実践 REST サーバ

Node.js、Restify、MongoDBによる バックエンド開発

豊沢 聡◉著

CUTT カットシステム

■**サンプルファイルのダウンロードについて**

本書掲載のサンプルファイルは、一部を除いてインターネット上のダウンロードサービスからダウンロードすることができます。詳しい手順については、本書の巻末にある袋とじの内容をご覧ください。

なお、ダウンロードサービスのご利用にはユーザー登録と袋とじ内に記されている番号が必要です。そのため、本書を中古書店から購入されたり、他者から貸与、譲渡された場合にはサービスをご利用いただけないことがあります。あらかじめご承知おきください。

・本書の内容についてのご意見、ご質問は、お名前、ご連絡先を明記のうえ、小社出版部宛文書（郵送またはE-mail）でお送りください。
・電話によるお問い合わせはお受けできません。
・本書の解説範囲を越える内容のご質問や、本書の内容と無関係なご質問にはお答えできません。
・匿名のフリーメールアドレスからのお問い合わせには返信しかねます。

はじめに

本書は、Node.js の Restify フレームワークを用いた REST サーバの構築方法を説明します。バックエンドデータベースには MongoDB を使います。

REST（Representational State Transfer）にはいろいろな定義がありますが、ここでは、サーバとクライアントのソフトウェアが、JSON というフォーマットでデータを交換するサービスと考えます。また、一般の Web サービスと同じように、通信プルトコルに HTTP を採用するものとします。そして、`GET` や `POST` などの HTTP メソッドを通じて、サーバに置かれたリソース（データ等）に対し、作成（create）、取得（read）、更新（update）、削除（delete）といった CRUD 操作を施す機能をクライアントに提供します。

HTTP サーバを構築するフレームワークはいくつかあります。Python なら Django、JavaScript/Node.js なら Express.js がポピュラーですが、本書が Node.js と Restify のペアを推すのは、Resitify が（その名のとおり）REST サービスに特化しており、REST サーバ構築に必要な機能を容易に利用できるからです。

簡単なデモを示します。クライアントからの `GET /welcome` リクエストに固定メッセージを返すだけなら、次のように 7 行で書けます。

```
const restify = require('restify');
let server = restify.createServer();
server.get('/welcome', function(req, res, next) {
  res.send({message: 'Hello World'});
  return next();
});
server.listen(8080);
```

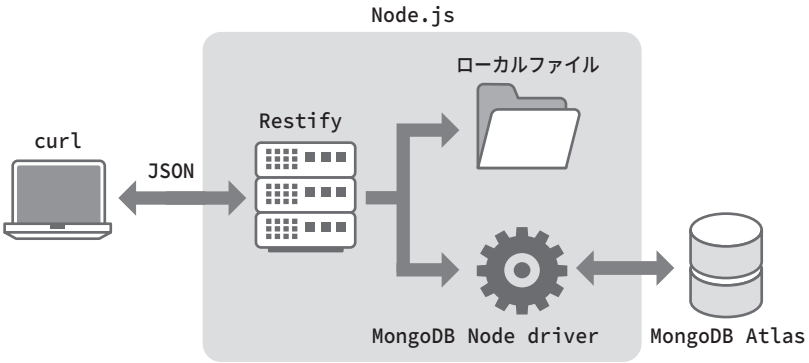
ブラウザから動作を確認します。

REST にはいろいろな用法があり、OS とインタフェースすることでシステムを設定したり、背後にデータベースを置くことでデータサービスを展開したりできます。ここでは、後者のやり方を示します。

データベースには MongoDB を採用しました。MongoDB が非リレーショナルデータベース（NoSQL）の中でもトップクラスにポピュラーであることに加え、JSON データとの親和性が高いからです。MongoDB にはいくつかのタイプがありますが、本書で用いるのはインストール不要で、（リソースに制限はあるものの）無償で利用できるクラスタサービスの Atlas です。Restify とデータベースのインタフェースには、MongoDB が提供する MongoDB Node.js ドライバを使います。

クライアントはとくに用意はしませんが、用例はコマンドライン志向の `curl` から示します。リクエストヘッダや HTTP メソッドを簡単に調整できて便利だからです。

以上、本書で構築する REST サーバのアーキテクチャを図にすると次のようになります。



これを機会に、Restify の利用が広がれば幸いです。

2024 年 3 月
豊沢 聡

■ 本書の構成

本書は 2 部構成になっています。第 I 部では REST サーバ構築に必要な Restify と MongoDB ドライバの使いかたを、第 II 部では関連する基盤技術をそれぞれ説明します。

Restify と MongoDB ドライバを使いこなすには、そのベースにある Node.js とその HTTP/HTTPS モジュール、TLS/SSL で使用するサーバ証明書、MongoDB のデータ構造やフィルタリング機能などの理解が欠かせません。Restify や MongoDB ドライバを使いながらその都度そうした基盤技術のポイントを説明してもよいのですが、それだと話の流れが損なわれます。そこで、これら基盤技術の説明は第 II 部にまとめました。これら技術のことはあらかじめわかっているのなら、飛ばしてくださって結構です。

第 I 部の目次を次に示します。

第 1 章　Restify サーバの基本

Restify のベーシックな機能を説明します。データはハードコーディングあるいはファイルベースです。

第 2 章　アクセス制御

ユーザ認証や流量制限など、サービスを保護する機能を紹介します。

第 3 章　バックエンドデータベース

Restify から MongoDB Atlas に接続することで、バックエンドデータベースに対しデータの作成、取得、更新、削除の操作を行います。

第 4 章　その他の機能

ここまでの章では取り上げなかった Restify のその他の便利な機能を紹介します。

第 II 部では、それぞれの基盤技術についてソフトウェアの概要、インストール方法あるいはアカウント作成方法、マニュアルの所在と読み方のコツ、第 I 部を理解する上で重要なポイントを説明します。

第 5 章　Node.js

基本情報に加え、Node.js パッケージ（`package.json`）の作成方法とモジュール読み込みで使う `require()` を説明します。

第 6 章　Restify

`Server.pre()` や `Server.use()` で導入するプラグイン、そして処理関数（ハンドラ）の処理順序を追加で説明します。

第 7 章　Node.js によるサーバ構築

Restify が内部で使用している HTTP/HTTPS/HTTP2 モジュール、およびそこで用いられる `http.IncomingMessage`（`req`）や `http.ServerResponse`（`res`）の用法を、サーバスクリプトの実装を通じて示します。

第 8 章　MongoDB Atalas

データベース、コレクション、ドキュメントの作成・表示方法を説明します。ただし、ネットワークサービスの Atlas だけをターゲットにしており、GUI や CLI のクライアントは取り上げません。また、使わなくなったときのために、アカウントの削除方法も示します。

第 9 章　curl

本書の範囲での用例を章末にまとめています。

第 10 章　OpenSSL

HTTPS サーバの運用に必要な自己署名サーバ証明書の作成方法を説明します。

■ ダウンロードサービス

出版社のダウンロードサービスから本書掲載のスクリプト（計 30 本）、サンプルデータ（JSON と HTML）、サンプル自己署名サーバ証明書（OpenSSL で作成した TLS/SSL 通信用）、参考文献（付録 A）がダウンロードできます。方法については扉裏をごらんください。

スクリプトは目的を達成できる最小限で書かれています。例外にはほとんど対処しないので、エラー終了することもあります。

■ インストール

実行環境は Node.js です。インストールがまだなら、第 II 部第 5 章を参照してください。

利用するパッケージ（npm）は Restify、Restify-client、MongoDB Node.js Driver、jsonwebtoken の 4 点です。ダウンロードサービスのパッケージには含まれていないので、パッケージを展開したら、そのディレクトリ上で次の要領でインストールしてください。

```
npm init -y                                    # npmパッケージを用意する
npm install restify
npm install restify-client
npm install mongodb
npm install jsonwebtoken
```

実行環境

Node.js はプラットフォームを問いません。

本書の用例は、いずれも Windows 10 上の Windows Subsystem For Linux（中身は Ubuntu）でのものです。

スクリプトの動作確認をする HTTP クライアントには、コマンドライン志向の curl を使います。用法は、その都度、次に示すコラムで説明します。

curl はコマンドライン志向の HTTP クライアントです（左のアイコンは curl のロゴ）。Windows や Unix にはデフォルトで搭載されています。curl の用法については、第 9 章にまとめがあるので参照してください。

curl は Windows 10 では WSL か Power Shell での利用をお勧めします。単一引用符を受け付けない Windows コマンドプロンプトは、使えないわけではないですが、かなり苦労します。

出力例は、見やすいように編集したうえで紙面に掲載しています。皆さんの実行結果と見栄えが異なりますが、内容は同じです。// や # などからコメントが挿入されているものもありますが、これらも説明用に追加したもので、実際の出力には現れません（JSON にはコメントが書き込めません）。

前提条件

本書では、読者には次の知識と技能のあることを前提に書かれています。

JavaScript

テンプレートリテラルや Promise などの中級レベルの技巧を使っているので、ある程度の JavaScript の経験が必要です。本書の JavaScript 実行環境である Node.js については、各種モジュールや require の使いかたをある程度はわきまえていると仮定しています。HTTP/HTTPS/HTTP2 モジュールについては、その仕組みが Restify と直接関係あるので、第 II 部第 7 章で説明します。

HTTP

GET や POST などのメソッド、Content-Type などのヘッダがどのように機能するかの大枠は知っているものとします。ヘッダのフォーマットや細かい特性はその都度説明します。

REST

URL（エンドポイント）で「リソース」にアクセスすると JSON が返ってくる、あるいは POST や PUT などのメソッドでサーバに JSON テキストを送ると「リソース」が作成されたり更新されたりするもの、と大雑把に知っていれば十分です。

JSON

[] や {} で構造化されたデータということがわかっていれば大丈夫です。データ型の定義は参考となるよう付録 C で説明します。

データベース

SQL の経験があるとわかりやすいですが、必須ではありません。本書で用いる MongoDB については、ネットワークサービス（Atlas）のアカウントの作成方法などを最初から説明します（第 II 部第 8 章）。膨大な機能があるので、本書が説明できるのはほんのとば口だけです。

HTML/CSS

使いません。

目次

第Ⅰ部

REST サーバ

第 I 部では、Node.js パッケージの Restify を使った REST サーバの構築方法を示します。

第 1 章　Restify サーバの基本

Restify のベーシックな機能を説明します。データはハードコーディングあるいはファイルベースです。

第 2 章　アクセス制御

ユーザ認証や流量制限など、サービスを保護する機能を紹介します。

第 3 章　バックエンドデータベース

Restify から MongoDB Atlas に接続することで、バックエンドデータベースに対しデータの作成、取得、更新、削除の操作を行います。

第 4 章　その他の機能

ここまでの章では取り上げなかった Restify のその他の便利な機能を紹介します。

Resifty と MongoDB を使って JavaScript/Node.js プログラミングをするには、それなりの知識が前提となりますが、ここでは、サーバ構築に専念するので、細かい周辺知識は説明しません。Node.js やその HTTP モジュール、MongoDB Atlas の用法などは第 II 部で説明します。不如意なところがあったら、その都度参照してください。

第 1 章
Restify サーバの基本

本章では、Restify のベーシックな機能を用いて REST サーバを構築します。柔軟なエンドポイントの記述方法、そしてプラグインを用いたクエリ文字列やボディの解析方法も示します。

HTTP のバージョンは 1.1 とし、メッセージは平文で交換します。末尾の 2 節ではこれを暗号で保護した HTTPS（バージョンは 1.1）および HTTP/2 にコンバートする方法を示します。

1.1 簡単な HTTP REST サーバ

■ 目的

暗号化（TLS/SSL）なしのシンプルな HTTP ベースの REST サーバを作成します。

受け付けるエンドポイント（URL）は /sake/kaiun と /sake/isojiman の 2 点です。これ以外のエンドポイントへのリクエストには「404 Not Found」を返します。使用できる HTTP メソッドは、前者では GET、後者では POST のみとします。それ以外のメソッドには「405 Method Not Allowed」を応答します。

Restify では、メソッドとエンドポイントの組がリクエストされたときの処理関数を登録することで REST サーバを構築します。

```
（メソッド，エンドポイント）=> 処理関数
```

この対応関係を設定することをルーティング（経路制御）といいます。処理関数はハンドラとも呼ばれます。Restify のコーディングは、サーバに必要なだけルーティングを準備することに他なりません。本節の場合は次のルーティングです（例題を簡単にするため、どちらのエンドポイントでも同じ関数を使っています）。

```
GET /sake/kaiun => respond
POST /sake/isojiman => respond
```

■ コード

HTTP/1.1 対応のシンプル REST サーバのコードを次に示します。

リスト 1.1 ● rest-http.js

```
1  const restify = require('restify');
2
3
4  function respond(req, res, next) {
5    res.send({
6      serverName: server.name,
7      httpVersion: req.httpVersion,
```

```
8       httpMethod:  req.method,
9       requestURI: req.url,
10      connection: `${req.socket.remoteAddress}:${req.socket.remotePort}`,
11      message: 'Drink me.'
12    });
13    return next();
14  }
15
16
17  let server = restify.createServer();
18  server.listen(8080, function () {
19    console.log('Listening on', server.url);
20    console.log('Association', server.address());
21  });
22  server.get('/sake/kaiun', respond);
23  server.post('/sake/isojiman', respond);
```

■ 実行例

コードを実行すると、サーバは localhost:8080 で待ち受けを開始し（listening）、次のようにURL のスキームと権限元、そしてサーバ側のソケットアソシエーションを表示します（コード 19 ～ 20 行目）。

```
$ node rest-http.js
Listening on http://[::]:8080
Association { address: '::', family: 'IPv6', port: 8080 }
```

[::] はオール 0 の IPv6 不定アドレスで、IPv4 では 0.0.0.0 に相当します。Restify のサーバ生成メソッド（そしてその基盤にある Node.js のメソッド）は、アドレスあるいはドメイン名が指定されなければ不定アドレスを用います。

不定アドレスでは、通常、そのホストのすべてのネットワークインタフェースのアドレスで待ち受けられます。これには localhost（127.0.0.1/8 あるいは ::1）も含まれます。また、IPv6 の localhost で待ち受けると、IPv4 でも待ち受けます。

クライアントからエンドポイント /sake/kaiun に GET メソッドからアクセスすれば、JSON テキスト（コード 5 ～ 12 行目で定義したオブジェクト）が応答されます。

```
$ curl -i localhost:8080/sake/kaiun
HTTP/1.1 200 OK
Server: restify
Content-Type: application/json
Content-Length: 150
Date: Wed, 31 Jan 2024 01:00:02 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "serverName": "restify",
  "httpVersion": "1.1",
  "httpMethod": "GET",
  "requestURI": "/sake/kaiun",
  "connection": "::ffff:127.0.0.1:55300",
  "message": "Drink me."
}
```

-i（ロングフォーマットは --include）は、レスポンスヘッダも表示する（デフォルトでは割愛される）コマンドオプションです。

スクリプトが用意している返信データは JavaScript オブジェクトですが、クライアントが受信したデータは JSON テキストです（プロパティキーが二重引用符でくくられている）。JSON の送受がメインである Restify は、このようにデータの JSON への変換を自動的に行ってくれるのです。変換メカニズムについては 4.4 節で説明します。

レスポンスヘッダの Content-Type フィールドが application/json なところもポイントです。Node.js ネイティブの HTTP モジュールでは、デフォルトでは Content-Type は挿入されません。これも、Restify が自動で加えています。

connection プロパティに示された IP アドレスは、IPv4 アドレスを IPv6 でそのまま用いるときの IPv4 射影アドレスです（先頭 80 ビットがすべて 0、続く 16 ビットがすべて 1、残りの 32 ビットが IPv4 アドレス）。

同様に、POST /sake/isojiman にアクセスします。POST はクライアントからサーバにデータを上げるメソッドですが、ここでは命令を受け付けはするものの、とくになにもせずに応答だけ返しています。応答の中身は GET /sake/kaiun とほぼ同じですが、受け付けたメソッドが POST に変わります。

```
$ curl localhost:8080/sake/isojiman -X POST
{
  "serverName": "restify",
  "httpVersion": "1.1",
  "httpMethod": "POST",                    // POSTになった
  "requestURI": "/sake/isojiman",
  "connection": "::ffff:127.0.0.1:55399",
  "message": "Drink me."
}
```

HTTP メソッドを指定するには -X（--request）オプションを用います。デフォルトは GET です。POST や PUT のように送信データがあるメソッドでは -d（--data）オプションからデータを指定します。ここでの用法のように -d が未指定なら、デフォルトで空文字が送信されます。

エンドポイント /sake/kaiun には GET しか定義されていないので、それ以外のメソッドにアクセスを試みると「405 Method Not Allowed」が返されます。コードで明示的にトラップを入れずとも、Restify が適切なエラーメッセージを自動的に送出してくれます。

```
$ curl -i localhost:8080/sake/kaiun -X PUT
HTTP/1.1 405 Method Not Allowed
Server: restify
Allow: GET
Content-Type: application/json
Content-Length: 58
Date: Fri, 05 Jan 2024 20:42:47 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "code": "MethodNotAllowed",
  "message": "PUT is not allowed"
}
```

Allow レスポンスヘッダには、このエンドポイントに使えるメソッド名が示されます。ここでは GET だけですが、複数あるときはカンマ , 区切りで列挙されます。

定義されていないエンドポイント、たとえば /sake/hananomai にアクセスすると、サーバは「404 Not Found」メッセージを応答します。これも、Restify のデフォルト動作です。

```
$ curl -i localhost:8080/sake/hananomai
HTTP/1.1 404 Not Found
Server: restify
Content-Type: application/json
Content-Length: 70
Date: Fri, 05 Jan 2024 20:53:53 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "code": "ResourceNotFound",
  "message": "/sake/hananomai does not exist"
}
```

サーバは、実行中のコンソールから Ctrl-C を押下すれば終了します。

■ 非推奨化警告

Restify を require() から読み込むと（コード 1 行目）、使用している Node.js と Restify のバージョンの組み合わせによっては、非推奨化警告が報告されます。次に示すのは、Node.js 20.0 と Restify 11.1 のペアからのものです。

```
(node:8904) [DEP0111]
DeprecationWarning: Access to process.binding('http_parser') is deprecated.
(Use `node --trace-deprecation ...` to show where the warning was created)
```

これは、HTTP/2 が正式な標準として採用されたのにともなって非推奨化された SPDY のモジュールから来ています。SPDY は利用しませんし、警告だけなので問題はありません。メッセージが邪魔なら、Node.js 実行時に --no-deprecation コマンドラインオプションを指定します。

```
$ node --no-deprecation rest-http.js
```

ファイル先頭のハッシュバンで /usr/bin/env を介して node を呼び出しているなら、次のように書きます。

```
#!/usr/bin/env -S node --no-deprecation
```

HTTP サーバ構築

以下、順にコードを説明していきます。

当然ながら、最初にまず Restify モジュールを読み込みます（1 行目）。

```
1 const restify = require('restify');
```

Restify を用いて HTTP サーバを構築するには、restify.createServer() メソッドを使います（17 行目）。

```
17 let server = restify.createServer();
```

このメソッドは Server オブジェクトを返します。

引数には各種のプロパティを設定できますが、普通に使うものは name くらいです。このオプションプロパティの値は、6 行目のクライアントへのレスポンスボディに書き込んでいる server.name で参照されるサーバ名文字列です。デフォルトは出力例で見たように、restify です。変更するには、次の要領で上書きします。

```
17 let server = restify.createServer({name: 'MyREST Server'});
```

Server.name の文字列はレスポンスヘッダの Server フィールドにも現れます。

Server の基幹部分は Node.js の http.Server クラスと同じものです。Server には server プロパティがあり、そこにネイティブの http.Server の情報が収容されています。ネットワーク関連の設定を変更するのなら、そちらを操作します。たとえば、リクエスト受け付けのタイムアウト時間は http.Server.requestTimeout から設定します。デフォルト値はどの設定でも http.Server と同じなので、詳細は Node.js のドキュメントを参照してください。

待ち受け

restify.createServer() はソケットを用意するだけです。このソケットにサーバの IP アドレスと TCP ポートを結び付け（bind する）、実際にクライアントの待ち受けを開始するには、Server.listen() メソッドを使います（18 ～ 21 行目）。

```
18 server.listen(8080, function () {
19   console.log('Listening on', server.url);
```

```
20    console.log('Association', server.address());
21  });
```

第 1 引数には、ポート番号を整数値から指定します。HTTP のウェルノウンポートは 80 番ですが、利用には管理者権限が必要です。ユーザレベルのポートには、伝統的に 8080 番が用いられます。第 1 引数が未指定、または 0 が指定されたときは、OS が適当で未使用な番号を決定します。

第 2 引数には、待ち受ける IP アドレスあるいはホスト名を文字列から指定します。18 行目の用法のように指定がなければ、デフォルトで不定アドレス（:: または 0.0.0.0）が用いられます。前述のように、不定アドレスのときはそのホストシステムのすべてのネットワークインタフェースのすべてのアドレスで待ち受けます。

オプションの第 3 引数には、Server オブジェクトに listening イベントが上がってきたときに実行するコールバック関数を指定します。ここでは、console.log() を含んだ無名関数を定義しています。

listening イベントは、サーバがクライアントの接続を受け付けられる状態になると上がってきます。ここでは、そのタイミングで。サーバの待ち受け URL を Server.url プロパティから、サーバ側のソケットアソシエーションを Server.address() メソッドから、それぞれ印字しています（19 ～ 20 行目）。これらプロパティは、ソケットに IP アドレスとポート番号が結び付けられるまで、つまり listening が上がるまでは未定義です。

ソケットアソシエーションは、クライアントとサーバの間の通信路の両端のソケットを識別するのに必要な情報の組である（サーバ側アドレス、サーバ側ポート、プロトコル、クライアント側アドレス、クライアント側ポート）を指します。server.address() はこのうち最初の 3 点を印字します（クライアントが接続してこないオープンな状態では後者 2 点は未定なので）。

listening イベント発生時の処理は、次のように Server.on() からも登録できます。

```
server.on('listening', function() { ... });
```

Server.listen() は Node.js ネイティブの HTTP モジュールの http.Server.listen() あるいはその親クラスの net.Server.listen() と等価なので、詳しくは Node.js ドキュメントを参照してください。

ルーティング

サーバオブジェクトを生成したら、ルーティングの設定をします。

```
22 server.get('/sake/kaiun', respond);
23 server.post('/sake/isojiman', respond);
```

22 行目が GET /sake/kaiun の、23 行目が POST /sake/isojiman の設定です。サーバにはこのように HTTP メソッド名と「ほぼ」一致するルーティング設定メソッドが用意されています。

これらメソッドはいずれも第 1 引数にエンドポイント（URL のパス部分）を、第 2 引数に処理関数を取ります。

HTTP メソッド

それぞれの HTTP のメソッドに対する Server のルーティング設定メソッドと REST における意味を、次の表にまとめて示します。

HTTP メソッド	Server メソッド	操作
GET	Server.get()	リソースの取得。
HEAD	Server.head()	HTTP レスポンスヘッダのみ取得。
POST	Server.post()	新規にリソースを作成。
PUT	Server.put()	既存のリソースをまるごと更新（全フィールドの情報が必要）。
PATCH	Server.patch()	既存リソースの一部更新。
DELETE	Server.del()	既存リソースの削除。
OPTIONS	Server.opts()	リソースに対する利用可能なメソッドの問い合わせ。

メソッドの機能は実装依存なものが多いため、上記とは微妙に異なる定義をしているサービスやシステムもあります。あまり突き詰めず、ガイドラインくらいに考えてください。

DELETE と OPTIONS だけは、HTTP メソッドと関数名が異なるので注意してください。

HTTP/1.1 から 3 までの共通仕様である「RFC 9110: HTTP Semantics」の第 9 章には GET、HEAD、POST、PUT、DELETE が定義されています。次の URL から閲覧できます。

```
https://www.rfc-editor.org/info/rfc9110
```

PATCH は追加の仕様で定義されており、メインの HTTP の仕様の一部ではありません。つまり、一般的な Web サーバにはなくてもよいのですが、REST で部分更新ができないのは不都合なので、REST サーバにはまず用意されています。PATCH の仕様については、次に URL を示す「RFC 5789:

PATCH Method for HTTP」を参照してください。

```
https://www.rfc-editor.org/info/rfc5789
```

あまり使われないものの、Microsoft や SAP 方面でときおり目にする MERGE メソッドは、たいていはほぼ PATCH と同じ扱いと考えられています。アプリケーション層プロトコルとして HTTP の一部ではなく、WebDAV（Web Distributed Authoring and Versioning）の仕様（RFC 3253）なので、無理に対応する必要はありません。Restify も、MERGE のルーティング設定メソッドは用意していません。

Node.js ネイティブの HTTP モジュールはもっと懐が広く、その METHODS 定数配列には MERGE などいろいろなメソッドが収容されています。Node.js のインタラクティブモード（REPL）から確認します。

```
> require('node:http').METHODS
[
  'ACL',          'BIND',        'CHECKOUT',    'CONNECT',     'COPY',
  'DELETE',       'GET',         'HEAD',        'LINK',        'LOCK',
  'M-SEARCH',     'MERGE',       'MKACTIVITY',  'MKCALENDAR',  'MKCOL',
  'MOVE',         'NOTIFY',      'OPTIONS',     'PATCH',       'POST',
  'PROPFIND',     'PROPPATCH',   'PURGE',       'PUT',         'REBIND',
  'REPORT',       'SEARCH',      'SOURCE',      'SUBSCRIBE',   'TRACE',
  'UNBIND',       'UNLINK',      'UNLOCK',      'UNSUBSCRIBE'
]
```

全部で 34 個あります。ちなみに、インターネット（IETF）が現在定義している HTTP メソッドは全部で 38 個です。リストは、次に URL を示す IANA の「Hypertext Transfer Protocol (HTTP) Method Registry」から確認できます。

```
https://www.iana.org/assignments/http-methods/http-methods.xhtml
```

ちなみに、Restify の兄貴分にあたる Express.js には、任意のメソッドと指定のエンドポイントのルーティングを構成する app.all() メソッドがありますが、Restify にはありません。