まえがき

フロントエンドは停滞しています。

もう長らく、多くのエンジニアの中で「フロントエンドは進化がはやく、技術の寿命が短すぎる」という言説が流行しています。そして、特にフロントエンドの開発が中心ではないエンジニアは、それを理由にフロントエンドを避け続けるという状態が続いています。

しかしながら、現役のフロントエンドエンジニアとして活動してるかたには、最近の動向に関しては、この言葉は間違いであるような印象を受けることが多いのではないでしょうか。

確かに Web Assembly や Web USB といった新興の技術が実用化される段階に入っており、Web ブラウザの上での世界は進化し続けています。

しかしながら、通常の SPA アプリケーション開発の基盤はここ数年全く変わっていません。

フロントエンドの開発のフレームワークは React, Angular, そして Vue.js の 3 つが圧倒的なシェアを誇っており、その他のフレームワークも、Web Components もしくは Virtual DOM どちらかの考え方にもとづいて作られているものがほとんどでしょう。

状態管理も、Redux が今は全盛期ですが、Flux 派生か Reactive Extension 派生のどちらかであり、根本は全く変わっていません。

一見、移り変わりが激しいようにみえる界隈ですが、Enhancement という意味で進化を続けているだけで、実はイノベーションは長らく起こっていないわけです。

逆に言えば、今のモダンフロントエンドの知識は次のイノベーションが到来するまでは長らく基盤として存在し続け、かつ到来したとしても、当分は続くものであると見ることができます。ちょうど、未だに jQuery やBackboneのお世話になっているプロジェクトが保守されることがあるように。

そうなった場合、次なるイノベーションが起こるまでは、現代の技術の中でも広く浸透しており、かつ強力な技術をベースとしたスキルスタックを存分に活用することが幸せにつながるでしょう。

そして、本書ではまさにそこをついているフロントエンドのフレームワークである Nuxt.js(以下 Nuxt) を取り扱います。Nuxt は世界的にも伸び続けており、日本国内では特に大きな盛り上がりをみせている Vue.js をベースとした、フルスタックな開発フレームワークです。

Vue.js の親しみやすさをもったまま、強力な機能を提供する Nuxt は、今のモダン JavaScript フレームワークの世界において、これから存分にその存在感を発揮してくれることでしょう。

本書は、私が Nuxt を知り、学び、そして現場で培ってきたノウハウを凝縮して一冊にしたものとなります。これまでの、そしてこれからのフロントエンドについての考えを本編の冒頭で知っていただいたうえで、いま Nuxt を学ぶことは、これからの Web 業界においてあなたが存在感を発揮するために大きく貢献することでしょう。

SPA 開発から、その欠点をおぎなう SSR に、進化させるための PWA の開発まで。現代のフロントエンドにおけるポピュラーなトピックを全てカバーする Nuxt を学び、その強力なシステムを存分に体験する機会となれば幸いです。

本書のサポートについて

本書の正誤表・サンプルコードは全て GitHub にて公開しています。必要に応じてご活用ください。クローズドな場所に移動する際は、パスワードを「tbf04_nuxt」とします。

その他の本書へのご意見やご感想も、レポジトリの Issue もしくは Twitter @potato4d にて受け付けています。

https://github.com/potato4d/nuxt-tech-book

著者紹介

花谷拓磨 a.k.a. potato4d

高校時代に Web 制作会社及びスタートアップで Web 制作・スマホアプリ開発・Webアプリ開発を経験後、ピクシブ株式会社に入社。「Pawoo Music」の初期開発などに携わったのち独立。現在はフリーランスとして、Vue.js などでのフロントエンド開発、PHP や Ruby on Rails でのバックエンドから、デザインまでを行うかたわら、中級者向けのハンズオン団体 JSLounge を主催している。

Vue.js への貢献実績として Vue.js JP Document メンテナ、Nuxt.js Document(en/ja) 貢献、Vue.js JP User Group スタッフ、執筆実績として「Vue.js製フレームワークNuxt.jsではじめるUniversalアプリケーション開発」など。

目次

1. イントロダクション

Nuxt についてご紹介する前に、まずは現代のフロントエンドの情勢、そしてこれからのフロントエンドについて考えてみましょう。

これまでの SPA ・これからのSPA

「 SPA 」という言葉はもう聞き慣れたかと思いますし、比較的新しいサービスであれば、サービス本体やそのファミリーの中のどこかではもう使われていることでしょう。

サーバーサイドのおまけであったフロントエンドは、全く別のレイヤーとして独立し、HTTP 通信をベースとして、バックエンドを API 化。HTML をサーバーサイドと送受信する時代は終わり、JSON データの送受信以外の全ての UI のフィードバックは、今や JavaScript を利用してブラウザの内部で完結するようになりました。

そうして、その開発用のフレームワークも進化を続け、双方向バインディング、Flux アーキテクチャ、仮想 DOM など、これまでのただの JavaScript とDOM の関係では成し得なかった技術が大量に生まれ、当たり前のように使われる時代となっています。

それらの考えは、当分腐ることがないでしょうし、React やAngular、Vue.js などのフレームワークも当分は使われることでしょう。

しかしながら、技術の寿命は有限です。だからこそ、当分使われる技術の知識を身に着けているのであれば、来るべき新たな技術に対して考え、少しでもキャッチアップを行うべきです。

iOS Mobile Safariでの実装完了により、正式に日の目を見る PWA 。

そして、その PWA において、UX 向上に貢献する Service Worker の存在など、新たな技術としての道しるべは多少なりとも示されている状況です。

また、フロントエンドでは今のモダンフロントエンドにたどり着くまで、長らく技術の停滞があったことを考えると、スマートフォンアプリ開発やサーバーサイド時代の変遷など、未来を考えるために必要な要素は十分に揃っているはずです。

そこまで示されているのであれば、今後の SPA 開発に必要な要素と、それを満たすためにフレームワークに求められるものも自ずとわかっているはずでしょう。まずは少しその点について考えてみましょう。

今後、フロントエンドのフレームワークに求められるモノ

今後フロントエンドのフレームワークに求められるモノ。それは、より高い生産性を誇る開発基盤と、これから流行する技術に対応できるだけの懐の広さです。

高い生産性を誇る開発基盤

バックエンドにおいて、「今最も多くの人に愛されている開発のフレームワークは何か」という質問があれば、おそらくは「 Laravel 」もしくは「Ruby on Rails」と答える人が過半数を越えることでしょう。

これらが愛される理由は、「フルスタックフレームワークであり、全貌を把握しきらずとも少しずつ理解しながらうまく利用することが可能」であり、かつ「本格的なアプリケーション開発においても識者がいれば十分活用が可能」というところにあるかと思います。

フロントエンドにおいてのフルスタックフレームワークというと Angular が真っ先に挙げられると思いますが、その一方で Angular は今あげたようなバックエンドのフレームワークとは違い、TypeScript が基本であり、かつ DI や Service層など、モダンなアーキテクチャにおける開発を支援するようなスキルスタックとなっています。状態管理も、C #文化から派生し、スマートフォンアプリ開発で人気の Rx をベースとした開発基盤となっています。

これらはたしかに堅牢な基盤ではありますが、その一方であまり柔軟な作りにはなっていないという課題があります。得てして開発におけるスピード感と堅牢さというものはトレードオフとなってしまいますが、こと Angular に関しては、完全に堅牢さに重きを置いた構成となっています。

そういった開発基盤となると、柔軟な使い方には勿論不向きですし、シンプルなユースケースから、複雑なアプリケーションまで、幅広く扱うことはできない状況となっています。

逆に React は非常にミニマルであり、かつ現代的な React 開発では関数型の考えを取り入れることによって、小さなツールで大きな課題を解決しようとしています。こちらは確かにどの規模にも対応できますし、開発において重要視するポイントを適宜選択することができますが、ベストプラクティスに不足しているように見えます。

そして Vue.js についても、ちょうど React とAngularの中間程度の感覚で利用が可能であり、かつどのフレームワークよりも小規模からスタートしやすいものとなっていますが、こちらは大規模開発での鉄板というものが圧倒的に不足しており、柔軟すぎるがゆえに自由な設計のまま進んでしまいがちです。

確かに、それぞれのアプローチにおいてはひとつずつ正解が出ていますが、しかしながら、爆発的人気を誇る技術は、どれも「柔軟に使えるが、縛るところは正しく縛られている」という特徴があり、今のフロントエンドのベースとなるライブラリにはそれが不足しています。

これからのフロントエンド開発では、「 Vue.js と同等の高い生産性を誇り、かつ Angular と同じくらいうまく物事をこなせる技術基盤」が必要になることでしょう。

これから流行する技術への対応

また、それだけではこの先のフロントエンド開発では不十分でしょう。先程も述べたように、既にTwitter Liteをはじめとして、大規模サービスでも実用例が増えているいわゆる「 PWA 」への対応も課題です。

また、これからはWeb Assemblyが流行るかもしれませんし、どういった形で情勢が動くかはまだまだ未知数な状況です。しかしながら、PWA に関しては、トレンドとなることが間違いないでしょう。

大切なことは、今これに対応するというだけでなく、これから来たる技術に対しても柔軟な対応が可能であることです。

その実現のためには、新興の技術に対してコミュニティ全体が関心を持っており、エコシステムの構築に前向きである技術を使うべきでしょう。

要求を満たす技術はどんなものであるか

そして、これらの要求を満たす技術は、おそらくは既存のライブラリ・フレームワークをベースとしたものであり、そのコミュニティと共存関係であるものでしょう。

そして、その上でベース技術の良い側面を取り入れながらも、破壊的な意思決定が敬遠されがちである、ベース技術の欠点を補うかのように、挑戦的な技術を積極的に取り入れる存在となっていることでしょう。

本書においては、その一つの形として、次のセクションから Nuxt をご紹介していきます。

2. Nuxt概論

それではいよいよ本書のテーマである Nuxt の概要についてご紹介していきます。

このセクションでは、Nuxt がどういった技術であるか。という簡単なご紹介のほか、前セクションにて考えた要件に対して、どれだけ Nuxt がマッチしているか。どういった時に Nuxt を利用すべきか。

逆に利用すべきでない場合はどういうときか。といったものをご紹介いたします。

Nuxtとは何か

Nuxt は、Vue.js をベースとして開発された、フロントエンドのフルスタックなフレームワークとなります。当初は Vue.js におけるサーバーサイドレンダリングが煩雑であるという課題を解決するために、先行の React 製のフレームワークである「 Next.js 」を模して開発されましたが、今やミニマルな機能だけを提供している Next.js とは裏腹に、Vue.js 開発で頻繁に利用される機能を全て取り込んだ、非常に強力なフレームワークとして仕上がっています。

サーバーサイドレンダリングは勿論、それ自体をオフにしてただの SPA 用のフレームワークとして利用することも可能ですし、SPA を SEO に強い HTML として正しく静的サイトとして吐き出してくれる機能まで持っている状態となり、様々な開発シーンで柔軟に使えるフレームワークとして仕上がっています。

サーバーサイドレンダリングを行わず、ただの SPA 基盤として使う場合でも、Flux パターンの Vue 実装である Vuex ストアのオートローディング機能や、ルーティングの自動生成機能、グローバルなレイアウトの共通化システムに、特定のルーティングに限って認証を付与するなど、アクセスに対してガードを張ることが可能なミドルウェアに、通常の Vue プラグインをより柔軟に扱うためのプラグインシステム、そして公式が提供するプラグインやミドルウェアを簡単に読み込むことができる Nuxt モジュールまで。簡単な SPA 開発では全ての機能を使い切れないほどの機能があります。

勿論、中規模大規模となってくると、特定のルーティングにおいてリクエスト内容を改ざんできるミドルウェアなどはなくてはならない存在となっていきます。そのため、それらを搭載した Nuxt は、大きくなるアプリケーションとともに成長していくのにベストなフレームワークとなっています。

今、SPA 開発に Nuxt が求められるワケ

確かに今まであげた機能は魅力的ですが、Vue.js 単体でこれまでの開発が行われてきたことを考えると、決して Nuxt に頼らずとも本格的な SPA 開発は十分こなせるはずです。では、その上でなぜ Nuxt が必要なのでしょうか?

その一つの回答として、秩序があります。Vue.js は、非常に柔軟なフレームワークとなっており、かつ React + Redux や Angular のように、鉄板パターンが提唱され、浸透しきっているわけではない状態であります。Vue.js は間口が広いこともあり、一般的に自由なコーディングが蔓延しやすい傾向にあり、なかなかにアーキテクチャが安定しないことを経験したことがあるかたは多いでしょう。

Nuxt は、その問題を規約ベースのルール設定と規約に沿った場合の強力なサポートによって縛ることが可能です。ディレクトリ名など、ある程度は規約を曲げることもできますが、基本的には Nuxt が良しとするルーティング構成で、Nuxt が良しとする Vuex Store 構成やコンポーネント構成にしておけば間違いなく、かつ強力な支援が得られることとなります。

また、コンポーネントやページはまだある程度プロジェクトごとにルールの制定することができますが、middleware や plugin に関しては、Vue.js デフォルトでは存在しないため、バラバラに管理されるか、index.js や App.vue の無闇な肥大化につながることが頻発します。この問題についても、Nuxt 側でシステムの実装が行われているため、そのルールに沿っていくことによって、秩序のある開発が可能となります。

Vue.js 開発で苦しんでいる人が、なによりも Nuxt に対して希望がもてるポイントは、きっと秩序の一点でしょう。

もう一つの回答は、前セクションで紹介した「流行する技術への追従」を正しく行っていることでしょう。

Nuxt は、その独自コミュニティの中で、PWA 用のモジュールや、サーバーサイドレンダリングの基盤を提供しています。確かに Vue.js でも、非常にアトミックな単位での SSR のシステムや、PWA のテンプレートは提供していますが、Nuxt ほど手厚いサポートではないでしょう。また、Vue.js 本体がそこまでサポートすることは逆にライブラリとしての方向性に疑問をもつかたも少なくないでしょう。

Nuxt は、それ単体のコミュニティよって進化を続けながら、Vue.js の本体のルールから逸脱しない形で進んでいるため、これからも先進的な機能をフル活用しながら、かつベースコミュニティとうまく付き合っていく理想的な形を進んでいくことでしょう。

Nuxtをオススメするシーン、しないシーン

それでは、その前提を踏まえた上で、どういったシーンにおいて Nuxt をオススメするかをご紹介いたします。

まず、前提として、新規の Vue.js での SPA 開発であれば、間違いなく Nuxt を選ぶことをオススメいたします。

Nuxt では、サーバーサイドレンダリングを行うことも行わないことも可能ですし、最終的に静的サイトとして書き出すことができるという特徴を考えると、サーバーサイドレンダリングか否かという点は、さほど重要ではありません。

必要であればサーバーサイドレンダリングをオンで運用することができる開発環境と考えるのが最も自然でしょう。

では逆にオススメしないシーンはどこか。

基本的にはあまり存在しませんが、以下のような場合に限っては、Nuxt の導入については考え直すほうが幸せでしょう。

- 既に Vue.js で書かれた大きめのコードベースであり、アプリケーション内に独自のレイヤーが存在する場合
- 既に Vue.js で書かれたもので、ブラウザ依存のコードが大量に存在する場合

- Webpacker や Laravel Mix など、完全な SPA ではない場合

いずれも Nuxt のルール内で取り扱うことができない分野となりますので、こういったプロジェクトに無理に Nuxt を導入する場合は、不整合が起こるため幸せとは言えないでしょう。

逆に言えば、SPA 新規開発においては Nuxt を選択しない積極的な理由はほとんどありませんで、基本的には SPA の開発であれば積極的に Nuxt を採用していきましょう。

そのほか、SPA ではなく、複数ページの Web サイトの開発においても Nuxt は有効に働きます。

Nuxt を静的サイトモードにて運用することによって、Vue.js の恩恵を受けながらも SEO 対策が万全な Web サイトを作ることが可能であるため、SPA を中心に触るエンジニアが Web サイトを作る場合においても、Nuxt を存分に活用することができるでしょう。

Nuxtによる事例紹介

最後に、実用事例をいくつかあげておきます。全て何かしらの形で筆者が開発に携わっているプロジェクトとなります。

Web サイトでの事例

- https://push7.jp
- https://corp.scouter.co.jp

どちらも私が作成した Nuxt での Web サイトとなります。Nuxt を中心に利用してサーバーサイドレンダリングを行っており、一部バックエンド実装が必要な箇所は Express と連携しています。

また、2018 年 11 月に開催される Vue.js オンリーのカンファレンス VueFes Japan 2018 の Web サイトも、Nuxt(静的サイトモード) + Netlify にてホスティングされています。

https://vuefes.jp

Web アプリケーションでの事例

また、その他フロントエンドの花形ともいえる SPA、Web アプリケーションの領域でも広く採用されています。最新の例では、ブロックチェーンメディア ALIS での実用例があります。

Nuxt を AWS Lambda 上にデプロイし、同じく Lambda 上で動作するサーバーレス API と連携をしている事例です。こちらはソースコードがオープンソースとなっており、誰でも閲覧が可能となっておりますので、実運用の例として是非一度アクセスください。

https://github.com/AlisProject/frontend-application

3. Nuxt による基本的なWebアプリケーション開発

ここまでで Nuxt のバックグラウンドやメリットについてご紹介してきましたが、ここからは Nuxt を触ったことがないかた向けに、Nuxt での簡単なWebアプリケーションの開発の流れをご紹介いたします。

Step-by-Step で Nuxt のパワフルな機能をご紹介していきますので、このセクションを一通り読んで実践していただくことで、Nuxt による開発の空気感を掴んでいただくことができるはずです。

それでは実際に、開発の様子を体験してみましょう。

また、もし手元に開発環境がないかたや、既に Nuxt にある程度慣れ親しんだかたで、ソースコードだけ確認したいかたは、Web 上にデモアプリケーションとソースコードを公開していますので、下記の URL から適宜ご活用ください。

- デモ: https://potato4d.github.io/nuxt-tech-book/examples/section03/03_Tutorial/
- ソースコード: https://github.com/potato4d/nuxt-tech-book/tree/master/examples/section03/03_Tutorial

前提となる知識と注意

これ以降のセクションでは、Nuxt のベースとなる Vue.js へのある程度の理解が必須となります。まだ Vue.js を触ったことがないかたは、この機会に一度 Vue.js の基礎を体験してからお読みいただくことで、より一層 Nuxt についての理解が進むでしょう。

また、既に Nuxt を使ったWebアプリケーション開発を行ったことがあるかたは、このセクションは基礎だけを扱うため、飛ばして次のセクションから読んでも問題はありません。

より実践的な機能について知りたいかたはセクション後半もしくは次の「Nuxt の機能をフル活用する」からお読みください。

事前準備

実際にプロジェクトを作成していく前に、以下の環境を揃えておいてください。

Node.js

本書では Node.js v8.9.4 (LTS) を対象として、導入されていることを前提として進めます。バージョンの確認はターミナルから以下で確認可能ですので、必ずバージョンをあわせた上で進めてください。

```
terminal
```

```
$ node -v
v8.9.4
```

Yarn

また、パッケージマネージャには Yarn を利用します。NPM と比較して、キャッシュや並列でのインストールにより高速であることのほか、nuxt本体の依存の固定化が package-lock.json ではなく yarn.lock で行われているため、そういった意味でもYarnを利用すべきでしょう。必須ではありませんが、可能な限りインストールしておいてください。

terminal

```
$ yarn -v
1.5.1
```

@vue/cli

また、このサンプルでは vue-cli で利用できる Nuxt 向けボイラープレートである nuxt-community/starter-template を利用してプロジェクトを初期化します。そのため、同時に vue-cli もインストールもしておいてください。

terminal

```
$ npm i -g @vue/cli
$ npm i -g @vue/cli-init
$ vue -V
3.0.0-alpha.10
```

本書執筆時点では vue-cli 3.0が開発中であるために vue-cli 2.x の互換機能である `vue init` を利用しますが、正式公開後は環境構築が大きく変わることが予想されます。もし古い情報である場合は、本書「はじめに」に記載されているGitHubレポジトリを参考に、適宜読み替えてください。

Vue.js devtools

Vue.js devtools は、Chrome / Firefox 向けの拡張機能で、Vue.js コンポーネントの DOM ツリーや、Vuex ストア、イベントログなどを覗き見・改ざんできる拡張機能となります。

Vue.js での開発では必須ともいえるツールですので、導入しておきましょう。

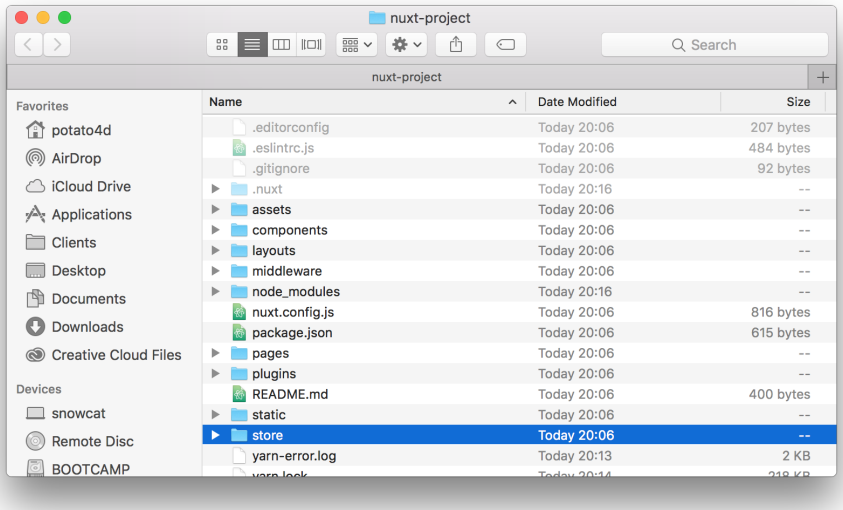
Nuxt プロジェクトのセットアップ

まずはプロジェクトを初期化します。適当なディレクトリに移動したのち、以下のコマンドでボイラープレートを展開できます。

terminal

```
$ vue init nuxt/starter nuxt-project
```

いくつか質問が出ますが、特に今回は変える必要もないでしょう。全てEnterしてください。以下のようなプロジェクト構成のディレクトリが作成されていると完了です。



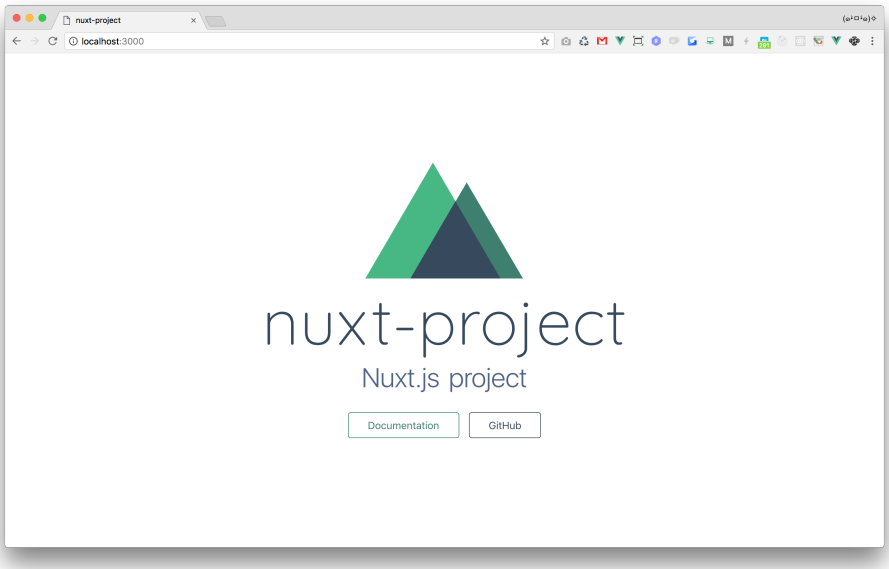
続いて、vue-cli の出力メッセージの通り、パッケージのインストールと実行まで行います。以下のようにコマンドを実行してください。

terminal

```
$ cd nuxt-project
$ yarn
$ yarn dev
```

`yarn dev` は `yarn run dev` のエイリアスとなっています。基本的に Yarn を利用する場合、runは省略可能ですので、楽するためにも省略しておくと良いでしょう。`yarn dev` を実行したことにより、開発サーバーが立ち上がります。

`[OPEN] http://localhost:3000` と表示された場合、初回ビルドとサーバーの立ち上げが完了した合図となりますので、そのまま localhost:3000 を開きましょう。VueのロゴがNuxtのロゴに変わるアニメーションつきのページが表示されることでしょう。ここまでで環境構築は完了となります。



Nuxtのプロジェクト構成について

プロジェクトのセットアップが完了したので、各ディレクトリの構造について、AtoZ順に簡単にご紹介します。

なお、layouts, middleware, pluginsについては、特殊な概念となりますので、ここでは取り扱わず、4章以降で詳しくご紹介します。

assets ディレクトリ

画像リソースや設定のJSONファイルなどのアプリケーションのソースコード外のものを管理します。

ここにあるものは、基本的にwebpackのfile-loader経由で読まれることを前提として利用することになりますので、例えば動的なURLの組み立てなどは不可な反面、宣言的に読み出せるものはアセットを最適化して利用できますので、可能な限りここに素材はおくと良いでしょう。

components ディレクトリ

.vueで終わる、Vueコンポーネントを管理するディレクトリとなります。

基本的にはここに全て集約していくこととなるので、適宜サブディレクトリを切って分類すると良いでしょう。

starter-template では `AppLogo.vue` が存在しています。

pages ディレクトリ

それぞれのページコンポーネントを配置するディレクトリとなります。

VueRouterなどをそのまま使う開発においても専用のページコンポーネントのディレクトリを設けることは多いかと思いますが、Nuxtの場合は、単なる明文化のためだけではなく、このディレクトリ配下はルーティングの自動生成の対象となる特別な意味をもつディレクトリとなっています。

starter-template では index.vue のみが存在しています。試しに書き換えて動作を見てみるのも良いでしょう。

static ディレクトリ

公開用の静的リソースを配置する場所となります。

starter-template で favicon.ico があるように、このディレクトリでは webpack によるファイル名のハッシュ化の影響を受けないので、固定の名前を用意したい静的リソースはここに配置すると良いでしょう。例えば、OGP 画像や、apple-touch-icon などが挙げられます。

store ディレクトリ

ここには Vuex ストアとそのモジュールのファイルを配置します。ここに配置されたストア及びモジュールは、Nuxt のストアオートローディングの対象となり、ファイルを作成するだけでそのモジュールをグローバルで利用可能となるなど、特別な扱いを受けることとなります。

詳しい挙動に関しては、この章の最後でご紹介いたします。

ルーティングとページコンポーネントの作成

まずはルーティングから作成、Nuxtの画面の構築について一通り学んでから、ロジックの実装を進めていくこととしましょう。

Nuxtのルーティング自動生成システムについて

次は簡単なルーティングとページのコンポーネントを作成してみましょう。Nuxt は、`pages` ディレクトリ内にディレクトリやファイルを作成すると、それに沿ったルールでルーティングを作成してくれます。例として、以下のような構造の場合

directory

```
pages/
--| index.vue
--| about.vue
--| users/
-----| index.vue
-----| _id.vue
```

以下のようにURLを解決してくれます。

directory

```
/          -> index.vue
/about     -> about.vue
/users/    -> users/index.vue
/users/1   -> users/_id.vue
```

index.vueは `/` を、 `_id` といった、 `_` から始まるものは、 `/users/:id` 形式をサポートしてくれます。この際、名称は `_id` に限った話ではなく、例えば `_name` や `_slug` なども可能です。この違いは、後述するルーティングパラメータの変数名の違いとなります。

また、単純な about.vue などは、そのまま `/about` をサポートするため、例えば users 配下に about.vueを作成すると、 `/users/about` にてアクセスが可能となります。

実際のルーティングファイルの作成

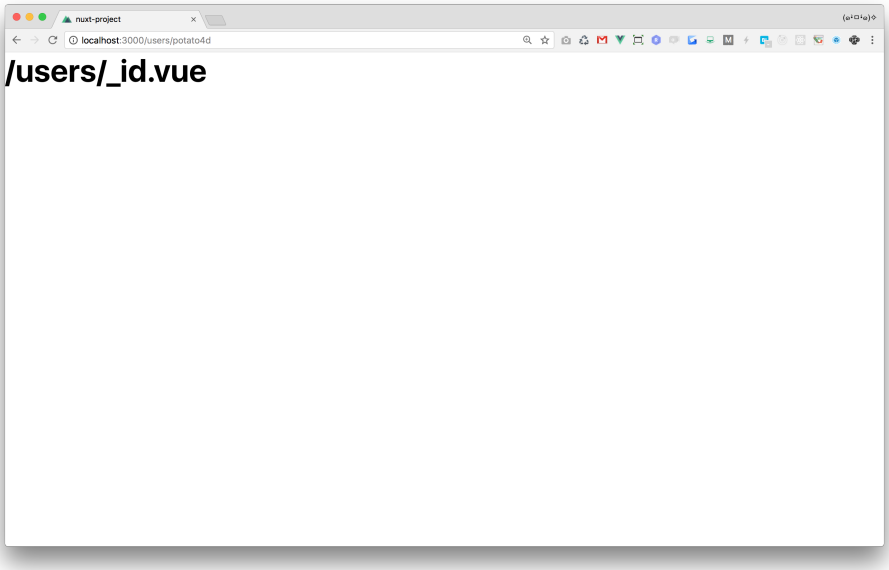
上記を踏まえて、実際にルーティングのファイルを作成してみます。今回はGitHubのユーザー情報を表示するアプリケーションのサンプルですので、 `pages/users/_id.vue` 辺りがあると十分でしょう。

`pages/users/_id.vue` を作成し、以下のようにコードを書いてみましょう。

_id.vue

```
<template>
  <div>
    <h1>/users/_id.vue</h1>
  </div>
</template>

<script>
export default {
}
</script>
```

その上で、http://localhost:3000/users/potato4d など、/users/:id 形式のURLにアクセスすると、期待通り、以下のように先程記述した h1 が表示されていることがわかるかと思います。



ルーティングに応じたコンテンツの出し分け

さて、これで `/users/:id` 形式のルーティングを全て一つのファイルで受けることが可能となっていますが、実際のコンテンツの表示出し分けも行ってみます。

コンテンツの出し分けの際にルーティングの情報を取得するには、通常のVue.jsのdataメソッドではなく、asyncDataメソッドを利用します。

これはNuxtが独自に処理するdataメソッドの拡張メソッドとなっており、これを利用することで、Vuex ストアやルーティングのパラメータ、リダイレクト関数などにアクセスすることが可能となっています。

サーバーサイドレンダリングの時に初期データをフェッチしたい、302リダイレクトを発生させたいなどのモチベーションが発生した際に有効に活用できるため、Nuxtのページコンポーネントでは、基本的に必ずdataのかわりにasyncDataを利用するようにしてください。

実際のルーティングパラメータの取得は、以下のように行います。分割代入を利用することでスマートにデータを取得できるので、活用すると良いでしょう。今回は、以下のように書いてみましょう。

_id.vue

```
<template>
  <div>
    <h1>{{id}}</h1>
  </div>
</template>
```

```
<script>
export default {
  asyncData ({ params }) {
    const { id } = params
    return {
      id
    }
  }
}
</script>
```

実行すると、以下のように URL に応じてコンテンツが変わっていることがわかるかと思います。適当に URL の id の部分を変えて変更させてみましょう。

これで実際のコンテンツの出し分けが可能となりました。ここまでできたら、あとは id の値に応じてデータを取得するだけで完成することでしょう。このまま外部リソースの取得に進みます。

asyncData と axios-module による外部リソースの取得

続いて先程書いた asyncData と、HTTP 通信のライブラリを組み合わせて GitHub APIを叩いてみましょう。

axios-module の導入

今回のチュートリアルでは、HTTP 通信のライブラリとして、人気の高い Isomorphic なライブラリである axios の公式 Nuxt ラッパーである axios-module を利用します。

axios-module は、Vue コンポーネントのシームレスな連携、Intercepter や、retry などを提供しており、非常に優秀なモジュールとなっています。そのため、Nuxt での開発のときは積極的に利用することをオススメします。

まずは Yarn を利用して axios-module を導入してください。

terminal

```
$ yarn add @nuxtjs/axios
```

その上で、nuxt.config.js を開き、loading や build と同じ階層に以下を追加してください。

nuxt.config.js

```
  ...
  loading: { color: '#3B8070' },
+++ modules: [
+++   '@nuxtjs/axios'
+++ ]
  /*
  ** Build configuration
  */
  build: {
  ...
```

これで axios が読み込まれ、Vue コンポーネントや Vuex ストアからそのまま呼び出すことができるようになります。

GitHub の Personal Access Token の取得

これは必須ではありませんが、GitHub API へのアクセスは 1IP あたり 1時間に 60 回、Access Token が存在する場合は 1000 回となります。Nuxt の場合はホットリロードが存在する都合上、この API 制限を超過することは頻繁にありますので、超過してエラーとなることを考えてできれば取得しておくと良いでしょう。

まずは https://github.com/settings/tokens にアクセスし、Generate new token を選択。Select scopes の repo にチェックを入れたものを生成しましょう。Description も適当に書き終わったら、そのまま Generate Token をしてください。

そうすると、以下のような表示となり、トークンが払い出されます。スクリーンショットではマスクしていますが、英数字のトークンが発行されるはずです。これを忘れずに控えておいてください。

axios-module の Intercepter による認証情報の追加

折角なので先程取得したトークンと、axios-module を利用して、Intercepter による処理の注入を試してみます。ここでは、GitHub の API 制限の緩和のために、`https://api.github.com` 宛のリクエストに関して Authorization ヘッダーを付与することとします。

まずは `plugins/axios.js` を追加。以下のように記述してください。

axios.js

```
export default function ({ $axios }) {
  $axios.onRequest( (config) => {
    if (config.url.indexOf('api.github.com') +1 ) {
      config.headers.Authorization = `token XXXXXXXXX`
      // XXXX is your access token
    }
  })
}
```

その後、nuxt.config.js に以下のように変更を施してください。

nuxt.config.js

```
  ...
  modules: [
    '@nuxtjs/axios',
  ],
+++ plugins: [
+++   '~/plugins/axios'
+++ ],
  ...
```

こうすることによって、$axios.onRequest に Intercepter が追加され、ドメインのチェックを行った上で認証ヘッダーを付けることが可能となりました。これで全ての準備は完了となります。

axios-module で GitHub API を叩く

それではいよいよ axios-module 経由でページコンポーネントから GitHub API を叩いてみます。`pages/users/_id.vue` を以下のように書き換えてください。

_id.vue

```
<template>
  <div>
    <h1>{{user.name}}</h1>
    <img :src="user.avatar_url">
  </div>
</template>

<script>
export default {
  async asyncData ({ params, app }) {
    const { id } = params
    const user = await app.$axios.$get(
      `https://api.github.com/users/${id}`
    )
    return { user }
  }
}
</script>
```

以下のように表示されていると成功です。

このように、axios-moduleで導入した axios は、Nuxt の app オブジェクトの配下として自動的に登録されるため、明示的な import を必要とせず、気軽に叩くことが可能です。

Vuex ストアへのデータの取り扱いの委譲

最後に、Vuex ストアとの連携についてご紹介します。Nuxt では、Vuex のモジュールを自動的に読み込んでくれるモジュールモードという便利なモードがありますが、単機能の場合は特に必要がないので、今回はクラシックモードを利用します。モジュールモードの利用については、次章以降を参考にしてください。

Nuxt の Vuex ストアのクラシックモードは、`store/index.js` にファイルを配置し、規定の記述を行うだけで自動的にモジュールなし・単一ストア／ステートの Vuex ストアを作成してくれる機能となります。

まずは最小限のストアを作成してみましょう。最小限の Vuex ストアは、以下のようなソースコードになります。

index.js

```
import Vuex from 'vuex'
const store = () => new Vuex.Store({
  state: {
    user: null
  },
  mutations: {},
  actions: {}
})
export default store
```

通常、Vue.js アプリケーションから Vuex を利用する場合は、ストアを import した上で `Vue.use(Vuex)` 後に `new Vue` に追加する必要がありますが、その作業は全て Nuxt 側で行われます。

`stores/index.js` を作成すると、すぐに HMR がかかり、ブラウザの Vue.js devtools にストアが表示されるようになるはずです。

ここまでできたらソースコードを移植するだけです。今回は Vuex 自体の詳しい解説は省きますが、おおよそ通常通りの使い方で利用できるようになっています。

まずは `store/index.js` を以下のように書き換えてください。

index.js

```
import Vuex from 'vuex'

const store = () => new Vuex.Store({
  state: {
    user: null
  },
  getters: {
    user: (state) => state.user
  },
  mutations: {
    saveUser (state, { user }) {
      state.user = user
    }
  },
  actions: {
    async getUser ({ commit }, { id }) {
      try {
```

```
        const user = await this.$axios.$get(
          `https://api.github.com/users/${id}`
        )
        commit('saveUser', { user })
      } catch (e) {
        return Promise.reject(e)
      }
    }
  }
})

export default store
```

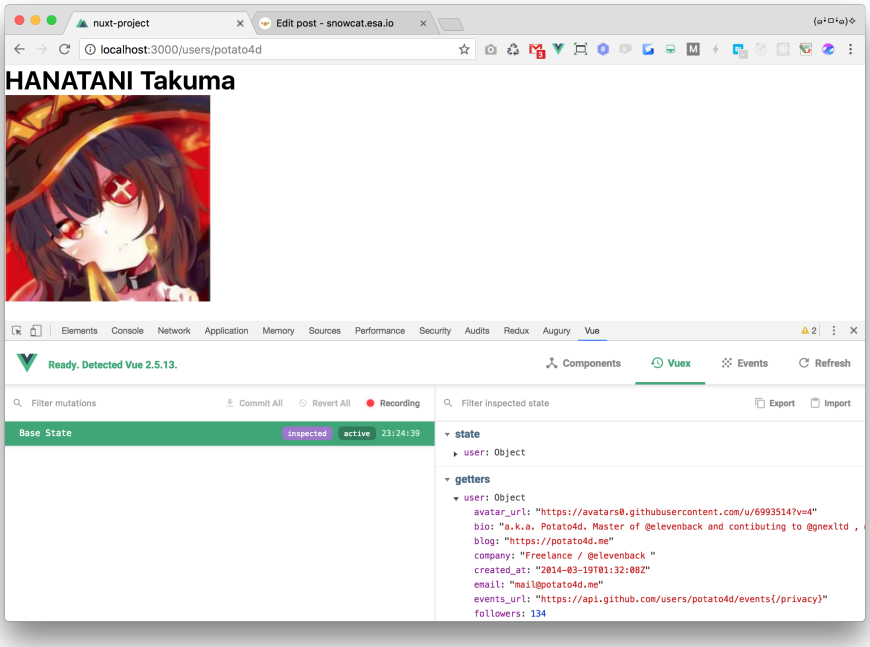
その上で、ページコンポーネント `pages/users/_id.vue` 側は以下のように書き換えましょう。特徴的な記述として、asyncData 実行時はまだ methods や computed にアクセスができないため、Action は必然的に asyncData に引数としてついてくる store から直に dispatch することとなります。

_id.vue

```
<template>
  <div>
    <h1>{{user.name}}</h1>
    <img :src="user.avatar_url">
  </div>
</template>

<script>
import { mapGetters } from 'vuex'

export default {
  async asyncData ({ params, store }) {
```

```
    const { id } = params
    await store.dispatch('getUser', { id })
    return {}
  },
  computed: {
    ...mapGetters(['user'])
  }
}
</script>
```

実際に実装すると、以下のように同じ見た目ながら、Vue.js devtools でみるときちんと Vuex が適用されているのがわかるかと思います。



これで Nuxt を利用した簡単な Web アプリケーションの実装の説明は以上となります。今回は Nuxt の機能の紹介のために外部 API を利用した GET のみでご紹介しましたが、勿論 axios-module と Vuex を組み合わせることによって POST や PUT, PATCH などのデータとその結果を取り扱うことも可能です。

ここから広げていくには?

この章において一通りの構造は掴んでいただけたかと思いますが、本格的なアプリケーションをうまく実装するには、レイアウトの共通化や Vuex のストア分割など、追加で行っていくべきことも多くあります。

豊富な機能とそれによる高い生産性は Nuxt の醍醐味ともいえますので、是非試しに小規模なアプリケーションを作りながら、次以降の「 Nuxt の機能をフル活用する」や「実践的なWebアプリケーション開発ノウハウ」を読み進めてみてください。

4. Nuxt の機能をフル活用する

前セクションにて、Nuxt の基本的な機能と使い方についてご紹介しましたが、ここからはより実践的な Nuxt の機能を機能単位で一つずつご紹介していきます。

いずれも Vue 単体の開発では頭を悩ませることの多い課題であり、その課題への解決策を Nuxt が提示してくれているポイントとなりますので、存分にご活用して効率的なアプリケーション開発に役立てていただけると幸いです。

また、この章に登場するサンプルコードは、全て nuxt-community/starter-template をベースで動かすことで、実際に動作させることが可能となっています。

layouts によるレイアウトの共通化

Nuxt には、ページごとの共通レイアウトを、Vueコンポーネントとして再利用するレイアウト機能が実装されています。

例えば、「基本的にはナビゲーションやヘッダーを共通化して表示したいけれど、ルートコンポーネント上に定義してしまうとログイン画面やトップページなど、表示したくないページで表示されてしまい不便。とはいえその解決のために下手に `v-if` で分岐すると次はルートのコンポーネントが汚れてしまう。」

といった、「共通化したいモチベーション」と「共通化した場合に管理コストが増大する」という問題が共存している場合には、layouts を利用することで解決が可能です。

その他にも、投稿型システムの単一投稿の表示ページにおいてレイアウト側で極力UIを定義しておくことで、投稿の本体をクリーンに保った上で単一投稿の表示UIを効率的に使いまわすなど、「特定のUIの形式をテンプレート化したい」という時にも利用でき、非常に便利な機能となります。

レイアウトのルールと default.vue の編集

それでは実際にレイアウトを利用してみましょう。Nuxt は、デフォルトのレイアウトとして default.vue を所持しており、特にレイアウト指定がない場合は default.vue が参照される仕様となっています。

今回は試しに、`components/AppNavigation.vue` というナビゲーションのコンポーネントを作成し、default.vue から読み込んでみます。

まずは `components/AppNavigation.vue` に、以下のコードを記述してください。

AppNavigation.vue

```
<tempalte>
  <ul>
    <li><router-link to="/">home</router-link></li>
    <li><router-link to="/child">child</router-link></li>
  </ul>
</template>
```

その上で、`layouts/default.vue` に以下のように記述してください。

default.vue

```
<template>
  <div>
    <AppNavigation />
    <nuxt />
  </div>
</template>

<script>
import AppNavigation from '~/components/AppNavigation.vue'

export default {
  components: {
    AppNavigation
  }
}
```

この状態で、ページへとアクセスすると、`pages/index.vue` には何も書かれていないにもかかわらず、ナビゲーションがレンダリングされることがわかります。

折角なのでナビゲーションにある `pages/child.vue` を作成して、こちらにもアクセスしてみましょう。`pages/child.vue` のソースコードは以下のようにしてください。

child.vue

```
<template>
  <div>
    <h1>Child</h1>
  </div>
</template>
```

この状態で `/child` に作成されても、フォールバックである default.vue が読み込まれていることがわかるかと思います。このように、レイアウトを設定することで、簡単に共通 UI を作成することが可能です。

今見ているページのレイアウトは Vue.js Devtools でも確認可能

また、今見ているページのレイアウトが本当に期待通りかどうかは、Vue.js Devtools からも確認できます。レイアウトもコンポーネントであるため、Vue.js Devtools 上で名前がついて表示されます。

ソースコードを追うまでもない場合においては、こちらも有効活用すると良いでしょう。

レイアウトファイル設計のベストプラクティス

このように、一見非常に便利なレイアウトシステムですが、多少クセがあります。

クセを適切に把握しない場合、破綻の原因となりますので、最後にラクにレイアウトを管理するためのオススメの方法をご紹介いたします。

starter テンプレートを初期化した時など、基本的には汎用レイアウトとトップページから編集をはじめることが多いかと思いますが、基本的にトップページはLPであったり、ダッシュボードのサマリーページであったり、他のページとは大きく異なるデザインを適用することがほとんどです。

そういったときに、もし default.vue をトップページ用に、それぞれの個別のページを single.vue などとした場合、全てのページコンポーネントに対して `layout: 'single'` を指定する必要があり、適用忘れなどのミスの原因となります。

これを防ぐためにも、default.vue はアプリケーション全体で使われるレイアウトとし、こちらに single 相当の記述を、トップページのために、専用の home.vue などのレイアウトファイルを作成して、トップページだけ `layout: 'home'` 指定するのが良いでしょう。

こうすることで、トップページは専用の home.vue により完全に隔離された上で、煩雑なレイアウトの設定も行う必要がなくなるため、ミスも減りますし、管理しやすい構成となります。

default.vue は大きなプロジェクトになると使わないほうがわかりやすい場合もありますが、小中規模ではフォールバック先として有効活用しましょう。

middleware を利用したルーティング結果の改ざん

次はミドルウェアです。サーバーサイドのプログラミングを行ったことがあるかたにはおなじみの、request / responseオブジェクトに対して、そのルーティングの実処理の前後に割り込み、request / responseオブジェクトを改ざんすることができる機能となります。

例えば、この機能を利用することで、

- Cookie に格納された認証トークンを確認した上で、正しい認証トークンの場合は Vuex のストアに認証情報を追加する機能

- 日本人が英語ページにきた際に自動で日本語ページへとリダイレクトをする機能

などの、Webアプリケーションの開発で頻繁に利用される割り込みを非常に簡単に実装することが可能となります。

特に、Nuxt では、request / responseオブジェクトの改ざんだけでなく、上述の通りVuexストアへのアクセスや、302リダイレクトの発行などが可能となっているため、非常に柔軟に利用することが可能です。

実際の Cookie 認証のサンプルは次の章で詳しくご紹介します。より実践的な記述についてはそちらをご参照ください。

ここでは特定のパスの場合のみリダイレクトするという簡単なサンプルをもとに、基本的な使い方を網羅する形でご紹介します。

最低限のミドルウェア雛形の作成

ミドルウェアは、`middleware/redirector.js` というかたちで、middleware 配下に作成します。

redirector.js

```
export default function ({ redirect, route }) {
  if (route.path == '/users/2') redirect('/')
}
```

その上で、グローバルで使う middleware は、nuxt.config.js に追記します。

nuxt.config.js

```
module.exports = {
//...
  router: {
    middleware: ['redirector']
  }
//...
}
```

また、ページコンポーネントに直接記述すると、そのページコンポーネントにアクセスされる時限定のミドルウェアを作成することも可能です。

例えば、`pages/users/_id.vue` だけに以下のような記述を加えると、`middleware/redirector.js` 側で path を参照せずとも、`users` 配下である前提で処理を記述することが可能です。

_id.vue

```
<script>
export default {
  middleware: 'redirector'
}
</script>
```

ミドルウェアの注意点

最後に、非常に強力な機能であるミドルウェアですが、2つほど注意点があるためご紹介します。

SSR 前提という原則と fetch フック

ミドルウェアは、requestオブジェクトへのアクセスを必要とする場合があるという都合上、SPAモードでは動作しません。SSRモードにおいても、初期アクセス時のみ有効な機能となっており、SPAモードに一旦移行してしまった場合、ミドルウェアは機能を失います。

SSR によるレイテンシ

また、勿論ですが、ミドルウェアの実行はサーバー側の作業の一環となりますので、外部APIへのフェッチなどを行う際は、レスポンス速度に多少なりとも影響があります。

HTTP リクエストの中で、更に HTTP リクエストが発生するということを十分に理解した上で利用すると良いでしょう。

筆者としては、ミドルウェアの利用はルーティングの認証機能の実装程度に留めておくことを強くオススメします。

plugin によるグローバルな機能拡張

ミドルウェアと似ているようで大きく違う、プラグイン機能についてもご紹介いたします。このプラグイン機能は、初期ロード時に自動で読み出され、特定の処理を行うというものです。

一見ミドルウェアと同じような仕組みに見えますが、利用用途は全く異なります。ここでは、簡単な例と共に、プラグインについてご紹介します。

プラグインの使いどころ

プラグインは、ミドルウェアのページの処理の前に挟まるフックではなく、単純にグローバルな機能拡張を行うときに利用します。

前述の通り、ミドルウェアと同じく、初期ロード時に読み出され、処理が実行されますが、ここでルーティングに対する改ざんを行うのではなく、グローバルな Vue ライブラリの import や、VueRouter のフック登録などに使います。あくまでもリクエスト・レスポンスが主体ものはミドルウェアに、アプリケーションに対してのライブラリ登録など、リクエストやレスポンスの内容に直接関係のないものは、プラグインに実装することとなります。

例えば、グローバルに影響するものであると「Google Analytics」の SPA でのロギングの有効化などがあります。ここでは、ロギングなどでは必須の、プラグインからの VueRouter フックの登録についてご紹介します。

プラグインの作成

今回は試しに VueRouter の beforeEach フックを利用して、ページ遷移が行われるたびにそのルーティングパスを `console.log` で表示するコードを書いてみます。

プラグインのコードは、plugins ディレクトリ以下に作成する必要があり、ここで作成されたものはプラグインとして利用が可能となります。

まずは `plugins/logger.js` を作成し、以下のように記述してみましょう。

logger.js

```
/* plugins/logger.js */
export default ({ app }) => {
  app.router.beforeEach((to, from, next) => {
    console.log(`move to "${to.fullPath}`)
    next()
  })
}
```

基本的には VueRouter を直で使う場合と同じく、3 つの引数を受け取った上で、before の場合は `next()` を実行しますが、その際の Router オブジェクトは app から降ってくることにご注意ください。

app を取得している関数の第一引数は、ミドルウェアや asyncData と同じく、コンテキストを受け取ります。ここには redirect や Vuex ストアも格納されているため、適宜読み出すと良いでしょう。

プラグインの登録と実行

プラグインが実装できたら、nuxt.config.js を書き換えます。export するオブジェクトに plugins を作成し、以下を追加しましょう。

nuxt.config.js

```
{
  // ...
  plugins: [ '~/plugins/logger' ]
  // ...
}
```

このように、plugins というキーで配列を作成することで、ここに登録されたプラグインは初期ロード時に実行されるようになります。

実際に動かしてみると、以下のスクリーンショットのように、ページ遷移のたびに `console.log` によってパスが表示されていることがわかるかと思います。Google Analytics や Mixpanel などのトラッキングを実装する必要がある場合は、このように beforeEach や afterEach と組み合わせてやると良いでしょう。

Vuexのモジュールモードを活用したオートローディング

次に、Vuex のオートローディングについて説明いたします。前章でも軽く説明しましたが、ここではより深く、実践的な利用方法についてご紹介いたします。

Vuexストアへのアクセスの二つのモード

Nuxt の Vuex ストアには クラシックモード と モジュールモード という二つのモードがあります。以下に簡単に違いについて説明します。

クラシックモードは、単一の名前空間をもつ大きな Vuex ストアをベースに、もしモジュールが必要であれば適宜手動で追加していく、通常の Vue 開発に近いモードです。

モジュールモードは、Nuxt の store ディレクトリ内にファイルを作成してゆき、Vuex ストアインスタンスではなく、state や mutation 、action などをそれぞれ別にエクスポートすることで、Nuxt が自動で全てのファイルを名前空間付きのモジュールとして解釈し、Vuex ストアインスタンスにまとめてくれるモードとなります。

筆者としては、数ページ規模の小規模アプリケーションまではクラシックモードで十分ですが、そういったもの以外ではモジュールモードの利用を強くオススメしています。

理由としては、ルーティングの自動生成などと同じく、それぞれのファイルの本質的な記述のみに専念することができ、また、namespaced オプションのつけ忘れなどのケアレスミスも防ぐことができるためです。

モジュールモードでの Vuex の利用

実際にモジュールモードにおいての Vuex の利用方法をご紹介します。

モジュールモードの Vuex では、stores ディレクトリに入っているファイルを基に自動判別し、ストアを生成します。その際のルールは、`index.js` がルートモジュール、それ以外の `.js` ファイルは、全てファイル名が名前空間となったモジュールとして登録されます。

試しに、`index.js` と `users.js` があるシチュエーションの例を記載しておきます。

まずは `index.js` から。`index.js` には、モジュール化するほどではない汎用のステートが書かれていることを想定し、簡単なローディングの状態などを書いていることとします。

index.js

```
export const state = () => ({
  isLoading: false
})

export const mutations = {
  setIsLoading (state, isLoading) {
    state.isLoading = isLoading
  }
}
```

その上で、例えば `users.js` に以下のようなユーザーリストの管理があったとします。

users.js

```
export const state = () => ({
  list: []
})

export const mutations = {
  addUser (state, user) {
    state.list.push(user)
  }
}

export const actions = {
  addUser ({ commit }, { user }) {
```

```
    commit('addUser', user)
  }
}
```

こうした構造の場合、自分で以下のような Vuex ストアを作成したときと同じように動作します。

```
new Vuex.Store({
  state: { isLoading: false },
  mutations: {
    setIsLoading (state, isLoading) {
      state.isLoading = isLoading
    }
  },
  modules: {
    users: {
      state: {
        list: []
      },
      mutations: {
        addUser (state, user) {
          state.list.push(user)
        }
      },
      actions: {
        addUser ({ commit }, { user }) {
          commit('addUser', user)
        }
      }
    }
  }
})
```

結果として出力されるものは同じであるため、名前空間付きで map メソッドを呼びたい時は `mapActions('users', ['addUser'])` などの形式で取得することが可能です。

通常開発を行う際に煩雑であり忘れやすい Vuex のモジュールの管理を自動化できるため、Nuxt での開発を行う際はこのモジュールモードを有効活用しましょう。

<no-ssr> と process.browser

次は <no-ssr> と process.browser のご紹介です。

これらは、Nuxt によって独自実装されている SSR のためのシステムとなります。本来であればクライアント・サーバー間で同一のコードを用いるべきですが、window オブジェクトを参照するものなど、両立できないコードもあります。

<no-ssr> と process.browser は、そんなときに役立つシステムとなります。

<no-ssr> コンポーネント

このコンポーネントはその名の通り、<no-ssr> 内の HTML については全てについてレンダリングをスキップし、代替テキストを表示するためのコンポーネントとなります。

このコンポーネントは、Nuxt 本体のものではなく、`egoist/vue-no-ssr` を移植したものとなりますが、Nuxt 側でグローバルに定義されているため、別途 import の必要なく利用が可能となります。

どうしても SSR をさせたくない物がある場合は、こちらを利用すると良いでしょう。

no-ssrの利用例.vue

```
<template>
  <no-ssr>
    content
  </no-ssr>
</template>
```

process.browser

process.browser は、Node.js の process オブジェクトに拡張された boolean 型の変数となります。

これは、SSR 実行時は false に、ブラウザ上での、SPA としての動作時は true へと、自動で Nuxt が切り替えてくれる変数となります。

window オブジェクトを参照する際は、該当処理の前に process.browser の値を if で評価してやるなどによって、SSR 時にエラーになることなく該当オブジェクトへの参照が可能となります。

基本的にはコンポーネント全体を process.browser で切り分けたい場合は <no-ssr> で問題ありませんが、レイアウトの計算などで一部だけ window オブジェクトを利用している場合は、こちらを利用すると良いでしょう。

エラーページのカスタマイズ

最後に、開発が完了した頃にあとから必要となりがちな、エラーページのカスタマイズについてご紹介いたします。

Nuxt では、4xx および 5xx エラーがでた際に、自動で `layouts/error.vue` を表示する機能が搭載されています。これによって通常のフレームワークであれば 404 だけであればフォールバックで対処できますが、幅広い対応となると少し手間がかかるハンドリングが必要となりますが、Nuxt では `error.vue` に全てのエラー情報が渡ってくるため、一つのテンプレートでまとめて管理することが可能となっています。

また、この `layouts/error.vue` は、レイアウトファイルではありますが、実体はページファイルとよく似ています。`<nuxt>` タグを持つことはなく、他のレイアウトをベースとして指定することができます。ここでは、簡単なエラー画面の実装例のみをご紹介します。サンプルの `error.vue` には、404 とそれ以外の表示の出し分けを行っています。

error.vue

```
<template>
  <div class="container">
    <template v-if="isNotFound">
      <h1>404 not found</h1>
      <nuxt-link to="/"> Back to home </nuxt-link>
    </template>
    <template v-else>
      <h1>Error</h1>
      <nuxt-link to="/"> Back to home </nuxt-link>
    </template>
```

```
  </div>
</template>
<script>
export default {
  props: ['error'],
  computed: {
    isNotFound () {
      return this.error.statusCode == 404
    }
  }
}
</script>
```

5. 実践的なWebアプリケーション開発ノウハウ

ここまで、基本的なアプリケーション開発と、Nuxt の特徴的な機能についてご紹介しました。

この章では少しコアな、シチュエーション単位の実践的な開発のノウハウについてご紹介します。先ほどまでは機能単位で紹介をすすめましたが、シチュエーション単位での実例を見ることにより、より実際の開発がイメージしやすくなることでしょう。

認証つきルーティングの機能の実装

認証つきルーティングについてご紹介します。Web アプリケーションを開発する場合、閲覧できるユーザーを制限する必要のある画面は頻繁に現れます。

例えば、会員登録を前提としたサービスの場合、ログインしていないユーザーをログインページにリダイレクトする必要があるでしょう。

今回は、そういった場合の対処ケースとして、「オーソドックスな API のトークン認証をベースとし、Cookie に Bearer トークンを保持していない場合は、特定のページにリダイレクトさせる」という機能を実装してみます。

この機能は、Nuxt のミドルウェア、もしくは Vuex の nuxtServerInit アクションを利用することで、非常に簡単に実装することが可能です。

共通利用する Cookie のインストール

Nuxt からの Cookie の読み書きは、NPM モジュールの universal-cookie を利用することで実装できます。

通常のフロントエンドというと、最近では localStorage でのデータ永続化が一般的ですが、Nuxt の場合、SSR と SPA でデータを共有できるという点で、Cookie を利用しておくと非常に便利です。

例によって例の如く CLI から導入しましょう。

terminal

```
$ yarn add universal-cookie
```

universal-cookie モジュールは、フロントエンド・サーバーサイド両方で利用可能な、Universal なクッキー操作のためのライブラリです。

非常に手軽にコードを SSR と SPA で共通化できるため、Nuxt 開発では、積極的に利用すると良いでしょう。

ミドルウェアベースでの認証の実装

早速サンプルのコードをみながら実装を進めてみましょう。`middleware/auth.js` を作成し、以下のように記述してください。

```
import Cookies from 'universal-cookie'

export default function ({ req, route, redirect, store }) {
```

```
  if (!process.server || ['/login'].includes(route.path)) {
    return
  }

  const cookies = new Cookies(req.headers.cookie)
  const credential = cookies.get('credential')

  if (credential) {
    // Main logic here...
    // e.g. store.dispatch('setToken', credential)
  } else {
    return redirect('/login')
  }
}
```

以上でミドルウェアは実装完了です。実装したミドルウェアは、そのままでは動作しませんので、ルーティングに紐付ける必要があります。

肝心のヒモ付ですが、ミドルウェアはページごとに設定するか、nuxt.config.js でグローバルに設定するかを選択することが可能です。

認証に関しては、ほぼすべてのルーティングにおいて必要となるはずですので、グローバルに設定してしまいましょう。

その際、上記のコードのように除外ルーティングだけをミドルウェアの本体に書いてしまうと円滑でしょう。

グローバルに設定する際の nuxt.config.js のコードは以下となります。

nuxt.config.js

```
module.exports = {
//...
  router: {
    middleware: ['auth']
  }
//...
}
```

試しに、`pages/index.vue` と `pages/login.vue` を用意して動作させてみるとわかりやすいでしょう。

`./pages/index.vue` を以下のように。

index.vue

```
<template>
  <section>
    <h1>HOME</h1>
    <p>
      <nuxt-link to="/login">
        Move to login
      </nuxt-link>
    </p>
  </section>
</template>

<script>
import AppLogo from '~/components/AppLogo.vue'

export default {
  components: {
    AppLogo
```

```
  }
}
</script>

<style scoped>
section {
  margin: 16px;
}
</style>
```

`./pages/login.vue` を以下のようにしてやるとわかりやすいでしょう。

login.vue

```
<template>
  <section>
    <h1>Login</h1>
    <p>
    <button type="button" @click="addCredential">
      Set credential
    </button>
    <button type="button" @click="removeCredential">
      Remove credential
    </button>
    </p>
    <p>
      <a href="/">Move to home</a>
    </p>
  </section>
</template>

<script>
import Cookies from 'universal-cookie'
```

```
let cookies

export default {
  mounted() {
    cookies = new Cookies()
  },
  methods: {
    addCredential() {
      cookies.set('credential', '1')
    },
    removeCredential() {
      cookies.set('credential', '')
    }
  }
}
</script>

<style scoped>
section {
  margin: 16px;
}
</style>
```

このように実装することで、非常に手軽に認証を実装することが可能となりました。実際の現場では、axios-module と併用して、API コールすることで 401 となるかのチェックを行っても良いでしょう。

最後に、これらを実際に実装したサンプルデモ及びソースコードが以下にありますので、適宜ご利用ください。

- Demo: https://potato4d.github.io/nuxt-tech-book/examples/section05/05_Auth_with_Middleware

- GitHub: https://github.com/potato4d/nuxt-tech-book/tree/master/examples/section05/05_Auth_with_Middleware

nuxtServerInit での認証の実装

また、もう一つの実装パターンとして、nuxtServerInit を利用する方法があり、こちらは Vuex に完全に載せることになります。

nuxtServerInit は、Vuex のルートモジュールに実装された "nuxtServerInit" の名前をつけられた Action のことを指し、この名前がつけられたアクションは、SSR の初期化時に自動で実行されます。

ミドルウェアと違い、ただひとつの関数しか実行できないため、複数の機能を持たせるには向きませんが、例えば「マスタデータの取得」といった、必須処理の際に使用しておくと非常に便利でしょう。

認証の場合は、

- 利用に会員登録が必須なアプリケーションでは nuxtServerInit
- メディアサイトのような、非会員での閲覧と会員での閲覧が混在するものでは、ミドルウェア

に、それぞれ書くのが良いでしょう。

余談ですが、nuxtServerInit は、ミドルウェアの実行より先に行われるため、例えばマスターデータの取得を nuxtServerInit に、認証をミドルウェアに実装した場合、認証のためにマスターデータを引き回すことも可能です。

下記に、実際の実装例をご紹介します。モジュールモードにて実装されている場合の、`store/index.js` の例となります。

index.js

```
import Vuex from 'vuex'
import Cookies from 'universal-cookie'

export default () => new Vuex.Store({
  actions: {
    nuxtServerInit({ commit }, { req, route, redirect }) {
      if (!process.server || ['/login'].includes(route.path)) {
        return
      }

      const cookies = new Cookies(req.headers.cookie)
      const credential = cookies.get('credential')

      if (credential) {
        // Main logic here...
        // e.g. commit('')
      } else {
        return redirect('/login')
      }
    }
  }
})
```

こちらも実際に実装したサンプルデモ及びソースコードを公開しています。適宜ご利用ください。

- Demo: https://potato4d.github.io/nuxt-tech-book/examples/section05/05_Auth_with_Vuex
- GitHub: https://github.com/potato4d/nuxt-tech-book/tree/master/examples/section05/05_Auth_with_Vuex

サーバーサイド JavaScript フレームワークとの連携

サーバーサイド JavaScript フレームワークとの連携についてもご紹介しておきます。

通常は Nuxt は主に View に関する責務を持ち、別途 API サーバーなどを用意しての開発となるかと思いますが、機能が数えられるほどしかない場合に、複数のサーバーを構築するのはコストが高いと言えるでしょう。

そういった場合は、Nuxt が SSR サーバーを立てることを利用し、前段に Express を噛ませ、Express 側で小さな API サーバーを構築すると非常に便利です。

Express との連携例は、`nuxt-community/express-template` などの公式テンプレートがありますが、ここでも最低限の例を紹介しておきます。

Express の導入とサーバーの起動

Express の Nuxt での導入方法を少しご紹介しておきます。Express 本体のインストールは、通常のサーバーサイドの開発と同じように、Nuxt プロジェクトに yarn add で追加します。

terminal

```
$ yarn add express body-parser
```

その上で、専用のサーバー用のディレクトリを作成し、ファイルを配置します。わかりやすくするために、`server/index.js` などに配置しておくと良いでしょう。

index.js

```
const express = require('express')
const { Nuxt, Builder } = require('nuxt')
const app = express()
const host = process.env.HOST || '127.0.0.1'
const port = process.env.PORT || 3000
const bodyParser = require('body-parser')

app.set('port', port)
app.use('/api/', bodyParser.json())

const config = require('../nuxt.config.js')
config.dev = !(NODE_ENV === 'production')

const nuxt = new Nuxt(config)

if (config.dev) {
  const builder = new Builder(nuxt)
  builder.build()
}

app.use(nuxt.render)
```

```
app.listen(port, host)

console.log(`Server listening on ${host}:${port}`)
```

最後に、npm scripts も変更の必要があります。

package.json を以下のように設定します。

package.json

```
{
//...
  "scripts": {
    "dev": "NODE_ENV=development node ./server/index.js",
    "start": "NODE_ENV=production node ./server/index.js"
  }
//...
}
```

このように記述しておくと、`http://localhost:3000` では Nuxt が、`http://localhost:3000/api/` では Express が動作する一体型サーバーが動作します。

あとは通常の Express アプリケーションと同様に開発が可能です。データの取得のサンプルのために、試しに以下を実装してみましょう。

index.js

```
// ...
app.use('/api/', bodyParser.json())
```

```
+++app.get('/api/todos', (req, res) => {
+++  res.json({
+++    todos: [
+++      { id: 1, name: 'First Todo' },
+++      { id: 2, name: 'Second Task' },
+++      { id: 3, name: 'Third Task' }
+++    ]
+++  })
+++})

const config = require('../nuxt.config.js')
// ...
```

このように書いて実行すると、`/api/todos` では Express が、`/` では Nuxt が呼び出されていることが確認できます。後は通常の Express アプリケーションのように開発するだけで OK です。

データの取得

Expressにて開発を行った上で、実際の Express サーバーとの連携は、axios-module を利用すると良いでしょう。3 章にて導入方法はご紹介しておりますので、導入までは省き、利用例を掲載しておきます。

axios-module を利用すると、withCredential オプションによる Nuxt / Express 間の Cookie の共有が可能であったり、デフォルトの baseURL に `/api` が設定されていたりと、単一のサーバー内での Nuxt + Express 連携においても非常に快適な HTTP リクエストが可能となります。

こちらも、実際に実装したサンプルデモ及びソースコードが以下にありますので、適宜ご利用ください。

- Demo: https://potato4d.github.io/nuxt-tech-book/examples/section05/05_Express_Intergration
- GitHub: https://github.com/potato4d/nuxt-tech-book/tree/master/examples/section05/05_Express_Intergration

実際の活用シーンについて

実際の活用においては、冒頭に書いたように小さな API サーバー限定で利用するのが良いでしょう。例えば、SSR で Web サイトを運用する場合に、問い合わせなどのために小さな API が必要になる場合などに有効に働いてくれます。

Nuxt はユニバーサルな技術であるがゆえに、Nuxt の管理下にあるコードは必ずクライアントに露出してしまいます。そのため、いくらサーバー側で動くからといってサーバー上でのみ行う処理を middleware などで実装するのは良い手とは言えません。

「サーバーサイドのフレームワークが必要なほどではないのではないか?」と思った場合でも、必ず Nuxt ではなくサーバーサイドのフレームワークと連携して実装しましょう。

筆者は実際に以下の Web サイトで Nuxt + Express による運用を行っております。

- https://push7.jp
- https://corp.scouter.co.jp

また、実運用についての情報については、Nuxt の範囲から逸脱するため今回は取り扱いませんが、筆者が情報をまとめて公開しているブログエントリがありますので、ご興味のあるかたは、こちらを参照してください。

- http://techblog.scouter.co.jp/entry/2018/03/19/115229

SEO やソーシャルにも役立つ適切な HTML メタの設定

前二項ではアプリケーションロジックについてご紹介しましたが、次はどのような形での開発でも便利な HTML メタ(以下: メタタグ)の設定についてご紹介いたします。※本来は HTML Attrs のメタ情報と Head におけるメタタグは別ですが、ここではわかりやすさを重視し、全て「メタタグ」として扱います。

Nuxt は SSR によって実際のコンテンツはクローラに正しく認識されるようになっていますが、その上で基本的なメタタグなどの設定も、勿論行う必要があります。

デフォルトではメタタグが設定されていないなど、body 以外は非常に弱い作りとなっています。

ここではそんな HTML メタの強化についてご紹介いたします。

Nuxt のメタタグの管理システム

実際のサンプルを追う前に、Nuxt のメタ情報のシステムについてご紹介しておきます。

Nuxt は、メタ情報の管理のライブラリとして vue-meta を利用しており、記法は全て vue-meta に則ったものとなります。そのため、Nuxt 自体のドキュメントでは vue-meta を利用している旨だけが書かれており非常に簡素な説明だけが書かれています。

少し不親切に思うかもしませんが、vue-meta 側のレポジトリにアクセスすると欲しい情報は一通り載っていますので、適宜そちらを参照するようにしておきましょう。

なお、vue-meta のレポジトリは以下です。

https://github.com/declandewet/vue-meta

適切な lang 属性の付与

それでは実際のコードサンプルにうつっています。まずは lang 属性から。

`<html>` タグには、lang という属性があり、これを使うことでマークアップ内の言語を明示することが可能です。lang の中身は、BCP 47 と呼ばれる規格のもと決まっていますが、ここでは `ja` や `en` 形式の指定と覚えておくだけで十分でしょう。

この lang タグですが、デフォルトで設定されていては問題ですので、Nuxt はデフォルトでは空となっています。しかしながら、サイト内の言語が決まっているのであれば、設定しておくべきでしょう。

こういった、グローバルに設定したいメタタグについては、nuxt.config.js に記述すると非常に便利です。nuxt.config.js に、以下のような設定を追加することで、`<html>` タグに属性を付与することができます。

nuxt.config.js

```
module.exports = {
   // ...
+++  head: {
+++    title: 'Nuxt Web Application',
+++    htmlAttrs: {
+++      lang: 'ja',
+++    }
+++  }
  // ...
}
```

このように、head キーで囲った中身が Web サイト全体のメタタグとして作用します。ここでグローバルで統一したいタイトルや description は設定しておくと良いでしょう。

なお、この設定は一番弱い設定となりますので、更に狭いスコープにおける設定があればオーバーライドされることとなります。詳しくは、次に説明します。

ページごとのタイトルとテンプレートの設定

グローバルの設定があるならローカルの設定もあります。vue-meta は、Vue コンポーネント単位でのメタタグへの干渉が可能ですので、例えばレイアウトファイルやページファイルでその効果を存分に発揮してくれることでしょう。

例えば、前述の lang 属性を付与する時に、同時にタイトルとして `Nuxt Web Application` を設定しました。トップページなどはこれで良いでしょうが、例えば `/users/:id` 形式のユーザーページにアクセスした際など、下層ページでは `Potato4d(@potato4d) | Nuxt Web Application` といった形で、`|` 区切りでタイトルを追加したい場合も多く存在するでしょう。

こういった場合は、グローバルでは `Nuxt Web Application` としておき、`layouts/default.vue` および `pages/*.vue` に、以下のような設定を行うと良いでしょう。

まずはレイアウトを以下のように記述します。

default.vue

```
<template>
  <nuxt />
</template>

<script>
export default {
  head () {
    return {
      titleTemplate: '%s | Nuxt Web Application'
    }
  }
}
```

その上で、ページを以下のように記述します。

user.vue

```
<template>
  <div />
</template>

<script>
export default {
  head () {
    return {
      title: 'ユーザーページ'
    }
  }
}
```

ポイントは `head()` および `titleTemplate` にあります。Nuxt には、Vue コンポーネントに拡張された head functions を自動で実行するシステムがあります。

上記のように記述した上で、戻り値にオブジェクトを渡してやると、nuxt.config.js で head を指定したときと同じ形式で、メタタグを上書きすることが可能となっています。また、その際親から子に伝わるに連れて優先度はあがっています。

上記をまとめると、`pages/user.vue` > `layouts/default.vue` > `nuxt.config.js` という優先度となり、`user.vue` のタイトル、`default.vue` のタイトルテンプレート、そして `nuxt.config.js` の htmlAttrs が最終的にメタタグとして出力されることとなります。

そして、このタイトルテンプレートについてです。タイトルテンプレートは、`titleTemplate` キーが指定されたデータのことであり、これを利用すると、`title` キーに設定されたテキストを含んだ、テンプレート化されたタイトルを出力することが可能です。

このテンプレートのフォーマットには、他の言語でよく見られる `sprintf` と同じ形式が採用されており、`%s` を置き換える形で

今回であれば `%s | Nuxt Web Application` というテンプレートに、**ユーザーページ** という文字列が置き換えれたことによって、**ユーザーページ** | `Nuxt Web Application` というタイトルが出力されます。

開発中はあまり意識することがなく、リリースが近づくにつれて気づくことが多いメタタグですが、タイトルはテンプレート化できること、`head()` をつけるだけでページ単位で上書きが可能であること、最後に、フォーマットは vue-meta を尊重することを覚えておくと、すぐに対応できることでしょう。

困ったら Nuxt のドキュメントだけではなく、vue-meta のドキュメントもあわせて読みながら、適切に設定を進めていきましょう。

アプリケーションのデプロイ

最後に、Nuxt アプリケーションのデプロイについてご紹介します。Nuxt は、その柔軟性から様々な方法でデプロイが可能です。

実際のメンテナンスやコストパフォーマンスを考えて、最適な運用方法を模索するための一つの目安をご紹介します。

ホスティング先サーバーの選定

まずはホスティング先の選定についてご紹介いたします。

静的サイトか SSR サーバーか?

まず一つに、静的サイトとして運用するか、SSR サーバーを建てるかという選択があります。Nuxt は、Vue 単体のアプリケーションにはない generate 機能が実装されています。

generate を利用すると、generate の段階で Nuxt の asyncData などが実行された上で、その SSR 結果を HTML ファイルとして出力します。

これにより、静的サイトホスティングサービスへのデプロイなどが可能となり、運用コストを時間面でも金銭面でも抑えることができるため、積極的に活用するべきでしょう。

本当の意味でのリアルタイムである必要がなく、ただ動的なコンテンツを更新管理したいという場合は、SSR にとらわれることなく、generate から検討することを強くオススメします。

特に、Web アプリケーションではなく、Web サイトを構築したい場合は、優先して考えるべきでしょう。

デプロイ先について

運用方法によりますが、今回はざっくりと「静的サイトホスティング」と「SSR サーバー運用」で分けてみましょう。

勿論、最終的にはただの静的 Web サイトもしくは Node.js サーバーとして動作させることが可能であるため、Amazon EC2 などの IaaS サーバーでの運用も可能ですが、今回は省きます。

静的サイトホスティング

静的サイトの場合は、個人やコミュニティ利用であれば GitHub Pages や Netlify 、商用では Amazon S3 + CloudFront などでの運用がオススメです。

GitHub Pages, Netlify ともに、Vue コミュニティでの運用実績があり、特に Netlify は、Vue.js 公式ドキュメントが GitHub Pages から移行するなど、大きな盛り上がりをみせています。

一方で、Nuxt のドキュメントは現在も GitHub Pages でホスティングされているため、非商用や個人利用であればどちらでもよろしいでしょう。筆者としては、将来性を考えて Netlify の利用をオススメしておきます。

商用の場合は可用性や柔軟性、ロギングなどを考えると Amazon S3 + CloudFront で運用するのがベストでしょう。

SSR サーバー運用

SSR サーバーを運用する場合、試しに Heroku 利用からはじめてみると良いでしょう。Heroku を利用すると、無料かつ最小限の設定で Nuxt サーバーを構築することができます。

大きな欠点として 現状 US/UK リージョンしかありませんので、レイテンシなどを考えると本格的なアプリケーションのホスティングには向きませんが、Hobby Use やステージング環境の構築においてはそのポテンシャルを遺憾なく発揮してくれることでしょう。

Heroku については、Nuxt 公式ドキュメント上でも言及されており、Nuxt サーバーの起動にあたって必要な設定がいくつか書かれています。詳しくは以下の URL からアクセスできるドキュメントをご参照ください。

> How to deploy on Heroku?
> https://nuxtjs.org/faq/heroku-deployment

また、プロダクションにおける実運用では、AWS ECS などの Docker ベースのコンテナエンジンを利用することとなるでしょう。Nuxt は、内部としてはオーソドックスな Node.js サーバーとして作られていますので、通常の Node.js v8 以降のイメージで動作します。

6. Nuxt のエコシステム

ここまでの内容で、Nuxt のシステムは一通り理解した上で、実際の開発を十分に行うことができる状態となりました。

勿論、このまま開発を進めることもできますが、Nuxt での開発を行うのであれば、公式やその周辺の Nuxt のためのエコシステムを十分に活用するべきでしょう。

エコシステムをうまく活用することで、公式コミュニティによってメンテナンスされている Nuxt にとって最適であり高品質なコードベースを、簡単に活用することができます。

このセクションでは、そんなエコシステムのライブラリの中から、公式プラグインやその他サードパーティ製の拡張まで、特に有用なものをいくつかご紹介します。

Nuxt の公式コミュニティプラグインの活用について

まずは Nuxt の公式コミュニティによって提供されているプラグインのご紹介します。

Nuxt のコードは、本体からドキュメントまで、すべて GitHub にて管理されていますが、公式のコミュニティプラグインも同様に管理されています。

公式コミュニティの成果物は、専用の Organization である https://github.com/nuxt-community にて管理されています。

この Organization では、プラグインだけではなく、ボイラープレートや CLI ツールも公開されているため、試しに覗いてみると良いでしょう。

また、公式コミュニティの成果物は、必ず nuxtjs のもと、Scoped Package として公開されています。

サードパーティ製かどうかを簡単に判断する一つの指標となっているため、適宜活用すると良いでしょう。

このセクションでは、そんな公式プラグインから、特にオススメなプラグインについて二つご紹介します。

axios-module からの proxy-module の呼び出し

ここまでで何度も登場してきましたが、Nuxt でのアプリケーション開発において、axios-module は必須と言えるでしょう。axios 単体ではカバーしきれていない、かゆいところに手が届くものであり、Vuex との連携もシームレスです。

その上で、本番環境で API がサブディレクトリ構造になっている場合などは、追加で proxy-module を使うのも良いでしょう。

proxy-module は、その名の通りプロキシ機能を提供するプラグインであり、「本番環境では nginx を利用して同一ドメイン内に同居しているけれど、開発環境は別々のポートでサーバーが立っている。」といった場合の差異を吸収してくれるものとなります。

マッピングを行うだけで、/api を、http://localhost:8000/api にプロキシしてくれると言った具合です。

しかしながら、この proxy-module は現在 axios-module が依存対象としており、axios-module を入れる場合は、axios-module にて proxy-module 設定を書くことができるようになっています。

ですが、今回は axios-module を使わない場合の開発も想定し、試しに proxy-module を別途入れて設定しています。

以下に、導入方法と利用方法をご紹介します。今回は、二つのプラグインを利用して、Qiita APIを proxy して叩いてみます。

axios / proxy の導入

Yarn で導入しましょう。

二つのパッケージ名は `@nuxtjs/axios` と `@nuxtjs/proxy` となっています。

```
$ yarn add @nuxtjs/axios @nuxtjs/proxy
```

axios 設定の記述

まずは axios 書きましょう。なお、今回はプロキシ設定のための準備ですので、axios-module 以外の HTTP ライブラリでも問題ありません。

nuxt.config.js を記述します。

nuxt.config.js

```
+++ modules: [
+++   '@nuxtjs/axios',
+++ ],
+++ axios: {
+++   prefix: '/api/v2'
+++ }
```

axios-module では、prefix キーに設定したものが、baseURL の接頭辞となります。例えばこの場合、`http://localhost:3000/api/v2` を叩くようになっています。

これで API 用の URL を指定できましたが、ただの Nuxt サーバーに API は生えていませんので、どこかにプロキシする必要があります。

プロキシ設定の記述

そういった時に、proxy-module の設定を追加すると、非常に簡単にプロキシが設定できます。

nuxt.config.js に、さらに設定を追記してください。

nuxt.config.js

```
  modules: [
    '@nuxtjs/axios',
+++   '@nuxtjs/proxy',
  ],
  axios: {
    prefix: '/api/v2'
```

```
  },
+++ proxy: {
+++   '/api/v2': 'http://qiita.com'
+++ }
```

このように設定すると、`/api/v2` にアクセスしたときに、パス以前を置き換えることができます。この場合、`http://localhost:3000` の部分が `https://qiita.com` に置き換わるため、`https://qiita.com/api/v2` が baseURL となります。

ですので、この状態で `this.$axios.$get('/users')` などを叩くと、`https://qiita.com/api/v2/users` が叩かれることとなります。試してみたいかたは、pages/index.vue に以下を記述してみると良いでしょう。

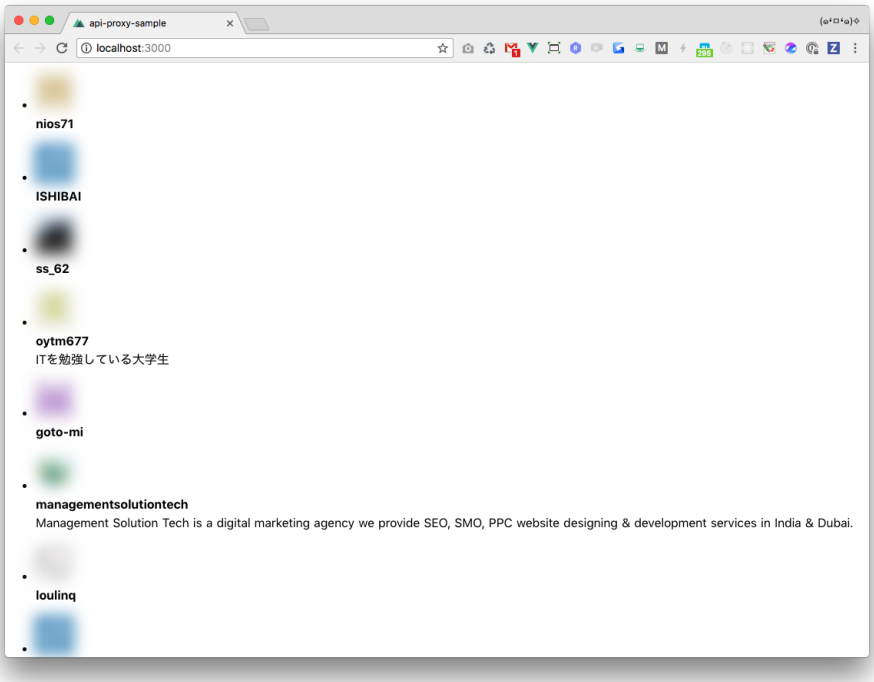
index.vue

```
<template>
  <section class="container">
    <ul>
      <li v-for="user in users" :key="user.id">
        <p>
          <img :src="user.profile_image_url" alt="" ><br>
          <strong>{{user.id}}</strong><br>
          {{user.description}}
        </p>
      </li>
    </ul>
  </section>
</template>

<script>
import AppLogo from '~/components/AppLogo.vue'
```

```
export default {
  async asyncData({ app }) {
    const users = await app.$axios.$get('/users')
    return {
      users
    }
  }
}
</script>

<style>
p {
  margin: 0 0 16px;
  line-height: 1.5;
}
</style>
```

実際に動作させると、以下のように表示されます。

※ サンプルではアイコンをぼかしています



こちらの動作デモ及びレポジトリは以下となります。適宜ご参照ください。

- Demo: https://potato4d.github.io/nuxt-tech-book/examples/section06/06_API_And_Proxy
- GitHub: https://github.com/potato4d/nuxt-tech-book/tree/master/examples/section06/06_API_And_Proxy

pwa-module によるオフライン対応

pwa-module を利用すると、イマドキな PWA の対応を、簡単な設定だけで行うことが可能です。

Service Worker ベースの技術である、オフライン対応や、OneSignal を利用しての Web プッシュ通知まで、PWA 開発に必要なものが nuxt.config.js への記述だけで解決する、非常に優秀なプラグインとなっています。

今回は、基本的な Server Worker のインストールと、デフォルトでついているオフライン対応を行ってみましょう。

pwa-module の導入

パッケージ名は `@nuxtjs/pwa` となります。

```
$ yarn add @nuxtjs/pwa
```

config への PWA オプションの追加

PWA への最低限の設定追加は、module の追加だけで可能です。あなたのプロジェクトの nuxt.config.js に以下を追加しましょう。

nuxt.config.js

```
+++  modules: [
+++    '@nuxtjs/pwa'
+++  ]
```

専用のファイルの gitignore の追加

次に、gitignore の追加を行います。PWA モジュールは、Service Worker のためのコードを追加で吐き出すため、ignore しないと誤ってビルド後のファイルが Git レポジトリに入ってしまうこととなります。

.gitignore に、`sw.*` を追加しておきましょう。

generate しての動作確認

導入は全てできたので、静的サイトを generate をして動作確認してみましょう。PWA モジュールは、開発環境での事故を避けるため、Service Worker のインストールを本番ビルドに限定して行っています。

スクリーンショット左のように、開発中はオンラインでしか動作しませんが、本番ビルドを行うと、スクリーンショット右のようにオフライン対応が可能となります。

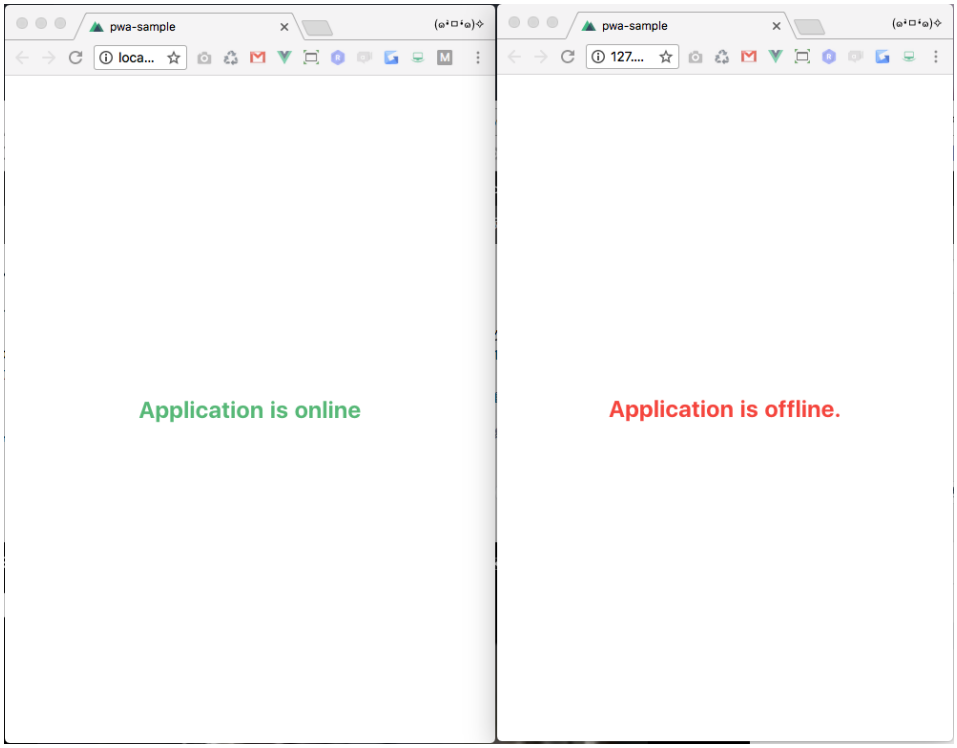
簡単に試したい場合は、以下のように行うと良いでしょう。

terminal

```
$ yarn generate
$ cd dist
$ php -S 0.0.0.0:8080
$ # or python -m SimpleHTTPServer
```

を実行した上で、立ち上がるサーバーにアクセス。

Chrome Devtools など、お使いのブラウザの開発者ツールでオフラインにしてみると、オフラインでアクセスできていることがわかるはずです。



最後に、実際のこの表示切り替えまでも含めたサンプルの URL を掲載します。スクリーンショットのように、オンラインかオフラインかで表示を切り替えるコードも追加しておりますので、是非ご活用ください。

- Demo: https://potato4d.github.io/nuxt-tech-book/examples/section06/06_PWA
- GitHub: https://github.com/potato4d/nuxt-tech-book/tree/master/examples/section06/06_PWA

7. Nuxt の情報収集・キャッチアップのススメ

最後に、Nuxt の最新情報のキャッチアップのヒントについてご紹介します。

正式リリース以後、破壊的変更も落ち着き、徐々に日本語資料が増えたことでハードルが下がったとはいえ、まだまだ Nuxt の資料はノウハウは少ない現状です。

最後となるこのセクションでは、そういった中、目的の情報を公開するにはどうしたら良いか。また、自ら Nuxt の課題を解決したい、あるいはした際に、どのようにコミュニティに還元していくのがベストかをご紹介します。

英語ドキュメントを効率的に読むコツ

Nuxt での開発を行う場合、英語ドキュメントや資料を読むことは避けられません。

日本語の資料も少しならあるものの、ディープな内容は不足しているため、実質英語資料が必須となっています。

ここでは、少しだけ英語資料の読むときの手引きをご紹介します。

公式ドキュメントの利用

まずは何より公式ドキュメントからです。

Nuxt は、Vue.js 同様公式ドキュメントが充実しており、基本的な使いかたから少し凝った利用でのサンプル、果てはいわゆる「FAQ」ガイドまであり、簡単なアプリケーションを作るために必要な情報は全て揃っています。

ローカライズ版の利用も視野に

また、日本語版は現在少し情報が古いものの、各言語にローカライズが行われていますので、もし英語が苦手な場合はローカライズ版を閲覧してもよろしいでしょう。

原文である英語版がほぼ全て翻訳されており、豊富な情報を母国語でスムーズに読むことが可能です。

注意点としては、日本語情報は常に一定以上遅れて更新されている都合上、互換性のない変更が追従されていないことで困る機会も少なくはないので、英語に抵抗がないかたは極力原文を読むことをおすすめいたします。

公式ドキュメントの検索の活用

Nuxt のドキュメントは、非常に情報が充実していると述べましたが、充実しすぎているが故に、はじめての人は目的の状況に辿り着きづらいかもしれません。

そういったときは、ぜひサイト内の検索バーを有効活用しましょう。インクリメンタルサーチ付きの検索バーが存在しており、「Vuex」や「Express」など、調べたいワードを入力すると、即時で検索が走り、目的の情報にすぐにたどり着くことができます。

エッジケース発生時のレポジトリの読みかた

いくらドキュメントが充実しているといっても、Nuxt 自体は非常に多くの独自機能を有しており、全てが網羅されているわけではありません。

中には説明不足であったり、明記されていないためどうすると良いかわからないという場合も出てくるでしょう。

そういったときは、Nuxt の公式のレポジトリのソースコードや Issue を眺めてみることをオススメします。

過去に同じパターンではハマった人がいる場合はそれ自体が一つの資料となりますし、ソースコードを読むことができるとわからないことはほぼなくなるでしょう。

以下に、Issue ベースでの問題の解決フローをみてみます。

問題の抽象化・置き換え

試しに国内ではよくありそうですが資料が見つからないであろう事例として「サブディレクトリでの Nuxt の generate 運用」を考えてみます。

国内では SEO 対策のためという名目でオウンドメディアがサブディレクトリ運用(/blog/ など)されることが多いですが、ドメインパワーなどの専用の文化や風習がないところでは、そもそもサブドメインでわけるほうがスマートですし、数としてはサブディレクトリ形式は減少していそうです。

それに、これを「サブディレクトリ形式」と呼ぶのが一般的かどうかもわかりづらいですし、こうなってくるとなかなか調べるのが困難に見えます。

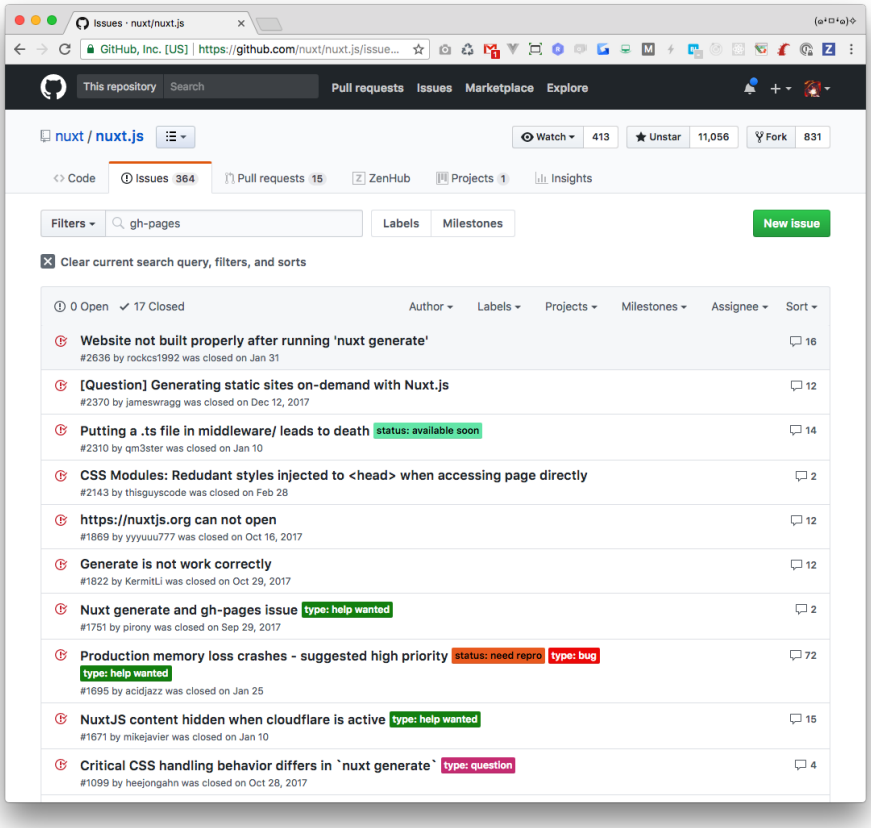
しかしながら、よく問題を考えると、単純に「どこかのドメイン配下をルートとした場合の問題」であり、もっと言うとこれは「GitHub Pages」スタイルでの運用の問題と言い換えることもできます。

「GitHub Pages」という具体的なサービス名で検索をすると、独自の呼びかたではないですし、ユーザーも多く存在しますし、問題が見つかりそうです。

このように、問題を抽象化した上で、逆に具体的な「サービスやアプリケーション」に落とし込むことで、Issue で見つかりそうなワードを探すことができました。

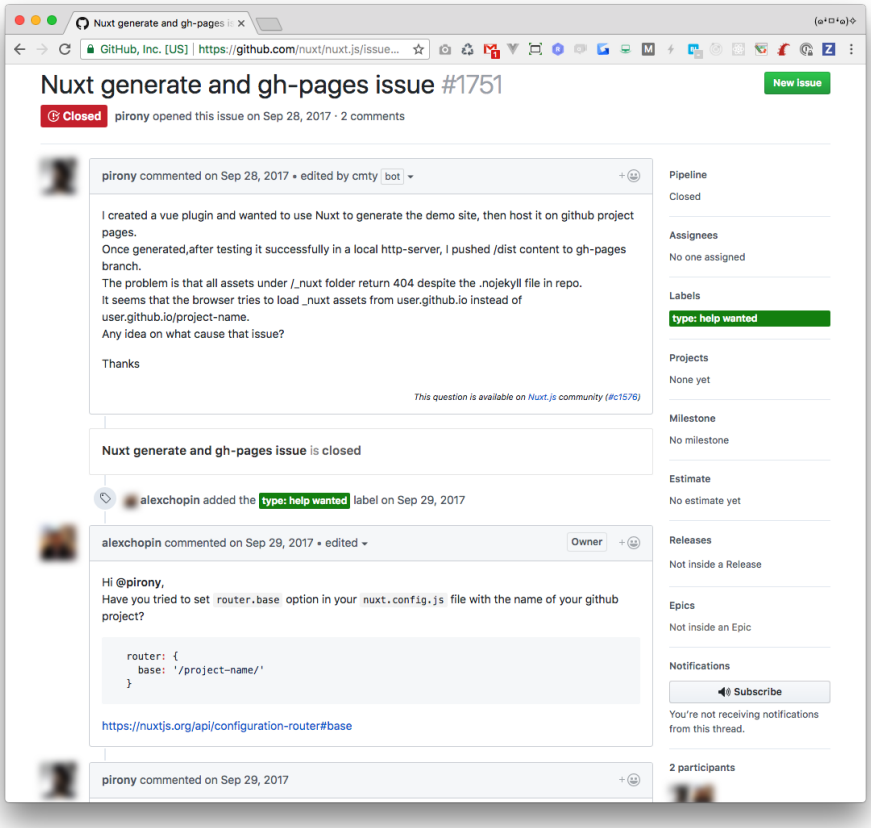
検索と解決方法の適用

ここまでできたなら、GitHub Pages へのデプロイで問題が起こっている例を調べると良さそうです。試しに Nuxt の Issues で、`gh-pages` で調べてみます。



検索すると、多くの Issue がでてきました。その中でも、ラベルが貼られている「Nuxt generate and gh-pages issue」辺りがジャストに見えます。

試しに見てみましょう。



どうやら丁度サンプルが書かれているようで、router.base を設定すると良いようです。実際に、この router.base は、generate をした時のルートディレクトリを設定できるものとなっていますが、ドキュメントの奥深くにあり、また、簡単にはイメージしづらい説明となっています。

https://nuxtjs.org/api/configuration-router#base

しかし、こういった場合でも、レポジトリとサンプルコードを探すことによって、高速に問題を解決することができます。

Nuxt まわりで詰まったことがあれば、適宜 GitHub Issues も利用して調べながら解決をすすめ、もし未解決や、解決しているがドキュメントがないようであれば、ソースコードやドキュメントに貢献していくと良いでしょう。

特に、ドキュメントは簡単に貢献していくことができますし、翻訳者も随時募集しているため、興味があればこの次のセクションものぞいてみてください。

ドキュメント日本語翻訳プロジェクトのご紹介

最後に、私も携わっている Nuxt 公式ドキュメントの日本語翻訳プロジェクトについてご紹介します。

Nuxt の公式ドキュメントは、各国の Nuxt ユーザーにより、活発にそれぞれの言語へのローカライズが行われています。

日本も例外ではなく、まだ Nuxt の正式バージョンが公開される前から、翻訳が行われており、随時原文の英語ドキュメントに追従できるように、有志によって翻訳の更新が行われています。

しかしながら、正式リリース前後で大きく仕様が変わったこと、正式リリース以降も精力的に開発が進んでいることもあり、まだまだ古い箇所が多くあります。

もし、ここまでの方法で情報をキャッチアップしていく中で、日本語の情報が古くて苦労した場合などは、ぜひ後述の翻訳への貢献方法を参考にして、実際に手を動かしてみてください。

少しだけの変更でも、多くの人が行うことで大きな成果につながるもの。一行の変更でも歓迎していますので、ぜひ翻訳に貢献してみてください。

翻訳プロジェクトへの参加方法

まず、翻訳に関する議論が行われているコミュニティに参加しておきましょう。

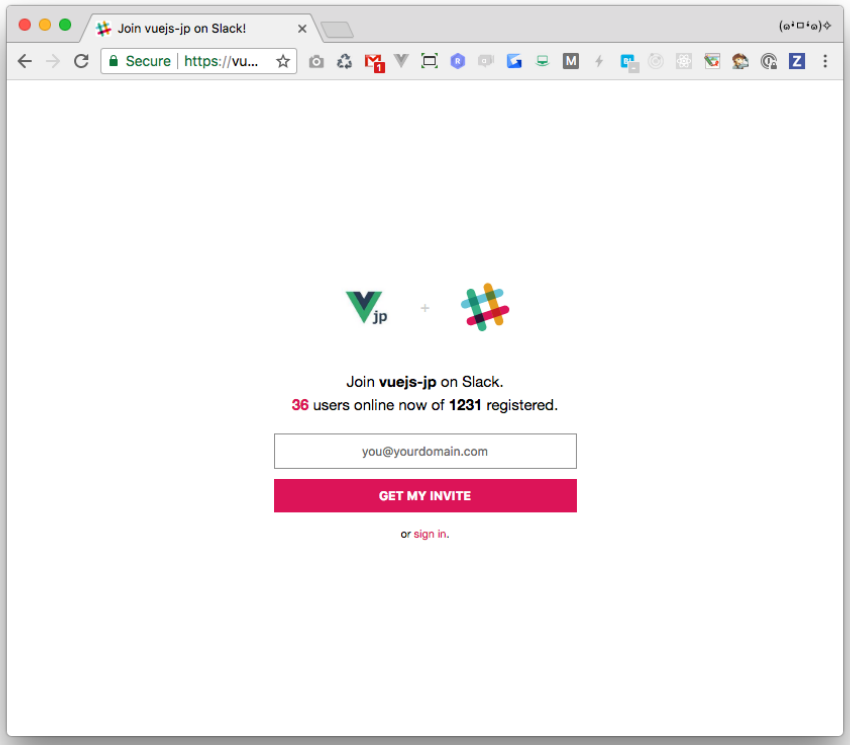
Nuxt ドキュメントの翻訳は、Vue.js 日本ユーザーグループの Slack の #translation チャンネルと、翻訳サービスである GitLocalize を利用しての行われています。

ここでは、実際に Slack への参加方法と、GitLocalize での翻訳方法についてご紹介します。

Vue.js JP Slack への参加方法

Vue.js JP Slack へは、どなたでも承認なしに参加することが可能です。特に翻訳を行うかたは、終わった後にレビュアーへとメンションするためにも、入っておくと良いでしょう。

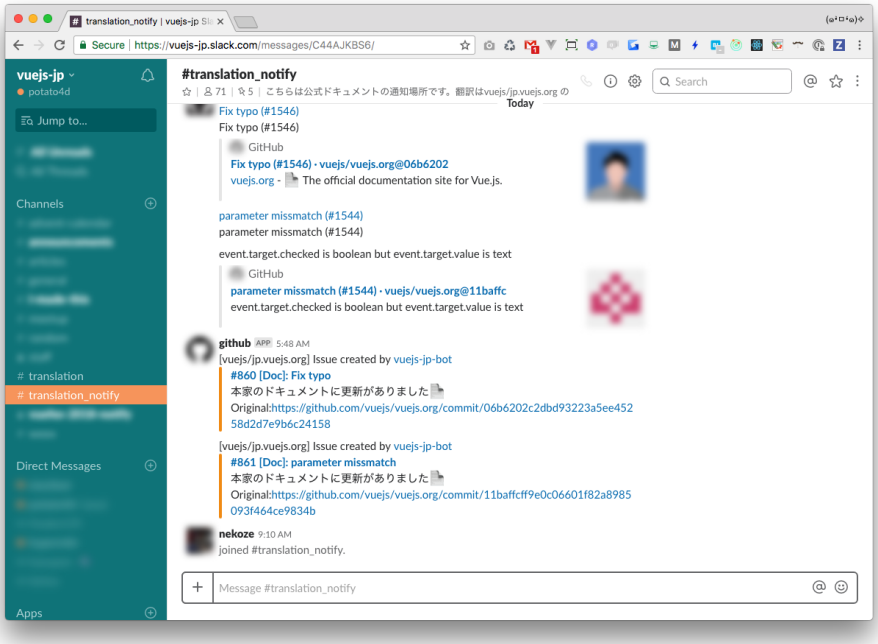
以下の URL より招待用のアプリケーションにアクセスし、メールアドレスを入力すると招待が届きますので、あとは通常通り参加してください。



https://vuejs-jp-slackin.herokuapp.com

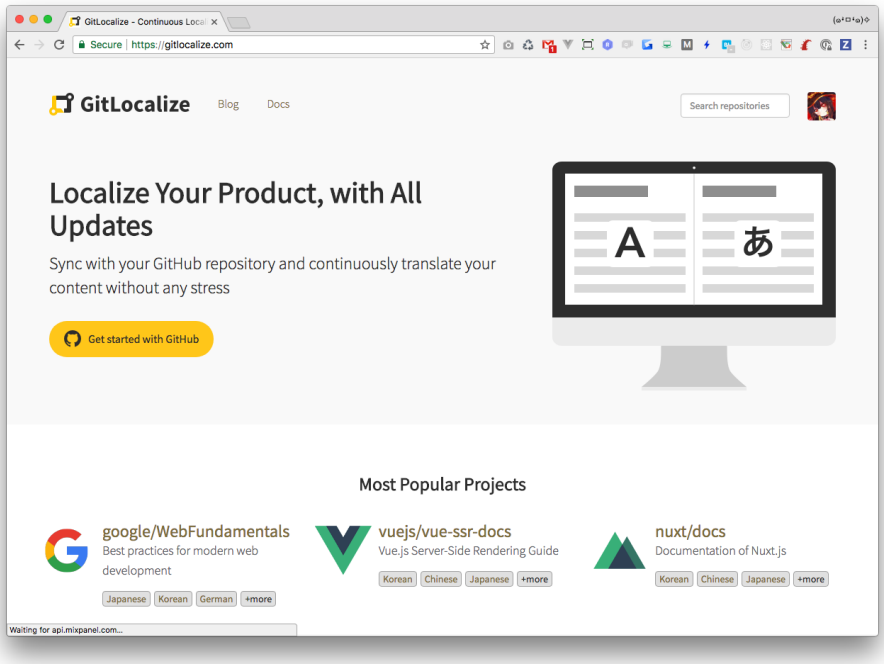
参加すると、デフォルトでは #general チャンネルに参加していますが、追加で #transition #transition_notify チャンネルにも入りましょう。

ここでは、Nuxt のドキュメントだけでなく、Vue.js 日本語ドキュメントについての翻訳プロジェクトなども走っています。もし気になるかたは、それぞれのドキュメントの GitHub レポジトリと並行して様子を見ておくと良いでしょう。

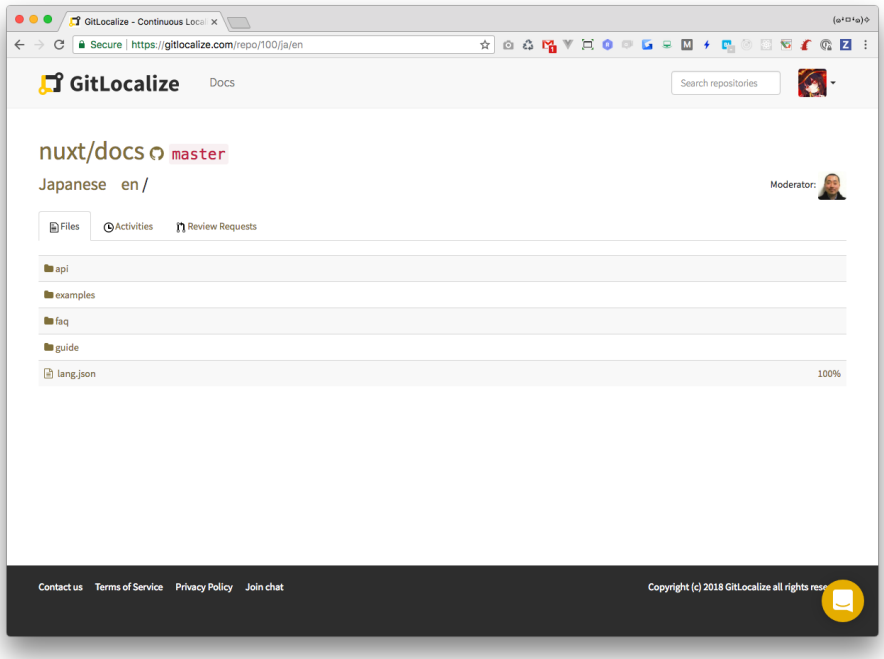


GitLocalize を利用した翻訳の手順

次の GitLocalize の登録です。https://gitlocalize.com にアクセスし、"Get started with GitHub" より、GitHub ログインを行います。

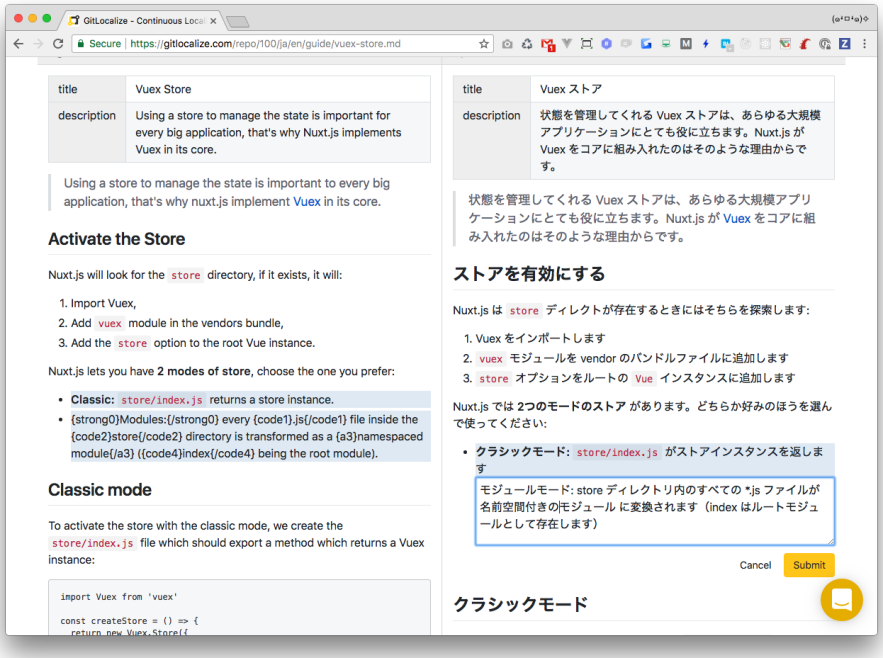


その後、https://gitlocalize.com/repo/100/ja/en にアクセスをすると、Nuxt のレポジトリの情報が出てきます。

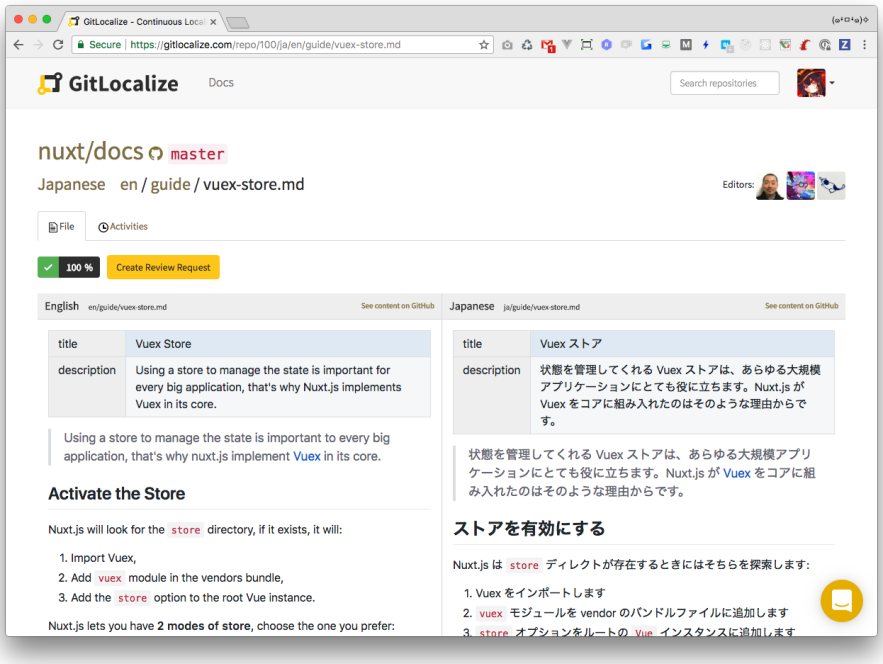


翻訳したい・翻訳を修正したいファイルにアクセスをします。今回は Vuex Store の部分を修正してみます。

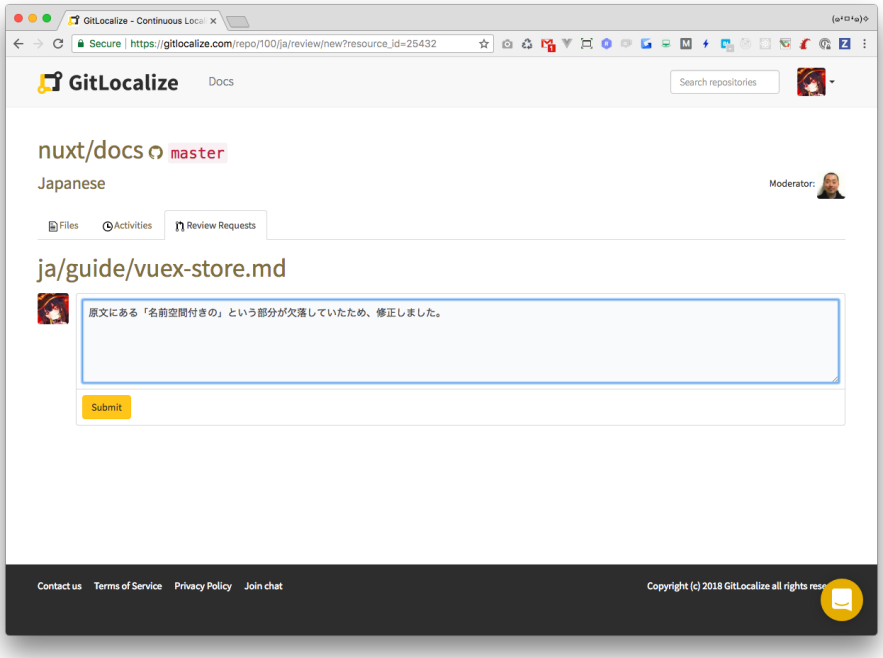
該当ファイルにアクセスすると、このように英和の対応表が出てきます。その上で、修正したい部分をクリックして書き換えると完了となります。



最後に、GitHub でいうところの Pull Request を送る必要があります。翻訳ページの最上部にある、"Create Review Request" をクリックして、説明文を書きましょう。



説明文を書いたら、Submit して完了となります。後はレビュアーフィードバックを待ちましょう。



Nuxt は進化が早いぶん、少しの気付きへの追従が多くの人に役立つ情報となりますので、もし翻訳について貢献できる箇所を見つけた場合、上記の手順に沿って積極的に翻訳してみてください。

GitLocalize 不具合時の GitHub からの Pull Request について

GitLocalize は新興のサービスであり、まれに翻訳がうまくできないことがあります。そういった場合は、現状の Nuxt ドキュメントの運用方針として、GitHub との相互運用を容認していますので、GitHub から直接 Pull Request を送信してください。

8. あとがき

私が Nuxt を現場に実践投入しはじめたのは、2017 年の夏頃でした。

その頃は丁度バージョンがアルファに乗った頃で、それから Stable リリースまでに追加したプロジェクトは、幾度の Breaking Changes に呑まれながらも更新を続けていました。現在では Stable 追従が完了し、これまでに携わった 10 を越える Nuxt プロジェクトが、全て安定稼働しています。

しかしながら、はじめの 2,3 のプロジェクトは、実用にあたって、「情報の不足」という大きな課題がありました。当時は技術の浸透が遅い日本においてでは情報がほとんど存在せず、英語資料に読み慣れている人でないと情報収集の収集は困難な状態でした。

確かに情報が少ない中で運用することはリスクであるため、流行らないことは納得できます。ですが、それと同時に素晴らしい技術が資料の数を理由に流行らないのは、大きな機会損失であると感じ、情報の発信をはじめました。はじめはコミュニティイベントによる登壇。そこから Qiita や HTML5Experts.jp といった、複数のメディアで情報発信を行い、寄与したかは定かではありませんが、結果的に基礎的な情報についてはネット上に十分広まる結果となりました。

それでもまだ、Nuxt をディープに使っている事例やナレッジは多くありません。本書の筆をとることを決めた段階で、丁度 Stable がリリースされたということもあり、「もっと Nuxt を使い倒していきたい人向けに、みちしるべとなるような情報があれば良いんじゃないか。」ということで、実践に重きを置いた書籍を執筆すること決めました。

ただ使うだけでも、SSR を推すだけでもなく、良い部分や悪い部分、適材適所から、他の技術との組み合わせまで。どういったレイヤーで、また、どのような形で Nuxt を使い、どういった技術と組み合わせて成果物としての形を成すのか。そういったことを伝えることを目標として、ここまで書き進めてきました。

本書が私の思う書籍コンセプトを十分に満たすことができる品質に仕上がったかどうかは、これを読んだ読者のみなさまによって、どれほど made by Nuxt な製品が生まれるかで実感できるかと思うと楽しみです。

是非、本書を全て読み終えたあなたが、Nuxt で素晴らしいアプリケーションを作っていただけることを楽しみにしています。

最後に、本書の品質向上に多大な貢献をいただいた いいんちょ(@e_ntyo)様、まさあき(@masaaki)様、柏 舜(@shun_kashiwa)様、なかひこくん(@takanakahiko)様(五十音順)。

ならびに、本書の原稿の印刷・入稿について助力を頂いた緑豆はるさめ(@spring_raining)様。

そして最後に、本書を多くの人に手を取っていただくために素晴らしい表紙絵を作成した抱いたおれきゅー(@orekyuu)様に心より感謝を申し上げます。

私のひとりでは、この経験や知識を書籍という形で公開するに至りませんでした。

そして本書を手に取っていただいた読者のみなさま。受け取る人がいるからこそ、発信する人はアウトプット続けていくことができますので、どうぞこれからも機会があればチェックいただければと思います。

Nuxt tech book

2018 年 4 月 22 日　技術書典4
発行者　　花谷拓磨 (@potato4d)
発行所　　帝都研究所 produced by ElevenBack
印刷会社　　（有）ねこのしっぽ