インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

Vue.js超入門

3.2対応

爰河 英憲 著

Webアプリの実装を通して
Vue3の基本を学ぶ!

impress R&D
An impress Group Company

GIJUTSU no IZUMI SERIES
POWERD by NEXTPUBLISHING
技術の泉 SERIES

インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

Vue.js超入門

3.2対応

爰河 英憲 著

Webアプリの実装を通して
Vue3の基本を学ぶ!

impress R&D

GIJUTSU no IZUMI SERIES
POWERD by NEXTPUBLISHING
技術の泉

An impress Group Company

技術の泉 SERIES

電子書籍閲覧に関するご注意

　本書では、プログラムリストに専用の等幅フォントを使用しています。ビューアによって以下の作業が必要になります。
・Kindle Paperwhiteの場合：フォント設定画面で「出版者のフォント」を選択
・kobo Androidアプリの場合：フォント画面で「オリジナル」を選択

目次

はじめに

私はVue.jsを学習するにあたり、まだまだその技術書が少なく、Vue.jsに限らず入門書と言いながらもわかりにくかったり、不要な情報に振り回されている本が多いと感じました。本書はそのようなことがないよう、極力Vueに関することに重点を置いていきたいと思います。

本書の目的は「**Webアプリの実装を通し、Vue3の基本的な動きを理解する**」ことです。

JavaScriptを取り巻く環境は、ここ数年で劇的に変わりました。ターゲットは自分のようなエンジニアでありながら、モダンなJavaScriptに置いて行かれた、けれど学びたい人向けです。対象読者は全くのWeb初心者ではなく、JavaScriptの経験があり、後述の「**前提**」の条件を満たすことのできる人を想定しています。

サンプルとして作成するWebアプリは最終的にWeb上に公開できるところまでを記述していますが、Web上に公開する必要がない、またはしたくない場合は、その操作を省略するだけで続けることができます。
本書は全体を通してひとつのアプリを作成するだけのものと捉えられるかもしれませんが、基本的なものは詰まっておりますので、次のアプリを作成する際の教科書的なものとすることができます。

プログラミング経験者にとって実際に動くものがあるというのは、土台を手に入れたようなものです。そこから応用や他の機能を試すことができます。入門書という一冊を通して、自分の土台作りができるものがあればいいなと思い執筆しました。

構文の説明だけでなくどのような場面で使うのかなど、できる限り実践に即して書いております。Vueの歴史やreactなどとの比較、最初に学ばなくてもよい機能やツールについては触れておりません。それにより、入門には最適なものになったと思っております。

自身が初学者だった気持ちを忘れず、なるべくわかりやすい説明を入れ、最後まで一緒に完走できるように尽くしました。

それでは早速始めていきましょう！

前提

- （必須）Node.js、パッケージマネージャー（npm/yarn）がインストールされていること。
- （任意）使用するエディターは何でもいいのですが、本書はMicrosoft社の「Visual Studio Code」（VSCode）を使用していきます。
  ※同じエディターを使用していただくことで説明がわかりやすくなる場合があります。
- （Web公開する場合）git（https://git-scm.com）がインストールされていること。
- （Web公開する場合）GitHub（https://github.com）がインストールされていること。
- （Web公開する場合）Vercel（https://vercel.com）のアカウントを持っていること。
  ※GitHubアカウントでログインしてもらえると、GitHubとVercelの連携がスムーズになります。

第1章　Vueの準備

　Vueでアプリを作るための準備をします。この章ではVueで新規プロジェクトを作り、実際に動くところまでの準備をします。

◆ここで学べること
・Vueプロジェクトの作成
・Vueプロジェクトの開始と終了

1.1　実行環境

　参考までに本書の実行環境を紹介します。本書のバージョン以上になるようにしてください。

```
# ターミナル or コマンドラインで実行
$ node -v
v16.13.0

$ npm -v
8.1.2

$ yarn -v
1.22.10
```

1.2　Viteでプロジェクトを作成する

　Vue3インストールは公式に提供されているVue CLIを利用するのが一般的でしたが、本書はVite（ヴィート）を使ってインストールしていきます。

　Viteは最近出てきたビルドツールですが、Vueを作ったEvan You氏が開発しており、Vue公式ページにもViteでのインストール方法が記載されています。今後はViteを使っての開発が一般的になることでしょう。今までのビルドツールより、かなり高速に実行することができます。

　ターミナルあるいはコマンドラインでプロジェクトを作成したい任意のフォルダーに移動し、以下のコマンドを実行します。プロジェクト名（Webアプリ名）は何でもいいですが、本書では「my-vite-todo」とします。

```
# npmの場合
$ npm init vite@latest my-vite-todo -- --template vue

# yarnの場合
$ yarn create vite my-vite-todo --template vue
```

　任意のフォルダーにプロジェクト名の「my-vite-todo」フォルダーが生成されていれば、成功です。

1.3　Vueアプリを確認する

　それでは、Vueの最初のアプリを動かしてみましょう。以下のコマンドを実行してください。

```
# npmの場合
$ cd my-vite-todo
$ npm install
$ npm run dev

# yarnの場合
$ cd my-vite-todo
$ yarn
$ yarn dev
```

　これで開発用のサーバーが起動できましたので、ブラウザーを起動し、以下のアドレスにアクセスしてください。

　http://localhost:3000/

　以下のような画面が表示されれば、Vue3アプリの最初の関門は突破できました！

図1.1:

Hello Vue 3 + Vite

Recommended IDE setup: VSCode + Volar

Vite Documentation | Vue 3 Documentation

count is: 0

Edit `components/HelloWorld.vue` to test hot module replacement.

1.4　Vueアプリを終了する

　開発用のサーバーを止めるには、ターミナルあるいはコマンドラインで「Ctrl+C」を押下します。次の章から開発用サーバーはVSCodeから動かしていくので、今は止めたままで大丈夫です。

第2章　VSCodeの設定

　本書では「Visual Studio Code」（VSCode）で開発を進めていくことを前提にしますので、事前にインストールをしていてください。
　この章での設定は任意であり、本書では以下の設定で進めていきます。

♦ここで学べること
・VSCodeの拡張機能
・Prettierの設定

2.1　プロジェクトを開く

　ターミナルもしくはコマンドラインでmy-vite-todoフォルダーまで移動し、以下を入力してください。

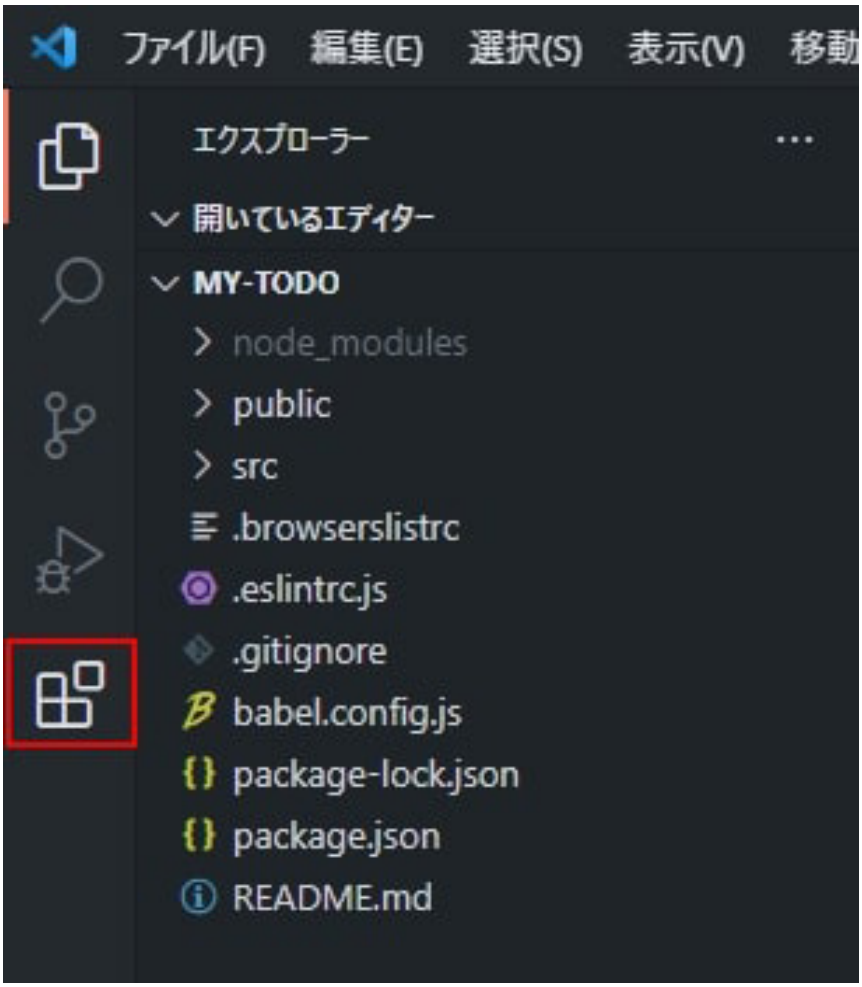
```
# code .（ドット）
$ code .
```

　VSCodeでmy-vite-todoプロジェクトを開くことができたでしょうか。もしできなかった場合は先にVSCodeを手動で起動し、メニューの「ファイル(F)」-「フォルダーを開く...」でmy-vite-todoを選択してください。my-vite-todoフォルダーを直接VSCodeにドラッグ＆ドロップする方法もあります。

2.2　拡張機能をインストールする

　VueをVSCodeを使って開発する上で快適にコーディングするために、拡張機能をインストールしていきます。以下で紹介する拡張機能は、インストールしてもしなくても構いません。
　サイドバーにある「拡張機能」アイコンを押します。

図2.1:



検索欄に拡張機能名を入力して、目的の拡張項目を探していきます。結果が表示されたら「インストール」ボタンでインストールを行います。検索結果を選択することで、右側にその機能の詳細が表示されます。

図2.2:

それでは、以下の拡張機能をインストールしてください。

・Volar

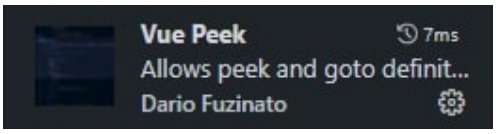
図2.3:

・Vue 3 Snippets

図2.4:

・Vue Peek

図2.5:

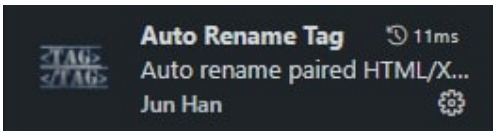


・Prettier

図2.6:

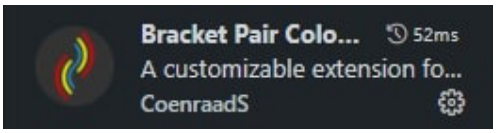
・Auto Rename Tag

図2.7:



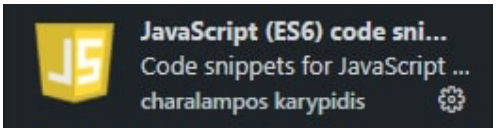
・Bracket Pair Colorizer 2

図2.8:



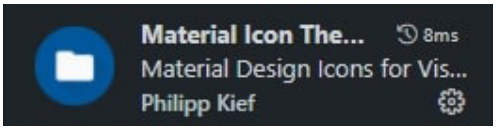
・JavaScript (ES6) code snippets

図2.9:



・Material Icon Theme

図2.10:



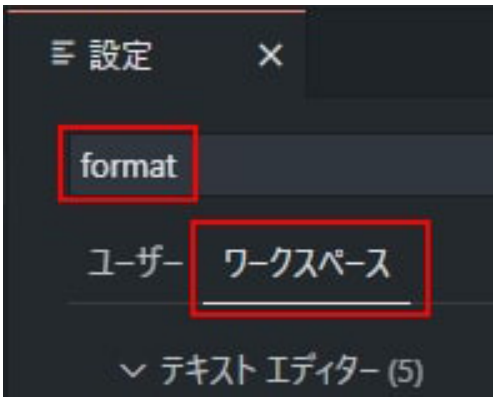
2.3　Prettierの設定を行う

　Prettierとは、ソースコードを自動で整形してくれるコードフォーマッターのことです。チーム開発において各自の整形ルールがバラバラだと統一性がなく、コードレビューしたときに読みづらくなってしまいます。
　文字列を囲むのは「'」（シングルクォート）なのか「"」（ダブルクォーテーション）なのかという些細なことも、Prettierに任せればどちらかに統一してくれます。
　それでは設定していきましょう。まず、サイドバーの下のほうにある「管理」アイコンを押し、「設定」を選択します。

図2.11:

「設定」タブの「設定の検索」欄に「format」と入力してください。その下に「ユーザー」と「ワークスペース」があります。「ユーザー」を選ぶとワークスペース全体、「ワークスペース」を選ぶと今回の場合、my-vite-todoプロジェクトのみの設定となります。今回はどちらでも構いません。

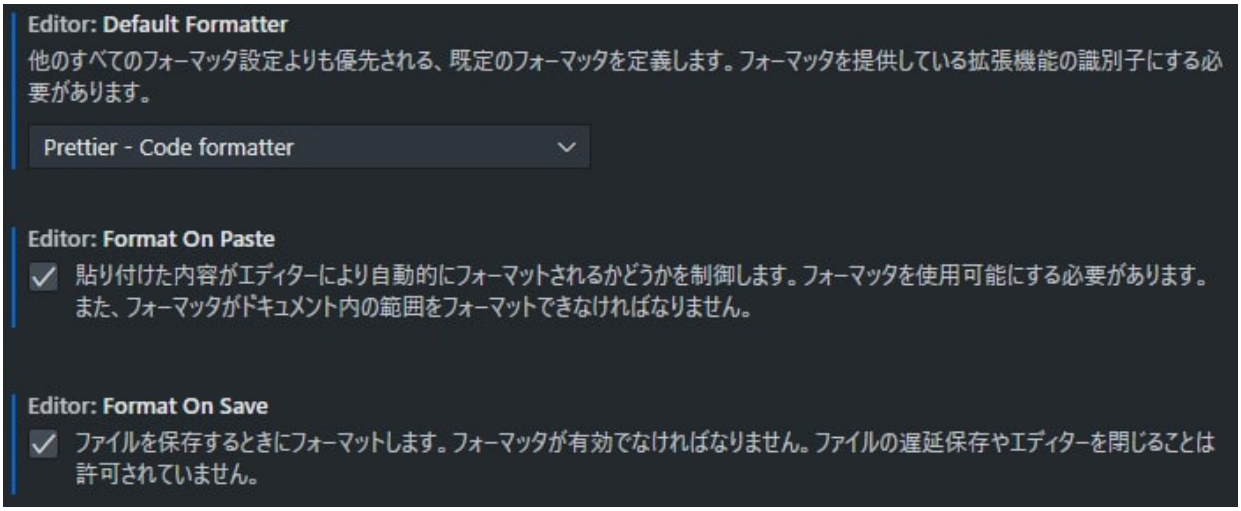
図2.12:



以下の通りに設定してください。

・「Editor: Default Formatter」から「Prettier - Code formatter」を選択
・「Editor: Format On Paste」をチェック
・「Editor: Format On Save」をチェック

図2.13:



ここからは好みの問題もありますが、JavaScriptでの文字列等を囲む記号をシングルクォートにするかダブルクォーテーションにするかの設定を行います。本書ではHTMLはダブルクォーテーション、JavaScriptは基本的にシングルクォートを使う設定にします。

「設定の検索」欄に「single quote」と入力します。そして「Prettier: Single Quote」をチェックします。

図2.14:

2.4　Prettierの設定を確認する

それではApp.vueを開き、以下の通りにコードを変更してください。

・import HelloWorld from './components/HelloWorld.vue';のコードの「;」（セミコロン）を削除。「'」（シングルクォート）は「"」（ダブルクォーテーション）に変更

・の「"」（ダブルクォーテーション）を「'」（シングルクォート）に変更

そのまま保存してください。

どうでしょうか。

・import HelloWorld from "./components/HelloWorld.vue"のダブルクォーテーションはシングルクォートになり、セミコロンが復活する

・のシングルクォートはダブルクォーテーションに変更される

以上のように保存時にコードが修正されていたら、Prettierの設定はうまくいっています。

余分な空白や空行、おかしなインデントにして保存してみてください。きれいなフォーマットに整形されることを確認してください。

第3章　GitHubとVercelの設定

　この章はWeb公開しない場合、読み飛ばしても問題ありません。またGitはインストールされており、GitHubとVercelのアカウントは持っている前提で話を進めていきます。まずはGitHubにログイン（Sign in）してください。

♦ここで学べること
・GitHubへのソースの連携
・GitHubとVercelの連携
・Webアプリの公開

3.1　リポジトリーを作成する

GitHubの画面にあるRepositoriesにある「New」ボタンを押します。

図3.1:

「Create a new repository」画面の「Repository name」にプロジェクト名（my-vite-todo）を入力してください。その他は特に何もしなくても大丈夫です。「Public」は今からコミットするソースを公開したい場合、「Private」は公開しない場合の設定です。どちらでも構いません。
そのまま「Create repository」ボタンを押してください。

図3.2:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.

Owner * Repository name *

[redacted] / my-vite-todo

Great repository names are short and memorable. Need inspiration? How about **jubilant-enigma**?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:
Skip this step if you're importing an existing repository.

- [ ] **Add a README file**
  This is where you can write a long description for your project. Learn more.
- [ ] **Add .gitignore**
  Choose which files not to track from a list of templates. Learn more.
- [ ] **Choose a license**
  A license tells others what they can and can't do with your code. Learn more.

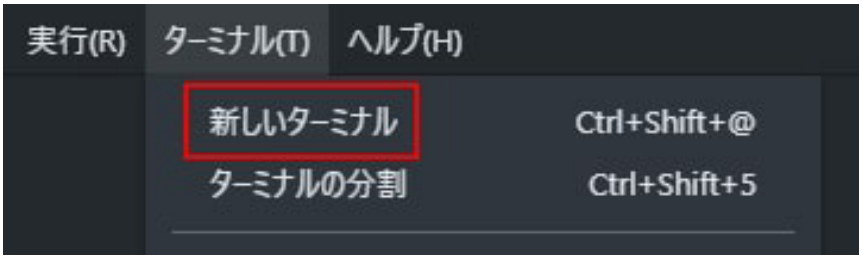
Create repository

3.2　ソースを連携する

　先ほど作ったGitHubのリポジトリーに、my-vite-todoのソースをコミットしていきましょう。ターミナルやコマンドラインでもできますが、ここではVSCode上でコミットします。

　VSCodeのメニューから「ターミナル」-「新しいターミナル」を押すと、画面下にターミナル画面が表示されます。

図3.3:



　そのターミナルで以下のコマンドを実行してください。

```
$ git init
$ git add .
$ git commit -m "first commit"
$ git branch -M main
$ git remote add origin git@github.com:<リポジトリー名>/my-vite-todo.git
$ git push -u origin main
```

実行後のターミナルに「Branch 'main' set up to track remote branch 'main' from 'origin'.」と表示されていれば成功です。

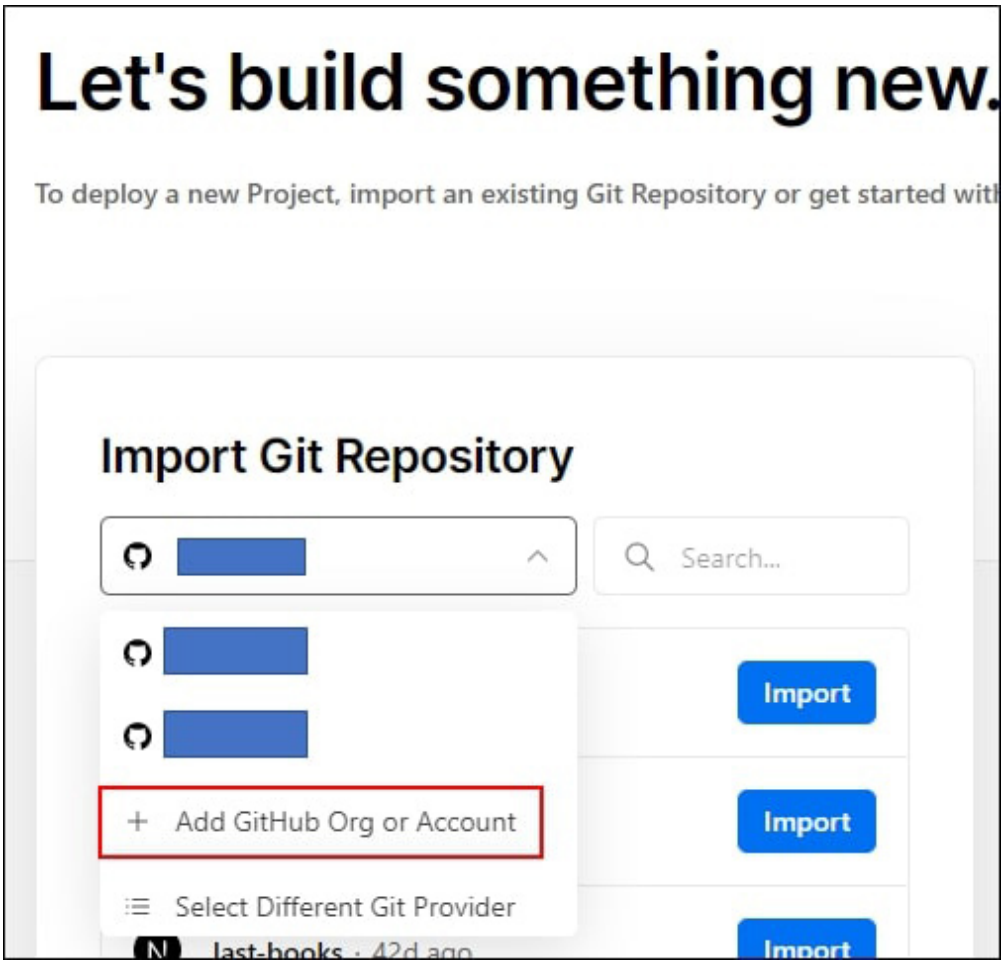
GitHubの画面に戻り、画面を更新してください。my-vite-todoのソースがアップロードされているはずです。

3.3　ページをWebに公開する

　続いて、このままの状態でWebに公開してみましょう。Webに公開するには、Vercelという会社のホスティングサービスを利用します。Vercelを利用すると、GitHubにコミットするだけで自動的にビルドとデプロイが実行されてWebに公開してくれます。個人で利用するには無料枠（Hobby）で充分です。
　Webに公開する上で、きちんとローカルで動いている状態を確認し、小さい単位で公開することは大事です。一気に公開してしまうと、Web上で動かなかった場合の原因究明が困難になります。

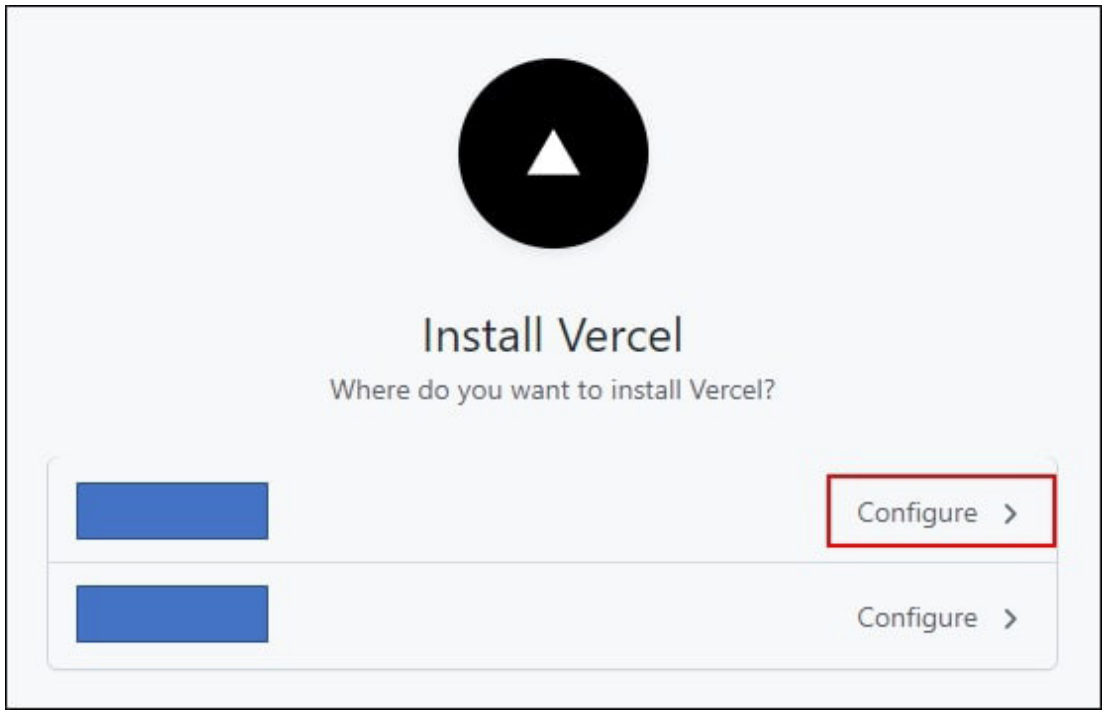
　まずはVercelにログインしてください。その後「New Project」ボタンを押します。すると、次の画面でGitHubのリポジトリー一覧が表示されると思います。もし「my-vite-todo」リポジトリーが表示されていない場合、「Import Git Repository」欄の下のプルダウンを開き、「Add GitHub Org or Account」を選択します。

図3.4:



「Install Vercel」画面で自分のリポジトリーの「Configure」を選択します。GitHubの確認パスワードを求められればそれに応じます。

図3.5:



GitHubのリポジトリーの権限設定をする画面に遷移したら、「Repository access」の「All repositories」（全リポジトリーを許可）か「Only select repositories」（選択したリポジトリーのみ）のどちらかをチェックします。

「Only select repositories」を選択した場合、Select repositoriesプルダウンからmy-vite-todoリポジトリーを選択します。その下の一覧にmy-vite-todoリポジトリーが追加されたことを確認します。

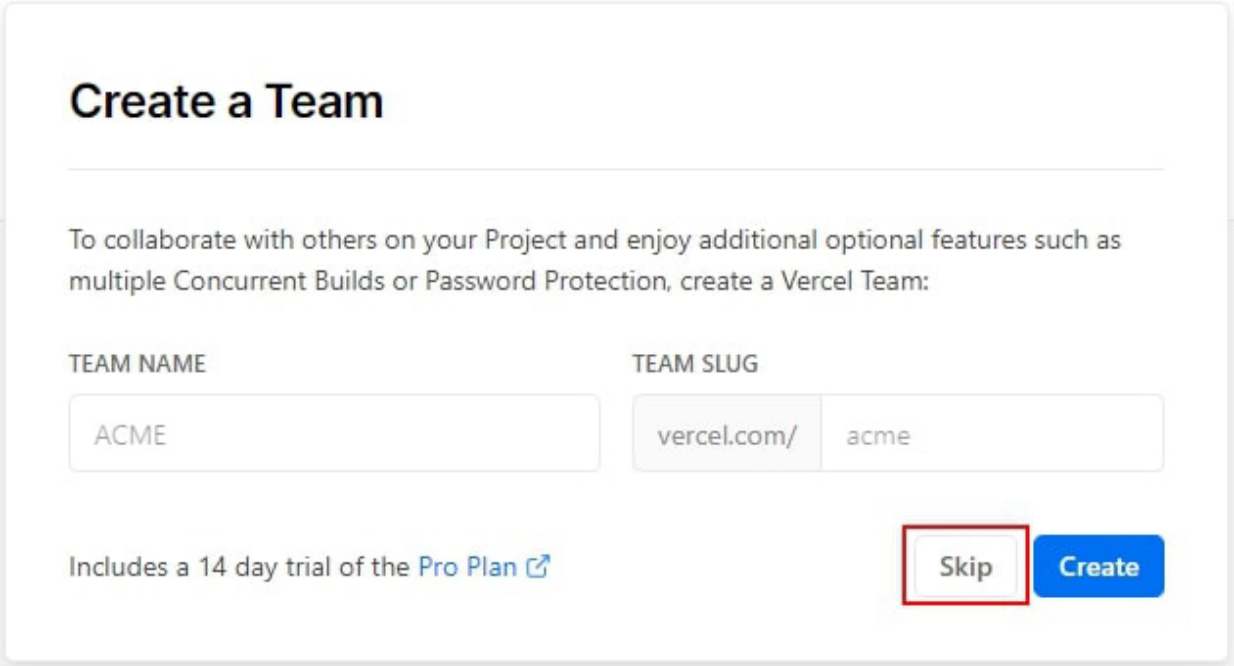
最後に「Save」ボタンを押して完了です。

図3.6:

再びVercelの画面に戻ってきましたら、「Import Git Repository」欄にmy-vite-todoが表示されていますので、その右側にある「Import」ボタンを押します。

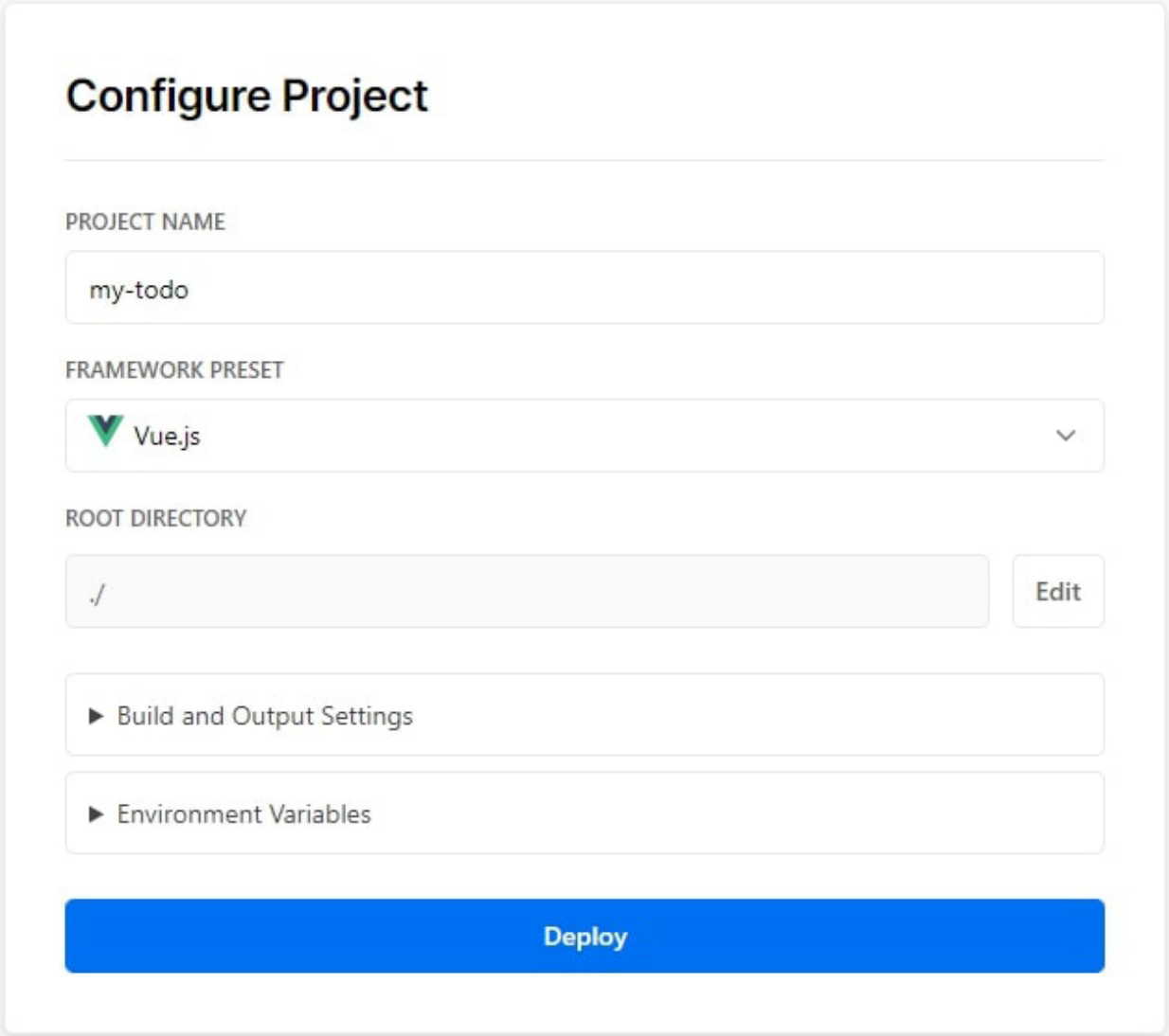
「Create a Team」はそのまま何もせずに、「Skip」ボタンを押してください。

図3.7:



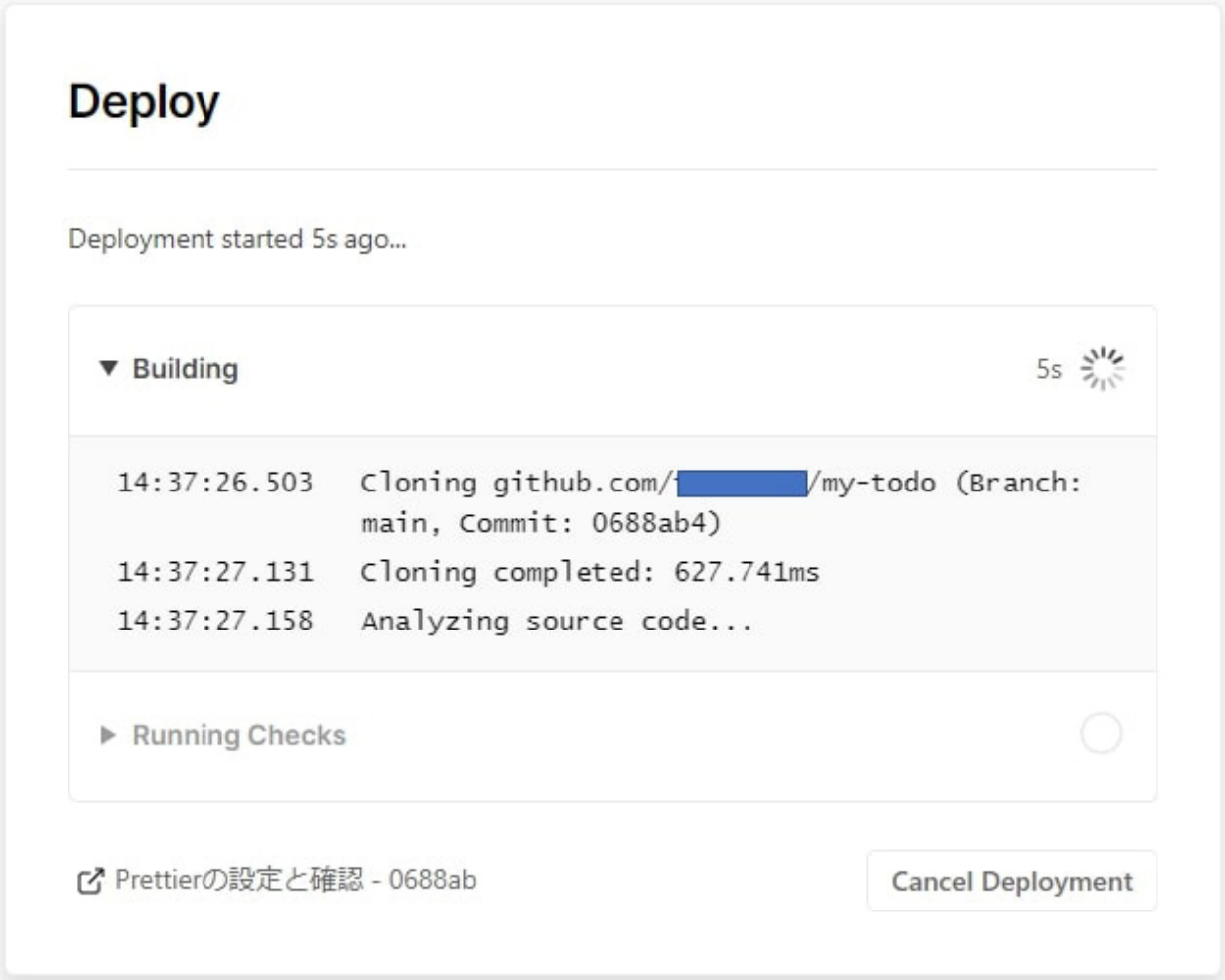
　すると、そのまま下にある「Configure Project」に移動するので、「Deploy」ボタンを押してください。

図3.8:



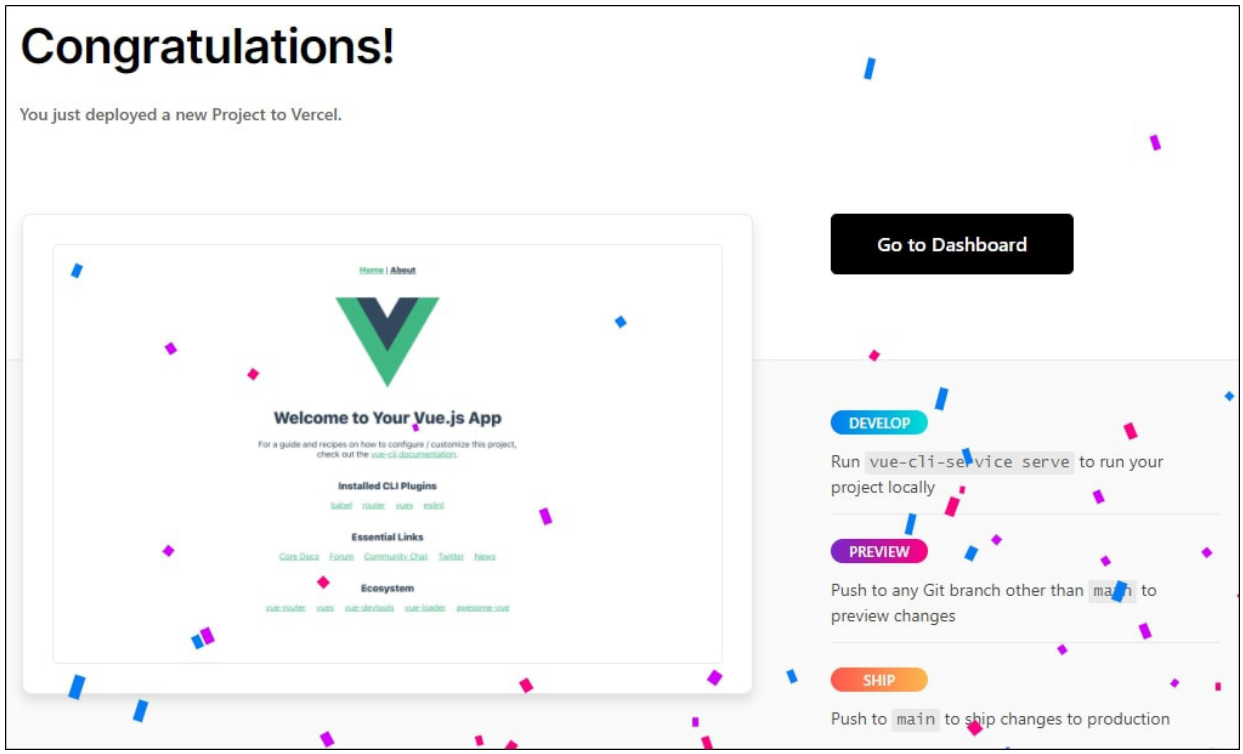
　デプロイが始まります。

図3.9:



Congratulations!の表示とともに紙吹雪が舞うと、デプロイ成功です！

図3.10:



「Go to Dashboard」ボタンを押し、ダッシュボード画面の右側にある「Visit」ボタンを押してください。

https://my-vite-todo-xxxx.vercel.app/

というあなた専用のアドレスでWeb公開されました。URLの「xxxx」は一意にするために割り与えられる文字列です。「my-vite-todo」というプロジェクト名で公開する人が他にもいると、被らないようにVercel側で自動的に付与されるものです。もし「my-vite-todo」というプロジェクト名が最初の人であれば、「xxxx」というのは付与されません。

お疲れさまでした！ここまで面倒な設定をしてきましたが、今後の他のプロジェクトでも応用は利きますので、覚えていて損はありません。

それでは、次の章からVue3を学習していきましょう！

第4章　Vueの基本

　この章では、Vue3の特徴と簡単な説明をします。Vueはバージョン2から3に変わり、後述するOptions APIからComposition APIと書き方、考え方も変わりました。
　Vue3から始める方は問題ありませんが、Vue2からの方は書き方が少し変わりますので、混乱しないようにしてください。

　♦ここで学べること
　・Vue3の構文
　・Vueの起動

4.1　SPA（Single Page Application）

　Vueで作るアプリケーションはSPA（シングル・ページ・アプリケーション）と呼ばれ、HTMLファイルはpublic配下のindex.htmlしか使いません。

　index.htmlの中を見ていくと、main.jsが呼び出されています。main.jsを見ると、createApp(App).mount('#app');という部分があり、index.htmlにある にマウントしていることが読み取れます。

　createApp(App)でApp.vueを取り込み、 に差し込むという簡単な認識で今は問題ありません。

　基本的にApp.vue以下を触っていくため、index.htmlを触ることはありませんが、<html lang="en">が英語になっているので<html lang="ja">と日本語に修正しておきましょう。

4.2　単一ファイルコンポーネント

　基本的なvueファイルには特別な書き方があり、以下の 3 つのパートに分かれています。

・<template>　・・・　テンプレート（HTMLを書くところ）
・<script>　　・・・　ロジック（JavaScriptを書くところ）
・<style>　　・・・　スタイル（CSSを書くところ）

　このようにテンプレート、ロジック、スタイルをひとつのファイルに書ける形式を「単一ファイルコンポーネント（SFC）」と呼び、「*.vue」ファイルで表します。

4.3　Options APIとComposition API

　Vue構文の特徴である<template>、<script>、<style>のうち、<script>について見ていきます。
　Vue2までの<script>内はOptions APIという、以下のような書き方でした。

リスト4.1:

```
 1: <script>
 2: import HelloWorld from '@/components/HelloWorld.vue';
 3:
 4: export default {
 5:   components: {
 6:     HelloWorld,
 7:   },
 8:   data () {
 9:     return {
10:       count: 0,
11:     }
12:   },
13:   methods: {
14:     increment() {
15:       this.count++;
16:     },
17:   },
18:   ...
19: };
20: </script>
```

　それが、Vue3以降はComposition APIという書き方にガラッと変わりました。大雑把に言いますと、export default内のdataや

mounted、methodsというものがVue3ではsetup関数内に書くようになりました。

リスト4.2:

```
 1: <script>
 2: import { ref } from "vue";
 3:
 4: export default {
 5:   setup() {
 6:     const count = ref(0);
 7:     const increment = () => {
 8:       count.value++;
 9:     };
10:     return {
11:       count,
12:       increment,
13:     }
14:   },
15: };
16: </script>
```

　setup内で定義した変数や関数は、returnすることで<template>で使うことができます。

4.4　Vue3.2のsetup

　Vue3.0から導入されたsetup関数ですが、Vue3.2から更に書き方が変わっています。しかしご安心ください。これはsetupの糖衣構文となっており、若干便利な書き方となっております。
　先ほどの例をsetupの糖衣構文で書き直してみます。

リスト4.3:

```
<script setup>
import { ref } from "vue";

const count = ref(0);
const increment = () => {
  count.value++;
};
</script>
```

　ずいぶんとスッキリとしたのではないでしょうか。

　<script>タグを<script setup>とすることでexport default、setup、returnが不要になりました。

　また、今までは<template>、<script>だった順番も<script>、<template>と入れ替わりました。この順番は入れ替えても問題ありませんので、ご自身の使いやすい順番で書いてください。

　本書はこのsetupの糖衣構文、そして<script>、<template>の順番で書いていきます。

4.5　Vueを起動する

　次の章から実際にサンプルアプリを作っていきますので、Vueを起動しておいてください。VSCode上でターミナルを開き、以下のコマンドを実行します。

```
# npmの場合
$ npm run dev

# yarnの場合
$ yarn dev
```

第5章　TODOアプリ

　ここからは実際のアプリ作りを通して、Vueの使い方を学んでいきます。サンプルアプリとしてよく例に挙げられる、TODOアプリを作っていきます。

♦ここで学べること
・Vueのコンポーネント
・CSSのスコープ（scoped）

5.1　サンプルアプリ

本書では、このようなTODOアプリを作成していきます。

図5.1:

ごく一般的なTODOアプリで、入力して「追加」ボタンを押すと一覧に表示され、そのTODOを編集したり削除したりできます。そして、チェックを付けると、取り消し線でそのTODOを達成したことがわかるようになっています。

5.2　レイアウトを決める

　サンプルアプリのようにヘッダー、メイン、フッターと縦に 3 分割したレイアウトを作ります。ヘッダーとフッターは固定で、メインの部分が変わっていくイメージです。

　App.vueを開き<template>と<style>だけにしてしまいましょう。

リスト5.1: my-vite-todo/src/App.vue

```
1: <template>
2:   <div class="wrap">
3:     <div>ヘッダー</div>
4:     <main class="main">メイン</main>
5:     <div>フッター</div>
6:   </div>
7: </template>
8:
9: <style></style>
```

単一ルート要素

Vue2までは<template>配下の要素（タグ）は のようにトップレベルのタグで囲んでひとつの要素にする必要がありましたが、Vue3ではその必要はなくなりました。 がなく、 タグのヘッダー、メイン、フッターがトップレベルに 3 つあるとVue2ではエラー、Vue3ではOKということになります。

画面で確認すると、左上に固まって表示されたのではないでしょうか。これを画面の中央に、フッターは画面の下になるようにしていきます。

<style>に以下を書き加えてください。

リスト5.2: my-vite-todo/src/App.vue

```
<style>
.wrap {
  display: flex;
  flex-direction: column;
  align-items: center;
  min-height: 100vh;
  width: 370px;
  margin: 0 auto;
  font-family: sans-serif;
}

.main {
  flex: 1;
  width: 100%;
}
</style>
```

5.3　リセットCSSでデザインしやすくする

　リセットCSSとは、各ブラウザーでの表示の違いを打ち消すためのCSSファイルのことです。これを使うことで、どのブラウザーで表示させても同じような表示にすることができます。これも必ず使わなければならないものではありませんが、本書では意図したデザインにならなかったため、使うことにしました。

　このリセットCSSもいろいろな種類のものが出ていますが、本書ではまっさらな状態から始められる「destyle.css」というものを使います。

　使い方は簡単です。assetsフォルダー配下にcssフォルダーを作ってください。次に以下のサイトに行き、「Source code(zip)」をダウンロードします。そして、それを解凍した配下にある「destyle.css」をcssフォルダにコピーしてください。

https://github.com/nicolas-cusan/destyle.css/releases/tag/v3.0.2

destyle.cssのバージョン

destyle.css のバージョンは必ず v3.0.2 を指定してください。v4.0.0 ではチェックボックスがうまく表示されない可能性があります。

ファイルを作成した後は、そのファイルをVueアプリケーションに適用します。トップレベルのファイルであるApp.vueに取り込むことにより、その下の階層にも適用されます。

@import 'assets/css/destyle.css';を<style>の先頭行に追加してください。

リスト5.3: my-vite-todo/src/App.vue

```
 1: <style>
 2: @import 'assets/css/destyle.css';  // 追加
 3:
 4: .wrap {
 5:    (略)
 6: }
 7:
 8: .main {
 9:    (略)
10: }
11: </style>
```

先ほどより画面の余白がなくなっていたら、うまく適用されています。リセットCSSを使うことで、今までタグに自動的に付与されていたスタイルもなくなっていることに注意してください。

たとえば<h1>タグは書くだけでフォントサイズが大きくなったり、上下のpaddingもある程度あったと思いますが、リセットCSS適用後はただの文字列で表示されます。

ただこれも慣れの問題で、余計なスタイルのために表示が崩れたり、異なるブラウザーでもスタイルが崩れたりすることがなくなりますので、快適にデザインすることができると思います。

5.4　コンポーネントを理解する

　Vue開発の特徴のひとつに、コンポーネントというものがあります。いろいろなサイトを見てみると、ヘッダー、フッター、メインやボタンなど様々な部品が集まってひとつの画面を構成していることがわかります。そのひとつひとつをコンポーネントという単位に分け、再利用できるようにしています。

　Vueのページはこのコンポーネントが集まって構成されていることになります。vueファイルというのは、コンポーネント化された部品そのものです。

　HTMLのタグを作るようなものといえばイメージしやすいでしょうか。たとえば<input type="text">と書けば、入力欄が表示されます。これを利用し、「InputDouble.vue」というようなvueファイルを作り、<input type="text"><input type="text">とふたつinputタグを書きます。そして<InputDouble />と呼び出せば、画面に入力欄がふたつ続いたものが表示されます。

　この規模でコンポーネント化する必要はないのかもしれませんが、責務を分ける意味でも、ヘッダーとフッターをコンポーネント化していきましょう。

5.5　ヘッダー、フッターをコンポーネント化する

　components フォルダーにある HelloWorld.vue を削除し、TheHeader.vueとTheFooter.vueファイルを作成してください。

コンポーネント名

　コンポーネント化するvueファイルの名前は
・パスカルケース　→　TheHeader.vue
・ケバブケース　　→　the-header.vue
　のように複数単語の名前にしてください。1 単語でも動作しますが、2 単語以上にすることが推奨されています。
・Header.vue　　→　×
・TheHeader.vue　→　○
　これは将来、新しいHTMLのタグが増えてもタグ名の衝突が起きないようにするためです。

　TheHeader.vueの内容は以下の通りです。scopedは一旦無視してそのまま書いて進めてください。

リスト5.4: my-vite-todo/src/components/TheHeader.vue

```
<template>
  <h1 class="title">TODO</h1>
</template>

<style scoped>
.title {
  width: 100%;
  background-color: #e3f2fd;
  text-align: center;
  font-size: 32px;
  font-weight: bold;
  padding: 8px 0;
}
</style>
```

　TheFooter.vueの内容は以下の通りです。

リスト5.5: my-vite-todo/src/components/TheFooter.vue

```
 1: <template>
 2:   <footer class="footer">
 3:     <small>&copy;my-vite-todo</small>
 4:   </footer>
 5: </template>
 6:
 7: <style scoped>
 8: .footer {
 9:   margin-top: 15px;
10:   height: 30px;
11: }
12: </style>
```

次にそのヘッダーとフッターのコンポーネントを取り込むため、App.vueを修正します。

リスト5.6: my-vite-todo/src/App.vue

```
 1: <script setup>
 2: import TheHeader from './components/TheHeader.vue';
 3: import TheFooter from './components/TheFooter.vue';
 4: </script>
 5:
 6: <template>
 7:   <div class="wrap">
 8:     <TheHeader />
 9:     <main class="main">メイン</main>
10:     <TheFooter />
11:   </div>
12: </template>
```

<script>内で先ほど作ったヘッダーとフッターをimportします。構文は以下の通りです。

リスト5.7:

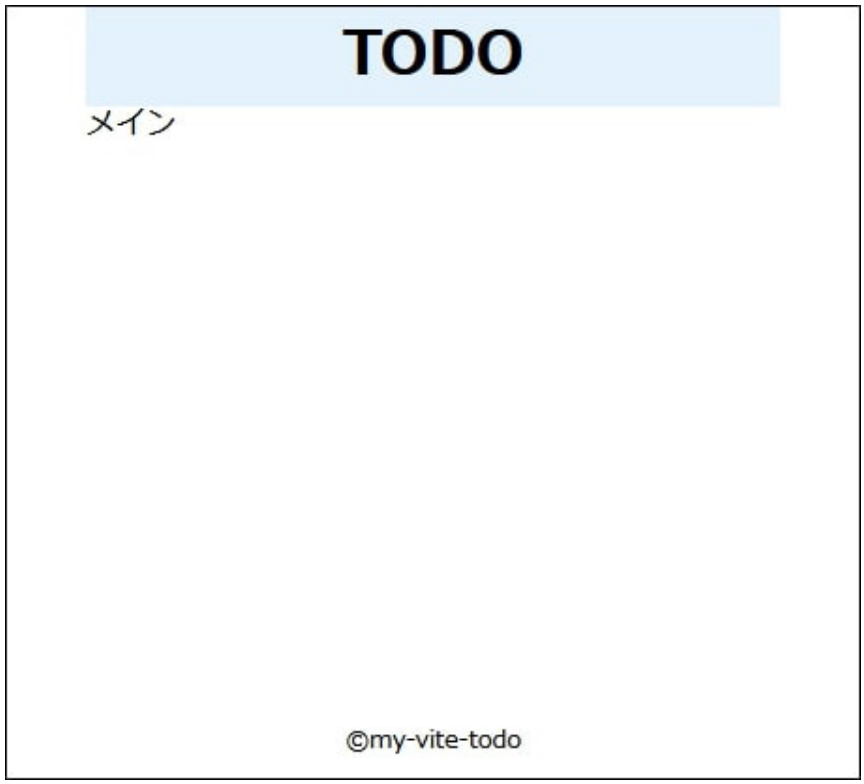
```
import ＜名称＞ from '＜vueファイルのパス＞';
```

そして、取り込んだ名称をタグ形式にして<template>配下の任意の場所に置きます。importしたときの名称と<template>で使うときの名称は、合わせる必要があります。

import AaaBbb from ～;とすれば、<template>では<AaaBbb />（パスカルケース）もしくは、<aaa-bbb />（ケバブケース）という形で使えます。

ここで画面のほうを確認してみましょう。以下のようになっていればOKです。

図5.2:



5.6　コミットとプッシュを行う

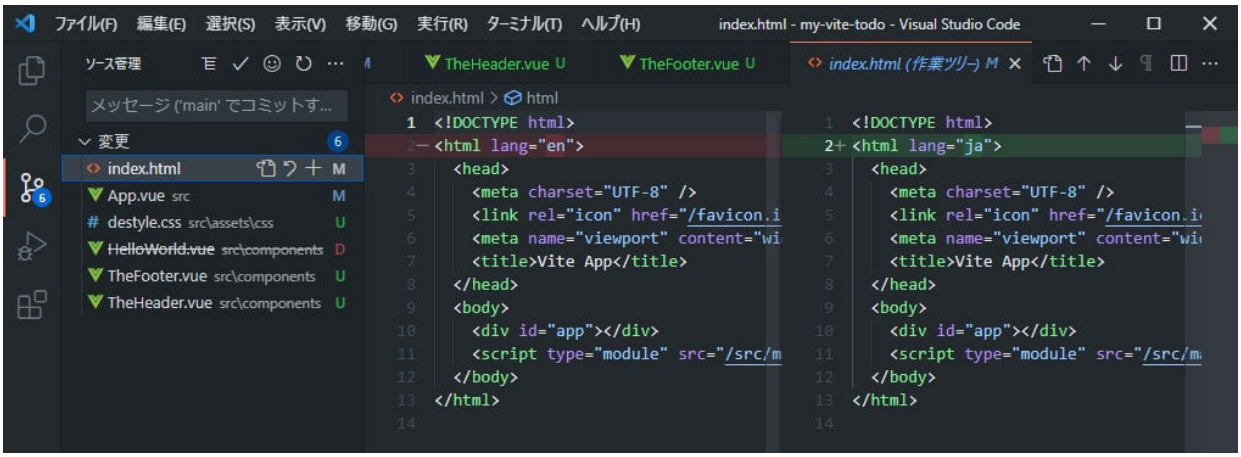
　それでは、ここまでの変更をGitHubにコミットしてプッシュしておきましょう。
　サイドバーの「ソースの管理」のアイコンに「6」と表示されているところを押します。すると、変更されたファイルが表示されていると思います。

図5.3:

　まずはindex.htmlファイルを選択してみましょう。すると隣の欄に変更箇所が表示されます。前回のコミットした分との比較ですが、左側が変更前、右側が変更後のソースになります。

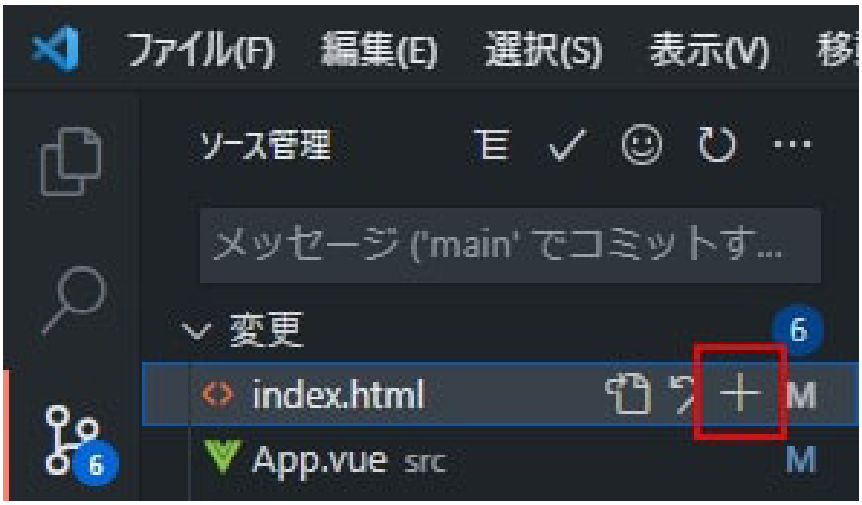
今回のindex.htmlファイルの変更は、langを「en」から「ja」にしました。

図5.4:



変更箇所を確認して問題がないのなら、ファイル右横にある「＋」を押してファイルをステージングに持っていきます。これはコミットする対象を選ぶ作業です。

図5.5:



その後、コミットメッセージを入力（「lang属性を日本語化」）し、その上にある「レ」チェックを押してください。するとコミットが完了します。他のファイルとは意味合いが違うので、コミットを分けます。

図5.6:

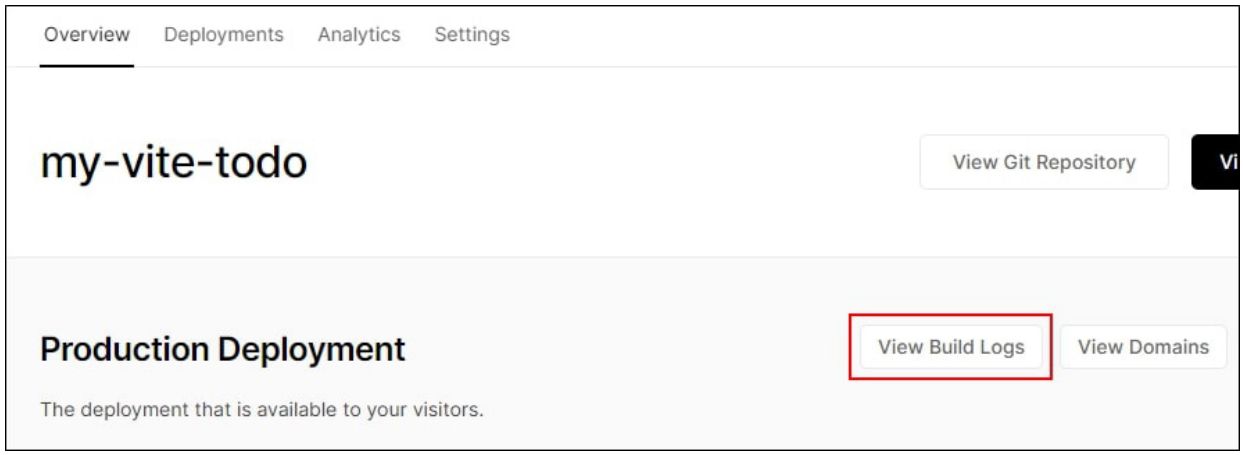
同じように残りの5ファイルも変更を確認したら、まとめてコミットしてください。コミットメッセージは「TODOレイアウト」とでもしましょう。

コミットが完了すると、VSCodeの左下に図のような上下の矢印で「0↓ 2↑」となっている部分があります。これを押すことで、GitHubにプッシュ（反映）することができます。

図5.7:

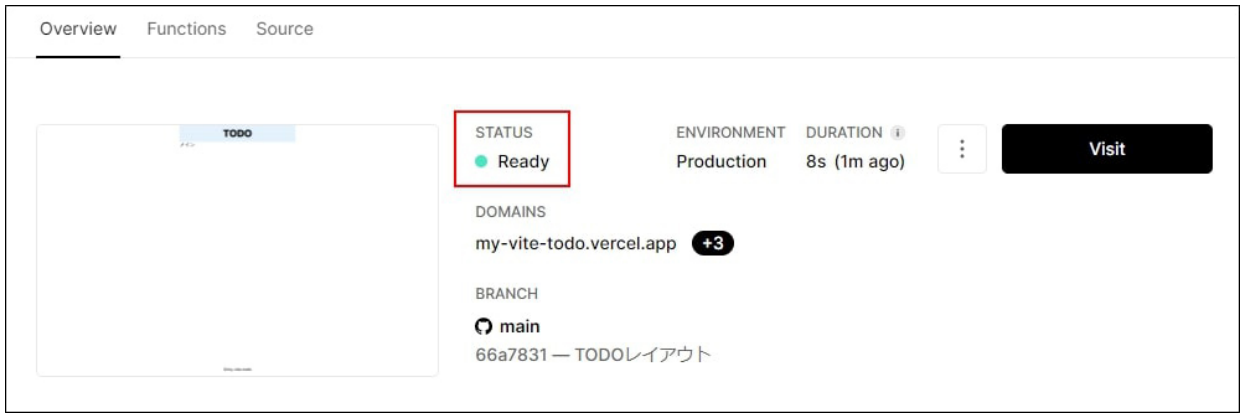
GitHubにプッシュできたことを確認した後、Vercelのサイトを見てください。「View Build Logs」ボタンを押すと、次画面の「Building」欄にデプロイしている様子が表示されます。

図5.8:



しばらく待って「STATUS」が「Ready」になると成功です。

図5.9:



その後、「Visit」ボタンを押して自分のサイトで確認してください。

5.7　scopedとは

　ここで<style>のscopedについて説明します。<style>にscoped属性を持つと、そのCSSは自身のコンポーネントの要素にのみ適用されます。

　簡単な実験をしてみましょう。フッターコンポーネントの<style>にヘッダーコンポーネントと同じクラス（title）を定義し、フォントの色を赤にしてください。

リスト5.8: my-vite-todo/src/components/TheFooter.vue

```
<style scoped>
.footer {
  margin-top: 15px;
}

// 追加
.title {
  color: red;
}
</style>
```

　このまま画面を再描画しても、ヘッダーの「TODO」は何も変わりません。次に<style scoped>からscopedを削除して、<style>としてみてください。

　どうでしょうか。ヘッダーの「TODO」が赤色に変わったと思います。このように、<style>にscoped属性を持っていないとグローバルなCSSになります。

scoped属性を持つことによって自身のコンポーネント内でのみ適用されるので、違うコンポーネントで同じクラス名を使っても競合することはありません。基本的に、scoped属性は常に持つようにしたほうがいいでしょう。

フッターに追加したtitleのCSSは使わないので、削除しておいてください。

第6章　TODOの登録

　ここからはTODOアプリのメイン部分に入っていきます。入力された値の取り方や、onClickなどのイベントの発生方法を学びます。JavaScriptやjQueryとの違いを感じてください。

♦ここで学べること
・双方向データバインディング（v-model）
・マスタッシュ構文
・イベントハンドリング
・アロー関数
・ローカルストレージへの登録

6.1　入力欄を作る

　まずはメイン部分もコンポーネント化します。componentsフォルダーに「MainTodo.vue」を作成し、TODOを登録するための入力欄と追加ボタンを記述します。

リスト6.1: my-vite-todo/src/components/MainTodo.vue

```
 1: <script setup></script>
 2:
 3: <template>
 4:   <div class="box_input">
  5:              <input   type="text"   class="todo_input"
placeholder="＋ TODOを入力" />
 6:     <button class="btn">追加</button>
 7:   </div>
 8: </template>
 9:
10: <style scoped>
11: .box_input {
12:   margin-top: 20px;
13: }
14:
15: .todo_input {
16:   width: 300px;
17:   margin-right: 8px;
18:   padding: 8px;
19:   font-size: 18px;
20:   border: 1px solid #aaa;
21:   border-radius: 6px;
22: }
23:
24: .btn {
25:   padding: 8px;
26:   background-color: #03a9f4;
```

```
  border-radius: 6px;
  color: #fff;
  text-align: center;
  font-size: 14px;
}
</style>
```

これをApp.vueから呼び出します。

リスト6.2: my-vite-todo/src/App.vue

```
<script setup>
import TheHeader from './components/TheHeader.vue';
import TheFooter from './components/TheFooter.vue';
import MainTodo from './components/MainTodo.vue';   // 追加
</script>

<template>
  <div class="wrap">
    <TheHeader />
    <main class="main"><MainTodo /></main>  // 変更
    <TheFooter />
  </div>
</template>
```

画面に入力欄ができました。追加ボタンを押しても今は何も起こりません。

図6.1:


TODO
＋ TODOを入力
追加
©my-vite-todo

6.2　入力欄と値を連動させる

　TODOを登録するために、入力欄の値を取らないといけません。素のJavaScriptなら、document.getElementById等のDOM操作で入力欄から値を取ってきました。

　Vueの場合は「双方向データバインディング」と呼ばれるもの（v-model）を利用して、data変数とinputのようなフォームの値とを連動させることができます。
　実際にやってみましょう。

リスト6.3: my-vite-todo/src/components/MainTodo.vue

```
<script setup>
// 追加
import { ref } from 'vue';
const todoRef = ref('');
</script>

<template>
  <div class="box_input">
    <input
      type="text"
      class="todo_input"
      v-model="todoRef"  // 追加
      placeholder="＋ TODOを入力"
    />
    <button class="btn">追加</button>
  </div>
</template>
```

１．import { ref } from 'vue';でref関数をvueから取り込みます。

2．const todoRef = ref('');でtodoRef変数を作成します。
3．<input>タグにv-model="todoRef"を追加します。
<input>タグにv-model="変数"とすることで<input>欄に入力した値が変数に反映されます。

6.3　マスタッシュ構文とは

　v-modelがうまく連動されているか、マスタッシュ構文を使って確認してみましょう。
　マスタッシュ構文とは、二重中括弧を利用したテキスト展開のことです。簡単に言えば、JavaScriptの変数の値をHTML内で見ることができます。先ほどの変数：todoRefを例に試してみましょう。

　<template>内の任意の場所に{{ todoRef }}を入れてみましょう。すると、<input>タグに入力した内容がすぐさま画面に表示されたはずです。これは変数：todoRefと<input>タグがv-modelを通して連動しているため、入力後すぐに反映されるからです。

templateでのリアクティブ変数の見方

<template>内でリアクティブ変数の値を見るには変数名のみでOKです。

→　todoRef

確認後、マスタッシュ構文は必要ないので、消しておいてください。

6.4　@clickを使ってみよう

　ボタンが押されたときのイベントハンドリングとしてonClickイベントがありますが、Vueの基本構文はv-on:clickと書きます。

　それの省略記法として「v-on」の代わりに「@」と置き換えて使うことができます。

・v-on:click　→　@click

　本書ではその省略記法である@clickを使っていきます。@clickで指定した関数を呼び出すことができます。

リスト6.4: my-vite-todo/src/components/MainTodo.vue

```
 1: <script setup>
 2: // 追加
 3: const addTodo = () => {
 4:   console.log(todoRef.value);
 5: };
 6: </script>
 7:
 8: <template>
 9:   <div class="box_input">
10:     <input
11:       type="text"
12:       class="todo_input"
13:       v-model="todoRef"
14:       placeholder="＋ TODOを入力"
15:     />
16:     <!-- 変更 -->
17:     <button class="btn" @click="addTodo">追加</button>
18:   </div>
19: </template>
```

文字を入力後に追加ボタンを押すと、入力した文字がコンソールに表示されることを確認してください。

scriptでのリアクティブ変数の見方

　<script>内でリアクティブ変数の値を見るためには、変数名の後ろに.valueを付けます。

　　→　todoRef.value

開発者ツール

　ブラウザーのChromeやFirefoxでは「Ctrl+Shift+i」でコンソールが見られる開発者ツールが開きます。

イベントハンドリング

　イベントハンドリングにはclickの他に、change、blur、focus、keydown、keyup、mouseover、mouseout、submitなど、いろいろな種類があります。

6.5　アロー関数とは

　以下のような書き方はアロー関数と呼ばれ、最近のJavaScriptではよく書かれる構文です。それまでの書き方でも動作しますが、本書ではこのアロー関数で書いていきます。

リスト6.5:

```
<script setup>
// アロー関数
const addTodo = () => {
  console.log('addTodo');
};

// こちらでもOK
function addTodo() {
  console.log('addTodo');
}

// または、こちらでもOK
const addTodo = function () {
  console.log('addTodo');
}
</script>
```

6.6　ローカルストレージに登録する

　TODOの登録はデータベースとのやり取りで行うのが多いのですが、本書ではVueの説明に重点を置きたいため、より簡単に登録できるローカルストレージを使用します。

　ローカルストレージとは、HTML5から導入されたWeb Storageというブラウザー上にデータを保存できる仕組みのひとつです。

　先ほどのaddTodo関数からローカルストレージにデータを保存してみましょう。addTodoを以下のように変更します。

リスト6.6: my-vite-todo/src/components/MainTodo.vue

```
1: const addTodo = () => {
2:   localStorage.todoList = todoRef.value;
3: };
```

　ローカルストレージに登録するのはとても簡単です。localStorageにドットを付けて好きな変数名（ここでは「todoList」）を付けるだけで、値を登録することができます。このまま入力欄に値を入れて「追加」ボタンを押し、開発者ツールで確認してみましょう。

　下図はchromeの開発者ツールです。

図6.2:

ローカルストレージに「todoList」という名前で入力した値が登録されていると思います。

6.7　TODOをリスト化する

　ここまででローカルストレージに入力したTODOを登録することができました。しかし、このままではひとつのTODOしか登録できませんので、登録する対象をリスト化することにします。

　まず、TODOを貯めておく配列を作成し、入力したTODOを格納していきます。その後にその配列をローカルストレージに登録します。配列はIDを持たせたいため、オブジェクトの配列にします。IDはユニークとなる値を設定しますが、ここでは簡単にミリ秒をIDとします。

リスト6.7: my-vite-todo/src/components/MainTodo.vue

```
 1: <script setup>
 2: import { ref } from 'vue';
 3: const todoRef = ref('');
 4: const todoListRef = ref([]); // 追加
 5:
 6: const addTodo = () => {
 7:   // IDを簡易的にミリ秒で登録する
 8:   const id = new Date().getTime();
 9:
10:   // 配列に入力TODOを格納
11:    todoListRef.value.push({ id: id, task: todoRef.value
});
12:
13:   // ローカルストレージに登録
14:                            localStorage.todoList        =
JSON.stringify(todoListRef.value);
15:
16:   // 登録後は入力欄を空にする
17:   todoRef.value = '';
18: };
19: </script>
```

ローカルストレージに普通に登録するとただの文字列になるため、配列やオブジェクトを登録する場合はJSONコードにシリアライズ化（JSON.stringify）する必要があります。

TODOを何個か登録してみてください。ローカルストレージにカンマ区切りで登録されていたら成功です。

リロード後のデータ

現時点ではTODOを何個か登録後にリロードし、その後にまたTODOを登録するとリロード前のTODOは上書きされます。

ここでまた、GitHubに変更をコミットしてプッシュしておきましょう。このように、ひとつの区切りや機能が終わったらコミットする習慣をつけておきましょう。

次の章からはご自身のタイミングでGitHubにコミットしてプッシュしていってください。

第7章　TODOの一覧表示

　この章では一覧表示の部分を作っていきますが、Vueの繰り返し構文：v-forを中心に見ていきます。v-forはVueでも比較的よく使われる構文なので、しっかりと覚えてください。

　♦ここで学べること
　・v-for（繰り返し構文）

7.1　TODO一覧をイメージする

　TODOの入力と登録ができたので、次は登録したTODOを表示する一覧を作っていきます。まずはTODO一覧のイメージをHTMLで書いてみましょう。

リスト7.1: my-vite-todo/src/components/MainTodo.vue

```
 1: <script setup>
 2:   （変更なし）
 3: </script>
 4:
 5: <template>
 6:   <div class="box_input">
 7:     <input
 8:       type="text"
 9:       class="todo_input"
10:       v-model="todoRef"
11:       placeholder="＋ TODOを入力"
12:     />
13:     <button class="btn" @click="addTodo">追加</button>
14:   </div>
15:   <!-- 追加 ↓ -->
16:   <div class="box_list">
17:     <div class="todo_list">
18:       <div class="todo">
19:               <input  type="checkbox"  class="check"  />
<label>TODO1</label>
20:       </div>
21:       <div class="btns">
22:         <button class="btn green">編</button>
23:         <button class="btn pink">削</button>
24:       </div>
25:     </div>
26:     <div class="todo_list">
```

```
27:       <div class="todo">
28:               <input  type="checkbox"  class="check"  />
<label>TODO2</label>
29:       </div>
30:       <div class="btns">
31:         <button class="btn green">編</button>
32:         <button class="btn pink">削</button>
33:       </div>
34:     </div>
35:   </div>
36:   <!-- 追加 ↑ -->
37: </template>
38:
39: <style scoped>
40: // 追加
41: .box_list {
42:   margin-top: 20px;
43:   display: flex;
44:   flex-direction: column;
45:   gap: 4px;
46: }
47:
48: .todo_list {
49:   display: flex;
50:   align-items: center;
51:   gap: 8px;
52: }
53:
54: .todo {
55:   border: 1px solid #ccc;
56:   border-radius: 6px;
57:   padding: 12px;
58:   width: 300px;
59: }
60:
61: .check {
62:   border: 1px solid red;
63:   transform: scale(1.6);
64:   margin: 0 16px 2px 6px;
65: }
```

```
66:
67: .btns {
68:   display: flex;
69:   gap: 4px;
70: }
71:
72: .green {
73:   background-color: #00c853;
74: }
75:
76: .pink {
77:   background-color: #ff4081;
78: }
79: </style>
```

下図のようになっていればOKです。

図7.1:

7.2　繰り返し構文

　先ほどのTODO一覧をよく見ると「TODO1」、「TODO2」と内容は違いますが、ほぼ同じ構成になっています。HTMLのほうも ～ の塊が繰り返されています。

　TODOが増えるたびにこの塊を増やしていくのは大変ですし、ソースも煩雑になっていきます。そこでVueのfor文を使って、この塊を増やしていきましょう。

　まずは簡単な例から慣れていきましょう。MainTodo.vueの任意の場所に以下を書いてください。

リスト7.2: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: const todoExample = ref(['example1', 'example2',
'example3']);
 3:
 4: // template
 5: <div v-for="(example, index) in todoExample"
:key="index">
 6: <p>{{ index }}.{{ example}}</p>
 7: </div>
```

図7.2:

Vueで繰り返しをする場合、v-forディレクティブという特別な構文を使います。HTMLタグの属性としてv-forを使うと、そのタグも含め、その配下と終了タグまでが繰り返しの対象となります。

v-forの基本的な構文はv-for="変数 in 配列"となります。v-forには仮想DOMのアルゴリズムのためにkey属性を持つことが推奨されています。このkeyはユニークである必要があります。v-forのふたつ目の引数には配列のインデックスがサポートされていますので、例ではこれを利用してkeyにインデックスを与えています。

key属性の注意点

　配列のインデックスをkey属性にするのはあまりよくありません。配列の増減によってインデックスが変わるからです。その他にユニークとなる候補がない場合の最終手段として覚えておきましょう。

　todoExampleの値を変更したり増やしたり減らしたりして、色々と試してみてください。

　確認した後は不要なので、先ほどのソースは削除しておいてください。

7.3　TODOを繰り返し表示する

それでは、実際に使う の塊を繰り返してみましょう。繰り返しの対象が のため、このタグにv-forを付けてみます。

先ほどの例はただの配列でしたが、実際に登録したものはオブジェクトの配列なので、<label>に囲まれた部分を{{ todo.task }}にします。そして、v-forのkeyもインデックスからIDにします。

リスト7.3: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // 変更
 3: const todoListRef = ref([
 4:   { id: 1, task: 'TODO1' },
 5:   { id: 2, task: 'TODO2' },
 6:   { id: 3, task: 'TODO3' },
 7: ]);
 8:
 9: // template
10: <!-- 置き換え -->
11: <div class="box_list">
12:    <div class="todo_list" v-for="todo in todoListRef"
:key="todo.id">
13:     <div class="todo">
14:       <input type="checkbox" class="check" />
15:       <label>{{ todo.task }}</label>
16:     </div>
17:     <div class="btns">
18:       <button class="btn green">編</button>
19:       <button class="btn pink">削</button>
20:     </div>
```

```
21:   </div>
22: </div>
```

下図のように、todoListRefで指定した値がTODO一覧として表示されたでしょうか。

図7.3:

これでtodoListRefに値が入れば、一覧表示される仕組みができました。

最後に、todoListRefの初期値をローカルストレージから取ってくるようにしましょう。ローカルストレージにはJSONコードにシリアライズ化されたデータが入っているので、取り出すときに配列に戻すようにします。それが以下のJSON.parseです。これを使って取り出します。

リスト7.4: my-vite-todo/src/components/MainTodo.vue

```
1: // ローカルストレージにtodoListRefが存在していればparseし、
2: // なければundifinedになるため空配列をセットする。
```

```
3: const todoListRef = ref([]);  // 変更
4: const ls = localStorage.todoList;  // 追加
5: todoListRef.value = ls ? JSON.parse(ls) : [];  // 追加
```

いかがでしょうか。これでページをリロードしてもブラウザーを閉じても、登録したTODOが消えずに表示されたかと思います。新たにTODOを追加登録しても大丈夫です。

第8章　TODOの編集・削除

　この章ではTODOの編集と削除の部分を作っていきますが、Vueのif構文：v-ifを中心に見ていきます。v-ifはv-forと同様にVueでも比較的よく使われる構文なので、しっかりと覚えてください。

◆ここで学べること
・条件付きレンダリング（v-if、v-show）
・非表示ラッパー

8.1　編集の仕様

　次はTODOを編集できるようにしましょう。TODOの横の「編」ボタンを押したときの処理です。編集においての仕様はいろいろあると思いますが、本書では以下の仕様にします。

・「編」ボタンを押すとそのTODOが入力欄に表示される。・・・ ①
・「追加」ボタンが「変更」ボタンに変わる。・・・ ②
・TODOを編集後に「変更」ボタンを押すと、編集後のTODOが一覧に表示される。・・・ ③
・「変更」ボタンを「追加」ボタンに戻す。・・・ ④
・入力欄を空にする。・・・ ⑤

8.2　入力欄にTODOを表示する

　仕様①の「編」ボタンを押したときに処理が実行されるように、編集ボタンに@clickを付与します。@clickから呼び出される関数はshowTodoとしますが、引数にTODOのIDを渡してどのTODOが編集対象かわかるようにします。

リスト8.1: my-vite-todo/src/components/MainTodo.vue

```
 1: <!-- 変更 -->
 2: <button class="btn green" @click="showTodo(todo.id)">編
</button>
```

　showTodoで受け取ったIDを利用して、入力欄に編集するTODOを表示させます。入力欄に表示させるには、v-modelで連動している変数todoRefに値を入れればよさそうです。
　TODOリスト（todoListRef）から該当のIDを持つオブジェクトを探します。配列から目的のデータを取得するには、ここではfind関数を使います。

リスト8.2: my-vite-todo/src/components/MainTodo.vue

```
 1: <script setup>
 2: // 追加
 3: const showTodo = (id) => {
 4:   // 配列（todoListRef.value）から引数のidと同じ要素を検索する。
 5:   // findの「(todo)」には配列の要素が引数として順番に入る。
 6:   // 「todo.id === id」がtrueならその時点の要素：todoが返る。
 7:     const todo = todoListRef.value.find((todo) =>
todo.id === id);
 8:   todoRef.value = todo.task; // 取得した要素からtaskを取り出す
```

```
 9: };
10: </script>
```

　これで各TODOの編集ボタンを押すたび、入力欄にそのTODOが表示されます。

find関数

構文：配列.find(callback関数)配列を順番にループし、callback関数がtrueを返す最初の要素を返します。

find関数を使うことで、for文を使うよりも配列内を検索するという意図が明確になります。

8.3　条件によってボタンを表示する

　仕様②のボタンの変更ですが、これは追加と変更のふたつのボタンを用意し、条件分岐でどちらかだけを表示させる方法にします。条件分岐にはv-ifディレクティブというものを使います。
　まずは判定する条件の変数と、追加・変更のふたつのボタンを用意します。変更ボタンを押したときの関数も空で作っておきましょう。

リスト8.3: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // 追加
 3: const isEditRef = ref(false); // 編集ボタンを押したときにtrueにする
 4: const editTodo = () => {}; // 変更ボタンを押したとき
 5:
 6: // template
 7: <!-- 変更 -->
 8:  <button class="btn green" @click="editTodo"> 変 更 </button>
 9: <button class="btn" @click="addTodo">追加</button>
```

　そして、isEditRefを条件判定に使って以下のようにボタンにv-ifを付与します。編集ボタンを押したときにisEditRefをtrueにする処理もshowTodoに入れておきます。

リスト8.4: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: const showTodo = (id) => {
 3:     const todo = todoListRef.value.find((todo) => todo.id === id);
```

```
  4:   todoRef.value = todo.task;
  5:   isEditRef.value = true; // 追加
  6: };
  7:
  8: // template
  9: <!-- 変更 -->
 10:   <button   class="btn   green"   @click="editTodo"   v-
if="isEditRef">変更</button>
 11:  <button  class="btn"  @click="addTodo"  v-else> 追 加
</button>
```

変更ボタンはv-ifでisEditRefがtrueになったら表示され、falseなったら表示されなくなります。追加ボタンのほうはv-elseで、その反対の動きになります。

8.4　v-ifとv-showの使い分け

　v-ifと同じような動きで、v-showというものがあります。使い方も同じでtrue/falseによって表示、非表示となります。

　先ほどの例をv-showに置き換えてみます。v-elseは使えないので、!isEditRefと条件を反転させます。

リスト8.5: my-vite-todo/src/components/MainTodo.vue

```
 1: // template
  2:   <button   class="btn   green"   @click="editTodo"   v-
show="isEditRef">変更</button>
    3:    <button    class="btn"    @click="addTodo"    v-
show="!isEditRef">追加</button>
```

　動かしてみても同じ動きになるはずです。では何が違うのかといいますと、HTMLを見ればわかります。v-ifの場合はfalseになったほうは描画されません。isEditRefがtrueになった段階で描画されます。

　v-showの方はといいますと、true/false関係なく描画はされます。ですが、falseの方はCSSのdisplay: none;が与えられ、画面から見えなくしています。ボタンの表示／非表示の切り替えはCSSで行います。

　Vueの公式ページによると、v-ifはtrue/falseの切り替え時に高いコストが掛かり、v-showは初期表示の描画に高いコストが掛かります。そのため、とても頻繁に何かを切り替える必要があれば v-showを選び、条件が実行時に変更することがほとんどない場合はv-ifを選びます、と書かれています。

　また先ほどのボタンの例のようにv-ifはv-elseやv-else-ifのように複数の条件で切り替えることができますので、その点も考慮してもいいのかもしれません。

8.5　非表示ラッパーとは

　今回のv-ifの対象がbuttonタグひとつでしたが、これが複数要素（タグ）の場合はどうなるのでしょうか。たとえば、以下のような感じです。isEditRefがtrueの場合に1、2を表示、それ以外の場合は3、4を表示したい場合です。

リスト8.6:

```
1: <p>1</p>  // isEditRef=true
2: <p>2</p>  // isEditRef=true
3: <p>3</p>  // isEditRef=false
4: <p>4</p>  // isEditRef=false
```

　<p>タグひとつずつにv-ifを設定しても実現できますが、数が多くなると現実的ではありません。その場合はtrue組とfalse組を タグで囲って、v-ifを付与する方法が考えられるでしょう。

リスト8.7:

```
1: <div v-if="isEditRef">
2:   <p>1</p>
3:   <p>2</p>
4: </div>
5: <div v-else>
6:   <p>3</p>
7:   <p>4</p>
8: </div>
```

　これはこれで正解ですが、もしこの タグがどうしても不要だった場合はどうしましょう。

その場合は非表示ラッパーとして提供されている<template>タグを使います。これは タグを<template>タグに置き換えるだけで済みます。そして、非表示ラッパーと呼ばれるだけあって、HTMLでは<template>タグは描画されません。

リスト8.8:

```
1: <template v-if="isEditRef">
2:   <p>1</p>
3:   <p>2</p>
4: </template>
5: <template v-else>
6:   <p>3</p>
7:   <p>4</p>
8: </template>
```

ちなみにこの<template>はv-ifやv-forでは使えますが、v-showでは使うことができません。

8.6　ローカルストレージの値を変更する

途中が長くなりましたが、編集ボタンを押したら入力欄に編集対象のTODOを表示させるところまでできました。次は変更したTODOをローカルストレージのデータに反映させましょう。

変更ボタンが押されたら、入力欄のTODOを取得し、配列todoListRefに反映させます。しかしここでは編集対象の情報を持っていないため、どのTODOかわかりません。そのためshowTodoの処理でTODOのIDを事前に保管し、そのIDを使って対象の情報を編集します。

リスト8.9: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: const isEditRef = ref(false); // 移動
 3: let editId = -1; // 追加
 4:
 5: const showTodo = (id) => {
 6:       const  todo  =  todoListRef.value.find((todo)  =>
todo.id === id);
 7:   todoRef.value = todo.task;
 8:   isEditRef.value = true;
 9:   editId = id; // 追加
10: };
11:
12: // 変更
13: const editTodo = () => {
14:   // 編集対象となるTODOを取得
15:   const todo = todoListRef.value.find(
16:     (todo) => todo.id === editId
17:   );
18:
```

```
19:   // TODOリストから編集対象のインデックスを取得
20:   const idx = todoListRef.value.findIndex(
21:     (todo) => todo.id === editId
22:   );
23:
24:   // taskを編集後のTODOで置き換え
25:   todo.task = todoRef.value;
26:
27:   // splice関数でインデックスを元に対象オブジェクトを置き換え
28:   todoListRef.value.splice(idx, 1, todo);
29:
30:   // ローカルストレージに保存
31:                          localStorage.todoList         =
JSON.stringify(todoListRef.value);
32:   isEditRef.value = false; // 編集モードを解除
33:   editIndex = -1; // IDを初期値に戻す
34:   todoRef.value = '';
35: };
```

8.7　TODOを削除する

　TODOの削除は編集と同じようにTODOのIDを利用して配列から削除し、ローカルストレージに残った配列を登録します。

　では、同じように削除ボタンに@clickを付与して実装していきましょう。念のため、削除する前に確認メッセージを表示させることにします。

リスト8.10: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: const deleteTodo = (id) => {
  3:       const  todo  =  todoListRef.value.find((todo)  =>
todo.id === id);
  4:     const  idx  =  todoListRef.value.findIndex((todo)  =>
todo.id === id);
 5:
 6:   const delMsg = '「' + todo.task + '」を削除しますか？';
 7:   if (!confirm(delMsg)) return;
 8:
 9:   todoListRef.value.splice(idx, 1);
10:                          localStorage.todoList        =
JSON.stringify(todoListRef.value);
11: };
12:
13: // template
14: <button class="btn pink" @click="deleteTodo(todo.id)">
削</button>
```

　これでTODOアプリの登録、編集、削除という一通りの操作ができるようになりました。

第9章　ロジックの分離

Vue3からロジックの共通化や機能の関心ごとに着目して、ファイルを分離することができるようになりました。この仕組みを使って、今までのロジックを分離してみましょう。

◆ここで学べること
・ロジックの分離、再利用

9.1　ロジックを分離する

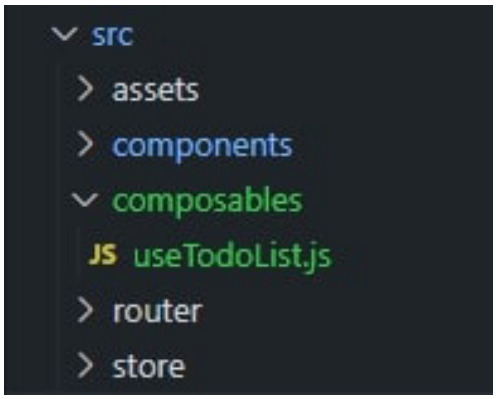
　これまでのソースで、TODOリストからIDに該当する要素やインデックスを取得するロジックが複数回出てきました。具体的には以下のソースです。

リスト9.1: my-vite-todo/src/components/MainTodo.vue

```
 1: // TODOリスト（todoListRef）からIDに一致するTODOを取得
 2: const todo = todoListRef.value.find((todo) => todo.id
=== id);
 3:
 4: // TODOリスト（todoListRef）からIDに一致するリストのインデックスを取得
 5: const idx = todoListRef.value.findIndex((todo) =>
todo.id === id);
```

　上記のTODOリストに関するロジックを分離して、再利用できる形にしていきましょう。
　まずは分離したロジックの置き場所を決めましょう。この辺は自由に決めてもらってもいいのですが、本書ではsrcフォルダー直下に「composables」フォルダーを作成し、そこを置き場所にします。その中に「useTodoList.js」ファイルも作成してください。

図9.1:



「useTodoList.js」ファイルも特に決まりはありませんが、分離したロジックとわかるようにファイル名の先頭に「use」を付けるのが慣例となっています。そして、このファイルにはJavaScriptのロジックしか書かないので、拡張子は「.js」としてください。

useTodoList.jsに共通化したいロジックを以下のようにそのままコピーしてください。

リスト9.2: my-vite-todo/src/composables/useTodoList.js

```
 1: // 外部から使えるようにexportする
 2: export const useTodoList = () => {
  3:       const  todo  =  todoListRef.value.find((todo)  =>
todo.id === id);
  4:     const  idx  =  todoListRef.value.findIndex((todo)  =>
todo.id === id);
 5:
 6:   // returnすることでtodoとidxを外部から使うことができる
 7:   return { todo, idx };
 8: };
```

簡単にですが、これでロジックの分離ができました。

分離したロジックは、元の場所であるMainTodo.vueとは何の繋がりもありません。そのためtodoListRefは、いきなり出てきた状態になって

しまっているので、このまま使うとエラーになってしまいます。

　そこで、todoListRefを取得する処理を追記してあげます。この処理もMainTodo.vueにあったものをそのままコピーして持ってきます。

　次にidですが、これは引数でもらうしかありません。
export const useTodoList = () => {を
export const useTodoList = (id) => {としてください。

リスト9.3: my-vite-todo/src/composables/useTodoList.js

```
 1: import { ref } from 'vue';
 2:
 3: export const useTodoList = (id) => {
 4:   // ローカルストレージにtodoListが存在していればparseし、
 5:   // なければundifinedになるため空配列をセットする。
 6:   const ls = localStorage.todoList;
 7:   const todoListRef = ref([]);
 8:   todoListRef.value = ls ? JSON.parse(ls) : [];
 9:
10:       const  todo  =  todoListRef.value.find((todo)  =>
todo.id === id);
11:     const  idx  =  todoListRef.value.findIndex((todo)  =>
todo.id === id);
12:
13:   return { todo, idx };
14: };
```

9.2　分離ロジックの使い方

一旦、今のままで共通化したロジックを使ってみましょう。使い方は簡単で、importして呼び出すだけです。todoとidxが揃っているdeleteTodoから修正して確認してみます。

リスト9.4: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // 追加
     3:     import     {     useTodoList     }     from
'/src/composables/useTodoList.js';
 4:
 5: const deleteTodo = (id) => {
 6:   // 削除
  7:    //  const  todo  =  todoListRef.value.find((todo)  =>
todo.id === id);
 8:   // const idx = todoListRef.value.findIndex((todo) =>
todo.id === id);
 9:
10:   const { todo, idx } = useTodoList(id); // 追加
11:   const delMsg = '「' + todo.task + '」を削除しますか？';
12:   if (!confirm(delMsg)) return;
13:
14:   todoListRef.value.splice(idx, 1);
15:                          localStorage.todoList        =
JSON.stringify(todoListRef.value);
16: };
```

以前と変わらず削除処理ができていたら成功です。

分割代入

const { todo, idx } = useTodoList(id); は分割代入といって、JavaScriptの機能です。useTodoList.jsでreturnしたtodoとidxを受け取り側で同じように記述することで、そのまま受け取れます。

9.3　分離ロジックのリファクタリング

このままdeleteTodoと同じようにshowTodoやeditTodoも置き換えてもいいですが、もう１歩進んでみましょう。useTodoListでtodoListRef（TODOリスト）をローカルストレージから取り出しているので、それをそのままreturnで返してあげます。するとdeleteTodoのように、useTodoListを呼び出すだけでTODOリストを取得することができます。

このように考えていくと、TODOリストをひとつの責務と考え、それに関する処理を集めることでロジックの見通しもよくなり、いろいろな所でも使えそうです。これでTODOリストに関することが各コンポーネントに散らばることなく、ひとつのファイルにまとめることができました。

それではuseTodoListにTODOリストに関する処理を追加していきましょう。一旦中身を空にして、todoListRefを取得して返すだけにしてみましょう。

リスト9.5: my-vite-todo/src/composables/useTodoList.js

```
 1: import { ref } from 'vue';
 2:
 3: export const useTodoList = () => {
 4:   // ローカルストレージにtodoListが存在していればparseし、
 5:   // なければundifinedになるため空配列をセットする。
 6:   const ls = localStorage.todoList;
 7:   const todoListRef = ref([]);
 8:   todoListRef.value = ls ? JSON.parse(ls) : [];
 9:
10:   return { todoListRef };
11: };
```

次に、TODOリストに追加する処理をコピーして持ってきます。入力したTODOは引数としてもらうようにします。混乱しないように、関数名をaddTodoからaddにしています。

リスト9.6: my-vite-todo/src/composables/useTodoList.js

```
 1: import { ref } from 'vue';
 2:
 3: export const useTodoList = () => {
 4:   // ローカルストレージにtodoListが存在していればparseし、
 5:   // なければundifinedになるため空配列をセットする。
 6:   const ls = localStorage.todoList;
 7:   const todoListRef = ref([]);
 8:   todoListRef.value = ls ? JSON.parse(ls) : [];
 9:
10:   // 追加処理
11:   const add = (task) => {
12:     const id = new Date().getTime();
13:     todoListRef.value.push({ id: id, task: task });
14:                             localStorage.todoList      =
JSON.stringify(todoListRef.value);
15:   };
16:
17:   return { todoListRef };
18: };
```

元々のaddTodoにあったtodoRef.value = '';は画面の入力欄に対する操作で、ここの責務とは関係がないため、削除します。同じように編集、削除処理も少し改変して持ってきますが、その前にtodoとidxが何回か出てくるので、関数化しておきましょう。名前も使いやすいように変更します。

リスト9.7: my-vite-todo/src/composables/useTodoList.js

```
 1: // TODOリストからIDを元にTODO情報を取得
 2: const findById = (id) => {
 3:    return todoListRef.value.find((todo) => todo.id ===
id);
 4: };
 5:
 6: // TODOリストからIDを元にそのインデックスを取得
 7: const findIndexById = (id) => {
 8:    return todoListRef.value.findIndex((todo) => todo.id
=== id);
 9: };
```

編集ボタンが押されたときの表示処理です。

リスト9.8: my-vite-todo/src/composables/useTodoList.js

```
 1: const editId = ref(-1);  // リアクティブにします
 2: const show = (id) => {
 3:   const todo = findById(id);
 4:   editId.value = id;
 5:   return todo.task;  // 画面処理させるために返します
 6: };
```

変更ボタンが押されたときの処理です。

リスト9.9: my-vite-todo/src/composables/useTodoList.js

```
 1: const edit = (task) => {
 2:   const todo = findById(editId.value);
 3:   const idx = findIndexById(editId.value);
 4:   todo.task = task;
 5:   todoListRef.value.splice(idx, 1, todo);
       6:                    localStorage.todoList       =
JSON.stringify(todoListRef.value);
```

```
  7:   editId.value = -1;
  8: };
```

削除ボタンが押されたときの処理です。

リスト9.10: my-vite-todo/src/composables/useTodoList.js

```
  1: const del = (id) => {
  2:   const todo = findById(id);
  3:   const delMsg = '「' + todo.task + '」を削除しますか？';
  4:   if (!confirm(delMsg)) return;
  5:
  6:   const idx = findIndexById(id);
  7:   todoListRef.value.splice(idx, 1);
       8:                     localStorage.todoList       =
JSON.stringify(todoListRef.value);
  9: };
```

チェックボックスが押されたときの処理です。

リスト9.11: my-vite-todo/src/composables/useTodoList.js

```
  1: const check = (id) => {
  2:   const todo = findById(id);
  3:   const idx = findIndexById(id);
  4:   todo.checked = !todo.checked;
  5:   todoListRef.value.splice(idx, 1, todo);
       6:                     localStorage.todoList       =
JSON.stringify(todoListRef.value);
  7: };
```

最後のreturn文です。

リスト9.12: my-vite-todo/src/composables/useTodoList.js

```
1: return { todoListRef, add, show, edit, del, check };
```

9.4　分離ロジックで置き換える

　先ほど作ったロジックを元の場所であるMainTodo.vueで使ってみましょう。一気にscript部分を載せます。

リスト9.13: my-vite-todo/src/components/MainTodo.vue

```
 1: <script setup>
 2: import { ref } from 'vue';
     3:      import      {      useTodoList      }      from
'/src/composables/useTodoList.js';
 4:
 5: const todoRef = ref('');
 6: const isEditRef = ref(false);
 7: const { todoListRef, add, show, edit, del, check } =
useTodoList();
 8:
 9: const addTodo = () => {
10:   add(todoRef.value);
11:   todoRef.value = '';
12: };
13:
14: const showTodo = (id) => {
15:   todoRef.value = show(id);
16:   isEditRef.value = true;
17: };
18:
19: const editTodo = () => {
20:   edit(todoRef.value);
21:   isEditRef.value = false;
22:   todoRef.value = '';
23: };
24:
25: const deleteTodo = (id) => {
26:   del(id);
```

```
27: };
28:
29: const changeCheck = (id) => {
30:   check(id);
31: };
32: </script>
```

ずいぶんとスッキリとした感じになったのではないでしょうか。このようにうまくコンポーネントやロジックを分けることにより、ソースの見通しがよくなります。

---

deleteTodoは他の処理がなければ、以下のように直接onClickに書いてもOKです。　→　@click="del(todo.id)"

第10章　TODOのチェック

　TODOアプリについては、ほぼ完成に近づいてきました。残りはTODOを完了したときにチェックして、完了済にすることです。
　仕様としては、チェックされたらTODOの色を変えつつ取り消し線を引くことにしましょう。

　◆ここで学べること
　・データバインディング（v-bind）

10.1　チェック情報を追加する

　保存しているTODOの情報にチェックボックスのチェック有無を入れようと思いますので、一旦全てのTODOを削除してください。
　そして、TODOを登録するロジックにチェック情報を追加してください。登録時なのでチェックは無し（false）です。

リスト10.1: my-vite-todo/src/composables/useTodoList.js

```
 1: const add = (task) => {
 2:   ...
 3:   // 「checked: false」を追加
  4:       todoListRef.value.push({  id:  id,  task:  task,
checked: false });
 5:   ...
 6: };
```

　これでまた何個かTODOを登録しておいてください。

10.2　チェックを保存する

　チェックボックスのチェックの有無を検知し、その状態を各TODOに保存する必要があります。そして、保存するには再びTODOのIDが必要になりそうです。ということで、チェックの有無を検知するのにonChangeを使い、引数にIDを渡します。

リスト10.2: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // check を追加
 3: const { todoListRef, add, show, edit, del, check } =
useTodoList();
 4:
 5: const chengeCheck = (id) => {
 6:   check(id);
 7: };
 8:
 9: // template
10: ...
11: <input
12:   type="checkbox"
13:   class="check"
14:   @change="changeCheck(todo.id)"
15: />
16: <label>{{ todo.task }}</label>
17: ...
```

　IDが手に入れば、編集時と同じようにするだけです。

リスト10.3: my-vite-todo/src/composables/useTodoList.js

```
 1: const check = (id) => {
 2:   const todo = findById(id);
 3:   const idx = findIndexById(id);
 4:   todo.checked = !todo.checked;  // 「true/false」を反転させる
 5:   todoListRef.value.splice(idx, 1, todo);
 6:                     localStorage.todoList       = JSON.stringify(todoListRef.value);
 7: };
 8:
 9: // check を追加
10: return { todoListRef, add, show, edit, del, check };
```

10.3　データバインディング（v-bind）

　チェックの有無はローカルストレージに登録されましたが、画面を更新するとチェックが外れている状態になります。これは、チェック情報を画面のほうに反映していないためです。
　チェックボックスはchecked属性を使って、チェックを制御することができます。チェックボックスに:checked="todo.checked"を追加してみてください。

リスト10.4: my-vite-todo/src/components/MainTodo.vue

```
1: <input
2:   type="checkbox"
3:   class="check"
4:   @change="chengeCheck(todo.id)"
5:   :checked="todo.checked"  // 追加
6: />
```

　checkedの前には:（コロン）が必要です。このコロンはVueのv-bindディレクティブの省略形ですので、:checked=""とv-bind:checked=""は同じ意味になります。この例ではtodo.checkedの値が設定されます。

v-bind

v-bindに設定する内容にはscriptの変数や関数、JavaScriptの構文を使うことができます。

TODOの取り消し線ですが、これはCSSのクラスを切り替えることで実現します。まずは取り消し線のCSSを設定します。ついでに背景色と文字色も少し変えましょう。

リスト10.5: my-vite-todo/src/components/MainTodo.vue

```
.fin {
  text-decoration: line-through;
  background-color: #ddd;
  color: #777;
}
```

次に、これをチェックボックスを囲っている に設定するのですが、todo.checkedの値がtrueのときがTODOの完了なので、そのときにfinクラスが当たるようにします。

リスト10.6: my-vite-todo/src/components/MainTodo.vue

```
// template
// :classを追加
<div class="todo" :class="{ fin: todo.checked }">
  <input
    type="checkbox"
    class="check"
    @change="chengeCheck(todo.id)"
    :checked="todo.checked"
  />
```

```
10:   <label>{{ todo.task }}</label>
11: </div>
```

:classのようにclassにバインドし、オブジェクトを渡すことでクラスを動的に切り替えることができます。finがCSSのクラス名で右辺のtodo.checkedの真偽値によってそのクラスの適用が決まります。

クラスとスタイルのバインディング

Vueはv-bindがclassとstyleと一緒に使われるとき、特別な拡張機能を提供します。文字列だけではなく、式はオブジェクトまたは配列を返すことができます。

- Vue公式ページより -

:classとclass

:classは通常のclassと同時に使うこともできます。また、カンマ区切りで複数の動的クラスも指定できます。<div class="todo" :class="{ fin: todo.checked, fin2: true }">

お疲れさまでした！
以上で第 1 章で見たTODOのサンプルアプリが完成しました！

次の章からはVueのその他の機能について見ていくことにしましょう。

第11章　その他の重要機能

　これまで見てきたVueの機能はほんの一部です。この章ではその他の機能を見ていきます。本書ではなかなか出番がなかったり、説明を飛ばしていたりした部分ですが、実際は使用頻度が高く重要な機能となります。

◆ここで学べること
・リアクティブ（ref、reactive）
・算出プロパティー（computed）
・スロットコンテンツ（slot）
・コンポーネント間の親から子へのデータ渡し（props）
・コンポーネント間の子から親への通知（emit）

11.1　リアクティブ変数

　リアクティブとは、ある変数の変更を検知可能な状態になっていることをいいます。第 6 章のマスタッシュ構文で示したように、todoRef変数をscript内で書き変えたとき、即座にHTMLに反映されるようなことです。

　これまではref関数を使ってリアクティブを作り出していましたが、同様の機能としてreactive関数というものが存在します。これらの使い分けですが、今のところは
　・ref　　・・・　プリミティブな値（オブジェクト以外）
　・reactive　・・・　オブジェクト
　でいいと思います。……歯切れの悪い言い方なのは、まだまだこの両者の使い分けが定まっていないためです。今後、知見が集まり定まってくると思いますが、今はこれで問題ありません。
＜使い方＞

リスト11.1:

```
 1: import { reactive, ref } from 'vue';
 2:
 3: // ref関数
 4: const count = ref(0);
 5: count.value++;
 6:
 7: // reactive関数
 8: const state = reactive({
 9:   count: 0,
10: });
11: state.count++;
```

ここで、reactiveに関しての注意点があります。reactive変数を分割代入をすることでリアクティブが失われてしまいます。公式ページの例を見てみます。

リスト11.2:

```
 1: import { reactive } from 'vue'
 2:
 3: const book = reactive({
 4:   author: 'Vue Team',
 5:   year: '2020',
 6:   title: 'Vue 3 Guide',
 7:    description: 'You are reading this book right now
;)',
 8:   price: 'free'
 9: })
10:
11: let { author, title } = book
```

この場合、authorとtitleはリアクティブでなくなるので、注意する必要があります。もし、分割代入で取得したい場合はtoRefs関数を使うと、リアクティブのまま取得することができます。具体的には以下のようにします。

リスト11.3:

```
1: let { author, title } = toRefs(book)
```

toRefs関数で取得した後の変数はrefとなりますので、script内で値を参照する場合は変数.valueになることにも注意してください。

11.2　算出プロパティー

VueではHTML内にマスタッシュ構文を使えば、JavaScriptの式を記述することができます。たとえば{{ 1 + 1 }}をHTML内に書くと、画面上は「2」が表示されます。{{ todoRef + 'を入力' }}のように変数も使うことができます。

このような簡単な式なら問題ありませんが、複雑な処理が必要になってくると使い勝手が悪くなっていきます。また、同じ処理を使いたい場合やそれをメンテナンスする場合にも、同様に使いづらくなっていきます。

そういう場合にVueには「算出プロパティ」（computed）という機能が用意されています。算出プロパティーは、関数で処理した結果を変数のように使うことができます。

では、TODOアプリを使って算出プロパティーを見てみましょう。TODOの完了と未完了の数を見られるように、機能を追加します。
まずは完了と未完了のレイアウトを作ります。templateとstyleの一番下に以下を記述します。

リスト11.4: my-vite-todo/src/components/MainTodo.vue

```
1: // template
2: <div class="finCount">
3:   <span> 完了：、</span>
4:   <span> 未完了：</span>
5: </div>
6:
7: // style
8: .finCount {
```

```
  9:   margin-top: 8px;
 10:   font-size: 0.8em;
 11: }
```

完了したTODOというのはcheckedがtrueになっていますので、filter関数を使って数えていきます。filter関数の使い方は前述のfind関数と同じです。

find関数は条件に一致した最初のものが返りますが、filter関数は条件に一致したものすべてが配列として返ってきます。

TODO一覧から完了したTODOを数える処理を算出プロパティーを使って書いてみます。useTodoList.jsに以下の処理を追加してください。書き方はこれまで書いてきたような関数をcomputed()で丸ごと囲むような形になります。

リスト11.5: my-vite-todo/src/composables/useTodoList.js

```
 1: import { computed, ref } from 'vue';  // computedを追加
 2:
 3: // 追加
 4: const countFin = computed(() => {
 5:   // todo.checkedは「true/false」が入っているため、trueのtodoが返る
 6:     const finArr = todoListRef.value.filter((todo) => todo.checked);
 7:   return finArr.length;
 8: });
 9:
10: // countFinを追加
11: return { todoListRef, add, show, edit, del, check, countFin };
```

呼び出す側のMainTodo.vueは以下です。

リスト11.6: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // countFinを追加
 3: const {
 4:   todoListRef,
 5:   add,
 6:   show,
 7:   edit,
 8:   del,
 9:   check,
10:   countFin
11: } = useTodoList();
12:
13: // template
14: <div class="finCount">
15:   <span> 完了：{{ countFin }}、</span>  <!-- countFinを追
加 -->
16:   <span> 未完了：</span>
17: </div>
```

これで画面のチェックボックスを操作してみてください。完了の数字がチェックに合わせて増減していれば、うまく機能しています。

次に未完了の方ですが、その前に先ほどの算出プロパティーと関数の違いを見ていきましょう。MainTodo.vueにTODOを数える関数を書いてみましょう。内容はcountFinと同じです。

リスト11.7: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: const countFinMethod = () => {
 3:     const  finArr  =  todoListRef.value.filter((todo)  =>
todo.checked);
 4:   return finArr.length;
 5: };
 6:
```

```
 7: // template
 8: <div class="finCount">
 9:   <span> 完了：{{ countFin }}、</span>
10:   <span> 未完了：{{ countFinMethod() }}</span>  <!-- 関数なので() が付きます -->
11: </div>
```

先ほどと同じようにチェックボックスを操作してみてください。完了も未完了も同じように数字が増減するはずです。

算出プロパティーと関数の結果だけ見ればまったく同じになりますが、算出プロパティーはリアクティブな依存関係に基づいてキャッシュされるという違いがあります。
算出プロパティーはリアクティブな変数が変更された場合にのみ再評価され、関数は再レンダリングが起こるたびに処理を実行します。

これらを確認するため、ログを算出プロパティーと関数に仕込んでください。

リスト11.8: my-vite-todo/src/composables/useTodoList.js

```
 1: const countFin = computed(() => {
 2:   console.log('computed');  // 追加
 3:     const finArr = todoListRef.value.filter((todo) => todo.checked);
 4:   return finArr.length;
 5: });
```

リスト11.9: my-vite-todo/src/components/MainTodo.vue

```
 1: const countFinMethod = () => {
 2:   console.log('method');  // 追加
 3:     const finArr = todoListRef.value.filter((todo) => todo.checked);
```

```
    return finArr.length;
};
```

チェックボックスにチェックを付けても外しても'computed'と'method'の両方がコンソールに表示されると思います。

入力欄に何か入力したり、編集ボタンを押してみてください。今度は'computed'は表示されず、'method'のみ何回も表示されたかと思います。

これは先ほどの説明した「リアクティブな変数が変更された場合にのみ再評価」が効いているからです。入力欄に何か入力したり、編集ボタンを押してもtodoListRefというリアクティブな変数は何も影響されないため、computedはキャッシュされた値を返すだけとなっています。

では、そろそろ未完了のほうも実装していきます。これは完了と同じように関数にしてもいいのですが、ここではマスタッシュ構文の中に式を書いてみましょう。

リスト11.10: my-vite-todo/src/components/MainTodo.vue

```
// template
<div class="finCount">
  <span> 完了：{{ countFin }}、</span>
   <span> 未完了：{{ todoListRef.length - countFin }}
</span>
</div>
```

11.3　コンポーネント間のやりとり（親から子へ）

　TODOアプリで使っているボタンは追加、編集、削除、変更といろいろありますが、<button>タグにCSSで変化を付けただけでほとんど同じ作りになっています。

　上記以外のボタンが増えても同じページで使う分には問題ありませんが、もし他のページで同じようなボタンが必要となった場合はどうなるのでしょうか。

　ボタンのCSSをコピーしてその他のページに持っていかなければなりません。そのCSSをグローバルな場所に持っていっても実現できますが、数が多くなってくると複雑になってくるのは目に見えていますので、ここはボタンをコンポーネント化してみましょう。

　コンポーネント化するにしても、そのプロジェクトの規模や作り手の判断でいろいろな方法が考えられます。本書ではその中のひとつの方法だということは覚えておいてください。唯一の方法ではありません。

　追加、編集、削除、変更のボタンの方を並べて見てみましょう。

リスト11.11:

```
   1:  <button  class="btn  green"  @click="editTodo"  v-
if="isEditRef">変更</button>
  2:  <button  class="btn"  @click="addTodo"  v-else> 追 加
</button>
 3: <button class="btn green" @click="showTodo(todo.id)">編
</button>
 4: <button class="btn pink" @click="deleteTodo(todo.id)">
削</button>
```

classや@clickが若干違うだけで、CSSを含めてほとんど同じです。では、それらのボタンの共通となる部分を抜き出して、基本となるコンポーネントを作ってみましょう。

componentsフォルダーにBaseButton.vueを作り、基本的なボタンを実装します。@clickは空だとエラーになるので、適当な関数を入れています。

リスト11.12: my-vite-todo/src/components/BaseButton.vue

```
 1: <script setup>
 2: const aaa = () => {};
 3: </script>
 4:
 5: <template>
 6:   <button class="btn" @click="aaa">XXX</button>
 7: </template>
 8:
 9: <style scoped>
10: .btn {
11:   padding: 8px;
12:   background-color: #03a9f4;
13:   border-radius: 6px;
14:   color: #fff;
15:   text-align: center;
16:   font-size: 14px;
17: }
18: </style>
```

この状態でMainTodoから呼び出してみましょう。

リスト11.13: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // 追加
        3:         import         BaseButton         from
```

```
'/src/components/BaseButton.vue';
 4:
 5: // template
 6: <template>
 7:   <BaseButton />  <!-- 追加 -->
```

以下のような状態になるかと思います。

図11.1:

次は「XXX」の部分を追加、編、削、変更にしますが、これは呼び出し側から指定してもらいます。今回はスロットコンテンツと呼ばれる<slot>要素を使います。

呼び出し側の<BaseButton />を以下のようにしてください。

リスト11.14: my-vite-todo/src/components/MainTodo.vue

```
1: // template
2: <BaseButton>追加</BaseButton>  // 変更
```

続いて、BaseButton.vueの方も<slot>要素を追加します。

リスト11.15: my-vite-todo/src/components/BaseButton.vue

```
1: <template>
2:   <!-- 「XXX」を<slot />に変更 -->
3:   <button class="btn" @click="aaa"><slot /></button>
4: </template>
```

画面のボタンが「XXX」から「追加」に変わったのではないでしょうか。「追加」の部分を変えることにより、いろいろなボタンが作成できます。このように<slot>を使うと、呼び出し側のコンポーネントタグで挟まれたものがそのコンポーネントの<slot>に置き換わります。

スロット内

スロットには、HTMLを含む任意のテンプレートコードを含めることもできます。

これでボタン名の変更はできましたので、次はボタンの色について見ていきます。これも、どの色にするかは呼び出し側から指定してもらわなければなりません。

MainTodoコンポーネントを「親」、BaseButtonコンポーネントを「子」とした場合、親から子へはプロパティーを通して情報を渡すことができます。

「編」ボタンは緑色なので、BaseButton.vue（子）に緑色のCSSを用意して、それを呼び出し側（親）から指定してもらうようにします。

MainTodo.vueに編集ボタンを追加します。このとき、緑色（green）のCSSをcolorというプロパティー名で指定します。

親から子へボタンの色をcolorというプロパティー（props）を通して子に伝えます。

リスト11.16: my-vite-todo/src/components/MainTodo.vue

```
 1: // template
 2: <BaseButton>追加</BaseButton>
 3: <BaseButton color="green">編</BaseButton>  <!-- 追加 --
>
```

BaseButton.vueに緑色のCSSを追加します。そして、親からのプロパティーを受け取るには、definePropsというものを使います。propsという名称は何でもよいですが、慣習的にpropsがよく使われます。そして、受け取ったpropsからcolorを取り出してCSSに当てます。

リスト11.17: my-vite-todo/src/components/BaseButton.vue

```
 1: <script setup>
 2: // 親からのプロパティー：colorを取得
 3: const props = defineProps({ color: String });
 4: const aaa = () => {};
 5: </script>
 6:
 7: <template>
 8:   <!-- :classを追加 -->
   9:          <button   class="btn"   :class="props.color"
@click="aaa">
10:     <slot />
11:   </button>
12: </template>
13:
14: <style scoped>
15: // 追加
16: .green {
17:   background-color: #00c853;
18: }
19: </style>
```

definePropsは、()の中にプロパティー名（color）とその型（String）を書きます。「プロパティー：color」は文字列で受け取りますよ、という意味になります。

プロパティーの型はStringの他にもNumber、Boolean、Array、Object、Function、Promiseとありますので、適切な型を割り当ててください。型と値が合っていなければ、コンソールに警告が表示されます。

複数の型

複数の型を使いたい場合は配列で指定します。
例）color: [String, Number]

青色もbtnクラスから取り出して独立させ、ピンク色もBaseButton.vueに追加してください。

リスト11.18: my-vite-todo/src/components/BaseButton.vue

```
<style scoped>
.btn {
  padding: 8px;
  background-color: #03a9f4;  // 削除
  border-radius: 6px;
  color: #fff;
  text-align: center;
  font-size: 14px;
}

// 追加
.blue {
  background-color: #03a9f4;
}

.green {
  background-color: #00c853;
}

// 追加
.pink {
  background-color: #ff4081;
}
</style>
```

そして、残りの削除ボタンと変更ボタンも作ります。

リスト11.19: my-vite-todo/src/components/MainTodo.vue

```
1: // template
2: <BaseButton color="blue">追加</BaseButton>
3: <BaseButton color="green">編</BaseButton>
4: <BaseButton color="pink">削</BaseButton>
5: <BaseButton color="green">変更</BaseButton>
```

図11.2:

11.4　コンポーネント間のやりとり（子から親へ）

　基本ボタンコンポーネントでの残りは、@clickになりました。それぞれのボタンが押されたら、それぞれに合った処理を行わないといけません。propsでそれぞれのボタンを判別する値を受け取り、@clickの処理を分岐させてもいいのですが、それ以外のボタンが増えるたびに処理も追加していくのは、現実的ではありません。

　こういう場合は、ボタンが押されたことを親に知らせます。そして、親側でボタンが押されたときの処理を行うようにします。そうすれば、この基本ボタンコンポーネントも汎用的に使うことができます。
　子から親に通知する方法にemitというものがあり、defineEmitsで定義します。

　ボタンが押されたときの処理@clickで親に通知してみます。
・① defineEmitsを使って親に伝えるための名称を定義します。
　→　'on-click'
・② emit関数を使って親に通知します。そのときにdefineEmitsで定義した名称を指定します。

リスト11.20: my-vite-todo/src/components/BaseButton.vue

```
<script setup>
const props = defineProps({ color: String });
const emit = defineEmits(['on-click']);  // ①
const onClick = () => {
  emit('on-click');  // ②
};
</script>

<template>
```

```
10:           <button    class="btn"    :class="props.color"
@click="onClick">
11:     <slot />
12:   </button>
13: </template>
```

・③ 親側では子コンポーネントにemitで指定した'on-click'に@を付けて受け取ります。あとは@clickのときと同じように使います。

リスト11.21: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: // 追加
 3: const test= () => {
 4:   console.log('test');
 5: };
 6:
 7: // template
 8: <!-- お試し用に追加ボタンのみ -->
  9:   <BaseButton   color="blue"   @on-click="test">  追  加
</BaseButton>
```

emitは引数も渡すことができます。emitに第 2 引数を与えるだけです。

リスト11.22: my-vite-todo/src/componeents/BaseButton.vue

```
 1: // script
 2: const onClick = (str) => {
 3:   emit('on-click', str);
 4: };
 5:
 6: // template
   7:          <button    class="btn"    :class="props.color"
@click="onClick('あああ')">
```

リスト11.23: my-vite-todo/src/components/MainTodo.vue

```
// script
const test = (str) => {
  console.log('test', str);
};
```

これでコンソールに「test あああ」と出ていれば、うまく引数が渡っています。確認後は元の引数なしの状態に戻しておいてください。

それでは、以上を踏まえてHome.vueのボタンをすべて置き換えていきましょう。その前にMainTodo.vueから確認用のtest関数と<template>直下のBaseButton×4は削除してください。

リスト11.24: my-vite-todo/src/components/MainTodo.vue

```
// 変更前
  <button  class="btn  green"  @click="editTodo"  v-
if="isEditRef">
  変更
</button>
 <button  class="btn"  @click="addTodo"  v-else> 追 加
</button>
<button class="btn green" @click="showTodo(todo.id)">
  編
</button>
<button class="btn pink" @click="deleteTodo(todo.id)">
  削
</button>

// 変更後
 <BaseButton  color="green"  @on-click="editTodo"  v-
if="isEditRef">
  変更
</BaseButton>
<BaseButton color="blue" @on-click="addTodo" v-else>
  追加
```

```
19: </BaseButton>
20:          <BaseButton          color="green"          @on-
click="showTodo(todo.id)">
21:   編
22: </BaseButton>
23:          <BaseButton          color="pink"          @on-
click="deleteTodo(todo.id)">
24:   削
25: </BaseButton>
```

変更前の<button>を変更後の<BaseButton>にしてください。TODOを追加、変更、削除をして、以前と同じ動きになっていれば成功です。

ボタンはこれで問題ないように思いますが、もう少し規模が大きくなっていき、ボタンも至るところで使われるようになると、誤差が出てくるかもしれません。たとえば、追加ボタンの色は青ですが、間違って緑にしてしまったり、「追加」を「追　加」と間にスペースが入ったりしたら統一感がなくなります。

これらを防ぐため、もう一段階コンポーネント化し、追加、編集、削除、変更専用のボタンを作ることもあります。追加ボタンを例にします。
componentsフォルダーにButtonAdd.vueを作り、以下のようにしてください。

リスト11.25: my-vite-todo/src/components/ButtonAdd.vue

```
 1: <script setup>
         2:          import          BaseButton          from
'/src/components/BaseButton.vue';
 3: const emit = defineEmits(['add-click']);
 4: </script>
 5:
 6: <template>
  7:       <BaseButton  color="blue"  @on-click="emit('add-
```

```
click')">
 8:     追加
 9:   </BaseButton>
10: </template>
```

onClickの処理がBaseButton → ButtonAdd → MainTodoと、バケツリレーになってしまうのですが、これで誰が追加ボタンを使っても同じ見た目にすることができます。emitもこれくらいの記述なら、@on-clickに"emit('add-click')"と直接書いてもいいでしょう。

MainTodo.vueのほうも書き変えます。

リスト11.26: my-vite-todo/src/components/MainTodo.vue

```
1: // script
2: import ButtonAdd from '/src/components/ButtonAdd.vue';
3:
4: // template
5: <ButtonAdd @add-click="addTodo" v-else />
```

このように専用のボタンを作ると使い勝手はよくなりますが、ファイルが増えたりそれ以外の文言のボタンを作るときは若干手数が増えます。

この辺りはプロジェクトの方針に従ってうまく調整しながら作ってみてください。

第12章　ライフサイクル

　Vueインスタンスの発生から消滅までの状態をライフサイクルといいます。その時々の状態をフックで処理を行うことができます。たとえば、画面が表示される前に何らかの処理をしたり、コンポーネントが更新された直後に処理したい場合です。

　◆ここで学べること
　・ライフサイクルフック

12.1　ライフサイクルフック

ライフサイクルフックについては以下の通りです。
・onBeforeMount、onMounted
・onBeforeUpdate、onUpdated
・onBeforeUnmount、onUnmounted
・onErrorCaptured
・onRenderTracked、onRenderTriggered
・onActivated、onDeactivated
このように様々な種類がありますが、本書ではよく使うonMounted、onUpdated、onUnmountedについて見ていきます。before～は、それぞれのフックの前に呼ばれると思ってください。

12.2　onMounted

Vue3以前にはcreatedというライフサイクルフックがあったのですが、Vue3からはsetup内に直接書くことにより実現するようになりました。createdはonMountedより前に呼ばれます。もう少しいいますと、createdはVueインスタンスの発生からDOMの生成前まで、onMountedはDOMの生成後に呼ばれます。

TODOアプリを使って見てみましょう。MainTodo.vueの<script>の上のほうにonMounted、下のほうにconsole.log('setup');を記述します。これで画面を更新してみてください。

リスト12.1: my-vite-todo/src/components/MainTodo.vue

```
<script setup>
import { onMounted, ref } from 'vue';

// 上のほうに記述
onMounted(() => {
  console.log('onMounted');
});

// 下のほうに記述
console.log('setup');
</script>
```

コンソールにsetup、onMountedの順番でログが出力されたのではないでしょうか。これでソースでの順番は前後してもsetup、onMountedの順番で呼び出されることが確認できました。

次にDOMの生成前後を見てみましょう。説明用にTODOの入力欄にid属性を持たせてください。そして、setup、onMountedでgetElementByIdを使ってDOMを操作します。

リスト12.2: my-vite-todo/src/components/MainTodo.vue

```
 1: // script
 2: const todoRef = ref('abc');  // TODO入力欄に初期値「abc」を持たせる
 3:
 4: // setupでTODO入力欄を取得
 5: const inpSetup = document.getElementById('inp');
 6: console.log(inpSetup);
 7:
 8: // onMountedでTODO入力欄を取得
 9: onMounted(() => {
10:                          const        inpMmount        = document.getElementById('inp').value;
11:   console.log(inpMmount);
12: });
13:
14: // template
15: <input
16:   id="inp"  // 追加
17:   type="text"
18:   class="todo_input"
19:   v-model="todoRef"
20:   placeholder="＋ TODOを入力"
21: />
```

setupで値が取れないのは、まだDOMが生成されていないため、TODO入力欄が存在しないからです。「.value」を付けると、存在しないDOMに対してアクセスしようとしてエラーになります。

onMountedのほうは「abc」という値が取れたと思います。これはDOMが生成された後にアクセスしたからです。

　setupではデータベース（本書ではローカルストレージ）からデータを取ってきたり、画面を表示する前の処理を行います。onMountedは画面が表示された後、たとえばTODO一覧が表示された後に特定のTODOにチェックを付けたりしたい場合に使います。

12.3　onUpdated

　onUpdatedはコンポーネントが再描画されたときに呼び出されます。入力欄の値をonUpdatedで見てみることにしましょう。以下のように記述し、入力欄に値を入力してください。

リスト12.3: my-vite-todo/src/components/MainTodo.vue

```
1: onUpdated(() => {
2:   console.log('onUpdated: ', todoRef.value);
3: });
```

　半角英数字モードで入力した場合、1 文字入力するごとにコンソールに値が表示されると思います。全角の場合でもEnterキーで確定した瞬間にその値が表示されたと思います。

　また、追加や編集、削除ボタンを押しても入力欄の値が表示されます。画面上ではわからなくても、Vueの中ではコンポーネントが再描画されています。そしてそれをonUpdatedが検知し、コンソールに値を表示しています。

12.4　onUnmounted

　onUnmountedはコンポーネントが破棄されたときに呼び出されます。これは破棄対象となるコンポーネントに書く必要があるので、ButtonAdd.vueにonUnmountedを記述してください。

リスト12.4: my-vite-todo/src/components/ButtonAdd.vue

```
1: // script
2: // 追加
3: import { onUnmounted } from 'vue';
4:
5: onUnmounted(() => {
6:   console.log('onUnmounted');
7: });
```

　編集ボタンを押してみてください。コンソールに「Unmounted」と表示されます。
　編集ボタンを押すことで、v-ifによって追加ボタンが破棄されます。その破棄されるタイミングでUnmountedは呼び出されます。

　このようにVueインスタンスのライフサイクルに沿っていろいろなフックが存在し、その時々の場面に応じた処理を行うことができます。

第13章　Vue Router

　Vueは基本的にSPA（シングル・ページ・アプリケーション）ですが、リンクで他のページに画面を遷移することもできます（厳密には、遷移したように見せると言ったほうがいいかもしれませんが）。
　それを実現するには、Vueの公式ルータである「Vue Router」を使用します。

　♦ここで学べること
　・Vue Router
　・リアクティブの監視（watch）

13.1　インストール

　まずはVue Routerをインストールします。以下のコマンドでインストールを実施してください。Vue3の場合はバージョン4を使います。

```
# npmの場合
$ npm install vue-router@4

# yarnの場合
$ yarn add vue-router@4
```

13.2　Vue Routerを使うための準備をする

　Vue Routerを使うための設定ファイルを作っていきます。src配下に「router」フォルダー - 「index.js」ファイルを作ってください。内容は以下の通りです。これが基本の設定となります。

図13.1:

リスト13.1: my-vite-todo/src/router/index.js

```
 1: import { createRouter, createWebHistory } from 'vue-
router';
 2:
 3: const routes = [];
 4:
 5: const router = createRouter({
 6:   history: createWebHistory(),
 7:   routes,
 8: });
 9:
10: export default router;
```

次に、main.jsにVue Routerを使うための設定をします。

リスト13.2: my-vite-todo/src/main.js

```
 1: import { createApp } from 'vue';
 2: import App from './App.vue';
 3: import router from './router';  // 追加
 4:
 5: // 「.use(router)」を追加
 6: createApp(App).use(router).mount('#app');
```

最後に、App.vueで<MainTodo />を呼び出している部分を<router-view />に置き換えます。

リスト13.3: my-vite-todo/src/App.vue

```
 1: <script setup>
 2: import TheHeader from './components/TheHeader.vue';
 3: import TheFooter from './components/TheFooter.vue';
 4: import MainTodo from './components/MainTodo.vue';   // 削除
 5: </script>
 6:
 7: <template>
 8:   <div class="wrap">
 9:     <TheHeader />
10:     <main class="main"><MainTodo /></main> <!-- 削除 -->
11:     <main class="main"><router-view /></main> <!-- 追加 -->
12:     <TheFooter />
13:   </div>
14: </template>
```

画面がTODOのタイトルのみの表示になっていれば準備は完了です。

Vue Routerを使うことでページという考えができました。ここからは、そのページ単位でアプリを作っていくことにします。それに伴い、src配下にviewsまたはpagesというページの置き場所を作ります。どちらでもいいのですが、本書ではpagesフォルダーにページを置いていくことにします。

MainTodo.vueをpagesフォルダー配下に移動してください。

13.3　ルートを設定する

　Vue Routerの準備はできたので、ルートを設定して画面が表示できるようにします。ルートを設定することで、<router-view />がそれに紐づいたページに置き換わります。今の画面が真っ白なのは、ルートの設定がされていないためです。

　index.jsにルートを設定していきます。ルートはconst routes = [];の配列に設定します。

リスト13.4: my-vite-todo/src/router/index.js

```
 1: import { createRouter, createWebHistory } from 'vue-
router';
 2: // 追加
 3: import MainTodo from '/src/pages/MainTodo.vue';  // ①
 4:
 5: const routes = [
 6:   // 追加
 7:   {
 8:     path: '/',             // ②
 9:     name: 'MainTodo',      // ③
10:     component: MainTodo,  // ④
11:   },
12: ];
13:
14: const router = createRouter({
15:   history: createWebHistory(),
16:   routes,
17: });
18:
19: export default router;
```

・① 表示したいページのコンポーネントをインポートします。
・② パスを設定します。ここではトップページとして「/」を設定します。
　　→　「http://localhost:3000/」でアクセス
・③ このルートに名前を付けます。（なくてもOK）
・④ 表示したいコンポーネントを設定します。（①でインポートした名称で）

Vue Routerを導入する前の表示に戻っていれば成功です。

もしも、「http://localhost:3000/mainTodo」でもアクセスしたい場合は、以下のようにpathとnameを設定します。

リスト13.5: my-vite-todo/src/router/index.js

```
 1: const routes = [
 2:   {
 3:     path: '/',
 4:     name: 'Top',
 5:     component: MainTodo,
 6:   },
 7:   {
 8:     path: '/mainTodo',
 9:     name: 'MainTodo',
10:     component: MainTodo,
11:   },
12: ];
```

nameを使う場合は名前が重複しないようにします。

これで

・http://localhost:3000/
・http://localhost:3000/mainTodo

のどちらにアクセスしても、MainTodoコンポーネントのページが表示されます。

13.4　ページを追加してみよう

　ここで新しくAboutページを追加してみましょう。pages配下にAbout.vueを追加します。

リスト13.6: my-vite-todo/src/pages/About.vue

```
<template>
  <div>About Page</div>
</template>
```

　ページを作ったら、次はルートの設定をします。

リスト13.7: my-vite-todo/src/router/index.js

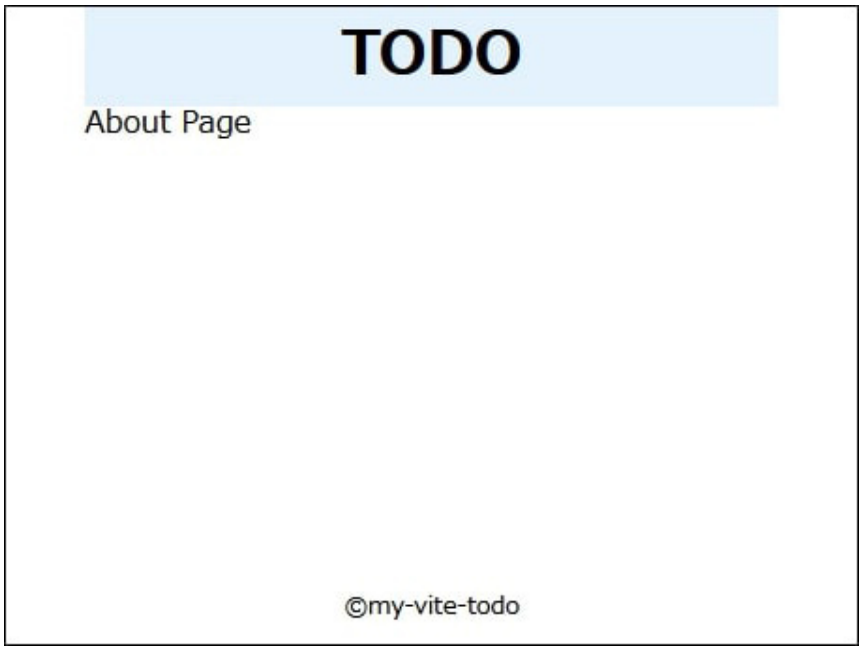
```
// 追加
import About from '/src/pages/About.vue';

const routes = [
  {
    path: '/',
    name: 'Top',
    component: MainTodo,
  },
  {
    path: '/mainTodo',
    name: 'MainTodo',
    component: MainTodo,
  },
  // 追加
  {
    path: '/about',
    name: 'About',
    component: About,
```

```
20:   },
21: ];
```

これで「http://localhost:3000/about」にアクセスしてみてください。以下のようになっていれば成功です。

図13.2:



13.5　遅延ローディングルートとは

　ローカル開発でビルドしている場合はあまり気にすることはありませんが、GitHubにプッシュした後にVercelではビルドが実行されています。「View Build Logs」-「Deployment Status」の「Building」でビルド時のログを見ることができます。

　ローカル開発でも以下のコマンドでビルドすることができます。

```
# npmの場合
$ npm run build

# yarnの場合
$ yarn build
```

　コマンド実行後、my-vite-todo配下にdistフォルダーが作られます。その中のassetsフォルダーに、「index.0db3f86a.js」というようなjsファイルが生成されます。

　これにTODOアプリのJavaScriptが書かれていますが、ページが増えたりしてアプリが大きくなっていくにつれて、このjsファイルも肥大化していきます。そうすると、最初の読み込みに時間が掛かるようになってしまいます。

　それを防ぐためページごとにjsファイルを分割し、そのページを訪れたタイミングで読み込むようにします。このことを「遅延ローディングルート」といいます。

その方法をaboutページを使って実現してみましょう。router/index.jsのAboutルートを以下のように書き変えてください。

リスト13.8: my-vite-todo/src/router/index.js

```
 1: import About from '/src/pages/About.vue';  // 削除
 2:
 3: const routes = [
 4:    (略)
 5:   {
 6:     path: '/about',
 7:     name: 'About',
 8:       component: () => import('/src/pages/About.vue'),
// 変更
 9:   },
10: ];
```

これでもう一度ビルドを実行してみてください。今度はassetsに「About.cbcc88bc.js」が作られていると思います。これは、index.jsからAboutページのJavaScriptが分離された状態です。そして、最初のトップページではAboutに関するJavaScriptは読み込まれず、Aboutページに遷移したタイミングで読み込まれます。

13.6　404ページに誘導する

現在のTODOアプリのアドレスは以下の 3 つです。
・http://localhost:3000/
・http://localhost:3000/mainTodo
・http://localhost:3000/about
もし、これ以外のページのアドレスが指定されたらどうなるでしょうか。
試しに「http://localhost:3000/other」などとしてみてください。ヘッダーとフッターしかない真っ白なページになったと思います。これは最初にルートを設定しなかった場合と同じです。

このように存在しないページが指定された場合、404ページに遷移するようにしたいと思います。
では、先に404エラーのページを作成します。pagesフォルダー配下にNotFound.vueを作成し、以下の内容をコピーしてください。

リスト13.9: my-vite-todo/src/pages/NotFound.vue

```
 1: <template>
 2:   <div class="error">
 3:     <div class="code">
 4:       <p>404</p>
 5:     </div>
 6:     <div class="msg_en">
 7:       <p>Page</p>
 8:       <p>Not</p>
 9:       <p>Found</p>
10:     </div>
11:   </div>
12:   <div class="msg_ja">お探しのページは見つかりませんでした。</div>
13: </template>
14:
```

```
15: <style scoped>
16: .error {
17:   margin-top: 20px;
18:   display: flex;
19:   justify-content: center;
20:   gap: 20px;
21: }
22:
23: .code {
24:   font-size: 64px;
25:   font-weight: bold;
26:   color: #888;
27: }
28:
29: .msg_en {
30:   margin-top: 8px;
31: }
32:
33: .msg_ja {
34:   font-size: 18px;
35:   text-align: center;
36: }
37: </style>
```

404ページの用意ができたら、次はそのルートを設定します。ルート設定にNotFoundを追加してください。

リスト13.10: my-vite-todo/src/router/index.js

```
1: // 追加
2: import NotFound from '/src/pages/NotFound.vue';
3:
4: const routes = [
5:   {
6:     path: '/',
7:     name: 'Top',
8:     component: MainTodo,
9:   },
```

```
10:
11:    ～（略）～
12:
13:    // 追加
14:    {
15:      // 存在しないアドレスにマッチするような指定
16:      path: '/:pathMatch(.*)*',
17:      name: 'NotFound',
18:      component: NotFound,
19:    },
20: ];
```

http://localhost:8080/otherにアクセスしてみてください。404ページが表示されるようになります。

図13.3:

13.7　リンクからページ遷移する

　Vue Routerのルートの書き方は先ほど学んだところですが、リンクからの遷移はどうするのでしょうか。aboutページにリンクから遷移できるようにしてみましょう。

　まずはリンクメニューを作っていきますが、Vue Routerの遷移は内部リンクの場合は<a>タグでなく</router-link>を使います。to属性に遷移したいルートのpathを指定します。

リスト13.11: my-vite-todo/src/App.vue

```
 1: <template>
 2:   <div class="wrap">
 3:     <TheHeader />
 4:     <!-- 追加↓ -->
 5:     <nav>
 6:       <router-link to="/">Home</router-link>
 7:       | <router-link to="/about">About</router-link>
 8:     </nav>
 9:     <!-- 追加↑ -->
10:     <main class="main"><router-view /></main>
11:     <TheFooter />
12:   </div>
13: </template>
```

リンクについて

内部リンク ・・・ プロジェクト内のページ
外部リンク ・・・ Yahoo!やGoogleのようなプロジェクト外のサイト

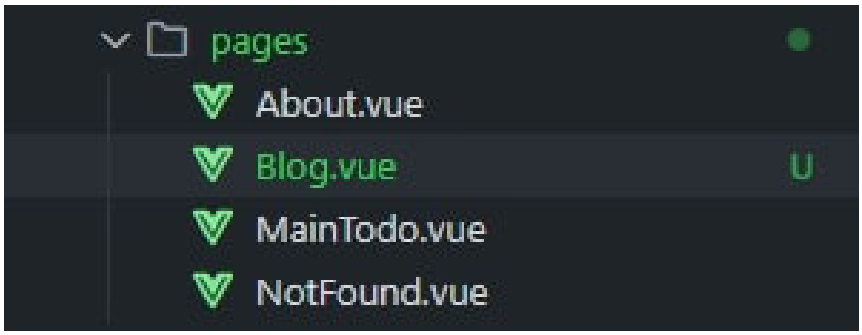
to属性に書いたアドレスとルートで設定したアドレスが一致しないと、リンクは成功しません。よって、プロジェクト外の外部リンクを設定しても遷移することはできません。外部リンクへは<a>タグを使って実現します。

内部リンクは<router-link>タグではなく<a>タグでも遷移はできますが、遷移するたびにページ全体を再読み込みします。<router-link>の場合は<router-view />内が置き変わるだけなので、高速に遷移することができます。

13.8　プログラムからページ遷移する

　次は、プログラム（script）から呼び出して遷移する方法を見ていきましょう。ボタンやリンクを押したとき、すぐに遷移するのではなく何らかの処理をした後に遷移したい場合などに使います。
　メニューに「Blog」を追加し、@clickでBlogページに遷移するようにしてみましょう。まずはpagesにBlog.vueファイルを作成します。

図13.4:



リスト13.12: my-vite-todo/src/pages/Blog.vue

```
<script setup>
console.log('blog');
</script>

<template>
  <p>blog page</p>
</template>
```

　次にメニューに「Blog」を追加し、@clickイベントも追加してください。

リスト13.13: my-vite-todo/src/App.vue

```
 1: <template>
 2:   <div class="wrap">
 3:     <TheHeader />
 4:     <nav>
 5:       <router-link to="/">Home</router-link>
 6:       | <router-link to="/about">About</router-link>
 7:       | <span @click="goBlog">Blog</span>  <!-- 追加 --
>
 8:     </nav>
 9:     <main class="main"><router-view /></main>
10:     <TheFooter />
11:   </div>
12: </template>
```

scriptの方にはgoBlogの行を追加し、onClickで関数を呼び出せるようにします。関数の方も以下のように追加してください。

リスト13.14: my-vite-todo/src/App.vue

```
 1: <script setup>
 2: import TheHeader from './components/TheHeader.vue';
 3: import TheFooter from './components/TheFooter.vue';
 4: import { useRouter } from 'vue-router';  // 追加
 5:
 6: // 追加
 7: const router = useRouter();
 8: const goBlog = () => {
 9:   router.push('/blog');
10: };
11: </script>
```

scriptではuseRouterをインポートし、router.push()で遷移することができます。
うまく遷移できたでしょうか。

図13.5:

13.9　動的ルート

　ブログ形式などでよく使われるURLに「blog/1」、「blog/2」というような、記事によって数字の部分が動的に変わることがあります。
　このような場合は、routesの設定を特別なものにします。パスの動的となる部分に「:」コロン付きの名前を当てます。今回はブログのIDを当ててみます。

リスト13.15: my-vite-todo/src/router/index.js

```
const routes = [
  // 追加
  {
    path: '/blog/:id',  // 「:id」を追加
    name: 'Blog',
    component: () => import('/src/pages/Blog.vue'),
  },
];
```

　リンクの設定にBlog1とBlog2を追加します。

リスト13.16: my-vite-todo/src/App.vue

```
<template>
  <div class="wrap">
    <TheHeader />
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
      | <span @click="goBlog">Blog</span>
      <!-- 追加 -->
      | <router-link to="/blog/1">Blog1</router-link>
      <!-- 追加 -->
```

```
11:       | <router-link to="/blog/2">Blog2</router-link>
12:     </div>
13:     <main class="main"><router-view /></main>
14:     <TheFooter />
15:   </div>
16: </template>
```

これで動的ルートの設定は完了しました。
しかし、ここで以下の問題が出てきました。
・①「/blog」のリンクが404エラーとなる
・②「/blog」、「/blog/1」、「/blog/2」が再描画されない
順番に見ていきましょう。

①は単純にroutesからpath: '/blog'の記述がなくなったため、存在しないURLとされたためです。routesに「/blog」のパスを追加することにより解決します。

リスト13.17: my-vite-todo/src/router/index.js

```
 1: {
 2:   path: '/blog/:id',
 3:   name: 'BlogId',  // 「Blog」から「BlogId」に変更
 4:   component: () => import('/src/pages/Blog.vue'),
 5: },
 6: // 追加
 7: {
 8:   path: '/blog',
 9:   name: 'Blog',
10:   component: () => import('/src/pages/Blog.vue'),
11: },
```

nameオプションは名前付きルートと呼ばれるときに使います。このとき、名前が重複しないように設定します。

nameオプション

　nameオプションの使い方は今までURLで指定したところをnameで指定した名前にします。すると、その名前のpathがルートに設定されます。

リスト13.18:

```
 1: // template
 2: <router-link :to="{ name: 'BlogId', params: { id: 1
}}">
 3:   Blog1
 4: </router-link>
 5:
 6: // script
 7: router.push({ name: 'BlogId', params: { id: 1 } })
```

　②は見た目ではわからないので、次の「パラメータ渡し」で見ていくことにしましょう。

13.10　パラメータ渡し

　今回、ブログのURLとして末尾にIDを設定しました。この動的ルートの使い方として、URLからIDを取得してそのIDの記事を取得したりします。

　routesで設定した'/blog/:id'のように「http://localhost:3000/blog/1」の「1」を取得するにはuseRoute()を使います。前述のuseRouter()と間違わないようにしてください。
　遷移先のblog.vueで、以下のように書きます。

リスト13.19: my-vite-todo/src/pages/Blog.vue

```
 1: <script setup>
 2: import { useRoute } from 'vue-router';
 3:
 4: const route = useRoute();
 5: const id = ref(route.params.id);  // routesで設定した「:id」
と同じ名前
 6: </script>
 7:
 8: <template>
 9:   <h1>blog page</h1>
10:   <p>blog id = {{ id }}</p>
11: </template>
```

　routeのparamsにroutesで設定した名前でURLから取り出すことができます。Blog1、Blog2とリンクを押してみてください。

　ここで気づかれたかと思いますが、アドレスは「/blog/1」と「/blog/2」と変わっていきますが、画面の「blog id = 」に続く数字は「1」か「2」の

どちらかしか表示されません。

　これが「②」の問題点です。「/blog/1」と「/blog/2」とパラメータは変わっていきますが、コンポーネントはBlog.vueの同じインスタンスを利用しているからです。

**　これはコンポーネントのライフサイクルフックが呼ばれないことを意味しています**

13.11　リアクティブの監視

「②」の問題点を解決するには、watch APIを使います。watchは特定の値を監視することができます。今回の場合、アドレス（ルート）を監視し、それの変化をキャッチすれば処理を行うことができます。

以下のように書き変えてください。IDの取得とコンソールへの出力をwatchの内外で比べてみましょう。

リスト13.20: my-vite-todo/src/pages/Blog.vue

```
 1: <script setup>
 2: import { watch } from 'vue';  // 追加
 3: import { useRoute } from 'vue-router';
 4:
 5: const route = useRoute();
 6: const id = route.params.id;
 7: console.log('watch外：', id);  // 追加
 8:
 9: // 追加
10: watch(route, () => {
11:   id.value = route.params.id;
12:   console.log('watch内：', id.value);
13: });
14: </script>
15:
16: <template>
17:   <p>blog page</p>
18:   <p>blog id = {{ id }}</p>
19: </template>
```

これでメニューの「Blog1」、「Blog2」と押してみてください。アドレスは「/blog/1」と「/blog/2」と変化しつつ、画面の「blog id = 」に続く数字とコンソールログもそれに応じた値になっていきます。

コンソールログの初回は「watch外：1」になり、その後は「watch内」になっていると思います。初回だけBlogコンポーネントが呼び出され（watch外）、その後、同じコンポーネントだった場合は初回のコンポーネントを利用するためです。そのためwatchで特定の値を監視する（watch内）ことでその変化を捉えます。

複数監視

監視したい対象が複数ある場合、配列にして渡すことで実現できます。

リスト13.21:

```
1: const valA = ref('A');
2: const valB = ref('B');
3: watch([valA, valB], () => {
4:   console.log(valA, valB);
5: });
```

第14章　外部API連携

　これまではローカルストレージでデータのやりとりをしていましたが、サーバー・クライアント間でやりとりする方法を見ていきます。
　この章はVueの機能ではありませんが、API連携はよく使う機能なので紹介します。

　♦ここで学べること
　・JSON Placeholder
　・fetch API

14.1　JSON Placeholderとは

　JSON Placeholderはダミーデータを返してくれるAPIサービスです。
　サイトはhttps://jsonplaceholder.typicode.com/です。ここにはブログ、コメント、写真やユーザーのようなダミーデータが用意されています。それをAPI経由でhttps://jsonplaceholder.typicode.com/postsのようにすれば、取得することができます。

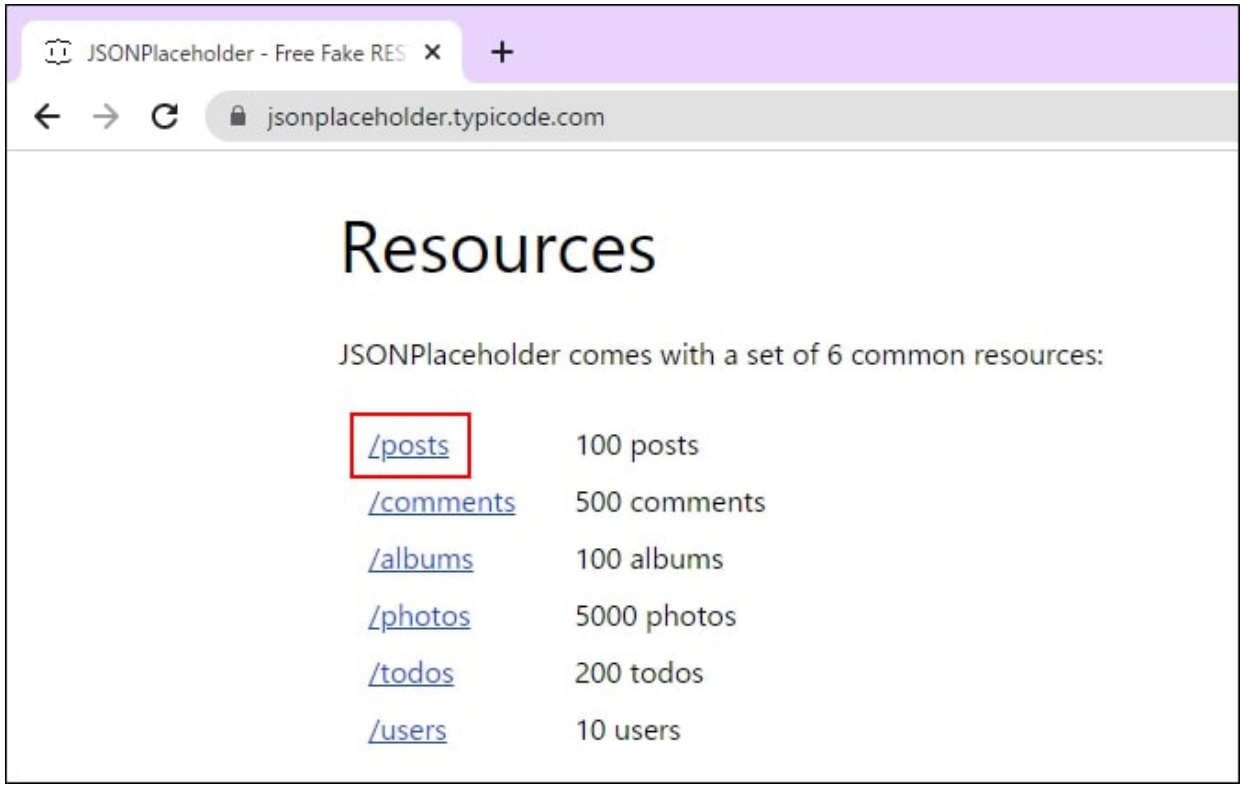
　まだ開発の初期でサーバー側を用意できないときに簡単にデータを取得することができて便利です。名前の通り、取得してくるデータはjson形式です。
　今回はこのJSON Placeholderを利用して、ブログのデータがサーバーに登録されているという想定で学習していきます。

14.2　Fetch APIでデータを取得する

　Fetch APIを簡単に説明すると、サーバーからデータを取得するためのインターフェースであり、Webブラウザーの標準APIです。Ajaxに変わる通信方法の主流になってきています。

　Fetch APIを使って、さきほどのJSON Placeholderからブログのデータを取得してみます。
　サイトの中ほどにある「Resources」の「/posts」をクリックしてみてください。以下のようなダミーデータが表示されたかと思います。このデータを取得してみます。

図14.1:



https://jsonplaceholder.typicode.com/posts

図14.2:


```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
    "body": "quia et suscipit¥nsuscipit recusandae consequuntur expedita et cum¥nreprehenderit
molestiae ut ut quas totam¥nnostrum rerum est autem sunt rem eveniet architecto"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae¥nsequi sint nihil reprehenderit dolor beatae ea dolores
neque¥nfugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis¥nqui aperiam non debitis
possimus qui neque nisi nulla"
  },
  {
    "userId": 1,
    "id": 3,
    "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut",
    "body": "et iusto sed quo iure¥nvoluptatem occaecati omnis eligendi aut ad¥nvoluptatem
doloribus vel accusantium quis pariatur¥nmolestiae porro eius odio et labore et velit aut"
  },
```


このデータを取得するには、以下のようにfetch APIを使います。

リスト14.1: my-vite-todo/src/pages/Blog.vue

```
 1: <script setup>
 2: ～（略）～
 3:
 4: const posts = ref([]);
 5: const fetchData = async () => {
 6:   const response =
        7:                                                    await
fetch('https://jsonplaceholder.typicode.com/posts');
 8:   posts.value = await response.json();
 9: };
10: fetchData();
11: </script>
```

fetch(URL)で取得することができますが、fetchは非同期なので、awaitを使って取得します。取得した結果をresponseで受け、json()とすることで、先ほどのダミーデータを取得することができます。

fetchData();とすることで、Blogページが呼ばれたと同時にデータを取得します。

次にBlogページにJSON Placeholderから取得したデータの一覧を表示したいと思います。

リスト14.2: my-vite-todo/src/pages/Blog.vue

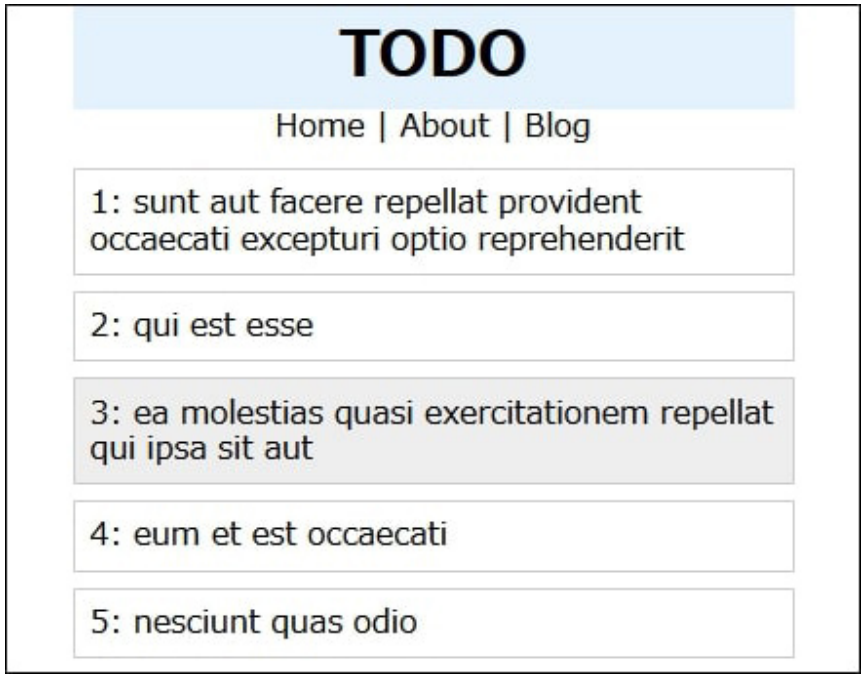
```
 1: <template>
 2:   <ul>
 3:     <li v-for="post in posts" :key="post.id">
 4:       {{ post.id }}:
 5:       <router-link :to="`/blog/${post.id}`">
 6:         {{ post.title }}
 7:       </router-link>
 8:     </li>
 9:   </ul>
10: </template>
11:
12: <style scoped>
13: ul {
14:   margin-top: 12px;
15: }
16:
17: li {
18:   margin-bottom: 8px;
19:   border: 1px solid #ccc;
20:   padding: 8px;
21: }
22:
23: li:hover {
24:   background-color: #eee;
25: }
26: </style>
```

テンプレートリテラル

　<router-link :to="`/blog/${post.id}`">の:toに書かれた「`」（バッククォート）は、囲まれた中に変数を書くことができます。
　変数は「${変数名}」と書くことで値が展開されます。
　バッククォートを使わない書き方なら、以下のようになります。　<router-link :to="'/blog/' + post.id">

　TODOアプリにブログが表示されることになって違和感はありますが、そこは目をつぶってこのまま進めていきましょう。

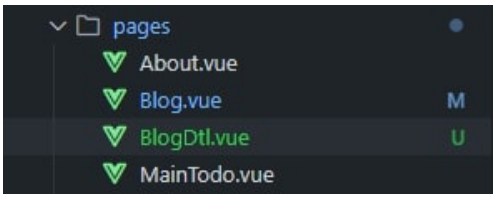
図14.3:



14.3　ブログの詳細ページを作る

　このままブログ一覧を選択しても何も起こらないので、その内容を表示するページを作成したいと思います。
　まずは、pages配下に詳細ページとなる「BlogDtl.vue」を作成します。

図14.4:



　そして、その詳細ページのルートを通しておきます。「BlogId」のルートを変更して使います。

リスト14.3: my-vite-todo/src/router/index.js

```
1: {
2:   path: '/blog/:id',
3:   name: 'BlogDtl',  // 「BlogDtl」に変更
4:   // 「BlogDtl.vue」に変更
5:   component: () => import('/src/pages/BlogDtl.vue'),
6: },
```

　ブログの一覧を表示するとき、router-linkのリンクURLに「/blog/id」というようにブログのIDを付与しました。詳細ページではこのブログIDをURLから取得し、そのIDから詳細データを表示します。まずはそのブログIDを取得して表示させます。

リスト14.4: my-vite-todo/src/pages/BlogDtl.vue

```
 1: <script setup>
 2: import { useRoute } from 'vue-router';
 3:
 4: const route = useRoute();
 5: const id = route.params.id;
 6: </script>
 7:
 8: <template>
 9:   {{ id }}
10: </template>
```

一覧から任意の行を選択し、その行のIDが詳細ページに表示されていれば成功です。

ブログ一覧を取得したものと同様に、ブログIDの内容を取得して表示させましょう。一覧で取得したURLにIDを付与すれば、そのデータを取得することができます。

→ https://jsonplaceholder.typicode.com/posts/ + ID

リスト14.5: my-vite-todo/src/pages/BlogDtl.vue

```
 1: <script setup>
 2: import { ref } from 'vue';
 3: import { useRoute } from 'vue-router';
 4:
 5: const route = useRoute();
 6: const id = route.params.id;
 7:
 8: const dtl = ref({});
 9: const fetchData = async () => {
10:   const response = await fetch(
11:     'https://jsonplaceholder.typicode.com/posts/' + id
12:   );
13:   dtl.value = await response.json();
14: };
```

```
15: fetchData();
16: </script>
17:
18: <template>
19:   <div>
20:     <h1 class="title">{{ dtl.title }}</h1>
21:     <div class="dtl">{{ dtl.body }}</div>
22:   </div>
23: </template>
24:
25: <style scoped>
26: .title {
27:   margin: 12px;
28:   font-size: 20px;
29:   font-weight: bold;
30:   text-align: center;
31: }
32:
33: .dtl {
34:   line-height: 1.5;
35: }
36: </style>
```

おわりに

本書を最後まで読んでいただきありがとうございました。

私にとってアウトプットとしての初めての技術書です。何か形になるものを残してみたいという思いで始めましたが、思いのほか難しい作業でした。

言葉で説明する場合、足りない説明があっても後戻りして補足したり、図や実例もその場で簡単にすることができますが、文章になるとそれができず、また口で説明するようにすると長くて読みづらくなってしまいます。

一気にVueの機能を使ってしまうと説明するのに渋滞してしまうため、少しずつサンプルアプリを進めながらその機能を出していくという構成にも苦労しました。

そして、限られたスペースにソースを書かなければならないため、極力シンプルなコードにしたり、CSSの装飾も質素なものにしました。

まだまだ書きたい機能や触れておきたい機能もありましたが、ボリュームの関係や「超」入門にはそぐわないという理由で断念しました。また、今ではスタンダードになりつつあるTypeScriptを使うことも諦めました。ですが、JavaScriptの開発では必須となってきたTypeScriptやESLintも、本書を理解した後にでも学習することをお勧めします。

本書サンプルの実装は必ずしもベストプラクティスではありません。まだまだVueの歴史は浅く、Vueも「3」にバージョンアップされて書き方も変わりました。フォルダー構成からコンポーネントの分け方から実際にベストプラクティスがないのも事実です。本書サンプルはそこまで気にする

規模ではありませんが、今後の開発でも使えるような構成を参考にしました。

　少しでも多くの人にVueの魅力が伝わり、自身のWebアプリ構築の経験を本書を通して共有できればと思います。

　Vueを学ぶにあたって最初に手にしてよかったと思っていただければ、嬉しく思います。

著者紹介

爰河 英憲（ここかわ ひでのり）

現役Javaエンジニア。独学でモダンなプログラミングを学習。JavaScript、TypeScript、Vue、Nuxt.js、React、Next.jsを中心に学習中。これらを使ってのWebアプリも複数リリース。プログラム言語の学習として最初に読んで良かったと思えるような1冊を目指している。

◎本書スタッフ
アートディレクター/装丁：岡田章志＋GY
編集協力：山部 沙織
ディレクター：栗原 翔
〈表紙イラスト〉
はこしろ
フリーランスのイラストレーター。書籍の表紙からweb用のイラスト、アナログゲームイラストまで、広く手がける。

技術の泉シリーズ・刊行によせて

技術者の知見のアウトプットである技術同人誌は、急速に認知度を高めています。インプレスR&Dは国内最大級の即売会「技術書典」（https://techbookfest.org/）で頒布された技術同人誌を底本とした商業書籍を2016年より刊行し、これらを中心とした『技術書典シリーズ』を展開してきました。2019年4月、より幅広い技術同人誌を対象とし、最新の知見を発信するために『技術の泉シリーズ』へリニューアルしました。今後は「技術書典」をはじめとした各種即売会や、勉強会・LT会などで頒布された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。エンジニアの"知の結晶"である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

株式会社インプレスR&D
技術の泉シリーズ　編集長　山城 敬

●お断り
掲載したURLは2022年12月1日現在のものです。サイトの都合で変更されることがあります。
また、電子版ではURLにハイパーリンクを設定していますが、端末やビューアー、リンク先のファイルタイプによっては表示されないことがあります。あらかじめご了承ください。

●本書の内容についてのお問い合わせ先
株式会社インプレスR&D　メール窓口
np-info@impress.co.jp
件名に「『本書名』問い合わせ係」と明記してお送りください。
電話やFAX、郵便でのご質問にはお答えできません。返信までには、しばらくお時間をいただく場合があります。
なお、本書の範囲を超えるご質問にはお答えしかねますので、あらかじめご了承ください。
また、本書の内容についてはNextPublishingオフィシャルWebサイトにて情報を公開しております。
https://nextpublishing.jp/

技術の泉シリーズ

Vue.js 超入門
3.2対応

2023年1月13日　初版発行Ver.1.0（リフロー版）

著　者　　爰河 英憲
編集人　　山城 敬
企画・編集　合同会社技術の泉出版
発行人　　井芹 昌信
発　行　　株式会社インプレスR&D
〒101-0051
東京都千代田区神田神保町一丁目105番地
https://nextpublishing.jp/

ISBN978-4-295-60152-4

NextPublishing®

◉本書はNextPublishingメソッドによって発行されています。
NextPublishingメソッドは株式会社インプレスR&Dが開発した、電子書籍と印刷書籍を同時発行できるデジタルファースト型の新出版方式です。 https://nextpublishing.jp/