クラウドネイティブ・アプリケーション開発のベストプラクティス

TIS株式会社
テクノロジー＆イノベーション本部
テクノロジー＆エンジニアリングセンター

商標

- Amazon Web ServicesおよびAWSは、米国および/またはその他の諸国における、Amazon.com, Inc.またはその関連会社の登録商標です。
- CNCFはThe Linux Foundationの登録商標です。
- Herokuはsalesforce.com, inc.の登録商標です。
- JBossは、米国およびその他の国における Red Hat, Inc. またはその子会社の登録商標です。
- OracleおよびJavaは、オラクルおよびその関連会社の登録商標です。
- Maven はApache Software Foundation の米国およびその他の国における商標または登録商標です。
- PythonはPython Software Foundationの登録商標です。
- DockerおよびDockerのロゴはDocker,Inc.の米国およびその他の国における商標または登録商標です。Docker,Inc.は本書で使用される他の用語について商標権を有している場合もあります。

その他の社名、製品名などは、一般にそれぞれの会社の商標または登録商標です。
なお、本文中では、TMマーク、Rマークは明記しておりません。

目次

1. クラウドネイティブとは
   - クラウドネイティブの定義
   - なぜクラウドネイティブなのか？
   - The Twelve-Factor App とは
   - コンテナとイミュータブル・インフラストラクチャ
2. クラウドネイティブなアプリの設計・実装のベストプラクティス
   - The Twelve-Factor App：移植性の最大化
   - The Twelve-Factor App：スケーラビリティ
   - The Twelve-Factor App：サーバ管理やシステム管理を不要に
   - The Twelve-Factor App：継続的デプロイのために
3. まとめ

研修の目的

- クラウド上で動作するアプリケーションを設計・構築する際にどのような考慮が必要なのかを学ぶ
- 自分たちの作ったアプリケーションがクラウド上で動作できるかを評価できるようにする

現場の課題

顧客が求めるスピードに応えきれていない。
モノリシックなアプリケーションが改善の足枷になっている。
- 変更を加える際の影響調査や全体テストが負担
- リリースは一大イベントでとても頻度を増やせない

自分たちで全ての環境を作るのは時間もコストもかかる。
非機能要件を考えて環境を用意するのは大変。
- 可用性の確保はどうしよう
- スケーリング時のことも考慮しなくては
- ログ収集、保管場所も設計しなくては

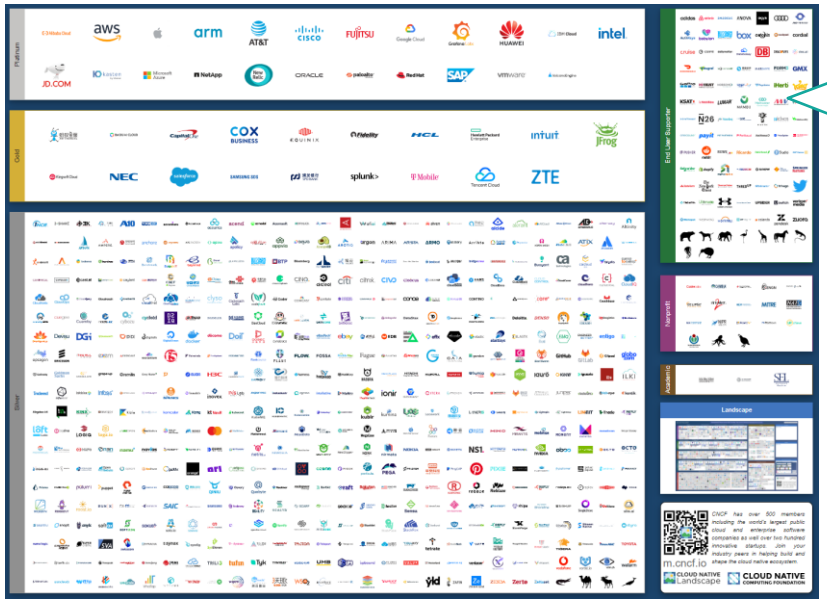
クラウドネイティブとは

Cloud Native Computing Foundation（CNCF）

コンテナ技術の推進と、その進化を取り巻くテクノロジー業界の足並みを揃えるために2015年に創設された非営利団体。

- クラウドの利点を生かすオープンソースの技術・サービスをホスト
- プロジェクトを支援することで企業や官公庁、公共機関などにおける**クラウドネイティブ化を推進**している。

CNCF 加入団体



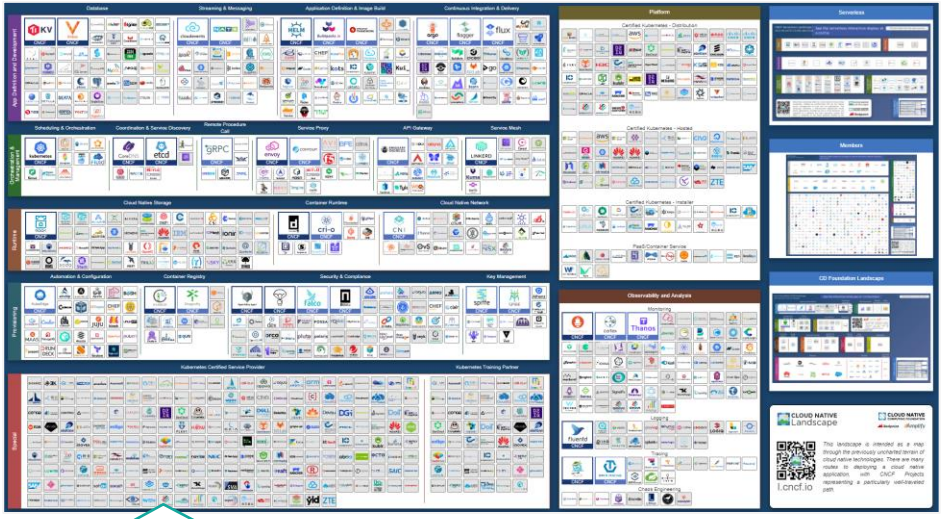
CNCF Cloud Native Landscape (https://landscape.cncf.io/)
2021年8月18日10時時点の情報を取得

大手クラウド事業者、ミドルウェア企業、ハードウェア製造企業、オープンソース・ソフトウェア企業、大学、その他非営利団体など、約500団体が加入している。(2021年8月時点)

Cloud Native Computing Foundation（CNCF）

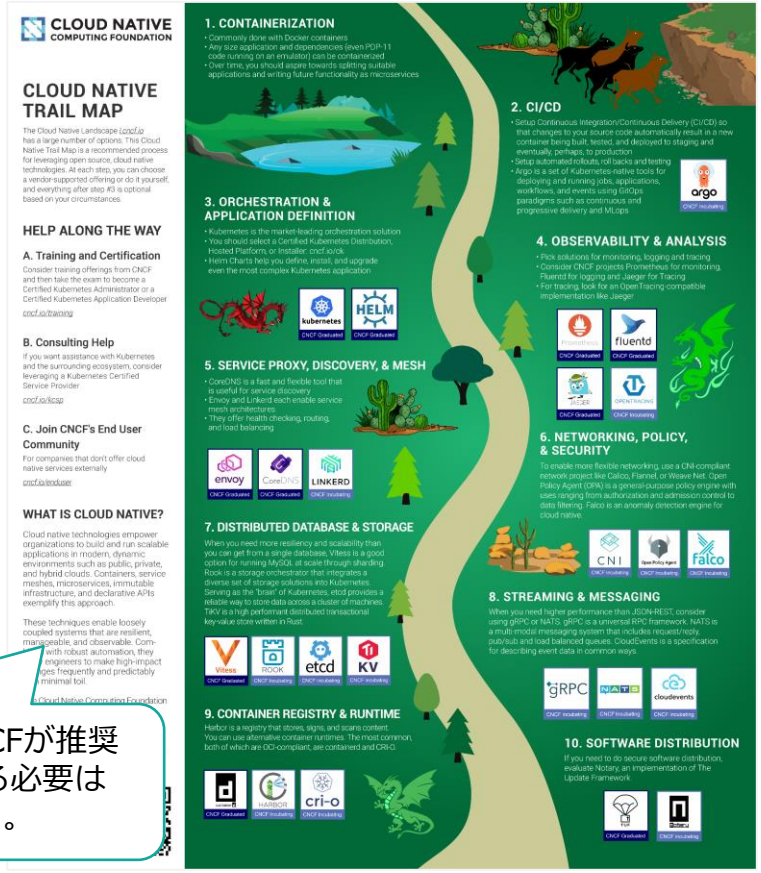
CNCFはその活動の一環として、企業がクラウドネイティブに向かう場合の指針をランドスケープとして公開。

Cloud Native Landscape



企業や開発者を支援するクラウドネイティブ技術のリソースマップ

Cloud Native Trail Map



クラウドネイティブへ向かうために、CNCFが推奨する10のステップ。特にこのとおりにやる必要はないが、よく通られる道をガイドしている。

CNCF Cloud Native Landscape (https://landscape.cncf.io/)
2021年8月18日10時時点の情報を取得

クラウドネイティブの定義（CNCF）

クラウドネイティブ技術は、パブリッククラウド、プライベートクラウド、ハイブリッドクラウドなどの近代的でダイナミックな環境において、**スケーラブル**なアプリケーションを構築および実行するための能力を組織にもたらします。このアプローチの代表例に、コンテナ、サービスメッシュ、マイクロサービス、イミュータブルインフラストラクチャ、および宣言型APIがあります。

これらの手法により、回復性、管理力、および可観測性のある**疎結合システム**が実現します。これらを堅牢な**自動化**と組み合わせることで、エンジニアはインパクトのある変更を最小限の労力で頻繁かつ予測どおりに行うことができます。

CNCF Cloud Native Definition v1.0
(https://github.com/cncf/toc/blob/main/DEFINITION.md#cncf-cloud-native-definition-v10)
2021年8月18日10時時点の情報を取得

なぜクラウドネイティブなのか？

クラウドネイティブは、クラウドの利点を**フル活用**してアプリケーションを構築・実行する考え方。技術の利用は、手段であり目的ではない。

柔軟なスケーリングを実現する

必要な時に必要な分のリソースを利用できるクラウドの特性を最大限に活かす。

- 並列化でシステムの冗長性を高め、堅牢にする

疎結合システム＋自動化により、アジリティを高める

ビジネスの要求に応えられるアプリケーションを俊敏に開発できるようにする。

- アーキテクチャレベルで影響を分離する
- CI/CDの導入などの自動化を活用する

クラウドネイティブなアプリの設計・実装のベストプラクティス

The Twelve-Factor App

2011年にHerokuのエンジニアが提唱したSaaS開発の方法論。
Heroku上でクラウドアプリケーションを開発・運用する際に行ってきたプラクティスから、**12の要素**を抽出したもの。

THE TWELVE-FACTOR APP

はじめに

現代では、ソフトウェアは一般にサービスとして提供され、*Web アプリケーション* や *Software as a Service* と呼ばれる。Twelve-Factor Appは、次のようなSoftware as a Serviceを作り上げるための方法論である。

- セットアップ自動化のために **宣言的な** フォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの **依存関係を明確化** し、実行環境間での **移植性を最大化** する。
- モダンな **クラウドプラットフォーム** 上への **デプロイ** に適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の **差異を最小限** にし、アジリティを最大化する **継続的デプロイ** を可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなく **スケールアップ** できる。

Twelve-Factorの方法論は、どのようなプログラミング言語で書かれたアプリケーションにでも適用できる。また、どのようなバックエンドサービス（データベース、メッセージキュー、メモリキャッシュなど）の組み合わせを使っていても適用できる。

背景

このドキュメントへの寄稿者は、何百ものアプリケーションの開発とデプロイに直接関わり、Herokuプラットフォーム上での仕事を通して、何百何千ものアプリケーションの開発・運用・スケールに間接的に立ち会った。

このドキュメントは、多種多様なSaaSアプリケーション開発現場での私たちの経験と観察をすべてまとめたものである。これは、アプリケーション開発における理想的なプラクティスを見出すための三角測量である。特に、アプリケーションが時間と共に有機的に成長する力学、アプリケーションのコードベースに取り組む開発者間のコラボレーションの力学、そしてソフトウェア腐敗によるコストの回避に注目している。

The Twelve Factor App (https://12factor.net/ja/)

The Twelve-Factor App が指向すること

- セットアップ自動化のために**宣言的な**フォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層の**OS**への**依存関係を明確化**し、実行環境間での**移植性を最大化**する。
- モダンな**クラウドプラットフォーム**上への**デプロイ**に適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の**差異を最小限**にし、アジリティを最大化する**継続的デプロイ**を可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなく**スケールアップ**できる。

➢ 言語やソフトウェアスタックに依存せず、適用範囲が広い。
➢ クラウド上で動作するアプリを設計・構築する際に、どのような考慮が必要かを示してくれる。

羅針盤やガイドラインとして有用

The Twelve Factor App (https://12factor.net/ja/)

The Twelve-Factor App

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III.設定
設定を環境変数に格納する

IV.バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI.プロセス
アプリケーションを1つもしくは複数のステートレスな
プロセスとして実行する

VI.ポートバインディング
ポートバインディングを通してサービスを公開する

VII.並行性
プロセスモデルによってスケールアウトする

VIII.廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX.開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI.管理プロセス
管理タスクを1回限りのプロセスとして実行する

その前に・・・

「The Twelve-Factor App」の各要素を説明する前に、
前提として押さえておきたい以下の説明をします。

- **コンテナ**
- **イミュータブル・インフラストラクチャ**

「The Twelve-Factor App」は、Herokuのエンジニアが提唱した方法論。
Herokuが実践していることを知ると、１２の要素が理解しやすくなる。

THE TWELVE-FACTOR APP

はじめに

現代では、ソフトウェアは一般にサービスとして提供され、*Web アプリケーション*や *Software as a Service* と呼ばれる。Twelve-Factor Appは、次のようなSoftware as a Serviceを作り上げるための方法論である。

- セットアップ自動化のために **宣言的な** フォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの **依存関係を明確化** し、実行環境間での **移植性を最大化** する。
- モダンな **クラウドプラットフォーム** 上への **デプロイ** に適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の **差異を最小限** にし、アジリティを最大化する **継続的デプロイ** を可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなく **スケールアップ** できる。

Twelve-Factorの方法論は、どのようなプログラミング言語で書かれたアプリケーションにでも適用できる。また、どのようなバックエンドサービス（データベース、メッセージキュー、メモリキャッシュなど）の組み合わせを使っていても適用できる。

背景

このドキュメントへの寄稿者は、何百ものアプリケーションの開発とデプロイに直接関わり、Herokuプラットフォーム上での仕事を通して、何百何千ものアプリケーションの開発・運用・スケールに間接的に立ち会った。

このドキュメントは、多種多様なSaaSアプリケーション開発現場での私たちの経験と観察をすべてまとめたものである。これは、アプリケーション開発における理想的なプラクティスを見出すための三角測量である。特に、アプリケーションが時間と共に有機的に成長する力学、アプリケーションのコードベースに取り組む開発者間のコラボレーションの力学、そしてソフトウェア腐敗によるコストの回避に注目している。

私たちの動機は、私たちがモダンなアプリケーション開発で見てきたある種のシステム的な問題への関心を高めること、この問題を議論するための共通の語彙を提供すること、そしてこの問題に対する広い概念的な解決策と専門用語を提供することである。フォーマットはMartin Fowlerの書籍、*Patterns of Enterprise Application Architecture* および *Refactoring* に着想を得ている。

背景
このドキュメントへの寄稿者は、何百ものアプリケーションの開発とデプロイに直接関わり、Herokuプラットフォーム上での仕事を通して、何百何千ものアプリケーションの開発・運用・スケールに間接的に立ち会った。

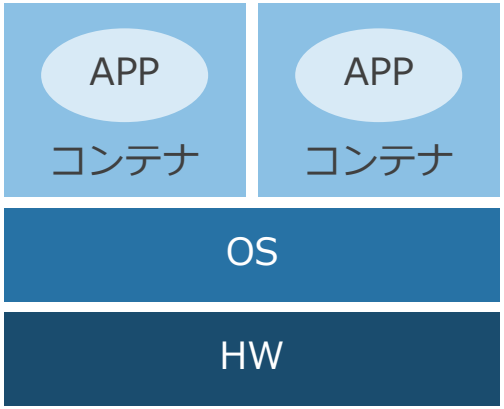
このドキュメントは、多種多様なSaaSアプリケーション開発現場での私たちの経験と観察をすべてまとめたものである。
～以下、省略～

コンテナ

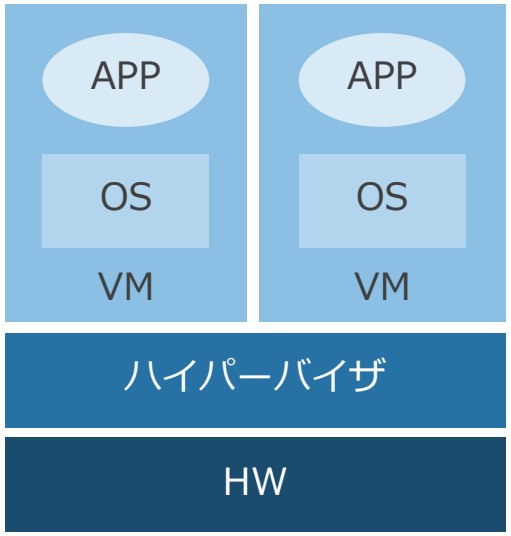
コンテナとは何か

コンテナはアプリケーションを実行する仮想的な環境です。
アプリケーション実行に必要な資材をコンテナイメージとして固め、イメージからコンテナを作成します。コンテナ同士は隔離されているため、独立した環境として動作します。

- 仮想マシンと比べて容量が小さく、起動が高速
- イメージを共有すればどこでも同じコンテナを動かすことができる

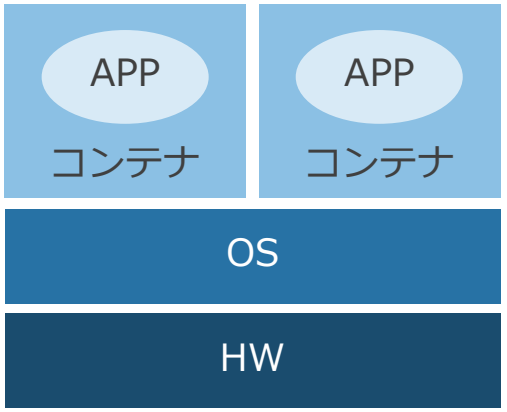


サーバ仮想化



- HWを共有する仮想化技術
- ハイパーバイザを使いVMを構成
- VMにはOSを含む

コンテナ



- OSを共有する仮想化技術
- コンテナエンジンを使いコンテナを構成
- コンテナにはOSを含まない

OSがない

コンテナにはOSが含まれていない。
VMに比べてコンテナの起動が**高速**であり、容量も少なく**軽量**。

使いまわしが容易

同じものを展開する事が容易。
（VMはクローンで複製出来ますが、コンテナはイメージから都度**生成**。）

一時的である

コンテナのIPアドレスやディスク領域は一時的に割り当てられたもの。
コンテナが停止するとそれらは**消え**、起動するとまた**新たに割り当て**られる。

コンテナの向いている場面

以下の要件にはコンテナの特性が活きてくる。

- 集約率を高めたい
- 大量に高速に起動したい
- システムに対して頻繁に変更をかけたい

具体的なシーン

- 物理サーバ台数の削減
- 負荷に応じた瞬時なスケール対応
- マイクロサービスアーキテクチャ
- DevOpsなど高速なアプリケーション開発

コンテナの特性

- 起動が高速かつ容量が軽量
- 同じものを展開することが容易
- 一時的である

コンテナの特性は、クラウドのメリットである必要なリソースを必要なだけ利用できるという点と**相性がよい**。

コンテナはクラウドをフル活用する上で重要な技術。

コンテナエンジンのデファクトスタンダード

Docker社が開発しているコンテナ型の仮想環境を作成、配布、実行するためのプラットフォーム。

Dockerはミドルウェアのインストールや各種環境設定をコード化して管理する。
これらにより、以下のような利点が生まれる。

- コード化されたファイル(Dockerfile) を共有することで、**どこでも誰でも同じ環境**が作れる
- 作成した環境(Dockerイメージ)を**配布しやすい**
- **スクラップ＆ビルドが容易**にできる

イミュータブル・インフラストラクチャ

イミュータブル・インフラストラクチャ

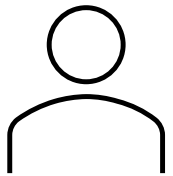
Immutable（イミュータブル）とは、「**不変**」を意味する英単語。

イミュータブルインフラストラクチャは、インフラを管理する手法の一つで、
一度構築した本番環境には更新やパッチの提供などの変更を加えず稼働させる
という考え方。

本番環境を変更・更新するときは、

1. 本番環境と同じ環境(開発環境)を別に用意する
2. 新しい環境で更新・パッチの提供・テストを実施する
3. 問題がなければ本番環境と入れ替え

変更しないというのは、廃棄を前提として
使い捨て可能な環境を構築することを意味します。



イミュータブル・インフラストラクチャで克服する課題

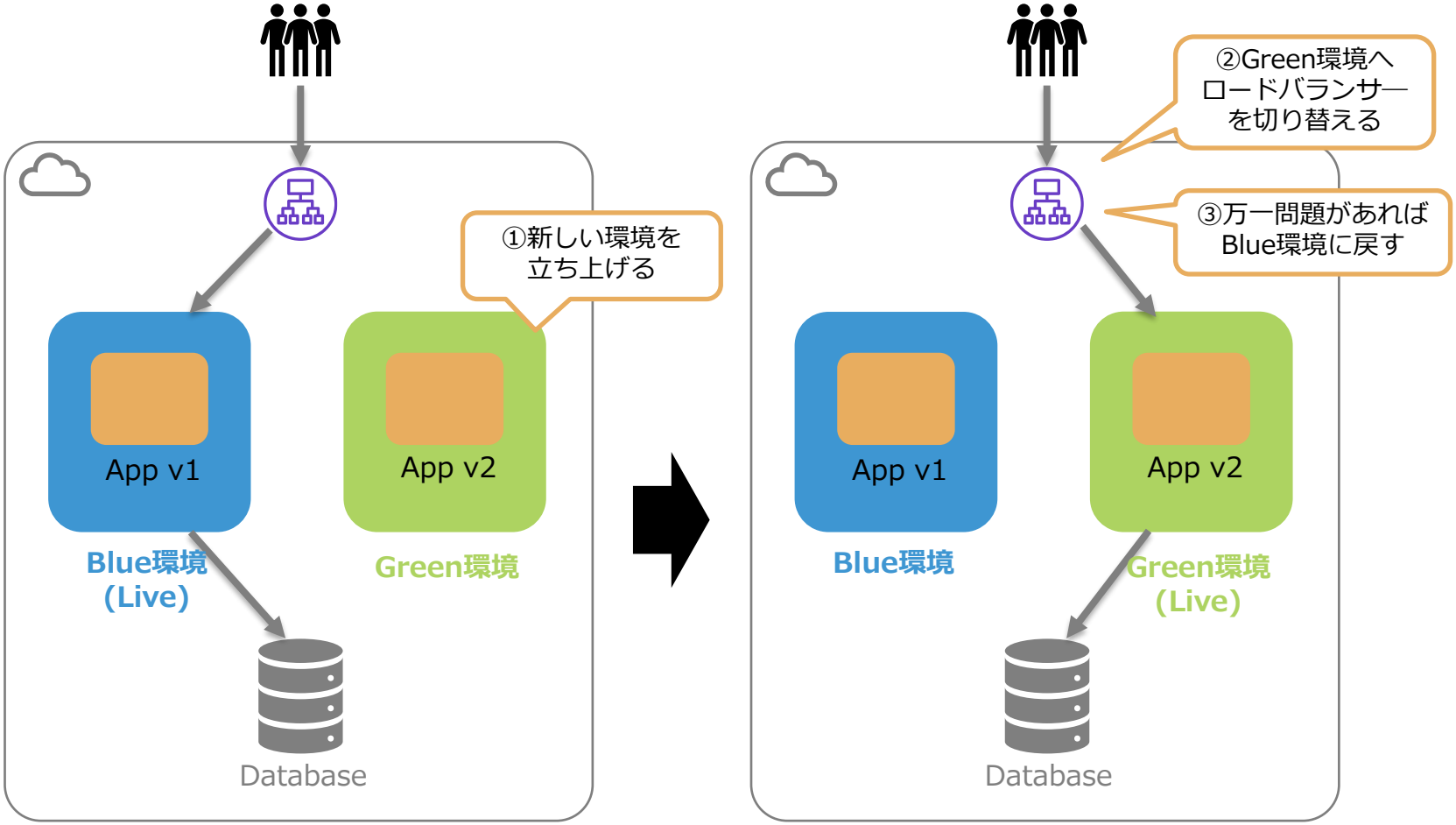
従来は・・・

- 本番環境に**直接**パッチの提供・更新作業・バグ修正などを行う
- リリースは、綿密に手順を整備し、リハーサルを実施するなどして万全を期す一大イベント

しかし、アプリケーションやパッチ適用が増えていくとシステムが複雑化していく。実際は以下のような事態を避けきれない。

- 把握しきれていない変更の蓄積に起因してリリースが失敗する
- リリース作業が予定時間を超過
- リリースに失敗した場合、切り戻しが非常に困難

ブルーグリーンデプロイメント



イミュータブル・インフラストラクチャで得られる効果

直接的な効果は、

- システムのダウンタイムを短くできる
- 新しい本番環境に切り替えて問題が生じた場合、すぐに現状の本番環境に切り戻せる

より本質的な効果は、

「アプリケーションコードと依存するソフトウェア一式の形成」
＋
「デプロイサイクルの繰り返し」

- 十分にテストされた検証済みの共通イメージを構築できる
- 予測できない動作やコード変更の意図しない結果が生じる可能性を削減する
- ビルド・デプロイの運用に必要な自動化を促進する

「上書きするリリース」から「切り替えるリリース」へ

イミュータブル・インフラストラクチャの考え方を適用すると、

- **リリースの敷居が下がる**
- **テスト容易性が向上する**

このときに**コンテナ**を使うとよい。
→　アプリケーションの展開が容易なため**スケーリング**しやすい！

ただし、アプリケーション設計や開発プロセスの**変化は必要**。

「The Twelve-Factor App」のプラクティスを取り入れることで解決。

あらためて・・・

クラウドネイティブなアプリの設計・実装のベストプラクティス

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスな
プロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- **下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。**
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

Ⅲ．設定

設定を環境変数に格納する

Configuration that varies between deployments should be stored in the environment.

概要

設定はコードに記述せず、環境変数として持たせる。
ここでの「設定」は、環境ごとに異なりえる設定値のことを指す。

「設定」に該当する例
- バックエンドシステムのURL
- データベースとの接続情報
- 外部システムとの接続用クレデンシャル

全環境で同じ値を取る場合は、ここでいう「設定」にはあたらない。
例えば、アプリケーション内部の設定は含まない。

なぜこのFactorが必要なのか

デプロイ対象の環境ごとにビルドを行わずに済むようにする。
言い換えると、同じビルドアーティファクトを全環境に適用できる。

- QA環境で動いたビルドアーティファクトであれば、
  本番環境でも動くことが保証できる

  「設定」は環境変数として、デプロイ対象の**環境側に保持**する。
  このような状況を作ると、CDを自動化してもリリースが**信頼**できるようになる。

  逆に言うと、単一のビルドアーティファクトを使えないようなら、
  各環境ごとにビルドが必要となり、そのテストも必要になってしまう。

このFactorに沿っているか否かの確認

- アプリケーションのコードベースを今からOSSにできるかを考える
  - 認証情報等を含んでいたらOSS化できない
  - 環境ごとの値を含んでいたらOSS化できない

どのようなケースが違反にあたるのか

× 「設定」をソースにハードコードしている
  - デプロイ対象環境ごとにビルドが必要になる

▲ プロパティファイルやYAMLファイルに「設定」を記述している
  - それがビルドアーティファクトに含まれるようなら、結局、デプロイ対象の環境ごとにビルドが必要になる
  - ビルドアーティファクトに含まれない場合であっても、ファイルフォーマットに関し実装言語ごとに流儀があり、統一的に扱うことができない

Parameter Store

- 設定データ管理と機密管理のための安全な階層型ストレージ
- パスワード、データベース文字列、ライセンスコードなどのデータをパラメータ値として保存可能
- プレーンテキストまたは暗号化されたデータとして保存可能
- 作成時に指定した一意の名前を使用して値を参照

コンテナのプロセスにAWSの Prameter Store から
環境変数を経由して「設定」を注入する例

dockerfile
```
ENTRYPOINT [“./entrypoint.sh”]
```

entrypoint.sh
```
export DB_HOST=$(aws ssm get-parameters ¥
    --name parameter_name ¥
    --query “Parameters[0].value”)
```

対策：設定を環境変数に外出しする

1. アプリケーションの設定ファイルを削除し、その設定値をデプロイ対象環境の環境変数として設定する
   - AWSのParameter Store
     コンテナのプロセスにAWSの Prameter Store から環境変数を経由して「設定」を注入する例

```
dockerfile
ENTRYPOINT [“./entrypoint.sh”]

entrypoint.sh
export DB_HOST=$(aws ssm get-paarameters ¥
    --name /database/sample/host ¥
    --query “Parameters[0].value”)
```

2. ソースコード上で設定ファイルから値を取得している箇所を環境変数から値を取得する方法に置き換える
   - Spring Bootで環境変数を取得するコード例

```
@Component
Public class Foo {
    @Value(“foo.bar”)
    private String bar = “hoge”; // default値
}
```

Ⅳ．バックエンドサービス

バックエンドサービスをアタッチされたリソースとして扱う

All backing services are treated as attached resources and attached and detached by the execution environment.

概要

DBや外部サービスはデタッチ・アタッチ可能なリソースとして捉える。環境変数に設定値を定義することでコードを修正しなくても切り替えられるようにする。

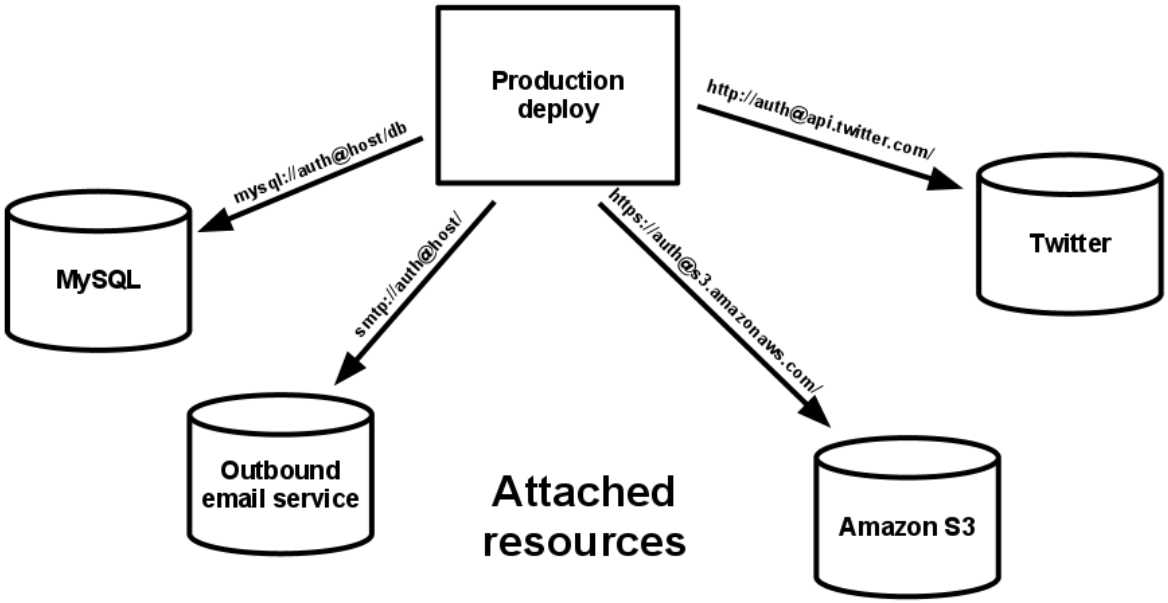
バックエンドサービスとは
アプリケーションが通常の動作の中でネットワーク越しに利用するすべてのサービス。データストア、メッセージング、キャッシュなど。ファイル置き場やディスクもバックエンドサービス(例えばAWSならS3)として扱う。

クラウドサービスの最大のメリットは、リソースを増やしたり、減らしたりすることが容易な点。切り替えが容易に行えることでメリットを享受できる。

なぜこのFactorが必要なのか

コード修正なしに、利用するサービスの切り替えを可能にする

- データベースが無応答になったとき、新しいデータベースを立ち上げて切り替える
- ローカルのPostgresqlデータベースをCloud SQL for PostgreSQLに切り替える



The Twelve Factor App - IV. バックエンドサービス
(https://12factor.net/ja/backing-services)

Ⅶ．ポートバインディング

ポートバインディングを通してサービスを公開する

Self-contained services should make themselves available to other services by specified ports.

概要

- アプリケーションにサーバーを組み込み、アプリケーション自らがポートをbindする
  - TomcatやJBoss、WebSphereといったアプリケーションサーバがポートをbindすることに頼らない
- 環境との「契約」＝アプリケーションがポートをbindすること
  - 環境は、その契約に基づいてポートの割り当てを行い、ルーティングやスケーリング、可用性の確保等を行う

なぜこのFactorが必要なのか

1. あるアプリケーションが他のアプリケーションのバックエンドサービスになることができる
2. コンポーネント間のやりとりがネットワークで分離され、疎結合化できる

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- **ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。**

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VII. ポートバインディング
ポートバインディングを通してサービスを公開する

VIII. 並行性
プロセスモデルによってスケールアウトする

IX. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

X. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

XI. ログ
ログをイベントストリームとして扱う

XII. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

Ⅷ. 並行性

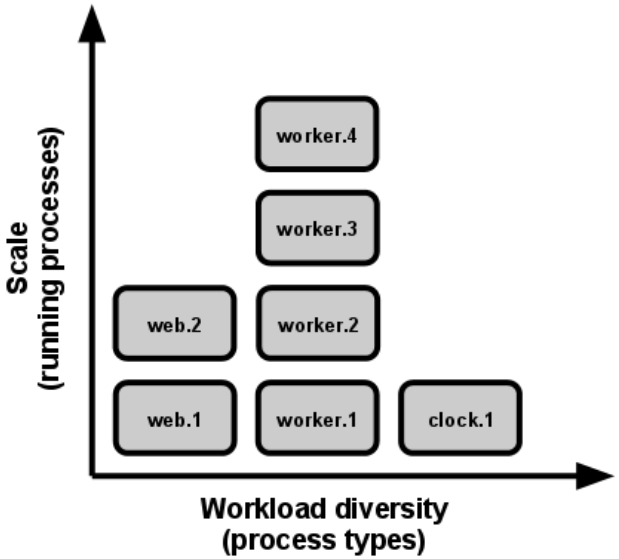
プロセスモデルによってスケールアウトする

Scale out via the process model.

概要

アプリケーションはプロセスモデルを用いて水平スケールアウトする

- 処理方式ごとにプロセスを分ける
- 個々のプロセスの数を増やし、負荷をプロセス間で分散させる



The Twelve Factor App - VIII. 並行性
(https://12factor.net/ja/concurrency)

なぜこのFactorが必要なのか

- スケーリングが容易になる

  × 垂直スケール（＝CPUやメモリの増強）だと、仮想マシンの再起動が必要
  → ダウンタイムが発生する

  ✓ 水平スケール（＝インスタンス数の増加）であれば、
  ダウンタイムは発生しない

Ⅵ．プロセス

アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

Applications should be deployed as one or more stateless processes with persisted data stored on a backing service.

概要

プロセスはステートレスかつシェアードナッシングにする。永続的な状態を持ちたい場合はDB等をバックエンドサービスとして利用する。

ステートレスの定義
- リクエスト処理前後のメモリの中身に仮定をおかないこと
- リクエストを跨ぐ「状態」はバックエンドサービスに外部化する

シェアードナッシングの定義
- ネットワークを除いてリソースを共有していない状態
- 異なるプロセス間でファイル等を共有しない
  - クラウド上ではスケーリング等により容易にプロセスが消失する
  - このプロセスの動的な増減に合わせてリソース共有を維持し続けるのは非常に難しい

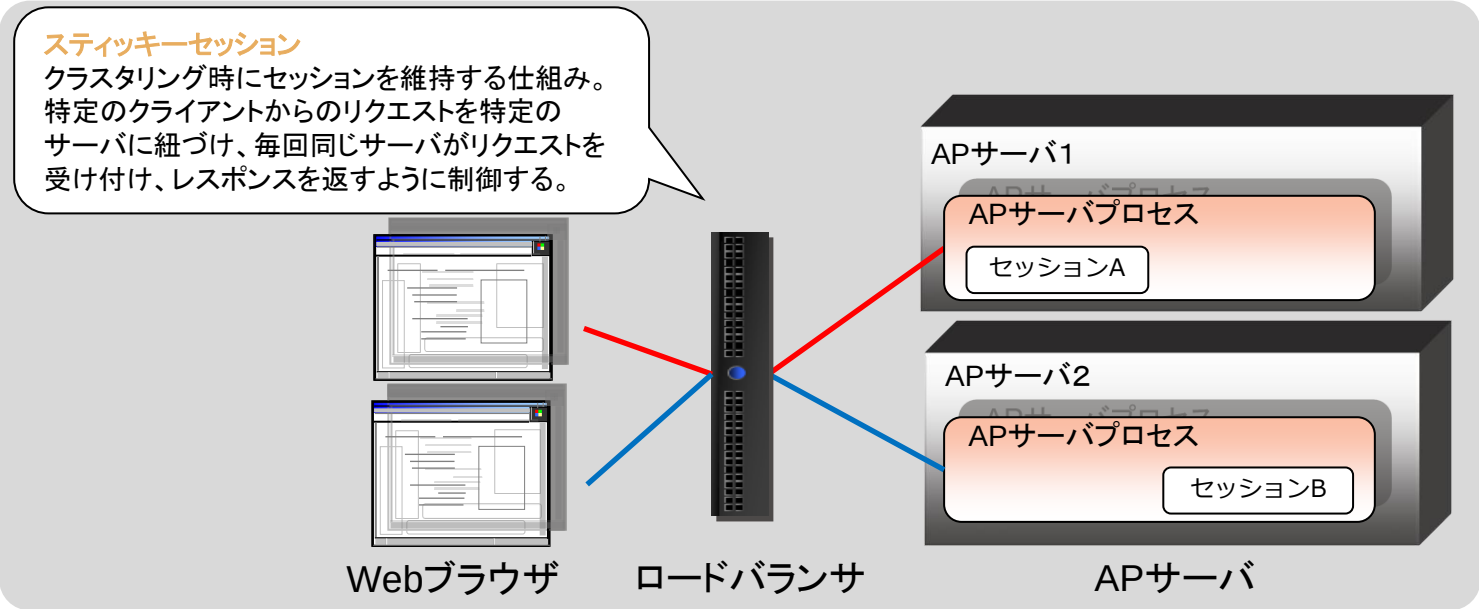
なぜこのFactorが必要なのか

- 個々のプロセスで水平スケーリングが可能になる

どうやって実現するのか

✓ セッションのようなデータをプロセス間で共有するときは
バックエンドサービスとして外部化する

× スティッキーセッションは利用しない



Ⅸ．廃棄容易性

高速な起動とグレースフルシャットダウンで堅牢性を最大化する

Fast startup and shutdown are advocated for a more robust and resilient system.

概要

1. アプリケーションは即座に起動させる
   - プロセスの起動が高速なほどスケールが容易になる
2. アプリケーションをグレースフルシャットダウンに対応させる
   - 停止前にリクエストを受け付け停止し、処理中のリクエストがなくなってから停止する

なぜこのFactorが必要なのか

1. リリースやスケーリングが早くなる
   - 遅い場合、立ち上がるまでユーザーからのリクエストが失敗する可能性すらある
2. 頑健性が増す
   - 容易にプロセスの移動や再起動が可能になる

よくある光景

- エンタープライズなアプリケーションの起動に数分かかる
  - 現代の高トラフィックの中では、この間に多くのリクエストが拒否されてしまう
  - ヘルスチェックが失敗してしまいトラフィックが受け付けられなくなる
  - 障害発生時のリカバリに時間がかかる
- 必要なタイミングで停止できない
  - バッチを停止させようとしても、処理が完了するまで止まらない
  - Webサービスがリクエストを受け付け続けてしまい、いつまでも停止できない

どう解決するのか

- 起動処理に重い処理を入れない
- グレースフルシャットダウンを実装する
  1. サービスポートのLISTENを中止して、新たなリクエストを受け付けない
  2. 処理中のリクエストが終了したことを確認して停止する

XI．ログ

ログをイベントストリームとして扱う

Applications should produce logs as event streams and leave the execution environment to aggregate.

概要

ログはアプリケーションから出力される、時間順に並んだイベントのストリームと考える。クラウドネイティブなアプリケーションでは、そのストリームの出力先や保管に一切関知しない。

なぜこのFactorが必要なのか

1. ファイルシステムの存在を前提にできない
2. アプリでログファイルを出力するとスケーリングの阻害要因になる
3. コードがよりシンプルになり、業界標準のツールやスタックに頼ることができる
4. クラウドプロバイダにログ集約や処理、保管といった非機能要件を担当させることができる

クラウドネイティブアプリは、ファイルの出力先には関知せず、stdout/stderrにログエントリを出力するのみ。

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- **モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。**
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

XII. 管理プロセス

管理タスクを1回限りのプロセスとして実行する

Run admin/management tasks as one-off processes.

概要

- 管理プロセス＝アプリのための一回限りの管理・メンテナス用タスク
  - DBのマイグレーション
  - 特定の修正のための一回限りのスクリプト
- 管理タスクは1回限りのプロセスとして、長時間実行されるプロセスと同じ環境で実行する
  - これらのプロセスは、あるリリースに対して実行され、そのリリースに対して実行されるすべてのプロセスと同じコードベースと設定を使う
  - 管理用のコードは、同期の問題を避けるためにアプリケーションコードと一緒にデプロイされるべきである

なぜこのFactorが必要なのか

管理プロセスを適用するために、都度、手作業でコマンドを叩く必要があるようではアプリケーションをスケールできない。

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- **開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。**
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

X．開発/本番一致

開発、ステージング、本番環境をできるだけ一致させた状態を保つ

All environments should be as similar as possible.

概要

すべての環境の差異を可能な限り小さく保つ

1. 同じスケーリング
2. 同じデータベース
3. 同じセキュリティルール、ファイアウォール
4. 同じデプロイ方法

環境に差異を与える要因

- **時間：**コードをチェックインしてからリリースまでの時間が長い
- **人：**デプロイ方法が組織によって異なる、多数の人を通してデプロイする
- **リソース：**開発環境と本番環境で異なるバックエンドサービスを使用するという妥協がある

時間に関わる問題

コードをチェックインしてからリリースまでの時間が長すぎる。

× 時間のギャップが発生すると、リリースにどんな変更を含めたのかがわからなくなる

× 開発者がコードの内容を忘れてしまう（とても大事）

チェックインから本番リリースまでの時間を短縮しなければならない。

- 数週間、数ヶ月といった単位から、数分から数時間といった単位へ
- 自動化されたテストとデプロイが必要(CI/CD)

クラウドがサポートする**ダウンダイムゼロ**のデプロイを利用する。

最初は恐怖を感じるシナリオだが、**当たり前になるとコード品質は大きく上がる**。

人(組織)に関わる問題

デプロイ方法が会社や組織によって異なる。

- 開発者自らがデプロイする
- 開発者から多数の人を通してデプロイする

自分の端末以外にアプリケーションをデプロイすべきではない。

適切なビルドパイプラインにより、すべてのアプリケーションはすべての環境に自動的にデプロイされる。

リソースに関わる問題

開発環境と本番環境で異なるバックエンドサービスを使用するという妥協がある。

- OracleやPostgreSQLの代わりにSQLiteを使う

妥協が環境間のギャップを大きくする。ギャップが大きくなるほど、
アプリケーションが動作することの予測可能性を下げてしまう。

予測可能性を下げると言うことは、信頼性を下げると言うこと。
信頼性を下げると言うことは、本番デプロイまで至る継続的なフローを失うということ。

「QAで動いて本番で動かない」というエラーは継続的デプロイに対する大きな摩擦を生む。継続的デプロイが妨げられることのリスクはトータルで非常に高くつく。

なぜこのFactorが必要なのか

- チームや組織に「アプリケーションはどこでも動く」という信頼感を与える
- QA環境では動くのに本番環境では動かないという状況を防ぐ

どうすれば良いのか

現代には様々な選択肢がある。

- クラウドの力を借りて、開発者に本番同様の環境を提供する
- Dockerなどのコンテナツールを利用して、本番同様の環境を開発者の端末上に構築する

Ⅴ．ビルド、リリース、実行

ビルド、リリース、実行の3つのステージを厳密に分離する

The delivery pipeline should strictly consist of build, release, run.

概要

コードは必ず以下のステージを経てデプロイまで至る。

1. **ビルド**
   - コードをビルドアーティファクトと呼ばれる実行可能な塊に変換する。
     この過程で依存ライブラリの収集も行う。
2. **リリース**
   - ビルドステージで生成されたビルドアーティファクトを受け取り、
     デプロイ対象環境の「設定」と合わせてリリースアーティファクトに変換する。
3. **実行**
   - 実行環境上で指定されたリリースアーティファクトを使い、アプリケーションの
     プロセスを起動する。

1. ビルドステージ

コードをビルドアーティファクト(Javaなら.jarや.war)に変換するステージ

ビルドアーティファクトとデプロイ環境 は 1：多 の関係にあり、複数の環境にデプロイされる。
ビルドアーティファクトの不変性(*immutability*)と環境の一貫性が
「QA環境で動けば本番環境でも動作する」という信頼性を生みだす。

- ビルドアーティファクトはコードリポジトリから生成する
- 宣言された依存ライブラリを取得し、ビルドアーティファクトにバンドルする
- ビルドアーティファクトはCIで生成する
  - テスト済みのコードをビルドする
  - ビルドアーティファクトとデプロイ環境は1:多の関係にあり、複数の環境にデプロイされる

2. リリースステージ

**ビルドアーティファクトと環境依存の「設定」を組み合わせ、
不変なリリースアーティファクトを生成するステージ**

- ビルドアーティファクトと環境設定からリリースする
  - これによりリリースアーティファクトの再生成が可能になる
- リリースアーティファクトに識別子を付与する
  - どの環境のリリースアーティファクトが失敗したのかがわかり、その原因も究明できる
  - どのアーティファクトでロールバックできるかがわかる

本番環境では難しいという場合であっても、ステージング環境へのデプロイからでも始められる。

3. 実行ステージ

選択されたリリースに対して、アプリケーションのいくつかのプロセスを起動するステージ

実行フェーズの多くはクラウドプロバイダに任せられるし任せるべきである。実行は基本的にはクラウドプロバイダ側で自動で起動されるようにし、開発者はアプリケーションに集中する。

- アプリケーションが生存し続けること
- ヘルスチェックがモニタリングされること
- ログが集約されること
- スケーリングできること

なぜこのFactorが必要なのか

本質は「自分たちのコードがどの環境でも動作する」という状態をつくること。「どの環境でも動作する」という自信が持てれば、本番リリースは怖くなくなる。

この信頼こそが、コードをチェックインして数時間でデプロイできるというようなクラウドネイティブな哲学の源泉。

「自分の環境だと動くんだけど…」は、これらステージの分離ができていないことを意味する。

Ⅱ．依存関係

依存関係を明示的に宣言し分離する

All dependencies should be declared, with no implicit reliance on system tools or libraries.

概要

- アプリケーションが必要とする依存関係は、アプリケーション側で厳密に宣言し、システムのツールやライブラリに頼らない。
- アプリケーションの依存関係をアプリケーション外部とは隔離し、漏れ出ないように保証すること。

なぜこのFactorが必要なのか

- これまでのアプリケーションでは、「アプリケーションに必要な依存ライブラリやツール」がサーバー側に用意されていることを前提としていた
- クラウド上でアプリケーションを動作させるとき、このようなライブラリ／ツールの存在は期待できないため、アプリケーション自身で動作前提を満たす必要がある

違反すると何が起きるか

- ステージング環境では動いたのに本番環境で動かない
  - ステージング環境と本番環境でインストールされているライブラリのバージョンが異なる
    - 実行時に初めて障害として検知される
    - 最悪の場合データ破損すらもたらし甚大な被害を与える
  - 環境ごとに包括的なテストを行わざるを得ず、テスト工数が爆発する要因となる
- ライブラリをバージョンアップしたら、他のアプリが動かなくなる
  - 依存関係が漏出している状態
  - チーム間の連携が必要となりアジリティを大きく落とす
- デプロイの自動化ができない
  - ライブラリの追加やバージョンアップの都度、サーバーへのライブラリインストール作業が必要になる

依存関係をアプリケーションに閉じ込める

- 依存関係管理ツールを利用する
  - 利用する言語に応じ、依存関係を宣言・解決するツールで依存関係を宣言する
    - Java:Maven、Gradle
    - Node.js:npm
    - Python:pip
    - Go:gomodules
    - Ruby:Bundler
- アプリケーションのみに影響する形で依存関係を解決する
  - アプリケーション専用のディレクトリにライブラリを配置することが多い
- リリースアーティファクト(成果物)に依存ライブラリを含める形でデプロイする

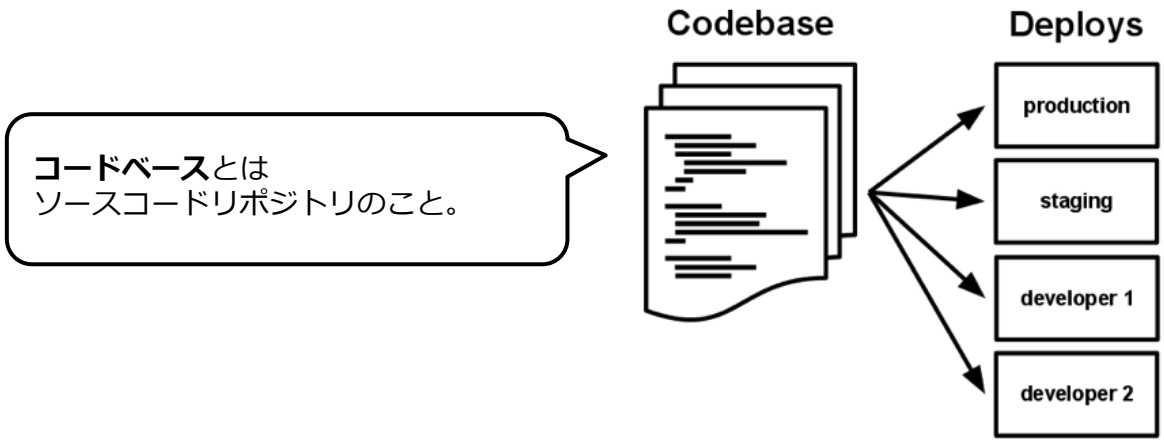
Ⅰ．コードベース

バージョン管理されている1つのコードベースと複数のデプロイ

There should be exactly one codebase for a deployed service with the codebase being used for many deployments.

概要

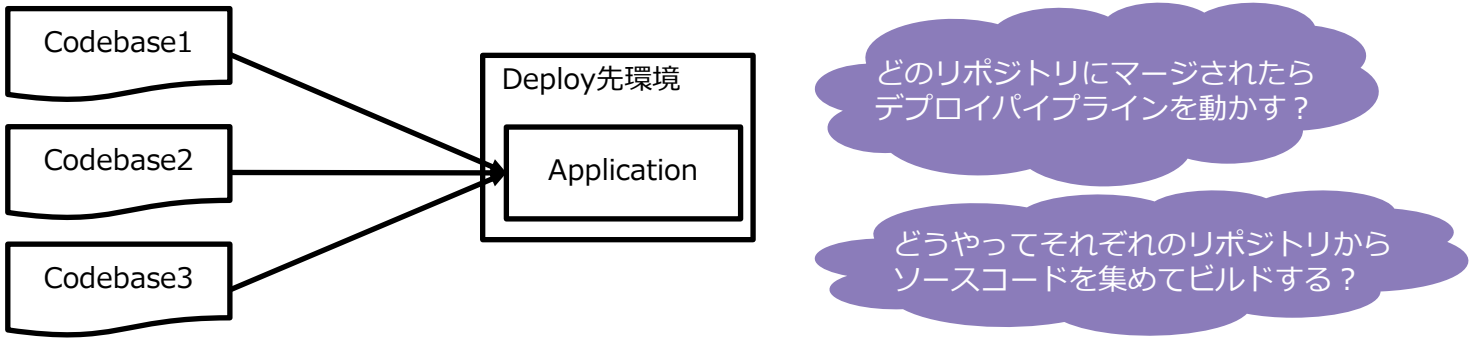
- デプロイするサービスに対し、コードベースを１：１に対応させる
- そのコードベースから、各環境へのデプロイ成果物を生成できるようにする



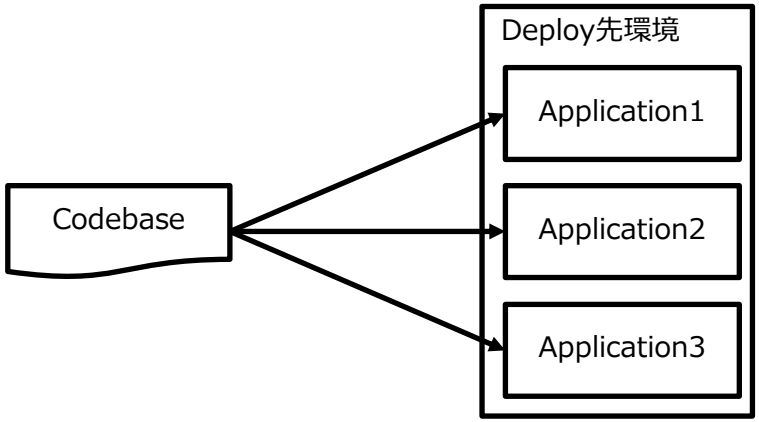
The Twelve Factor App - I. コードベース
(https://12factor.net/ja/codebase)

どのようなケースが違反にあたるのか

- １つのアプリケーション、複数のコードベースで構成される場合
  →ビルドプロセスやデプロイプロセスの自動化が難しくなる



- １つのリポジトリ、複数のアプリケーションが作られる場合
  →アプリケーションが密結合になる可能性が高く、拡張性が低下する



まとめ

様々な環境(QAや本番環境)へリリースする際も、デプロイするアーティファクト（生成物）は同じコードベースから生成する。

- １つのチームで共同作業を行う上では、１アプリケーションに対して１コードベースでスムーズに開発が進む
- チーム開発だけでなく、CI/CDに関する自動化も行いやすい
- コードベースが同じなので、各環境に対して安定したリリースを行うことができる

まとめ

まとめ

「The Twelve-Factor App」とは何であったか？
次のようなSaaS開発の方法論。

- セットアップ自動化のために**宣言的な**フォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層の**OS**への**依存関係を明確化**し、実行環境間での**移植性を最大化**する。
- モダンな **クラウドプラットフォーム**上への**デプロイ**に適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の**差異を最小限**にし、アジリティを最大化する**継続的デプロイ**を可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなく**スケールアップ**できる。

The Twelve Factor App
(https://12factor.net/ja/)

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- **下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。**
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- **ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。**

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VII. ポートバインディング
ポートバインディングを通してサービスを公開する

VIII. 並行性
プロセスモデルによってスケールアウトする

IX. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

X. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

XI. ログ
ログをイベントストリームとして扱う

XII. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- **モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。**
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

The Twelve-Factor App

- セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- **開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。**
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VI. ポートバインディング
ポートバインディングを通してサービスを公開する

VII. 並行性
プロセスモデルによってスケールアウトする

VIII. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

IX. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

X. ログ
ログをイベントストリームとして扱う

XI. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

The Twelve-Factor App

- **セットアップ自動化のために宣言的なフォーマットを使い、プロジェクトに新しく加わった開発者が要する時間とコストを最小化する。**
- 下層のOSへの依存関係を明確化し、実行環境間での移植性を最大化する。
- モダンな クラウドプラットフォーム上へのデプロイに適しており、サーバー管理やシステム管理を不要なものにする。
- 開発環境と本番環境の差異を最小限にし、アジリティを最大化する継続的デプロイを可能にする。
- ツール、アーキテクチャ、開発プラクティスを大幅に変更することなくスケールアップできる。

I. コードベース
バージョン管理されている1つのコードベースと複数のデプロイ

II. 依存関係
依存関係を明示的に宣言し分離する

III. 設定
設定を環境変数に格納する

IV. バックエンドサービス
バックエンドサービスをアタッチされたリソースとして扱う

V. ビルド、リリース、実行
ビルド、リリース、実行の3つのステージを厳密に分離する

VI. プロセス
アプリケーションを1つもしくは複数のステートレスなプロセスとして実行する

VII. ポートバインディング
ポートバインディングを通してサービスを公開する

VIII. 並行性
プロセスモデルによってスケールアウトする

IX. 廃棄容易性
高速な起動とグレースフルシャットダウンで堅牢性を最大化する

X. 開発/本番一致
開発、ステージング、本番環境をできるだけ一致させた状態を保つ

XI. ログ
ログをイベントストリームとして扱う

XII. 管理プロセス
管理タスクを1回限りのプロセスとして実行する

まとめ

The Twelve-Factor App のメリットをいかにクラウド上で活かすか？

The Twelve-Factor App を設計・実装するときにDockerは最適！
→「環境」との契約を守るインタフェースになる

- **依存関係：**Dockerイメージの中に依存性を封じ込めることができる
- **設定：**環境変数経由でしかDockerコンテナに設定値を渡せない（渡しにくい）
- **ビルド、リリース、実行：**どんな言語で実装されたアプリケーションでもビルドアーティファクトをDockerイメージとしてしまえば統一的に扱える
- **プロセス：**アプリケーションはDocker Container単位でスケールアウトできる
- **ポートバインディング：**ポートバインディングさえしてしまえば、あとはクラウドに任せられる
- **ログ：**ログドライバ経由でstdout/stderrに渡す

The Twelve-Factor App に従えば、
Dockerイメージとしてアプリケーション化が容易。

まとめ

どこから始めるべきか？

理想的なCloud Nativeアプリケーションに関する情報はたくさん存在するが、
我々は以下の現実に直面する。

- **開発チームのスキルセット**
- **予算**

それでも「**少しずつ**」というアプローチは適用できる。

- **The Twelve-Factor App**の12要素を**Cloud Nativeの採点表**にしよう
- 特定の要素に従うと、他の要素にも従いやすくなる（好循環が生まれる）

ゴールイメージを設定して、段階を踏んでいくことで実現できます！

THANK YOU

ITで、社会の願い叶えよう。

TIS INTEC
Group