

Docker講座

0-1 このtechプレゼンの進め方

40分のプレゼン

- ・ 0章-イントロダクション
- ・ 1章-概要説明
- ・ 2章-実演しながら使いから紹介
- ・ 3章-まとめ

-

0-2 ゴール

この40分間のTechプレゼンで、次のことが理解できるようになります：

- Dockerが解決する課題
- Dockerの基本的な仕組みと構成要素
- 仮想マシンとの違い
- コンテナ技術がもたらす開発・運用へのインパクト
- 開発現場やチームでの活用イメージ

プレゼンの後にgithubの専用レポジトリの中のデータをつかって各々実践

-



自宅サーバーに専用のVMを用意したので、興味がある人は実際のサーバーを使ってデプロイできます！

1-1. そもそも「Docker」とは？

▶ 開発現場のあるある

- 「自分の環境では動いたんだけど...」
- 開発環境・本番環境の違いによる不具合
- セットアップに時間がかかる

▶ Dockerを一言で説明すると...

「アプリケーションとその動作環境を一つのパッケージ (コンテナ) として実行できる技術」

...とは聞いていても、普段個人開発で使っていてあまりそのメリットを体験することはあまり多くないと思います

1-2. Dockerの仕組み

Dockerがどう動いているのか？

コンテナがどうやって作られて、動くのか？

ここでは仕組みをわかりやすく順番に説明していきます。

▶ Dockerは「コンテナ」という軽量な仮想環境を作る仕組み

- Dockerは仮想マシンのようにOSごと仮想化するものではありません。
 - ホストOSのカーネルを共有しながら、アプリケーションごとに分離された実行環境 (コンテナ) を作ります。
-

▶ Dockerの基本構成図 (イメージ)

```
[ ホストOS ]  
  ↓  
[ Docker Engine ( デーモン ) ]  
  ↓  
[ イメージ ] → [ コンテナ ]
```

1-3. 仮想マシンとの違い

比較項目	仮想マシン (VM)	Docker (コンテナ)
起動時間	数十秒～数分	数秒
オーバーヘッド	高い (OSごと仮想化)	少ない (ホストOSを共用)
イメージサイズ	GB単位が多い	MB～数百MB
使用用途	完全な仮想環境が必要なとき	軽量な環境分離が必要なとき

➡ Dockerは「速い・軽い・手軽」な仮想環境

1-4. なぜ今Dockerが流行りなのか？

- クラウド時代との相性がいい
→ AWS, GCP, Azureなど各社が対応
dockerを使えばクラウド間の環境差異を吸収したり、高速にスケーリングができる
 - CI/CDパイプラインでの活用
現代の開発では、コードを書いたらすぐにテストする傾向にあり、
「問題なければ自動で本番反映」といった「自動化の流れ (CI/CD) 」が当たり前になっている。
Dockerを使えば、本番環境へのデプロイもDockerイメージで統一できる
ため、一貫した自動化パイプラインが作れる。
 - インフラエンジニアだけでなく、全ての開発者に役立つ
個々の開発者が自分で環境構築・運用まで簡単にできるため、チーム全体で「同じ環境」を共有できるようになり、全開発者の生産性向上ツールとなっている。
-

1-5. 現場での使われ方 (イメージ)

- 開発環境の統一：新しいメンバーも `docker-compose up` で即参入
- マイクロサービス構築：複数サービスをコンテナで管理
- 自動テスト環境の構築：毎回クリーンな状態でテスト実行

- スケーラブルな本番環境：Kubernetesと連携して自動スケーリング

▶ 豆知識「dockerの導入事例」

実際の導入事例を見ていきます

Docker導入事例

企業/団体名	活用内容
Spotify	音楽ストリーミングサーバーやデータパイプラインをDockerコンテナ化し、開発チームごとに独立した環境を運用。環境構築スピードとデプロイの一貫性を大幅改善。
PayPal	アプリケーションの開発・テスト環境をDockerベースに移行し、従来の仮想マシン環境よりも90%以上の起動時間短縮。
ADP（給与・人事サービス）	各国ごとに異なる法律・仕様対応をコンテナ化して分離し、迅速なリリースサイクルを実現。

Kubernetes導入事例

企業/団体名	活用内容
Airbnb	数百種類のマイクロサービスをKubernetes上で管理。オートスケールと自動復旧による可用性向上を達成。
Pinterest	毎日数億件以上のユーザーリクエストを処理。Kubernetesでトラフィックの急増にも自動対応できる体制を整備。
日本経済新聞社	記事配信システムをKubernetesへ移行。コンテンツ配信の高速化、トラフィック変動への柔軟対応を実現。

1-6. Dockerの中身：どうやって動く？

主な構成要素

- イメージ (Image)
アプリとその環境を一体化した"設計図"
- コンテナ (Container)
イメージを実行した実体。軽量の仮想環境。
- Dockerfile
イメージを自動で作るためのレシピ。
- Docker Engine
コンテナを動かす仕組みそのもの。

▶ 例えると...

あなたは唐揚げ定食のお店(webアプリケーション)を経営しています。しかしお客さんがとても増えて待たせてしまうことが多くなってしまいました。また、他の地域のお客さんから「近くにお店が欲しい」とよく言われるようになります。

そこで自分のお店を暖簾分けする(dockerコンテナを増やす)ことにしましたが、また人気の唐揚げ定食のお店(webアプリケーション)を0から作ろうとなるととても大変です。

なので、名物の唐揚げのレシピや運営のノウハウをまとめた秘伝の書物(dockerfile)をフランチャイズ店の店長(dockerエンジン)に渡したので、唐揚げ定食のお店のノウハウがなかった新しい店長でも簡単に運営を軌道に乗せることができるようになりました。

ちなみにk8sの話を付け加えると...

そんな感じであなたは唐揚げ定食のお店を全国で100店舗以上展開する大規模なチェーンを築き上げました。ここまで大規模になると個人で管理しきれない規模ではないので、会社を立ち上げて(k8sを導入して)運営しています。

しかし、何処かの国の不動産王の政策の影響で日本全国が不況に陥ってしまい唐揚げ定食の需要が激減してしまいました。ただ幸いにもあなたを含めた経営陣はチェーン網をいち早く再編したため(オーケストレーションを意識してコンテナを組んでいたため)リストラは発生してしまいましたが、何処かの「突然出てきそうな店名のステーキ店」のように経営危機に落ちいることは回避できました。

1-7. よくある誤解

- ✕ DockerはVMの代わり → 違います！より軽量で目的が違う
 - ✕ Dockerを使えばセキュリティ万全 → セキュリティ設計は別途必要
 - ✕ 難しそう → 基本操作だけなら数コマンドで始められる
-

1-8. dockerをどう活かす？

- 自分のアプリをDocker化してポートフォリオに
 - チームで共通環境をDockerで構築・共有
 - 今後の「Kubernetes」「CI/CD」「クラウド活用」にスムーズにつながる
-

1-9 概要まとめ

- Dockerはアプリ + その環境をひとまとめにして動かす技術
 - 軽量・高速・再現性が高く、現代の開発スタイルにマッチ
 - 小規模でも導入メリットあり
-

1-10. 上級者向け

さらにDockerの内部構造まで理解したい方向けに、上級者向けパートを用意しました。(時間の都合上プレゼンでは扱いません)

▶ 1-10-1. Dockerのネットワーク仕組み

Dockerでは、コンテナ同士やホストとの通信を仮想ネットワークで管理しています。

種類	特徴
bridge (デフォルト)	同一ホスト内コンテナ同士の通信に利用
host	ホストマシンのネットワークをそのまま使う

none	ネットワークに接続しない (完全隔離)
overlay	複数ホスト間でネットワークを構成 (Swarmなど)

◆ よく使うコマンド

- ネットワーク一覧を表示

```
docker network ls
```

- 新しいネットワークを作成

```
docker network create my-network
```

- コンテナ起動時にネットワークを指定

```
docker run --network=my-network my-container
```

▶ 1-10-2. Dockerのデーモンとクライアントアーキテクチャ

Dockerは**クライアント-サーバーアーキテクチャ**を採用して動いています。

ユーザーがコマンドを打つだけでコンテナが作られるのは、裏側でデーモンがリクエストを受け取って動いているためです。

◆ アーキテクチャの基本構成

役割	説明
Docker CLI	ユーザーが操作するコマンドラインツール (`docker run` など)
Docker Daemon	バックグラウンドで動き、コンテナやイメージを実際に管理するサーバープロセス
Docker API	CLIとデーモンの間で通信を行う仕組み。UNIXソケットやTCPで接続される

◆ 通信の流れ

```
```plaintext
```

```
[ユーザー]
```

```
↓ (CLI コマンド)
```

```
[Docker CLI]
```

```
↓ (API リクエスト)
```

[Docker Daemon]  
↓ (レスポンス)  
[Docker CLI]

- CLIは、Docker API経由でリクエストを送信します。
- Daemonがそのリクエストを受け取り、実際にコンテナ作成・起動・削除などを行います。
- 結果をCLIに返します。

### ◆ デーモンが待ち受ける場所

通常：

/var/run/docker.sock ( Unixドメインソケット )

ローカルのCLIとデーモンがこのソケットを介して通信します。

リモート管理時 ( オプション )：

TCPポート ( 例：2375番ポート )

設定により、リモートマシンのDockerデーモンを直接操作することも可能です。

### ◆ デーモンとセキュリティ

/var/run/docker.sock にアクセスできるユーザーは、root権限と同等と見なされます。

セキュリティ上、アクセス制御 ( dockerグループ管理やRootlessモード ) を設定することが推奨されます。

## ▶ 1-10-3. コンテナのライフサイクルと状態管理

Dockerコンテナは、ただ「起動する・停止する」だけでなく、  
**ライフサイクル ( 生成～削除までの流れ ) と、それぞれの状態を持っています。**

これを理解すると、より正しくDockerを扱うことができるようになります。

(2-0も参照)

## ▶ 1-10-4. イメージのレイヤー構造

実のところDockerイメージは単なるファイルの塊ではなく、  
**レイヤー ( 層 ) 構造によって成り立っています。**



---

## ◆ レイヤーとは？

- Dockerfileの各命令 ( FROM、RUN、COPYなど ) ごとにレイヤーが作られる
- レイヤーは読み取り専用
- 変更があった場合は、新しいレイヤーが上に積み重なる

---

## ◆ レイヤーの仕組み図 ( イメージ )

```
[ベースイメージ] ← FROM
 ↓
[パッケージインストール] ← RUN apt install
 ↓
[アプリケーションファイル] ← COPY . .
 ↓
[最終レイヤー : 完成したイメージ]
```

## ◆ レイヤーを見るコマンド

イメージのレイヤー構造を確認するには：

```
docker history <イメージIDまたはイメージ名>
```

例：

```
docker history my-next-app
```

どのコマンドがどのレイヤーを作ったか、サイズ・日時と共に表示されます。

## ◆ レイヤーキャッシュの効果

Dockerは過去に作成したレイヤーをキャッシュします。

変更がないレイヤーは再ビルドせずに流用されるため、ビルドが圧倒的に速くなります。

## ◆ レイヤー最適化のコツ

頻繁に変わるもの ( ソースコード ) は最後にCOPYする

依存パッケージインストール ( npm install など ) はなるべく早い段階で実行する

不要ファイルはビルドに含めない ( .dockerignoreを活用 )

最適化例 :

```
悪い例 (全部まとめてコピー)
COPY . .

良い例 (package.jsonだけ先にコピーして依存インストール)
COPY package*.json ./
RUN npm install
COPY . .
```

## ▶ 1-10-6. Dockerのセキュリティについて

Dockerは非常に便利ですが、適切なセキュリティ対策をしなければ  
ホストシステムごと危険にさらすリスクがあります。

このセクションでは、Dockerを安全に運用するための基本知識を紹介します。

### ◆ なぜDockerのセキュリティが重要なのか？

- コンテナはホストカーネルを共有しているため、  
コンテナ内での攻撃がホストに影響を与えるリスクがある
- デフォルトではコンテナがほぼroot権限で動く
- 悪意あるイメージをpullすると、脆弱性を持ち込む可能性がある

### ◆ セキュリティ対策の基本

対策	説明
公式イメージや信頼できるイメージのみ使用	不明な作者のイメージを使わない
最小権限の原則	コンテナに必要最低限の権限だけを与える
Rootlessモードの利用	非rootユーザーでDockerデーモンを動かす
Dockerfileでユーザーを指定	<b>USER</b> 命令でroot以外のユーザーで実行する
ネットワークやボリュームのアクセス制御	不要なポート・共有ディレクトリを制限する

セキュリティアップデートの定期実施	ベースイメージ・パッケージを常に最新に保つ
コンテナイメージをスキャン	脆弱性スキャンツール（例：Trivy）を使う

---

#### ◆ Dockerfileで最小権限設定の例

```
FROM node:18-alpine

通常ユーザーを作成
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

作業ディレクトリ設定
WORKDIR /app

アプリをコピー
COPY . .

権限を最小化して実行
USER appuser

CMD ["node", "app.js"]
```

- USER  
appuserを使うことで、root権限ではなく一般ユーザーでアプリが動くようになります。

#### ◆ コンテナのセキュリティスキャン例（Trivy）

Trivyを使えば、イメージの脆弱性を簡単に検査できます。

```
trivy image my-next-app
```

脆弱なライブラリや設定ミスがないかチェックできます。

本番運用では必ず導入を検討する必要あり。

#### ▶ 1-10-7.

#### Dockerセキュリティ超実践編（AppArmor/SELinux/Capability制御など）

Dockerのセキュリティをさらに強化するために、Linuxカーネルレベルでコンテナの動作を制御する技術を取り入れます。

---

## ◆ AppArmor と SELinuxとは？

技術名	説明
AppArmor	ファイルパスベースでアクセス制御を行う（Ubuntu系に標準搭載）
SELinux	セキュリティコンテキストによるアクセス制御（RHEL/CentOS系に標準搭載）

これらを使うことで、万が一コンテナ内で侵害が発生しても、ホストシステムへの被害を最小化できます。

## ◆ AppArmorをDockerコンテナに適用する例

```
docker run --security-opt apparmor=docker-default t nginx
```

- docker-defaultはDocker標準の緩やかなAppArmorプロファイルです。
- 独自プロファイルを作成して適用することも可能です。

## ◆ Capability制御（細かい権限設定）

コンテナ起動時に、必要最小限の権限だけ付与することができます。

すべてのcapabilityを削除し、必要なものだけ追加する例：

```
docker run --cap-drop ALL --cap-add NET_BIND_SERVICE nginx
```

--cap-drop ALLですべて削除

--cap-add NET\_BIND\_SERVICEで、80番/443番ポートバインドだけ許可

## ☑ まとめ

カーネルレベルでの制御（AppArmor/SELinux）を使うとセキュリティが飛躍的に向上

Capability管理でコンテナのシステムアクセスを必要最小限にできる

## ▶ 1-10-8. Rootless Docker運用と最小権限実践例

通常、Dockerデーモンはroot権限で動きます。

しかし、本番環境では\*\*Rootless Docker（非root運用）\*\*を使うのが推奨されています。

## ◆ Rootless Dockerとは？

Dockerデーモンとコンテナの両方を一般ユーザー権限で動かす仕組み  
ホストシステムへの侵害リスクを大幅に低減できる

## ◆ Rootless Dockerのインストール例 ( Ubuntu )

```
sudo apt install docker-ce-rootless-extras
dockerd-rootless-setup tool . sh install
```

その後、環境変数を設定してDockerコマンドを実行できるようにします。

## ◆ 最小権限Dockerfile実践例

```
FROM node:18-alpine
```

## 通常ユーザー作成

```
RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /app

COPY . .

通常ユーザーで実行
USER appuser

CMD ["node", "server.js"]
```

USER appuserを指定することで、コンテナ内も非rootで実行されます。

## ☑ まとめ

Rootless Docker導入でホストセキュリティを大幅向上

コンテナ内部でも最小権限ユーザー運用を徹底するべき

## ▶ 1-10-9. Docker Content Trustとイメージ署名管理

イメージの改ざんやなりすましを防ぐために、Dockerには\*\*Content Trust ( DCT ) \*\*機能があります。

### ◆ Docker Content Trust ( DCT ) とは？

pullやpushを行うイメージにデジタル署名を付与/検証する仕組み

改ざんされたイメージや不正なイメージを防止できる

### ◆ DCTを有効化する方法

DCTを有効にするには、環境変数を設定するだけです。

```
export DOCKER_CONTENT_TRUST=1
```

その状態で、pullやpushを行うと、署名チェックが行われます。

### ◆ 注意点

署名未対応のリポジトリ ( 例：一部プライベートレジストリ ) ではエラーになる

本番運用では、DCT対応レジストリ ( Docker Hub, Harborなど ) を利用推奨

### ☑ まとめ

Content Trustでイメージ配布時の信頼性を確保

本番環境ではDCT有効化を必須にすべき

pull/push前に環境変数設定を忘れずに！

---

## 2-0 Docker コマンド一覧

実演パートに入る前にdockerのコマンドを確認します

### イメージ関連コマンド

コマンド	説明
<code>docker pull &lt;イメージ名&gt;</code>	イメージをリポジトリから取得
<code>docker build -t &lt;イメージ名&gt; .</code>	Dockerfileからイメージをビルド
<code>docker images</code>	ローカルにあるイメージ一覧を表示
<code>docker rmi &lt;イメージID&gt;</code>	イメージを削除

<code>docker tag &lt;元イメージ&gt; &lt;新しいタグ&gt;</code>	イメージに新しいタグを付ける
-----------------------------------------------------	----------------

## □ コンテナ操作コマンド

コマンド	説明
<code>docker run -it &lt;イメージ名&gt;</code>	コンテナを起動 ( 対話モード )
<code>docker run -d &lt;イメージ名&gt;</code>	コンテナをバックグラウンド起動
<code>docker ps</code>	起動中のコンテナ一覧を表示
<code>docker ps -a</code>	すべてのコンテナを表示
<code>docker start &lt;コンテナID&gt;</code>	コンテナを開始
<code>docker stop &lt;コンテナID&gt;</code>	コンテナを停止
<code>docker restart &lt;コンテナID&gt;</code>	コンテナを再起動
<code>docker rm &lt;コンテナID&gt;</code>	コンテナを削除
<code>docker exec -it &lt;コンテナID&gt; /bin/sh</code>	コンテナ内でシェルを起動
<code>docker logs &lt;コンテナID&gt;</code>	コンテナのログを表示

## ネットワーク関連コマンド

コマンド	説明
<code>docker network ls</code>	ネットワーク一覧を表示
<code>docker network create &lt;ネットワーク名&gt;</code>	新しいネットワークを作成
<code>docker network inspect &lt;ネットワーク名&gt;</code>	ネットワークの詳細を表示
<code>docker network rm &lt;ネットワーク名&gt;</code>	ネットワークを削除

## ボリューム関連コマンド

コマンド	説明
<code>docker volume ls</code>	ボリューム一覧を表示
<code>docker volume create &lt;ボリューム名&gt;</code>	新しいボリュームを作成
<code>docker volume inspect &lt;ボリューム名&gt;</code>	ボリュームの詳細を表示
<code>docker volume rm &lt;ボリューム名&gt;</code>	ボリュームを削除

## クリーンアップ系コマンド

コマンド	説明
<code>docker system prune</code>	未使用のコンテナ・イメージ・ネットワークを一括削除
<code>docker container prune</code>	停止中のコンテナを削除
<code>docker image prune</code>	未使用イメージを削除
<code>docker volume prune</code>	未使用ボリュームを削除
<code>docker network prune</code>	未使用ネットワークを削除

## まとめ

- **イメージ関連** : pull, build, images, rmi, tag
- **コンテナ関連** : run, ps, start, stop, rm, exec, logs
- **ネットワーク関連** : network create/ls/inspect/rm
- **ボリューム関連** : volume create/ls/inspect/rm
- **クリーンアップ** : system prune など一括掃除！



## 2 章 Docker 実演

### 🎯 実演の目的

- macOS上でNext.jsアプリをDocker化して起動
  - DebianサーバーにDockerでデプロイ
  - クライアントとサーバーの環境差を吸収できることを確認
- 

### 2-1. 前提環境

#### クライアント ( macOS )

- Docker Desktop インストール済み
- VSCode推奨

#### サーバー ( Debian )

- Docker + Docker Compose インストール済み
  - SSH接続できる状態
- 

### 2-2. プロジェクト作成 ( macOS上 )

```
npx create-next-app my-next-app
cd my-next-app
```

### 2-3. Dockerfileの作成

Next.jsアプリをDockerで動かすには、まず **Dockerfile** を用意します。

このファイルでは、アプリをどういう環境で、どんな手順でセットアップし、どのコマンドで起動するかを記述します。

以下がDockerfile

```
Node.jsの軽量版イメージをベースにする (alpineは最小構成)
FROM node:18-alpine
```

```
アプリケーションを配置するディレクトリを作成
WORKDIR /app

パッケージ定義ファイルをコピー
COPY package*.json ./

依存関係をインストール
RUN npm install

残りのアプリケーションのコードを全てコピー
COPY . .

アプリが使用するポート (Next.jsのデフォルトは3000)
EXPOSE 3000

アプリを開発モードで起動
CMD ["npm", "run", "dev"]
```

## 2-4 dockerignoreの作成

.dockerignore

ファイルは、Dockerに「コピーしないでいいファイル」を指示するためのファイルです。

特に開発中は node\_modules や .next

ディレクトリなど、イメージ内に不要な一時ファイルを除外してビルドを軽量に保ちます。

## 2-5. Dockerでビルド&実行 ( ローカル : macOS )

作成したDockerfileを使って、ローカルでNext.jsアプリをDockerイメージにビルドし、実行します。

### イメージのビルド

```
docker build -t my-next-app .
```

### コンテナの起動

```
docker run -p 3000:3000 my-next-app
```

-p 3000:3000 : ホストの3000番ポートを、コンテナの3000番ポートにマッピングします。

ブラウザで以下にアクセスして確認 :

<http://localhost:3000>

## 2-6. Docker Composeの導入

Docker

Composeを使うと、複数の設定（ビルド・ポート・ボリューム・環境変数など）をdocker-compose.ymlファイルで一括管理できる。

docker-compose.yml の作成

```
version: "3.8"

services:
 web:
 build: .
 ports:
 - "3000:3000"
 volumes:
 - .:/app
 - /app/node_modules
 environment:
 - NODE_ENV=development
```

docker-composeの起動

```
docker compose up
```

## 2-7 サーバーへのデプロイ

ローカルで作成したプロジェクトを、Debianサーバーに転送してデプロイします。

```
scp -r ./my-next-app username@<サーバーIP>:/home/username/
```

注:SCPコマンドはSecureShellプロトコルを利用してデータを転送するコマンドです。(-r:ディレクトリを再帰的にコピー)(username@<サーバーIP>:ログインユーザーとサーバーIPを指定)

## 2-8 サーバー上でDocker Composeによる起動

SSHでサーバーにログイン

```
ssh username@<サーバーIP>
```

プロジェクトフォルダに移動し、起動

```
cd /home/username/my-next-app
docker compose up -d
```

## 2-9 外部からの動作確認

ブラウザで次のURLにアクセスして、サーバー上のNext.jsアプリが動作しているか確認します

- `http://<サーバーIP>:3000`

## 2-10. 本番用Dockerfileの書き換え ( 任意 )

開発モード ( `npm run dev` ) では、

- 自動リロード
  - 開発者向けログ出力
- など、開発に便利な機能が有効になっています。

しかし本番環境では、

- 高速な応答
  - セキュリティ向上
  - 安定稼働
- を重視するため、ビルド済みのコードを「本番モード」で起動する必要があります。

そのため、本番向けにDockerfileをマルチステージビルド形式に書き換えます。

---

### 🎯 本番用Dockerfile ( 完成形 )

```
--- ビルドステージ ---
FROM node:18-alpine AS builder

WORKDIR /app

package.jsonとpackage-lock.jsonだけ先にコピーしてキャッシュ活用
COPY package*.json ./

依存関係をインストール
RUN npm install

アプリ本体をコピー
COPY . .

Next.jsのビルド (静的ファイルや最適化ファイルを生成)
RUN npm run build
```

```
--- 実行ステージ (軽量) ---
FROM node:18-alpine

WORKDIR /app

builderステージからビルド成果物をコピー
COPY --from=builder /app ./

devDependenciesをインストールせずに、本番用依存関係だけにする
RUN npm install --omit=dev

環境変数で本番モードを指定
ENV NODE_ENV=production

Next.jsアプリが使用するポート
EXPOSE 3000

アプリを本番モードで起動
CMD ["npm", "start"]
```

## それぞれのステップ解説

### ① FROM node:18-alpine AS builder

ビルド専用のイメージを作成します。

AS builderと名前を付けて後で参照できるようにしています。

### ② WORKDIR /app

/appディレクトリを作業ディレクトリに設定します。

以降の操作はすべてこの中で行われます。

### ③ COPY package\*.json ./

最初に package.json と package-lock.json だけをコピーします。

これにより、依存関係のインストールをキャッシュでき、ビルド時間を短縮できます。

### ④ RUN npm install

アプリの依存パッケージをインストールします。

#### ⑤ COPY ..

ソースコード全体をコピーします。

この順番にすることで、もしコードだけが変更された場合でも、依存関係のインストールを再実行せずに済みます。

#### ⑥ RUN npm run build

Next.jsアプリをビルドします。

静的ファイル ( HTML/CSS/JSなど ) や最適化ファイルが .next/ フォルダに生成されます。

#### ⑦ 本番用実行ステージに切り替え

軽量化のため、ビルドとは別のクリーンなコンテナイメージを使います。

#### ⑧ COPY --from=builder /app ./

ビルド成果物 ( アプリ全体 ) をコピーします。

開発時にしか使わない一時ファイルなどは含まれません。

#### ⑨ RUN npm install --omit=dev

本番環境に不要な devDependencies

をインストールしないことで、イメージをさらに軽量化します。

#### ⑩ ENV NODE\_ENV=production

環境変数で「本番モード」を明示します。

本番用にNext.jsやNode.jsの挙動が最適化されます。

#### ⑪ EXPOSE 3000

Next.jsはデフォルトで3000番ポートを使用します。

これによりDockerが外部公開するポートを認識します。

#### ⑫ CMD ["npm", "start"]

npm startはnext startを呼び出し、ビルド済みアプリを本番モードで起動します。

## 2-11 上級者向け NGINXも一緒に起動してみる

次に

- Next.js ( 本番モード )

Nginx ( リバースプロキシ + 静的配信 )

-

を一緒に起動する流れをカバーします。

(複数のアプリを起動することで本番に近い環境でdockerを利用できます。)

## 🎯 ゴール

- Next.js本番用アプリをDockerコンテナで動かす
- その前段にNginxを置き、リバースプロキシとしてHTTPS/ポート番号非表示対応を可能にする
- **複数サービスをDocker Composeでまとめて起動・管理する**

## 📁 プロジェクト構成例

```
my-next-app/
├── Dockerfile
├── docker-compose.yml
├── nginx/
│ └── default.conf
├── package.json
├── next.config.js
└── その他のNext.jsファイル
```

## ①.NEXT.js用のdockerfile作成

```
--- ビルドフェーズ ---
FROM node:18-alpine AS builder

WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

--- 実行フェーズ ---
FROM node:18-alpine

WORKDIR /app
COPY --from=builder /app ./
RUN npm install --omit=dev

ENV NODE_ENV=production
EXPOSE 3000
CMD ["npm", "start"]
```

## ② Nginx設定ファイル ( リバースプロキシ設定 )

ファイル名 : nginx/default.conf

```
server {
 listen 80;

 server_name _;

 location / {
 proxy_pass http://web:3000; #nextのポート番号
 proxy_http_version 1.1;
 proxy_set_header Upgrade $http_upgrade;
 proxy_set_header Connection 'upgrade';
 proxy_set_header Host $host;
 proxy_cache_bypass $http_upgrade;
 }
}
```

注:コンテナ間通信なので、localhostではなくサービス名webを指定します。

## ③ docker-compose.ymlの作成

```
services:
 web:
 build: .
 container_name: nextjs_app
 expose:
 - "3000" # 内部だけで公開する
 environment:
 - NODE_ENV=production

 nginx:
 image: nginx:latest
 container_name: nginx_proxy
 ports:
 - "80:80"
 volumes:
 - ./nginx/default.conf:/etc/nginx/conf.d/default.conf
 depends_on:
 - web
```

## ④ 起動方法

プロジェクトディレクトリで実行 :

```
docker compose up -d
```

これでWEBアプリが配信できる。



## 3章 まとめ

### 3-1. 本日のゴールは達成できましたか？

以下のポイントを自分の言葉で説明できれば 合格 です。

- Docker が解決する "動く環境格差" の課題と、仮想マシンとの本質的な違い
- イメージ → コンテナの流れ / Dockerfile の役割
- 開発 → テスト → 本番へ続く CI/CD パイプラインにおける Docker の価値

### 3-2. キーコンセプト早見表

トピック	キーワード	30 秒で思い出すフレーズ
コンテナ	<code>docker run</code>	"速い・軽い・手軽" な実行単位
イメージ	<code>docker build</code>	アプリ + 環境を 1 ファイルで配布
レイヤー (上級)	<code>docker history</code>	変更がなければキャッシュで爆速
ネットワーク (上級)	<code>bridge / host / overlay</code>	コンテナ間通信の土台
ボリューム (上級)	<code>docker volume</code>	データ永続化の解答
セキュリティ (上級)	<code>Rootless / Trivy</code>	"ホストは守る、脆弱性は出さない"
オーケストレーション	Kubernetes	"100 店舗を 1 つのダッシュボードで"

### 3-3. ハンズオン復習タスク

#### 1. ローカル

1. githubから教材を取得
2. `docker compose up` で Next.js を起動
3. ブラウザで `http://localhost:3000` を確認

#### 2. 本番想定

1. `.env.production` を編集して機密値を設定
  2. `docker build -t my-next-app:prod .`
  3. `docker run -d -p 80:3000 my-next-app:prod`
3. セキュリティ強化 ( 上級 )
1. `export DOCKER_CONTENT_TRUST=1`
  2. `trivy image my-next-app:prod` で脆弱性スキャン

### 3-4. つまづきポイント & 解決策



症状	よくある原因	ワンポイント解決
コンテナが起動しない	ポート競合 / ENV ミス / イメージタグ違い	<code>docker logs &lt;id&gt;</code> → 原因特定 & ポート / ENV 修正
コンテナがすぐ終了する	CMD・ENTRYPOINT の typo / アプリ側エラー	<code>docker inspect --format '{{.State.ExitCode}}' &lt;id&gt;</code> + エラーログ確認
イメージが巨大	キャッシュ無効化 / 不要ファイル COPY	マルチステージ & <code>dockerignore</code> で削減
ビルドが遅い	レイヤーキャッシュ無効 / ネットワーク遅延	依存パッケージ COPY を先頭に + 社内ミラー活用
ポートが外に出ない	EXPOSE 忘れ / <code>-p</code> マッピング漏れ	<code>docker ps --format '{{.Ports}}'</code> で確認し再 run
権限エラー (EACCES)	root 運用 / volume マウント時の UID 不一致	Rootless + <code>USER</code> 指定 or <code>--user \$(id -u)</code>
ネットワークで通信不可	bridge と host の混在 / firewall	<code>docker network inspect</code> で IP 確認 + <code>--network</code> 指定
volume データが消えた	匿名ボリューム / bind mount パス違い	名前付き volume を使い <code>docker volume ls</code> で管理
DNS 解決できない	<code>/etc/resolv.conf</code> 上書き /	<code>--dns 8.8.8.8</code> か Docker Desktop の

	corporate proxy	DNS 設定変更
<code>permission denied on docker.sock</code>	非 docker グループ / Rootless 未設定	<code>sudo usermod -aG docker \$USER</code> 後再ログイン
Pull が遅い / 失敗	レジストリ障害 / 帯域制限	ミラー (例 AWS ECR, GHCR) 使用 + <code>--platform</code> 明示
ログが多すぎて追えない	無限ループ出力 / json-file ドライバ膨張	<code>docker logs --tail 200 -f &lt;id&gt;</code> + logrotate or Loki



### 3-5. 次なる一歩

- CI/CD 実践 : GitHub Actions で `docker buildx & push` を自動化
- Kubernetes 入門 : minikube でローカルからクラスタ体験
- 監視・ロギング : Prometheus + Grafana, Loki で可 observability
- セキュリティ深掘り : AppArmor/SELinux プロファイルを自作してみる

### おすすめ資料集

カテゴリ	資料名 & リンク ( 公式 / 主 要 )	ポイント	こんな時に役立つ
 公式ドキュメント	Docker Docs (docs.docker.com)	最新仕様・CLI/BP解説が最速 で反映	コマンドやオプションを正 確に調べたい
	Docker Compose File v3-v4 リファレンス	service/volume/network など構文詳細	Compose の書き方を迷ったとき
	BuildKit Best Practices	<code>--target</code> やキャッシュインポート等の 高速化術	ビルド時間を縮めたい / CI を高速化
 書籍 ( 和・ 洋 )	『Docker/Kubernete s 開発・運用実践ガ	日本語で CI/CD, セキュリティ, k8s まで網羅	全体像を体系的に学びたい

	イド 第2版』(技術評論社)		
	"Docker Deep Dive" 4th Ed. (Nigel Poulton)	イメージ・ネットワーク内部 の挙動まで図解	仕組みをしっかりと腹落ちさせたい
	"Kubernetes Patterns" (O'Reilly)	マニフェスト設計パターンを ケース別に整理	コンテナ Orchestration に踏み出す前
 ハンズオン	Play with Docker (labs.play-with-docker.com)	ブラウザのみで 4h 無料環境	インストール不要で即実験
	Katacoda "Docker for Developers"	シナリオ型で段階学習	初学者が手を動かしながら 覚える
	Docker Labs (github.com/docker/labs)	マルチステージ・Swarm 等の実践例	現場レベルのサンプルを探す
 動画 & コース	CNCF / Docker YouTube	公式チュートリアル・会議講演が無償	最新トレンドや実運用事例を追う
	Udemy 「Docker & Kubernetes: The Practical Guide」 (2024)	ハンズオン + 図解 30 h 超	フルコースで流れを一気に押さえない
 セキュリティ	Docker Bench for Security	自動スクリプトでベンチマーク実施	本番運用前のセルフチェック
	Aqua Trivy (github.com/aquasecurity/trivy)	イメージ脆弱性・SBOM 生成	CI 組込で脆弱性を可視化
	OWASP "Docker Top-10" Cheat Sheet	攻撃パターン別に対策	ポリシー策定・レビュー時
 CI/CD & 実運用	GitHub Actions – official Docker actions	<a href="#">build-push-action</a> でマルチプラットフォーム対応	GitHub Flows にコンテナビルドを組む

	GitLab CI/CD Templates (Docker)	キャッシュ共有例・DCT 有効化例	GitLab runner × Docker の最短パス
	Jenkins “Docker Pipeline Plugin” Docs	declarative で Build/Push/Deploy 定義	既存 Jenkins を生かしたい
 デバッグ & 視覚化	dive (github.com/wagood man/dive)	イメージレイヤーの容量・変 更点を可視化	不要ファイル削減・最適化
	cTop / docker stats	リアルタイムで CPU・メモリ監視	負荷調査・ボトルネック確 認
 コミュニテ ィ	Docker Community Slack #jp	日本語でコア開発者に質問可	ハマりどころを素早く相談
	Japan Container SIG (connpass)	月例勉強会 / 発表資料アーカ イブ	国内事例・最新アップデー ト共有

### 3-6 お疲れさまでした！

今日得た知識を 明日からの開発環境改善 にぜひ役立ててください。