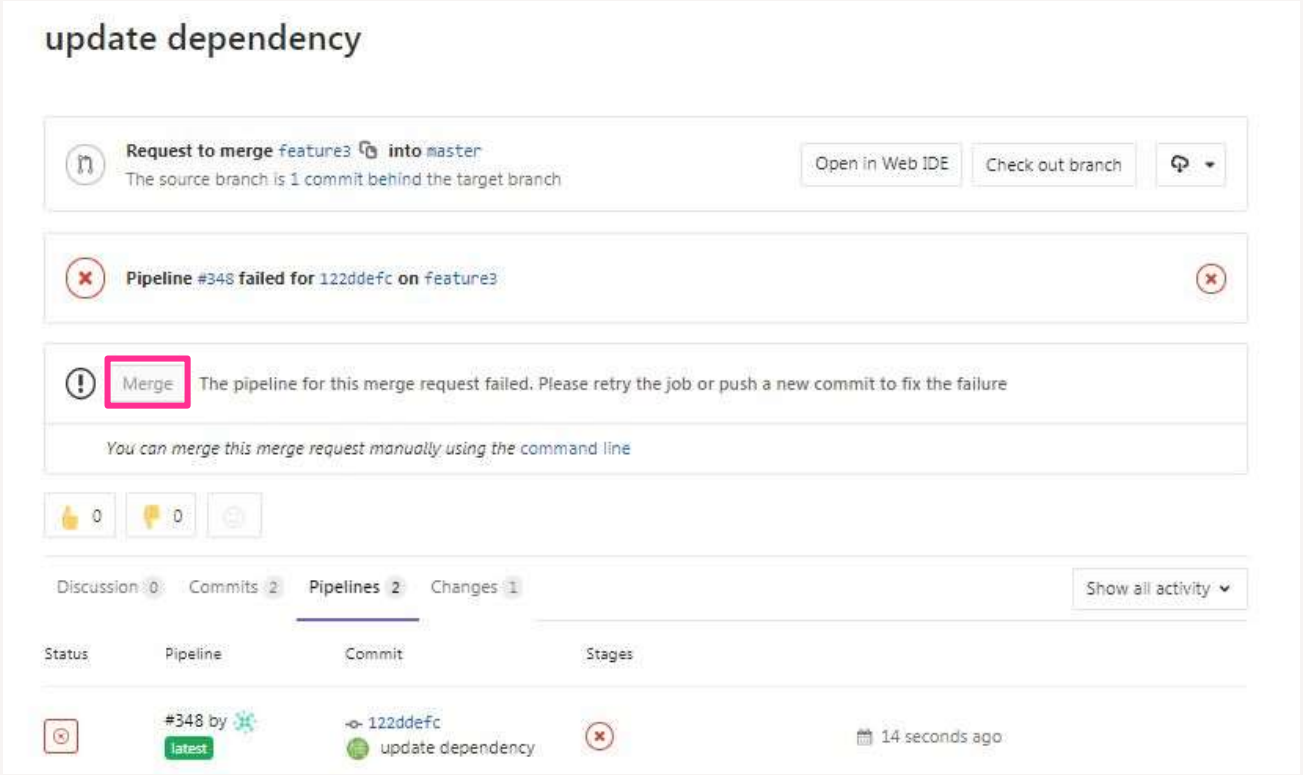
1.2 パイプライン実行の対象



常にテスト済みのコードがMerge Requestされる

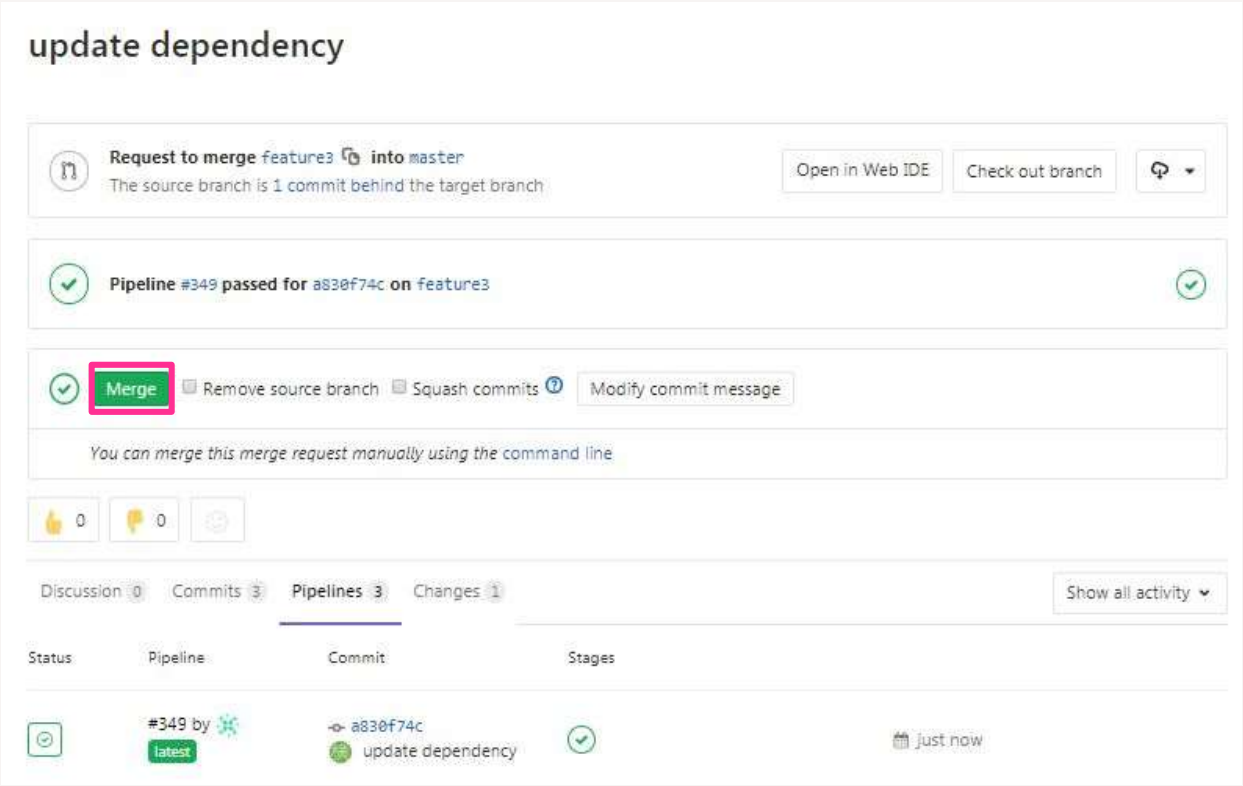
フルコンテナ構成のCI/CD環境について

CIに失敗したMerge Requestはマージできない
➡ masterにはCIが通ったものだけマージが可能



フルコンテナ構成のCI/CD環境について

CIに失敗したMerge Requestはマージできない
➡ masterにはCIが通ったものだけマージが可能



フルコンテナ構成のCI/CD環境について

1. 開発中のバグを早期発見し、品質を担保する品質ゲートの構築

目標:
正常なコードおよび実行可能なモジュールが常に用意できていること

1.1 実行可能[illegible]ラインを設定する
➡ DBを含むテスト、[illegible]対象とする

1.2 バグが混[illegible]terにマージさせない
➡ featureブラ[illegible]

テストの品質が担保された

テストのプロセスが担保された

フルコンテナ構成のCI/CD環境について

2. 横展開できるリファレンスケース

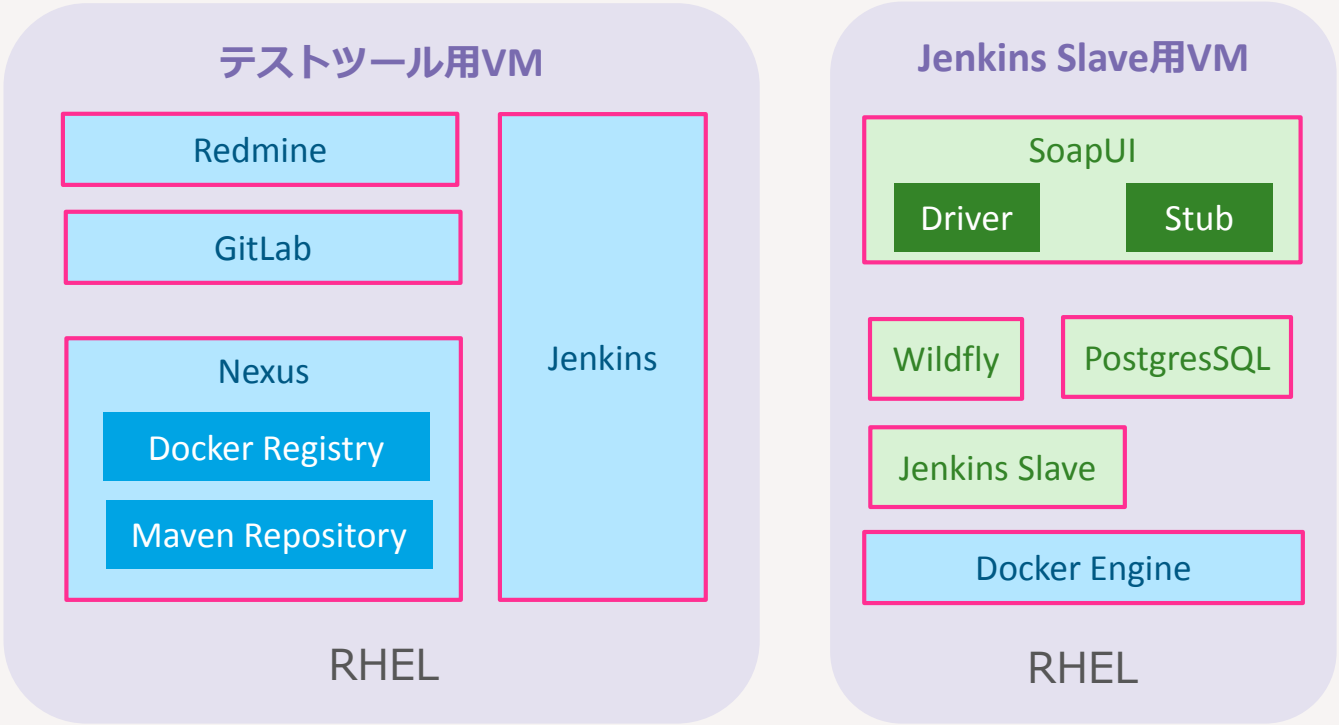
目標:
プロジェクト規模、CI/CD実行環境（オンプレ/クラウド）に関わらず適用可能なリファレンスケースを作る

2.1 出来るだけ実行環境・テストツールにロックインされないこと
➡ OSSとコンテナを用いてポータビリティーを確保

2.2 プロジェクト規模に応じてスケール可能なテスト実行方法
➡ テストにコンテナを用いてスケーラビリティーを確保

フルコンテナ構成のCI/CD環境について

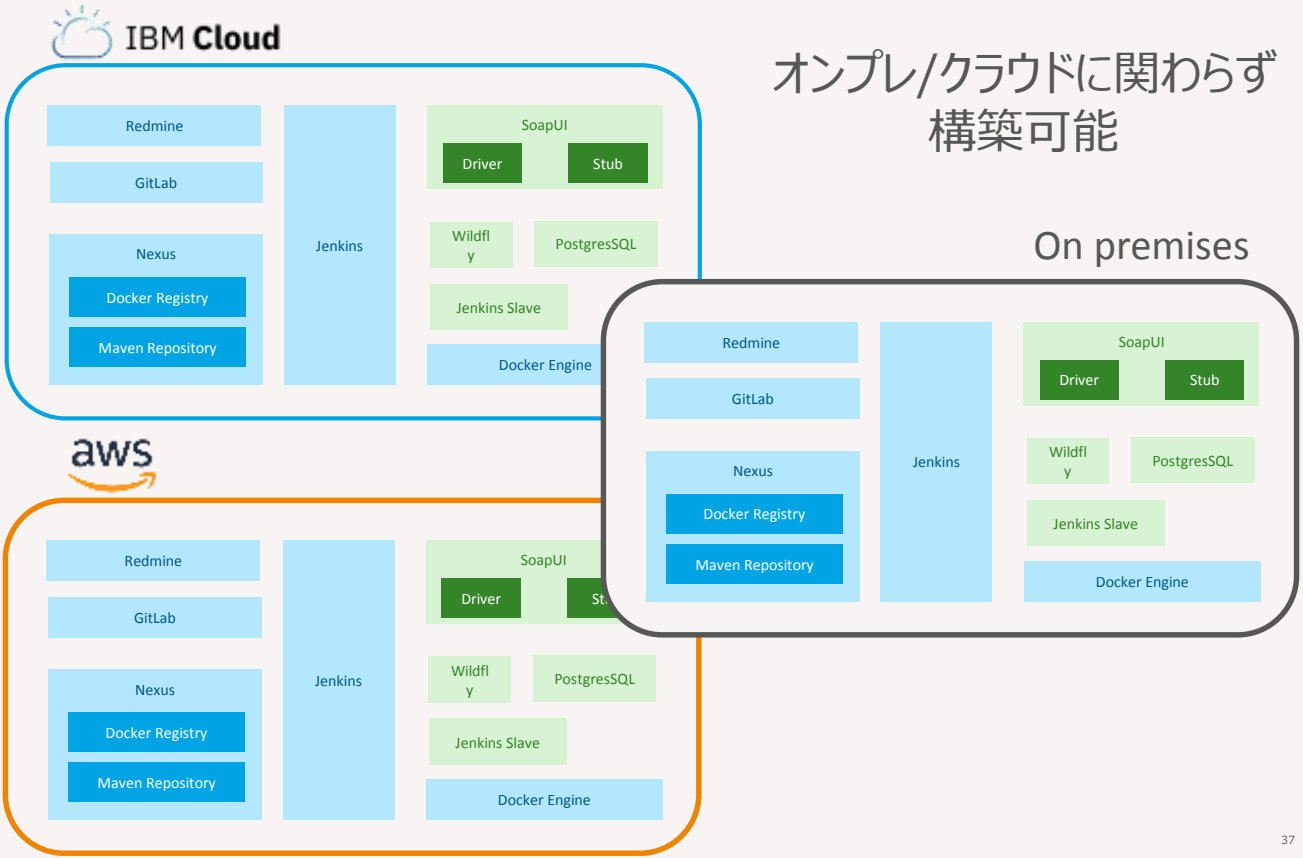
2.1 出来るだけ実行環境・テストツールにロックインされないこと



全てOSSで構成されている
➡ 実行環境に制約はない

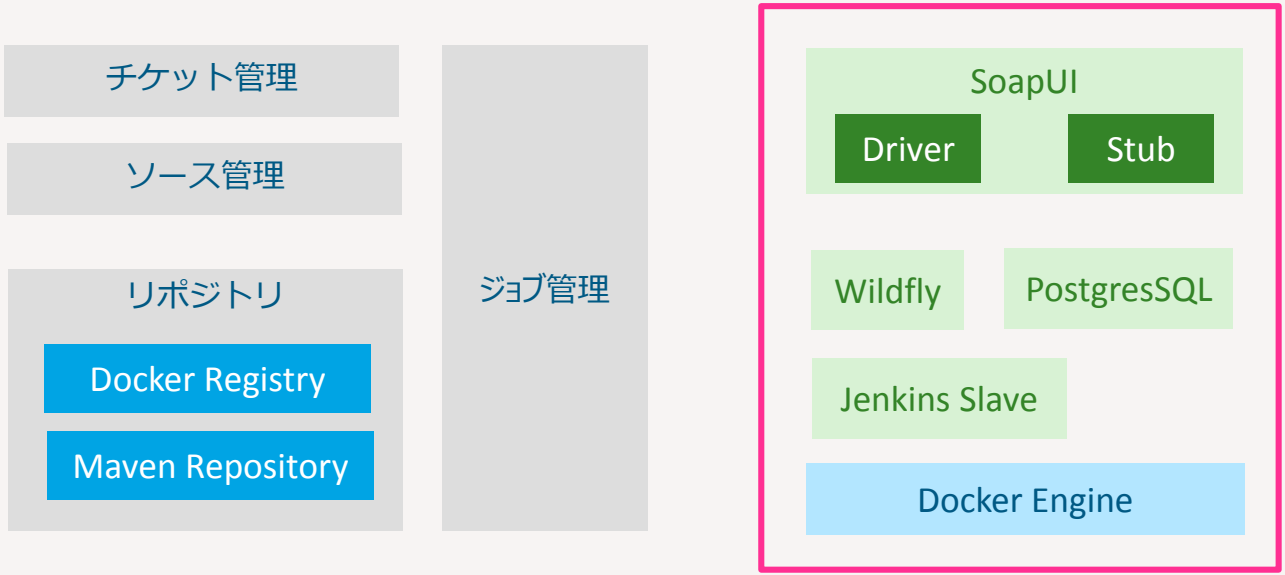
フルコンテナ構成のCI/CD環境について

2.1 出来るだけ実行環境・テストツールにロックインされないこと



フルコンテナ構成のCI/CD環境について

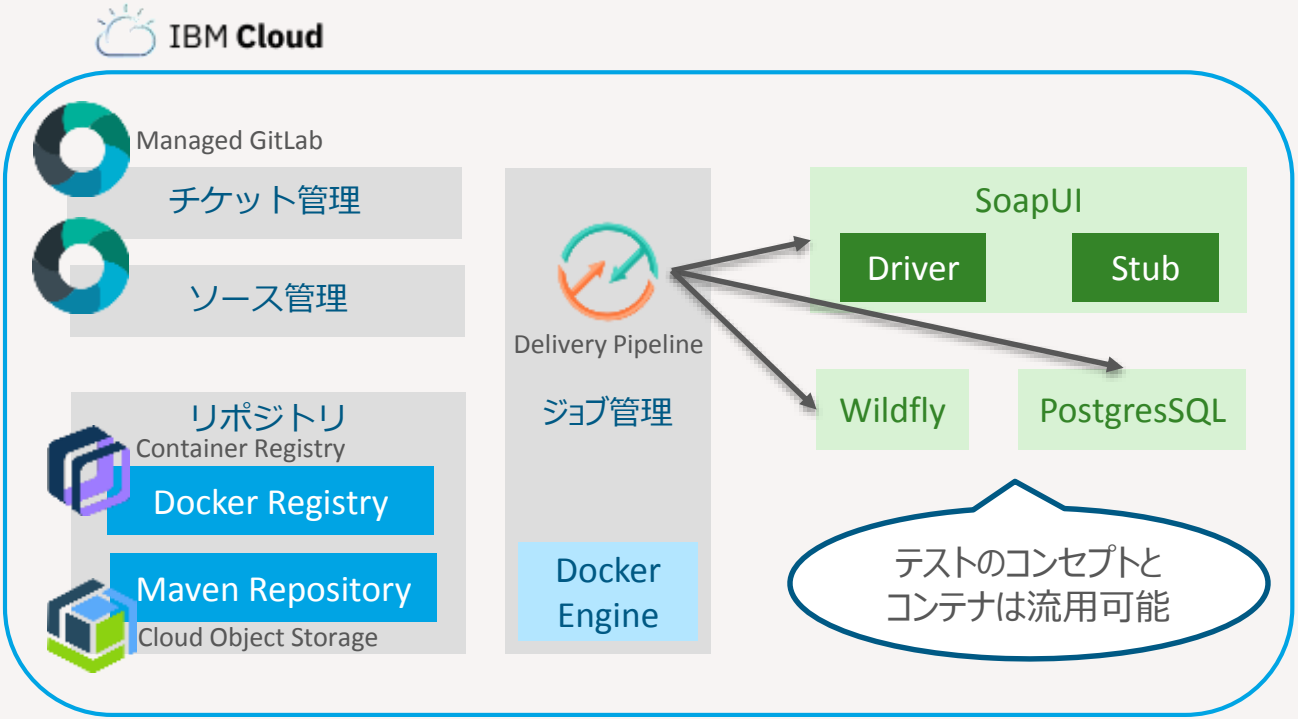
2.1 出来るだけ実行環境・テストツールにロックインされないこと



テストツールにはあえて自由度を持たせて
コンテナのみ流用するという考え方もできる
※ただし、ジョブなどはツールに合わせて一から実装する必要がある

フルコンテナ構成のCI/CD環境について

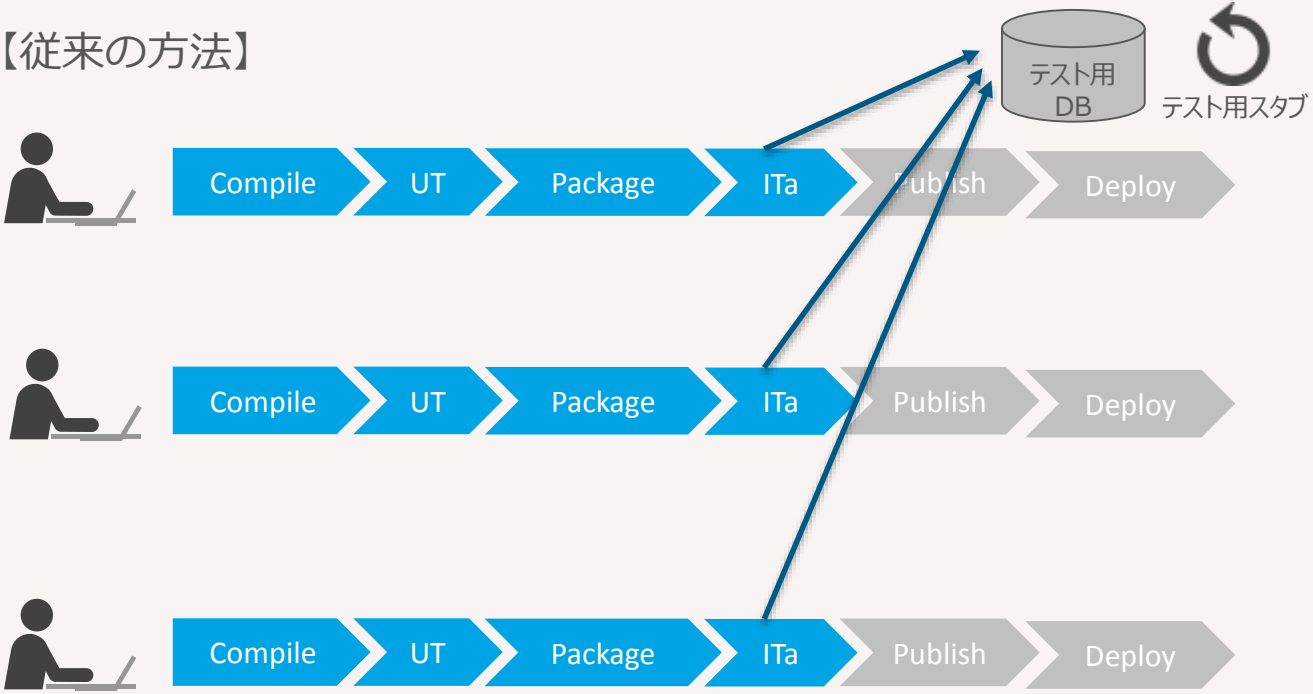
2.1 出来るだけ実行環境・テストツールにロックインされないこと



フルコンテナ構成のCI/CD環境について

2.2 プロジェクト規模に応じてスケール可能なテスト実行方法

【従来の方法】



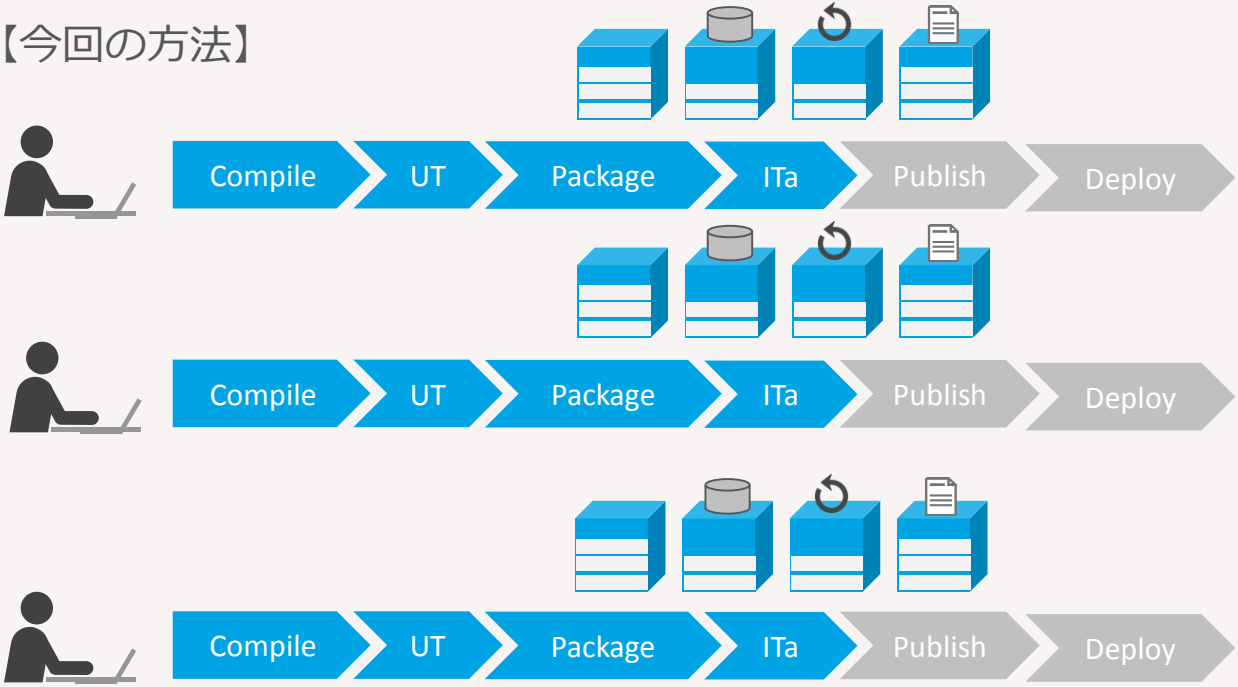
一つのDB・スタブを共用する

➡ パイプラインのプロセスを並列実行できない

フルコンテナ構成のCI/CD環境について

2.2 プロジェクト規模に応じてスケール可能なテスト実行方法

【今回の方法】

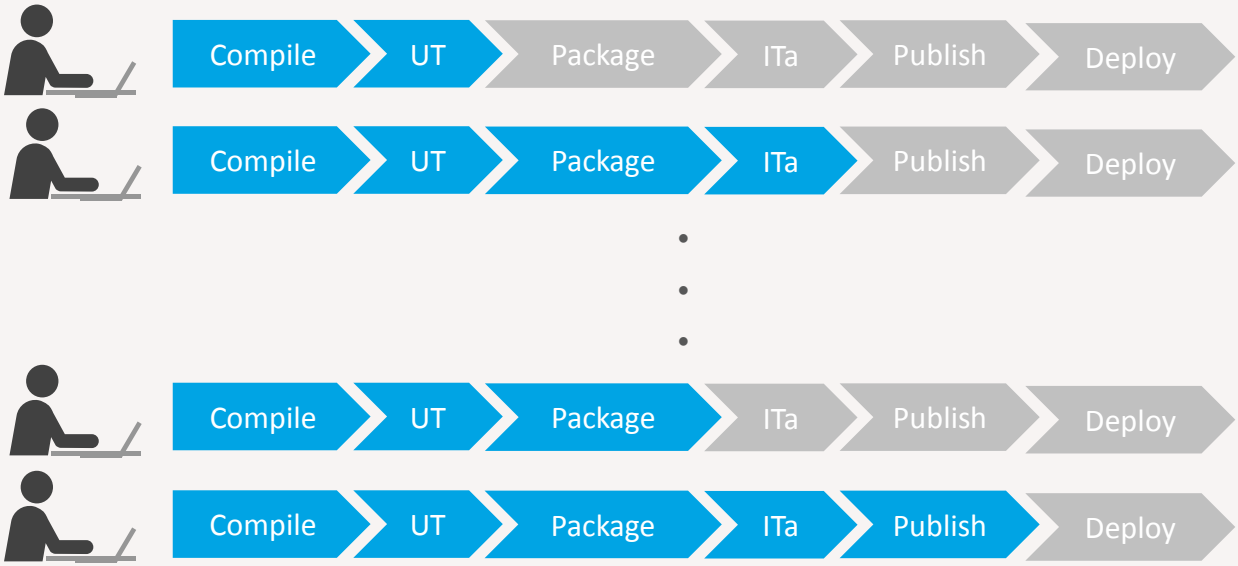


一連のパイプラインのプロセス 1 回ごとに独立してコンテナが起動する
➡ 並列実行に機能的な制限はない

フルコンテナ構成のCI/CD環境について

2.2 プロジェクト規模に応じてスケール可能なテスト実行方法

【今回の方法】



物理リソースの限界までスケールさせることが可能
※ただし、オーケストレーション・ツールは必須性が高い（後述）

フルコンテナ構成のCI/CD環境について

2. 横展開できるリファレンスケース

目標:
プロジェクト規模、CI/CD実行環境（オンプレ/クラウド）に関わらず
適用可能なリファレンスケースを作る

2.1 出来るだ ベンダーロックインされないこと
➡ OSSとコンテナを用い 確保

2.2 プロジェク 可能なテストの実行方法
➡ テストにコンテナ 確保

ポータビリティーが確保できた

スケーラビリティーが確保できた

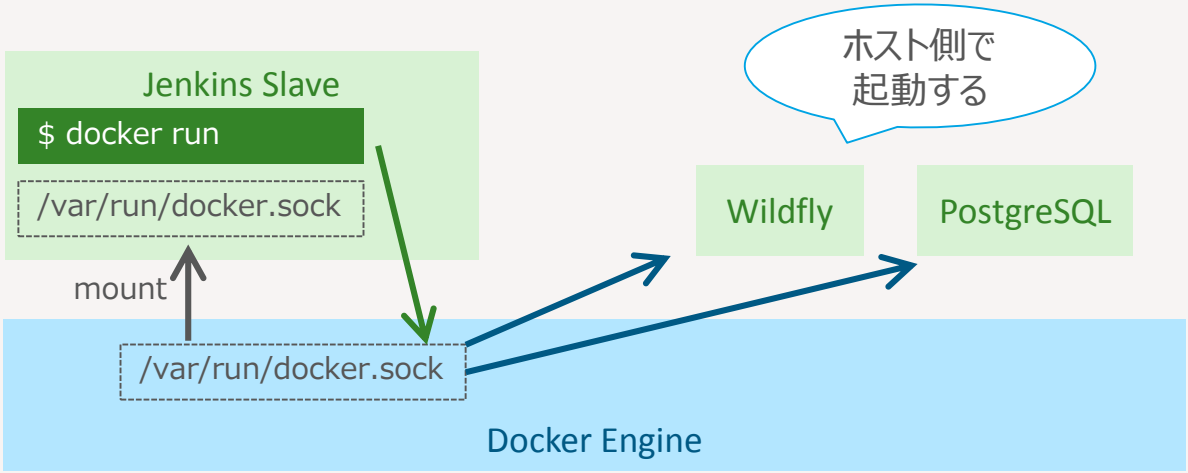
フルコンテナ構成のCI/CD環境について

実践的なプラクティスをいくつかご紹介

フルコンテナ構成のCI/CD環境について

Docker実行方法

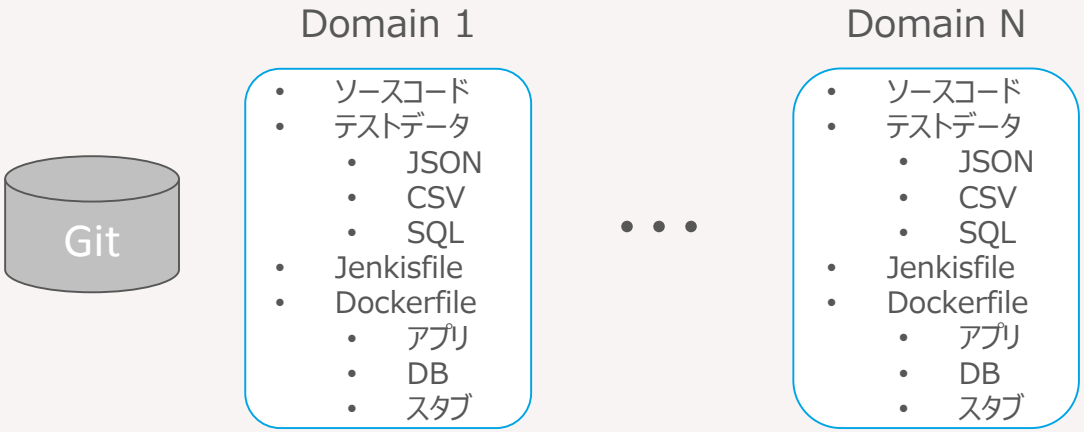
- Docker outside of Docker (DooD)
  - Slaveのコンテナ上で実行されたDockerコマンドは
    ホスト側のDocker環境で実行される



フルコンテナ構成のCI/CD環境について

パイプラインの設定はソースコードと一緒にリポジトリで管理

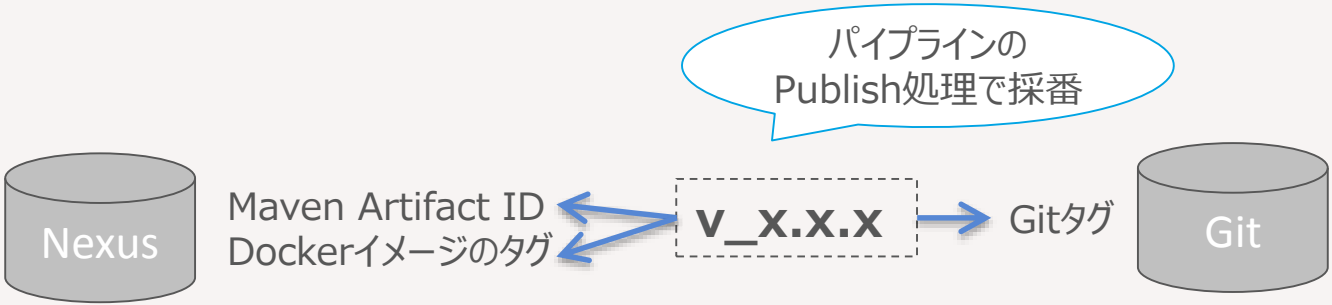
- ソースコード、テストデータ、Dockerfile、Jenkinsfile等はドメインごとにGitリポジトリで管理している
- Infrastructure as Codeの観点
- ローカル環境とCI/CD環境で同一のテストが実行可能（ローカル環境でDockerを使ったテストが行える）



フルコンテナ構成のCI/CD環境について

パイプラインの成果物は全て一意のリリースID持たせて管理する

- MavenのArtifact ID、Dockerイメージのタグ、Gitリポジトリのタグを揃える
- 問題が発生したとき、以前のリリースにロールバックできる



アジェンダ

- SI現場のよくある問題
- Microservice プロジェクトについて
- Javaのテスト実装について
- フルコンテナ構成のCI/CD環境について
- **まとめ**

まとめ

まとめ

フルコンテナ構成のCI/CD環境を構築した

- DBテスト、ITa相当のテストを含んだパイプラインを構築できた
- ポータビリティとスケーラビリティを兼ね備えたリファレンスケースを作成できた

課題

- オーケストレーション・ツールの必要性が高い

課題

オーケストレーション・ツールの必要性
① Port Binding
パイプラインプロセスごとに起動するコンテナに割り当てるポートの管理

- 現状
  - 現在の実装ではGitリポジトリ単位で固定のポートを利用
    - Jenkinsの処理実行時に読み込むプロパティを切り替える
    - ポートも指定してアクセスする

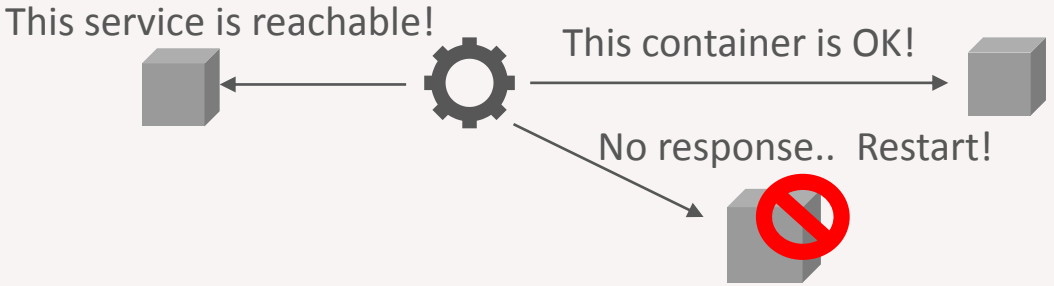
- 今後
  - オーケストレーション・ツールの提供する機能に応じて実装を行う
    - サービスはポート指定なしでアクセスできるようにする
    - ポートとホストを紐付け、デプロイされているアプリケーションは紐付けを変更するだけで即座にサービスとして公開できる

課題

オーケストレーション・ツールの必要性
② コンテナおよびサービス稼動確認
(例)SoapUIコンテナ上でスタブが起動しているか？のチェック

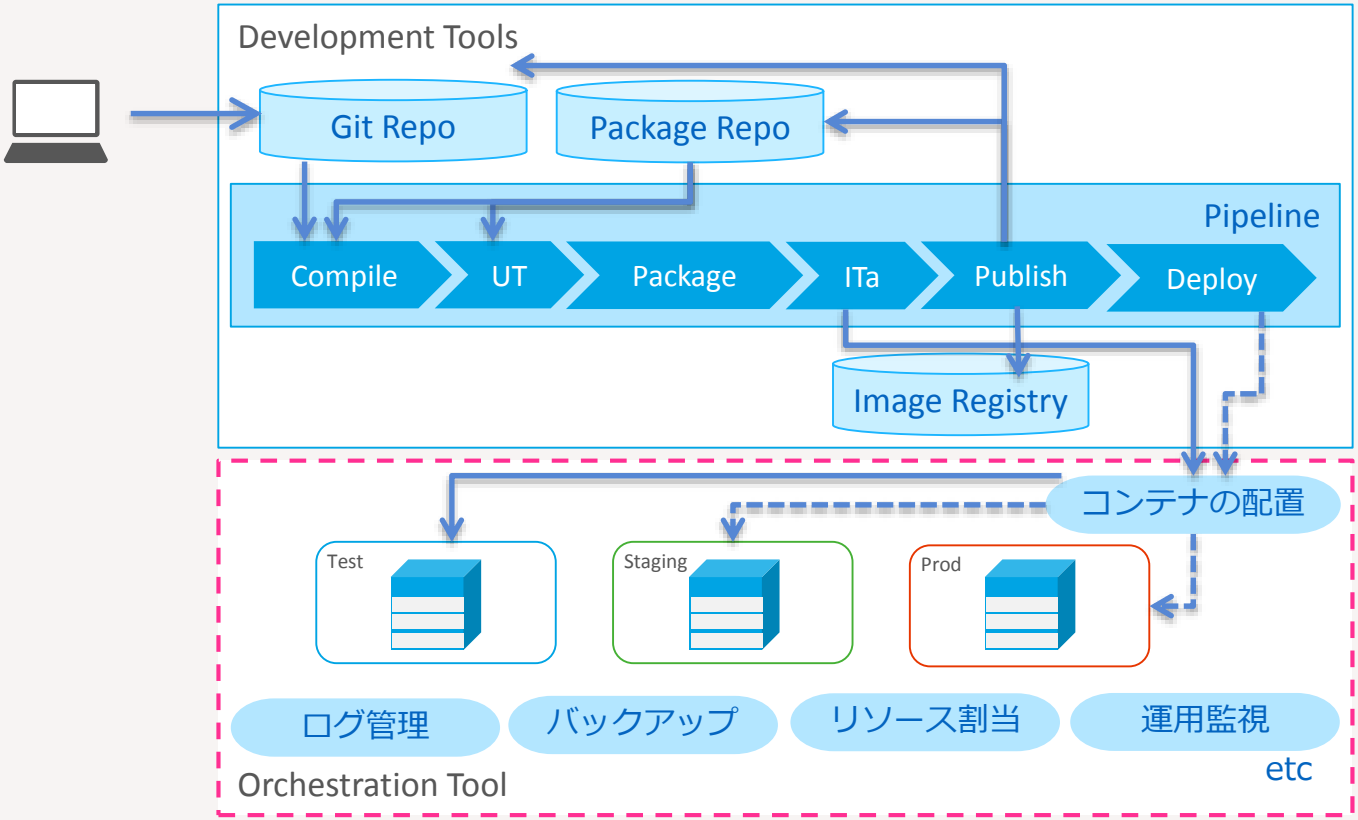
- 現状
  - Jenkinsfile内では起動時間の実績値を元にsleepで待つ実装（イケテナイ）
  - 想定する時間内で起動しない場合にはジョブは失敗してしまう

- 今後
  - オーケストレーション・ツールの提供する機能に応じて実装を行う
    - Liveness Probe：コンテナが実行中でなければ再起動を実施
    - Readiness Probe：サービス提供可能かチェックしエンドポイントに公開



課題

オーケストレーション・ツールを用いた理想的な構成を作りたい



ご静聴ありがとうございました

川沼 大輝

Twitter: @Santea3173

Email: kawanuma@jp.ibm.com

横山 夏実

Email: nykym@jp.ibm.com

EOF