成果物のデプロイ

TypeScriptで作ったアプリケーションの開発環境の作り方を、バリエーションごとに紹介してきました。それぞれの環境でビルド方法についても紹介しました。本章ではデプロイについて紹介します。

> **課題**
>
> npmパッケージ to npm npmパッケージ to nexus
>
> https://qiita.com/kannkyo/items/5195069c65350b60edd9

npmパッケージとしてデプロイ

この方法でデプロイする対象は以下の通りです。

- Node.js用のライブラリ
- Node.js用のCLIツール
- Node.js用のウェブサービス

ビルドしたらアーカイブファイルを作ってみましょう。これで、package.tgzファイルができます。npm installにこのファイルのパスを渡すとインストールできます。アップロードする前に、サンプルのプロジェクトを作ってみて、このパッケージをインストールしてみて、必要なファイルが抜けていないか、必要な依存パッケージが足りているかといったことを確認してみましょう。また、展開してみて、余計なファイルが含まれていないことも確認すると良いでしょう。

```
$ npm pack
```

- ビルドには必要だが、配布する必要のないファイルが含まれている

  `.npmignore`ファイルにそのファイル名を列挙します。パッケージを作る時に無視されます。

- ビルドには必要だが、利用環境でインストール不要なパッケージがある

  `package.json`の`dependencies`から、`devDependencies`に移動します。

npmのサイトにアップロードしてみましょう。

パッケージリポジトリはnpm以外にもあります。例えば、Nexusを使えばローカルにパッケージリポジトリが建てられます。GitHubもパッケージリポジトリを提供しています。

サーバーにアプリケーションをデプロイ

Node.jsはシングルコアを効率よく使う処理系です。サーバーアプリケーションでマルチコアを効率よく使うには、プロセスマネージャを利用します。本書ではpm2を利用します。

- https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/

サンプルとしては次のコードを使います。

src/main.ts

```
import express, { Request, Response } from "express";
import compression from "compression";
import bodyParser from "body-parser";
import gracefulShutdown from "http-graceful-shutdown";

const app = express();
app.use(compression());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.get("/", (req: Request, res: Response) => {
    res.json({
        message: `hello ${req.headers["user-agent"]}`,
    });
});

const host = process.env.HOST || "0.0.0.0";
const port = process.env.PORT || 3000;

const server = app.listen(port, () => {
    console.log("Server is running at http://%s:%d", host, port);
    console.log("  Press CTRL-C to stop\n");
});

gracefulShutdown(server, {
    signals: "SIGINT SIGTERM",
    timeout: 30000,
    development: false,
    onShutdown: async (signal: string) => {
        console.log("... called signal: " + signal);
        console.log("... in cleanup");
        // shutdown DB or something
    },
    finally: () => {
        console.log("Server gracefulls shutted down.....");
    },
});
```

`tsconfig.json`は`npx tsc --init`で生成したものをひとまず使います。`package.json`は以下のものを利用します。nccを使ってビルドする前提となっています。

package.json

```
{
  "name": "webserver",
  "version": "1.0.0",
  "scripts": {
    "build": "ncc build src/main.ts"
  },
  "author": "Yoshiki Shibukawa",
  "license": "ISC",
  "dependencies": {
    "pm2": "^4.4.0"
  },
  "devDependencies": {
    "@types/body-parser": "^1.19.0",
    "@types/compression": "^1.7.0",
    "@types/express": "^4.17.7",
    "@zeit/ncc": "^0.22.3",
    "body-parser": "^1.19.0",
    "compression": "^1.7.4",
    "express": "^4.17.1",
    "http-graceful-shutdown": "^2.3.2",
    "typescript": "^3.9.7"
  }
}
```

nccでビルドすると、dist/index.js``という一つの.jsファイルが生成されます。実行は`node dist/index.js`の変わりに次のコマンドを利用します。これで、CPUコア数分Node.jsのインスタンスを起動し、クラスタで動作します。

```
pm2 start dist/index.js -i max
```

pm2で起動すると、デーモン化されてアプリケーションが起動します。`pm2 logs`コマンドでログをみたり、`pm2 status`や`pm2 list`で起動しているプロセスの状態を知ることができます。

デーモン化させないでフォアグラウンドで動作させる場合は`--no-daemon`をつけて起動します。

Dockerイメージの作成

この方法でデプロイする対象は以下の通りです。

- Node.js用のCLIツール
- Node.js用のウェブサービス
- ウェブフロントエンド

なお、本章のサンプルはベースイメージのバージョンは細かく指定していませんが、突然メジャーバージョンが上がってビルドできなくなることもあります。特に業務開発ではDocker Hubのイメージ情報のタグを見て、適宜バージョンを固定することをお勧めします。

コンテナとは何か

コンテナは1アプリケーションだけが格納されたミニOS環境です。Linux上でもmacOS上でもWindows上でもクラウド環境でも、アプリケーションからは同じOS環境のように見えます。ポータブルなアプリケーション配布・実行環境としてますます地位が高まっています。コンテナは動いている環境のことを指します。コンテナは実行時にイメージを読み込んで環境を構築します。これは実行に必要なファイルと起動時のコマンドなどがセットになったものです。開発者が作るのはイメージです。

コンテナ関係のシステムは、実行のランタイムやビルド方法など、それぞれにいくつか選択肢がありますが、開発時の環境として一番ポピュラーなのがDockerです。本書ではコンテナ=Dockerコンテナとして説明をします。

ローカルではコンテナはDocker for Desktopを使ってイメージの作成や動作のテストができます。運用環境として、どのクラウド事業者もKubernetesを使ってコンテナベースで本番運用環境の維持管理できるサービスを提供しています。1つのノードにリソースが許すかぎり多数のコンテナを配置することができ、実行時の効率も上がります。 それ以外にも、AWS ECSやAWS Fargate、GCP Cloud Runなど、単体のDockerイメージやDockerイメージ群を起動できるサービスもあります。コンテナはウェブアプリケーションのような起動し続けるサービスにも使えますし、一度実行して終了するバッチ処理にも活用できます。

Dockerコンテナ内のアプリケーションは外部の環境と切り離されて実行されますが、Dockerの実行時のオプションで外界と接点を設定できます。複雑な設定が必要なアプリケーションの場合は、設定ファイルをコンテナ内の特定のパスに置くこともできますが、推奨されるのは環境変数のみによって制御されるアプリケーションです。

- 環境変数
- ネットワークの設定
  - 特定のポートをlocalhostに公開
  - localhostとコンテナ内部のを同一ネットワークにするかどうか
- ファイルやフォルダのマウント
- 最後に実行するコマンドのオプション

コンテナは上記のように、クラウドサービスに直接デプロイして実行できます。

複数のコンテナに必要な設定を与えてまとめて起動するツール（コンテナオーケストレーションツール）もあります。それがdocker-composeやKubernetesです。

Dockerのベースイメージの選択

Dockerイメージを作成するには`Dockerfile`という設定ファイルを作成し、`docker build`コマンドを使ってイメージを作成します。ベースイメージと呼ばれる土台となるイメージを選択して、それに対して必要なファイルを追加します。ビルド済みのアプリケーションを単に置く、という構築

方法もありますが（公式イメージの多くはそれに近いことをしている）、アプリケーション開発の場合はソースコードをDocker内部に送り、それをDocker内部でビルドして、実行用イメージを作成します。できあがったイメージをコンパクトにするために、ビルド用イメージと、実行用イメージを分ける（マルチステージビルド）が今の主流です。

Node.jsの公式のイメージは以下のサイトにあります。

- https://hub.docker.com/_/node/

バージョンと、OSの組み合わせだけイメージがあります。その中でおすすめの組み合わせが次の3つです。

用途	バリエーション	ビルド用イメージ	実行用イメージ
Node.js(CLI/ウェブアプリ)	鉄板	nodeのDebian系	nodeのDebian-slim系
Node.js(CLI/ウェブアプリ)	ネイティブ拡張なし	nodeのDebian-slim系	nodeのDebian-slim系
Node.js(CLI/ウェブアプリ)	セキュリティ重視	nodeのDebian-slim系	distrolessのnode.js
ウェブフロントエンド配信		nodeのDebian-slim系	nginx:alpine

課題

Denoはこちらのスレッドを見守る https://github.com/denoland/deno/issues/3356

DebianはLinuxディストリビューションの名前です。buster (Debian 10)、stretch (Debian 9)、jessie (Debian 8)が執筆時点ではコンテナリポジトリにあります。それぞれ、無印がフル版で、gccや各種開発用ライブラリを含みます。いろいろ入っていて便利ですが、イメージサイズは大きめです。slimがつくバージョンがそれぞれにあります。これはNode.jsは入っているが、gccなどがないバージョンです。例えば、最新LTS（執筆時点で12）のDebianの開発環境込みのイメージであれば、`node:12-buster`を選びます。

もう一つ、GCPのコンテナレジストリで提供されているのがdistrolessです。こちらはシェルもなく、セキュリティパッチも積極的に当てていくという、セキュリティにフォーカスしたDebianベースのイメージです。シェルがないということはリモートログインができませんので、踏み台にされる心配がないイメージです。これはGCPのコンテナレジストリに登録されており、`gcr.io/distroless/nodejs`という名前で利用可能です。

Alpineというサイズ重視のOSイメージはありますが、あとから追加インストールしなければならないパッケージが増えがちなのと、パッケージのバージョン固定がしにくい（古いパッケージが削除されてしまってインストールできなくなる）などの問題がありますし、他のイメージがだいたいDebianベースなので、Debianベースのもので揃えておいた方がトラブルは少ないでしょう。

Dockerイメージはサイズが重視されますが、ビルド時間や再ビルド時間も大切な要素です。開発ツールなしのイメージ（slimやalpineなど）を選び、必要な開発ツールだけをダウンロードするのはサイズの上では有利ですが、すでにできあがったイメージをただダウンロードするのよりも、依存関係を計算しながら各パッケージをダウンロードする方が時間がかかります。

CLI/ウェブアプリケーションのイメージ作成

CLIとウェブアプリケーションの場合の手順はあまり変わらないので一緒に説明します。ベースイメージの選択では3種類の組み合わせがありました。

- C拡張あり（Debian系でビルド）
- C拡張なし（Debian-slim系でビルド）
- セキュリティ重視（destrolessに配信）

前2つ目はベースイメージが `node:12-buster` から `node:12-buster-slim` に変わるだけですので、まとめて紹介します。

なお、Node.jsはシングルコアで動作する処理系ですので、マルチコアを生かしたい場合はインスタンスを複数起動し、ロードバランスをする仕組みを外部に起動する必要があります。

Debianベースのイメージ作成とDockerの基礎

まず、イメージにするアプリケーションを作成します。コンテナの中のアプリケーションは終了時にシグナルが呼ばれますので、シグナルに応答して終了するように実装する必要があります。

`Dockerfile` はコンテナのイメージを作成するためのレシピです。行志向のスクリプトになっています。WindowsやmacOSでは、Linuxが動作している仮想PCの中でDockerのサーバーが動作しており（Windowsの場合はWindowsも動作しますが、ここでは無視します）、ビルドを実行すると、実行されているフォルダの配下のファイル（コンテキスト）と `Dockerfile` が、まとめてサーバーに送られます。サーバーの中で、 `Dockerfile` に書かれた命令に従ってベースとなるディスクイメージに手を加えていきます。実行されるのはローカルコンピュータではないので、記述できるコマンドも、そのベースイメージのLinuxで使えるものに限られます。

そのため、 `Dockerfile` に問題があれば、Windows上でビルドや実行をしても、macOS上でビルドや実行をしても、まったく同じエラーが発生するはずです。バージョン番号や内部で使われるパッケージのバージョンはその時の最新を使うこともできるため、いつでもまったく同じにはならないかもしれませんが、どこで誰がどのOSで実行しても、同じイメージが作られます。この冪等性がDockerが重宝されるポイントです。

次のファイルが、実際に動作する `Dockerfile` です。順を追って見ていきます。

Dockerfile

```
# ここから下がビルド用イメージ

FROM node:12-buster AS builder

WORKDIR app
COPY package.json package-lock.json ./
RUN npm ci
COPY tsconfig.json ./
COPY src ./src
RUN npm run build

# ここから下が実行用イメージ

FROM node:12-buster-slim AS runner
WORKDIR /opt/app
COPY --from=builder /app/dist ./
USER node
EXPOSE 3000
CMD ["node", "/opt/app/index.js"]
```

`FROM`はベースとなるイメージを選択する命令です。ここではビルド用のベースイメージと、実行用のベースイメージと2箇所`FROM`を使用しています。`FROM`から次の`FROM`、あるいはファイルの終行までがイメージになります。ここでは2つイメージが作られていますが、最後のイメージが、このDockerfileの成果物のイメージとなります。わざわざ2つに分けるのは、アクロバティックなことをしないで最終的なイメージサイズを小さくするためです。

それぞれのイメージの中ではいくつかの命令を使ってイメージを完成させていきます。

`COPY`は実行場所（コンテキスト）や別のイメージ（`--from`が必要）からファイルを取得してきて、イメージ内部に配置する命令です。このサンプルでは使っていませんが、`ADD`命令もあり、こちらは`COPY`の高性能バージョンです。ネットワーク越しにファイルを取得できますし、アーカイブファイルを展開してフォルダに配置もできます。`RUN`は何かしらの命令を実行します。

重要なポイントが、このイメージ作成のステップ（行）ごとに内部的にはイメージが作成されている点です。このステップごとのファイルシステムの状態は「レイヤー」と呼ばれます。このレイヤーはキャッシュされて、コンテキストとファイルシステムの状態に差分がなければキャッシュを利用します。イメージの内部にはこのレイヤーがすべて保存されています。実行用イメージはこのレイヤーとサイズの問題は心の片隅に置いておく方が良いですが（優先度としては10番目ぐらいです）、ビルド用のイメージはサイズが大きくなっても弊害とかはないので、なるべくステップを分けてキャッシュされるようにすべきです。また、キャッシュ効率をあげるために、なるべく変更が少ない大物を先にインストールすることが大切です。

上記のサンプルではパッケージ情報ファイル（`package.json`と`package-lock.json`）取得してきてサードパーティのライブラリのダウンロード（`npm install`）だけを先に実行しています。利用パッケージの変更はソースコードの変更よりもレアケースです。一方、ソースコードの変更は大量に行われます。そのためにソースコードのコピーを後に行っています。もし逆であれば、ソースコ

ード変更のたびにパッケージのダウンロードが走り、キャッシュがほとんど有効になりません。このようにすれば、ソースコードを変更して再ビルドするときは`COPY src`の行より以前はスキップされてそこから先だけが実行されます。

実行用イメージの最後は`CMD`命令を使います。シェルスクリプトで実行したいプログラムを記述するのと同じように記述すれば問題ありません。最後の`CMD`は、常時起動しつづけるウェブサーバーであっても、一通り処理を実行して終了するバッチコマンドであっても、どちらでもかまいません。

注釈

Dockerとイメージサイズ削減

今回紹介している、ビルド用イメージと実行用イメージなど、複数イメージを利用してイメージを作成する方法は「マルチステージビルド」と呼ばれます。

すべてのステップがレイヤーとして保存されると紹介しました。例えば、ファイルを追加、そして削除をそれぞれ1ステップずつ実行すると、消したはずのファイルもレイヤーには残ってしまい、イメージサイズは大きなままとなります。

マルチステージビルドがなかった時代は、なるべくレイヤーを作らないことでイメージサイズを減らそうとしていました。例えば、C++コンパイラをパッケージマネージャを使ってインストールし、ビルドを実行し、コンパイラを削除するというところまで、一つのRUNコマンドで行うこともありました。これであれば、結果のレイヤーは一つですし、ビルド済みのバイナリだけが格納されるのでサイズは増えません。次のコードはRedisの実際のリポジトリから取得してきたものです。Redis 3.0当時のもので、今はもっと複雑ですが、この場合は試行錯誤すると、毎回パッケージのダウンロードが始まります。もはやこのようなことは不要です。

```
RUN buildDeps='gcc libc6-dev make' \
    && set -x \
    && apt-get update && apt-get install -y $buildDeps --no-install-recommends \
    && rm -rf /var/lib/apt/lists/* \
    && mkdir -p /usr/src/redis \
    && curl -sSL "$REDIS_DOWNLOAD_URL" -o redis.tar.gz \
    && echo "$REDIS_DOWNLOAD_SHA1 *redis.tar.gz" | sha1sum -c - \
    && tar -xzf redis.tar.gz -C /usr/src/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
    && rm -r /usr/src/redis \
    && apt-get purge -y --auto-remove $buildDeps
```

注釈

ビルド時間を確実に短くするテクニック

最初に、ローカルのファイルを一式Dockerのサーバーに送信すると説明しました。`.dockerignore`ファイルがあれば、そのサーバーに送るファイルを減らし、ビルドが始まるまでの時間が短縮されます。また、余計なファイルが変更されることでキャッシュが破棄されることを減らします。

明らかに巨大になるのが次の2ファイルです。この2つは最低限列挙しましょう。`.git`を指定しないと、ブランチを切り替えただけで再ビルドになります。

.dockerignore

```
node_modules
.git
```

Dockerイメージのビルドと実行

Dockerfileができたらイメージをビルドして実行してみましょう。名前をつけなくても、最後に作成したものなので実行直後は迷子にはならないのですが、何かしら名前をつけておく方が何かと良いです。ハッシュな識別子はかならず生成されるので、削除や実行にはこの識別子も使えます。

```
# ビルド。作成したイメージにwebserverと名前をつける
$ docker build -t [イメージ名] .

# イメージ一覧
$ docker images

# イメージ削除
$ docker rmi [イメージ名]
```

実行は次の通りです。

```
# 実行
$ docker run -it --rm --name [コンテナ名] -p "3000:3000" [イメージ名]

# 実行中コンテナの表示（-aをつけると停止中のものも表示）
$ docker ps -a

# コンテナ停止（-dで実行していなければ、Ctrl+Cで停止可能）
$ docker stop [コンテナ名]

# 停止済みコンテナの削除
$ docker rm [コンテナ名]
```

説明によってはデーモン化のための`-d`オプションを紹介しているものもありますが、ログを見たりもあると思いますし、簡単に停止ができるようにこのオプションは付けずに実行する方が便利です。代わりに、実行しているターミナルに接続して情報を出力するために`-it`（ハイフンは一つ）を付けます。また、Dockerを停止すると、停止状態になり、その後削除は`docker rm`コマンドを実行しなければなりませんが、`--rm`をつけると、停止時に削除まで行います。`-p`はあけたいポート番号です。左がホスト、右がDocker内部のプロセスのポートです。今回は同じなので、*-p "3000:3000"*を指定します。もし、コンテナ内部が80であれば、*-p "3000:80"*になります。

distrolessベースのDockerイメージの作成

distrolessはシェルが入っておらず、外部からログインされることもなく安全というGoogle製のDockerイメージです。標準Linuxに入っているようなツールも含めて、最小限にカットされています。Node.js、Java、Python、.netなど言語のランタイムだけが入ったバージョン、llvmベースのコンパイラで作成したコードを動かすだけのバージョン、何もないバージョンなど、いくつかのバリエーションが用意されています。今回はNode.jsを使います。

現在、8種類タグが定義されています。latestはLTSが終わるまでは10のままです。`debug`がついているものはデバッグ用のシェルが内蔵されています。

- `gcr.io/distroless/nodejs:latest`: 10と同じ
- `gcr.io/distroless/nodejs:10`
- `gcr.io/distroless/nodejs:12`
- `gcr.io/distroless/nodejs:14`
- `gcr.io/distroless/nodejs:10`: 10-debugと同じ
- `gcr.io/distroless/nodejs:10-debug`
- `gcr.io/distroless/nodejs:12-debug`
- `gcr.io/distroless/nodejs:14-debug`

一般的な`Dockerfile`は、`ENTRYPOINT`がシェル、`CMD`がそのシェルから呼び出されるプログラムです。distrolessはシェルがなく、`ENTRYPOINT`にNode.jsが設定されているので、`CMD`にはJavaScriptのスクリプトを設定します。拡張を使わないコードなら簡単に動作します。先ほどの`Dockerfile`と、ビルド部分はまったく同じです。

Dockerfile

```
# ここから下がビルド用イメージ

FROM node:12-buster AS builder

WORKDIR app
COPY package.json package-lock.json ./
RUN npm ci
COPY tsconfig.json ./
COPY src ./src
RUN npm run build

# ここから下が実行用イメージ

FROM gcr.io/distroless/nodejs AS runner
WORKDIR /opt/app
COPY --from=builder /app/dist ./
EXPOSE 3000
CMD ["/opt/app/index.js"]
```

ウェブフロントエンドのDockerイメージの作成

本節ではウェブフロントエンドが一式格納されたDockerイメージを作成します。もし、Next.jsでサーバーサイドレンダリングを行う場合、それは単なるNode.jsのサーバーですので、前節の内容に従ってNode.jsのコンテナを作成してください。本節で扱うのは、サーバーを伴わないHTML/JavaScriptなどのフロントエンドのファイルを配信する方法です。

シングルページアプリケーションをビルドすると静的なHTMLやJS、CSSのファイル群ができます。これらのファイルを利用する方法はいくつかあります。

- CDNやオブジェクトストレージにアップする
- Dockerコンテナとしてデプロイする

このうち、CDNやオブジェクトストレージへのアップロードはそれぞれのサービスごとの作法に従って行います。ここではDockerコンテナとしてデプロイする方法を紹介します。Dockerコンテナにするメリットはいくつかあります。主にテスト環境の構築がしやすい点です。

デプロイ用のバックエンドサーバーをDocker化する流れは今後も加速していくでしょう。しかし、フロントエンドが特定のマネージドサービスにアップロードする形態の場合、デプロイ手段がバックエンドと異なるため、別のデプロイ方式を取る必要が出てきます。ちょっとしたステージング環境や、開発環境を構築する際に、フロントエンドもDockerイメージになっていて本番環境のCDNをエミュレートできると、ローカルでもサーバーでも、簡単に一式のサービスが起動できます。PostgreSQLのイメージや、Redisのイメージ、クラウドサービスのエミューレータなどと一緒に、docker-composeで一度に起動するとテストが簡単に行えます。

サンプルとして、次のコマンドで作ったReactのアプリケーションを使います。もし、E2EテストのCypressのような、ダウンロードが極めて重い超巨大パッケージがある場合は、`optionalDependencies`に移動しておくことをお勧めします。この`Dockerfile`ではoptionalな依存

はインストールしないようにしています。

```
$ npx create-react-app --template typescript webfront
```

Dockerコンテナにする場合、ウェブサーバーのNginxのコンテナイメージを基に、ビルド済みのJavaScript/HTML/その他のリソースを格納したコンテナイメージを作成します。次の内容が、フロントエンドアプリケーションの静的ファイルを配信するサーバーです。実行用イメージでは`nginx：alpine`イメージをベースに使っています。このイメージは最後にNginxを立ち上げる設定がされているため、実行用コンテナに必要なのは生成されたフィイルと、`nginx.conf`をコピーするだけです。

Dockerfile

```
# ここから下がビルド用イメージ

FROM node:12-buster AS builder

WORKDIR app
COPY package.json package-lock.json ./
RUN npm ci --no-optional
COPY tsconfig.json ./
COPY public ./public
COPY src ./src
RUN npm run build

# ここから下が実行用イメージ

FROM nginx:alpine AS runner
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=builder /app/build public
EXPOSE 80
```

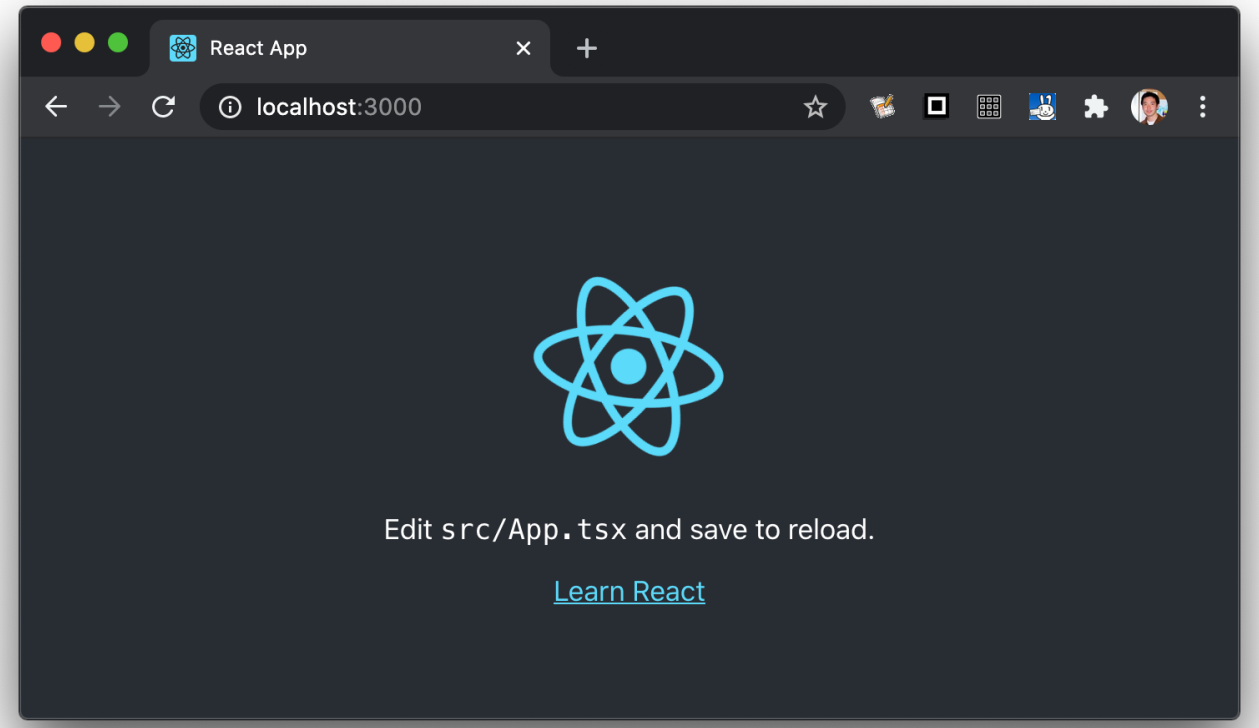
配信用のnginxの設定です。シングルページアプリケーションにとって大切なパートが`try_files`です。シングルページアプリケーションでは1つのHTML/JSがあらゆるページを作り上げます。そしてその時にURLを書き換えます。しかし、そこでブラウザリロードをすると、JavaScriptによって作られた仮想的なURLを読みにこうとします。この`try_files`を有効にすると、一度アクセスしに行って見つからなかった場合にオプションで設定したファイルを返せます。ここでは`index.html`を返すので、そこでReact Routerなどのフロントで動作しているウェブサービスが仕分けを行い、もしどこにもマッチしなければフロント側のRouterがエラーをハンドリングできます。

nginx.conf

```
worker_processes 2;

events {
    worker_connections 1024;
    multi_accept on;
    use epoll;
}

http {
    include /etc/nginx/mime.types;
    server {
        listen 80;
        server_name 127.0.0.1;

        access_log /dev/stdout;
        error_log stderr;

        location / {
            root /public;
            index index.html;
            try_files $uri $uri/ /index.html =404;
            gzip on;
            gzip_types text/css application/javascript application/json image/svg+xml;
            gzip_comp_level 9;
        }
    }
}
```

それではビルドして実行してみましょう。正しく動作していることが確認できます。

```
$ docker run -it --rm --name [コンテナ名] -p "3000:80" [イメージ名]
```



ビルドした結果を*Nginx*で配信

注釈

https://qiita.com/shibukawa/items/6a3b4d4b0cbd13041e53

Kubernetesへのデプロイ

このセクションでは、作成したアプリケーションのDockerイメージをKubernetesの上で動かすための基本について説明します。

Kubernetesの概要

KubernetesはGoogleが自社の基盤をOSSとして1から再実装したソフトウェアです。アプリケーションのデプロイ単位としてDockerコンテナを用いており、開発者はコンテナイメージを作成して配信することによって、Kubernetesの上でアプリケーションを動かすことができます。

コンテナアプリケーションを本番で稼働するにあたっては、サービスを停止せずにアップデートする方法や、複数のコンテナを水平にスケールしながら負荷分散する機能など、さまざまな運用面での課題を解決する必要がありますが、Kubernetesではそれらの機能を一貫して提供してくれるメリットがあります。

Kubernetesをローカルで実行する

Kubernetesをローカルで実行するには以下のようなツールを使う方法があります。

- minikube
- kind
- microk8s

このうち、minikubeとkindでは手元のDocker上でKubernetesを動かすことができるため、Linux上でDockerを動かさずとも手元のmacOSやWindows上でDocker Desktopを使って簡単にKubernetesを立ち上げることができます。microk8sに関してはLinux上でのサポートに限られます(特にUbuntuが推奨されます)が、依存するパッケージが少ないためインストールがシンプルであるメリットがあります。ここではminikubeを使ってKubernetesを動かしてみましょう。

minikubeのインストール方法は以下の公式ドキュメントにあります。

- https://minikube.sigs.k8s.io/docs/start/

```
# minikubeを起動
$ minikube start
😄  Ubuntu 20.04 上の minikube v1.12.1
✨  Automatically selected the docker driver

❗  'docker' driver reported a issue that could affect the performance.
💡  Suggestion: enable overlayfs kernel module on your Linux

👍  Starting control plane node minikube in cluster minikube
🔥  Creating docker container (CPUs=2, Memory=2200MB) ...
🐳  Docker 19.03.2 で Kubernetes v1.18.3 を準備しています...
🔎  Verifying Kubernetes components...
🌟  Enabled addons: default-storageclass, storage-provisioner
🏄  Done! kubectl is now configured to use "minikube"
```

minikubeのセットアップが終わったら、kubectlをインストールします。kubectlはKubernetesのCLIツールで、Kubernetesそのものとは別途インストールする必要があります。各環境でのセットアップは以下のサイトを参考に行ってみてください。

- https://kubernetes.io/ja/docs/tasks/tools/install-kubectl/

これでKubernetesを触るための準備が整いました。次に、アプリケーションの準備を行います。

あらかじめ、手元のDockerで任意のタグを付けてアプリケーションをビルドしておきます。例では `typescript-kubernetes:1.0.0` とします。

```
$ docker build -t typescript-kubernetes:1.0.0 .
$ docker images
# イメージー覧が返ってくることを確認
```

そうしたら、以下のYAMLファイルを作成します。

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: typescript-kubernetes-deployment
spec:
selector:
    matchLabels:
    app: typescript-kubernetes
replicas: 3
template:
    metadata:
    labels:
        app: typescript-kubernetes
    spec:
    containers:
    - name: typescript-kubernetes
        image: typescript-kubernetes:1.0.0
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 80
---
kind: Service
apiVersion: v1
metadata:
name: typescript-kubernetes-service
labels:
    app: typescript-kubernetes
spec:
ports:
- port: 80
    targetPort: 80
selector:
    app: typescript-kubernetes
type: ClusterIP
```

作成したYAMLをKubernetesに適用します。

```
$ kubectl apply -f app.yaml
deployment.apps/typescript-kubernetes-deployment created
service/typescript-kubernetes-service created
$ kubectl get pod
NAME                                            READY   STATUS    RESTARTS   AGE
typescript-kubernetes-deployment-8bfd76d4c-2tsl6   1/1     Running   0          3m16s
typescript-kubernetes-deployment-8bfd76d4c-h6sdz   1/1     Running   0          3m13s
typescript-kubernetes-deployment-8bfd76d4c-sg2jz   1/1     Running   0          3m12s
$ kubectl get service
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes                    ClusterIP   10.96.0.1       <none>        443/TCP   35m
typescript-kubernetes-service   ClusterIP   10.107.196.16   <none>        80/TCP    7m10s
```

作成したDeployment(複数のコンテナアプリケーションをまとめて管理できるリソース)とService(複数のコンテナアプリケーションをロードバランスしてくれるネットワークリソース)が稼働していることを確認したら、今度は動作を確認します。手元のシェルで `kubectl port-forward` を

実行し、Kubernetes上のアプリケーションを手元のブラウザで接続できるようにします。

```
$ kubectl port-forward service/typescript-kubernetes-service 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

ブラウザで `localhost:8080` にアクセスすると、作成されたアプリケーションがnginxの上で動いていることが確認できます。なお、本番などでマネージドサービスを利用する場合、ClusterIP + port-forwardを利用しなくとも、以下のようにLoadBalacnerサービスを使用することで、パブリッククラウドのロードバランサーと簡単に連携させることができます。

```
kind: Service
apiVersion: v1
metadata:
name: typescript-kubernetes-service
labels:
   app: typescript-kubernetes
spec:
ports:
- port: 80
   targetPort: 80
selector:
   app: typescript-kubernetes
type: LoadBalancer
```

最後に、作成したminikubeの環境を削除します。

```
$ minikube delete
🔥  docker の「minikube」を削除しています...
🔥  Deleting container "minikube" ...
🔥  /home/kela/.minikube/machines/minikube を削除しています...
💀  Removed all traces of the "minikube" cluster.
```

まとめ

TypeScriptで作成したコードが価値を産むのは、何かしらのデプロイ作業を通じてになります。本章ではそのデプロイの方法について、さまざまな方法をを説明しています。

Dockerは強力な武器です。ぜひ使いこなせるようになってください。