# Red Hat build of Node.js 18

## Node.js ランタイムガイド

Node.js 18 を使用して、OpenShift とスタンドアロンの RHEL で実行するスケーラブルなネットワークアプリケーションを開発する

Last Updated: 2023-08-22

# Red Hat build of Node.js 18 Node.js ランタイムガイド

Node.js 18 を使用して、OpenShift とスタンドアロンの RHEL で実行するスケーラブルなネットワークアプリケーションを開発する

## 法律上の通知



## 概要

本ガイドでは、Node.js ランタイムの使用方法を説明します。

# 目次

# はじめに

本ガイドでは、概念と、開発者が Node.js ランタイムを使用するために必要な実用的な詳細情報を説明します。

# 第1章 NODE.JS でのアプリケーション開発の概要

本セクションでは、Red Hat ランタイムでのアプリケーション開発の基本概念を説明します。また、Node.js ランタイムの概要についても説明します。

## 1.1. RED HAT RUNTIMES でのアプリケーション開発の概要

Red Hat OpenShift は、クラウドネイティブランタイムのコレクションを提供するコンテナーアプリケーションプラットフォームです。ランタイムを使用して、OpenShift で Java または JavaScript アプリケーションを開発、ビルド、およびデプロイできます。

Red Hat Runtimes for OpenShift を使用したアプリケーション開発には、以下が含まれます。

- OpenShift で実行するように設計された Eclipse Vert.x、Thorntail、Spring Boot などのランタイムのコレクション。
- OpenShift でのクラウドネイティブ開発への規定的なアプローチ。

OpenShift は、アプリケーションのデプロイメントおよびモニタリングの管理、保護、自動化に役立ちます。ビジネス上の問題を小規模なマイクロサービスに分割し、OpenShift を使用してマイクロサービスをデプロイし、監視し、維持することができます。サーキットブレーカー、ヘルスチェック、サービス検出などのパターンをアプリケーションに実装できます。

クラウドネイティブな開発は、クラウドコンピューティングを最大限に活用します。

以下でアプリケーションをビルドし、デプロイし、管理できます。

**OpenShift Container Platform**

Red Hat のプライベートオンプレミスクラウド。

**Red Hat CodeReady Studio**

アプリケーションの開発、テスト、およびデプロイを行う統合開発環境 (IDE)。

本ガイドでは、Node.js ランタイムに関する詳細情報を提供します。その他のランタイムの詳細は、関連する ランタイムドキュメント を参照してください。

## 1.2. NODE.JS の概要

Node.js は、Google からの V8 JavaScript エンジン をベースとしており、サーバー側の JavaScript アプリケーションを作成できます。効率的なアプリケーションの作成を可能にするイベントおよび非ブロッキング操作をベースとした I/O モデルを提供します。Node.js は、npm と呼ばれる大規模なモジュールエコシステムも提供します。Node.js の詳細は、関連情報 を参照してください。

Node.js ランタイムを使用すると、ローリング更新、継続的デリバリーパイプライン、サービス検出、カナリアデプロイメントなどの OpenShift プラットフォームの利点と利便性を提供しながら、Node.js アプリケーションとサービスを OpenShift で実行できます。また、OpenShift を使用すると、外部化設定、ヘルスチェック、サーキットブレーカー、フェイルオーバーなどの一般的なマイクロサービスパターンをアプリケーションに実装することが容易になります。

Red Hat は、Node.js のさまざまなサポート対象リリースを提供しています。サポートの利用方法の詳細は、Getting Node.js and support from Red Hat を参照してください。

### 1.2.1. Node.js でサポートされるアーキテクチャー

Node.js は以下のアーキテクチャーをサポートします。

- x86_64 (AMD64)
- OpenShift 環境の IBM Z (s390x)
- OpenShift 環境の IBM Power System (ppc64le)

### 1.2.2. 連邦情報処理標準 (FIPS) のサポート

FIPS (Federal Information Processing Standards) は、コンピューターシステムやネットワーク間のセキュリティーおよび相互運用性を強化するためのガイドラインと要件を提供します。FIPS 140-2 および 140-3 シリーズは、ハードウェアおよびソフトウェアの両レベルで暗号化モジュールに適用されます。

連邦情報処理標準 (FIPS) 140-2 は、米国政府および業界の作業グループが、暗号化モジュールの品質を検証するために開発されたコンピューターセキュリティー標準です。NIST Computer Security Resource Center で公式の FIPS の刊行物を参照してください。

Red Hat Enterprise Linux (RHEL) は、FIPS 140-2 コンプライアンスシステム全体を有効にする統合フレームワークを提供します。FIPS モードで操作する場合、暗号化ライブラリーを使用するソフトウェアパッケージはグローバルポリシーに従って自己設定されます。

コンプライアンスの要件は、Red Hat Government Standards ページを参照してください。

Node.js の Red Hat ビルドは、FIPS 対応の RHEL システムで実行し、RHEL が提供する FIPS 認定ライブラリーを使用します。

#### 1.2.2.1. 関連情報

- FIPS モードを有効にして RHEL をインストールする方法は、FIPS モードが有効になっている RHEL 8 システムのインストール を参照してください。
- RHEL をインストールした後に FIPS モードを有効にする方法は、FIPS モードへのシステムの切り替え を参照してください。

#### 1.2.2.2. Node.js が FIPS モードで実行していることを確認する

**crypto.fips** を使用して、Node.js が FIPS モードで実行していることを確認できます。

**前提条件**

- RHEL ホストで FIPS が有効になっている。

**手順**

1. Node.js プロジェクトで、**app.js** などの名前のアプリケーションファイルを作成します。
2. **app.js** ファイルで、次の詳細を入力します。

```
const crypto = require('crypto');
console.log(crypto.fips);
```

3. **app.js** ファイルを保存します。

**検証**

- Node.js プロジェクトで、**app.js** ファイルを実行します。

```
node app.js
```

FIPS が有効になっていると、アプリケーションはコンソールに **1** を出力します。FIPS が無効になっていると、アプリケーションはコンソールに **0** を出力します。

# 第2章 NODE.JS アプリケーションの開発およびデプロイ

新しい Node.js アプリケーションを作成して OpenShift にデプロイできます。

## 2.1. NODE.JS アプリケーションの開発

基本的な Node.js アプリケーションの場合は、Node.js メソッドを含む JavaScript ファイルを作成する必要があります。

**前提条件**

- **npm** がインストールされている。

**手順**

1. 新しいディレクトリー **myApp** を作成し、そのディレクトリーに移動します。

```
$ mkdir myApp
$ cd MyApp
```

これは、アプリケーションのルートディレクトリーです。

2. **npm** でアプリケーションを初期化します。
この例の残りの部分では、エントリーポイントが **app.js** であると想定しています。これは、**npm init** の実行時に設定するように求められます。

```
$ cd myApp
$ npm init
```

3. **app.js** という名前の新規ファイルにエントリーポイントを作成します。

例: **app.js**

```
const http = require('http');

const server = http.createServer((request, response) => {
  response.statusCode = 200;
  response.setHeader('Content-Type', 'application/json');

  const greeting = {content: 'Hello, World!'};

  response.write(JSON.stringify(greeting));
  response.end();
});

server.listen(8080, () => {
  console.log('Server running at http://localhost:8080');
});
```

4. アプリケーションを起動します。

```
$ node app.js
Server running at http://localhost:8080
```

5. **curl** またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で稼働していることを確認します。

```
$ curl http://localhost:8080
{"content":"Hello, World!"}
```

**追加情報**

- Node.js ランタイムは、Node.js API ドキュメント に記載されているコア Node.js API を提供します。

## 2.2. NODE.JS アプリケーションの OPENSHIFT へのデプロイ

Node.js アプリケーションを OpenShift にデプロイするには、**nodeshift** をアプリケーションに追加し、**package.json** ファイルを設定してから **nodeshift** を使用してデプロイします。

### 2.2.1. OpenShift デプロイメントに向けた Node.js アプリケーションの準備

OpenShift のデプロイメント用に Node.js アプリケーションを準備するには、以下の手順を実行する必要があります。

- **nodeshift** をアプリケーションに追加します。
- **openshift** および **start** エントリーを **package.json** ファイルに追加します。

**前提条件**

- **npm** がインストールされている。

**手順**

1. **nodeshift** をアプリケーションに追加します。

```
$ npm install nodeshift --save-dev
```

2. **openshift** および **start** エントリーを **package.json** の **scripts** セクションに追加します。

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "openshift": "nodeshift --expose --
dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12",
    "start": "node app.js",
    ...
  }
  ...
}
```

**openshift** スクリプトは **nodeshift** を使用して、アプリケーションを OpenShift にデプロイします。

注記

Universal Base Image および RHEL イメージが Node.js で利用できます。イメージ名の詳細は、Node.js リリースノートを参照してください。

3. **オプション**: **package.json** に **files** セクションを追加します。

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    ...
  },
  "files": [
    "package.json",
    "app.js"
  ]
  ...
}
```

**files** セクションは、**OpenShift** にデプロイするときに含めるファイルとディレクトリーを nodeshift に指示します。**nodeshift** は **node-tar** モジュールを使用して、**files** セクションにリストしたファイルとディレクトリーに基づいて tar ファイルを作成します。この tar ファイルは、**nodeshift** がアプリケーションを OpenShift にデプロイする際に使用されます。**files** セクションが指定されていない場合、**nodeshift** は現在のディレクトリー全体を送信します。ただし、以下は除外されます。

- **node_modules/**
- **.git/**
- **tmp/**
  OpenShift へのデプロイ時に不要なファイルを含めないように **package.json** に **files** セクションを含めることが推奨されます。

### 2.2.2. Node.js アプリケーションの OpenShift へのデプロイ

**nodeshift** を使用して Node.js アプリケーションを OpenShift にデプロイできます。

**前提条件**

- CLI クライアント **oc** がインストールされている。
- **npm** がインストールされている。
- ルートを設定する際に、アプリケーションが使用するポートがすべて正しく公開されていることを確認する。

**手順**

1. **oc** クライアントを使用して OpenShift インスタンスにログインします。

```
$ oc login ...
```

2. **nodeshift** を使用して、アプリケーションを OpenShift にデプロイします。

```
$ npm run openshift
```

## 2.3. NODE.JS アプリケーションのスタンドアロンの RED HAT ENTERPRISE LINUX へのデプロイ

**npm** を使用して、Node.js アプリケーションをスタンドアロンの Red Hat Enterprise Linux にデプロイできます。

**前提条件**

- Node.js アプリケーション。
- npm 6.14.8 がインストールされている。
- RHEL 7 または RHEL 8 がインストールされている。
- Node.js がインストールされている。

**手順**

1. プロジェクトの **package.json** ファイルに追加の依存関係を指定した場合は、アプリケーションを実行する前にその依存関係をインストールしてください。

```
$ npm install
```

2. アプリケーションの root ディレクトリーからアプリケーションをデプロイします。

```
$ node app.js
Server running at http://localhost:8080
```

**検証手順**

1. **curl** またはブラウザーを使用して、アプリケーションが **http://localhost:8080** で実行していることを確認します。

```
$ curl http://localhost:8080
```

# 第3章 NODE.JS ベースのアプリケーションのデバッグ

このセクションでは、Node.js ベースのアプリケーションのデバッグと、ローカルデプロイメントとリモートデプロイメントの両方でデバッグロギングを使用する方法を説明します。

## 3.1. リモートのデバッグ

アプリケーションをリモートでデバッグするには、デバッグモードで起動し、デバッガーを割り当てる必要があります。

### 3.1.1. アプリケーションのローカルでの起動およびネイティブデバッガーの割り当て

ネイティブデバッガーを使用すると、ビルトインデバッグクライアントを使用して Node.js ベースのアプリケーションをデバッグできます。

**前提条件**

- デバッグするアプリケーション。

**手順**

1. デバッガーを有効にしてアプリケーションを起動します。
   ネイティブデバッガーは自動的に割り当てられ、デバッグプロンプトを提供します。

   デバッガーが有効になっているサンプルアプリケーション

   ```
   $ node inspect app.js
   < Debugger listening on ws://127.0.0.1:9229/12345678-aaaa-bbbb-cccc-0123456789ab
   < For help see https://nodejs.org/en/docs/inspector
   < Debugger attached.
   ...
   debug>
   ```

   アプリケーションに別のエントリーポイントがある場合は、コマンドを変更してそのエントリーポイントを指定する必要があります。

   ```
   $ node inspect path/to/entrypoint
   ```

   たとえば、express generator を使用してアプリケーションを作成する場合、エントリーポイントはデフォルトで **./bin/www** に設定されます。

2. デバッガープロンプトを使用して デバッグコマンド を実行します。

### 3.1.2. アプリケーションをローカルに起動して V8 インスペクターをアタッチする

V8 インスペクターを使用すると、Chrome Debugging Protocol を使用する Chrome DevTools などの他のツールを使用して Node.js ベースのアプリケーションをデバッグできます。

**前提条件**

- デバッグするアプリケーション。

- Google Chrome ブラウザーで 提供されるような V8 インスペクターがインストールされている。

**手順**

1. V8 インスペクターの統合を有効にして アプリケーションを起動します。

```
$ node --inspect app.js
```

アプリケーションに別のエントリーポイントがある場合は、コマンドを変更してそのエントリーポイントを指定する必要があります。

```
$ node --inspect path/to/entrypoint
```

たとえば、express generator を使用してアプリケーションを作成する場合、エントリーポイントはデフォルトで **./bin/www** に設定されます。

2. V8 インスペクターをアタッチし、デバッグコマンドを実行します。
たとえば、Google Chrome を使用している場合は、以下のようになります。

   a. **chrome://inspect** に移動します。

   b. 以下の **リモートターゲット** からアプリケーションを選択します。

   c. これで、アプリケーションのソースを確認し、デバッグアクションを実行できるようになりました。

### 3.1.3. デバッグモードでの OpenShift でのアプリケーションの起動

OpenShift で Node.js ベースのアプリケーションをリモートでデバッグするには、コンテナー内で **NODE_ENV** 環境変数を **development** に設定し、リモートデバッガーからアプリケーションに接続できるようにポート転送を設定する必要があります。

**前提条件**

- アプリケーションが OpenShift で実行している。
- **oc** バイナリーがインストールされている。
- ターゲット OpenShift 環境で **oc port-forward** コマンドを実行できる。

**手順**

1. **oc** コマンドを使用して、利用可能なデプロイメント設定を一覧表示します。

```
$ oc get dc
```

2. アプリケーションのデプロイメント設定の **NODE_ENV** 環境変数を **development** に設定して、デバッグを有効にします。以下に例を示します。

```
$ oc set env dc/MY_APP_NAME NODE_ENV=development
```

3. 設定変更時に自動的に再デプロイするように設定されていない場合は、アプリケーションを再デプロイします。以下に例を示します。

```
$ oc rollout latest dc/MY_APP_NAME
```

4. ローカルマシンからアプリケーション Pod へのポート転送を設定します。

   a. 現在実行中の Pod を一覧表示し、アプリケーションが含まれる Pod を検索します。

   ```
   $ oc get pod
   NAME                     READY     STATUS      RESTARTS   AGE
   MY_APP_NAME-3-1xrsp      0/1       Running     0          6s
   ...
   ```

   b. ポート転送を設定します。

   ```
   $ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5858
   ```

   ここで、**$LOCAL_PORT_NUMBER** はローカルマシンで選択した未使用のポート番号になります。リモートデバッガー設定のこの番号を覚えておいてください。

5. V8 インスペクターをアタッチし、デバッグコマンドを実行します。
   たとえば、Google Chrome を使用している場合は、以下のようになります。

   a. **chrome://inspect** に移動します。

   b. **Configure** をクリックします。

   c. **127.0.0.1:$LOCAL_PORT_NUMBER** を追加します。

   d. **Done** をクリックします。

   e. 以下の **リモートターゲット** からアプリケーションを選択します。

   f. これで、アプリケーションのソースを確認し、デバッグアクションを実行できるようになりました。

6. デバッグが完了したら、アプリケーション Pod の **NODE_ENV** 環境変数の設定を解除します。
   以下に例を示します。

```
$ oc set env dc/MY_APP_NAME NODE_ENV-
```

## 3.2. デバッグロギング

デバッグロギングは、デバッグ時に詳細な情報をアプリケーションログに追加する方法です。これにより、以下が可能になります。

- アプリケーションの通常の操作中のロギングの出力を最小限に抑えて、読みやすさを改善し、ディスク領域の使用量を削減します。
- 問題の解決時にアプリケーションの内部作業に関する詳細情報を表示します。

### 3.2.1. デバッグロギングの追加

この例では、デバッグパッケージ を使用しますが、デバッグロギングを処理できる その他のパッケージも利用可能です。

**前提条件**

- デバッグするアプリケーションがある。

**手順**

1. **debug** ロギング定義を追加します。

```
const debug = require('debug')('myexample');
```

2. デバッグステートメントを追加します。

```
app.use('/api/greeting', (request, response) => {
  const name = request.query ? request.query.name : undefined;
  //log name in debugging
  debug('name: '+name);
  response.send({content: `Hello, ${name || 'World'}`});
});
```

3. デバッグ モジュールを **package.json** に追加します。

```
...
"dependencies": {
    "debug": "^3.1.0"
  }
```

アプリケーションによっては、このモジュールはすでに含まれている場合があります。たとえば、express generator を使用してアプリケーションを作成する場合、**debug** モジュールはすでに **package.json** に追加されています。

4. アプリケーションの依存関係をインストールします。

```
$ npm install
```

### 3.2.2. localhost でのデバッグログへのアクセス

アプリケーションを起動し、デバッグロギングを有効にする場合は、**DEBUG** 環境変数を使用します。

**前提条件**

- デバッグロギングを使用するアプリケーション。

**手順**

1. アプリケーションを起動し、デバッグロギングを有効にする場合は、**DEBUG** 環境変数を設定します。

```
$ DEBUG=myexample npm start
```

**debug** モジュールでは、ワイルドカード を使用してデバッグメッセージをフィルターできます。これは **DEBUG** 環境変数を使用して設定されます。

2. アプリケーションをテストしてデバッグロギングを呼び出します。

たとえば、次のコマンドは、**/api/greeting** メソッドで **name** 変数をログに記録するようにデバッグログが設定されている REST API レベル 0 アプリケーションの例に基づいています。

```
$ curl http://localhost:8080/api/greeting?name=Sarah
```

3. アプリケーションログを表示して、デバッグメッセージを表示します。

```
myexample name: Sarah +3m
```

### 3.2.3. OpenShift での Node.js デバッグログへのアクセス

OpenShift のアプリケーション Pod で **DEBUG** 環境変数を使用して、デバッグロギングを有効にします。

**前提条件**

- デバッグロギングを使用するアプリケーション。
- CLI クライアント **oc** がインストールされている。

**手順**

1. **oc** CLI クライアントを使用して、OpenShift インスタンスにログインします。

```
$ oc login ...
```

2. アプリケーションを OpenShift にデプロイします。

```
$ npm run openshift
```

これにより、**openshift** npm スクリプトを実行します。これは nodeshift への直接呼び出しをラップします。

3. Pod の名前を見つけ、ログを追跡して起動を監視します。

```
$ oc get pods
....
$ oc logs -f pod/POD_NAME
```

重要

Pod が起動したら、このコマンドを実行したままにして、新しいのターミナルウィンドウで残りの手順を実行します。これにより、ログを **追跡** でき、そのログの新しいエントリーを確認することができます。

4. アプリケーションをテストします。
たとえば、次のコマンドは、**/api/greeting** メソッドで **name** 変数をログに記録するようにデバッグログが設定されている REST API レベル 0 アプリケーションの例に基づいています。

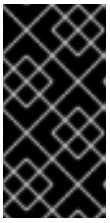
```
$ oc get routes
...
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

5. Pod ログに戻り、ログにデバッグロギングメッセージがないことに注意してください。
6. **DEBUG** 環境変数を設定して、デバッグロギングを有効にします。

```
$ oc get dc
...
$ oc set env dc DC_NAME DEBUG=myexample
```

7. Pod ログに戻り、更新ロールアウトを監視します。
   更新がロールアウトされると Pod が停止し、ログをフォローしなくなります。
8. 新規 Pod の名前を見つけ、ログを追跡します。

```
$ oc get pods
....
$ oc logs -f pod/POD_NAME
```



重要

Pod が起動したら、このコマンドを実行したままにして、別のターミナルウィンドウで残りの手順を実行します。これにより、ログを **追跡** でき、そのログの新しいエントリーを確認することができます。具体的には、ログにはデバッグメッセージが表示されます。

9. アプリケーションをテストして、デバッグロギングを呼び出します。

```
$ oc get routes
...
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

10. Pod ログに戻り、デバッグメッセージを表示します。

```
...
myexample name: Sarah +3m
```

デバッグロギングを無効にするには、Pod から **DEBUG** 環境変数を削除します。

```
$ oc set env dc DC_NAME DEBUG-
```

関連情報

環境変数の詳細は、OpenShift のドキュメント を参照してください。

# 付録A NODESHIFT について

Nodeshift は、Node.js プロジェクトで OpenShift デプロイメントを実行するためのモジュールです。

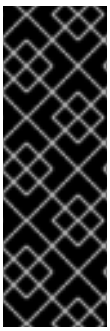
重要

Nodeshift は、**oc** CLI クライアントがインストールされ、OpenShift クラスターにログインしていることを前提としています。また、Nodeshift は、**oc** CLI クライアントが使用している現在のプロジェクトを使用します。

Nodeshift は、プロジェクトの root にある **.nodeshift** フォルダーのリソースファイルを使用して、OpenShift Routes、Services、および DeploymentConfig の作成を処理します。Nodeshift の詳細は、Nodeshift プロジェクトページ を参照してください。

# 付録B サンプルアプリケーションのデプロイメント設定の更新

サンプルアプリケーションのデプロイメント設定には、ルート情報や readiness プローブの場所などの OpenShift でのアプリケーションのデプロイおよび実行に関連する情報が含まれます。サンプルアプリケーションのデプロイメント設定は YAML ファイルのセットに保存されます。Fabric8 Maven プラグインを使用する例では、YAML ファイルは **src/main/fabric8/** ディレクトリーにあります。Nodeshift を使用する例では、YAML ファイルは **.nodeshift** ディレクトリーにあります。



重要

Fabric8 Maven Plugin および Nodeshift が使用するデプロイメント設定ファイルは完全な OpenShift リソース定義である必要はありません。Fabric8 Maven Plugin と Nodeshift の両方がデプロイメント設定ファイルを取り、不足している情報を追加して完全な OpenShift リソース定義を作成できます。Fabric8 Maven Plugin によって生成されるリソース定義は **target/classes/META-INF/fabric8/** ディレクトリーにあります。Nodeshift によって生成されるリソース定義は **tmp/nodeshift/resource/** ディレクトリーにあります。

**前提条件**

- 既存のサンプルプロジェクト。
- CLI クライアント **oc** がインストールされている。

**手順**

1. 既存の YAML ファイルを編集するか、または設定を更新して追加の YAML ファイルを作成します。
   - たとえば、サンプルに **readinessProbe** が設定された YAML ファイルがすでにある場合は、**path** の値を別の利用可能なパスに変更し、readiness の有無を確認することができます。

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - **readinessProbe** が既存の YAML ファイルで設定されていない場合は、**readinessProbe** 設定を使用して新規 YAML ファイルを同じディレクトリーに作成することもできます。
2. Maven または npm を使用して、サンプルの更新バージョンをデプロイします。
3. 設定更新が、デプロイ済みのサンプルに表示されることを確認します。

```
$ oc export all --as-template='my-template'

apiVersion: template.openshift.io/v1
kind: Template
```

```
metadata:
  creationTimestamp: null
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
  spec:
    ...
    template:
      ...
      spec:
        containers:
          ...
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /path/to/different/probe
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 60
            periodSeconds: 30
            successThreshold: 1
            timeoutSeconds: 1
          ...
```

関連情報

Web ベースのコンソールまたは CLI クライアント **oc** を使用してアプリケーションの設定を直接更新している場合は、その変更を YAML ファイルへエクスポートして追加します。**oc export all** コマンドを使用して、デプロイされたアプリケーションの設定を表示します。

# 付録C NODESHIFT で NODE.JS アプリケーションをデプロイするための JENKINS フリースタイルプロジェクトの設定

ローカルホストの nodeshift を使用して Node.js アプリケーションをデプロイするのと同様に、Jenkins を nodeshift を使用して Node.js アプリケーションをデプロイするように設定できます。

**前提条件**

- OpenShift クラスターへのアクセス
- 同じ OpenShift クラスターで実行している Jenkins コンテナーイメージ。
- Jenkins サーバーに The Node.js プラグイン がインストールされている。
- nodeshift および Red Hat ベースイメージを使用するように設定されている Node.js アプリケーション。

  **nodeshift での Red Hat ベースイメージの使用例**

  ```
  $ nodeshift --dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12 ...
  ```

- GitHub で利用可能なアプリケーションのソース。

**手順**

1. アプリケーションの新規 OpenShift プロジェクトを作成します。
   a. OpenShift Web コンソールを開き、ログインします。
   b. **Create Project** をクリックして、新しい OpenShift プロジェクトを作成します。
   c. プロジェクト情報を入力し、**Create** をクリックします。
2. Jenkins がそのプロジェクトにアクセスできるようにします。
   たとえば、Jenkins のサービスアカウントを設定している場合は、アカウントに、アプリケーションのプロジェクトへの **edit** アクセスがあることを確認してください。
3. Jenkins サーバーで新しい フリースタイルの Jenkins プロジェクト を作成します。
   a. **New Item** をクリックします。
   b. 名前を入力し、**Freestyle プロジェクト** を選択して **OK** をクリックします。
   c. **Source Code Management** で **Git** を選択し、アプリケーションの GitHub URL を追加します。
   d. **Build Environment** で、 **Provide Node & npm bin/ folder to PATH**が確認され、Node.js 環境が設定されていることを確認してください。
   e. **Build** で、**Add build step** を選択し、**Execute Shell** を選択します。
   f. 以下を **コマンド** エリアに追加します。

```
npm install -g nodeshift
nodeshift --dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12 --
namespace=MY_PROJECT
```

**MY_PROJECT** をアプリケーションの OpenShift プロジェクトの名前に置き換えます。

g. **Save** をクリックします。

4. Jenkins プロジェクトのメインページから **Build Now** をクリックし、アプリケーションの OpenShift プロジェクトにアプリケーションのビルドおよびデプロイを確認します。
アプリケーションの OpenShift プロジェクトでルートを開いて、アプリケーションがデプロイされていることを確認することもできます。

## 次のステップ

- GITSCM ポーリング を追加すること、または **Poll SCM** ビルドトリガー を使用することを検討してください。これらのオプションにより、新規コミットが GitHub リポジトリーにプッシュされるたびにビルドを実行できます。
- Node.js プラグインを設定 する際に、nodeshift をグローバルパッケージとして追加することを検討してください。これにより、**Execute Shell** ビルドステップを追加する際に、**npm install -g nodeshift** を省略できます。
- デプロイ前にテストを実行するビルドステップを追加することを検討してください。

# 付録D PACKAGE.JSON プロパティーの内訳

nodejs-rest-http/package.json

```
{
  "name": "nodejs-rest-http",
  "version": "4.0.0",
  "author": "Red Hat, Inc.",
  "license": "Apache-2.0",
  "scripts": {
    "pretest": "eslint --ignore-path .gitignore .",
    "test": "nyc --reporter=lcov mocha", (1)
    "prepare": "echo 'To confirm CVE compliance, run \"npm audit\"' ",
    "release": "standard-version -a",
    "openshift": "nodeshift --dockerImage=registry.access.redhat.com/ubi8/nodejs-16", (2)
    "start": "node ." (3)
  },
  "main": "./bin/www", (4)
  "standard-version": {
    "scripts": {
      "postbump": "npm run postinstall && node release.js",
      "precommit": "git add .openshiftio/application.yaml"
    }
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/nodeshift-starters/nodejs-rest-http.git"
  },
  "files": [ (5)
    "package.json",
    "app.js",
    "public",
    "bin",
    "LICENSE"
  ],
  "bugs": {
    "url": "https://github.com/nodeshift-starters/nodejs-rest-http/issues"
  },
  "homepage": "https://github.com/nodeshift-starters/nodejs-rest-http",
  "devDependencies": { (6)
    "eslint": "^7.32.0",
    "eslint-config-semistandard": "^16.0.0",
    "js-yaml": "^4.1.0",
    "mocha": "^9.1.3",
    "nodeshift": "~8.6.0",
    "nyc": "~15.1.0",
    "standard-version": "^9.3.2",
    "supertest": "~6.1.6"
  },
  "dependencies": { (7)
    "body-parser": "~1.19.0",
    "debug": "^4.3.3",
    "express": "~4.17.1",
    "pino": "^7.5.1",
```

```
    "pino-debug": "^2.0.0",
    "pino-pretty": "^7.2.0"
  }
}
```

1. ユニットテストを実行するための **npm** スクリプト。**npm run test** で実行します。
2. このアプリケーションを OpenShift Container Platform にデプロイするための **npm** スクリプト。**npm run openshift** で実行します。
3. このアプリケーションを起動する **npm** スクリプト。**npm start** で実行します。
4. **npm start** で実行する際のアプリケーションのプライマリーエントリーポイント。
5. OpenShift Container Platform にアップロードされるバイナリーに含まれるファイルを指定します。
6. **npm** レジストリーからインストールする開発依存関係の一覧。これらは、テストおよび OpenShift Container Platform へのデプロイに使用されます。
7. **npm** レジストリーからインストールされる依存関係の一覧。

# 付録E 追加の NODE.JS リソース

- Node.js ホームページ
- npm ホームページ

# 付録F アプリケーション開発リソース

OpenShift でのアプリケーション開発に関する詳細は、以下を参照してください。

- OpenShift インタラクティブラーニングポータル

# 付録G SOURCE-TO-IMAGE (S2I) ビルドプロセス

Source-to-Image (S2I) は、アプリケーションソースのあるオンライン SCM リポジトリーから再現可能な Docker 形式のコンテナーイメージを生成するビルドツールです。S2I ビルドを使用すると、ビルド時間を短縮し、リソースおよびネットワークの使用量を減らし、セキュリティーを改善し、その他の多くの利点を使用して、アプリケーションの最新バージョンを実稼働に簡単に配信できます。OpenShift は、複数の ビルドストラテジーおよび入力ソース をサポートします。

詳細は、OpenShift Container Platform ドキュメントの Source-to-Image (S2I) ビルド の章を参照してください。

最終的なコンテナーイメージをアセンブルするには、S2I プロセスに 3 つの要素を指定する必要があります。

- GitHub などのオンライン SCM リポジトリーでホストされるアプリケーションソース。
- S2I Builder イメージ。アセンブルされたイメージの基盤となり、アプリケーションを実行しているエコシステムを提供します。
- 必要に応じて、S2I スクリプト によって使用される環境変数およびパラメーターを指定することもできます。

このプロセスは、S2I スクリプトで指定された指示に従ってアプリケーションソースおよび依存関係を Builder イメージに挿入し、アセンブルされたアプリケーションを実行する Docker 形式のコンテナーイメージを生成します。詳細は、OpenShift Container Platform ドキュメントの S2I ビルド要件、ビルドオプション および ビルドの仕組み を参照してください。