

Webサイト高速化のための

静的サイトジェネレーター 活用入門

GatsbyJSで実現する高速&実用的なサイト構築



サポートPDF

Gatsby v3
gatsby-source-contentful v5
new Gatsby image plugin
対応ガイド

Build blazing-fast websites with GatsbyJS

本 PDF は下記書籍のサポート PDF です。

PDF は GitHub (<https://github.com/ebisucom/gatsbyjs-book/>) で配布しています。



Webサイト高速化のための 静的サイトジェネレーター活用入門

GatsbyJSで実現する高速&実用的なサイト構築

<https://book.mynavi.jp/ec/products/detail/id=115483>

<https://ebisu.com/gatsbyjs-book/>

<https://amzn.to/2x5nuyq>

- ・ 本書に記載された内容は、情報の提供のみを目的としております。したがって、本書を用いての運用はすべてお客様自身の責任と判断において行ってください。
- ・ 本書の制作にあたっては正確な記述につとめましたが、著者や出版社のいずれも、本書の内容に関してなんらかの保証をするものではなく、内容に関するいかなる運用結果についてもいっさいの責任を負いません。あらかじめご了承ください。
- ・ 本書中に掲載している画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。ハードウェアやソフトウェアの環境によっては、必ずしも本書通りの画面にならないことがあります。あらかじめご了承ください。
- ・ 本書は2021年3月段階での情報に基づいて執筆されています。本書に登場するソフトウェアのバージョン、URL、製品のスペックなどの情報は、すべてその原稿執筆時点でのものです。執筆以降に変更されている可能性がありますので、ご了承ください。
- ・ 本書中に登場する会社名および商品名は、該当する各社の商標または登録商標です。本書では®およびTMマークは省略させていただきます。

はじめに



Gatsby v3 がリリースされました。これに伴い、公式プラグインも更新されています。特に、

- ✓ gatsby-plugin-image@1.0.0 への移行を推奨
- ✓ gatsby-source-contentful@5.0.0 への移行が必要

となっています。

どちらも、これまでのプラグインとはデータの扱い方が変わっていますので、注意が必要です。本 PDF では、Gatsby v3 に対応する際の注意点と、こうした新しいプラグインに対応する方法をまとめています。

▶ Gatsby v3への対応

- 1 プラグインのバージョンとインストール
- 2 index.jsの画像
- 3 StaticImageを使う
- 4 footer.jsの画像
- 5 about.jsのアイキャッチ画像
- 6 blogpost-template.js (blogpost.js) のアイキャッチ画像
- 7 blogpost-template.js (blogpost.js) のリッチテキスト
- 8 blog-template.js (blog.js) & cat-template.jsのアイキャッチ画像

Build blazing-fast websites with GatsbyJS

GatsbyJS

Gatsby v3への対応



本書のサンプルを v3 に対応させる際の注意点と方法をまとめました。
Gatsby の v3 への更新と、それに合わせたプラグインの更新となります。

■ Node 12

Gatsby v3 では、Node 12.13.0 以降が必要となりました。
環境を確認して、必要に応じてバージョンアップしてください。

■ Gatsby v3 & プラグインのインストール

Gatsby を v3 に更新します。

```
$ yarn add gatsby@latest
```

または

```
$ npm install gatsby@latest
```

続いて、アップデートの必要なプラグインを確認します。

```
$ yarn outdated
```

または

```
$ npm outdated
```

Gatsby 公式のプラグインがリストアップされます。

Package	Current	Wanted	Latest	Package Type	URL
gatsby-plugin-image	0.7.1	0.7.2	1.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-plugin-manifest	2.12.0	2.12.1	3.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-plugin-offline	3.10.0	3.10.2	4.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-plugin-react-helmet	3.10.0	3.10.0	4.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-plugin-sharp	2.14.1	2.14.3	3.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-source-contentful	4.6.1	4.6.3	5.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-source-filesystem	2.11.0	2.11.1	3.0.0	dependencies	https://github.com/gatsbyjs/...
gatsby-transformer-sharp	2.12.0	2.12.0	3.0.0	dependencies	https://github.com/gatsbyjs/...

これらをすべて最新のものに更新します。

```
$ yarn add gatsby-plugin-image@latest gatsby-plugin-manifest@latest gatsby-plugin-offline@latest gatsby-plugin-react-helmet@latest gatsby-plugin-sharp@latest gatsby-source-contentful@latest gatsby-source-filesystem@latest gatsby-transformer-sharp@latest
```

yarn の場合、次のコマンドで選択して更新することもできます。

```
$ yarn upgrade-interactive --latest
```

以上で、v3 への対応は終了です。v3 への変更に伴うその他の修正に関しては、下記を参考にしてください。

Migrating from v2 to v3

<https://www.gatsbyjs.com/docs/reference/release-notes/migrating-from-v2-to-v3/>

1 プラグインのバージョンとインストール



この原稿を書いている時点の Gatsby と各プラグインのバージョンは以下のとおりです。

■ gatsby@3.0.1

■ gatsby-plugin-image@1.0.0

■ gatsby-plugin-sharp@3.0.0

■ gatsby-transformer-sharp@3.0.0

gatsby-plugin-image

<https://github.com/gatsbyjs/gatsby/tree/master/packages/gatsby-plugin-image>

書籍 P.62

各プラグインは書籍の P.62 でインストールしています。新しい構成では「gatsby-plugin-image」が増えていますので、注意してください

```
$ yarn add gatsby-plugin-image gatsby-plugin-sharp gatsby-transformer-sharp
```

```
module.exports = {  
  /* Your site config here */  
  plugins: [  
    `gatsby-plugin-image`,  
    `gatsby-transformer-sharp`,  
    `gatsby-plugin-sharp`,  
    {  
      resolve: `gatsby-source-filesystem`,  
      options: {  
        name: `images`,  
        path: `${__dirname}/src/images/`,  
      },  
    },  
  ],  
}
```

gatsby-config.js

■ gatsby-source-contentful@5.0.0

書籍 P.118

gatsby-source-contentful プラグインは書籍の P.118 でインストールしています。

すでに作成が進んでいる場合、プラグインを入れ替えると `blogpost-template.js` で「json フィールドがない」というエラーが出ます。そこで、`json` のクエリを削除し、そのデータを処理している部分をコメントアウトして進めてください。

```
content {  
  json  
}
```

`src/templates/blogpost-template.js`

それでは、対応していきます。

2 index.jsの画像

書籍 P.65 ~

これまでの gatsby-Image では、フラグメントを利用して複数のデータを取得する必要がありました。しかし、gatsby-plugin-image を利用する場合は gatsbyImageData を取得することになります。



file>childImageSharp内に「gatsbyImageData」があります。

そのため、クエリは次のようにシンプルなものになります。

```
export const query = graphql`
  query{
    file(relativePath: {eq: "hero.jpg"}) {
      childImageSharp {
        gatsbyImageData(layout: FULL_WIDTH)
      }
    }
  }
`
```

src/pages/index.js

これまでの fluid の指定は「Full width」となり、layout で次のように指定します。

```
gatsbyImageData(layout: FULL_WIDTH)
```

また、Fluid (Full width)、Fixedに加えて、新しく Constrained が追加されました。使い方のヒントやパラメーターに関しては下記を参照してください。

Gatsby Image plugin

<https://www.gatsbyjs.com/docs/reference/built-in-components/gatsby-plugin-image/>

書籍 P.70 ~

取得したデータを表示するためには、新しい GatsbyImage をインポートします。

```
import { GatsbyImage } from "gatsby-plugin-image"
```

src/pages/index.js

そして、これまでの gatsby-image と同じようにクエリの結果を渡します。

```
<GatsbyImage image={data.file.childImageSharp.gatsbyImageData} alt="" />
```

src/pages/index.js



画像が表示されます。

※プレースホルダの表示については
本PDFのP.15を参照してください。

書籍 P.76 ~

これらを踏まえてクエリを置き換えると、以下のようになります。

```
export const query = graphql`
  query {
    hero: file(relativePath: {eq: "hero.jpg"}) {
      childImageSharp {
        gatsbyImageData(layout: FULL_WIDTH)
      }
    }
    fruit: file(relativePath: {eq: "fruit.jpg"}) {
      childImageSharp {
        gatsbyImageData(width: 320, layout: CONSTRAINED)
      }
    }
    grain: file(relativePath: {eq: "grain.jpg"}) {
      childImageSharp {
        gatsbyImageData(width: 320, layout: CONSTRAINED)
      }
    }
    beverage: file(relativePath: {eq: "beverage.jpg"}) {
      childImageSharp {
        gatsbyImageData(width: 320, layout: CONSTRAINED)
      }
    }
    berry: file(relativePath: {eq: "berry.jpg"}) {
      childImageSharp {
        gatsbyImageData(layout: FULL_WIDTH)
      }
    }
    pattern: file(relativePath: {eq: "pattern.jpg"}) {
      childImageSharp {
        gatsbyImageData(quality: 90, layout: FULL_WIDTH)
      }
    }
  }
`
```

src/pages/index.js



画面幅に合わせて横幅を変える画像 (Full width)。



画面幅に合わせて横幅を変えつつ、最大幅を320pxにする画像 (Constrained)。最大幅は width で指定。



画面幅に合わせて横幅を変える画像 (Full width)。



画面幅に合わせて横幅を変える画像 (Full width)。クオリティは quality で指定。

※作業が進んでいる場合、pattern.jpg の設定は footer.js で行います。

gatsbyImageDataで取得されるデータ

gatsbyImageData で取得されるデータは次のようになっています。
WebP のデータも取得され、標準で対応していることがわかります。

■ Full width の画像の場合

Full width の画像の場合、デフォルトでは主要なデバイスの画面幅に合わせた横幅 750、1080、1366、1920 ピクセルの画像が生成されます（breakpoints で設定できます）。ただし、オリジナルよりも大きいサイズは生成されません。
オリジナルの横幅が 1600 ピクセルの hero.jpg の場合、750、1080、1366、1600 の画像が生成され、gatsbyImageData で取得されます。

さらに、sizes の値は「100vw」に設定され、画像の表示サイズが常に画面の横幅に合わせて変化することを示しています。

The screenshot shows the GraphQL IDE interface. On the left, the 'Explorer' pane shows the file structure with 'gatsbyImageData' expanded and 'layout: FULL_WIDTH' selected. The main editor shows a GraphQL query and its JSON response. The query is:

```
query MyQuery {
  file(relativePath: {eq: "hero.jpg"}) {
    childImageSharp {
      gatsbyImageData(layout: FULL_WIDTH)
    }
  }
}
```

The response shows the 'data' field with 'file' containing 'childImageSharp' which has 'gatsbyImageData'. The 'gatsbyImageData' object includes 'layout' (FULL_WIDTH), 'backgroundImage' (a data URI), 'images' (an array of image sources for different breakpoints), and 'sources' (an array of image sources for different breakpoints). A red box highlights the 'layout: FULL_WIDTH' in the query, and a red arrow points to the 'sources' field in the response, which is labeled '取得データ。' (Acquired data).

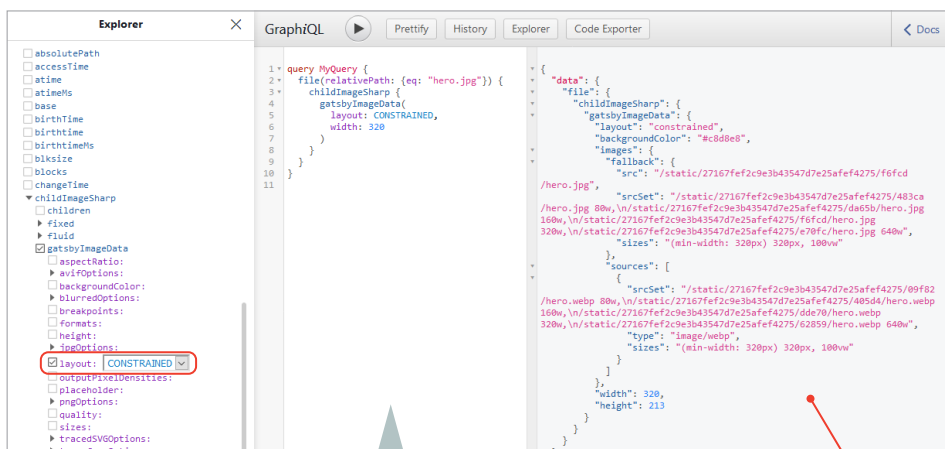
```
query MyQuery {
  file(relativePath: {eq: "hero.jpg"}) {
    childImageSharp {
      gatsbyImageData(layout: FULL_WIDTH)
    }
  }
}
```

取得データ。

■ Constrained の画像の場合

Constrained の画像の場合、width で指定した最大幅に合わせたサイズが用意されます。たとえば、最大幅を「320」ピクセルにすると、80、160、320、640 の画像が取得されることがわかります。

sizes の値は「(min-width: 320px) 320px, 100vw」に設定され、画像の表示サイズが 320 ピクセル以上の横幅にならないことを示しています。



```
query MyQuery {
  file(relativePath: {eq: "hero.jpg"}) {
    childImageSharp {
      gatsbyImageData(
        layout: CONSTRAINED,
        width: 320
      )
    }
  }
}
```

取得データ。

書籍 P.273 ~

Contentful から最新記事を表示する部分も調整が必要です。Contentful から `gatsbyImageData` が取得できるようになりましたので、クエリを次のように指定します。

```
export const query = graphql`
  query {
    hero: file(relativePath: {eq: "hero.jpg"}) {
      ...
    }
    allContentfulBlogPost(
      sort: { order: DESC, fields: publishDate }
      skip: 0
      limit: 4
    ) {
      edges {
        node {
          title
          id
          slug
          eyecatch {
            gatsbyImageData(width: 573, layout: CONSTRAINED)
            description
          }
        }
      }
    }
  }
`
```

src/pages/index.js



`allContentfulBlogPost>edges>node>eyecatch` 内に「`gatsbyImageData`」があります。

あとは、表示部分を `GatsbyImage` で指定します。以上で、`index.js` は完了です。

```
<figure>
  <GatsbyImage
    image={node.eyecatch.gatsbyImageData}
    alt={node.eyecatch.description}
    style={{ height: "100%" }}
  />
</figure>
```

src/pages/index.js



Contentfulの画像が表示されます。ここではConstrainedで画像の最大幅を573pxに指定しています。

レスポンシブイメージをより細かく最適化する

gatsbyImageData に用意されたさまざまなオプションを利用すると、レスポンシブイメージをより細かく最適化できるようになっています。

たとえば、モバイル版でも2段組みの記事一覧の場合、デフォルトの設定では大きいサイズの画像が読み込まれてしまいます。sizes 属性の設定により、画像の表示幅が「画面幅が 573px 以上のときは 573px、それ以外のときは 100vw」として処理されるためです。

```
gatsbyImageData(  
  width: 573  
  layout: CONSTRAINED  
)
```

```
<img ...  
  sizes="(min-width: 573px) 573px, 100vw"  
  srcset="... 143w, ... 287w, ... 573w, ... 1146w">
```

sizes を次のように設定すると、表示に使用する画像サイズをより最適化することができます。ここでは画像の表示幅が「画面幅が 1146px 以上のときは 573px、それ以外のときは 50vw」として処理されるように指定しています。

```
gatsbyImageData(  
  width: 573  
  layout: CONSTRAINED  
  sizes: "(min-width: 1146px) 573px, 50vw"  
)
```

```
<img ...  
  sizes="(min-width: 1146px) 573px, 50vw"  
  srcset="... 143w, ... 287w, ... 573w, ... 1146w">
```



画面幅375pxでの表示。

表示に使用される画像



DPR 1のデバイスの場合:
横幅573pxの画像。



DPR 2のデバイスの場合:
横幅1146pxの画像。



画面幅375pxでの表示。

表示に使用される画像



DPR 1のデバイスの場合:
横幅287pxの画像。

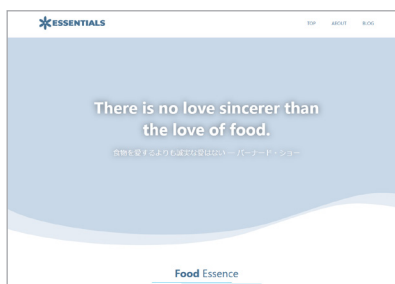


DPR 2のデバイスの場合:
横幅573pxの画像。

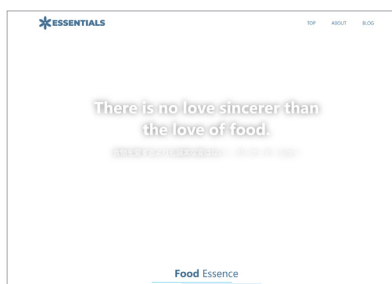
※DPR=Device Pixel Ratio

画像のプレースホルダ

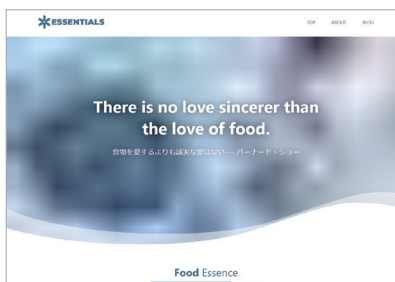
画像のプレースホルダは、デフォルトでは画像から抽出した単色を使用する「Dominant color (DOMINANT_COLOR)」になります。Contentfulの画像の場合、ブラー画像を使用する「Blurred (BLURRED)」になります。どのプレースホルダを使用するかは、placeholderで設定できます。



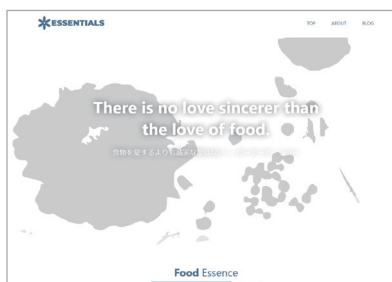
placeholder: DOMINANT_COLOR



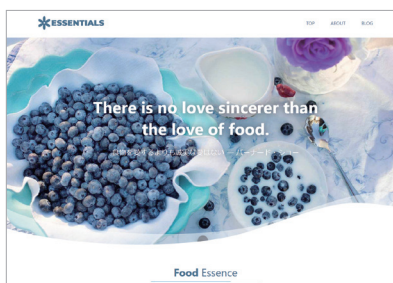
placeholder: NONE



placeholder: BLURRED



placeholder: TRACED_SVG



※現在のところ、「Dominant color」の表示はビルドすることで確認できます。

3 StaticImageを使う



gatsby-plugin-image では、新たに StaticImage も利用できるようになりました。これにより、クエリを使わずにファイルを直接指定して GatsbyImage と同等の最適化を行うことができます。利用するためには、StaticImage をインポートします。

```
import { GatsbyImage, StaticImage } from "gatsby-plugin-image"
```

続いて、ローカルファイルを表示している GatsbyImage を書き換えます。たとえば、hero.jpg は次のように書き換えることができます。画像ファイルのパスは相対パスで指定します（URL が分かる場合は、リモートのファイルを指定することも可能です）。

クエリで指定していた layout は、こちらではコンポーネント側で指定します。

```
<GatsbyImage
  image={data.hero.childImageSharp.gatsbyImageData}
  alt=""
  style={{ height: "100%" }}
/>
```



```
<StaticImage
  src="../../images/hero.jpg"
  alt=""
  layout="fullWidth"
  style={{ height: "100%" }}
/>
```

layoutの値は、GatsbyImageでは「FULL_WIDTH」、
「CONSTRAINED」と指定しましたが、StaticImage
では「fullWidth」、「constrained」と指定します。

画像を直接指定できるケースではクエリを用意する必要もなくなり、非常にシンプルな構造になります。

4 footer.jsの画像



書籍 P.107 ~

index.js と同様に、GatsbyImage へと置き換えます。まず、GatsbyImage をインポートします。

```
import { GatsbyImage } from "gatsby-plugin-image"
```

src/components/footer.js

続いて、クエリで gatsbyImageData を取得するようにします。

```
const data = useStaticQuery(graphql`
  query {
    pattern: file(relativePath: {eq: "pattern.jpg"}) {
      childImageSharp {
        gatsbyImageData(quality: 90, layout: FULL_WIDTH)
      }
    }
  }
`)
```

src/components/footer.js

最後に、表示部分を書き換えて完了です。

```
<GatsbyImage
  image={data.pattern.childImageSharp.gatsbyImageData}
  alt=""
  style={{ height: "100%" }}
/>
```

src/components/footer.js



フッターの画像が表示されます。画面幅に合わせて横幅を変えるFull widthに設定しています。

StaticImage を使って、クエリをなくしてしまっても問題ありません。

5 about.jsのアイキャッチ画像



書籍 P.124 ~

こちら、index.jsと同様にアイキャッチ画像の設定を変更します。まず、GatsbyImageをインポートします。

```
import { GatsbyImage } from "gatsby-plugin-image"
```

src/pages/about.js

続いて、クエリでgatsbyImageDataを取得するようにします。

```
export const query = graphql`
  query {
    about: file(relativePath: { eq: "about.jpg" }) {
      childImageSharp {
        gatsbyImageData(layout: FULL_WIDTH)
        original {
          src
          height
          width
        }
      }
    }
  }
`
```

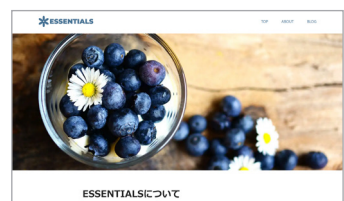
※originalは書籍P.166~で追加する設定です。

src/pages/about.js

表示部分を書き換えて完了です。

```
<GatsbyImage
  image={data.about.childImageSharp.gatsbyImageData}
  alt=" ブルーベリー & ヨーグルト "
/>
```

src/pages/about.js



アバウトページのアイキャッチ画像が表示されます。

6 blogpost-template.js (blogpost.js) のアイキャッチ画像



書籍 P.208 ~

こちら、アイキャッチ画像の設定を変更します。まず、GatsbyImage をインポートします。

```
import { GatsbyImage } from "gatsby-plugin-image"
```

src/templates/blogpost-template.js (src/pages/blogpost.js)

続いて、クエリで gatsbyImageData を取得します。

```
export const query = graphql`
  ...
  eyecatch {
    gatsbyImageData(layout: FULL_WIDTH)
    description
    file {
      details {
        image {
          width
          height
        }
      }
      url
    }
  }
  ...
`
```

※fileは書籍P.243~ で追加する設定です。

src/templates/blogpost-template.js (src/pages/blogpost.js)

表示部分を書き換えて完了です。

```
<GatsbyImage
  image={data.contentfulBlogPost.eyecatch.gatsbyImageData}
  alt={data.contentfulBlogPost.eyecatch.description}
/>
```

src/templates/blogpost-template.js (src/pages/blogpost.js)



ブログ記事ページのアイキャッチ画像が表示されます。

7 blogpost-template.js (blogpost.js) のリッチテキスト

書籍 P.212 ~

gatsby-source-contentful が v4 となり、それに合わせてリッチテキストの扱い方が変更になりましたので、それに合わせて設定していきます。

まずは、GraphQL でフィールドの構成を確認しておきます。これまでのプラグインではリッチテキストのデータを取得するために

`contentfulBlogPost > content > json`

というフィールドが用意されていました。取得できるデータも json 形式でした。

The screenshot shows a GraphQL IDE with the following query and result:

```

1 query MyQuery {
2   contentfulBlogPost {
3     content {
4       json
5     }
6   }
7 }

```

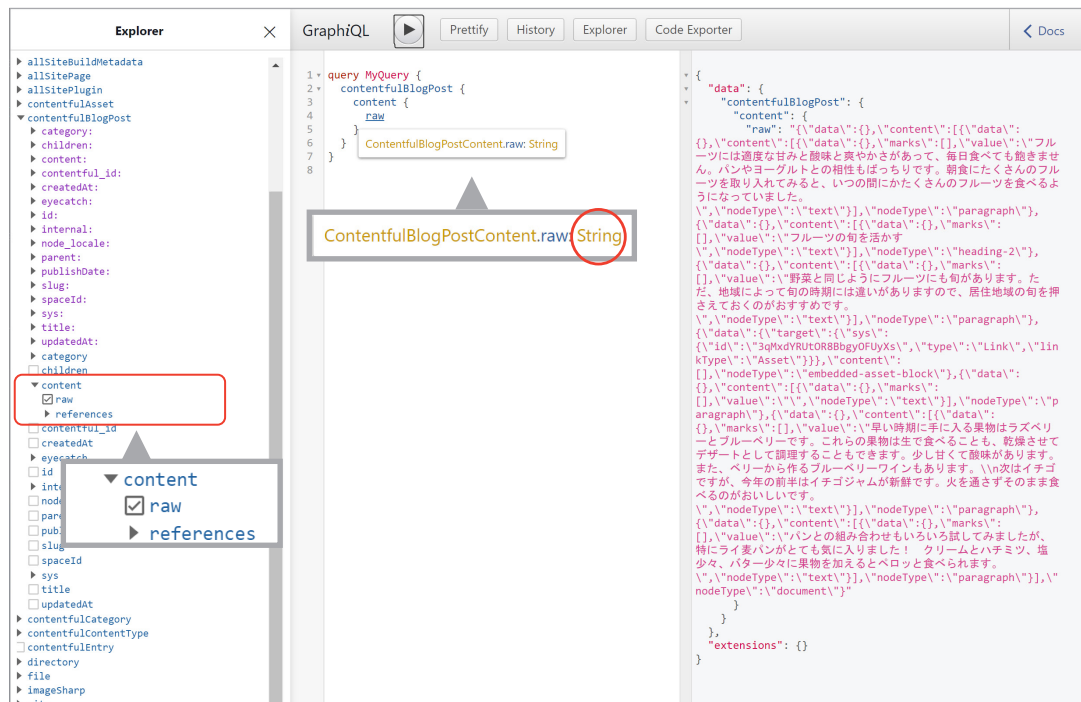
The result is a JSON array of contentfulBlogPostContentRichTextNode objects. A red box highlights the 'json' field in the Explorer and the 'json' field in the result. A red circle highlights the 'JSON' text in the result. A red box highlights the 'contentfulBlogPostContentRichTextNode.json JSON' text in the result.

これまでのプラグインで取得できるデータ。

しかし、新しくなったプラグインでは

`contentfulBlogPost > content > raw`

というフィールドへと変更されています。取得できるデータも文字列 (String) のデータとなっていますので、注意してください。



新しくなったプラグインで取得できるデータ。

まずは、クエリで `raw` を取得するように指定します。

```
content {
  raw
}
```

`src/templates/blogpost-template.js` (`src/pages/blogpost.js`)

書籍 P.214 ~

取得したデータを React コンポーネントに変換します。

これまでの `documentToReactComponents` の代わりに、`renderRichText` を利用しますのでインポートします。`renderRichText` は、`gatsby-source-contentful` に含まれていますので、新たにインストールするものではありません。

```
import { renderRichText } from "gatsby-source-contentful/rich-text"
```

`src/templates/blogpost-template.js` (`src/pages/blogpost.js`)

そして、次のようにすることでデータを変換し、出力することができます。`data.contentfulBlogPost.content` の最後に「`.raw`」をつけていないことに注意してください。

```
<div className="postbody">
  {renderRichText(data.contentfulBlogPost.content)}
</div>
```

`src/templates/blogpost-template.js` (`src/pages/blogpost.js`)



リッチテキストのコンテンツが表示されます。

リッチテキスト内の要素の扱いについては、基本的に変更はありませんので `options` の設定も進めてください。

ただし、リッチテキスト内の画像（`EMBEDDED_ASSET`）については、取得するためのクエリが必要になりましたので対応していきます。

書籍 P.218 ~

EMBEDDED_ASSET に関する情報は、これまでであれば json の中に含まれる形で取得できていました。しかし、新しいプラグインでは EMBEDDED_ASSET に関する情報が必要な場合には、それをクエリに追加する必要があります。

クエリのサンプルはプラグインの GitHub のページにありますので、これを参考にします。

Contentful Rich Text

<https://github.com/gatsbyjs/gatsby/tree/master/packages/gatsby-source-contentful#contentful-rich-text>

また、新しいプラグインでは EMBEDDED_ASSET に対しても `gatsbyImageData` を取得できますので、以下のようなクエリを追加します。

```
content {
  raw
  references {
    ... on ContentfulAsset {
      contentful_id
      __typename
      gatsbyImageData(layout: FULL_WIDTH)
      title
      description
    }
  }
}
```

`src/templates/blogpost-template.js` (`src/pages/blogpost.js`)

これは、`... on ~` (Inline Fragments) を利用して `ContentfulAsset` がある場合にはデータを取得し、`__typename` (Meta fields) を利用してオブジェクトの型の名前を取得しています。

前ページのクエリによって取得されるデータを確認するため、GitHub のページのサンプルから次の設定を options に追加します。

```
[BLOCKS.EMBEDDED_ASSET]: node => {
  return (
    <>
    <h2>Embedded Asset</h2>
    <pre>
      <code>{JSON.stringify(node, null, 2)}</code>
    </pre>
    </>
  )
},
```

src/templates/blogpost-template.js (src/pages/blogpost.js)

ページ上には次のように表示され、データの内容を確認することができます。



Embedded Asset

```
{
  "data": {
    "target": {
      "contentful_id": "3qMxdYRUtOR8BgyOFUyXs",
      "typename": "ContentfulAsset",
      "gatsbyImageData": {
        "images": {
          "sources": [
            {
              "srcSet": "///images.ctfassets.net/k3ffctx2yqhy/3qMxdYRUtOR8BgyOFUyXs/e99ddf9988d54b41c85e519beb0a8919/season.jpg?w=750&h=750&format=webp",
              "sizes": "100vw",
              "type": "image/webp"
            }
          ],
          "fallback": {
            "src": "///images.ctfassets.net/k3ffctx2yqhy/3qMxdYRUtOR8BgyOFUyXs/e99ddf9988d54b41c85e519beb0a8919/season.jpg?w=750&h=750&format=webp",
            "srcSet": "///images.ctfassets.net/k3ffctx2yqhy/3qMxdYRUtOR8BgyOFUyXs/e99ddf9988d54b41c85e519beb0a8919/season.jpg?w=750&h=750&format=webp",
            "sizes": "100vw"
          }
        },
        "layout": "fullWidth",
        "width": 1,
        "height": 0.5,
        "placeholder": {
          "fallback": "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAQABQAQAA"
        }
      }
    },
    "title": "season",
    "description": "旬のぶどう",
    "sys": {
      "type": "Asset",
      "id": "3qMxdYRUtOR8BgyOFUyXs"
    }
  },
  "content": [],
  "nodeType": "embedded-asset-block"
}
```

gatsbyImageDataが用意されています。

あとは、このデータを使う形で EMBEDDED_ASSET のコードを用意します。gatsbyImageData を直接扱えるため、これまでよりもシンプルな構成になっています。

```
[BLOCKS.EMBEDDED_ASSET]: node => (
  <GatsbyImage
    image={node.data.target.gatsbyImageData}
    alt={
      node.data.target.description
        ? node.data.target.description
        : node.data.target.title
    }
  />
),
```

src/templates/blogpost-template.js (src/pages/blogpost.js)



リッチテキスト内の画像が表示されます。

書籍 P.241 ~

記事のメタデータとして、記事の説明を用意します。EMBEDDED_ASSET を含まないデータは data.contentfulBlogPost.content.raw として取得できていますので、これを利用します。ただし、最初に確認したとおり、このデータは文字列です。一方、documentToPlainTextString では json 形式が必要です。そこで、文字列から json への変換を加えます。以上で、リッチテキストまわりの設定は完了です。

```
pagedesc={`${documentToPlainTextString(
  JSON.parse(data.contentfulBlogPost.content.raw)
).slice(0, 70)}...`}
```

src/templates/blogpost-template.js (src/pages/blogpost.js)

```
<meta data-react-helmet="true" name="description"
  content=" フルーツには適度な甘みと酸味と爽やかさがあって、毎日食べても飽きません。パンやヨーグルトとの相性もばっちりです。朝食にたくさんのフルーツを取...">
```

```
<meta data-react-helmet="true" property="og:description"
  content=" フルーツには適度な甘みと酸味と爽やかさがあって、毎日食べても飽きません。パンやヨーグルトとの相性もばっちりです。朝食にたくさんのフルーツを取...">
```



画面表示には影響しません。

8 blog-template.js (blog.js) & cat-template.jsのアイキャッチ画像



書籍 P.254 ~

記事一覧ページの各記事のアイキャッチ画像を GatsbyImage を使った設定にします。まず、GatsbyImage をインポートします。

```
import { GatsbyImage } from "gatsby-plugin-image"
```

src/templates/blog-template.js (src/pages/blog.js) または cat-template.js

続いて、クエリで gatsbyImageData を取得します。

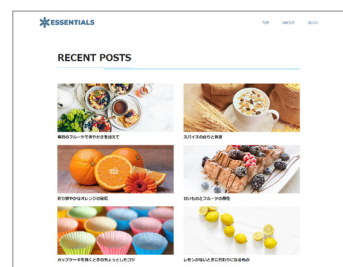
```
export const query = graphql`
  ...
  edges {
    node {
      title
      id
      slug
      eyecatch {
        gatsbyImageData(width: 500, layout: CONSTRAINED)
        description
      }
    }
  }
  ...
`
```

src/templates/blog-template.js (src/pages/blog.js) または cat-template.js

表示部分を書き換えて完了です。

```
<GatsbyImage
  image={node.eyecatch.gatsbyImageData}
  alt={node.eyecatch.description}
  style={{ height: "100%" }}
/>
```

src/templates/blog-template.js (src/pages/blog.js)
または cat-template.js



各記事のアイキャッチ画像が表示されます。

■ 著者

エビスコム

<https://ebisu.com/>

さまざまなメディアにおける企画制作を世界各地のネットワークを駆使して展開。コンピュータ、インターネット関係では書籍、デジタル映像、CG、ソフトウェアの企画制作、WWW システムの構築などを行う。

主な編著書：『HTML5&CSS3 デザイン 現場の新標準ガイド【第2版】』マイナビ出版刊
『CSS グリッドレイアウト デザインブック』同上
『6 ステップでマスターする「最新標準」HTML+CSS デザイン』同上
『WordPress レッスンブック 5.x 対応版』ソシム刊
『フレキシブルボックスで作る HTML5&CSS3 レッスンブック』同上
『CSS グリッドで作る HTML5&CSS3 レッスンブック』同上
『HTML&CSS コーディング・プラクティスブック』エビスコム電子書籍出版部刊
『グーテンベルク時代の WordPress ノート テーマの作り方（入門編）』同上
『グーテンベルク時代の WordPress ノート テーマの作り方
（ランディングページ&ワンカラムサイト編）』同上

ほか多数

Web サイト高速化のための 静的サイトジェネレーター活用入門【サポート PDF】

Gatsby v3

gatsby-source-contentful v5

new Gatsby image plugin 対応ガイド

2021 年 2 月 12 日 ver.1.0 発行

2021 年 3 月 6 日 ver.1.1 発行