1.1 Astro とは

本書で解説する Astro は、いわゆる JavaScript によるフロントエンドアプリケーション開発用のフレームワークというよりは、Web サイト制作で威力を発揮するフレームワークです。本節では、その理由も含めて、Astro の特徴を紹介します。

1.1.1 Web サイト制作の問題点とコンポーネント

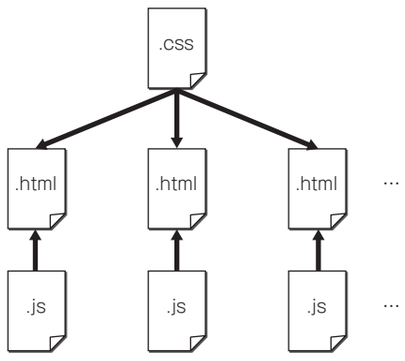
Web のフロントエンドの世界というのは、ものすごく乱暴な言い方をすれば、いわゆるフロントエンド Web アプリケーションと、いわゆる Web サイトに分かれます。もちろんこれらの境界というのは非常に曖昧ですが、前者であるアプリケーションは、ユーザとデータのやり取りを行う類のもの、例えば、ユーザが指定した地点の天気情報をネット上のどこかから取得したり、ユーザが入力した商品の注文情報をサーバに送信したり、などが該当します。これらの内容を実現しようとなると、いわゆるプログラミングを行った Web アプリケーションを作成する必要があります。このようなフロントエンド Web アプリケーションに適したフレームワークとしては、**Vue** や **React** が有名です。

一方、例えば、どこかの企業情報を表示したり、おしゃれなカフェの営業時間を紹介したりといった、あらかじめ用意されたデータを表示する類のものは、後者、すなわち、Web サイトといえます。そして、本書で紹介する Astro が活躍する場は、前者の Web アプリケーションではなく、後者の Web サイトです。

従来の Web サイト制作手法

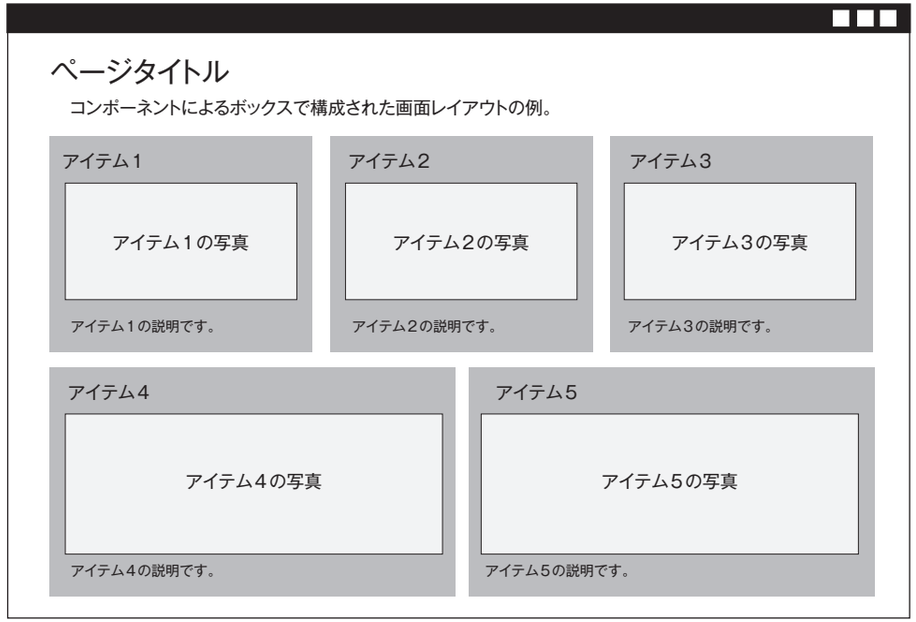
Web サイトの制作では、サイト全体で利用される css ファイルを制作し、画面ごとの html ファイルを作成、場合によっては、その html ファイルで利用する js ファイルを作成する、という方法が採用されているのが通常です（図 1-1）。

▼ **図 1-1　Web サイトのファイル構成イメージ**



もちろん、この方式で問題があるのかといえば、基本的にはありません。ただ、各ページ内、あるいは、サイト全体で繰り返し表示されているような部分があると、少し煩わしい面が出てきます。例えば、図 1-2 のように、画像とその説明をタイル状に並べて表示するような画面の場合、画像とその説明、場合によっては、さらにその画像の詳細説明の画面へのリンクなどがワンセットとなった div タグのカタマリを、まとめて、コピー & ペーストを行い、違う部分を書き換えていく、という作業を行うことになります。

▼ **図 1-2　Web ページにおける画面部品の繰り返し**



コンポーネントベースの Web サイト制作

この作業を自動化できれば、サイト制作が楽になります。すなわち、画像と説明などがワンセットとなった div タグを、ひとつの独立した画面部品として定義し、画面に埋め込む際の差分、すなわち、画像の違いや説明文、リンク先などを埋め込みながら、画面に次々部品を配置してくれる、そのようなツールがあると、コピー & ペーストによる作業が不要となります。これにより、Web サイト制作が一段階楽になるだけでなく、コピー & ペースト後の修正ミスによる不手際（アプリケーションでいうところのバグ）を減らす効果が期待できます。

ここでは、画面部品として div タグなどの HTML コードを例としましたが、HTML コードだけでなく、その部品内で利用する CSS コード、さらに、動作を担う JavaScript コードもワンセットとして部品にすることができます。このように、ひとつの画面中の HTML + CSS + JavaScript を部品化し、それらを組み合わせることでひとつの画面を生成する手法を、**コンポーネントベースの Web サイト制作**といいます。というのは、画面を部品化した際のひとつひとつの部品を、**コンポーネント**というからです。

2-1 Astro のコンポーネントと基本構文

1.1.1 項から 1.1.2 項で紹介したように、Astro コーディングは、コンポーネント単位です。コンポーネントに関して、これらの項ではごく簡単に説明しただけでしたが、本節では、もう少し詳しく踏み込みながら、Astro コーディングの基本構文を紹介していきます。

2.1.1 Astro のコンポーネント

図 1-2 をもう一度見てください。そこで軽く解説したように、Web サイトでは、さまざまな画面部品が繰り返し利用されています。この画面部品は、HTML タグだけで構成されているのではなく、その HTML タグを装飾する CSS、画面に動作を付ける JavaScript コードが関連付いていることが多々あります。そこで、この HTML + CSS + JavaScript をワンセットとして部品化したものが、**コンポーネント**です。

ここで、前章で作成した hello-astro プロジェクトの src/components/Welcome.astro ファイルを開いて、中のコードを見てください。リスト 2-1 のような構造になっています。

▼ リスト 2-1　hello-astro/src/components/Welcome.astro

```
---                                                    ❶
 ⋮
---

<div …>                                                ❷
    ⋮
</div>

<style>                                                ❸
    ⋮
</style>
```

大きく次の 3 パートに分かれているのがわかります。

❶ コンポーネントスクリプト

TypeScript[*1] コードを記述する部分です。**フロントマターコンポーネントスクリプト**（**Frontmatter Component Script**）といいます。また、**---** のように、このスクリプトコードの範囲を指定するための 3 文字以上の区切りのことを、**コードフェンス**（**Code Fence**）といいます。

*1　Astro のバージョン 4 までは、JavaScript を利用するプロジェクトの作成が可能であり、その場合は、コンポーネントスクリプトには JavaScript コードを記述していました。

❷ コンポーネントテンプレート

HTML タグを記述します。

❸ コンポーネントスタイル

CSS コードを記述し、style タグで囲みます。

Astro の公式ドキュメントには、この部分の正式な名称の記載はありませんが、コンポーネントスクリプト、コンポーネントテンプレートという名称に合わせて、本書では便宜的にコンポーネントスタイルと呼ぶことにします。

このように、従来の Web サイト制作では、別々のファイルに記述していた HTML、CSS、JavaScript コードを、画面部品ごとにひとつのファイル内にまとめているわけです。そして、このようなファイルを作成する場合、Astro では拡張子を **.astro** とします。

2.1.2 Astro コンポーネントの基本は HTML

Astro コンポーネントの 3 パートは、すべて揃っていなければならないわけではありません。さらには、そのファイル内の記述は、HTML を基本としています。ここで、簡単なサンプルを作成しながら、そのことを確認していくことにしましょう。1.3 節を参考にしながら、本節で作成する新たなプロジェクトとして、basic を作成してください。その上で、index.astro をリスト 2-2 の内容に書き換えてください。それに伴い、basic プロジェクトでは、src フォルダ中の assets フォルダと components フォルダと layouts フォルダは不要となりますので、削除してもかまいません。

▼ リスト 2-2　basic/src/pages/index.astro

```
<html lang="ja">
    <head>
        <meta charset="utf-8" />
        <link rel="icon" type="image/svg+xml" href="/favicon.svg" />
        <meta name="viewport" content="width=device-width" />
        <meta name="generator" content={Astro.generator} />          ❶
        <title>初めてのAstro</title>
    </head>
    <body>
        <h1>Hello Astro</h1>
        <p>初めてのAstroコンポーネントです。</p>
    </body>
</html>
```

コーディングが終了したら、プロジェクトを起動して、次の URL で表示を確認しておきましょう。

```
http://localhost:4321/
```

この URL にアクセスすると、図 2-1 の画面が表示されます。

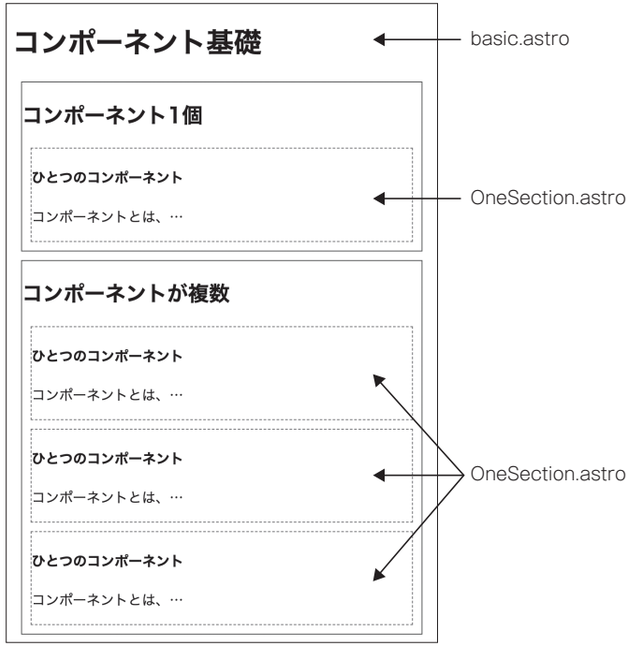
3.1 コンポーネントの埋め込み

前章で紹介した内容は、index.astro や variables.astro など、ひとつのページとして表示させるコンポーネント内での記述方法でした。これらページ用コンポーネント以外のコンポーネントでも、コンポーネント内の記述方法は基本的に同じです。本章では、コンポーネントを組み合わせる方法を紹介していきます。そのうち、本節では、コンポーネントを組み合わせる方法の基礎として、もうひとつコンポーネントを作成し、そのコンポーネントをページ表示用のコンポーネントに埋め込む方法を紹介していきます。

3.1.1 コンポーネントの再確認

図 3-1 は、これから作成するサンプルの画面です。

▼ 図 3-1　これから作成するサンプルの画面



この画面において、図 3-2 の部分は、何回か登場します。

▼ 図 3-2　図 3-1 で何回か登場する画面の一部

ひとつのコンポーネント

コンポーネントとは、…

Web の画面を作成していくと、このように、画面の一部が何回か登場します。そして、そのような画面の一部が**コンポーネント**となり、再利用できるようになります。Astro では、このコンポーネントを **.astro** ファイルとして作成し、その中に HTML+CSS+TypeScript を記述できます。index.astro をはじめとして、前章でコーディングしてきたページ用のコンポーネントも、もちろんコンポーネントのひとつであり、この index.astro を起点にすべての画面表示が始まります。

3.1.2 コンポーネントの作り方

では、早速コーディングを行っていきましょう。まずは、プロジェクトの作成です。本章で作成するサンプルをまとめるプロジェクトとして、プロジェクトフォルダ名を components として作成してください。次に、図 3-2 のコンポーネントから先に作成します。図 3-1 の通り、このコンポーネントは OneSection.astro です。そして、Astro では、このような画面部品のコンポーネントは、図 1-14（p.18）の通り、src/components フォルダ内に作成するのが一般的となっています。src/components フォルダ内に、リスト 3-1 の OneSection.astro ファイルを作成してください。それに伴い、components フォルダ内の Welcome.astro は削除してもかまいません。同様に、assets フォルダ、layouts フォルダも削除してもかまいません。

▼ リスト 3-1　components/src/components/OneSection.astro

```
<section class="box">
    <h4>ひとつのコンポーネント</h4>
    <p>
        コンポーネントとは、…
    </p>
</section>

<style>
.box {
    border: green 1px dashed;
    margin: 10px;
}
</style>
```

❶

❷

OneSection.astro は、特に変数や制御構文などを必要としないコンポーネントなので、リスト 3-1 ではコンポーネントスクリプトは存在しておらず、テンプレート部分としては、❶の section タグのみとなっています。

図 3-2 には枠線が表示されていますが、これは、説明上コンポーネントの境界をはっきりさせるためのもの

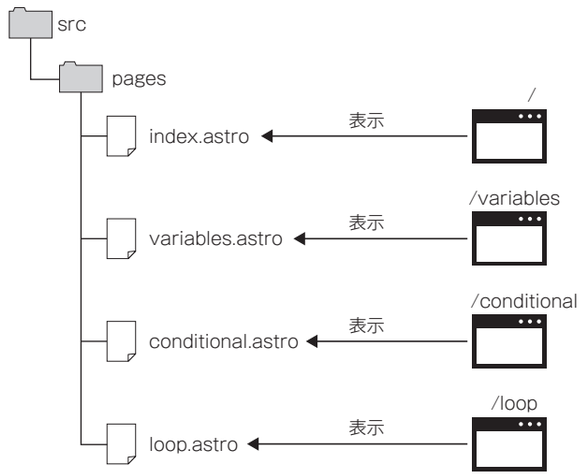
4.1 ルーティングとルートパラメータ

本章では、複数画面による画面遷移を Astro でどのように実現するかを紹介していきます。その際、活躍するのが pages フォルダです。といっても、これまでも pages フォルダにファイルを追加してそのファイルの内容を表示させてきています。そのカラクリをここで明かすところから話を始めていきましょう。

4.1.1 ファイルベースのルーティング

本章のメインテーマである**ルーティング**というのは、あるパス（URL のドメイン以降の部分）に対してどのような画面を表示させるかを決めるための仕組みです。このルーティングを、Astro では pages フォルダにファイルを追加し、そのファイル名をパス名とすることで実現しています。このような仕組みのことを、**ファイルベースルーティング**（**File-based Routing**）といいます（図 4-1）。そして、この pages フォルダ内のファイルのことを、**ページ**（**Pages**）、あるいは、**ページファイル**（**Page Files**）といいます。これらページファイルのうち、これまではページ用コンポーネントと表現してきた .astro ファイルのことを、正式には、**Astro ページ**といいます。というのは、ページファイルとして利用できるファイル形式が、.astro のみではないからです。これらに関しては、後の章で紹介します。なお、pages フォルダ中の index.astro は、index.html と同様に、そのパスはルート（/）となります。

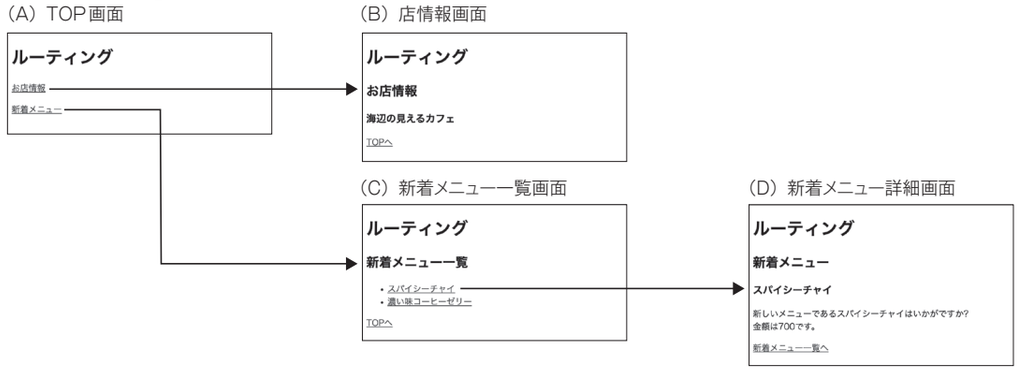
▼ **図 4-1　basic プロジェクトでのファイルベースルーティング**



4.1.2 サンプルプロジェクトの作成

Astro のファイルベースルーティングの仕組みを掘り下げていくために、ここからサンプルプロジェクトを作成していきながら理解していくことにしましょう。今回作成するサンプルプロジェクトである routing は、図 4-2 の 4 画面から成り立っています。

▼ **図 4-2　routing プロジェクトの画面遷移**



画面記号と画面名、その画面を表すパスの対応関係は、表 4-1 の通りです。

▼ **表 4-1　routing プロジェクトの画面**

画面記号	画面名	パス
A	TOP 画面	/
B	店情報画面	/about
C	新着メニュー一覧画面	/newMenus
D	新着メニュー詳細画面	/newMenus/22345

アプリケーションを表示させると、（A）TOP 画面が表示されます。そのため、この TOP 画面のパスは、/（ルート）となっています。その TOP 画面中の［お店情報］のリンクをクリックすると、（B）店情報画面が表示されます。その際のパスは、表 4-1 の通り、/about です。また、［新着メニュー］のリンクをクリックすると、（C）新着メニュー一覧画面が表示されます。その際のパスは /newMenus です。新着メニュー一覧画面の各新着メニュー名がリンクとなっており、それをクリックすると（D）新着メニュー詳細画面が表示されます。その際のパスは、表 4-1 では /newMenus/22345 ですが、パス末尾の「22345」はクリックする新着メニューによって変化します。こちらに関しては、4.1.5 項で紹介します。

プロジェクトの作成と index.astro の改変

早速、このプロジェクトを作成していくことにしましょう。プロジェクトフォルダ名を routing としてプロジェクトを作成してください。その上で、index.astro をリスト 4-1 の内容に改変してください。なお、これまで通り、components フォルダ内の Welcome.astro、assets フォルダ、layouts フォルダは削除してもかまいません。

4.2 レイアウト機能

前節で、ルーティングとルートパラメータに関して一通り解説したことになります。ただし、リスト 4-8 のコーディングの際に述べたように、routing プロジェクトには、いろいろとコードの重複が見られ、無駄の多いコードとなっています。本節では、これらのコードの重複を排除しながら、レイアウト機能を紹介していきます。

4.2.1 routing プロジェクトの問題点と新プロジェクトの作成

routing プロジェクトの重複箇所をまとめると、以下の 4 点になります。

1. 同じ Menu インターフェースが 2 箇所で定義されている。
2. Props インターフェース定義が Menu と同じ。
3. 配列と Map の違いはあれど、同じ新着メニューリストデータが 2 箇所で定義されている。
4. 各 Astro ページファイル内のコードに関して、head タグ内のコードなど、ほぼ同じものがある。

これらの問題を解決する方法を解説するために、routing プロジェクトと同じ動作をする新しいプロジェクトを作成することにします。プロジェクトフォルダ名を layout として、プロジェクトを作成してください。その上で、これまで通り、不要な components フォルダ内の Welcome.astro と assets フォルダを削除してください。そして、layouts フォルダ内の Layout.astro ファイルも削除してください。今回、layouts フォルダごと削除しないのは、のちにこのフォルダを利用していくからです。

プロジェクトのコピー

本節で作成する layout プロジェクトは、前節で作成した routing プロジェクトと同じ動作となります。ということは、routing プロジェクトを丸々複製して、必要ファイルを改変すればよいのではないか、と思われたかもしれません。しかし、この方法はやめておいた方がよいです。というのは、Astro では、プロジェクトを作成する際、そのプロジェクト名（= ルートフォルダ名）をプロジェクト内の設定ファイルに記録しています。これらのファイルは、現段階では、package.json と package-lock.json です。もちろん、この仕様は変わる可能性もあります。そのため、プロジェクトを丸々複製した上で改変する方法は危険といえます。やはり、新しいプロジェクトを作成し、必要なファイルやソースコードをコピー & ペーストした方が安全です。

4.2.2 インターフェースファイル

プロジェクトができたところで、まず、問題点 1 を解決していくことにします。問題点 1 は Menu インターフェースの定義コードが 2 箇所に記述されていることです。本来、Menu インターフェースのようにプロジェクト全体で利用されるようなインターフェースは、ひとつのファイルにのみ定義し、他のファイルでは、その定義ファイルからインポートする、というのがメンテナンス性を考えても必須のコーディング手法といえます。

そこで、layout プロジェクトでは、src フォルダ直下にインターフェース定義だけを集めるための interfaces.ts ファイルを作成し、Menu インターフェースの定義コードを記述することにします。これは、リスト 4-9 のコードとなります。

▼ **リスト 4-9　layout/src/interfaces.ts**

```
export interface Menu {
    id: number;
    name: string;
    price: number;
}
```

このように、インターフェースなどの型定義をひとつのファイルにまとめておき、それぞれをエクスポートしておくことで、プロジェクト内の他のファイルは、このファイルからインポートするだけで済みます。結果、同じ型定義コードがプロジェクト内に散在することを防げます。

ただし、少し注意が必要です。次項で実際にこの Menu インターフェースをインポートして利用していきます。その際、このような型定義をインポートする場合は、次のコードのように、通常の import 文ではなく、**import type** と、型定義のインポート文を記述する必要がある点には注意しておいてください。

```
import type {Menu} from "./interfaces";
```

4.2.3 Props のプロパティをオブジェクトにする

次に、問題点 2、すなわち、Props インターフェース定義が Menu と同じという問題点を解決していくことにします。これは、NewMenuListItem コンポーネントの問題です。そこで、まず、routing プロジェクトから NewMenuListItem.astro ファイルを layout プロジェクトの components フォルダ内にファイルごとコピー & ペーストし、リスト 4-10 の太字の部分を変更してください。

▼ **リスト 4-10　layout/src/components/NewMenuListItem.astro**

```
---
import type {Menu} from "../interfaces";  ❶

interface Props {
    menu: Menu;  ❷
}
```

5.1 コンポーネントベースの Web サイト制作ケーススタディの概要

本章では、Astro を利用したコーディングの応用編として、ケーススタディを通じてコンポーネントベースの Web サイト制作を紹介していきます。本節では、その内容を概観しておきましょう。

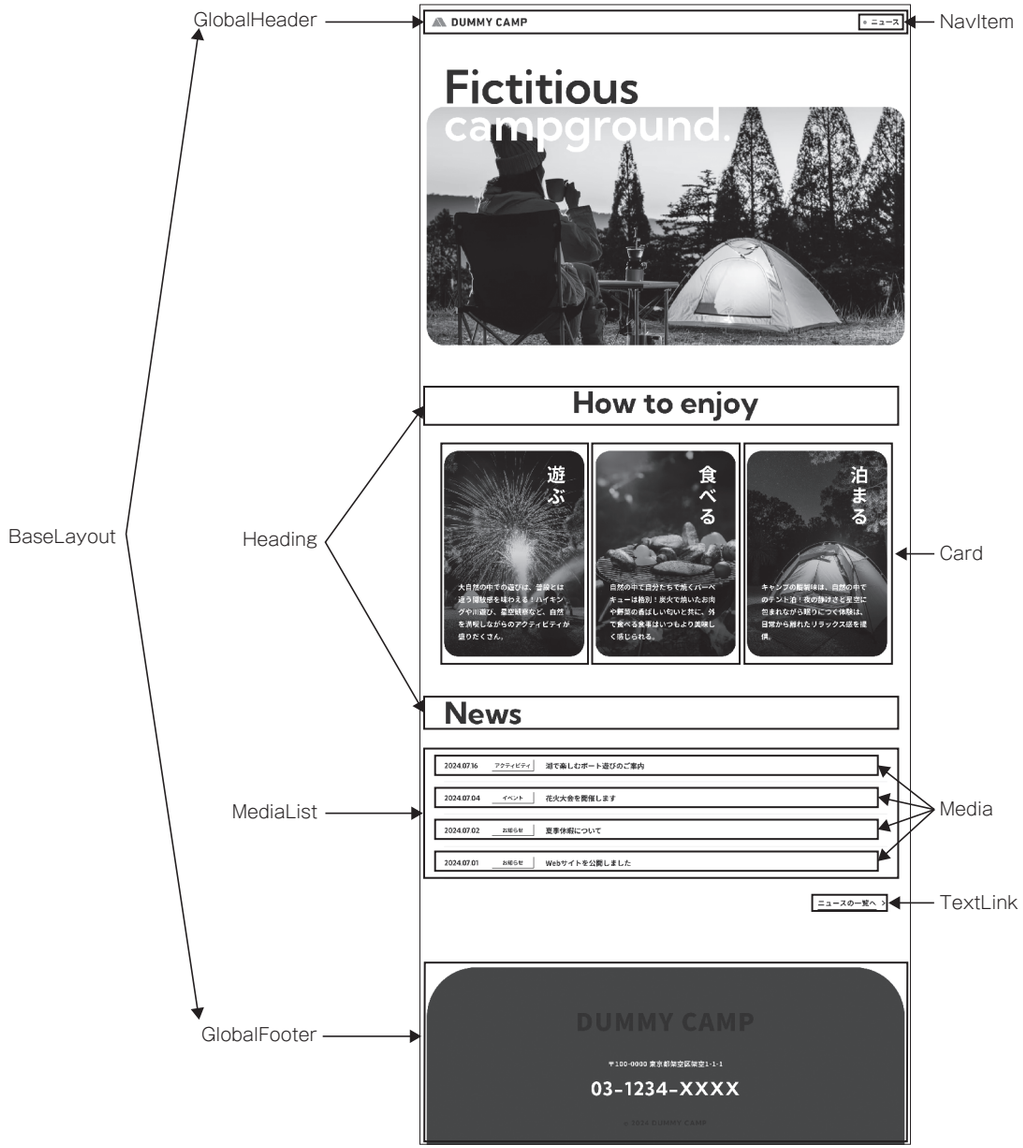
5.1.1 ケーススタディの概説とコンポーネント構成

本章で作成するケーススタディは、架空のキャンプ場を紹介する Dummy Camp サイトです。TOP ページは図 5-1 の（A）、下層ページのひとつであるニュース一覧ページは図 5-1 の（B）の画面となります。

▼ 図 5-1　Dummy Camp サイトの画面（左：(A)、右：(B)）

通常、このようなサイトを作成する場合は、サイト共通で利用される css ファイルを作成した上で、画面それぞれに対応した html ファイルを作成します。一方、コンポーネントベースによる Web サイト制作の場合は、画面を部品に分け、それぞれをコンポーネントファイルとして作成し、最終的に Astro ページ上に組み込んでひとつの画面とします。この考え方を TOP ページに適用すると、図 5-2 のようなコンポーネントに分けることができます。

▼ 図 5-2　コンポーネントに分割した TOP ページ



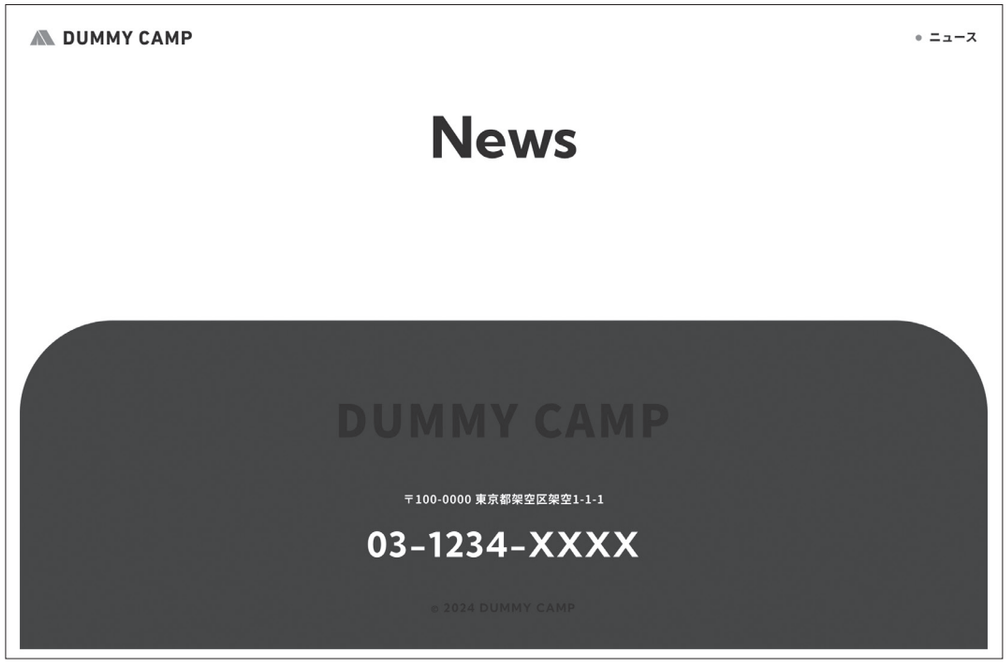
5-4 ニュースページとマークダウンファイル

TOP ページも残すところ News セクションのみとなりました。ただ、図 5-1 の（A）と（B）を見比べてみても、News セクションは、ニュース一覧ページのメインセクションとあまり変わりません。そこで、TOP ページのコーディングは、ここでいったん保留し、下層ページであるニュース一覧ページを作成していくことにします。

5.4.1 ニュース一覧ページの作成

ニュース一覧ページへのリンクは、リスト 5-11 の navLinkArray のデータを見てもわかるように、/news/ です。そのため、ニュース一覧ページファイルは、pages/news/index.astro となります。このファイルを作成し、コーディングしていきますが、ニュースリスト部分はのちに追記することにして、それ以外の部分を先に作ります。理由については後述します。これは、図 5-14 の画面となります。

▼ 図 5-14　ニュースリスト部分以外を表示させたニュース一覧ページ



ニュース一覧ページは、図 5-3 のコンポーネント構成からもわかるように、BaseLayout 内に LocalHeader コンポーネントが配置されているだけです。そこで、まず、LocalHeader コンポーネントを作成します。これは、リスト 5-20 のコードとなります。

▼ リスト 5-20　dummy-camp-static/src/components/LocalHeader.astro

```
---
//Propsインターフェースの定義。
interface Props {
    title: string;
}
//Propsデータの取得。
const {title} = Astro.props;
---

<header class="local-header">
    <h1 class="page-title">
        {title}
    </h1>
</header>

<style>
/* ページのヘッダ */
.local-header {
    ⋮
}
/* ページのタイトル */
.page-title {
    ⋮
}
</style>
```

次に、ニュース一覧ページ本体である news/index.astro を作成します。これは、リスト 5-21 のコードとなります。コード中のコメントとして記載している「ここにニュースリストを表示」としている部分に、5.4.5 項でニュースリストを表示させるコードを追記します。

▼ リスト 5-21　dummy-camp-static/src/pages/news/index.astro

```
--
//BaseLayoutのインポート。
import BaseLayout from "@/layouts/BaseLayout.astro";
//LocalHeaderのインポート。
import LocalHeader from "@/components/LocalHeader.astro";
---

<BaseLayout title="News | DUMMY CAMP" description="DUMMY CAMPのニュース一覧です。">
    <div class="container">
        <LocalHeader title="News"/>
        <!-- ここにニュースリストを表示 -->
```

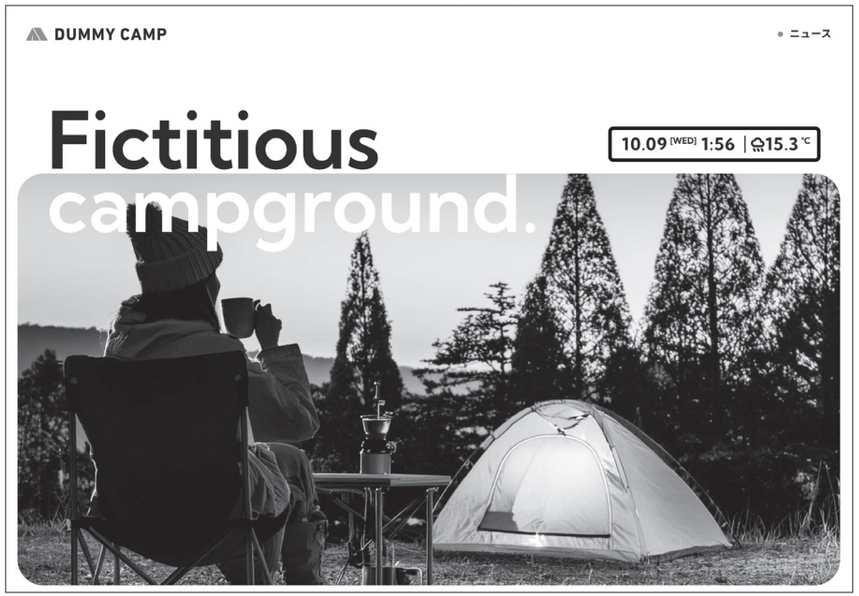
6.1 ケーススタディの概要と準備

本章では動的サイトの作り方を紹介します。題材は、前章で作成した Dummy Camp サイトをベースにしていきます。本節では、扱う内容を紹介するとともに、コーディングのための準備を進めます。

6.1.1 追加するコンポーネント

本章で作成する Dummy Camp サイトの改造版は、図 6-1 の画面となります。

▼ 図 6-1　日時・天気・気温の表示が追加された Dummy Camp サイトの TOP ページ



前章の TOP ページの完成形である図 5-1 の（A）と見比べると、現在の日時、天気、気温が表示された部分（図 6-1 の線で囲まれた部分）が追加されています。ここで表示されている天気や気温は、Dummy Camp の所在地である東京の現在の天気と気温を表しています。そして、このような情報は、表示のたびに最新情報が表示されていなければなりません。つまり、表示内容がその都度変わる動的サイトです。

本章では、図 6-1 の追加表示部分を WeatherInfo コンポーネントとし、このコンポーネントファイルを作成しながら、Astro で動的サイトを作成する方法を紹介していきます。

6.1.2 プロジェクトの準備

まず、プロジェクトの準備を行いましょう。サイトの基本部分は、前章で作成した Dummy Camp サイトなので、dummy-camp-static プロジェクトを改造する形で進めてもかまいません。ただ、dummy-camp-static はそのまま残しておきたいので、ここでは、別プロジェクトとして、dummy-camp-dynamic を作成し、各種設定を行い、必要ファイルをコピー & ペーストした上で、そこに必要なコードを追加します。手動で dummy-camp-dynamic プロジェクトの準備を行う場合は、項末の Note を参照してください。一方、ダウンロードサンプルには、スケルトンプロジェクトとして dummy-camp-dynamic-skeleton を用意していますので、この dummy-camp-dynamic-skeleton を dummy-camp-dynamic と名称変更し、利用することをお勧めします。

NOTE

手動での dummy-camp-dynamic の準備方法

手動で dummy-camp-dynamic プロジェクトを準備する場合、プロジェクト作成後、次のコマンドを実行し、依存ライブラリをインストールします。

```
npm install postcss-preset-env --save-dev
npm install dayjs
npm install @fontsource-variable/noto-sans-jp
npm install @fontsource-variable/kumbh-sans
```

その後、次のフォルダ、および、ファイルを丸ごと dummy-camp-static から dummy-camp-dynamic の同じ位置にコピー & ペーストします。既存ファイルが存在する場合は、置き換えてください。

- public フォルダの中身
- src フォルダの中身
- postcss.config.cjs
- tsconfig.json

6.1.3 フォントの追加

前項で用意した dummy-camp-dynamic に天気情報コンポーネントを追加する前に、もうひとつフォントを追加しておく必要があります。追加するフォントは、**Material Symbols Outlined** で、図 6-1 の天気情報の天気アイコン部分で利用されています。プロジェクト直下で次のコマンドを実行し、フォントライブラリを追加しておいてください。

```
npm install @fontsource-variable/material-symbols-outlined
```

6.5 SSR デプロイ

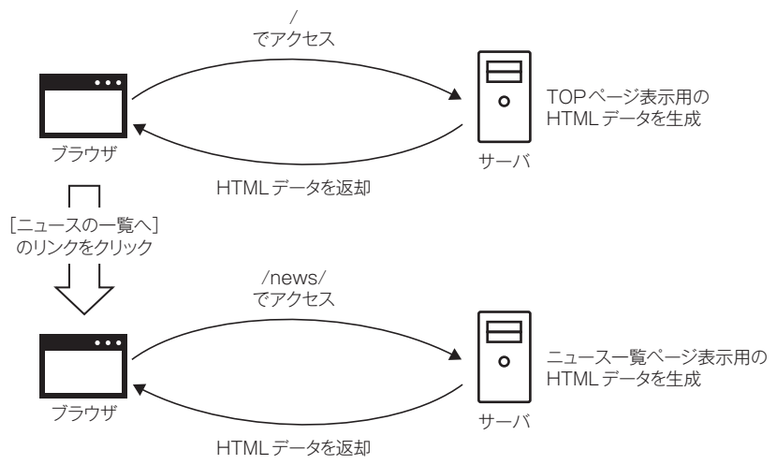
Astro で動的サイトを作成する方法は、もうひとつあります。ただし、コンポーネント単位で行うのではなく、サイト全体で動的に行う方法です。本章の最後に、その方法を紹介します。

6.5.1 Astro の SSR とホスティングサービス

Astro で作成したサイトを公開する場合、これまで紹介した方法というのは、ビルドを行い、生成されたファイル一式をどこかの Web サーバに配置する方法でした。その場合、ブラウザに表示する HTML（+CSS）データは、あらかじめファイルとして生成されています。

一方、ブラウザがアクセスしてきたそのタイミングで、サーバ上で HTML（+CSS）データを生成し、ブラウザにレスポンスとして返すという方法があります。これを、**SSR**（**Server-Side Rendering**）、あるいは、**オンデマンドレンダリング**（**On-demand Rendering**）といいます（図 6-11）。この方法は、PHP や Java などによる、いわゆるサーバサイド Web アプリケーションとまったく同じです。

▼ 図 6-11　SSR の動作イメージ



Astro プロジェクトは、この SSR での運用にも対応しています。ただし、Astro プロジェクトが動作するサーバ環境を用意する必要があります。その動作環境の基本は Node.js ですので、もちろん、手動で Node.js をインストールし、そのようなサーバ環境を構築することも可能です。一方で、Astro プロジェクトの SSR 運用が可能なホスティングサービスがあります。これらのうち、原稿執筆時点では **Netlify**、**Vercel**、**Cloudflare** の 3 種に対して、Astro から公式の連携ライブラリ（Integrations）が用意されています。3 種とも月額料金が設定されていますが、無料利用も可能です。本書では、このうち、Netlify を利用していきます。Netlify の特徴、および使い方は後述します。

その Netlify にデプロイする Astro プロジェクトは、dummy-camp-dynamic プロジェクトと同内容のものとします。ただし、dummy-camp-dynamic をそのまま利用するのではなく、あくまで別プロジェクトとして作成し、少し改造を行ったのちに、SSR の設定を行い、デプロイを行うことにします。

6.5.2 プロジェクトの準備

まずは、プロジェクトの用意です。プロジェクト名を dummy-camp-ssr とし、各種設定を行い、必要ファイルをコピー & ペーストした上で、そこから改造や SSR の設定を行います。これまで同様、手動で dummy-camp-ssr プロジェクトの準備を行う場合は、項末の Note を参照してください。一方、ダウンロードサンプルには、スケルトンプロジェクトとして dummy-camp-ssr-skeleton を用意していますので、この dummy-camp-ssr-skeleton を dummy-camp-ssr と名称変更し、利用することをお勧めします。

なお、スケルトンプロジェクトに関して 2 点ほど注意点があります。まず、WeatherInfo コンポーネントに関しては、リスト 6-5 のものを利用します。script タグが記述された WeatherInfo.astro や WeatherInfo.vue は不要です。それに伴い、index.astro での WeatherInfo コンポーネントの埋め込みコードは、リスト 6-6 の内容となります。すなわち、次のコードとなり、client ディレクティブは記述しません。

```
<WeatherInfo/>
```

また、tsconfig.json 中の「"jsx": "preserve"」の部分も不要です。もし手動でプロジェクトを作成する場合は、この 2 点に注意してください。

NOTE

手動での dummy-camp-ssr の準備方法

手動で dummy-camp-ssr プロジェクトを準備する場合、プロジェクト作成後、次のコマンドを実行し、依存ライブラリをインストールします。

```
npm install postcss-preset-env --save-dev
npm install dayjs
npm install @fontsource-variable/noto-sans-jp
npm install @fontsource-variable/kumbh-sans
npm install @fontsource-variable/material-symbols-outlined
```

その後、次のフォルダ、および、ファイルを丸ごと dummy-camp-dynamic から dummy-camp-ssr の同じ位置にコピー & ペーストします。既存ファイルが存在する場合は、置き換えてください。

- public フォルダの中身
- src フォルダの中身
- postcss.config.cjs
- tsconfig.json

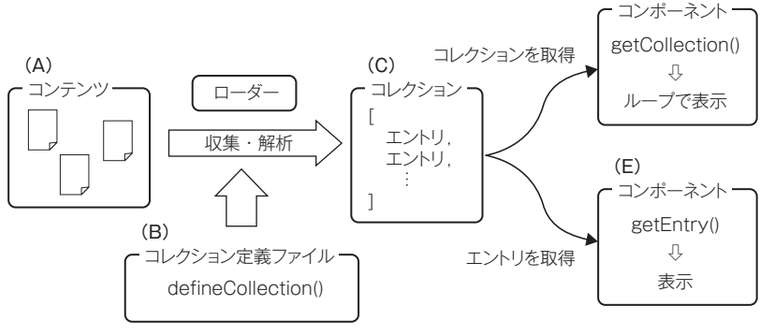
7.1 コンテンツコレクション

第 5 章、第 6 章を通して作成してきた Dummy Camp サイトのニュースページは、マークダウンファイルをページファイルとして利用するものでした。もちろん、この方法でもマークダウンファイルを追加することで、ニュースページを増やすことが可能です。一方で、Astro では、このようにページの増減も含めて、サイト内で可変なコンテンツを効率よく管理する仕組みとして、**コンテンツコレクション**（**Content Collections**）という機能があります。本章では、このコンテンツコレクションを利用したサイトの作成方法を紹介していきます。

7.1.1 コンテンツコレクションの概要

コンテンツコレクションの仕組みとその利用手順をまとめると、図 7-1 のようになります。

▼ **図 7-1　コンテンツコレクションの概要**



コンテンツコレクションは、マークダウンファイルなどのコンテンツがひとまとまりとなった場所（図 7-1 中の（A））からコンテンツデータを収集します。この収集するものを**ローダー**といい、コンテンツデータの収集先やコンテンツデータの形式に応じて適切なローダーを選択します。そのローダーとしてどれを利用するかも含めて、コンテンツの取集方法を定義するのが、**コレクション定義ファイル**（**Collection Config File**）であり、ファイル中に **defineCollection()** 関数で定義します（図 7-1 中の（B））。

ローダーは、収集したコンテンツを解析し、オブジェクトの配列を生成します（図 7-1 中の（C））。この生成された配列を**コレクション**（**collection**）といいます。また、配列の各要素、すなわち、コンテンツデータのオブジェクトを**エントリ**（**entry**）といいます。

各コンポーネントでは、コンテンツコレクションの **getCollection()** 関数を利用して、コレクションを取得し、その配列をループさせることでリスト表示できます（図 7-1 中の（D））。また、**getEntry()** 関数を利用して、

ひとつエントリを取得し、ひとつのコンテンツを表示できます（図 7-1 中の（E））。

7.1.2 ケーススタディの概要とプロジェクトの準備

では、実際にコーディングしていきましょう。本章ではニュースコンテンツの管理に注目していくため、第 5 章で作成した天気情報表示部分のない、すなわち、動的コンポーネントのない dummy-camp-static プロジェクトをベースにします。まず、プロジェクトの準備を行いましょう。これまでと同様に、dummy-camp-static はそのまま残しておき、別プロジェクトとして、dummy-camp-content を作成し、各種設定を行い、必要ファイルをコピー & ペーストした上で、そこに必要なコードを追加します。手動で dummy-camp-content プロジェクトの準備を行う場合は、下記の Note を参照してください。一方、ダウンロードサンプルには、スケルトンプロジェクトとして dummy-camp-content-skeleton を用意していますので、この dummy-camp-content-skeleton を dummy-camp-content と名称変更し、利用することをお勧めします。

NOTE

手動での dummy-camp-content の準備方法

手動で dummy-camp-content プロジェクトを準備する場合、プロジェクト作成後、次のコマンドを実行し、依存ライブラリをインストールします。

```
npm install postcss-preset-env --save-dev
npm install dayjs
npm install @fontsource-variable/noto-sans-jp
npm install @fontsource-variable/kumbh-sans
```

その後、次のフォルダ、および、ファイルを丸ごと dummy-camp-static から dummy-camp-content の同じ位置にコピー & ペーストします。既存ファイルが存在する場合は、置き換えてください。

- public フォルダの中身
- src フォルダの中身
- postcss.config.cjs
- tsconfig.json

7.1.3 コンテンツファイルの配置とコード変更

これまでの Dummy Camp プロジェクトでは、コンテンツにあたるマークダウンファイルを、そのままソースファイルの一種であるページファイルとしていました。コンテンツコレクションを利用する場合は、コンテンツにあたるマークダウンファイルは、ソースファイルとは別管理とできます。そのため、ソースファイルを格納する src フォルダ内とは別にすることをお勧めします。そこで、プロジェクト直下にニュースコンテンツのマー