1.1 Vue と Nuxt の関係

本書で解説する Nuxt は、JavaScript によるフロントエンド開発のフレームワークである Vue をベースに、さまざまな機能を付け加えたフレームワークです。本節では、その Nuxt は Vue にどのような機能が付け加わったのか、Nuxt の特徴は何かを紹介します。

1.1.1 フロントエンドフレームワーク Vue

Nuxt をこれから習得しようという本書の読者諸兄姉は、すでに **Vue** がどのようなフレームワークなのかはご存じと思いますが、ここで簡単に紹介することにします。Vue の本体となる **Vue.js** が、Evan You によってリリースされたのが 2014 年です。その Vue.js は、JavaScript コード上の変数と DOM 要素が自動的に連動する**リアクティブシステム**をコアに含みながらも、軽量であり、それゆえに軽快に動作するフレームワークでした。そのためか、人気を博すようになり、順当にアップデートを重ねていきます。

そして、2020 年 9 月にメジャーアップデートであるバージョン 3 がリリースされます。ただし、このアップデートは Vue.js 本体のみであり、それに追随する主要なモジュール類がバージョン 3 に対応できておらず、遅れるところ約 1 年半、2022 年 2 月 7 日に、Vue.js 本体のみならず、主要なモジュールも含めて、Vue を作成するプロジェクトのデフォルトバージョンが 3 になりました。

このような新しい Vue プロジェクトに含まれる主なモジュールを列挙すると、次の通りです。

- シングルページアプリケーションを手軽に実現できる **Vue Router**
- コンポーネント間横断でデータを管理できる**ステート管理**モジュールとしての **Pinia**
- Vue のユニットテストを容易にできる **Vitest** と **Vue Test Utils**

さらに、これらのプロジェクト内のソースコードをひとまとめにして、実行できるファイル類へと変換してくれるツール、つまり、プロジェクトのビルドツールとして **Vite** が、そのプロジェクトの根幹となっています。

1.1.2 Vue 3 の特徴

では、これらのモジュール類が Vue.js のバージョン 3 に対応するまでに、なぜこれほどの時間がかかったのかというと、それがそのまま、Vue.js のバージョン 3 の特徴とバージョン 2 からの変化の大きさを物語っています。以下、主要な特徴を 3 点紹介します。

TypeScript の採用

一番大きな変化と言えるのが、この **TypeScript** の採用です。Vue.js のバージョン 3 をリリースするにあたり、Evan You は内部コードを、それまでの JavaScript から、全て TypeScript へと書き換えています。そのおかげで、Vue 3 プロジェクトは、TypeScript コーディングを標準でサポートするようになります。TypeScript の型システムをフル活用したコーディングが可能となり、型安全なアプリケーション作成が行えるようになりました。

これは、逆からみると型厳密を意識せずに Vue.js と連携させてきたモジュールはことごとく対応できないか、かなりの大改造をしなければならないことになります。

その最も典型的な例が、ステート管理モジュールです。Vue 2 では、**Vuex** というモジュールが利用されていましたが、この Vuex が最後まで TypeScript に対応できず、Vue 3 では、代わりに **Pinia** が開発、採用されるようになります（もちろんその他の理由もありますが）。

Composition API の採用

TypeScript の採用と同じくらいの大きな変化があるのが、**Composition API** とその簡略化した記述である **script setup** タグの標準採用です。Vue.js バージョン 2 では、コンポーネント内のさまざまなデータや処理を定義するにあたり、オブジェクトリテラルの形式で記述されていました。この記述方法を、**Options API** といいます。

そのような記述形式から、**setup()** というひとつの関数内にデータも処理も定義できるように変わり、この記述方法のおかげで、非常にスッキリしたコードが記述できるようになりました。さらに、script setup タグを採用することで、setup() 関数すらも記述する必要がなくなり、よりスッキリしたコードが記述できるようになりました。

Vite の標準採用

さらには、**Vite** の標準採用も大きな変化です。Vue は、バージョン 2 までのプロジェクト基盤として、**Webpack**[*1] を採用していました。この Webpack でも問題なく動作していたのですが、プロジェクトのビルドに時間がかかり、そのため、開発効率が落ちるという問題がありました。

その問題を解決するために、より早く動作する Vite が開発されました。この Vite は「ヴィート」と発音し、フランス語の「速い」の意味です。実際に、その名称通り、Webpack と比べてかなり軽快に動作するようになっています。

> **NOTE　Vite が速い理由**
>
> Vite が軽快に動作する理由は、開発段階では、複数のモジュールファイルをひとつにまとめる、すなわち、モジュールバンドルを行わないからです。ブラウザに、モジュールファイルをそのまま読み込ませて実行させるため、モジュールバンドルの処理時間が不要となり、高速に動作するようになっています。

＊1　複数の JavaScript モジュールファイルを、実行可能なひとつの JavaScript ファイルにまとめてくれるツールのひとつ。このようなツールを**モジュールバンドラ**といい、Webpack はモジュールバンドラのデファクトスタンダードとして利用されてきました。

2.3 ステートの利用

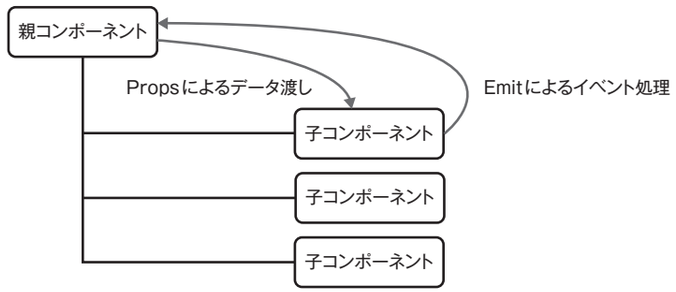
コンポーネント内の記述方法、コンポーネント間連携方法と紹介してきました。これまでの内容は、一部 Nuxt 独特の内容も含まれていますが、基本的には Vue 3 の内容そのままです。

本節でようやく本格的に Nuxt 独特の構文が登場します。それは、コンポーネントをまたいでデータを共有する方法です。

2.3.1 Props+Emit の問題点

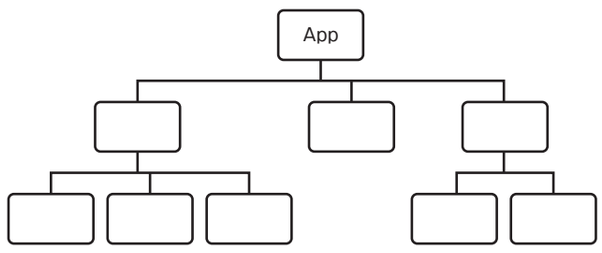
前節で紹介したコンポーネント間連携の方法を図としてまとめると、図 2-8 のようになります。

▼ **図 2-8　Props ダウン、イベントアップ**



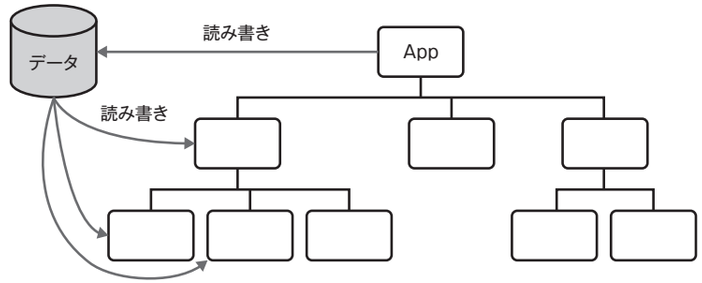
親コンポーネントから子コンポーネントへデータを渡す Props と、子コンポーネントから親コンポーネントのイベント処理を呼び出すことでデータを渡す Emit を合わせて、「Props ダウン、イベントアップ」といいます。しかし、この方法は、単純な親子関係では有効ですが、コンポーネント数が増え、構造が複雑になるとすぐに破綻します（図 2-9）。

▼ **図 2-9　Props ダウン、イベントアップだけでは難しい**



そこで、アプリケーション全体でデータを保持しておき、各コンポーネントはそこからデータを取得したり、そのデータを書き換えたりという仕組みが必要になります（図 2-10）。

▼ **図 2-10　アプリケーション全体でのデータ管理が必要**



Vue 単体では、これを実現する仕組みとして、**Provide** と **Inject** というのがあります。あるいは、より本格的にデータを管理できるモジュールとして、**Pinia** というものがあり、Vue プロジェクト作成時に追加することも可能です。

このように、コンポーネントをまたいでのデータ管理のことを、**ステート管理**といい、Pinia は、Vue 3 から新たに導入されたステート管理モジュールです。

そして、これらの機能は、そのまま Nuxt でも利用できます。一方、Nuxt には独自のステート管理の仕組みがあります。次に、それを紹介していきます。

2.3.2 Nuxt のステート管理を利用したサンプル作成

では、早速、Nuxt のステート管理の仕組みを利用したサンプルを作成しましょう。まず、新しいプロジェクトとして、state プロジェクトを作成してください。

このサンプルの動作は、components-emit と同じですが、コンポーネントをまたいでデータを管理できる様子が理解しやすいように、コンポーネントを増やします。また、インターフェース Member もコンポーネントをまたいで利用されるので、別ファイル化します。

まず、そのファイルとして、interfaces.ts をプロジェクト直下に作成し、リスト 2-27 の Member インターフェースをエクスポートするコードを記述してください。

▼ **リスト 2-27　state/interfaces.ts**

```
export interface Member {
  id: number;
  name: string;
  email: string;
  points: number;
  note?: string;
}
```

次に、リスト 2-28 の components/OneMember.vue ファイルを作成してください。

3.1 Nuxt ルーティングの基本

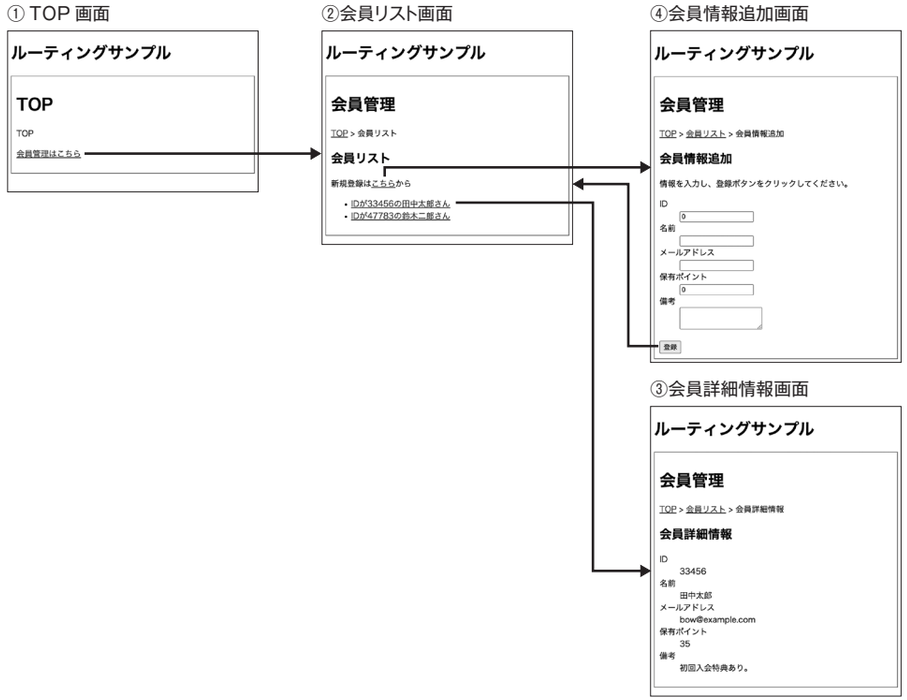
Vue アプリケーションで画面遷移、すなわち、ルーティングを行う場合、**Vue Router** を利用するのが通常です。Nuxt では、この Vue Router を内部に含みながら、より簡単に Vue Router を利用できる仕組みがあります。本節では、Nuxt でのルーティングの基本を紹介していきます。

3.1.1 本節のサンプルの概要

先述のように、Nuxt でのルーティングは、Vue Router を使いやすくしたものです。実際に、その内部では、Vue Router が動作しています。そのため、基本的な考え方は、Vue Router と同じであり、したがって、本書では、Vue Router を習得済みであることを前提に、その差分で紹介していくものとします。

その Nuxt ルーティングの解説する題材として、本節で作成するサンプルプロジェクトである routing-basic の概要をまず紹介しておきます。routing-basic の画面遷移を図にすると、図 3-1 のようになります。

▼ 図 3-1　routing-basic プロジェクトの画面遷移



画面番号と、画面名とその画面を表すリンクパス（URL のドメイン以降の部分）の対応関係は、表 3-1 の通りです。

▼ 表 3-1　routing-basic プロジェクトの画面

番号	画面名	リンクパス
①	TOP 画面	/
②	会員リスト画面	/member/memberList
③	会員詳細情報画面	/member/memberDetail/33456
④	会員情報追加画面	/member/memberAdd

まず、アプリケーションを表示させると、① TOP 画面が表示されます。そのため、この TOP 画面のリンクパスは、/（ルート）となっています。

その TOP 画面中の［会員管理はこちら］のリンクをクリックすると、②会員リスト画面が表示されます。その際のパスは、表 3-1 の通り、/member/memberList です。

会員リスト画面の各会員情報がリンクとなっており、それをクリックすると③会員詳細情報画面が表示されます。その際のパスは、表 3-1 では /member/memberDetail/33456 ですが、リンクパス末尾の「33456」はクリックする会員によって変化します。すなわち、**ルートパラメータ**です。

一方、②会員リスト画面の「新規登録はこちらから」の［こちら］リンクをクリックすると④会員情報追加画面が表示されます。その際のパスは、/member/memberAdd です。この画面に必要情報を入力し、［登録］ボタンをクリックすると、入力した会員情報が保存され、②会員リスト画面に戻ります。もちろん、その際、新たに保存された会員情報がリストに追加されています。

3.1.2 ルーティング表示領域を設定する NuxtPage タグ

このようなルーティングを行う場合、Vue Router では、ルーティング設定情報をファイルに記述する必要がありました。これが、Nuxt では不要です。では、どのようにルーティングを行うのか、その辺りを、実際にプロジェクトを作成しながら、解説していくことにします。

早速、routing-basic プロジェクトを作成し、2.3.2 項で作成した state プロジェクトの interfaces.ts（リスト 2-27）をファイルごと、routing-basic プロジェクト直下にコピー＆ペーストしてください。その上で、app.vue をリスト 3-1 の内容に書き換えてください。なお、スクリプトブロックは、同じく state プロジェクトの app.vue（リスト 2-30）と同じですので、省略しています。

▼ リスト 3-1　routing-basic/app.vue

```
<script setup lang="ts">
～省略（リスト2-30と同じ）～
</script>

<template>
  <header>
    <h1>ルーティングサンプル</h1>                    ❶
  </header>
  <main>                                             ❷
```

4.3 useAsyncData() と $fetch() を簡潔に書ける useFetch()

前節で紹介した useAsyncData() は、そのハンドラ内部で $fetch() 関数を利用してデータ取得を行うことがほとんどです。ですので、Nuxt では、この useAsyncData() と $fetch() の組み合わせをより簡潔に書ける関数として、**useFetch()** が用意されています。次にこれを紹介します。

4.3.1 サンプルプロジェクトの共通部分の作成

前節と同じように、本節で作成するサンプルである usefetch プロジェクトについても、まずその共通部分を作成しましょう。usefetch プロジェクトを作成し、interfaces.ts ファイルを asyncdata プロジェクトから、usefetch フォルダ直下にファイルごとコピー＆ペーストしてください。同様に、asyncdata プロジェクトの pages/index.vue を、usefetch/pages フォルダ内にファイルごとコピー＆ペーストしてください。

次に、app.vue 内のコードを、asyncdata プロジェクトの app.vue のソースコードとまるまる置き換えてください。ただし、asyncdata プロジェクトと同様に、テンプレートブロックの h1 タグだけは、リスト 4-11 のように変更しておいた方がよいでしょう。

▼ リスト 4-11　usefetch/app.vue

```
～省略～
<template>
  <header>
    <h1>useFetchサンプル</h1>
  </header>
  ～省略～
</template>
```

最後に、pages/WeatherInfo/[id].vue ファイルを作成し、asyncdata プロジェクトの pages/WeatherInfo/[id].vue ファイルと同じコード記述してください。念のために、ここまでの内容で、動作確認を行ってください。

4.3.2 useFetch() の使い方

プロジェクトの基本部分ができたところで、pages/WeatherInfo/[id].vue に useFetch() を利用したデータ取得コードを追記していきましょう。これは、リスト 4-12 の太字のコードになります。これまでと同様に、appid の値を各自のものに置き換えるのを忘れないでください。ただし、❶の行はもはや不要となります。リスト 4-12 ではコメントアウトしていますが、削除してもかまいません。

▼ リスト 4-12　usefetch/pages/WeatherInfo/[id].vue

```
<script setup lang="ts">
import type {City} from "@/interfaces";
～省略～
// const weatherDescription = ref("");                                    ❶
const params:{                                                            ❷
  lang: string;
  q: string;
  appid: string;
} =
{
  lang: "ja",
  q: selectedCity.value.q,
  //APIキーのクエリパラメータ。ここに各自の文字列を記述する!!
  appid: "xxxxxx"
}
const asyncData = await useFetch(                                         ❸
  "https://api.openweathermap.org/data/2.5/weather",                      ❹
  {
    key: `/WeatherInfo/${route.params.id}`,                               ❺
    query: params,                                                        ❻
    transform: (data: any): string => {                                   ❼
      const weatherArray = data.weather;
     const weather = weatherArray[0];
      return weather.description;
    }
  }
);
const weatherDescription = asyncData.data;
</script>

<template>
  ～省略～
</template>
```

追記が終了したら、一度動作確認を行ってください。無事天気情報が表示されます。

リスト 4-12 では、❸で **useFetch()** 関数を利用しています。この useFetch() 関数を構文としてまとめると、次のようになります。

useFetch()

```
useFetch(
  アクセス先URL,
  オプションオブジェクト
);
```

useAsyncData() 関数との決定的な違いは、ハンドラを記述する必要がないということです。useAsyncData() 関数のハンドラ内で $fetch() 関数に渡していたアクセス先 URL を、直接 useFetch()

5.3 サーバサイドルーティング

前節で、一通り、Nuxt でサーバ API エンドポイントを用意する方法の紹介が終了しました。

ただし、その際のサーバサイド URL は全て /api 配下となっています。一方、Web の世界、特にサーバ API エンドポイントの世界では、REST という考え方があります。この考え方に従うと、前節で作成したサーバ API エンドポイントには少し問題があります。その辺りを解消しながら、Nuxt のサーバサイドルーティングを紹介します。

5.3.1 REST API とその URL

REST とは、**Representational State Transfer** の略であり、2000 年にロイ・フィールディング（Roy Fielding）によって提唱された考え方です。その考え方によると、URL は、Web 上のリソース（データ）を特定するように設計する必要があるということです。詳細は他媒体に譲りますが、この考え方を前節で作成した会員情報管理アプリに適用すると、表 5-1 のようになります。

▼ **表 5-1　REST の考え方を適用した会員情報管理サーバ API エンドポイント設計**

	データ内容	HTTP メソッド	パス例
①	会員リスト情報	GET	/member-management/members
②	特定の会員情報	GET	/member-management/members/33456
③	会員情報の登録	POST	/member-management/members

まず、会員情報管理のように、何かデータを管理するサーバ API エンドポイントの場合は、表 5-1 の member-management のように、画面表示用の URL とは別のデータ管理を表すようなパス配下にまとめます。そして、そのデータを複数取得する場合、続けて、対象データの複数形とします。例えば、会員情報ならば、members とします。それが、①です。

さらに、そのうちの 1 件のデータのみを取得する場合は、②のように特定するためのキーをパスに続けます。②の場合は、会員 ID を members に続けています。

では、データ登録の場合はどうするかというと、HTTP メソッドで区別します。対象データを複数まとめたもの、つまり、members に 1 件データを追加する場合は、パスはそのまま members とし、その URL に **POST** 送信します。それが、③であり、①と③を区別するのは、HTTP メソッドが **GET** か POST かだけであり、URL に差はありません。もし特定の会員情報を更新する場合は、URL は②と同じものを利用し、代わりに HTTP メソッドとして **PUT** を利用します。削除する場合は、同じく URL は②と同じものとし、**DELETE** メソッドとします。

このように、HTTP メソッドを組み合わせることにより、URL を参照するだけで、それがどのようなデータを表すのかを一目瞭然として URL を設計するのが、REST の特徴です[*5]。

5.3.2 サーバサイドルーティングプロジェクトの準備

このような REST に従った API を実現したプロジェクトを Nuxt で作成しようとすると、前節までの知識では問題があります。というのは、サーバサイド処理の URL が /api 配下となってしまうからです。

もちろん、それを解決する方法はあります。それを紹介する前に、前節で完成した server-basic プロジェクトを複製して server-routes プロジェクトとし、その server-routes プロジェクトを表 5-1 のパスを利用したものへと改造していきましょう。

まず、server-routes プロジェクトを作成し、server-basic プロジェクトの次のファイル一式を、ファイルごと server-routes プロジェクトの同階層にコピー＆ペーストしてください。なお、コピー＆ペーストした時点で、サーバサイドファイルが存在しないため、プロジェクトがエラーとなりますが、現時点ではそのままとしておいてください。のちの改造で解消していきます。

- interfaces.ts
- membersDB.ts
- layouts/default.vue
- layouts/member.vue
- pages/index.vue
- pages/member/memberList.vue
- pages/member/memberAdd.vue
- pages/member/memberDetail/[id].vue

また、server-routes プロジェクトの app.vue の中のソースコードも、server-basic プロジェクトの app.vue のものをコピー＆ペーストして、丸々書き換えておいてください。

その上で、layouts/default.vue と layouts/member.vue のテンプレートブロックの h1 タグを、リスト 5-12 のように変更しておいた方がよいでしょう。

▼ **リスト 5-12　server-routes/layouts/default.vue と server-routes/layouts/member.vue**

```
〜省略〜
<template>
  <header>
    <h1>サーバサイドルーティングサンプル</h1>
  </header>
  〜省略〜
</template>
```

＊5　より詳しくは、REST API Tutorial サイトの次のページを参照してください。
https://restfulapi.net/resource-naming/

6.1 Nuxt のエラー発生とエラー処理タグ

本章では、エラーをテーマにさまざまな仕組みを紹介していきます。その際、前半では、error-basic というプロジェクトを作成しながら、エラー処理のさまざまなパターンを紹介していきます。その後、後半では、5.4 節で作成した server-storage プロジェクトを移植した error-practical プロジェクトにエラー処理を組み込みながら、より実践的なエラー処理を紹介します。

その前半でまず紹介するのは、エラーを効率よく発生させる仕組みと、そのエラーを処理する専用タグです。

6.1.1 サンプルプロジェクトの準備

早速、前半のプロジェクトである error-basic を作成していきましょう。ここでは、図 6-1 のような画面を作成します。

▼ 図 6-1　error-basic に作成する最初の画面

エラーを [発生]

戻る

この画面用コンポーネントを errorHandlerBasic とすると、画面中の「エラーを発生」と表示されている部分は、子コンポーネントの ErrorGeneratorBasic とします。そして、この［発生］ボタンをクリックすると、エラーが発生します。本来の動作としては、このボタンが何かの処理、例えば、サーバにアクセスしてデータを取得して表示させる処理のようなものを想定してください。その際に、何かのエラーが発生してしまうとします。それを擬似的に再現するようにします。

プロジェクトの基本部分の作成

では、早速作成しましょう。error-basic プロジェクトを作成し、app.vue をリスト 6-1 のように書き換えてください。

▼ リスト 6-1　error-basic/app.vue

```
<template>
  <NuxtPage />
</template>
```


トップ画面の作成

次に、図 6-1 の画面へのリンクが掲載されたトップ画面を作成しましょう。これは、リスト 6-2 の pages/index.vue です。

▼ リスト 6-2　error-basic/pages/index.vue

```
<template>
  <ul>
    <li>
      <NuxtLink v-bind:to="{name: 'errorHandlerBasic'}">
        エラー表示実験
      </NuxtLink>
    </li>
  </ul>
</template>
```

エラー処理画面の作成

次に、リスト 6-2 の［エラー表示実験］のリンク先のページ、すなわち、図 6-1 の画面用コンポーネントである errorHandlerBasic.vue を作成しましょう。これは、リスト 6-3 の内容です。

▼ リスト 6-3　error-basic/pages/errorHandlerBasic.vue

```
<template>
  <ErrorGeneratorBasic/>
  <p>
    <NuxtLink v-bind:to="{name: 'index'}">
      戻る
    </NuxtLink>
  </p>
</template>
```

戻るリンク以外は、子コンポーネントとして ErrorGeneratorBasic をレンダリングしているだけのコードです。

エラー生成コンポーネントの作成

最後に、この ErrorGeneratorBasic.vue を components フォルダ内に作成しましょう。これは、リスト 6-4 の内容です。

▼ リスト 6-4　error-basic/components/ErrorGeneratorBasic.vue

```
<script setup lang="ts">
const onThrowsErrorClick = (): void => {
  throw createError("擬似エラー発生。"); ❶
};
</script>
```

7.1 ログイン機能の実装

本章のテーマは、ミドルウェアです。そのミドルウェアとは何か、作り方はどうすればいいのか、などの本題は次節から紹介することとします。

本節では、まず、ミドルウェアを導入しやすいアプリケーションとして、前章で作成したerror-practicalプロジェクトを移植し、そこにログイン・ログアウト機能を実装していきます。ミドルウェアの解説に入る前に、少し横道にそれますが、お付き合いください。

7.1.1 サンプルプロジェクトの準備

では、早速作成していきましょう。まずは、プロジェクトの移植からです。middleware-fundamentalプロジェクトを作成し、error-practicalプロジェクトの次のファイル一式を、ファイルごとmiddleware-fundamentalプロジェクトの同階層にコピー&ペーストしてください。

- interfaces.ts
- server/routes/member-management/members.get.ts
- server/routes/member-management/members.post.ts
- server/routes/member-management/members/[id].get.ts
- layouts/default.vue
- layouts/member.vue
- pages/index.vue
- pages/member/memberList.vue
- pages/member/memberAdd.vue
- pages/member/memberDetail/[id].vue

また、middleware-fundamentalプロジェクトのapp.vueの中のソースコードも、error-practicalプロジェクトのapp.vueのものをコピー&ペーストして、丸々書き換えておいてください。

その上で、layouts/default.vueとlayouts/member.vueのテンプレートブロックのh1タグを、リスト7-1のように変更しておいた方がよいでしょう。

▼ **リスト7-1　middleware-fundamental/layouts/default.vueとerror-practical/layouts/member.vue**

```
<template>
  <header>
    <h1>ミドルウェアサンプル</h1>
  </header>
  ～省略～
</template>
```

移植が終了したら、プロジェクトを起動し、error-practicalプロジェクトと同様の動作になるか確認しておいてください。

7.1.2 ログイン機能の概要

ここから、このmiddleware-fundamentalプロジェクトにログイン機能を実装していきます。今から実装する画面は、図7-1の①の画面です。この画面に適切なログインIDとパスワードを入力すると、これまで通り②のトップ画面が表示されます。ただし、ヘッダ部分に現在ログインしているユーザ名とログアウトのリンクが表示されます。このログインしているユーザ名とログアウトリンクのヘッダ表示は、例えば③の会員リスト画面のように、これまでのプロジェクトで表示させてきた会員管理関係の全ての画面で表示させるようにします。ログアウトリンクをクリックすると、ログアウト処理が行われ、①のログイン画面に遷移します。

▼ **図7-1　middleware-fundamentalプロジェクトで新たに実装する画面**

プロジェクト全体としては、ログイン画面を表示させるコンポーネントとログアウト処理を行うコンポーネント、さらに、サーバAPIエンドポイント側として、ログインのためのユーザ認証に使うファイルが追加されることになります。それぞれ、表7-1の内容となります。

▼ **表7-1　middleware-fundamentalプロジェクトに新たに追加する主なファイルの内容**

内容	パス	プロジェクト内ファイルパス
ログイン画面	/login	pages/login.vue
ログアウト処理	/logout	pages/logout.vue
ログイン用ユーザ認証エンドポイント	/user-management/auth	server/routes/user-management/auth.post.ts

8.1 npm runのオプション

これまで作成してきた Nuxt アプリケーションを起動するために利用してきたコマンドは、npm run dev です。これは、開発用サーバが起動するコマンドです。一方、npm run には他のオプションがあり、実運用と関連しています。本節では、そのあたりを紹介していきます。

8.1.1 build オプション

まず紹介したいオプションは、**build** オプション、すなわち、次のコマンドです。このコマンドを実行することで、本運用で動作するファイル一式が作成されます。

本運用ファイル一式作成コマンド

```
npm run build
```

試しに、このコマンドを前章で作成した middleware-fundamental プロジェクトで実行してみてください。すると、リスト 8-1 のようにさまざまなメッセージがコンソールに表示され、最終的にプロンプトが戻ってきます。

▼ リスト 8-1　middleware-fundamental プロジェクトの npm run build の実行結果

```
% npm run build

> build
> nuxt build

Nuxi 3.6.2
Nuxt 3.6.2 with Nitro 2.5.2
ℹ Building client...
vite v4.3.9 building for production...
✓ 146 modules transformed.
.nuxt/dist/client/manifest.json                    5.98 kB
〜省略〜
✓ Client built in 1616ms
ℹ Building server...
vite v4.3.9 building SSR bundle for production...
✓ 102 modules transformed.
.nuxt/dist/server/_nuxt/app-styles.2f36be12.mjs     0.08 kB
〜省略〜
✓ Server built in 618ms
✓ Generated public .output/public
ℹ Building Nitro Server (preset: node-server)
✓ Nitro server built
  ├── .output/server/package.json (1.19 kB) (444 B gzip)
〜省略〜
Σ Total size: 3.24 MB (764 kB gzip)
✓ You can preview this build using node .output/server/index.mjs ―― ❶
```


すると、プロジェクトフォルダ内には、これまでなかった **.output** フォルダが作成されており、その中に、図 8-1 のようにさまざまなフォルダやファイルが作成されています。

▼ 図 8-1　npm run build によって生成された .output フォルダ

8.1.2 ビルドされたプロジェクトの実行

ここで作成された .output フォルダ内のファイル一式、すなわち、ビルドファイル一式は、このままで Node.js 上で動作するように作成されています。試しに実行させてみましょう。

その実行コマンドは、実は、リスト 8-1 のビルドメッセージ内に表示されており、❶が該当します。再掲載すると、次の構文です。そして、このコマンドの通り、図 8-1 にもある .output/server フォルダ内に生成された index.mjs ファイルを Node.js に実行させることで、プロジェクト全体が動作するようになります。

ビルドされたプロジェクトの実行コマンド

```
node .output/server/index.mjs
```

9.2 Netlifyへのデプロイ

概論はここまでにして、早速、実際にデプロイを行っていきましょう。最初は、Netlify へのデプロイです。

9.2.1 Netlify とは

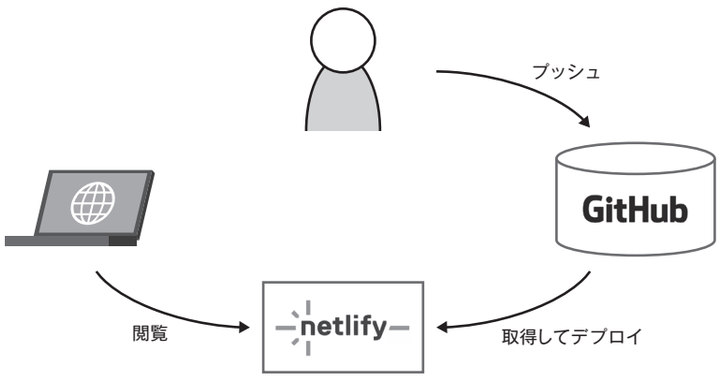
Nuxt アプリケーションのデプロイを紹介していく本章のトップバッターとして **Netlify**（ネトリファイ）を選んだのには、理由があります。そして、その理由が Netlify の特徴そのものを表しています。Netlify の URL は次の通りです。

https://www.netlify.com/

表 9-1 にもあるように、Netlify（ネトリファイ）は、Node.js のホスティングサービスです。Node.js 上で動作する Web アプリケーションならば、動作する環境を自動で用意することができます。自動というのは、ほぼ設定の必要がない、ということです。もちろん、設定をカスタマイズすることもできますが、基本的な環境はほぼ全て自動で用意されます。ということは、同じく Node.js 上で動作する Nuxt アプリケーションを実行させるには、最適な選択肢といえます。

しかも、デプロイの方法も非常に簡単であり、GitHub などの Git ホスティングサービスでソースコードの管理を行っていれば、そこから自動で取得、ビルドを行ってくれるようになっています（図 9-1）。プログラマは、ソースコードを zip などのアーカーブにしてアップロードしたり、その中に必要なライブラリ類を含めたり、といった操作は不要です。このお手軽さを、次項以降味わっていただこうと思います。

▼ 図 9-1　Netlify でのデプロイ



このように、非常に便利な Netlify なため、かなりの Nuxt プログラマは第 1 選択として Netlify を利用しています。しかも、表 9-1 にある通り、料金設定としては月額料金が必要とはいえ、原稿執筆時点では、次のような制約下で利用する限りは、無償となっています。

- デプロイの管理を行えるユーザが一人
- ひとつのサイトに複数のビルドパターンを併用できない
- 100GB/ 月の帯域
- ビルド時間の制限が 300 分 / 月

上記を見てもわかるように、かなりの範囲を無料利用の範囲で済ますことができます。

ただし、欠点がないわけではありません。まず、用意されている環境が Node.js 環境のみですので、データベースなどのデータ保存部分に関しては、別のサービスを独自に用意し、連携させる必要があります。そのため、8.3 節で作成したような Redis を利用したアプリケーションの場合は、Netlify 単独では実現できません。

この制約のため、本書で作成したサンプルのうち、第 7 章で作成した middleware-fundamental プロジェクトは適切に動作しますが、Redis を利用しないため、データの永続化は行われていません。また、第 4 章で作成した composables プロジェクトも動作します。そこで、本節では、この 2 個のプロジェクトを Netlify 用に移植し、デプロイしていくことにします。

9.2.2 middleware-fundamental プロジェクトの移植

前項で説明したように、Netlify へのデプロイは、Git ホスティングサービス経由で行うことになっています。そこで、本書では GitHub を利用し、GitHub 経由で Netlify へデプロイします。

まず、middleware-fundamental プロジェクトを移植するプロジェクトとして、gihyonuxt-members-netlify プロジェクトを作成してください。middleware-fundamental プロジェクトの次のファイル一式を、ファイルごと gihyonuxt-members-netlify プロジェクトの同階層にコピー＆ペーストしてください。

- interfaces.ts
- server/routes/member-management/members.get.ts
- server/routes/member-management/members.post.ts
- server/routes/member-management/members/[id].get.ts
- server/routes/user-management/auth.post.ts
- middleware/loggedin-check.ts
- layouts/default.vue
- layouts/loggedout.vue
- layouts/member.vue
- components/TheLoggedInSection.vue
- pages/index.vue
- pages/login.vue
- pages/logout.vue
- pages/member/memberList.vue

付録 3 Nuxt Devtools

付録 2 で紹介した Vue Devtools とは別に、Nuxt 専用の開発者ツールとして **Nuxt Devtools**[*2] があります。この Nuxt Devtools は Nuxt アプリケーションに埋め込む形で利用し、Vue Devtools と併用できます。さらに、Nuxt 用の開発者ツールというだけのことはあって、Vue Devtools では参照できない Nuxt 専用のデータを参照することができます。

Nuxt Devtools のインストール

原稿執筆時点での Nuxt の最新バージョンである 3.6 では、プロジェクトを作成するだけで自動的に Nuxt Devtools が組み込まれます。もし、手動で既存のプロジェクトに組み込む場合は、次のコマンドを実行してプロジェクトにモジュールを追加します。

```
npm install --save-dev @nuxt/devtools
```

その後、nuxt.config.ts に次の太字の 1 行を追記します。

```
export default defineNuxtConfig({
  devtools: {enabled: true}
})
```

Nuxt Devtools の利用

Nuxt Devtools が有効になると、Nuxt アプリケーションを起動した画面最下部に、図 A-9 のように アイコンが表示されます。

▼ 図 A-9　Nuxt Devtools が組み込まれた画面

＊2　https://devtools.nuxtjs.org/

このアイコンをクリックすると、図 A-10 のようなモーダルウィンドウが表示されます。これが Nuxt Devtools の画面です。

▼ 図 A-10　Nuxt Devtools の画面が表示された状態

ここから、画面左側の各アイコンをクリックすることで表示内容が変わり、さまざまな情報が確認できます。図 A-10 では、このナビ部分はアイコンのみの表示ですが、画面を広げると、図 A-11 のように、アイコンとその名称も表示されます。

▼ 図 A-11　アイコンとナビ名が表示された Nuxt Devtools の画面