Gatsby

Build blazing-fast websites with GatsbyJS

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-source-contentful v4 &
new Gatsby image plugin
対応ガイド

エビスコム 編著

マイナビ

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

はじめに

Gatsby で Contentful のデータを扱うプラグイン「gatsby-source-contentful」が v4 からリッチテキストまわりの扱いに大幅な修正が入りました。さらに、新しい Gatsby image plugin である「gatsby-plugin-image」にも対応しました。

本 PDF では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。

Build blazing-fast websites with GatsbyJS

GatsbyJS

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@2.32.3

- gatsby-plugin-image@0.7.1
- gatsby-plugin-sharp@2.14.1
- gatsby-transformer-sharp@2.12.0

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@4.6.1

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
    }
```

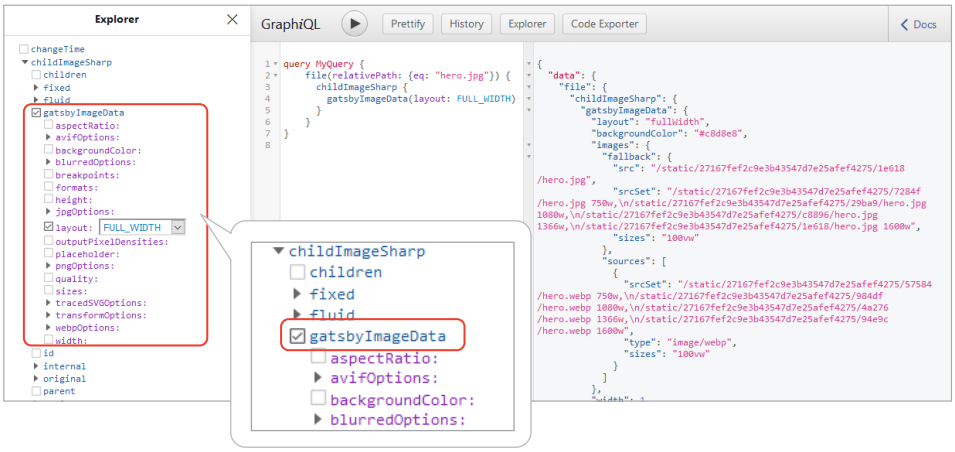
src/templates/blogpost-template.js

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.13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。

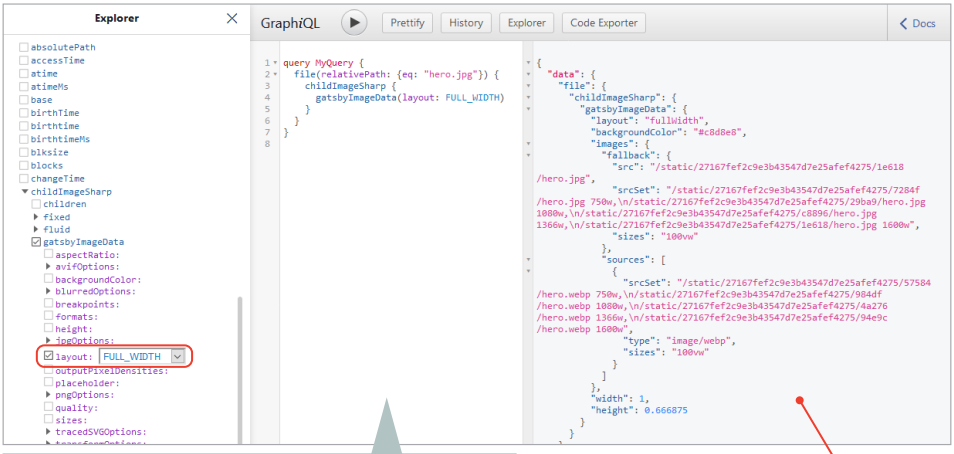
gatsbyImageDataで取得されるデータ

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。



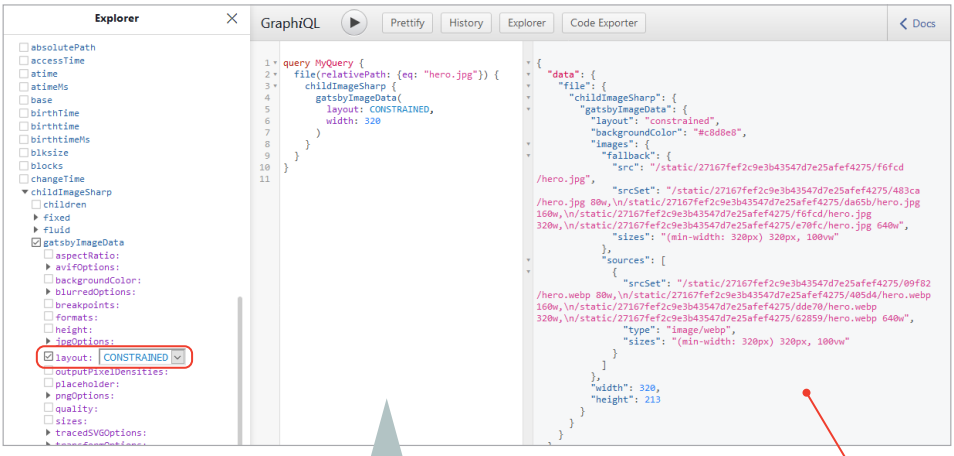

```
query MyQuery {
    file(relativePath: {eq: "hero.jpg"}) {
      childImageSharp {
        gatsbyImageData(layout: FULL_WIDTH)
      }
    }
}
```

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。



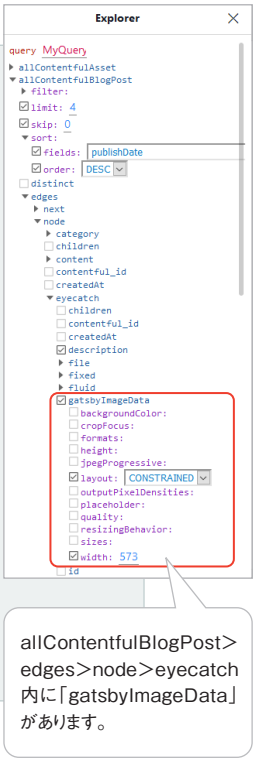

```
query MyQuery {
    file(relativePath: {eq: "hero.jpg"}) {
      childImageSharp {
        gatsbyImageData(
            layout: CONSTRAINED,
            width: 320
        )
      }
    }
}
```

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 からも 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">
```

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。

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。

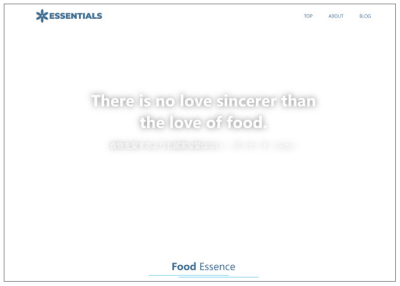
DPR 2のデバイスの場合:
横幅573pxの画像。

※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。

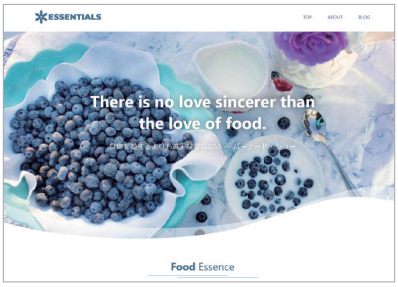
placeholder: DOMINANT_COLOR



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 ～

こちらも、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
        }
      }
    }
  }
`
```

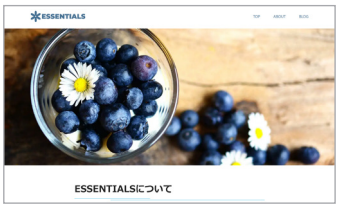
※originalは書籍P.166～ で追加する設定です。

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 ～

こちらも、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

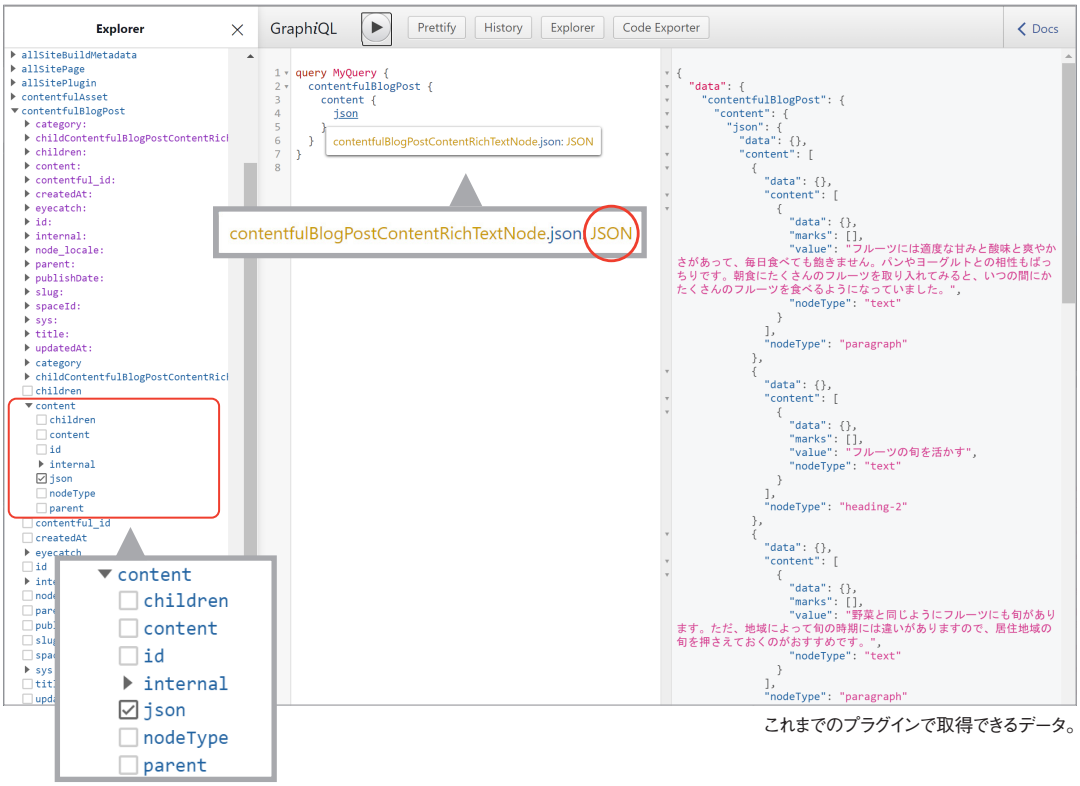
書籍 P.212 ～

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、GraphiQL 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。

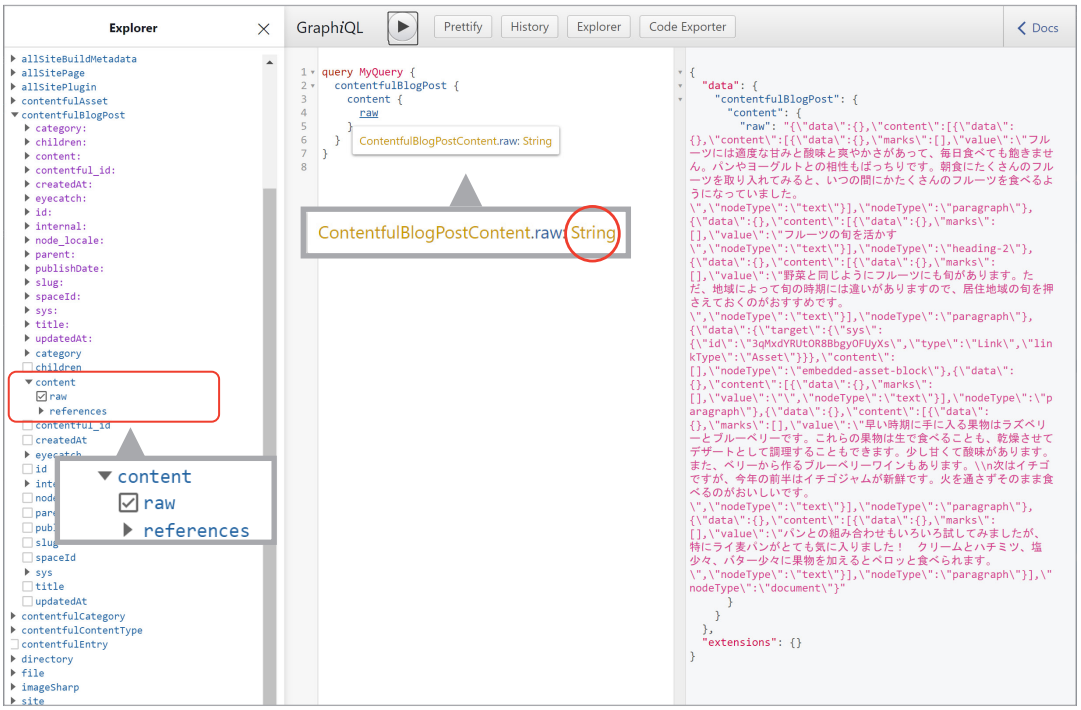


これまでのプラグインで取得できるデータ。

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）

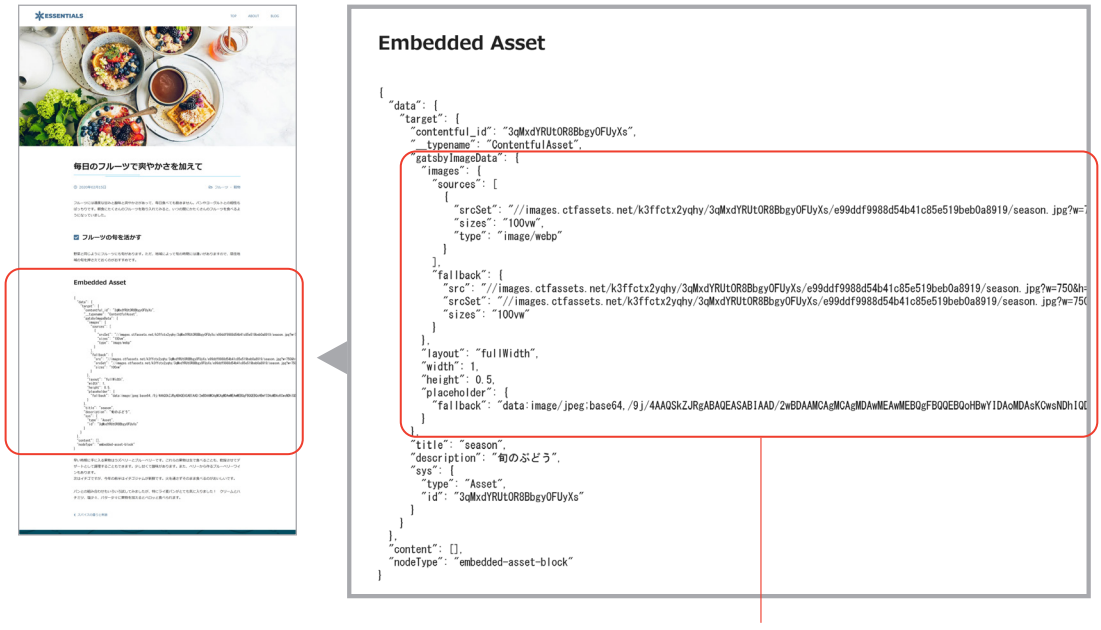
これは、... on ～ （Inline Fragments）を利用して ContentfulAsset がある場合にはデータを取得し、__typename（Meta fields）を利用してオブジェクトの型の名前を取得しています。

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。



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。

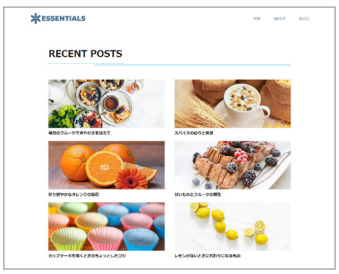
```
export const query = graphql`
  ...
      edges {
        node {
          title
          id
          slug
          eyecatch {
            gatsbyImageData(width: 500, layout: CONSTRAINED)
            description
          }
        }
      }
  ...
`
```

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-source-contentful v4 & new Gatsby image plugin 対応ガイド

2021 年 2 月 12 日　ver.1.0 発行