

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



副読本

microCMS対応ガイド

Build blazing-fast websites with GatsbyJS



エビスコム 編著



本 PDF は下記書籍の副読本です。この PDF では、書籍の第 2 部で完成させたプロジェクトをもとに、microCMS へ対応させていきます。

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

- URL: <https://book.mynavi.jp/ec/products/detail/id=115483>
- FLIP: <https://ebisu.com/gatsbyjs-book/>
- AMAZON: <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 2020 年 4 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いております。

CONTENTS

もくじ

CHAPTER

1

アカウントとコンテンツの準備

5

STEP 1-1	アカウントの準備	6
└	microCMS	6
└	microCMS のアカウントを作成する	6
STEP 1-2	コンテンツの準備	8
└	サービスの作成	9
└	API の作成	10
└	コンテンツデータのインポート	14
STEP 1-3	GraphQL で microCMS のデータを扱うための準備	17
└	API キーの確認	17
└	プラグインのインストールと設定	17
STEP 1-4	変更点の確認	20
└	contentfulBlogPost と microcmsBlog	20
└	contentfulCategory と microcmsCategory	23

CHAPTER

2

記事一覧

24

STEP 2-1	トップページ	25
└	クエリを追加する	26
└	microCMS からのデータに置き換える	27
└	レスポンシブイメージの設定を行う	28

STEP 2-2	記事一覧ページ	30
└	クエリを追加する	30
└	microCMS からのデータに置き換える	31
STEP 2-3	カテゴリーページ	33
└	クエリを追加する	33
└	microCMS からのデータに置き換える	35

CHAPTER

3

記事

37

STEP 3-1	記事ページ	38
└	microCMS からのデータに置き換える	38
└	メタデータ	39
└	アイキャッチ画像	40
└	記事	41
STEP 3-2	リッチテキストの処理	43
STEP 3-3	アイキャッチ画像の表示スペースを確保してレイアウトシフトを防ぐ	46
└	アイキャッチ画像の横幅と高さを取得する	47
└	画像の横幅と高さを元に表示スペースを確保する	49
STEP 3-4	OGP 画像の横幅と高さを指定する	51
STEP 3-5	仕上げ	52
	COLUMN microCMS によるサイトの更新	53

APPENDIX **gatsby-image**を使う **54**

APPENDIX A	createRemoteFileNode	55
APPENDIX B	gatsby-plugin-imgix	57
	COLUMN 表示 CHECK	59

microCMS

1

アカウントとコンテンツ の準備

- 1-1 アカウントの準備
- 1-2 コンテンツの準備
- 1-3 GraphQLでmicroCMSのデータを扱うための準備
- 1-4 変更点の確認

Build blazing-fast websites with GatsbyJS

GatsbyJS

1-1 アカウントの準備

microCMS

microCMS は日本製ヘッドレス CMS サービスで、非常にわかりやすいインターフェースが特徴です。無料のプランが用意されていますので、手軽に試すことができます。

- API 数： 10 個
- Webhook 数： 10 個
- カスタムフィールド数： 2 個
- コンテンツ数： 無制限
- データ転送量： 100GB

※プラン詳細はこちら <https://microcms.io/pricing/>

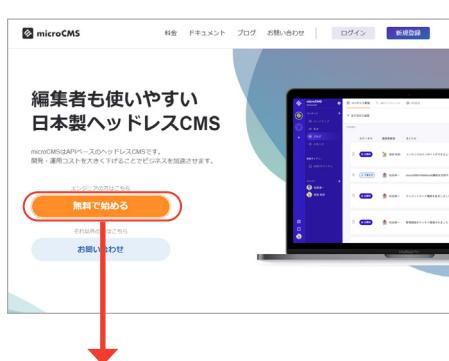
また、画像処理には imgix の API が利用可能です。



microCMS
<https://microcms.io/>

microCMSのアカウントを作成する

①



上記 microCMS のサイトにアクセスし、「新規登録」または「無料で始める」をクリックします。

②



アカウント登録

メールアドレス
mailaddress@microcms.io

パスワード

8文字以上で入力してください。
大文字、小文字、数字を含める必要があります。

○ 利用規約、プライバシーポリシーに同意します

登録する

アカウント登録へ
こちらの方はこちら

「アカウント登録」ページが表示されますので、メールアドレスとパスワードを入力します。

「利用規約、プライバシーポリシー」を確認し、問題がなければ同意をオンにして「登録する」をクリックします。

③



確認コード入力

メールに記載された確認コードを入力してください。

確認コード
123456

送信する

確認コードの入力を求められますので、メールで送られてきたコードを入力し、「送信する」をクリックします。

④



サービス情報を入力

microCMSにご登録ありがとうございます

下記のステップでコンテンツを構築していくましょう！

01 1. プロジェクト
プロジェクト名を入力して下さい。プロジェクト名は、英数字で構成してください。

02 2. API登録
API登録情報を登録して下さい。API登録情報を登録して下さい。API登録情報を登録して下さい。

03 3. コンテンツ登録
コンテンツ登録情報を登録して下さい。コンテンツ登録情報を登録して下さい。

1/3ステップ

無事にアカウントが作成されると、左のような画面が表示されます。

メッセージを閉じたら、コンテンツの準備をしていきます。



サービス情報を入力

サービス名
初期状態でプロジェクト名を入力して下さい。

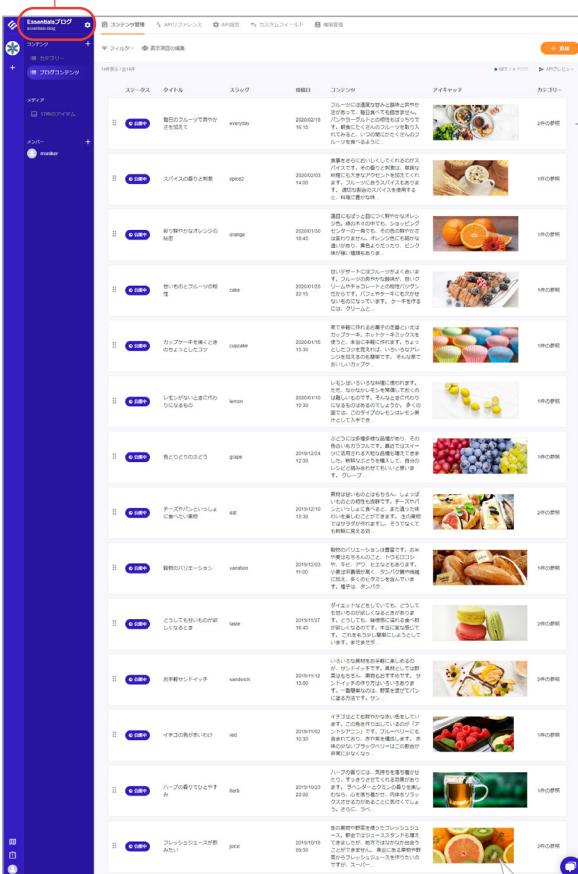
サービス
サービス名を入力する場合は、サブドメインを指定します。例: microcms.jp

1/3ステップ

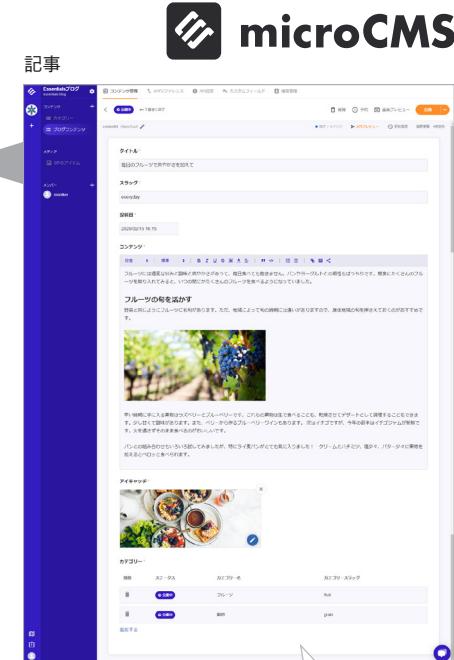
1-2 コンテンツの準備

microCMS でコンテンツを準備します。ここでは Contentful で構築したコンテンツと同様の構造で作成していきます。

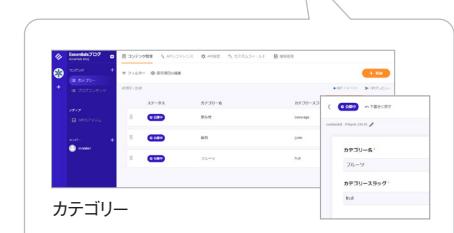
サービス



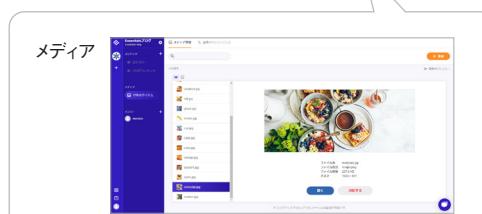
記事



コンテンツ



メディア



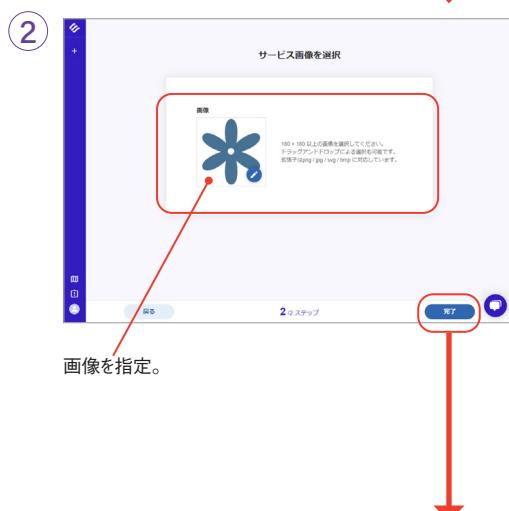
サービスの作成

まずは、サービスの作成です。Contentful のスペースに相当します。



サービス名とサービス ID を指定します。自分でわかりやすいもので問題ありません。ここではサービス名を「Essentials ブログ」、サービス ID を「essentials-blog」と指定しています。

設定ができたら「次へ」をクリックします。



サービスのアイコンとして表示される画像を指定し、「完了」をクリックします。



「サービス登録が完了しました」と表示されますので、表示された URL をクリックします。

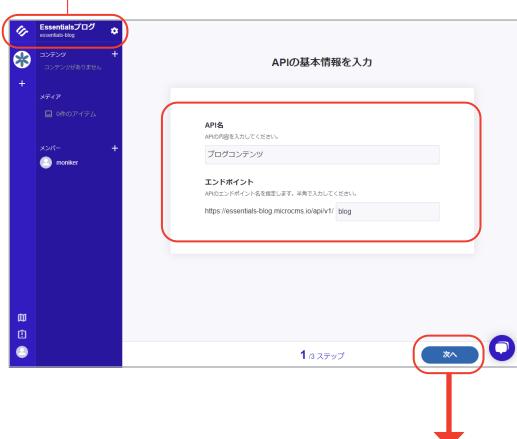


APIの作成

続いて、APIを作成していきます。APIはContentfulのコンテンツタイプに相当するものです。

1 ブログコンテンツ用のAPIを作成する

「Essentialsブログ」サービス。



まずは、ブログのコンテンツデータのためのAPIを作成します。ここではAPI名を「ブログコンテンツ」、エンドポイントを「blog」と指定しています。

エンドポイントで指定したものはGraphQLのフィールド名として、「microcmsBlog」といった形で使用されます。

設定ができたら「次へ」をクリックします。



「APIの型を選択」では「リスト形式」を選択して、次へをクリックします。



ダウンロードデータに同梱したapi-blogpost.jsonをインポート。

「APIスキーマ (インターフェース) を定義」では、コンテンツを管理するためのフィールドを用意していきます。

本書ではインポートデータを用意しましたので、「ファイルをインポートする場合はこちらから」からapi-blogpost.jsonをインポートします。



問題がなければ左のような表示になりますので、完了をクリックします。

ここでは6つのフィールドを用意しています。

フィールドID	表示名	種類
title	タイトル	テキストフィールド
slug	スラッグ	テキストフィールド
publishDate	投稿日	日時
content	コンテンツ	リッチエディタ
eyecatch	アイキャッチ	画像
category	カテゴリー	複数コンテンツ参照

※全フィールド必須項目
(入力必須)に設定。

設定をインポートせずにフィールドを作成する場合、フィールドID、表示名、種類を指定して作成します。

フィールドID。

表示名。

種類。

ExDvAFRUu

フィールドID	表示名	種類
title	タイトル	テキストフィールド

種類を選択してください

テキストフィールド 自由入力の行テキストです。タイトル等に適しています。	テキストエリア 自由入力の複数行テキストです。ブランク入力による入力になります。	リッチエディタ 自由入力の複数行テキストです。リッチエディタによる入力が可能です。HTMLが表示できます。
画像 画像用のフィールドです。APIからは画像URLが返却されます。	ファイル Starterプラン以降でご利用いただけます	日時 Date型のフィールドです。カレンダーから日時を選択することができます。
真偽値 Boolean型のフィールドです。スイッチでオフ/オフを切り替えることができます。	コンテンツ参照 各コンテンツのリストを置きこむことで、各コンテンツのリストを複数選択することができます。	複数コンテンツ参照 各コンテンツを複数選択することができます。リスト形式は動作形態となります。
数字 Number型のフィールドです。入力時は数値型のキー入力が強制されます。	カスタム カスタムフィールドです。設定済みのカスタムフィールドを用いて入力できます。	繰り返し 内halbのカスタムフィールドを複数選択し、繰り返し入力が可能です。

必須項目
設定をONにすると入力時の入力が必須となります

説明文
入稿画面に表示する説明文です。入稿者にとってわかりやすい説明を入力しましょう。

例: 1160x400サイズの画像を選択してください

閉じる

フィールドを削除

入力を必須にする設定は
「詳細設定」に用意されています。

選択できる
フィールドの種類。

11

コンテンツの準備 1-2

2 カテゴリー用のAPIを作成する



続いて、カテゴリーのための API を作成します。
「コンテンツ」の右にある「+」をクリックし、API 名を「カテゴリー」、エンドポイントを「category」と指定します。



「API の型を選択」ではこちらもリスト形式を選択し、api-category.json をインポートします。

ダウンロードデータに同梱したapi-category.jsonをインポート。



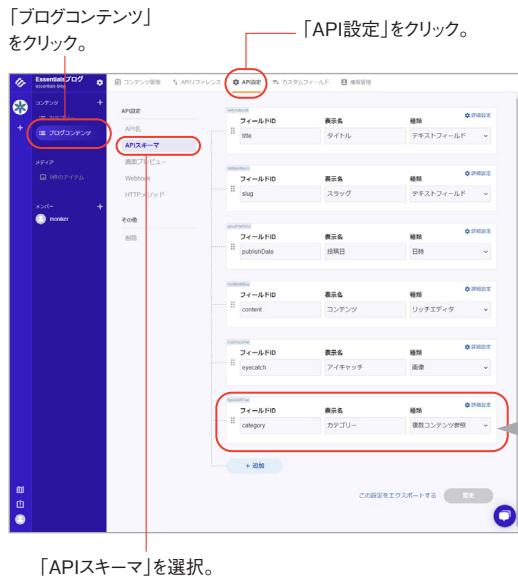
問題がなければ左のような表示になりますので、完了をクリックします。

ここでは2つのフィールドを用意しています。

フィールドID	表示名	種類
category	カテゴリー名	テキストフィールド
categorySlug	カテゴリーslug	テキストフィールド

※全フィールド必須項目
(入力必須)に設定。

3 複数コンテンツの参照を設定する



「APIスキーマ」を選択。



「カテゴリー(category)」を選択。



最後に、複数コンテンツの参照を設定します。
「ログコンテンツ」をクリックし、「API 設定」で「API スキーマ」を選択します。

作成した API スキーマが表示されますので、category のフィールドの「種類」の欄をクリックします。



「種類」の欄をクリック。

種類の選択画面が開きますので、「複数コンテンツ参照」を選択します。参照したいコンテンツとして「カテゴリー (category)」を選択し、「決定」をクリックします。

元の画面に戻ります。最後に「変更」をクリックして設定を保存したら完了です。

コンテンツデータのインポート

コンテンツのデータをインポートしていきます。

1 カテゴリーのデータをインポートする



「フルーツ」カテゴリーの
編集画面。

まずはカテゴリーのデータをインポートするため、「カテゴリー」を選択して、「コンテンツ管理」を開きます。

空の状態ですので、「インポートする」をクリックします。

「ファイルの選択」で `input-category.csv` を選択し、インポートを開始します。

ダウンロードデータに同梱した `input-category.csv` を選択。

インポートが完了すると、左のように表示されます。ここでは3つのカテゴリーを用意しています。

クリックして、中身を確認しておいてください。

新しいカテゴリーを追加する場合、「コンテンツ管理」の画面で「追加」をクリックします。



カテゴリー名などを
入力して「公開」を
クリックします。

2 ブログ記事のデータをインポートする



続けて、ブログ記事のデータをインポートするため、「ブログコンテンツ」の「コンテンツ管理」を開きます。

カテゴリーのときと同様に、「インポートする」から input-blog.csv をインポートします。

ダウンロードデータに同梱した
input-blog.csvを選択。

インポートが完了すると、以下のように表示されます。ここでは 14 件の記事を用意しています。ただし、アイキャッチ、リッチテキスト内の画像と見出しの設定、カテゴリーの設定はインポートできていないため、各記事の編集画面を開いて設定をしていきます。

「毎日のフルーツで爽やかさを加えて」の編集画面。

記事を編集したら「公開」をクリックし、編集内容を保存するのを忘れないようにします。すべての記事を編集したら、コンテンツの準備は完了です。

編集後は「公開」をクリックして保存。

The image shows two screenshots of the microCMS interface. The left screenshot displays a list of 15 blog posts with titles like '毎日のフルーツで爽やかさを加えて', 'スパイスの香りと刺激', and 'フレッシュジュースが飲みたい'. The right screenshot shows a detailed edit view for the post '毎日のフルーツで爽やかさを加えて'. It includes a preview of the post content, a rich text editor, and a sidebar for settings like 'タイトル', 'スラッグ', and '公開日'. A red box highlights the '公開' (Publish) button in the top right corner. Another red box highlights the '見出し2' (H2) heading in the rich text editor. A third red box highlights the '画像' (Image) icon in the rich text editor, pointing to a thumbnail of a blueberry image. A fourth red box highlights the 'アイキャッチ' (Thumbnail) icon in the sidebar, pointing to a thumbnail of a fruit platter. A fifth red box highlights the 'カテゴリー' (Category) icon in the sidebar, pointing to a list of categories including 'フルーツ' (Fruit), '穀物' (Grain), and '飲み物' (Drink).

本書のサンプルでは次のように設定しています。使用する画像はダウンロードデータ内の `base/images-blogpost/` フォルダに収録しています。

タイトル	アイキャッチ	カテゴリー	リッチテキスト内の画像と見出し
毎日のフルーツで爽やかさを加えて	everyday.jpg	フルーツ、穀物	画像 (season.jpg)、見出し
スパイスの香りと刺激	spice.jpg	飲み物	画像 (spice01.jpg)、見出し
彩り鮮やかなオレンジの秘密	orange.jpg	フルーツ	画像 (color.jpg)
甘いものとフルーツの相性	cake.jpg	穀物	
カップケーキを焼くときのちょっとしたコツ	cup.jpg	穀物	
レモンがないときに代わりになるもの	lemon.jpg	フルーツ	
色とりどりのぶどう	grape.jpg	フルーツ	見出し
チーズやパンといっしょに食べたい果物	eat.jpg	フルーツ、穀物	
穀物のバリエーション	variation.jpg	穀物	
どうしても甘いものが欲しくなるとき	taste.jpg	飲み物、穀物	
お手軽サンドイッチ	sandwich.jpg	フルーツ、穀物	
イチゴの色が赤いわけ	red.jpg	フルーツ	
ハーブの香りでひとやすみ	herb.jpg	飲み物	
フレッシュジュースが飲みたい	juice.jpg	フルーツ、飲み物	

1-3

GraphQLでmicroCMSのデータを扱うための準備



APIキーの確認

microCMS のデータを利用するため、必要になる API キーを確認しておきます。



サービス名の右にある歯車のアイコンをクリックし、「API-KEY」を選択します。

X-API-KEY の「表示」ボタンをクリックし、キーの値を確認します。

プラグインのインストールと設定

書籍の第 2 部で完成させた Gatsby のプロジェクトを用意し、ブログ部分を microCMS のデータに置き換えていきます。まずは、Gatsby から microCMS を利用するための、ソースプラグインをインストールします。

```
$ yarn add gatsby-source-microcms
```

gatsby-source-microcms

<https://www.gatsbyjs.org/packages/gatsby-source-microcms/>

続けて、次ページのように gatsby-config.js にプラグインの設定を追加します。ここでは Contentful の設定の下に2つ分の API の設定を追加しています。

この段階でContentfulの設定を削除してしまうと、プロジェクトが立ち上がらなくなります。

```

...
{
  resolve: `gatsby-source-contentful`,
  options: {
    spaceId: process.env.CONTENTFUL_SPACE_ID,
    accessToken: process.env.CONTENTFUL_ACCESS_TOKEN,
    host: process.env.CONTENTFUL_HOST,
  },
},
{
  resolve: "gatsby-source-microcms",
  options: {
    apiKey: process.env.microCMS_API_KEY,
    serviceId: "essentials-blog",
    endpoint: "blog",
    query: {
      limit: 100,
    },
  },
},
{
  resolve: "gatsby-source-microcms",
  options: {
    apiKey: process.env.microCMS_API_KEY,
    serviceId: "essentials-blog",
    endpoint: "category",
    query: {
      limit: 100,
    },
  },
},
],
}

```

Contentfulの設定。

microCMSの
ブログコンテンツ用APIの設定。

microCMSの
カテゴリー用APIの設定。

gatsby-config.js

apiKey

apiKey には先程確認した API キーを指定しますが、ここでは環境変数で指定できるようにしています。

serviceId

serviceId には、P.9 で設定したサービス ID を指定します。

endpoint

endpoint には、各 API (P.10 と P.12) で指定したエンドポイントを指定します。

limit

limit にはいくつのポストまで取得するかを指定します。デフォルトでは 10 に設定されています。

設定が完了したら、開発サーバーを起動します。起動の際には、Contentful に加えて microCMS の環境変数も指定するのを忘れないでください。

```
$ microCMS_API_KEY=xxxxxxxxx CONTENTFUL_SPACE_ID=xxxxxxxxx CONTENTFUL_ACCESS_TOKEN=xxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxx CONTENTFUL_HOST=cdn.contentful.com gatsby develop -H 0.0.0.0
```

無事に起動すると、**GraphiQL** には microCMS のフィールドが現れます。

The screenshot shows the GraphiQL interface with the following details:

- Explorer** tab is selected.
- GraphiQL** tab is active.
- Code Exporter** tab is visible.
- query MyQuery** is defined in the code editor.
- Result** pane shows the JSON response for the query.
- Sidebar** shows available fields:
 - allContentfulAsset**
 - allContentfulBlogPost**
 - allContentfulBlogPostContentRichText**
 - allContentfulCategory**
 - allContentfulContentType**
 - allDirectory**
 - allFile**
 - allImageSharp**
 - allMicrocmsBlog**
 - filter:** (with options: limit, skip, sort, distinct, edges, next, node, blogId, category, children, content, createdAt, eyecatch, id, internal, parent, publishDate, slug, title, updatedAt, previous, group)
- Result pane** displays the following JSON response for the query:


```
query MyQuery {
  allMicrocmsBlog {
    edges {
      node {
        title
      }
    }
  }
}

{
  "data": {
    "allMicrocmsBlog": {
      "edges": [
        {
          "node": {
            "title": "毎日のフルーツで爽やかさを加えて"
          }
        },
        {
          "node": {
            "title": "スパイスの香りと刺激"
          }
        },
        {
          "node": {
            "title": "彩り鮮やかなオレンジの秘密"
          }
        },
        {
          "node": {
            "title": "甘いものとフルーツの相性"
          }
        },
        {
          "node": {
            "title": "カップケーキを焼くときのちょっとしたコツ"
          }
        },
        {
          "node": {
            "title": "レモンがないときに代わりになるもの"
          }
        }
      ]
    }
  }
}
```
- Annotations:**
 - A red box highlights the **allMicrocmsBlog** and **allMicrocmsCategory** fields in the sidebar.
 - A line connects these highlighted fields to the **edges** field in the query results.
 - A red arrow points from the **title** field in the results to the **title** field in the highlighted sidebar items.
 - A callout box with a red border points to the **edges** field in the results, with the text "microCMSのフィールド。" (microCMS field).
 - A callout box with a red border points to the **title** field in the results, with the text "microCMSから取得した記事のタイトル。" (Title of the article obtained from microCMS).

1-4 変更点の確認

GraphQL を利用して、取得できるデータの違いを確認していきます。

contentfulBlogPost と microcmsBlog

まずは、contentfulBlogPost と microcmsBlog です。構成を揃えるように準備していますので、取得できるデータに違いはありません。



contentfulBlogPost

```

query MyQuery {
  contentfulBlogPost {
    title
    publishDate
    category {
      categorySlug
      id
    }
  }
}

```

data

```

{
  "data": [
    {
      "contentfulBlogPost": {
        "title": "毎日のフルーツで爽やかさを加えて",
        "publishDate": "2020-02-15T16:14+09:00",
        "category": [
          {
            "category": "フルーツ",
            "categorySlug": "fruit",
            "id": "de75ce8b-c72b-5b61-bf62-c0c9e95d6f94"
          },
          {
            "category": "穀物",
            "categorySlug": "grain",
            "id": "e047d955-1642-544d-a579-5f520f6e49b1"
          }
        ]
      }
    }
  ]
}

```



microcmsBlog

```

query MyQuery {
  microcmsBlog {
    title
    publishDate
    category {
      categorySlug
      id
    }
  }
}

```

data

```

{
  "data": [
    {
      "microcmsBlog": {
        "title": "毎日のフルーツで爽やかさを加えて",
        "publishDate": "2020-02-15T07:15:00.000Z",
        "category": [
          {
            "category": "フルーツ",
            "categorySlug": "fruit",
            "id": "R9qmK-XKUB"
          },
          {
            "category": "穀物",
            "categorySlug": "grain",
            "id": "pzbwS_wF7P"
          }
        ]
      }
    }
  ]
}

```

ただし、eyecatch に関してはかなりの違いがあります。Contentful からはgatsby-image 用のデータをはじめ、さまざまなデータが取得できますが、microCMS から取得できるのは画像の url のみとなっています。そのため、画像関連はちょっと工夫が必要です。

Explorer

```

query MyQuery {
  contentfulBlogPost {
    title
    eyecatch {
      fluid {
        base64
        aspectRatio
        src
        srcSet
        srcWebp
        sizes
      }
    }
    description
    file {
      details {
        image {
          height
          width
        }
      }
    }
  }
}

```

GraphiQL

Prettify History Explorer Code Exporter Docs



Explorer

```

query MyQuery {
  microcmsBlog {
    title
    eyecatch {
      url
    }
  }
}

```

GraphiQL

Prettify History Explorer Code Exporter Docs



また、リッチテキストのデータが、Contentful では AST が取得できるのに対して、microCMS では HTML が取得できるあたりも異なります。



The screenshot shows the Contentful GraphQL Explorer interface. On the left, the 'Explorer' sidebar lists the fields of a 'contentfulBlogPost' content type, including category, children, content, contentful_id, createdAt, eyecatch, id, internal, node_locale, parent, publishDate, slug, spaceId, sys, title, and updatedAt. A 'category' field is expanded, showing its children: 'content', 'id', 'internal', and 'parent'. The 'content' field is expanded, showing its children: 'content', 'id', and 'internal'. The 'content' field is checked, and its 'json' field is checked. The 'parent' field is checked. The 'contentful_id' field is checked. The 'createdAt' field is checked. The 'eyecatch' field is checked. A 'category' field is expanded, showing its children: 'content', 'id', 'internal', and 'parent'. The 'content' field is expanded, showing its children: 'content', 'id', and 'internal'. The 'content' field is checked. The 'parent' field is checked. The 'contentful_id' field is checked. The 'createdAt' field is checked. The 'eyecatch' field is checked.

On the right, the 'GraphiQL' tab is active, showing a query for a 'contentfulBlogPost' node. The query is:

```
query MyQuery {
  contentfulBlogPost {
    title
    content {
      json
    }
  }
}
```

The results pane shows the JSON response for the query. The response is:

```
{
  "data": {
    "contentfulBlogPost": {
      "title": "毎日のフルーツで爽やかさを加えて",
      "content": {
        "json": {
          "data": {},
          "content": [
            {
              "data": {},
              "content": [
                {
                  "data": {},
                  "marks": [],
                  "value": "フルーツには適度な甘みと酸味と爽やかさがあって、毎日でも飽きません。パンやヨーグルトとの相性もばっちりです。朝食にたくさんフルーツを取り入れてみると、いつの間にかたくさんフルーツを食べようになりました。"
                },
                {
                  "nodeType": "text"
                }
              ],
              "nodeType": "paragraph"
            },
            {
              "data": {},
              "content": [
                {
                  "data": {},
                  "marks": [],
                  "value": "フルーツの旬を活かす",
                  "nodeType": "text"
                }
              ],
              "nodeType": "paragraph"
            }
          ]
        }
      }
    }
}
```

The results pane also contains a note in Japanese: 'と爽やかさがあって、毎日でも飽きません。パンやヨーグルトとの相性もばっちりです。朝食にたくさんフルーツを取り入れてみると、いつの間にかたくさんフルーツを食べようになりました。'.



Explorer X GraphQL  Prettyfy History Explorer Code Exporter 

microcmsBlog

- blogId:
- category:
- children:
- content:
- createdAt:
- eyecatch:
- id:
- internal:
- parent:
- publishDate:
- slug:
- title:
- updatedAt:

blogId

category

children

content

createdAt

eyecatch

id

internal

parent

publishDate

slug

title

updatedAt

microcmsCategory

site

siteBuildMetadata

sitePage

sitePlugin

```
query MyQuery {  
  microcmsBlog {  
    title  
    content  
  }  
}
```

```
query MyQuery {  
  microcmsBlog {  
    title  
    content  
  }  
}  
  
{  
  "data": {  
    "microcmsBlog": {  
      "title": "毎日のフルーツで爽やかさを加えます",  
      "content": "<p>フルーツには適度な甘みと酸味で爽やかさがあって、毎日食べても飽きません。パンやヨーグルトとの相性もはっちゃります。朝食にたくさんのフルーツを取り入れてみると、いつの間にかたくさんの方々が野菜を食べるようになっていました。<br></p><h3 id=\"Haa8jXuV1Z1\">フルーツの旬を活かす</h3><br>野菜と同じようにフルーツの旬があります。ただ、地域によって旬の時期には違いがありますので、居住地域の旬を押さえておくのがおすすめです。<br><br><img src=\"https://images.microcms-assets.io/protected/ap-northeast-1:bef0665c-97b7-4807-bfac-f6dce328db3c/service/essentials-blog/media/season.jpg\" alt=<br><br>早い時期に手に入る果物はラズベリーとブルーベリーです。これらの果物は生で食べることも、乾燥させてデザートとして調理することができます。少し甘くて酸味があります。また、ベリーから作るブルーベリーウィンもあります。次はイチゴですが、今年の前半はイチゴジャムが新鮮です。火を通さずそのまま食べるのがおいしいです。<br><br>パンとの組み合わせもいろいろ試してみましたが、特にライ麦パンがとても気に入りました！クリームとハチミツ、塩少々、バターシャキに果物を加えるとベロッと食べられます。</p>"  
}
```



contentfulCategory と microcmsCategory

続いて、contentfulCategory と microcmsCategory です。こちらも構造は揃えていますので、カテゴリーに関する情報に違いはありません。

ただし、contentfulCategory ではそのカテゴリーに含まれる記事が参照できるのに対して、microcmsCategory では記事の情報を得ることはできません。そのため、カテゴリーに属した記事の数は自分で用意することになります。



Explorer X GraphiQL  Prettify History Explorer Code Exporter  Docs

```

contentfulCategory
  > blogpost:
  > category:
  > categorySlug:
  > children:
  > contentful_id:
  > createdat:
  > id:
  > internal:
  > node_locale:
  > parent:
  > spaceld:
  > sys:
  > updatedAt:
  > blogpost
    > category
    > childContentfulBlogPostContentRi
      < children
      < content
      < contentful_id
      < createdAt
      < eyecatch
      < id
      > internal
      < node_locale
      < parent
      < publishDate
      < slug
      < spaceId
      > sys
      < title
      < updatedAt

```

```

query MyQuery {
  contentfulCategory {
    category
    categorySlug
    blogpost {
      title
    }
  }
}

{
  "data": [
    {
      "contentfulCategory": {
        "category": "穀物",
        "categorySlug": "grain",
        "blogpost": [
          {
            "title": "毎日のフルーツで爽やかさを加えて"
          },
          {
            "title": "お手軽サンドイッチ"
          },
          {
            "title": "どうしても甘いものが欲しくなるとき"
          },
          {
            "title": "穀物のバリエーション"
          },
          {
            "title": "チーズやパンといっしょに食べたい果物"
          },
          {
            "title": "甘いものとフルーツの相性"
          },
          {
            "title": "カップケーキを焼くときのちょっとしたコツ"
          }
        ]
      }
    }
  ]
}

```



Explorer X GraphiQL  Prettify History Explorer Code Exporter  Docs

```

microcmsBlog
  > microcmsCategory
    > category:
    > categoryId:
    > categorySlug:
    > children:
    > createdat:
    > id:
    > internal:
    > parent:
    > updatedAt:
    < categoryId
    < categorySlug
    < children
    < createdat
    < id
    > internal
    < parent
    < updatedAt
  > site

```

```

query MyQuery {
  microcmsCategory {
    category
    categorySlug
  }
}

{
  "data": [
    {
      "microcmsCategory": {
        "category": "飲み物",
        "categorySlug": "beverage"
      }
    }
  ]
}

```

他にも異なる部分はありますが、ひとまずこのあたりをおさえつつ対応を進めます。

microCMS

2

記事一覧

2-1 トップページ

2-2 記事一覧ページ

2-3 カテゴリーページ

Build blazing-fast websites with GatsbyJS

GatsbyJS

2-1 トップページ

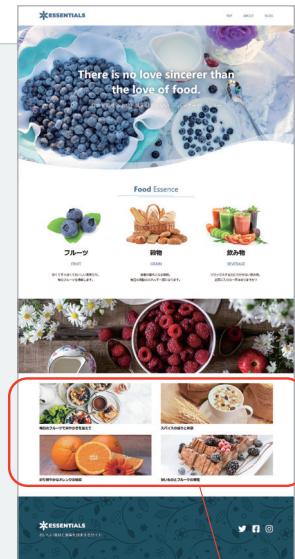
トップページ (index.js) の記事一覧を microCMS に対応させます。index.js でヘッドレス CMS のデータを利用しているのは、この部分です。

```

<section>
  <div className="container">
    <h2 className="sr-only">RECENT POSTS</h2>
    <div className="posts">
      {data.allContentfulBlogPost.edges.map(({ node }) => (
        <article className="post" key={node.id}>
          <Link to={`/blog/post/${node.slug}`}>
            <figure>
              <Img
                fluid={node.eyecatch.fluid}
                alt={node.eyecatch.description}
                style={{ height: "100%" }}
              />
            </figure>
            <h3>{node.title}</h3>
          </Link>
        </article>
      ))}
    </div>
  </div>
</section>

```

src/pages/index.js



トップページの記事一覧。

id、**slug**、**title** はそのまま置き換えるべき問題はありません。しかし、gatsby-image はそのままでは使えません。

Contentful では管理している画像の gatsby-image 用のデータを直接取得することができます。しかし、そうした機能がない場合にはリモートの画像をダウンロードした上で変換し、gatsby-image 用のデータを GraphQL から利用できる形で用意することになります。

この一連の処理をしてくれるのが、`createRemoteFileNode` です。ただし、ビルドのたびにこの作業が行われるため、処理が重くなるという問題があります。

また、microCMS では高機能な画像処理 API である imgix が利用できます。そこで、imgix を活用する形で対応を進めていきます (gatsby-image を使いたい場合は、P.54 の Appendix を参照してください)。



imgix
<https://www.imgix.com/>

クエリを追加する

まずは Contentful のクエリに倣って、microCMS のクエリを追加します。

```

...
export const query = graphql` 
query {
  ...
  allContentfulBlogPost(
    sort: { order: DESC, fields: publishDate }
    skip: 0
    limit: 4
  ) {
    edges {
      node {
        title
        id
        slug
        eyecatch {
          fluid(maxWidth: 573) {
            ...GatsbyContentfulFluid_withWebp
          }
          description
        }
      }
    }
  }
  allMicrocmsBlog(
    sort: { order: DESC, fields: publishDate }
    skip: 0
    limit: 4
  ) {
    edges {
      node {
        title
        id
        slug
        eyecatch {
          url
        }
      }
    }
  }
}
`
```

Contentfulのクエリ。

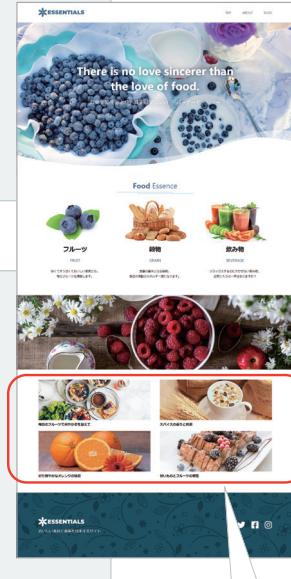
microCMSのクエリ。

src/pages/index.js

microCMSからのデータに置き換える

allContentfulBlogPost を allMicrocmsBlog に置き換え、gatsby-image を で書き換えます。description がないため、alt は空にしています。

```
<section>
  <div className="container">
    <h2 className="sr-only">RECENT POSTS</h2>
    <div className="posts">
      {data.allContentfulBlogPost.edges.map(({ node }) => (
        <article className="post" key={node.id}>
          <Link to={`/blog/post/${node.slug}`}>
            <figure>
              <Img
                fluid={node.eyecatch.fluid}
                alt={node.eyecatch.description}
                style={{ height: "100%" }}
              />
            </figure>
            <h3>{node.title}</h3>
          </Link>
        </article>
      ))}
    </div>
  </div>
</section>
```



```
<section>
  <div className="container">
    <h2 className="sr-only">RECENT POSTS</h2>
    <div className="posts">
      {data.allContentfulBlogPost.edges.map(({ node }) => (
        <article className="post" key={node.id}>
          <Link to={`/blog/post/${node.slug}`}>
            <figure>
              <img src={node.eyecatch.url} alt="" />
            </figure>
            <h3>{node.title}</h3>
          </Link>
        </article>
      ))}
    </div>
  </div>
</section>
```

src/pages/index.js

```
<figure>

</figure>
```

これで、microCMSからのデータへの置き換えが完了です。ただし、画像はオリジナルサイズ（横幅 1600px）のものを src 属性で読み込んでいるだけの状態です。

レスポンシブイメージの設定を行う

レスポンシブイメージの設定を行い、解像度などに応じて最適なサイズの画像で表示します。microCMS では imgix の API が利用できますので、react-imgix を利用します。

```
$ yarn add react-imgix
```

react-imgix をインストールしたら Imgix を import し、先程の を次のように書き換えます。これで、レスポンシブイメージの設定が出力されるようになります。

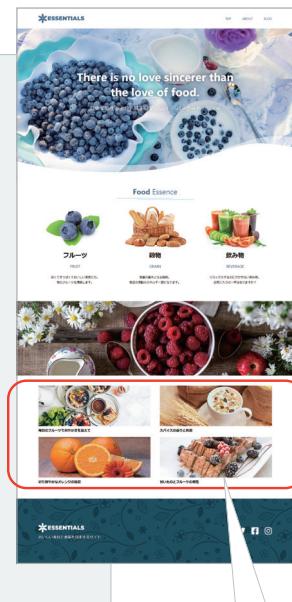
```
...
import Imgix from "react-imgix"

export default ({ data }) => (
  ...
  <section>
    <div className="container">
      <h2 className="sr-only">RECENT POSTS</h2>
      <div className="posts">
        {data.allContentfulBlogPost.edges.map(({ node }) => (
          <article className="post" key={node.id}>
            <Link to={`/blog/post/${node.slug}`}>
              <figure>
                <Imgix
                  src={node.eyecatch.url}
                  sizes="(max-width: 573px) 100vw, 573px"
                  htmlAttributes={{
                    alt: "",
                  }}
                />
              </figure>
              <h3>{node.title}</h3>
            </Link>
          </article>
        ))}
      </div>
    </div>
  </section>
)
```

src/pages/index.js

react-imgix

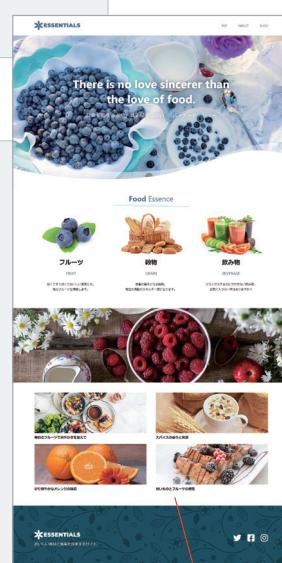
<https://github.com/imgix/react-imgix>



```
<figure>
  
</figure>
```

最後に Contentful へのクエリを削除して、トップページ (index.js) の対応は完了です。

```
...
export const query = graphql`  
query {  
  ...
  pattern: file(relativePath: { eq: "pattern.jpg" }) {  
    childImageSharp {  
      fluid(maxWidth: 1920, quality: 90) {  
        ...GatsbyImageSharpFluid_withWebp  
      }
    }
  }
  allContentfulBlogPost(
    sort: { order: DESC, fields: publishDate }
    skip: 0
    limit: 4
  ) {
    ...
  }
  allMicrocmsBlog(
    sort: { order: DESC, fields: publishDate }
    skip: 0
    limit: 4
  ) {
    ...
  }
}
`
```



```
...
export const query = graphql`  
query {  
  ...
  pattern: file(relativePath: { eq: "pattern.jpg" }) {  
    childImageSharp {  
      fluid(maxWidth: 1920, quality: 90) {  
        ...GatsbyImageSharpFluid_withWebp  
      }
    }
  }
  allMicrocmsBlog(
    sort: { order: DESC, fields: publishDate }
    skip: 0
    limit: 4
  ) {
    ...
  }
}
`
```

Contentfulのクエリを削除しても表示には影響しません。

src/pages/index.js

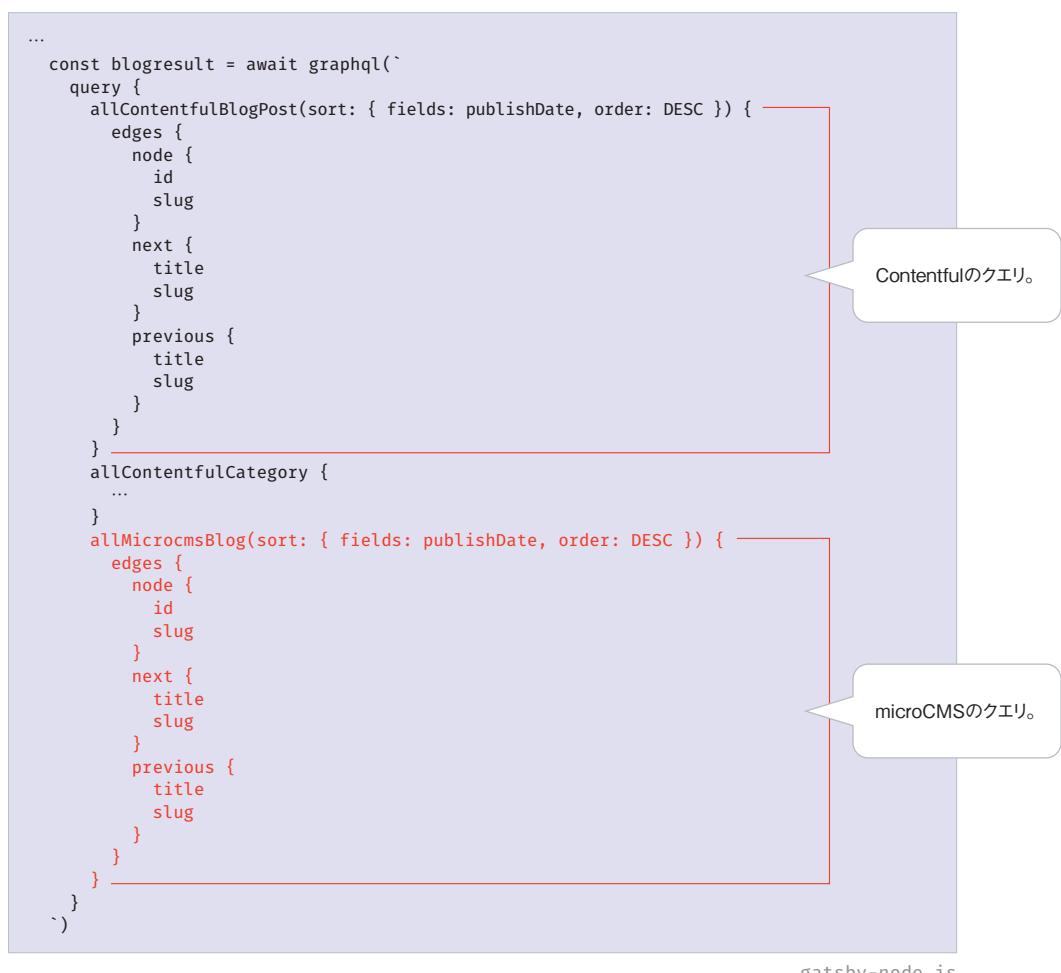
2-2 記事一覧ページ

記事一覧ページ（blog-template.js）を microCMS に対応させます。

クエリを追加する

まずは gatsby-node.js を開き、context で blog-template.js へ送るデータのクエリを追加します。

```
...
const blogresult = await graphql(`  
  query {  
    allContentfulBlogPost(sort: { fields: publishDate, order: DESC }) {  
      edges {  
        node {  
          id  
          slug  
        }  
        next {  
          title  
          slug  
        }  
        previous {  
          title  
          slug  
        }  
      }  
    }  
    allContentfulCategory {  
      ...  
    }  
    allMicrocmsBlog(sort: { fields: publishDate, order: DESC }) {  
      edges {  
        node {  
          id  
          slug  
        }  
        next {  
          title  
          slug  
        }  
        previous {  
          title  
          slug  
        }  
      }  
    }  
  }  
`)
```



Contentfulのクエリ。

microCMSのクエリ。

gatsby-node.js

追加したクエリに合わせて、blog-template.js を呼び出している createPage 周辺を対応させます。ここでは allContentfulBlogPost を allMicrocmsBlog に変更しています。

```
const blogPostsPerPage = 6 // 1ページに表示する記事の数
const blogPosts = blogresult.data.allContentfulBlogPost.edges.length // 記事の総数
...
```

```
const blogPostsPerPage = 6 // 1ページに表示する記事の数
const blogPosts = blogresult.data.allMicrocmsBlog.edges.length // 記事の総数
const blogPages = Math.ceil(blogPosts / blogPostsPerPage) // 記事一覧ページの総数

Array.from({ length: blogPages }).forEach((_, i) => {
  createPage({
    path: i === 0 ? `/blog/` : `/blog/${i + 1}/`,
    component: path.resolve("./src/templates/blog-template.js"),
    context: {
      ...
    },
  })
})
```

gatsby-node.js

microCMSからのデータに置き換える

blog-template.js は index.js と同様に microCMS からデータを取得するクエリに書き換えます。

```
export const query = graphql`  
query($skip: Int!, $limit: Int!) {  
  allContentfulBlogPost {  
    sort: { order: DESC, fields: publishDate }  
    skip: $skip  
    limit: $limit  
  } {  
    edges {  
      node {  
        title  
        id  
        slug  
        eyecatch {  
          fluid(maxWidth: 500) {  
            ...GatsbyContentfulFluid_withWebp  
          }  
          description  
        }  
      }  
    }  
  }  
}
```

```
export const query = graphql`  
query($skip: Int!, $limit: Int!) {  
  allMicrocmsBlog {  
    sort: { order: DESC, fields: publishDate }  
    skip: $skip  
    limit: $limit  
  } {  
    edges {  
      node {  
        title  
        id  
        slug  
        eyecatch {  
          url  
        }  
      }  
    }  
  }  
}
```

src/templates/blog-template.js

そして、gatsby-image を利用していた部分を Imgix を利用する形に書き換えます。

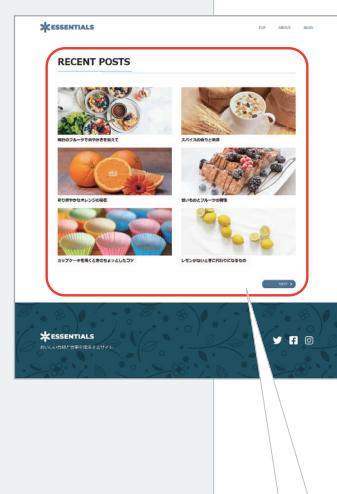
```
export default ({ data, location, pageContext }) => (
  ...
  <section className="content bloglist">
    <div className="container">
      <h1 className="bar">RECENT POSTS</h1>
      <div className="posts">
        {data.allContentfulBlogPost.edges.map(({ node }) => (
          <article className="post" key={node.id}>
            <Link to={`/blog/post/${node.slug}`}>
              <figure>
                <Img
                  fluid={node.eyecatch.fluid}
                  alt={node.eyecatch.description}
                  style={{ height: "100%" }}
                />
              </figure>
              <h3>{node.title}</h3>
            </Link>
          </article>
        ))}
      </div>
    </div>
  </section>
)
```

```
import Img from "gatsby-image"
...
import Imgix from "react-imgix"

export default ({ data, location, pageContext }) => (
  ...
  <section className="content bloglist">
    <div className="container">
      <h1 className="bar">RECENT POSTS</h1>
      <div className="posts">
        {data.allMicrocmsBlog.edges.map(({ node }) => (
          <article className="post" key={node.id}>
            <Link to={`/blog/post/${node.slug}`}>
              <figure>
                <Imgix
                  src={node.eyecatch.url}
                  sizes="(max-width: 500px) 100vw, 500px"
                  htmlAttributes={{
                    alt: "",
                  }}
                />
              </figure>
              <h3>{node.title}</h3>
            </Link>
          </article>
        ))}
      </div>
    </div>
  </section>
)
```

src/templates/blog-template.js

他にImgを使用している箇所はないので削除。



```
<figure>

</figure>
```

microCMSからのデータに置き換わります。

以上で、対応は完了です。

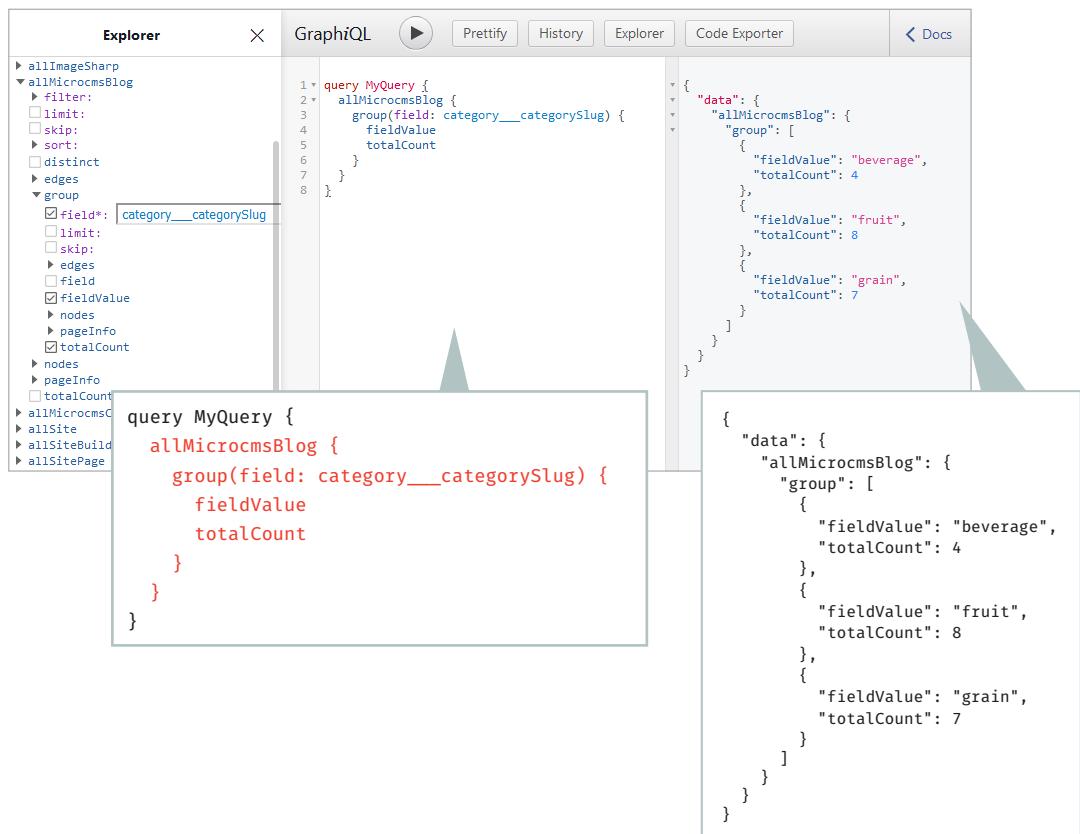
2-3 カテゴリーページ

カテゴリーページ (cat-template.js) を microCMS に対応させます。

クエリを追加する

カテゴリーページで問題となるのが、各カテゴリーに属する記事の数です。取得したデータからカウントしてもよいのですが、ここではGraphQLのgroupを利用します。

GraphiQLで次のようなクエリを作成すると、各カテゴリーに属するポストの数が取得できます。



The screenshot shows the GraphiQL interface with the following components:

- Explorer** sidebar: Shows available queries and types, including `allMicrocmsBlog` and its `group` field.
- GraphiQL** input area: Displays the GraphQL query:

```
query MyQuery {
  allMicrocmsBlog {
    group(field: category__categorySlug) {
      fieldValue
      totalCount
    }
  }
}
```
- Results** area: Shows the JSON response for the query:

```
{
  "data": {
    "allMicrocmsBlog": [
      {
        "group": [
          {
            "fieldValue": "beverage",
            "totalCount": 4
          },
          {
            "fieldValue": "fruit",
            "totalCount": 8
          },
          {
            "fieldValue": "grain",
            "totalCount": 7
          }
        ]
      }
    ]
  }
}
```

そこで、このクエリと、`allMicrocmsCategory` からデータを取得するためのクエリを `gatsby-node.js` に追加します。

```
...
const blogresult = await graphql(`

query {
  ...
  allMicrocmsBlog(sort: { fields: publishDate, order: DESC }) {
    edges {
      ...
    }
    group(field: category__categorySlug) {
      totalCount
      fieldValue
    }
  }
  allMicrocmsCategory {
    nodes {
      category
      categorySlug
      categoryId
    }
  }
}
`)
```

gatsby-node.js

そして、このクエリから得られるデータを利用して `createPage` 周辺を対応させます。

```
blogresult.data.allContentfulCategory.edges.forEach(({ node }) => {
  const catPostsPerPage = 6 // 1ページに表示する記事の数
  const catPosts = node.blogpost.length // カテゴリーに属した記事の総数
  const catPages = Math.ceil(catPosts / catPostsPerPage) // カテゴリーページの総数

  Array.from({ length: catPages }).forEach(_, i) => {
    createPage({
      path:
        i === 0
          ? `/cat/${node.categorySlug}/`
          : `/cat/${node.categorySlug}/${i + 1}/`,
      component: path.resolve(`./src/templates/cat-template.js`),
      context: {
        catid: node.id,
        catname: node.category,
        catslug: node.categorySlug,
        skip: catPostsPerPage * i,
        limit: catPostsPerPage,
        currentPage: i + 1, // 現在のページ番号
        isFirst: i + 1 === 1, // 最初のページ
        isLast: i + 1 === catPages, // 最後のページ
      },
    })
  }
})
```

```
blogresult.data.allMicrocmsBlog.group.forEach(node => {
  const catPostsPerPage = 6 // 1ページに表示する記事の数
  const catPosts = node.totalCount // カテゴリーに属した記事の総数
  const catPages = Math.ceil(catPosts / catPostsPerPage) // カテゴリーページの総数

  Array.from({ length: catPages }).forEach((_, i) => {
    createPage({
      path:
        i === 0
          ? `/cat/${node.fieldValue}/`
          : `/cat/${node.fieldValue}/${i + 1}/`,
      component: path.resolve(`./src/templates/cat-template.js`),
      context: {
        catid: blogresult.data.allMicrocmsCategory.nodes.find(
          n => n.categorySlug === node.fieldValue
        ).categoryId,
        catname: blogresult.data.allMicrocmsCategory.nodes.find(
          n => n.categorySlug === node.fieldValue
        ).category,
        catslug: node.fieldValue,
        skip: catPostsPerPage * i,
        limit: catPostsPerPage,
        currentPage: i + 1, // 現在のページ番号
        isFirst: i + 1 === 1, // 最初のページ
        isLast: i + 1 === catPages, // 最後のページ
      },
    })
  })
})
```

gatsby-node.js

microCMSからのデータに置き換える

cat-template.js のクエリも、microCMS へのクエリに書き換えます。

```
export const query = graphql`  
  query($catid: String!, $skip: Int!, $limit: Int!) {  
    allContentfulBlogPost(  
      sort: { order: DESC, fields: publishDate }  
      ...  
    ) {  
      edges {  
        node {  
          title  
          id  
          slug  
          eyecatch {  
            fluid(maxWidth: 500) {  
              ...GatsbyContentfulFluid_withWebp  
            }  
            description  
          }  
        }  
      }  
    }  
  }`
```

```
export const query = gql`  
  query($catid: String!, $skip: Int!, $limit: Int!) {  
    allMicrocmsBlog(  
      sort: { order: DESC, fields: publishDate }  
      ...  
    ) {  
      edges {  
        node {  
          title  
          id  
          slug  
          eyecatch {  
            url  
          }  
        }  
      }  
    }  
  }`
```

src/templates/cat-template.js

gatsby-image を利用していた部分を Imgix を利用する形に書き換えます。

```
export default ({ data, location, pageContext }) => (
  ...
  <section className="content bloglist">
    <div className="container">
      <h1 className="bar">CATEGORY: {pageContext.catname}</h1>
      <div className="posts">
        {data.allContentfulBlogPost.edges.map(({ node }) => (
          <article className="post" key={node.id}>
            <Link to={`/blog/post/${node.slug}/`}>
              <figure>
                <Img
                  fluid={node.eyecatch.fluid}
                  alt={node.eyecatch.description}
                  style={{ height: "100%" }}
                />
              </figure>
              <h3>{node.title}</h3>
            </Link>
          </article>
        ))}
      </div>
    </div>
  </section>
)
```

↓

```
import Img from "gatsby-image"
...
import Imgix from "react-imgix"

export default ({ data, location, pageContext }) => (
  ...
  <section className="content bloglist">
    <div className="container">
      <h1 className="bar">CATEGORY: {pageContext.catname}</h1>
      <div className="posts">
        {data.allMicrocmsBlog.edges.map(({ node }) => (
          <article className="post" key={node.id}>
            <Link to={`/blog/post/${node.slug}/`}>
              <figure>
                <Imgix
                  src={node.eyecatch.url}
                  sizes="(max-width: 500px) 100vw, 500px"
                  htmlAttributes={{
                    alt: "",
                  }}
                />
              </figure>
              <h3>{node.title}</h3>
            </Link>
          </article>
        ))}
      </div>
    </div>
  </section>
)
```

src/templates/cat-template.js

他にImgを使用している箇所はないので削除。



```
<figure>

</figure>
```

以上で、対応は完了です。

microCMSからのデータに置き換わります。

36

カテゴリー ページ 2-3

microCMS

3

記事

- 3-1 記事ページ
- 3-2 リッチテキストの処理
- 3-3 アイキャッチ画像の表示スペースを確保してレイアウトシフトを防ぐ
- 3-4 OGP画像の横幅と高さを指定する
- 3-5 仕上げ

Build blazing-fast websites with GatsbyJS

GatsbyJS

3-1 記事ページ

記事ページ (blogpost-template.js)
を microCMS に対応させます。



microCMSからのデータに置き換える

まずは、blogpost-template.js を呼び出しているgatsby-node.js の createPage 周辺を対応させます。ここでは allContentfulBlogPost を allMicrocmsBlog に変更しています。

```
blogresult.data.allContentfulBlogPost.edges.forEach(
  ({ node, next, previous }) => {
  ...

```

```
blogresult.data.allMicrocmsBlog.edges.forEach(
  ({ node, next, previous }) => {
    createPage({
      path: `/blog/post/${node.slug}/`,
      component: path.resolve(`./src/templates/blogpost-template.js`),
      context: {
        id: node.id,
        next,
        previous,
      },
    })
  }
)
```

gatsby-node.js

blogpost-template.js を開き、microCMS へのクエリに書き換えます。

```
export const query = graphql`  
  query($id: String!) {  
    contentfulBlogPost(id: { eq: $id }) {  
      title  
      publishDateJP: publishDate(formatString:  
        "YYYY 年 MM 月 DD 日")  
      publishDate  
      category {  
        category  
        categorySlug  
        id  
      }  
      eyecatch {  
        fluid(maxWidth: 1600) {  
          ...GatsbyContentfulFluid_withWebp  
        }  
      }  
      description  
      file {  
        details {  
          image {  
            width  
            height  
          }  
        }  
        url  
      }  
      content {  
        json  
      }  
    }  
  }`
```

```
export const query = graphql`  
  query($id: String!) {  
    microcmsBlog(id: { eq: $id }) {  
      title  
      publishDateJP: publishDate(formatString:  
        "YYYY 年 MM 月 DD 日")  
      publishDate  
      category {  
        category  
        categorySlug  
        id  
      }  
      eyecatch {  
        url  
      }  
      content  
    }  
  }`
```

src/templates/blogpost-template.js

あとは、上から順に対応していきます。

メタデータ

メタデータ <SEO /> の中身を microcmsBlog から取得したデータに書き換えます。
ただし、pagedesc (説明) はリッチテキストの HTML から用意しなければなりません。ここでは html-to-text を利用してテキストを用意します。

```
$ yarn add html-to-text
```

html-to-text

<https://github.com/werk85/node-html-to-text>

インストールしたら `htmlToText` を `import` し、`pagedesc` を次のように書き換えます。また、この段階ではアイキャッチ画像の横幅と高さが取得できていませんので、OGP の `pageimgw` と `pageimgh` をコメントアウトしておきます。

```
export default ({ data, pageContext, location }) => (
  <Layout>
    <SEO
      pagetitle={data.contentfulBlogPost.title}
      pagedesc={`${documentToPlainTextString(
        data.contentfulBlogPost.content.json
      ).slice(0, 70)}...`}
      pagepath={location.pathname}
      blogimg={`https:${data.contentfulBlogPost.eyecatch.file.url}`}
      pageimgw={data.contentfulBlogPost.eyecatch.file.details.image.width}
      pageimgh={data.contentfulBlogPost.eyecatch.file.details.image.height}
    />
```



```
import htmlToText from "html-to-text"
...
export default ({ data, pageContext, location }) => (
  <Layout>
    <SEO
      pagetitle={data.microcmsBlog.title}
      pagedesc={`${htmlToText
        .fromString(data.microcmsBlog.content, {
          ignoreImage: true,
          ignoreHref: true,
        })
        .slice(0, 70)}...`}
      pagepath={location.pathname}
      blogimg={data.microcmsBlog.eyecatch.url}
      // pageimgw={data.contentfulBlogPost.eyecatch.file.details.image.width}
      // pageimgh={data.contentfulBlogPost.eyecatch.file.details.image.height}
    />
```

src/templates/blogpost-template.js

アイキャッチ画像

アイキャッチ画像は `Imgix` を利用する形に書き換えます。

```
<div className="eyecatch">
  <figure>
    <Img
      fluid={data.contentfulBlogPost.eyecatch.fluid}
      alt={data.contentfulBlogPost.eyecatch.description}
    />
  </figure>
</div>
```



```

import Imgix from "react-imgix"
...
<div className="eyecatch">
  <figure>
    <Imgix
      src={data.microcmsBlog.eyecatch.url}
      sizes="(max-width: 1600px) 100vw, 1600px"
      htmlAttributes={{
        alt: "",
      }}
    />
  </figure>
</div>

```

src/templates/blogpost-template.js

記事

<article> の部分は、記事のタイトル、投稿日、カテゴリーを microcmsBlog から取得したデータを使うように書き換えます。

記事の本文 <div className="postbody"> は、取得した HTML をそのまま表示するように、dangerouslySetInnerHTML で書き換えています。

```

<article className="content">
  <div className="container">
    <h1 className="bar">{data.contentfulBlogPost.title}</h1>
    <aside className="info">
      <time dateTime={data.contentfulBlogPost.publishDate}>
        <FontAwesomeIcon icon={faClock} />
        {data.contentfulBlogPost.publishDateJP}
      </time>
      <div className="cat">
        <FontAwesomeIcon icon={faFolderOpen} />
      <ul>
        {data.contentfulBlogPost.category.map(cat => (
          <li className={cat.categorySlug} key={cat.id}>
            <Link to={`/cat/${cat.categorySlug}/`}>{cat.category}</Link>
          </li>
        ))}
      </ul>
    </div>
  </aside>
  <div className="postbody">
    {documentToReactComponents(
      data.contentfulBlogPost.content.json,
      options
    )}
  </div>
  ...

```

```

<article className="content">
  <div className="container">
    <h1 className="bar">{data.microcmsBlog.title}</h1>
    <aside className="info">
      <time dateTime={data.microcmsBlog.publishDate}>
        <FontAwesomeIcon icon={faClock} />
        {data.microcmsBlog.publishDateJP}
      </time>
    </aside>
    <div className="cat">
      <FontAwesomeIcon icon={faFolderOpen} />
      <ul>
        {data.microcmsBlog.category.map(cat => (
          <li className={cat.categorySlug} key={cat.id}>
            <Link to={`/cat/${cat.categorySlug}/`}>{cat.category}</Link>
          </li>
        )))
      </ul>
    </div>
  </div>
  <div
    className="postbody"
    dangerouslySetInnerHTML={{ __html: data.microcmsBlog.content }}
  ></div>
  ...

```

src/templates/blogpost-template.js

これで、ページが表示されるようになります。



```

<figure>

</figure>

```

```

<article className="content">
  <div className="container">
    <h1 className="bar">毎日のフルーツで爽やかさを加えて </h1>
    ...
    <div className="postbody">
      <p>フルーツには適度な甘みと酸味と爽やかさがあつて、… </p>
      <h2 id="hkAA8jXWJZ">フルーツの旬を活かす </h2>
      <p>…</p>
       ...
      </p>
    </div>
    ...

```

3-2 リッチテキストの処理

リッチテキストの HTML をそのまま表示するだけでよければ **3-1** までの設定で十分です。しかし、見出しにアイコンを付けたり、記事中の画像をレスポンシブイメージにする場合には、Contentful のときと同様に htmlAST に変換した上で処理するのが簡単です。

そこで、unified を利用した処理を追加します。
以下のようにライブラリをインストールしてください。

unified

<https://github.com/unifiedjs/unified>

rehype

<https://github.com/rehypejs/rehype>

```
$ yarn add unified rehype-parse rehype-react
```

インポートして、htmlAST への変換処理を追加します。HTML をパースして htmlAST にするシンプルなものですですが、fragment: true を忘れる `<html>`、`<head>`、`<body>` の付いた htmlAST に変換されますので注意が必要です。

```
import unified from "unified"
import parse from "rehype-parse"
import rehypeReact from "rehype-react"
...
export default ({ data, pageContext, location }) => {
  const htmlAst = unified()
    .use(parse, { fragment: true })
    .parse(data.microcmsBlog.content)
```

```
  return (
    <Layout>
    ...
  </Layout>
)
}
```

注意

注意

src/templates/blogpost-template.js

続いて、htmlAST をレンダリングする処理を用意します。

```
...
const renderAst = new rehypeReact({
  createElement: React.createElement,
  Fragment: React.Fragment,
  components: {},
}).Compiler

export default ({ data, pageContext, location }) => {
  const htmlAst = unified()
    .use(parse, { fragment: true })
    .parse(data.microcmsBlog.content)
...
...
```

src/templates/blogpost-template.js

そして、その処理を利用して表示します。この段階では出力される HTML は変化しません。

```
<div
  className="postbody"
  dangerouslySetInnerHTML={{ __html: data.microcmsBlog.content }}
></div>
```



```
<div className="postbody">{renderAst(htmlAst)}</div>
```

src/templates/blogpost-template.js



```
<div className="postbody">
  <p>フルーツには適度な甘みと酸味と爽やかさがあって、…</p>
  <h2 id="hkAA8jXWVJZ">フルーツの旬を活かす </h2>
  <p>…</p>
   …
</div>
```

あとは、Contentful のときと同様に、options を参考にして `<h2>` と `` への処理を追加します。最後に、options や Contentful 関連の設定を削除して、リッチテキストの設定は完了です。

```

import Img from "gatsby-image" 削除
...
import { documentToReactComponents } from "@contentful/rich-text-react-renderer"
import { BLOCKS } from "@contentful/rich-text-types"
import useContentfulImage from "../utils/useContentfulImage"
import { documentToPlainTextString } from "@contentful/rich-text-plain-text-renderer"
...
const options = { 削除
  ...
}

const renderAst = newrehypeReact({
  createElement: React.createElement,
  Fragment: React.Fragment,
  components: {
    h2: props => {
      return (
        <h2>
          <FontAwesomeIcon icon={faCheckSquare} />
          {props.children}
        </h2>
      )
    },
    img: props => {
      return (
        <Imgix
          src={props.src}
          sizes="(max-width: 785px) 100vw, 785px"
          htmlAttributes={{
            alt: props.alt,
          }}
        />
      )
    },
  },
})

```

リッチテキスト内の見出し `<h2>` に Font Awesome のアイコンを追加。

リッチテキスト内の画像 `` をレスポンシブイメージに設定。

src/templates/blogpost-template.js

アイコンが表示されます。



```

<div class="postbody">
  <p>フルーツには適度な甘みと酸味と爽やかさがあって、毎日でも飽きません。一口でヨゴルトとの相性もよさそうです。朝食にたっぷりのフルーツを取り入れてみると、いつの間にかきのう食べるよくなっています。</p>
  <h2><img alt="check-square" /></h2>
  <p>フルーツの旬を活かす</p>
  
</div>

```

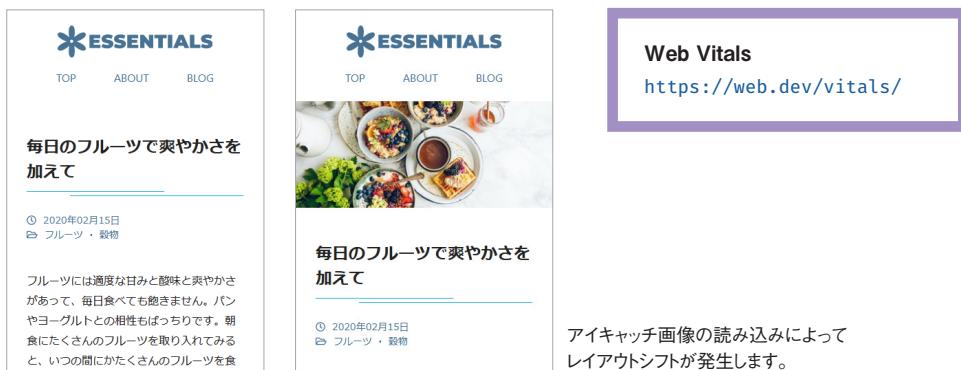
レスポンシブイメージになります。

3-3

アイキャッチ画像の表示スペースを確保してレイアウトシフトを防ぐ

問題なく記事ページが表示できるようになりました。しかし、アイキャッチ画像が読み込まれるタイミングでレイアウトがずれるのが気になります。

この現象はレイアウトシフトと呼ばれ、サイトの健全性を示す Google の新しい指標 Web Vitals では、「CLS (Cumulative Layout Shift)」として計測されます。CLS ではできるだけレイアウトが動かないようにすることが求められます。



レイアウトシフトを防ぐためには、画像の表示スペースを確保しておく必要があります。Safari を含めた主要ブラウザで表示スペースを確保するため、ここでは padding を使ったテクニックで対応していきます。これは gatsby-image でも使用されています。

ただし、このテクニックを使うためには画像の横幅と高さの値が必要です。

アイキャッチ画像の横幅と高さを取得する

microCMS では、画像の URL に「?fm=json」を追加することで画像の情報を取得することができます。たとえば、次の画像で情報を取得してみると、横幅は「PixelWidth」で、高さは「PixelHeight」で確認できることがわかります。



<https://images.microcms-assets.io/protected/ap-northeast-1:b6f0665c-97b7-4807-bfac-f6dce328dbc3/service/essentials-blog/media/everyday.jpg?fm=json>

画像のURLに「?fm=json」を付けてアクセス。

```
{  
  "Exif": {  
    "PixelXDimension": 1600,  
    "DateTimeDigitized": "2020:02:12 22:50:18",  
    "ExifVersion": [  
      2,  
      3,  
      1  
    ],  
    "PixelYDimension": 661,  
    "ColorSpace": 65535  
  },  
  "Orientation": 1,  
  "Content-Type": "image/jpeg",  
  "Output": {},  
  "DPIWidth": 72,  
  "Content-Length": "212893",  
  "IPTC": {  
    "DigitalCreationDate": "20200212",  
    "DigitalCreationTime": "225018+0900"  
  },  
  "PixelHeight": 661,  
  "Depth": 8,  
  "TIFF": {  
    "ResolutionUnit": 2,  
    "DateTime": "2020:02:16 10:30:07",  
    "Orientation": 1,  
    "YResolution": 72,  
    "Software": "Adobe Photoshop CC 2019 (Windows)",  
    "XResolution": 72,  
    "PhotometricInterpretation": 2  
  },  
  "PixelWidth": 1600,  
  "ColorModel": "RGB",  
  "DPIHeight": 72  
}
```

これをを利用してノードを追加するため、axios をインストールします。

axios

<https://github.com/axios/axios>

```
$ yarn add axios
```

Gatsby の公式チュートリアルの 7 や書籍の Appendix D などを参考に、ノードを追加する処理を `gatsby-node.js` に追加します。アイキャッチ画像の情報を取得し、`createNodeField` を使ってノードを追加するというシンプルなものです。

```

const path = require("path")
const axios = require("axios")

exports.createPages = async ({ graphql, actions, reporter }) => {
  const { createPage } = actions
  ...
}

exports.onCreateNode = async ({ node, actions }) => {
  const { createNodeField } = actions

  if (node.internal.type === `MicrocmsBlog`) {
    const results = await axios.get(`${node.eyecatch.url}?fm=json`)
    const { data } = results

    createNodeField({
      node,
      name: "width",
      value: data.PixelWidth,
    })

    createNodeField({
      node,
      name: "height",
      value: data.PixelHeight,
    })
  }
}

```

gatsby-node.js

開発サーバーを再起動して **GraphiQL** で確認すると、`fields` の中に `width` と `height` が追加されています。クエリを実行すると、データも問題なく取得できることが確認できます。

The screenshot shows the GraphiQL interface with the following details:

- Explorer:** Shows the schema structure. The 'fields' node under 'microcmsBlog' is highlighted with a red box.
- GraphiQL:** Shows the following query:


```
query MyQuery {
  microcmsBlog {
    eyecatch {
      url
    }
    fields {
      height
      width
    }
  }
}
```
- Results:** Shows the JSON response from the query:


```
{
  "data": {
    "microcmsBlog": {
      "eyecatch": {
        "url": "https://images.microcms-assets.io/protected/ap-northeast-1:b6f0665c-97b7-4807-bfac-f6dce328dbc3/service/essentials-blog/media/everyday.jpg"
      },
      "fields": {
        "height": 661,
        "width": 1600
      }
    }
  }
}
```

画像の横幅と高さを元に表示スペースを確保する

blogpost-template.js に戻り、クエリを追加します。

```
export const query = graphql`  
query($id: String!) {  
  microcmsBlog(id: { eq: $id }) {  
    ...  
    eyecatch {  
      url  
    }  
    fields {  
      height  
      width  
    }  
    content  
  }  
}
```

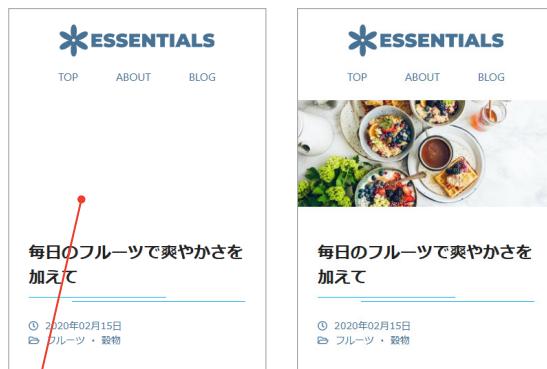
src/templates/blogpost-template.js

このデータを元に、画像の縦横比で表示スペースを確保するのに必要な padding-bottom の値を用意します。ここではアイキャッチ画像をラッパー <div class="eyecatch-wrapper"> で囲み、padding-bottom を適用しています。

```
export default ({ data, pageContext, location }) => {  
  const htmlAst = unified()  
    .use(parse, { fragment: true })  
    .parse(data.microcmsBlog.content)  
  
  const pb =  
    (data.microcmsBlog.fields.height / data.microcmsBlog.fields.width) * 100  
  
  return (  
    ...  
    <div className="eyecatch">  
      <figure>  
        <div className="eyecatch-wrapper" style={{ paddingBottom: `${pb}%` }}>  
          <Imgix  
            src={data.microcmsBlog.eyecatch.url}  
            sizes="(max-width: 1600px) 100vw, 1600px"  
            htmlAttributes={{  
              alt: "",  
            }}  
          />  
        </div>  
      </figure>  
    </div>  
  )
```

src/templates/blogpost-template.js

最後に、CSS を追加すれば完了です。



画像の読み込み中も表示スペースが確保されるようになります。

```
/* アイキャッチ画像 */
.eyecatch-wrapper {
  position: relative;
  width: 100%;
  height: 0;
  overflow: hidden;
}

.eyecatch-wrapper img {
  position: absolute;
  width: 100%;
  height: 100%;
  left: 0;
  top: 0;
}
```

src/components/layout.css



記事一覧の画像は CSS で高さを揃え、切り出して表示するように設定しています。そのため、padding のテクニックを使わなくとも表示スペースが確保されます。

記事一覧ページの
画像の表示。

3-4 OGP画像の横幅と高さを指定する



アイキャッチ画像の横幅と高さが取得できるようになりましたので、これを利用して `<SEO />` の `pageimgx` と `pageimgh` を指定します。

```
return (
  <Layout>
    <SEO
      ...
      pagepath={location.pathname}
      blogimg={data.microcmsBlog.eyecatch.url}
      // pageimgw={data.contentfulBlogPost.eyecatch.file.details.image.width}
      // pageimgh={data.contentfulBlogPost.eyecatch.file.details.image.height}
    />
```



```
return (
  <Layout>
    <SEO
      ...
      pagepath={location.pathname}
      blogimg={data.microcmsBlog.eyecatch.url}
      pageimgw={data.microcmsBlog.fields.width}
      pageimgh={data.microcmsBlog.fields.height}
    />
```

src/templates/blogpost-template.js

指定した値が OGP 画像の横幅と高さに設定されます。



```
<meta property="og:image" content="https://
images.microcms-assets.io/protected/ap-
northeast-1:b6f0665c-97b7-4807-bfac-
f6dce328dbc3/service/essentials-blog/media/
everyday.jpg" data-react-helmet="true">
<meta property="og:image:width" content="1600"
data-react-helmet="true">
<meta property="og:image:height" content="661"
data-react-helmet="true">
```

3-5 仕上げ

最後に、プロジェクト全体で必要なくなったライブラリやコンポーネント、クエリを削除していきます。まずは、gatsby-node.js に残っている Contentful へのクエリを削除します。useContentfulImage.js も忘れずに削除し、gatsby-config.js からは Contentful のソースプラグインの設定も削除します。

```
const blogresult = await graphql(`  
query {  
  allContentfulBlogPost(sort: { ... DESC }) {  
    ...  
  }  
  allContentfulCategory {  
    ...  
  }  
  allMicrocmsBlog(sort: { ... DESC }) {  
    ...  
  }  
}
```

gatsby-node.js

```
`gatsby-plugin-offline`,  
{  
  resolve: `gatsby-source-contentful`,  
  options: {  
    spaceId: process.env.CONTENTFUL_SPACE_ID,  
    accessToken: process.env.CONTENTFUL_ACCESS_TOKEN,  
    host: process.env.CONTENTFUL_HOST,  
  },  
  {  
    resolve: "gatsby-source-microcms",  
    ...  
  }  
}
```

gatsby-config.js

Gatsby

```
mysite  
└── src  
    ├── components  
    ├── images  
    ├── pages  
    ├── templates  
    └── utils  
        └── useContentfulImage.js  
└── static  
...
```

設定を削除したら、ソースプラグインをアンインストールします（アンインストールしないと、ビルトでエラーになる場合があります）。

```
$ yarn remove gatsby-source-contentful
```

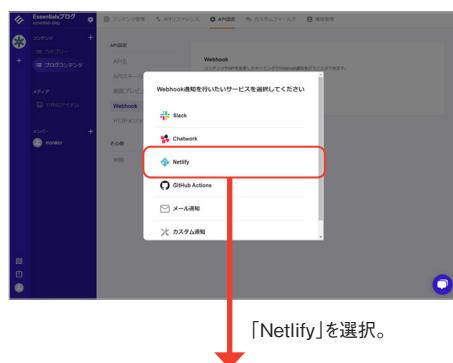
以上で、microCMS に対応する設定は完了です。

microCMSによるサイトの更新

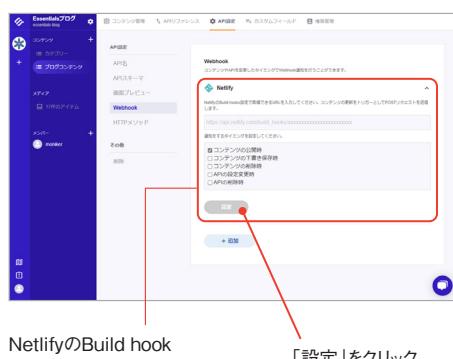
microCMS での変更を Netlify に反映させるためには、Webhook を利用します。



Webhook の設定は、API ごとに「API 設定> Webhook」で行います。新規に追加する場合は「追加」をクリックします。



サービスの選択肢が表示されますので、「Netlify」を選択します。



Netlify で取得した Build hook (ビルドフック) の URL と、Netlify でデプロイが行われるトリガー (通知するタイミング) を指定します。

Build hook についてはセットアップ PDF の 3-5 を参照してください。

「設定」をクリックすれば完了です。

microCMS

APPENDIX

`gatsby-image` を使う

Build blazing-fast websites with GatsbyJS

GatsbyJS

A

createRemoteFileNode



Lazy Load やプレースホルダといった設定を追加することを考えると、ビルト時間や転送量が増えても gatsby-image が使いたいという場合もあります。そのような場合は、gatsby-source-filesystem の Helper functions である createRemoteFileNode を利用します。

createRemoteFileNode

<https://www.gatsbyjs.org/packages/gatsby-source-filesystem/#createremotefilenode>

gatsby-node.js にインポートし、公式ドキュメントの Preprocessing External Images (<https://www.gatsbyjs.org/docs/preprocessing-external-images/>) などを参考にして、onCreateNode の中を以下のように書き換えます。microCMS からは最高画質でダウンロードするように指定しています。

```
const { createRemoteFileNode } = require(`gatsby-source-filesystem`)

...

exports.onCreateNode = async ({  
  node,  
  store,  
  cache,  
  actions,  
  createNodeId,  
}) => {  
  if (node.internal.type === `MicrocmsBlog`) {  
    const fileNode = await createRemoteFileNode({  
      url: `${node.eyecatch.url}?q=100`,  
      parentNodeId: node.id,  
      cache,  
      store,  
      createNode: actions.createNode,  
      createNodeId: createNodeId,  
    })  
  
    if (fileNode) {  
      node.eyecatchimg__NODE = fileNode.id  
    }  
  }  
}
```

gatsby-node.js

これで、eyecatchimg というノードが追加され、gatsby-image で必要なデータが取得できるようになります。アイキャッチ画像の width と height も取得できますので、<SEO /> も問題ありません。

B

gatsby-plugin-imgix



gatsby-plugin-imgix というプラグインの開発が進んでいます。プラグインをインストールし、gatsby-config.js にプラグインの設定を追加します。

gatsby-plugin-imgix

<https://www.gatsbyjs.org/packages/gatsby-plugin-imgix/>

```
$ yarn add gatsby-plugin-imgix
```

```
...  
},  
{  
  resolve: "gatsby-plugin-imgix",  
  options: {  
    domain: "images.microcms-assets.io",  
    fields: [  
      {  
        nodeType: "MicrocmsBlog",  
        fieldName: "featuredImage",  
        getUrl: node => node.eyecatch.url,  
      },  
      ],  
    },  
  ],  
}
```

gatsby-config.js

domain には、microCMS の画像の URL のドメインを指定します。あとは、対象となるノードのノードタイプと URL を指定します。

開発サーバーを再起動すると、fieldName で指定した featuredImage というフィールドが追加され、Contentful の場合と同様に microCMS のリモートファイルを使った形で gatsby-image 用のデータが取得できるようになります。

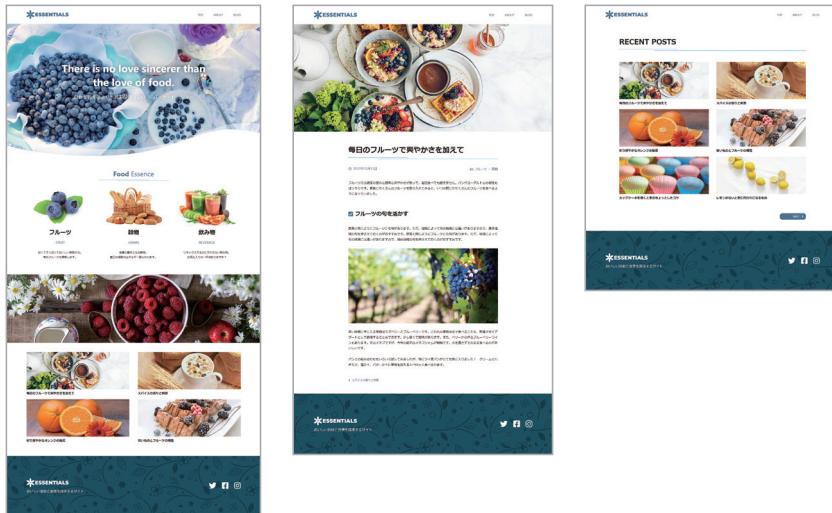
右のようなフラグメントも用意されています。
 また、クエリに imgix の API のパラメーターを指定で
 きますので、gatsby-image と imgix の API を組み
 合わせて利用することも可能です。

詳細は Gatsby の公式にあるプラグインページの情報を
 参照してください。

GatsbyImgixFixed
 GatsbyImgixFixed_noBase64
 GatsbyImgixFluid
 GatsbyImgixFluid_noBase64

表示CHECK

microCMS を利用して構築したサイトを Netlify にデプロイし、公開したものです。



本 PDF (3-5) の完成サンプル

<https://microcms-essentials-1.netlify.app/>

APPENDIX B の完成サンプル

<https://microcms-essentials-2.netlify.app/>

■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：『CSS グリッドレイアウト デザインブック』マイナビ出版刊
『HTML5&CSS3 デザイン 現場の新標準ガイド』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 1』エビスコム刊
『HTML&CSS コーディング・プラクティスブック 2』同上
『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【副読本】

microCMS 対応ガイド

2020 年 6 月 1 日 ver.1.0 発行