

→



# Herokuの代替となり得るPaaS「Render」で簡単なToDoアプリを作って遊んでみた (Nestjs,Nextjs)

slug todo-app-with-render

published false (下書き)

type tech (技術記事)

topics Render TypeScript Nextjs Nestjs Postgres

## はじめに

! この記事はSun\* Advent Calendar 2022の9日目の記事です。

株式会社 Sun Asterisk に所属する森真輝人です。社内では Web エンジニアとして働いていて Heroku を用いたシステム開発を行っています。

## 本記事の概要

Heroku はアプリケーションをホスティングする PaaS として有名ですが残念なことに 2022 年 11 月 28 日で無料プランが終了してしまいました。現在の仕事には影響はないものの今後どんな PaaS が来るのかなといろいろ探しているところ Render を見つけました。

この記事では簡単な ToDo アプリを作成して Render にデプロイする中でメリット・デメリットがお話しできればと思います。

## 作成した ToDo アプリ

アプリは[こちら](#)にデプロイしました。アプリの見た目はこんな感じです（ほんとに簡素ですね）。その代わりフロントエンド、バックエンド、データベースはすべて無料でホスティングでき

ています。無料プランで作成したので15分アクセスがないとサーバが停止します。停止した状態でアクセスするとサーバの再起動に時間がかかるので気を付けてください。

## Please Add Your Task!! (Task limit is 10)

New Task



Advent Calendar 2022



Sun Asterisk



ToDo App on Render

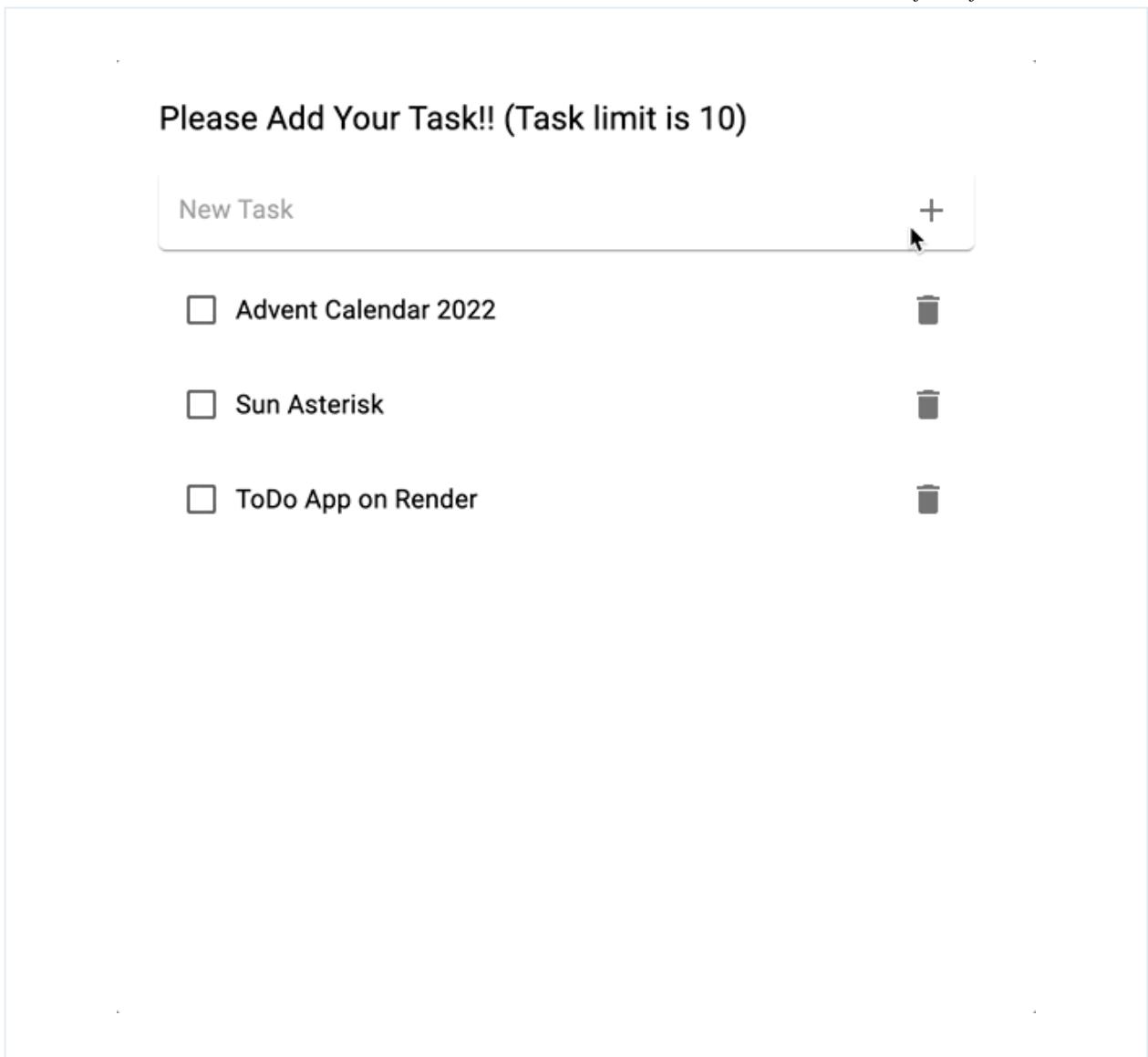


ToDo アプリとしての機能は

- タスク追加（10件まで）
- タスク削除（物理削除）
- タスク完了と未完了の切り替え

と少なめですがぜひ遊んでみてください（エラーハンドリングはしてないので不具合が起こっても許してください）。

▼ ToDo アプリを実際に動かしてみた様子（GIF）



## 利用した技術スタック

ToDo アプリを作成するにあたって主に利用したライブラリとそのバージョンは以下の通りです。

- Node.js : v16.13.1
- バックエンド（リポジトリは[こちら](#)）

ライブラリ	バージョン
@nestjs/cli	^8.0.0
prisma	^4.7.1
typescript	^4.3.5

- フロントエンド（リポジトリは[こちら](#)）

ライブラリ	バージョン
next	13.0.6
react	18.2.0
typescript	4.9.4
@mui/icons-material	^5.10.16

- データベース

種類	バージョン
PostgreSQL	14

PostgreSQL のバージョンを 14 にしたのは Heroku で対応しているバージョンが 14 だったので Render でもなんとなく 14 にしました。Render では 15 まで使うことができます。

## Render とは

！ 今回紹介している PaaS は「Render」なのだろうか。それとも「render」なのだろうか。  
「Render」の方がかっこいいので本記事では「Render」と書いてます。

Render は Web アプリケーション、ウェブサイト、データベース、クーロンなどを作成することができるプラットフォームです。Render は Ruby、Java、Node.js、PHP などさまざまな言語に対応しています。

公式サイトで[Heroku VS Render](#)なるページがありそこでは

We've built Render to help developers and businesses avoid the cost and inflexibility traps of legacy Platform-as-a-Service solutions like Heroku. Our customers often tell us Render is what Heroku could have been. This page explains why so many former Heroku customers consider Render to be the best Heroku alternative.

日本語訳としては

私たちは、開発者や企業が Heroku のようなレガシーな Platform-as-a-Service ソリューションのコストや柔軟性の問題を回避できるように、Render を開発しました。私たちの顧客はしばしば、Render は Heroku がなり得たかもしれないものだと言います。このページでは、なぜ多くの元 Heroku ユーザーが Render を最高の Heroku の代替品と考えるのかを説明します。

となかなか挑戦的なことを言っています笑。個人的な感想としては正直 Heroku の方が使いやすい感はありますが Render にしかない機能もあります。Render にしかない機能も含め Render を使うメリットをいくつか紹介します。

## Render のメリット

ドキュメントを読んだり、アプリを作っていく中で感じた Render を使うメリットを 5 つ紹介します。

### 守備範囲の広さ

まず対応言語が Node、Python、Ruby、Elixir、Go、Rust,PHP と人気どころは抑えられているのではないか？。また Docker、静的なサイトの作成、クーロン、バックグラウンドジョブなどにも対応しているので特別なことをしなければ大体の Web サービスが Render だけで作れそうです。

## Quickstarts

Run your code in just a few clicks.

 <b>Node</b> Express Puppeteer Redwood	 <b>Docker</b> Elasticsearch MySQL Ackee	 <b>Static Sites</b> See all Create React App Jekyll Gatsby
 <b>Python</b> Django Flask Celery	 <b>Ruby</b> Rails Rails with Sidekiq Sinatra with Sidekiq	 <b>Elixir</b> Phoenix Elixir Cluster Phoenix Distillery
 <b>Go</b> Gin Beego Pgweb	 <b>Rust</b> Rocket Actix GraphQL	 <b>PHP</b> Laravel

### ダウンタイムなしでのデプロイ

ビルトが失敗してもアプリケーションがダウンすることはない Render は謳っています。

### GitHub との連携機能

GitHub（や GitLab）と連携して特定のブランチ（main ブランチなど）へのプッシュをトリガーに自動デプロイできます。これに関しては Render に限らず Heroku や Netlify などでも存在する機能ですね。Render ではダッシュボードから自動デプロイをオフにできます。

おそらく Render 特有と思われますがおもしろい連携機能が 2 つありました。

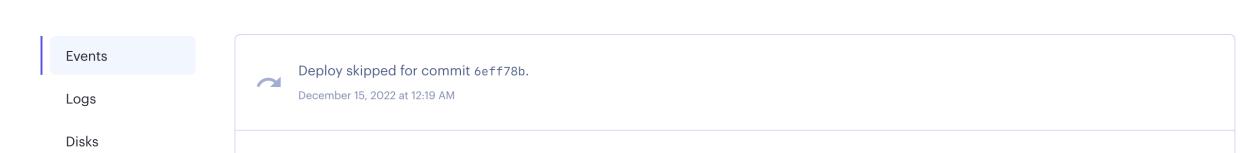
1. コミットメッセージに特定の文言を入れることで自動デプロイをスキップできる
2. ダッシュボードから特定のコミットの状態にロールバックできる

まず 1 つ目の「コミットメッセージに特定の文言を入れることで自動デプロイをスキップできる」の方ですが Git のコミットメッセージに [`<KEYWORD> skip`] か [`skip <KEYWORD>`] が含まれていると自動デプロイがスキップされます。KEYWORD は

- render
- deploy
- cd

の 3 つのうちの 1 つを使ってください。たとえばコミットメッセージを [render skip] Update README として GitHub にプッシュすれば自動デプロイはスキップされます。

自動デプロイがスキップされたかどうかはダッシュボードからも確認できます。Deploy skipped for ... とありますね。



README.md だけを更新してデプロイの必要がない時に使えそうですね。

**How Deployes Work | Render · Cloud Hosting for Developers**

Render makes deploying your application as easy as pushing your co...

[render.com](https://render.com)

The easiest cloud for developers and startups

次に 2 つ目の「ダッシュボードから特定のコミットの状態にロールバックできる」の方ですが Rollback to this deploy をクリックすればロールバックできます。間違えてデプロイしてしまった時に特定の時点の状態へ戻したい時に使えますね。

The screenshot shows the deployment history on the Render platform. It lists five successful deployments:

- Deploy live for 040bcf8: タスクは10個以上登録させない** (December 13, 2022 at 11:57 PM) - Includes a green checkmark icon and a "Rollback to this deploy" button.
- Deploy started for 040bcf8: タスクは10個以上登録させない** (New commit, December 13, 2022 at 11:52 PM) - Includes a cloud icon and a "Rollback to this deploy" button.
- Deploy live for 8ad3815: CRUDのエンドポイント作成** (December 12, 2022 at 9:55 PM) - Includes a green checkmark icon and a "Rollback to this deploy" button.
- Deploy started for 8ad3815: CRUDのエンドポイント作成** (New commit, December 12, 2022 at 9:51 PM) - Includes a cloud icon and a "Rollback to this deploy" button.
- Deploy live for 13658e4: タスク全取得のエンドポイント追加** (December 12, 2022 at 9:35 PM) - Includes a green checkmark icon and a "Rollback to this deploy" button.

## 無料プランでも PostgreSQL に IP 制限ができる

Heroku で DB に IP 制限をかけようと思うと割と高額なエンタープライズプランを契約しないといけません。Render では無料プランでも IP 制限ができるのは嬉しいですね。

The screenshot shows the "Access Control" settings page on Render. It includes the following elements:

- Access Control** heading
- A note: **1 IP range is allowed from outside of your private network.**
- A note: **Sources are specified CIDR block notation.**
- Source** input field: `e.g. 0.0.0.0/0` (highlighted with a blue border) and a `Use my IP address` button.
- Description** input field: A large text area with a red trash bin icon in the top right corner.
- Test an IP address** input field with a help icon ( ⓘ ) and a red trash bin icon.
- + Add source** button.
- Cancel** and **Save** buttons at the bottom right.

IP 制限もダッシュボードから簡単に行えます。自分の環境からのみアクセスできるようにしたい時は `Use my IP address` をクリックすれば自動で入力されるので地味に嬉しい機能です。

## インフラのコード化

いわゆる Infrastructure as Code(IaC) が実現できます。`render.yaml` を作成して GitHub にプッシュするとその設定を読み込みインフラを構築してくれるようです。

## Render のデメリット

デメリットも感じたので 3 つ紹介します。

### デプロイが割と遅い

無料プランの場合だけかもしれませんがデプロイが割と遅いです。ダッシュボードに「デプロイは遅いですか？アップグレードしますか？」的な文章が目に入ります笑。今回の作成した Next、Nest ともにデプロイ完了まで 3 分～5 分くらいかかります。同じくらいのサイズのアプリを Heroku にデプロイする際は無料プランでも 1 分もかかってなかったのでここは Heroku に軍配が上がります。

### 無料プランのインスタンスはアクセスがないと停止する

無料プランの Web Service（アプリをホスティングするサーバ）の説明で

Web Services on the free instance type are automatically spun down after 15 minutes of inactivity. When a new request for a free service comes in, Render spins it up again so it can process the request. This can cause a response delay of up to 30 seconds for the first request that comes in after a period of inactivity.

とありました。日本語訳は

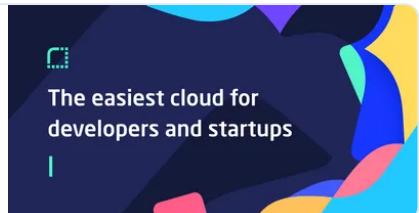
フリーインスタンスタイプの Web サービスは、15 分間使用されないと自動的にスピンダウンされます。無料サービスに対する新しいリクエストが来ると、Render はリクエストを処理できるよう再びスピナップします。このため、一定期間使用されていない状態で最初に来たリクエストに対して、最大で 30 秒の応答遅延が発生する可能性があります。

です。15 分アクセスがないとそのアプリは止まっちゃうってことですね。リクエストが来たら再起動して最大 30 秒の応答遅延があると書いてますがそんなに短くないです。少なくとも数分は待つ必要がありました。

#### Free Instance Types | Render · Cloud Hosting for Developers

With Render's free instance types you can spin up Web Services and...

 [render.com](https://render.com)



Render で対応しているリージョンが

- Oregon, USA
- Frankfurt, Germany
- Ohio, USA
- Singapore

の 4 つで日本に近いところがないのが遅延の原因かもしれません。

## npm が使えない(かもしれない)

デメリットでもないですが私が操作した限り Render のダッシュボードで指定するビルドコマンドとスタートコマンドに npm が使えませんでした。  
ビルドコマンドに関するドキュメントには例として yarn が使われていましたが npm が使えないとは書いてなかったのでもしかしたら npm も使えるかもしれません。

# ToDo アプリ作成

---

以下の 3 本立てでアプリ作成の流れを説明します。

1. データベース編（Postgres）
2. バックエンド編（Nestjs）
3. フロントエンド編（Nextjs）

## データベース編(Postgres)

ダッシュボードのヘッダーにある「New+」から「PostgreSQL」を選択します。

The screenshot shows the Render web interface. At the top, there are navigation links for 'Community' and 'Help', and a blue button labeled 'New +'. On the left, there's a section for 'PostgreSQL' with a blue elephant icon, describing Render's managed PostgreSQL service with various connectivity options and backup features, and a 'Learn more.' link. On the right, a sidebar lists service types: 'Static Site', 'Web Service', 'Private Service', 'Background Worker', 'Cron Job', 'PostgreSQL' (which is highlighted with a blue background), 'Redis', and 'Blueprint'.

次にデータベースの名前などを入力し、スペックを選択して「Create Database」をクリックするとデータベースが作成されます。

**New PostgreSQL**

Name

Database  randomly generated unless specified

User  randomly generated unless specified

Region  
The region where your Database runs. Services must be in the same region to communicate privately and you currently have services running in Singapore.  
 Singapore (Southeast Asia)

PostgreSQL Version  14

Datadog API Key

Please enter your payment information to select a plan with higher limits.

Instance Type	RAM	CPU	Storage	Price
<input checked="" type="radio"/> Free	256 MB	Shared	1 GB	\$0 / month
<input type="radio"/> Starter	256 MB	Shared	1 GB	\$7 / month
<input type="radio"/> Standard	1 GB	1 CPU	16 GB	\$20 / month
<input type="radio"/> Pro	4 GB	2 CPU	96 GB	\$95 / month
<input type="radio"/> Pro Plus	8 GB	4 CPU	256 GB	\$185 / month

Need a custom plan? We support up to 512 GB RAM, 64 CPUs, and 5 TB storage.

i Free databases will expire in 90 days and will be deleted if not upgraded. Learn more about [free instance type limits](#).

**Create Database**

Feedback Invite a Friend Contact Support

なお、無料プランではデータベースは1つしか作れないので気をつけてください。

データベースが作成できたらダッシュボードの Info の欄から接続情報を確認してください。

「Internal Database URL」の値は次のバックエンド編で使います。

## Connections

Hostname ⓘ dpg-ceat9kda4996mecgcj40-a

Port 5432

Database nest\_prisma

Username nest\_prisma\_user

Password



Internal Database URL



External Database URL



## バックエンド編(Nestjs)

言語のフレームワークは [Nestjs](#)、OR マッパーは [Prisma](#)を使います。

バックエンドの構築はこちらを参考にしました。

### Build a REST API with NestJS, Prisma, PostgreSQL and Swagger

Learn how to build a backend REST API with NestJS, Prisma, Postgres...

[www.prisma.io](https://www.prisma.io)



## テンプレート作成

NestCLI でテンプレートを作成します。 -p のオプションでパッケージマネージャーを yarn に指定できます。

```
npx @nestjs/cli new render-app-nest -p yarn
```

Docker コンテナの上に PostgreSQL を立てます。 docker-compose.yaml は以下のようにします。

```
version: "3.8"
services:
  postgres:
    image: postgres:14
    restart: always
```

**environment:**

```
POSTGRES_USER: postgres
POSTGRES_PASSWORD: mypassword
POSTGRES_DB: render-postgres
```

**volumes:**

```
- postgres:/var/lib/postgresql/data
```

**ports:**

```
- 5432:5432
```

**volumes:**

```
postgres:
```

データベース編で PostgreSQL のバージョンは 14 にしたので yaml ファイルでもバージョンは 14 にしてください。

最後に以下のコマンドを実行してデータベースを起動してください。

```
docker-compose up
```

バックグラウンドで起動したい場合は以下のように -d をつけて実行してください。

```
docker-compose up -d
```

## Prisma の導入

OR マッパーの Prisma の設定を行います。まずは Prisma のインストールと初期化を行います。

```
npm i -D prisma
npx prisma init
```

このコマンドを実行すると .env ファイルが作成されています。docker-compose.yaml で指定した値を元に .env ファイルの DATABASE\_URL の部分を

```
DATABASE_URL="postgresql://postgres:mypassword@localhost:5432/render-postgres?sslmode=prefer"
```

と書き換えてください。

次にスキーマを作成します。テーブルとそのテーブルが持つカラムを決めます。

```
prisma/schema.prisma
```

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

+ model Task {
+   id        Int      @id @default(autoincrement())
+   content   String
+   done      Boolean  @default(false)
+   createdAt DateTime @default(now())
+   updatedAt DateTime @updatedAt
+ }
```

- タスクの内容: content
- タスクの完了フラグ: done

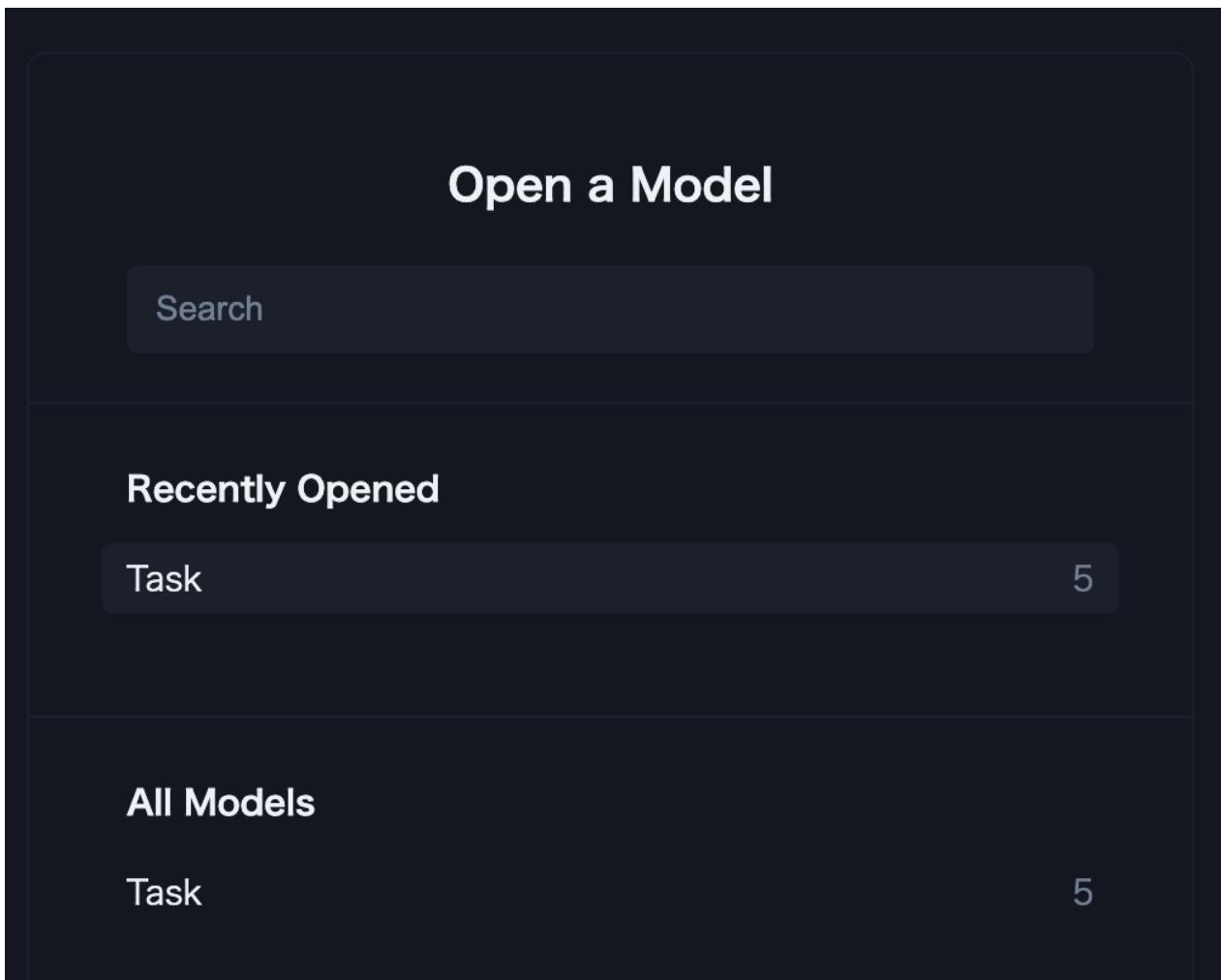
としています。タスクを生成した時、タスクは未完了なので done はデフォルトで false としておきました。今回の ToDo アプリに編集機能はないですがないと気持ち悪いので更新日のカラム updatedAt もつけています。

スキーマができたので次はマイグレーション（テーブルの作成）を行います。

```
npx prisma migrate dev --name "init"
```

実際にテーブルが作成されたかブラウザで確認するために Prisma が用意している Prisma Studio を起動します。

```
npx prisma studio
```



Task が作成されていれば OK です。レコードが入っていなければ 5 ではなく 0 と表示されています。

次にレコードを 2 件追加するコードを作成します。

```
▼ seed.ts

prisma/seed.ts

import { PrismaClient } from '@prisma/client';

// Prismaクライアントの初期化
const prisma = new PrismaClient();

async function main() {
    // ダミーデータの作成
    const task1 = await prisma.task.upsert({
        where: { id: 1 },
        update: {},
        create: {
            content: 'Prisma Adds Support for MongoDB',
        },
    });
}
```

```
Herokuの代替となり得るPaaS「Render」で簡単なToDoアプリを作つて遊んでみた（Nestjs,Nextjs）のレビュー
const task2 = await prisma.task.upsert({
  where: { id: 2 },
  update: {},
  create: {
    content: 'アドベントカレンダー',
  },
});

console.log({ task1, task2 });
}

main()
  .catch((e) => {
    console.error(e);
    process.exit(1);
  })
  .finally(async () => {
    // Prismaクライアントのクローズ
    await prisma.$disconnect();
  });
}
```

このコードを実行するために package.json を編集し

package.json

```
"scripts": {
  // ...
},
+ "prisma": {
+   "seed": "ts-node prisma/seed.ts"
+ }
```

次のコマンドを実行するとレコードが 2 件追加されます。

npx prisma db seed

コマンドを実行して Prisma Studio で以下のようにレコードが 2 件追加されていることを確認してください。

Task					
	Filters	None	Fields	All	Showing 2 of 2
	id #	content A	done	createdAt	updatedAt
	1	Prisma Adds Support for ...	false	2022-12-12T12:21:47.6...	2022-12-12T12:21:47.6...
	2	アドベントカレンダー	false	2022-12-12T12:21:47.7...	2022-12-12T12:21:47.7...

最後に Prisma のサービスを作成します。Prisma のサービスでは Prisma クライアントインスタンスの作成とデータベースへの接続を行います。

Prisma のサービスと Prisma サービスをエクスポートするためのモジュールを作成します。

```
npx nest generate module prisma
npx nest generate service prisma
```

サービスでは Graceful shutdown を行うメソッドを記載します。

```
src/prisma/prisma.service.ts

import { INestApplication, Injectable } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient {
    //Graceful shutdownを行う
    async enableShutdownHooks(app: INestApplication) {
        this.$on('beforeExit', async () => {
            await app.close();
        });
    }
}
```

モジュールは以下のようにして他のモジュールから Prisma サービスを使えるようにします。

```
src/prisma/prisma.module.ts

import { Module } from '@nestjs/common';
import { PrismaService } from './prisma.service';

@Module({
    providers: [PrismaService],
    exports: [PrismaService],
```

```
})
export class PrismaModule {}
```

## エンドポイントの作成

データの投入ができたので次はデータの CRUD 様式を行うためのエンドポイントを作成するためには以下のコマンドを実行します。

```
npx @nestjs/cli generate resource
```

いくつか質問されるので

1. What name would you like to use for this resource (plural, e.g., "users")? **tasks**
2. What transport layer do you use? **REST API**
3. Would you like to generate CRUD entry points? **Yes**

と答えてください。これで CRUD のエンドポイントが作成されました。

Prisma のサービスを用いてデータの作成、取得、削除を行いたいので以下のようにしてください。

```
src/tasks/tasks.service.ts

@Injectable()
export class TasksService {
  constructor(private prisma: PrismaService) {}

  //新規作成
  create(createTaskDto: CreateTaskDto) {
    return this.prisma.task.create({ data: createTaskDto });
  }

  //全件取得
  findAll() {
    return this.prisma.task.findMany();
  }

  //削除
  remove(id: number) {
    return this.prisma.task.delete({ where: { id } });
  }
}
```

```
    }  
}
```

なおタスクの DTO は以下の通りです。

```
src/tasks/dto/create-task.dto.ts
```

```
export class CreateTaskDto {  
    //タスクの内容  
    content: string;  
  
    //タスクを完了したか  
    done: boolean = false;  
}
```

最後に全件取得だけ動作確認します。

```
yarn start:dev
```

で NestJS を起動して <http://localhost:3000/tasks> ヘアクセスした時に登録したレコードが 2 件返ってきていれば OK です。

```
[  
{  
    id: 1,  
    content: "Prisma Adds Support for MongoDB",  
    done: false,  
    createdAt: "2022-12-12T12:21:47.670Z",  
    updatedAt: "2022-12-12T12:21:47.670Z",  
},  
{  
    id: 2,  
    content: "アドベントカレンダー",  
    done: false,  
    createdAt: "2022-12-12T12:21:47.707Z",  
    updatedAt: "2022-12-12T12:21:47.707Z",  
},  
];
```

## Render で Nestjs のコードをデプロイ

作成した Nestjs のコードを Render にデプロイします。Render のダッシュボードから「Web Service」を作成します。

The screenshot shows the Render dashboard interface. At the top, there are navigation links for 'Community' and 'Help', and a blue 'New +' button. On the left, there's a large card for 'Web Service' with a globe icon, a title, and a description: 'Web services are kept up and running at all times, with native SSL and HTTP/2 support. Add a persistent disk or custom domain. Scale up and down with ease.' Below this is a 'Learn more.' link. To the right of this card is a vertical list of service types: 'Static Site', 'Web Service' (which is highlighted with a blue background), 'Private Service', 'Background Worker', 'Cron Job', 'PostgreSQL', 'Redis', and 'Blueprint'. The 'Web Service' item is the current selection.

次に「Connect GitHub」をクリックして Render と GitHub を連携します。

## Create a new Web Service

Connect your Git repository or use an existing public repository URL.

**Connect a repository**

Connect your Render account to GitHub or GitLab to begin using your existing repos for new services.

+ Connect GitHub

+ Connect GitLab

GitHub

+ Connect account

GitLab

+ Connect account

次にデプロイのための設定を行います。

You are deploying a web service for [render-nest-app](#).

**Name**  
A unique name for your web service.

**Region**  
The [region](#) where your web service runs. Services must be in the same region to communicate privately and you currently have services running in [Singapore](#).

**Branch**  
The repository branch used for your web service.

Region は日本から一番近い「Singapore (Southeast Asia)」を選択しました（日本のリージョンもいくつか出てほしいなあ）。

**Environment**  
The runtime environment for your web service.

**Build Command**  
This command runs in the root directory of your repository when a new version of your code is pushed, or when you deploy manually. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app.

**Start Command**  
This command runs in the root directory of your app and is responsible for starting its processes. It is typically used to start a webserver for your app. It can access environment variables defined by you in Render.

Nestjs で作成したので Environment は Node を選択してください。

Build Command はマイグレーションとデータの登録を行うので少し長くなりますが

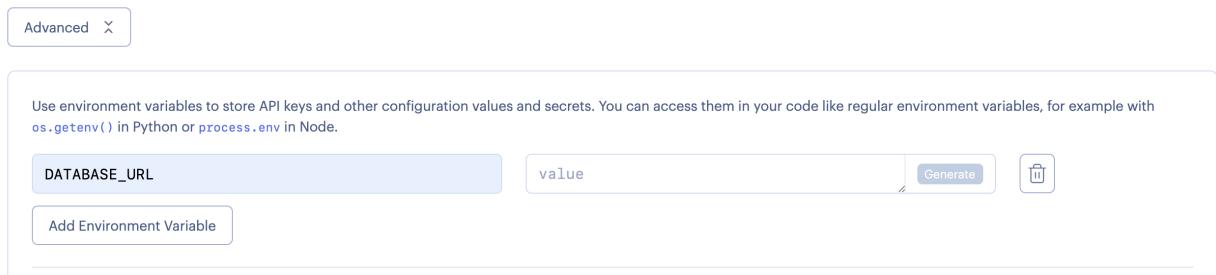
```
yarn && yarn build && npx prisma migrate deploy && npx prisma db seed
```

とします。

Start Command は

```
yarn start:dev
```

とします。これらのコマンドはすべて package.json で設定した script を拝借しています。最後に Advanced のところで環境変数を設定します。



- キー：DATABASE\_URL
- バリュー：データベースを作成した時に生成された接続情報の「Internal Database URL」の値

として「Create Web Service」をクリックすればデプロイが完了します。今後 main ブランチにプッシュすればそれをトリガーにして自動デプロイされます。

## フロントエンド編(Nextjs)

言語のフレームワークは Nextjs、HTTP クライアントは Axios、CSS フレームワークは MUI を使います。

### テンプレート作成

Create Next App でテンプレートを作成します。

```
npx create-next-app render-app-next --ts
```

Axios と MUI をインストールします。

```
yarn add axios  
yarn add @mui/material @emotion/react @emotion/styled  
yarn add @mui/icons-material
```

## API Routes 作成

なんどなくバックエンドのエンドポイントはブラウザから確認されたくないのでバックエンドのリクエストは API Routes 経由で行います。

pages/api/task/index.ts

```
type Task = {
  id: number;
  content: string;
  done: boolean;
  createdAt: Date;
  updatedAt: Date;
};

//レスポンスに必要な型
type Data = Task[] | Task;

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse<Data>
) {
  const { method, body } = req;
  switch (method) {
    case "GET":
      // Get data from your database
      const options: AxiosRequestConfig = {
        url: `${process.env.HOST}/tasks`,
        method: "GET",
      };
      const r: AxiosResponse<Task[]> = await axios(options);
      const { data } = r;
      res.status(200).json(data);
      break;

    case "POST":
      // Create data
      const optionsPost: AxiosRequestConfig = {
        url: `${process.env.HOST}/tasks`,
        method: "POST",
        data: body,
      };
      const r2: AxiosResponse<Task> = await axios(optionsPost);
  }
}
```

```

const { data: dataPost } = r2;
res.status(200).json(dataPost);
break;

default:
  res.setHeader("Allow", ["GET", "PUT"]);
  res.status(405).end(`Method ${method} Not Allowed`);
}
}

```

リクエストのメソッドで処理を分岐させています。

- GET:全件取得
- POST:新規作成

としています。

削除については以下の通りです。

pages/api/task/[id].ts

```

export default async function taskHandler(
  req: NextApiRequest,
  res: NextApiResponse<number>
) {
  // [id]は動的
  const {
    query: { id },
    method,
    body,
  } = req;
  switch (method) {
    case "DELETE":
      // Delete data in your database
      const optionsDelete = {
        url: `${process.env.HOST}/tasks/${id}`,
        method: "DELETE",
        headers: { "Accept-Encoding": "gzip,deflate,compress" },
      };
      const r3: AxiosResponse<number> = await axios(optionsDelete);
      const { data: deletedId } = r3;
      res.status(200).json(deletedId);
  }
}

```

```

break;

default:
  res.setHeader("Allow", ["DELETE"]);
  res.status(405).end(`Method ${method} Not Allowed`);

}
}

```

[id] はタスクを特定するための ID です。削除のリクエストを送る際にどのタスクかを判定するために利用しています。

## View 部分

いかんせん長いのでアコーディオンにしました。ざっくり分けると以下の通りです。

- 前半部分：タスクの作成、削除、完了・未完了ロジックの記述
- 後半部分：MUI でフォームの作成

### ▼ index.tsx

```

pages/index.tsx

type Task = {
  id: number;
  content: string;
  done: boolean;
  createdAt: Date;
  updatedAt: Date;
};

type Props = {
  staticTasks: Task[];
};

const Example: NextPage<Props> = ({ staticTasks }) => {
  const [tasks, setTasks] = useState(staticTasks);
  const [newTask, setNewTask] = useState("");
  const [deletedTaskId, setDeletedTaskId] = useState(-1); //削除ボタンを押したタスクを

  //タスクの作成
  const handleSubmit = async (e: SyntheticEvent) => {
    e.preventDefault();

    const optionsPost: AxiosRequestConfig = {
      url: "api/task",

```

Herokuの代替となり得るPaaS「Render」で簡単なToDoアプリを作つて遊んでみた（Nestjs,Nextjs）のプレビュー

```

method: "POST",
data: { content: newTask },
};

const { data } = await axios(optionsPost);
if (!data) return;
setTasks((prev) => {
  return [...prev, data];
});

//タスクの入力フォームをクリア
setNewTask("");
};

//タスクの削除
const handleDelete = async (id: number) => {
  const options: AxiosRequestConfig = {
    url: `api/task/${id}`,
    method: "DELETE",
    // headers: { "Accept-Encoding": "gzip,deflate,compress" },
  };
  const { data } = await axios(options);
  setTasks((prev) => {
    const newTasks = prev.filter((task) => {
      return task.id !== data.id;
    });
    return newTasks;
  });
};

//タスクの完了・未完了のトグル
const handleTaskStatus = (e: React.ChangeEvent<HTMLInputElement>) => {
  setTasks((prevTasks) => {
    const newTasks = prevTasks.map((task) => {
      const selectedTaskId = Number(e.target.value);
      if (task.id === selectedTaskId) {
        return { ...task, done: !task.done };
      }
      return task;
    });
    return newTasks;
  });
};

return (
  <Box sx={{ flexGrow: 1, maxWidth: 500, mx: "auto" }}>
    <Grid container spacing={0}>
      <Grid item xs={12} md={12}>
        <Typography sx={{ mt: 4, mb: 2 }} variant="h6" component="div">
          Please Add Your Task!! (Task limit is 10)
        </Typography>
      </Grid>
    </Grid>
  </Box>
);

```

```

<form onSubmit={handleSubmit}>
  <Paper sx={{ p: "2px 4px", display: "flex", alignItems: "center", }}>
    <>
      <InputBase sx={{ ml: 1, flex: 1 }} placeholder="New Task" inputProps={{ "aria-label": "new task" }} value={newTask} onChange={(e) => { setNewTask(e.target.value); }} />
      <IconButton type="button" sx={{ p: "10px" }} aria-label="search" onClick={handleSubmit}>
        <AddIcon />
      </IconButton>
    </>
  </Paper>
  <List>
    {tasks.map((task) => {
      return (
        <ListItem key={task.id}>
          <FormControlLabel control={
            <Checkbox checked={task.done} value={task.id} onChange={handleTaskStatus} />
          }
          label={task.content}
          sx={{
            textDecoration: task.done ? "line-through" : "auto",
            wordBreak: "break-word",
            flex: 1,
            mr: 0,
          }}
        </>
        <Box sx={{ ml: "auto", mr: "4px" }}>
          <IconButton edge="end" aria-label="delete">

```

```

        onClick={() => {
          setDeletedTaskId(task.id);
          handleDelete(task.id);
        }}
        disabled={task.id === deletedTaskId}
      >
    <DeleteIcon />
  </IconButton>
</Box>
</ListItem>
);
)
}
</List>
</form>
</Grid>
</Grid>
</Box>
);
};

export default Example;

export const getServerSideProps: GetServerSideProps = async () => {
  const res = await axios.get(`process.env.HOST/tasks`, {
    headers: { "Accept-Encoding": "gzip,deflate,compress" },
  });

  const data = res.data;
  return {
    props: {
      staticTasks: data,
    },
  };
}

```

## R

最後に作成した Nextjs のコードを Render にデプロイします。Nestjs の場合とほぼ同じです。大まかには以下のような流れです。

1. Web Service の作成
2. 連携先の GitHub リポジトリを選択
3. Build Command の設定

```
yarn && yarn build
```

#### 4. Start コマンドの設定

```
yarn start
```

#### 5. 環境変数の設定

バックエンドヘリクエストするために必要な HOST の値を設定します。

- キー：HOST

として「Create Web Service」をクリックすればデプロイされます。これで ToDo アプリ作成のすべての工程が完了しました！

## まとめ

2022年11月28日で Heroku の無料プランが終了してしまいました。これを機に ToDo アプリの作成を通じて Render を触ってみました。

メリット、デメリットは以下の通りです。

- メリット
  - 守備範囲の広さ
  - ダウンタイムなしでのデプロイ
  - GitHub との連携機能
  - 無料プランでも PostgreSQL に IP 制限ができる
  - インフラのコード化
- デメリット
  - デプロイが割と遅い
  - 無料プランのインスタンスはアクセスがないと停止する
  - npm が使えない（かもしれない）

デプロイが遅かったりデメリットはありますが、GUI も見やすくてできることも多いので個人的には好印象です。日本にもリージョンを作ってくれることを切に願っています笑

**Sun\* Advent Calendar 2022**、明日 12/22 は弊社 Sun\*エンジニア Trong Tran Ba さんの「自然言語  
処理・機械学習を実装した事例」をお送りいたします！