Blazor (client-side) アプリケーションプログラミング自習書

目次

本自習書について

本自習書は、C# で Single Page Web Application (SPA) を実装できるフレームワーク "Blazor (ブレイザー)" の client-side 版 (WebAssembly 版) を、ステップバイステップで体験しながら学んだり感触を試したりするための自習書です。

ステップごとの完動ソースコードと、Git リポジトリを同梱しています。

本自習書についての連絡は、下記 GitHub リポジトリの Issue までお願いいたします。

https://github.com/jsakamoto/self-learning-materials-for-blazor-jp/issues

主な対象者

本自習書では、サーバー側実装として ASP.NET Core MVC を採用しています。
また Blazor は、基本的にプログラミング言語は C# が想定されています。

そのため、本自習書では下記のような開発者を主な対象者として想定しております。

- HTML/CSS/JavaScript を用いた Web アプリケーション開発の知識がある
- C# によるプログラミングの知識がある
- 加えて ASP.NET Core MVC によるサーバーサイド Web アプリケーション開発の知識があるとなお可

Blazor のバージョンとエディション

本自習書が対象としている Blazor のバージョンは、本稿執筆時点での最新版である v.3.2.0 です。

また、Blazor には以下の "エディション" があります。

- ブラウザ上で実行される **"client-side" 版** (WebAssembly 版)
- ASP.NET Core サーバー側実装と SignalR 双方向通信で結ばれて実行される **"server-side" 版**

本自習書が対象としている Blazor のエディションは **"client-side" 版** (WebAssembly 版) です。
以降、特に明記なく "Blazor" とだけ記してある場合は、"client-side" 版 (WebAssembly 版) の Blazor を指すものとします。

なお、"client-side" 版の Blazor はブラウザ上の WebAssembly エンジンで実行されるので、静的コンテンツサーバーへの配置だけでも動作し、本質的にはサーバー側実装を必要としません。

しかしながら本自習書では、データの永続化や通信などの目的で、ASP.NET Core MVC によるサーバー側実装もからめた内容となっています。

開発環境

本稿執筆時点で本自習書による Blazor (client-side) 開発を実践するにあたり必要な開発環境は下記のとおりです。

- **.NET Core 3.1 SDK (3.1.300 かそれ以降)**
  https://dotnet.microsoft.com/download/dotnet-core/3.1
- **Visual Studio 2019 - 16.6.0 以降**
  (※利用条件に抵触しなければ無償版の Community Edition 可)
  https://visualstudio.microsoft.com/vs/
  - "ASP.NET と Web 開発" ワークロードが選択されていること
- **以上の環境をインストールし利用可能な Windows OS**

なお、Blazor アプリ開発にあたっては、最低限、

- .NET Core 3.1 SDK
- および任意のテキストエディタ

さえあれば、"dotnet" CLI (Command Line Interface) を用いて、Linux 各種ディストリビューションや macOS 上でも実践可能です。

特にテキストエディタとして、**Visual Studio Code** に "C# for Visual Studio Code" 拡張 v.1.21 以上 (https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp) を追加して使用する場合は、OS に依らずに、本自習書で説明しているような Visual Studio 2019 と同等の開発支援が得られます。

本自習書では Windows OS 上で Visual Studio 2019 16.6 以降を使っての手順で説明いたします。

この自習書で作成する Web アプリケーション

ブラウザ上の操作で、あらかじめ登録しておいたコンピューターの MAC アドレスに対し、Wakeup On LAN (以下 WOL) のマジックパケットを送信することで、目的のコンピューターの電源を入れる、Single Page Web アプリケーションを、Blazor (client-side) を使って実装します。

WOL マジックパケット送信して電源を入れる対象のコンピューターを、その名称と MAC アドレスで新規登録および編集するページを備えます。

Step 1.　ボイラープレートのビルド

前述の開発環境が整うと、Visual Studio 2019 のプロジェクトテンプレートにて、Blazor を選んでプロジェクト新規作成することができます (左図)。

しかしながら、左図プロジェクトテンプレートから作成した Blazor プロジェクトは、はじめからルーティングの仕組みが実装済みであったり、Bootstrap が組み込んであったり、ある程度作りこまれた形となっています。

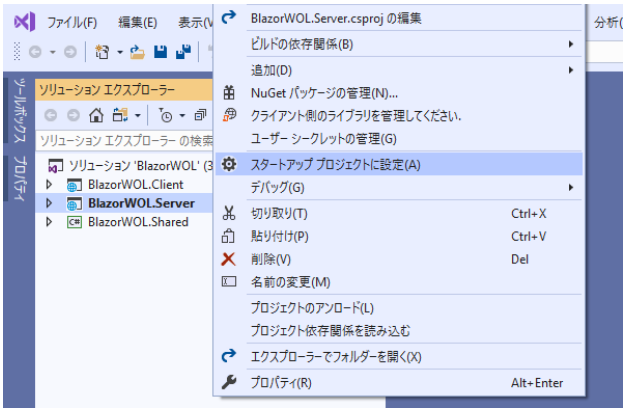
この形は、一歩ずつ何もないところから理解を積み上げていくタイプの学習には向いていません。

そこでこの自習書では、別途 Zip アーカイブの形で提供します、ほぼ素の状態のプロジェクトファイル一式を解凍いただいて自習の開始地点とします。

まずは本自習書に同梱の BlazorWOL-Step01-Boilerplate.zip を好みの作業フォルダに解凍してください。

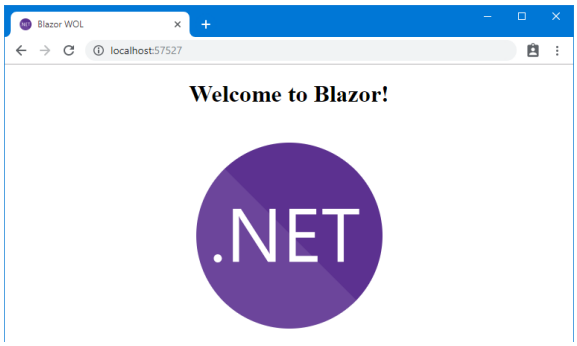
> ※注意: **解凍前**に、ダウンロードした Zip ファイルの**「ブロックの解除」**を**忘れずに**行っておいてください。

解凍すると、"BlazorWOL.sln" というソリューションファイルがあります。
この "BlazprWOL.sln" を Visual Studio 2019 で開きます。



このソリューションファイルを開くと、Visual Studio のソリューションエクスプローラウィンドウにて、3 つのプロジェクトが収録されているのがわかります。

これらプロジェクトのうち、**BlazorWOL.Server** をマウスで**右クリック**して**「スタートアッププロジェクトに設定」をクリック**し、**常にこのプロジェクトを実行**するように固定しておきます （左図）。



こうして Ctrl + F5 を押してビルド・デバッガなし実行してみてください。
ビルドが無事完了して、Web アプリサーバーが起動し、ブラウザが開いて、左図のとおり表示されれば成功です。

> ※ Visual Studio Code での開発も容易とするため、本自習書に同梱のボイラープレート BlazorWOL-Step01-Boilerplate.zip には、**Visual Studio Code 用のビルドタスク設定も収録**しています (".vscode" サブフォルダ)。そのため、Visual Studio Code でこのボイラープレート解凍場所のフォルダを開き、Ctrl + F5 といった "デバッグなしで開始 (Start Without Debugging)" のタスクを実行すれば、上記 Visual Studio 2019 での開発時と同じく、ビルドとブラウザの起動が実行されます。

補足 - プロジェクトの構造

この BlazorWOL アプリケーションのプロジェクト構造を少し掘り下げてみます。

BlazorWOL.Server

3 つあるプロジェクトのうち、BlazorWOL.Server は、従来からある ASP.NET Core MVC Web アプリとほぼ相違ない、サーバー側実装です。

BlazorWOL.Client

この自習書でのいちばんの注目ポイントは、BlazorWOL.Client プロジェクトです。
このプロジェクトで実装するコードはすべて、ブラウザ上で実行される、クライアント側実装となります。

この BlazorWOL.Client プロジェクトで記述したビューやロジックは、コンパイルされて.NET アセンブリファイル (.dll ファイル) となります。

そして、そのほか参照している必要な .NET アセンブリファイルともども、ブラウザ上の WebAssembly エンジン上で実行中の dotnet.wasm によってブラウザ上に読み込まれ、SPA アプリケーションとして実行されます。

BlazorWOL.Shared

BlazorWOL.Shared プロジェクトは、これらクライアント側とサーバー側との双方で共通に使用する型や機能を収録する、.NET Standard 2.1 クラスライブラリです。
このプロジェクトは、BlazorWOL.Client プロジェクトと BlazorWOL.Server プロジェクトの両方から参照設定されています。
クライアント側とサーバー側との通信でやりとりするデータ型を実装するのは、この共通用途のクラスライブラリの主な用途です。

補足 - Blazor アプリケーションが立ち上がるまでの流れ

index.html

ブラウザ上に最初に読み込まれるのは、BlazorWOL.Client プロジェクトにある wwwroot¥index.html です。

index.html には、<app> というタグが記述されています。

この <app> タグが、Blazor におけるコンポーネントを指しています。

<app> タグ内には "Loading…" のテキストが記述されており、ブラウザが最初に index.html を読み込んだ直後はこのテキストがブラウザ画面上に表示されています。

その後、dotnet.wasm による Blazor アプリケーションのロードと実行が始まると、この <app> タグが、Blazor コンポーネントによる DOM に差し変わります。

Program.cs

ここで BlazorWOL.Client プロジェクトの Program.cs を見てみましょう。
この Program.cs の実装内容は C# コンパイラで .dll にビルドされますが、ブラウザ上に読み込まれてブラウザ上で動作することを思い出してください。

Blazor アプリケーションの初期化処理が、この Program.cs 中の Main メソッドです。
ここに下記記述があります。

```
builder.RootComponents.Add<App>("app");
```

この記述により、先の index.html 中の <app> 要素に、App クラスという Blazor コンポーネントを充てる仕組みとなっています。

ではこの App クラスはどこで定義されているのでしょうか。

App.razor

BlazorWOL.Client プロジェクトには、App.razor というファイルが収録されています。
この .razor ファイルが Blazor コンポーネントであり、App クラスの実装です。

Blazor アプリケーションのプロジェクトにおいては、.razor ファイルは C#コードにコンパイルされて最終的に.NET アセンブリファイル (.dll ファイル) にコンパイルされます。

このときに、 .razor ファイルは、そのファイル名と同じ名前のクラスとしてコンパイルされます。
(すなわち、App.razor のコンパイルによって、App クラスができあがる)

以上の一連の定義によって、Blazor アプリケーションが立ち上がります。

Step 2.　CSS スタイルシートを実装

さて、先へ進む前に、あらかじめ作成しておいた CSS スタイルシートファイルを適用しておきます。

Blazor は Web 標準の要素で動く Single Page Web アプリケーションです。
よって、外観の実装には、通常、カスケードスタイルシート (CSS) が用いられます。

そのような用途で、いわゆる "CSS フレームワーク" と呼ばれる、Bootstrap や Materialize-CSS などのライブラリが使われることがあります。
実際、Blazor は、それら CSS フレームワーク/ライブラリを使用して外観を実装することができます。

しかしながら、それら CSS フレームワーク/ライブラリの使用は本自習書の目的ではないため、すでに作り置きしてあるスタイルシートファイル (「step-02-define-styles」フォルダ以下、BlazorWOL.Client プロジェクト内の wwwroot フォルダの styles.css) を適用しておいてください。

Step 3.　タイトルの変更 – Blazor コンポーネントの記述構造の理解と、データバインディング

それでは App.razor の記述内容を見てみましょう。

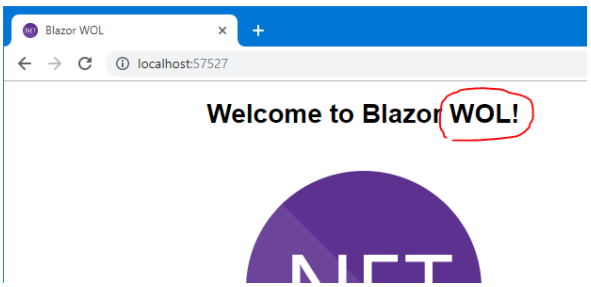
App.razor には HTML タグの記述と、C#のコードブロックが含まれています。

.razor 中のコードブロックに記述した内容は、.razor ファイル名のクラスのメンバー (フィールド、プロパティ、メソッド) になります。

そして、 .razor 中の HTML 記述中に、"@" に続けて記述する C#コードで、この.razor 中のコードブロックで実装したメンバーを参照 (バインド) できます。

試しに App.razor 中のコードブロックに記載のある、Title フィールドに代入している文字列を、"Blazor" から "Blazor **WOL**" に変更してみてください。

```
@code {
    string Title = "Blazor WOL";
}
```



App.razor を上記のとおり編集し保存後、ビルドして、ブラウザで再読み込みを実行すると、たしかに h1 要素のテキストが書き換わっているのが確認できます（左図）。

> ※ .razor ファイルは、ブラウザにとっての静的コンテンツではありません。C#コードにコンパイルされ、最終的に.NET アセンブリファイル (.dll ファイル) を成すものです。よって、**.razor ファイルを変更・上書き保存したら、再ビルド**して .dll ファイルを更新してからブラウザで再読み込みする必要があります。
> **.razor ファイルは、特殊な書式の C#ソースコードである**、と理解するのがよいでしょう。

Step 4.　モデルクラスの追加

概要

それではいよいよ、目標のアプリケーションの実装へと作業を進めていきましょう。
まずは、**WOL で電源を入れる対象のデバイスを表現する、Device クラス**を実装します。

Device クラスは以下のプロパティを持たせます。

- オブジェクトを一意に識別するための **Id プロパティ** (GUID 型)
- デバイスの登録名である **Name プロパティ** (文字列型)
- WOL で電源を入れる宛先となる **MAC アドレス** (文字列型)

本自習書で作成する BlazorWOL アプリケーションでは、この Device クラスのオブジェクトを、追加・変更・削除する機能・ユーザーインターフェースを実装していきます。

まずはクライアント側の実装を進めることにします。
そのため暫くは、Device クラスは BlazorWOL.Client プロジェクト上でのみ使用します。

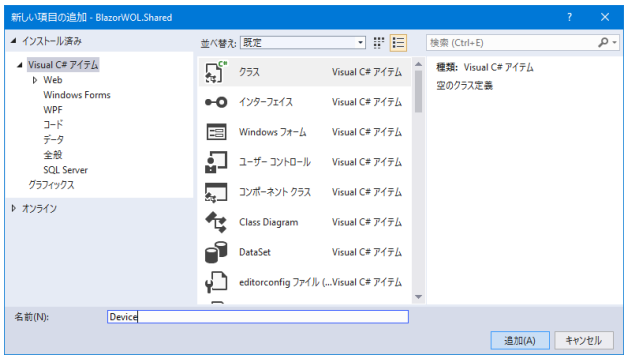
しかしいずれ、サーバー側 BlazorWOL.Server プロジェクトでも、永続化や実際のマジックパケット送信などで Device クラスを参照することになります。

このように **Device クラスはクライアント側・サーバー側の両方で使用される型**になります。
そこで、Device クラスは、**クライアント側・サーバー側の双方から参照される共有クラスライブラリプロジェクト**である、**BlazorWOL.Shared** プロジェクトに実装することにしましょう。

手順

1. Visual Studio のソリューションエクスプローラ上で BlazorWOL.Shared プロジェクトを右クリックし、メニューから [追加(D)]-[クラス(C)...] をクリックします。

2. 「新しい項目の追加」ダイアログが現れるので、[名前(N)] に "Device" と入力して [追加(A)] ボタンをクリックします。



Device.cs ファイルが追加されるので、内容を以下のとおり実装します。

> ※Device クラスのアクセス制御 "public" を追加するのを忘れずに。
> ※プロパティのコーディングには、"prop"[TAB]と入力すると簡単にプロパティのひな型を生成できます。

```
using System;

namespace BlazorWOL.Shared
{
    public class Device
    {
        public Guid Id { get; set; } = Guid.NewGuid();

        public string Name { get; set; }

        public string MACAddress { get; set; }
    }
}
```

Step 5.　デバイス一覧ページの実装 - コンポーネントの追加

概要

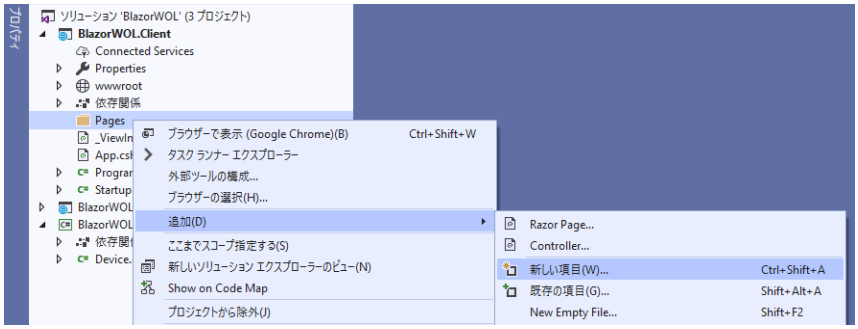
続けて、Device クラスのオブジェクトを表示する手はずを進めていきましょう。
BlazorWOL は SPA として実装しますから、この表示機能はクライアント側で実装します。

まずは Device クラスを表示する Blazor コンポーネント "DevicesComponent.razor" を新規作成します。
慣例的に、Blazor アプリケーションプロジェクトにおいて、(App.razor を除く) コンポーネント = .razor ファイルは Pages フォルダに配置します。

とりあえずはダミーデータとして用意したひとつの Device オブジェクトを表示できるところまで進めましょう。

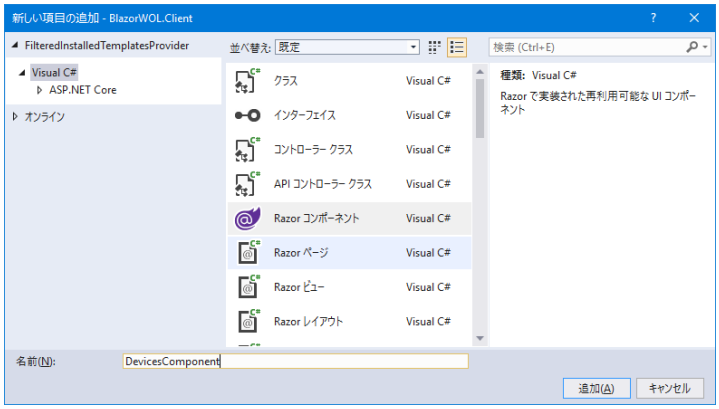
手順

1. Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトの Pages フォルダを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。



2. 「新しい項目の追加」ダイアログが現れるので、
   - ダイアログ中央のアイテム一覧から「**Razor コンポーネント**」をクリックして選択して、
   - [名前(N)] に "**DevicesComponent**" と入力してから
   - [追加(A)] ボタンをクリックします。

> ※Visual Studio Code や dotnet CLI で開発の場合は、単純に、"DevicesComponent.razor" というファイル名の空ファイルを、Pages フォルダ内に touch コマンドなどで新規作成するだけでよいです。



3. DevicesComponent.razor ファイルが Pages フォルダ内に追加され、エディタ画面に開かれます。
   DevicesComponent.razor 内の既存の記述はいったんすべて削除してから、以下のとおり記述します。
   - 「@code {～}」コードブロックを作成し、Device 型のフィールド Device を定義します。
   - Device フィールドには、ダミーデータとして適当な内容で Device オブジェクトを new して割り当てます。
   - HTML で、Device フィールドの内容 (表示名と MAC アドレス) をバインドします。

```
<div class="device">
    <div class="name">
        <span class="caption">デバイス名</span>
        <span class="value">@Device.Name</span>
    </div>
    <div class="mac-address">
        <span class="caption">MAC アドレス</span>
        <span class="value">@Device.MACAddress</span>
    </div>
</div>

@code {
    Device Device = new Device
    {
        Name = "Odin",
        MACAddress = "00:15:5D:52:CA:B6"
    };
}
```

Step 6.　デバイス一覧ページの実装 – App コンポーネント内への埋め込み

概要

ここまでの手順で、Device オブジェクトを表示する新しい Blazor コンポーネント "DevicesComponent" が実装できました。
ですが、これだけではまだ、DevicesComponent コンポーネントはどこからも使われていません。
よってこのままでは、DevicesComponent コンポーネントはブラウザ上に表示されません。

そこで、この DevicesComponent コンポーネントを App コンポーネント内に埋め込むことで、ブラウザ上に表示するようにします。

Blazor コンポーネント (.razor ファイル) は、その .razor フォルダが置かれている**サブフォルダ名の名前空間**、および、**ファイル名と同じクラス名**をタグ名として、他のコンポーネント内からそのタグ名を記述することで参照、埋め込むことができます。

手順

はじめに、DevicesComponent コンポーネントをクラス名のタグだけで参照できるように、名前空間を開いておくことにします。

このような目的で便利に使える特別な .razor ファイルとして、**_Imports.razor ファイル**があります。

この _Imports.razor ファイルは、いずれの .razor ファイルにも読み込まれる特別なファイルであるため、いずれの .razor ファイルからもよく使われる名前空間を @using 節で開いておく、などといった使われ方をします。

本自習書で用意したボイラープレートにも _Imports.razor ファイルが含まれています。

Visual Studio でこの _Imports.razor ファイルを開き、Pages フォルダに配置される .razor ファイルの名前空間 "BlazorWOL.Client.Pages" を開いておく @using 節を追加してください。

```
...前半変更なし...
@using BlazorWOL.Client
@using BlazorWOL.Shared
@using BlazorWOL.Client.Pages
```

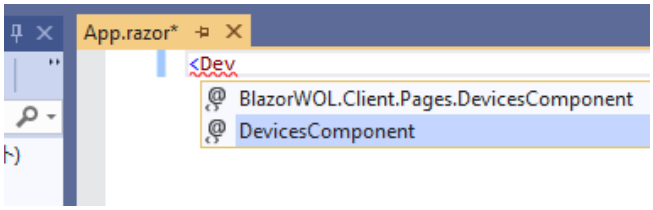
次は App.razor です。

App.razor を Visual Studio で開き、既存の内容をすべて削除します。

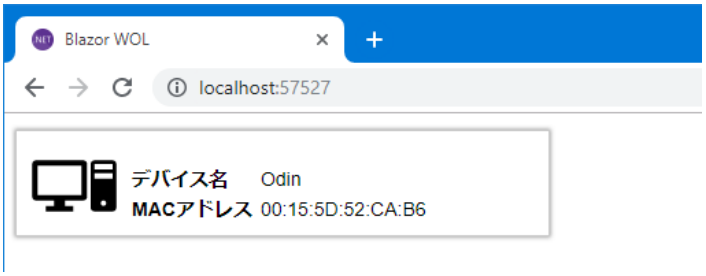
そして、下記のように DevicesComponent コンポーネントのクラス名のタグを追記します。

```
<DevicesComponent></DevicesComponent>
```

なお、"Dev～" と、ある程度 Blazor コンポーネント名を入力すると、下図のとおり**インテリセンスでコンポーネント名の候補に挙がってきます**ので、この候補から選択して入力するとよいでしょう。



以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行すると、ダミーデータとして用意した Device オブジェクトがブラウザ上に表示されます (下図)。



Step 7.　デバイス一覧ページ - リスト化 (繰り返し)

概要

ひとつの Device オブジェクトを表示するところまではできました。

次は DevicesComponent コンポーネントを改造し、複数の Device オブジェクトを表示できるようにしましょう。

> ※ただし、まだこの段階では、複数表示する Device オブジェクト群は、ダミーデータとして即値で用意します。

複数のオブジェクトの表示には、C# の foreach 構文による繰り返しで実装します。

この構文は、ASP.NET Core MVC のサーバー側ビュー実装における Razor 構文と同じです。

手順

1. DevicesComponent.razor を Visual Studio 内で開きます。
2. コードブロック中、Device 型のフィールド Device の定義をいったん削除します。
   代わりに、Device 型の**配列**のフィールド Device**s** に書き換えます。
3. Devices フィールドに適当なダミーデータを割り当てます。

```
Device[] Devices = {
    new Device {Name = "Odin", MACAddress = "00:15:5D:52:CA:B6"},
    new Device {Name = "Thor", MACAddress = "00:0C:29:30:7D:5D"},
    new Device {Name = "Fenrir", MACAddress = "00:50:56:01:43:86"}
};
```

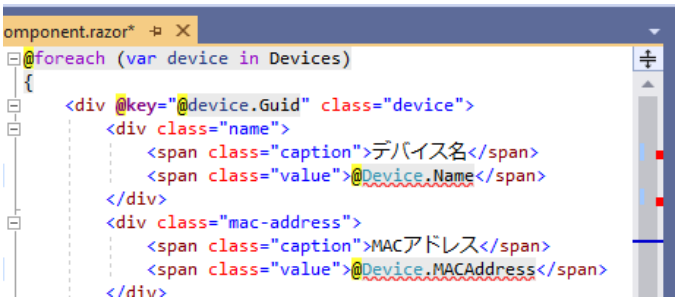
4. HTML全体を、@foreach (var device in Devices){ ～ } ブロックで囲みます。
5. foreach ループ直下のいちばん親の div 要素に、@key ディレクティブで、列挙するオブジェクトを一意に識別できるキーを指定します。
   列挙する個々の Device オブジェクトを一意に識別できる因子は Id プロパティなので、これを指定します。

6. フィールド変数 "Device" にバインドしていた箇所を、foreachのループ変数 "device" に書き換えます。

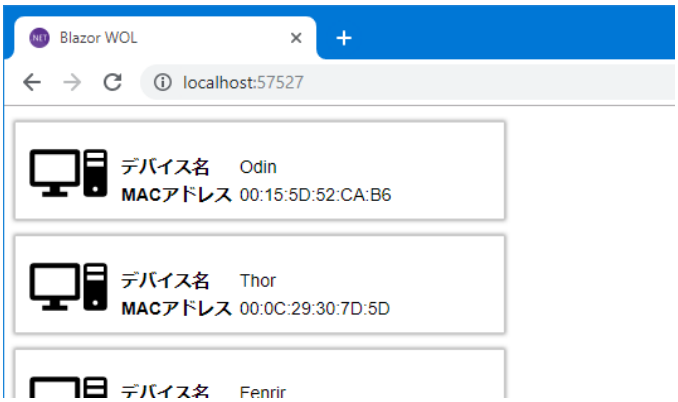
```
@foreach (var device in Devices)
{
    <div @key="device.Id" class="device">
        <div class="name">
            <span class="caption">デバイス名</span>
            <span class="value">@device.Name</span>
        </div>
        <div class="mac-address">
            <span class="caption">MAC アドレス</span>
            <span class="value">@device.MACAddress</span>
        </div>
    </div>
}
```

> ※ ループによるオブジェクトの列挙中に、@key ディレクティブを使って個々のオブジェクトを識別する必要については、下記記事中の "New Razor features" - "@key" の節を参照ください。
>
> *"ASP.NET Core and Blazor updates in .NET Core 3.0 Preview 6"*
>
> https://devblogs.microsoft.com/aspnet/asp-net-core-and-blazor-updates-in-net-core-3-0-preview-6/

なお、上記編集中、Visual Studio のエディタ内には、コードの変更に伴って不整合が生じた箇所は、下図のように赤波線で表示され、スクロールバーにも赤いインジケーターで表示されます。



この機能により、まだ変更・修正が残されている個所がどこであるかを容易に把握できたり、ビルドするまでもなく不整合個所を発見したりすることができます。



以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行すると、ダミーデータとして用意した Device オブジェクトが表示されます(左図)。

Step 8.　デバイス情報の取得・登録を行うサービスの実装 – DI の使用

概要

引き続き、デバイスの追加や編集のユーザーインターフェースの作りこみへと進んでいきます。
ですがその前に、デバイス情報の蓄積およびデバイス一覧の取得や追加などを行うサービスクラスを実装して、これを使うようにしましょう。

サービスクラスでデバイス情報を取り扱うことにより、このあと実装するルーティング機構によってアクティブな Blazor コンポーネントが差し変わるようになっても、デバイス情報が永続化して取り扱われるようになります。
また、さらにはサーバー側の実装が進んで、デバイス情報をサーバー側で永続化し、クライアント側と HTTP 通信でデバイス情報をやりとりするようになっても、このサービスクラスでその実装変更を吸収できるようになります。

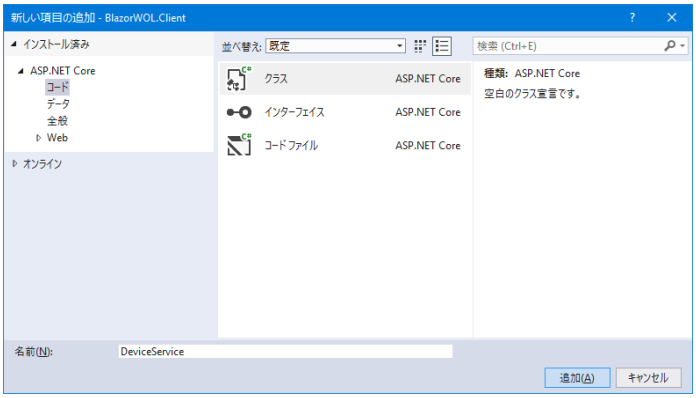
サービスオブジェクトは、Blazor に備わっている **DI (Dependency Injection:依存性注入)** 機構を介して、各 Blazor コンポーネントから使用します。
Blazor アプリケーションの開始地点でサービスオブジェクトを Blazor の DI 機構に登録しておくいっぽう、各 Blazor コンポーネントでは、**「@inject」ディレクティブ**を記述することで、必要なサービスオブジェクトの参照を DI 機構から入手します。

ということで、デバイス情報を蓄え、デバイス一覧の取得や追加の操作を提供するサービスクラスとして **"DeviceService" クラス**を実装し、Blazor の DI 機構に登録、使用することにします。

手順

1. Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトを右クリックし、メニューから [追加(D)]-[クラス(C)...] をクリックします。

2. 「新しい項目の追加」ダイアログが現れるので、[名前(N)] に "DeviceService" と入力して [追加(A)] ボタンをクリックします。



3. DeviceService.cs ファイルが追加されるので、内容を以下のとおり実装します。
   - プライベートなプロパティとして Device クラスのリストを持たせます。
   - この Device クラスのリストに、とりあえず今はまだ、ダミーデータを設定しておきます。
   - 格納している Device オブジェクトの集合を返す、"GetDevices()" メソッドを追加・実装します。

   最終的に DeviceService.cs は下記のようになります。

```
using System.Collections.Generic;
using BlazorWOL.Shared;

namespace BlazorWOL.Client
{
    public class DeviceService
    {
        private List<Device> Devices { get; } = new List<Device> {
            new Device {Name = "Odin", MACAddress = "00:15:5D:52:CA:B6"},
            new Device {Name = "Thor", MACAddress = "00:0C:29:30:7D:5D"},
            new Device {Name = "Fenrir", MACAddress = "00:50:56:01:43:86"}
        };

        public IEnumerable<Device> GetDevices()
        {
            return Devices;
        }
    }
}
```

4. 次に、こうして実装した DeviceService クラスを、Blazor の DI 機構に登録します。
   BlazorWOL.Client プロジェクトの Program.cs を Visual Studio で開き、Main メソッド中、既存の「`builder.Services.AddTransient(sp => new HttpClient { BaseAddress = ...});`」行の直後に「`builder.Services.AddSingleton<DeviceService>();`」と追記して、DI 機構への DeviceService クラスの登録処理を記載します。

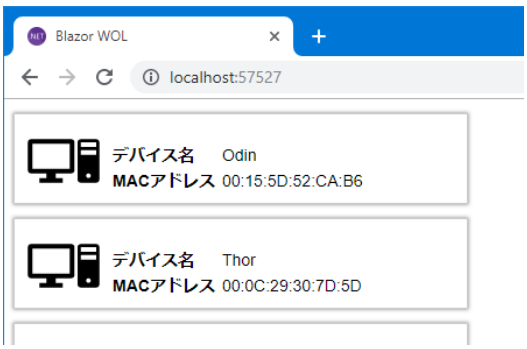
```
...
public static async Task Main(string[] args)
{
    ...
    builder.Services.AddTransient(sp => new HttpClient { BaseAddress = ... });
    builder.Services.AddSingleton<DeviceService>();
    ...
```

5. 次は、こうして DI 機構に登録された DeviceService オブジェクトを DevicesComponent で使用します。
   BlazorWOL.Client プロジェクトの DevicesComponent.razor を Visual Studio で開き、行頭に「`@inject DeviceService DeviceService`」の行を追加します。
   この @inject ディレクティブにより、Blazor コンポーネント DevicesComponent のプロパティとして、**DeviceService 型のプロパティ DeviceService が追加**されます。この DeviceService プロパティには、Blazor の DI 機構に登録された **DeviceService オブジェクトが自動で設定済み**となる仕組みです。
6. (コンポーネント内で直に記載していたダミーデータではなく) DI 経由で入手した DeviceService オブジェクトから、デバイス情報一覧を取得するように、DevicesComponent.razor を変更します。
   まずは、DevicesComponent.razor のコードブロック中、メンバーフィールド Devices の型を `Device[]` から `IEnumerable<Device>` に変更し、初期設定していたダミーデータは空の配列に置き換えます。
7. 次に、メンバーフィールド Devices に、DeviceService オブジェクトの GetDevices() メソッドで取得したデバイス情報一覧を設定する処理を足します。
   この処理は Blazor コンポーネントが備える OnInitialized 仮想メソッドをオーバーライドして行います。

```
@inject DeviceService DeviceService
...この部分は変更なし...
@code {
    IEnumerable<Device> Devices = new Device[0];

    protected override void OnInitialized()
    {
        Devices = DeviceService.GetDevices();
    }
}
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してください。すると、見た目は前回とまったく変わりませんが、サービス経由で取得した Device オブジェクトがブラウザ上に表示されることが確認できます (右図)。



Step 9.　非同期処理化

概要

さて、このままデバイス情報の追加・編集へと邁進してもよいのですが、いずれデバイス情報をサーバー側で永続化して HTTP 通信でやりとりするようになった際は、サーバー側とのやりとりは**非同期処理が必須**となります。

そこで、今はまだメモリ上の List を使ったダミーデータ実装ではありますが、この時点で、デバイス情報サービス (DeviceService) が公開するメソッドを**非同期バージョンに改造**しておきましょう。
今のうちにこの改造を済ませておけば、最終的にサーバー側実装が進んだ時に、同期処理を非同期処理に書き換える手間がなくなります。

Blazor は JavaScript と同じようにブラウザ上の WebAssembly 実行エンジンで動いていますが、C#による実装なので、**一般的な C#プログラミングと同じく async/await 構文や Task クラスを使用できます**。

手順

1. DeviceService.cs を Visual Studio で開き、GetDevices() メソッドを非同期処理に書き換えます。
   - メソッドの戻り値の前に、キーワード "async" を書き足します。
   - メソッドの戻り値を、Task<戻り値に返したい値の型>型に変更します。
   （※名前空間 System.Threading.Tasks が必要になるので using も追加しておきます。）
   - メソッド名の末尾に "～Async" を追記します。
   - ダミーのデバイス情報の返し方は、Task クラスの FromResult 静的メソッドを経由することであえて非同期化し、これを await して非同期処理完了待ちして返すようにします。
   変更後の GetDevices() メソッドは下記のようになります。

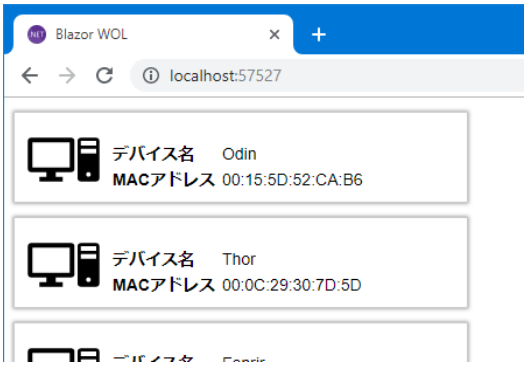
```
using System.Threading.Tasks;
...
public async Task<IEnumerable<Device>> GetDevicesAsync()
{
    return await Task.FromResult(Devices);
}
```

2. 次は、利用する側、DevicesComponent を変更します。
   コンポーネント初期化のタイミングの仮想メソッドとして、OnInitialized ではなく、非同期処理対応バージョンの OnInitializedAsync 仮想メソッドのオーバーライドに変更し、async キーワードを追加します。
   そして DeviceService オブジェクトの GetDevicesAsync() メソッドを await して呼び出し、結果にデバイス一覧が返ってきますからこれを Devices メンバーフィールドに格納します。

変更後の OnInitializedAsync 仮想メソッドは下記のようになります。

```
protected override async Task OnInitializedAsync()
{
    Devices = await DeviceService.GetDevicesAsync();
}
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行すると、引き続きまだ見た目は前回とまったく変わりませんが、非同期処理に改造しても、正しくダミーデータとして用意した Device オブジェクトがブラウザ上に表示されることが確認できます (下図)。



Step 10. デバイス追加フォームを追記 – 入力とイベントのバインディング

概要

いよいよデバイスの追加ができるようにしましょう。

デバイス一覧 DevicesComponent の HTML 末尾に、デバイス情報の入力欄 (input type=text 要素) を設け、この入力欄と DevicesComponent のプロパティとを双方向バインドすることで、入力内容を取得できるようにします。

また、「OK」ボタンを設け、このボタンのクリックイベントをハンドルして、デバイス情報サービス (DeviceService) にデバイス情報の追加を行うようにします。
このために、デバイス情報サービス (DeviceService) には、デバイス情報を追加するメソッドを実装します。

デバイス情報サービス (DeviceService) を介してデバイス情報一式が更新されれば、データバインディングの仕掛けによって、ブラウザ上のデバイス一覧の表示も更新されます。

手順

1. まずはデバイス情報サービス (DeviceService) にデバイス情報を追加するためのメソッドを実装しましょう。
   メソッド名は AddDeviceAsync とし、(今はまだメモリ上の List に記憶するダミー実装ですが、あえて) 非同期処理として実装します。
   BlazorWOL.Client プロジェクトの DeviceService.cs を Visual Studio で開き、AddDeviceAsync 非同期メソ

ッドを追加します。
プライベートプロパティの Devices リストへのオブジェクトの追加は同期処理なのですが、サーバー側実装との HTTP 通信化の際に非同期処理に改造することをふまえ、あえて Task.Run() でくるむことで非同期処理に仕立てます。
AddDeviceAsync メソッドの実装は下記のようになります。

> ※実装作業中、名前空間の不足が発生したら、**Ctrl + . によるクイックフィックス**などによって、適宜、using 節を追加してください。

```
public async Task AddDeviceAsync(Device device)
{
    await Task.Run(() => Devices.Add(device));
}
```

2. 続けて、ユーザーインターフェースの作りこみをします。
   BlazorWOL.Client プロジェクトの DevicesComponent.razor を Visual Studio で開き、コードブロック中に、新規デバイス入力フォームとバインドするためのメンバーフィールド "NewDevice" を追加します。
   このメンバーフィールドは新規追加用のオブジェクトを設定しておきます。

```
Device NewDevice = new Device();
```

3. また、このあとコーディングする OK ボタンがクリックされたときの処理として、OnOK メソッドを同じくコードブロック内に追加します。
   OnOK メソッド内では、メンバーフィールド NewDevice を、デバイス情報サービス (DeviceService) に先ほど実装した AddDeviceAsync() メソッドに引き渡して、デバイス情報の追加を行います。
   すかさず、再びの新規追加に備えて、メンバーフィールド NewDevice に新しいデバイス情報オブジェクトを設定しなおします。
   なお、非同期処理が絡むので、OnOK メソッドは Task を戻り値とする async キーワード付きの非同期メソッドとして実装し、AddDeviceAsync() 非同期メソッドの処理待ちのために await キーワードを付与して呼び出すようにします。

```
async Task OnOK()
{
    await DeviceService.AddDeviceAsync(NewDevice);
    NewDevice = new Device();
}
```

4. あとは新規デバイス入力用のフォームの HTML をコーディングしましょう。
   DevicesComponent の HTML の末尾に、以下のように HTML をコーディングします。

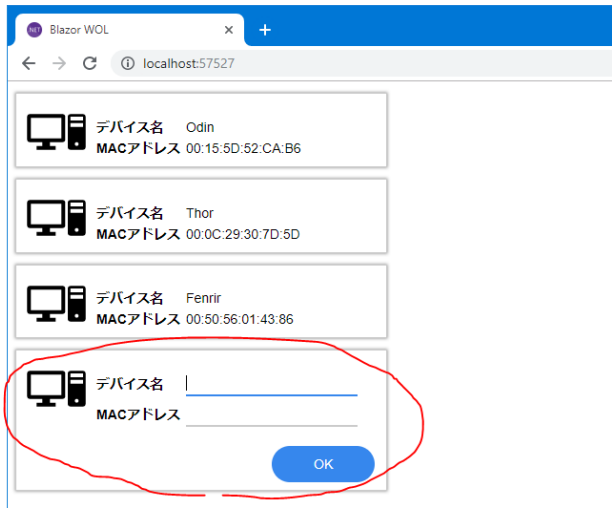
まず input 要素による入力欄は @bind ディレクティブで DevicesComponent のメンバーとバインドします。OK ボタン (button 要素) のクリックイベントのハンドリングは、@onclick= に続けてハンドラメソッドを指定することで行います。

DevicesComponent に追加される HTML は下記のようになります。

```
<div class="device">
    <div class="name">
        <span class="caption">デバイス名</span>
        <span class="input-field">
            <input type="text" @bind="NewDevice.Name" />
        </span>
    </div>
    <div class="mac-address">
        <span class="caption">MAC アドレス</span>
        <span class="input-field">
            <input type="text" @bind="NewDevice.MACAddress" />
        </span>
    </div>
    <div class="actions">
        <button class="button" @onclick="OnOK">OK</button>
    </div>
</div>
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行すると、デバイス一覧の下に、新規デバイス追加用の入力欄が増えているのが確認できます (下図)。

そしてこの入力欄に何か適当に入力して OK ボタンをクリックすると、入力した内容が新規デバイスとしてデバイス一覧に追加されることが確認できます。



Step 11. 入力内容のチェックと正規化

概要

ここまででデバイス情報の新規追加ができるようになりました。
しかしながら、デバイス名や MAC アドレスが空欄のままでも、デバイス情報の追加ができてしまいます。

そこで、下記の入力チェックを実装してみます。

- デバイス名および MAC アドレスが空欄でないこと
- デバイス名は 20 文字まで
- MAC アドレスの書式が、16 進数文字列 2 桁の、コロンかハイフン区切りによる 6 パートであること (16 進数文字列の英字大小は問わない)

また、入力チェックとあわせて、MAC アドレスについては下記正規化を行うこととします。

- 16 進数文字列の英字は大文字に変換
- 区切り文字のハイフンはコロンに変換

さて、まずは入力チェックの実装方法ですが、本自習書では
「入力対象のオブジェクトのクラス宣言において、各プロパティに "属性" で適格条件を付記する」
という技法を用いたいと思います。

BlazorWOL アプリにおいては、"入力対象のオブジェクトのクラス" は Device クラスです。すなわち、Device クラスの各プロパティに、入力チェックの適格条件を、別途用意されている属性クラスで付記します。

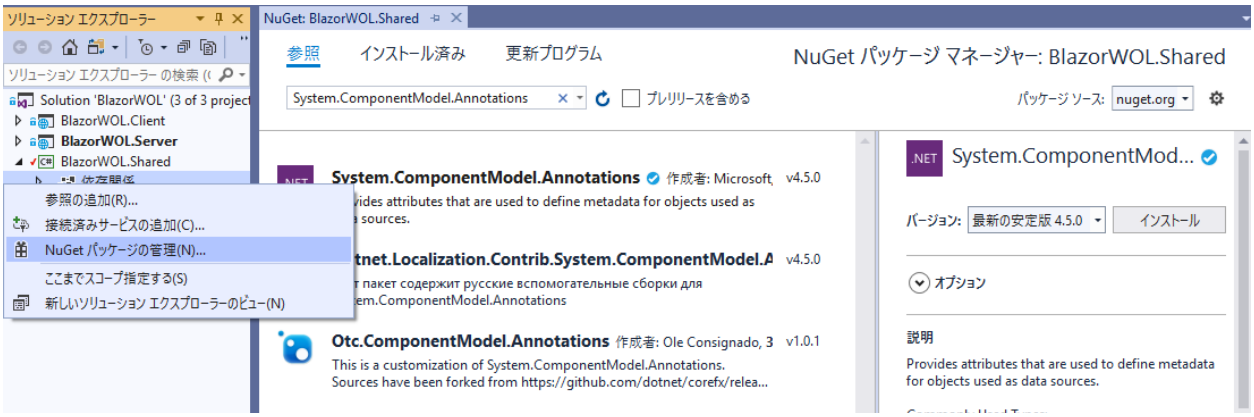
手順

適格条件を付記する対象である Device クラスを収録しているのは、BlazorWOL.Shared プロジェクトです。

他方、適格条件を付記するための代表的な属性クラス群を収録している System.ComponentModel.Annotations という NuGet パッケージがあります。

そこで、BlazorWOL.Shared プロジェクトに System.ComponentModel.Annotations NuGet パッケージ参照を追加します。

これを Visual Studio 上からグラフィカルユーザーインターフェース経由で行なうには、Visual Studio のソリューションエクスプローラ上から BlazorWOL.Shared プロジェクトを右クリックし、[NuGet パッケージの管理(N)...] を選択、開いた NuGet パッケージマネージャーウィンドウにて "System.ComponentModel.Annotations" を検索して選択の上、 [インストール] をクリックします。



※ dotnet CLI でこれを行なうには、コマンドプロンプト(ターミナル)にてカレントディレクトリを BlazorWOL.Shared プロジェクトがあるフォルダに移動した上で、下記のコマンドを実行します。

```
> dotnet add package System.ComponentModel.Annotations
```

これで適格条件付記用の属性クラスを使う準備が整いました。

続けて、BlazorWOL.Shared プロジェクト内の Device.cs を開き、適格条件付記用の属性クラスを収録している名前空間 "System.ComponentModel.DataAnnotations" を using 節で開いておきます。

```
using System;
using System.ComponentModel.DataAnnotations;

namespace BlazorWOL.Shared
{
    public class Device
    {
```

次に各プロパティを適格条件付記用の属性クラスで修飾していきます。

まずはデバイスの名称を示す Name プロパティに対し、

- 空欄でないこと (入力必須)
- 最大 20 文字まで

の適格条件を記述します。

これら条件はそれぞれ、以下の属性クラスの付加によって表現します。

- RequiredAttribute
- StringLengthAttribute

Required 属性は引数なしでプロパティに付記すればよいです。
StringLength 属性は最大文字数を引数に指定します。
また、各々の属性には、その属性が示す条件を入力内容が逸脱していた場合のエラーメッセージを指定します。

```
...
public class Device
{
    public Guid Id { get; set; } = Guid.NewGuid();

    [Required(ErrorMessage = "デバイス名を入力してください。")]
    [StringLength(20, ErrorMessage = "デバイス名は 20 文字までです。")]
    public string Name { get; set; }
...
```

同じ要領で、デバイスの MAC アドレスを示す MACAddress プロパティにも属性を付記します。

MACAddress プロパティについては、Name プロパティと同様に入力必須であることを示す Required 属性を付記することに加え、入力内容が "16 進数文字列 2 桁の、コロンかハイフン区切りによる 6 パート" であることの適格条件も属性で記述します。
この目的には、入力内容が指定された正規表現パターンに合致するかどうかを適格条件として指定できる RegularExpressionAttribute 属性クラスを用います。

```
[Required(ErrorMessage = "MAC アドレスを入力してください。")]
[RegularExpression(@"(?i)^[\da-f]{2}((:|-)[\da-f]{2}){5}$", ErrorMessage = "MAC
アドレスの書式が正しくありません。")]
public string MACAddress { get; set; }
...
```

以上で、Device クラスに対する各プロパティの適格条件を属性で記述することができました。

次は入力フォームで、これら適格条件の属性を参照して入力チェックが行なわれるように実装していきます。

Blazor には、このような入力対象オブジェクトのクラス定義に記述された属性に基づいて入力チェックを行なうコンポーネントが、Microsoft.AspNetCore.Components.Forms 名前空間に用意されています。

そこで、デバイス情報の入力フォームを、それら Microsoft.AspNetCore.Components.Forms 名前空間のコンポーネントを使った実装に書き換えていきます。

まず使うのは、**EditForm コンポーネント**です。
BlazorWOL.Client プロジェクトの DevicesComponent.razor を Visual Studio で開き、既存の入力フォーム部分を **EditForm コンポーネント**でくるむようにします。

このとき、入力対象のオブジェクト (ここでは NewDevice フィールド) を、EditForm コンポーネントの **Model プロパティ**に引き渡し、また、入力内容の適格条件がすべて満たされたうえでフォーム送信が発動したときに呼び出すメソッド (ここでは OnOK メソッド) を **OnValidSubmit プロパティ**に指定します。

```
<div class="device">
  <EditForm Model="NewDevice" OnValidSubmit="OnOK">
    <div class="name">
      <span class="caption">デバイス名</span>
      ...
      <button class="button" onclick="OnOK">OK</button>
    </div>
  </EditForm>
</div>
```

次に、**DataAnnotationsValidator コンポーネント**を、EditForm コンポーネント内に追加します。
この DataAnnotationsValidator コンポーネントが、外側の EditForm コンポーネントと協調し、入力対象オブジェクトのクラス定義における属性指定に基づいた入力内容の適格判定を司ります。

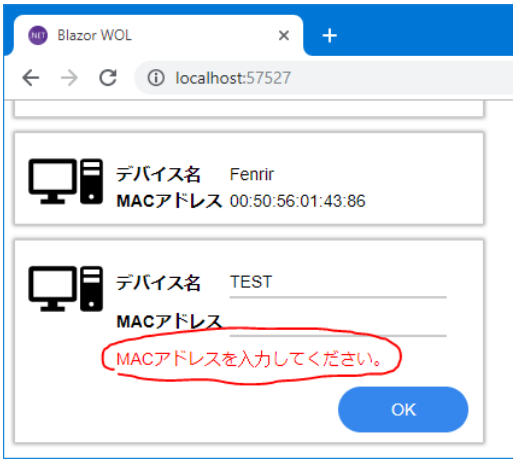
```
<div class="device">
  <EditForm Model="NewDevice" OnValidSubmit="OnOK">
    <DataAnnotationsValidator />
      ...
  </EditForm>
</div>
```

OK ボタンですが、OK ボタンのクリックイベントから直接にデバイス追加処理を呼び出すのをやめて、いま追加した EditForm コンポーネントからの、すべての入力チェックがパスしたときに発動する OnValidSubmit イベントに任せるようにしますので、OK ボタンの onclick イベントハンドラは削除しておきます。

```
        ...
        <div class="actions">
            <button class="button">OK</button>
        </div>
    </div>
</EditForm>
```

あとは、EditForm および DataAnnotationsValidator コンポーネントが実施した入力チェックの結果、不備があった場合の**入力エラーメッセージを表示する機能をもつ、ValidationSummary コンポーネント**を、EditForm コンポーネント内に配置します。
今回は、OK ボタンの上に配置することにします。

```
        ...
        </div>
        <div class="error-message">
            <ValidationSummary></ValidationSummary>
        </div>
        <div class="actions">
            <button class="button">OK</button>
        ...
```



ここまでできたら、いったん実行して動作を試してみます。
すべての変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行します。

すると、デバイス情報追加欄について、**Device クラスの各プロパティに付記した属性指定のとおりに入力チェックが働き、**入力チェックでエラーが発生したら、エラーメッセージが ValidationSummary コンポーネント内に表示されているのが確認できます (左図)。

仕上げとして、入力された MAC アドレスを正規化する処理を、OnOK メソッド内に書き足します。

```
    async Task OnOK()
    {
        // MAC アドレスの正規化
        NewDevice.MACAddress = NewDevice.MACAddress.Replace("-", ":").ToUpper();

        await DeviceService.AddDeviceAsync(NewDevice);
        NewDevice = new Device();
    }
```

以上で、デバイス追加フォームにおける入力内容のチェックと正規化ができるようになりました。

> ※ Blazor で用意されている入力フォーム系コンポーネントとしては他にも、InputText コンポーネントなどの input 要素に対応するコンポーネントもあります。これら Input 系のコンポーネントは、入力チェックエラーが発生したときに"invalid"という CSS クラス名を自身に追加するなどのいくつかの便利な機能を備えます。
> 詳細は Blazor 公式ドキュメントサイトの下記コンテンツを参照ください。
> https://docs.microsoft.com/en-us/aspnet/core/blazor/forms-validation?view=aspnetcore-3.0

補足 - なぜモデルクラスの属性で適格条件を記述するのか - データアクセスを例に

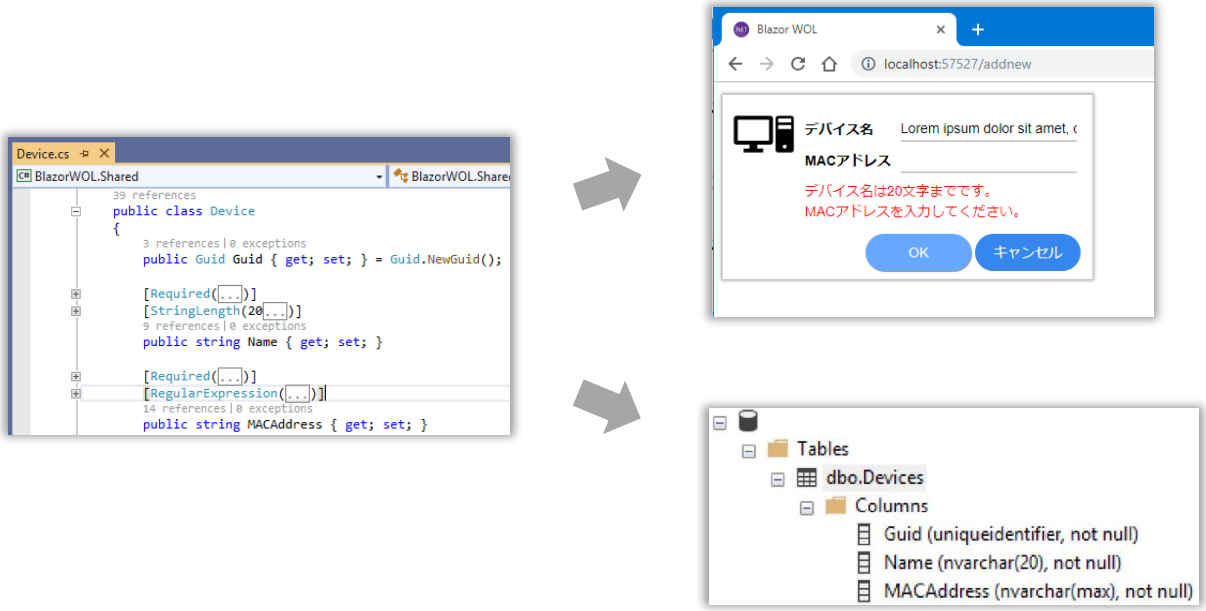
ASP.NET Core による Web アプリ開発においては、その要件として、リレーショナルデータベースに情報やデータを保存したり読み出したりすることがよくあります。

そのような要件に対し、Entity Framework Core というデータアクセス用のフレームワークを採用し、かつ、"コードファースト" と呼ばれる技法でリレーショナルデータベースのテーブル構築も含めて実行することがあります。

この方式では、
- モデルクラスをリレーショナルデータベースのテーブルに、
- モデルクラスの各プロパティをテーブルの列に
対応付け (マッピング) して、テーブル定義の決定と構築、及び、モデルクラスのオブジェクトを媒介としたテーブル行の読み書きを行ないます。

このテーブル定義の決定と構築に際して、モデルクラスの各プロパティに付与した適格条件属性が使われるのです。

例えば、入力必須の入力チェックのために付与した Required 属性は、対応する列の定義を NULL 不許可 ("NOT NULL") としますし、最大文字数の入力チェックのために付与した StringLength(*n*) 属性は、対応する列の定義における最大文字数 ("NVARCHAR(n)") として使われます。



このように、モデルクラスの各プロパティに属性で適格条件を記述する方式であれば、

- ユーザーインターフェース上での入力チェック条件や、
- 永続化先のデータベース定義、
- 他にも、上記では触れませんでしたが、ASP.NET Core MVC コントローラでの要求バインド時のチェック

などなど、さまざまな場面で必要とされる適格条件定義を、**モデルクラス定義の一箇所で実装、一元管理できる**、という利点があるのです。

補足 - Visual Studio 2019 上での Blazor WebAssembly プログラムのデバッグ

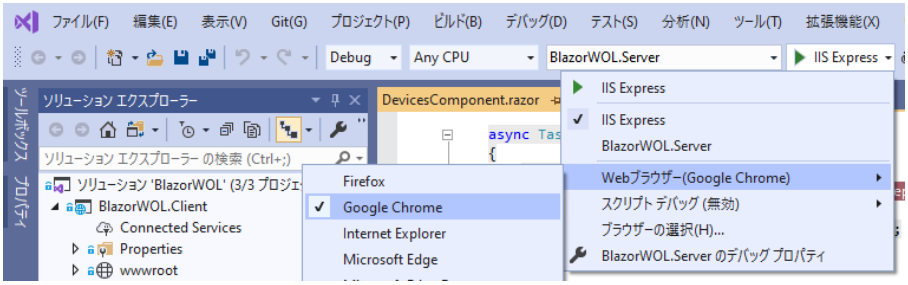
さてここで、もう少し脱線して、Visual Studio 2019 上での Blazor WebAssembly プログラムのデバッグについて触れておきます。

まだまだ荒削りではあるのですが、2020 年 4 月から、Visual Studio 2019 上で Blazor WebAssembly プログラムのデバッグ対応が始まっています。
すなわち、Blazor WebAssembly プログラムに対し、ブレークポイントの設定と停止や、変数のウォッチなどが、Visual Studio 2019 上で行える、ということです。

但しブラウザ側の対応も必要で、2020 年 5 月時点では、以下の Web ブラウザでのみデバッグできるようです。

- Microsoft Edge (Chromium 版)
- Google Chrome

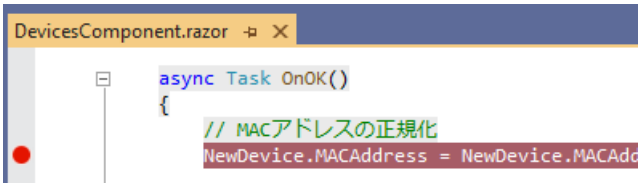
Visual Studio でいずれのブラウザを使うかは、ツールバー上のドロップダウンメニューから選択できます(下図)。



上記いずれかの Web ブラウザを起動するように Visual Studio 上で設定済みでしたら、試しに、このステップで実装した入力内容正規化の部分で試してみましょう。

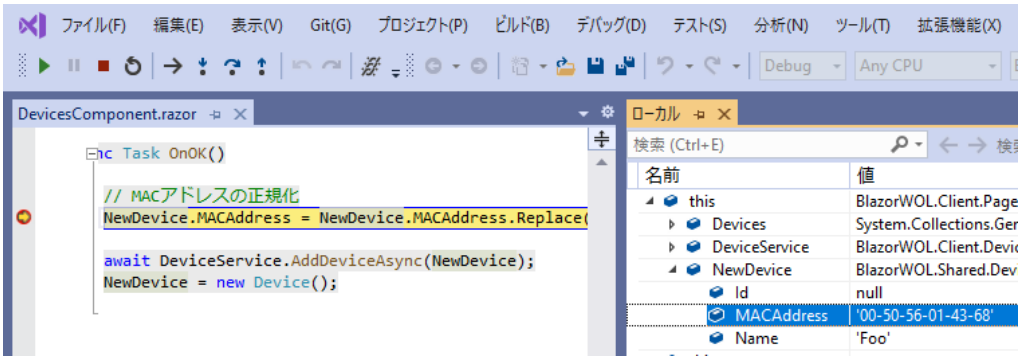
まずは DevicesComponent.razor ファイルを Visual Studio 内に開き、OnOK メソッド内の MAC アドレスの正規化を行なっている行にブレークポイントを設定します。具体的には、この行の行頭マージンをマウスでクリックするか、この行にキャレットがある状態でキーボードの F9 を押します。

この操作により行頭マージンに赤い丸印が表示され、ブレークポイントが設定されたことが示されます (下図)。



この状態で、キーボードの F5 キーを (Ctrl キーは押さないで!) 押して実行開始してみます。
すると Web ブラウザが立ち上がり、Blazor WebAssembly アプリが読み込まれてデバイス追加フォーム表示されます。
デバイス追加フォームに何か適切な入力をして [OK] ボタンをクリックしてみます。

すると、ちゃんと Visual Studio 内にて、コードの実行がブレークポイントで停止し、ローカル変数の内容を見ることができるのがわかります（下図）。



このように、Visual Studio 2019 上で F5 実行するだけで、Blazor WebAssembly プログラムのデバッグが可能となっています。

ただし現時点ではまだまだデバッグ機能は開発途上です。
例えば、非同期処理をステップ実行することがうまくできなかったり、配列の内容が変数ウィンドウで見られなかったり、デバッガがアタッチされる前の OnInitialized などのタイミングでブレークできなかったり... と、できないことが多々あります。

とはいえ、このデバッグ機能はとても魅力的です。
Web ブラウザ側の WebAssembly エンジンで実行されているクライアント側処理の C#コードから、サーバー側 ASP.NET Core 実装の C#コードまで、透過的に、且つ同じ IDE 上で一貫して、ブレークポイントを張りながら処理の流れを追いかけることも可能になるからです。

今後の Visual Studio の機能向上に期待しましょう。

> ※ Visual Studio Code でも、Blazor WebAssembly プログラムのデバッグ実行が可能となっているようです。
> 詳細は下記 ASP.NET 開発チームのブログ記事を参照ください。
> https://devblogs.microsoft.com/aspnet/blazor-webassembly-3-2-0-preview-3-release-now-available/

Step 12. デバイス追加を独立した URL に切り出し - ルーティング

概要

引き続き、ユーザーインターフェースを拡充していきます。

次は、デバイス情報追加のユーザーインターフェースを、独立した URL に切り出しましょう。
つまり、デバイス情報一覧のページと、デバイス情報追加のページを分け、これらページ間を往復・遷移するユーザーインターフェースとします。

このため、URL と該当する Blazor コンポーネントとの対応付け・割り当てを行うために、今までは使ってこなかった Blazor のルーティング機構を有効にします。

Blazor のルーティング機構において、どの URL にどの Blazor コンポーネントを割り当てるかは、各 Blazor コンポーネント自身の記述中で、@page ディレクティブを用いて URL パターンを記述することで行います。

手順

1. BlazorWOL.Client プロジェクトの、いちばん根本の Blazor コンポーネントである App.razor を開きます。
   現状では、デバイス情報一覧である DevicesComponent コンポーネントの埋め込みが記述されています。これを削除し、代わりに下記のとおり、Blazor に備え付けの **Router コンポーネントの使用**に書き換えます。

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

   Router コンポーネントは子の描画要素として、**Found** と **NotFound** の 2 つの描画要素を取ります。
   Found 描画要素は、ブラウザのアドレスバーに現れた URL パスに割り当てられた対象のコンポーネントが見つかったときに、その内容が描画されます。
   通常は上記のように **RouteView コンポーネント**を指定して、URL パスに一致する割り当てを持つコンポーネントを描画するようにします。

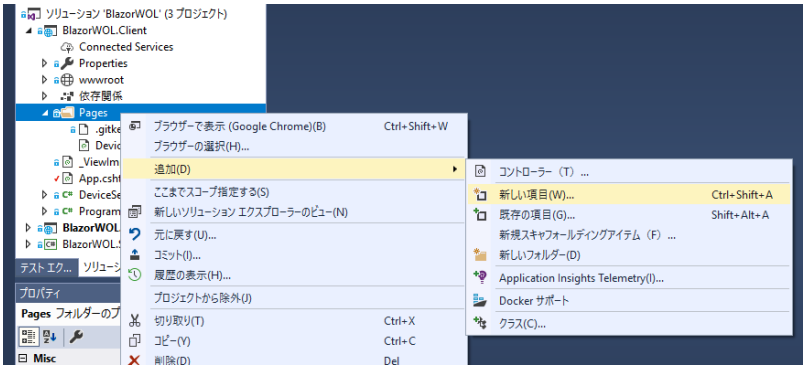
   NotFound 描画要素はこれとは逆に、URL パスに一致するコンポーネントが見つからなかったときにその内容が描画されます。
   通常は上記のように、"指定された URL では表示するものがない" 旨のメッセージを表示するようにします。

2. これで Blazor のルーティング機構が有効となりました。
   しかし、このままでは、どの URL のときにどの Blazor コンポーネントを描画するのかが定まっていません。
   さしずめ、デバイス情報一覧 DevicesComponent を、ルート URL「/」に割り当てたいと思います。
   BlazorWOL.Client プロジェクトの DevicesComponent.razor を Visual Studio で開き、行頭に「`@page "/"`」の記述を追加します。

3. ここまでの変更を保存したら、いちどプロジェクトをビルドしてブラウザで再読み込みを実行し、とりあえず見た目上の振る舞いは変更前と変わりなく正常動作することを確認しておきましょう。

4. 続けて、デバイス情報追加のユーザーインターフェースを独立した Blazor コンポーネントに切り出し、「/addnew」の URL を割り当てましょう。

   新しい Blazor コンポーネントファイルを追加するべく、Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトの Pages フォルダを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします (右図)。

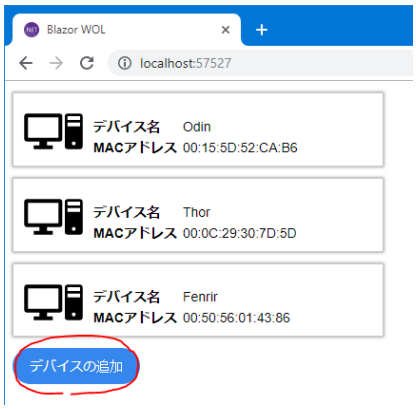


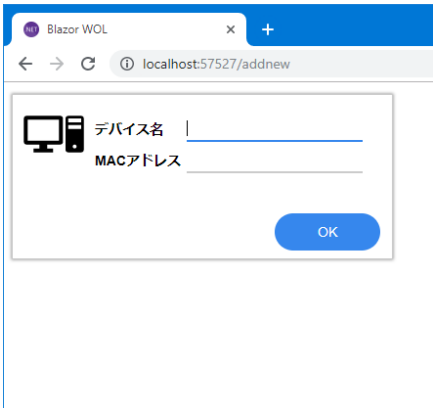
5. 「新しい項目の追加」ダイアログが現れるので、
   - ダイアログ中央のアイテム一覧から「Razor コンポーネント」をクリックして選択して、
   - [名前(N)] に "**AddDevice**" と入力してから
   - [追加(A)] ボタンをクリックします。
6. AddDevice.razor ファイルが Pages フォルダ内に追加され、Visual Studio 内に開かれます。
   AddDevice.razor ファイル内に、以下のとおり実装します。
   - URL ルーティングの指定「`@page "/addnew"`」を行頭に追記
   - デバイス情報サービスを DI 経由で入手する「`@inject DeviceService DeviceService`」を次行に追記
7. さらに続けて、デバイス情報一覧コンポーネント DevicesComponent.razor から、以下の要素をカット (切り取り) してきて AddDevice.razor に貼り付けます。
   - HTML パート中、foreach ループの下に追加した、デバイス情報追加フォームのマークアップ
8. AddDevice.razor に「`@code {～}`」コードブロックを作成し、DevicesComponent.razor から以下のメンバーをカットして貼り付けます。
   - NewDevice フィールド
   - OnOK メソッド
9. 以上で、DevicesComponent.razor から、デバイス情報追加の機能に関する要素をひととおり、AddDevice.razor へ移動することができました。
   最後に、デバイス情報一覧 ("/") に、デバイス情報追加 ("/addnew") へのリンクを設けましょう。
   DevicesComponent.razor を Visual Studio で開き、HTML パート部分の末尾に、"/addnew" へのリンクを下記のように追加します。

```
<div>
    <a class="button" href="/addnew">デバイスの追加</a>
</div>
```

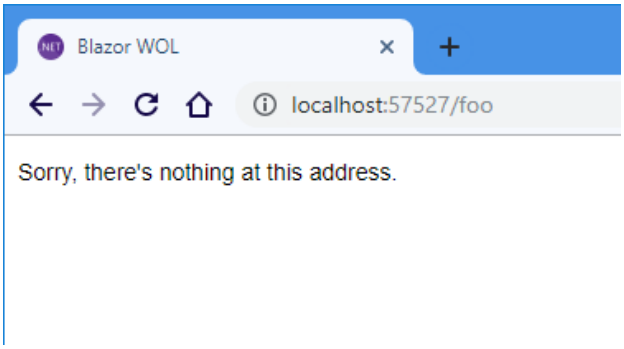
以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報一覧 ("/") に、「デバイスを追加」が増えています。これをクリックすると、URL が "/addnew" に遷移し、デバイス情報追加のユーザーインターフェースが表示されることが確認できます。





なお、デバイス情報追加のページで、デバイス名と MAC アドレスとを正しく入力して「OK」ボタンをクリックしても、デバイス情報追加のページに居座ったままです。
これは、デバイス情報追加のページにて OK ボタンがクリックされた時の動作に、デバイス情報一覧に戻る処理をまだ実装していないためです。
とはいえ、ブラウザの「戻る」でデバイス情報一覧に戻ってみると、OK ボタンクリックして追加したデバイス情報が、たしかにデバイス情報一覧の末尾に追加されていることを確認できます。



また、試しにブラウザのアドレスバーに直接、「/foo」のようにいずれのコンポーネントでもルート定義されていない URL パスを入力してみてください。

App.razor 中に記述した Router コンポーネントの機能により、その Router コンポーネントの NotFound 描画要素がブラウザに表示されることが確認できるはずです（左図）。

Step 13. OK/キャンセルボタンで一覧に戻る - コード中からのページナビゲーション

概要

次は、デバイス情報追加ページで OK ボタンを押したら、一覧ページの URL に戻るように実装していきましょう。ついでに、デバイス情報追加ページにキャンセルボタンも実装しておきます。

Blazor コンポーネント内の C#コード操作で、任意の URL にページ遷移するには、**Blazor に備え付けの NavigationManager サービス**を使います。NavigationManager サービスは DI 機構を介して入手します。

手順

1. まずは、デバイス情報追加コンポーネントにて NavigationManager サービスを DI 経由で入手するよう実装します。BlazorWOL.Client プロジェクトの AddDevice.razor を Visual Studio で開き、DeviceService を注入している次行に、下記のとおり NavigationManager 注入の行を追記します。

```
@page "/adddevice"
@inject DeviceService DeviceService
@inject NavigationManager NavigationManager
```

2. 次に、AddDevice.razor のコードブロック中、OnOK メソッドの最後のほうで、デバイス情報サービスに新規デバイス情報を追加しおわった後の処理を、下記のように NavigationManager を使用して URL "/" に遷移するように書き換えます。

```
    ...
    await DeviceService.AddDeviceAsync(NewDevice);
    NavigationManager.NavigateTo("/");
}
```

3. 以上で、デバイス情報追加ページで OK ボタンをクリックすると、無事デバイス情報を追加できたら、そのままデバイス情報一覧ページ ("/") に遷移するようになります。
   仕上げに、デバイス情報追加ページにキャンセルボタンも取り付けておきましょう。AddDevice.razor の HTML パート中、OK ボタンの HTML 要素の次行に、URL "/" へのリンクとしてキャンセルボタンの HTML を記述します。

```
<div class="actions">
    <button class="button" onclick="OnOK">OK</button>
    <a class="button" href="/">キャンセル</a>
</div>
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報一覧ページから「デバイスを追加」をクリックしてデバイス情報追加ページに遷移したあと、必要事項を入力して「OK」ボタンをクリックしたら、デバイス情報一覧ページに遷移して、かつ、入力したデバイス情報が一覧に追加されていることを確認してください。

また、デバイス情報追加ページにキャンセルボタンも増えており、これをクリックすることで、デバイス情報の追加を行わずに一覧ページに戻れることを確認してください。

Step 14. デバイス情報入力フォームをさらに切り出し
— 子コンポーネントへの変数受け渡しとイベントハンドリング

概要

さて、デバイス情報の "追加" までできるようになりました。次はデバイス情報の "編集" に取り掛かります。

ですがその前に、デバイス情報の入力ユーザーインターフェースを、**"デバイス情報フォーム" コンポーネント**として切り出しておきましょう。

そうすることで、デバイス情報の "追加" ページと、(このあと作成に着手する) デバイス情報の "編集" ページとの双方のコンポーネントから、それぞれ "デバイス情報フォーム"コンポーネントを子コンポーネントとして使用することで、コードの共有化が図れます。

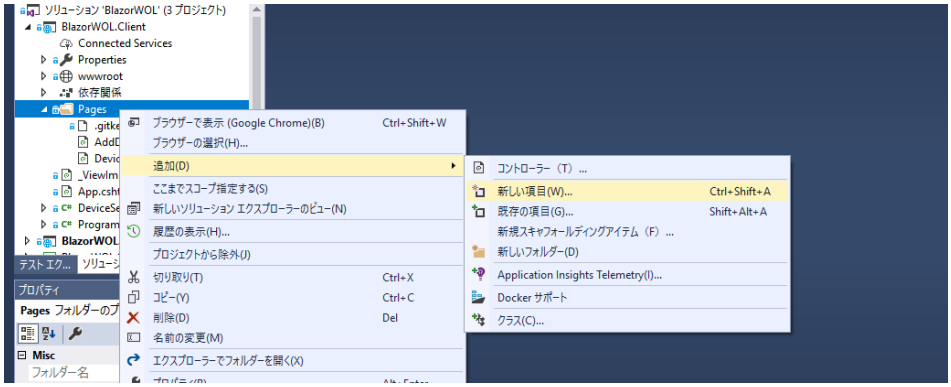
"デバイス情報フォーム" コンポーネントでは、(デバイス情報の追加、または編集の) 親コンポーネントから、フォーム上で取り扱う対象のデバイス情報オブジェクトを受け取る必要があります。
また、"デバイス情報フォーム" コンポーネント内で発生した OK ボタンクリックなどのイベントを親コンポーネントに伝える必要もあります。

この用途には、Blazor コンポーネントに **[Parameter] 属性付きのパブリックプロパティ**を実装することで実現できます。
Blazor コンポーネントの [Parameter] 属性付きパブリックプロパティは、そのコンポーネントのマークアップ時、属性として親コンポーネントの値をバインドすることが可能です。

イベントの伝達も同様で、コールバックハンドラの型のパブリックプロパティを [Parameter] 属性で公開することで実現できます。

手順

1. まずは新しい Blazor コンポーネントファイルを追加します。Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトの Pages フォルダを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。

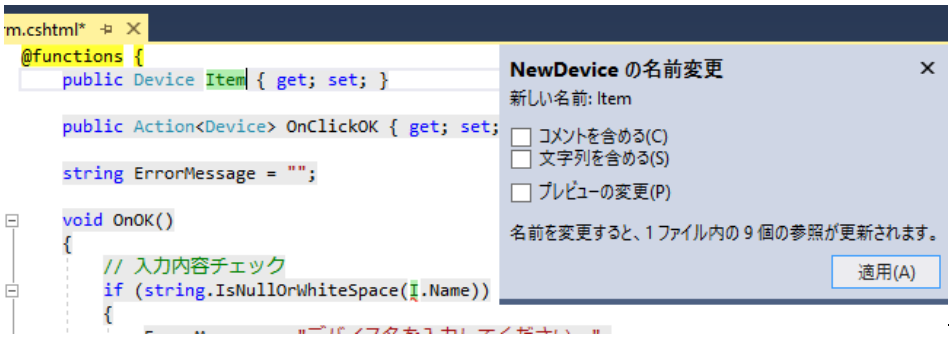


2. 「新しい項目の追加」ダイアログが現れるので、
   - ダイアログ中央のアイテム一覧から「Blazor コンポーネント」をクリックして選択して、
   - [名前(N)] に "**DeviceForm**" と入力してから
   - [追加(A)] ボタンをクリックします。
3. DeviceForm.razor ファイルが Pages フォルダ内に追加され、Visual Studio 内に開かれます。
   DeviceForm.razor ファイル内に、いったん AddDevice.razor の内容すべてをコピーして貼り付けます。
4. ただし DeviceForm.razor はあくまでも子コンポーネントとして使うので、URL ルーティングへの割り当てである「@page ～」ディレクティブの行は削除します。
   また、デバイス情報サービスへのモデル更新や、コード中からのページナビゲーションは親コンポーネントで行うことであり、DeviceForm.razor では関与しません。
   そこで、これらサービスの注入を行う「@inject ～」ディレクティブもすべて削除します。
5. 次に、編集対象のデバイス情報オブジェクトは、親コンポーネントから引き渡されるものとします。
   そのために、メンバーフィールド NewDevice を、[Parameter] 属性付きの public プロパティに実装を書き換えます。また、新規デバイス情報オブジェクトの初期設定は不要なので削除しましょう。

```
@code {
    // "Device NewDevice = new Device()" からの書き換え
    [Parameter]
    public Device NewDevice { get; set; }
```

6. ところで、このプロパティの名前、"**New**Device" は、今後この DeviceForm コンポーネントがデバイス情報の "編集" ページからも使用することを考えると、"New～" から始まる名前はあまりふさわしくありません。
   そこで、より一般的で無個性なプロパティ名 "**Item**" に変更することにします。
   ~~このためには、Visual Studio のリファクタリング機能を用いてプロパティ名を変更するのがよいでしょう (標準のキーボードショートカットだと Ctrl + R, Ctrl + R) 。~~

~~Visual Studio のリファクタリング機能を用いてプロパティ名を変更すれば、同コンポーネント内の HTML パートおよびコードブロック内のすべての必要箇所で、プロパティ名が NewDevice から Item に変更されます。~~



※2020 年 5 月現在、Visual Studio 2019 において、上記、"NewDevice" から "Item" への名前変更リファクタリング機能が動作しません。Visual Studio の不具合の可能性があります。当面はエディタの**通常の文字列置換機能で書き換え**るか、**Visual Studio Code を使って開発し、Ctrl+F2 でプロパティ名を変更**して下さい。

7. 次に、OK ボタンクリックを親コンポーネントに伝達するため、コールバック関数型の public プロパティを、DeviceForm.razor のコードブロック内にさらに追加します。
   プロパティ名は "OnClickOK" としましょう。
   型は、このコンポーネントでのバインド対象のデバイス情報オブジェクトを引数にひとつ取る非同期関数として、`Func<Device, Task>` とします。

```
@code {
    [Parameter]
    Public Device Item { get; set; }

    [Parameter]
    public Func<Device, Task> OnClickOK { get; set; }
```

8. DeviceForm.razor の仕上げとして、OnOK メソッド内末尾にて、入力チェックと正規化が完了したあとの処理を、OnClickOK プロパティにバインドされる親コンポーネントへのコールバック呼び出しに書き換えます。

```
async Task OnOK()
{
    ...
    // MAC アドレスの正規化
    Item.MACAddress = Item.MACAddress.Replace("-", ":").ToUpper();

    await OnClickOK?.Invoke(Item);
}
```

9. 次に AddDevice.razor を、こうして作成した DeviceForm コンポーネントを使うように変更します。
   AddDevice.razor を Visual Studio で開き、HTML パートはいまや DeviceForm コンポーネントに任せますので、下記のとおり DeviceForm コンポーネントのマークアップのみに書き換えます。

```
<DeviceForm Item="new Device()" OnClickOK="OnClickOK"></DeviceForm>
```

   上記のとおり、デバイス情報フォームコンポーネントでバインドするデバイス情報オブジェクトは、新規にその場でインスタンス化したデバイス情報オブジェクトを Item プロパティに渡しています。

10. 続けて、デバイス情報フォームコンポーネントの OnClickOK プロパティにバインドするコールバック関数 (メソッド) を、AddDevice.razor のコードブロック内に実装します。
    メソッド名はバインド先のプロパティと同じく OnClickOK とします。
    いまやほとんどの処理が DeviceForm コンポーネントに移行したので、AddDevice.razor 内のコードブロックはほとんど削除し、下記コードだけ残して OnClickOK メソッドとして実装します。

```
@code {
    async Task OnClickOK(Device newDevice) {
        await DeviceService.AddDeviceAsync(newDevice);
        NavigationManager.NavigateTo("/");
    }
}
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

内部構造は変わりましたが、見た目上の振る舞いは変更前と変わらず、正常にデバイス情報の追加などが操作できることを確認してください。

Step 15. デバイス情報の編集 - ルーティング引数

概要

それではデバイス情報の編集機能の実装に着手していきましょう。
デバイス情報の編集ページの URL は "/edit/{編集対象のデバイス情報の ID}" としましょう。

Blazor コンポーネントで、URL に含まれる引数情報を受け取るには、URL ルーティング定義の「@page ～」ディレクティブにおいて、**URL パターンの記述に、引数部分をブレースで囲って "{identifier}" と記載**します。
すると、その Blazor コンポーネントの identifier という名前の [Parameter] 属性付きプロパティに、この URL パターンの該当する部分が設定される仕掛けとなっています。

ということで、デバイス情報編集ページコンポーネントの URL ルーティング定義は、"/edit/{DeviceId}" とし、同コンポーネントに DeviceId という名前の [Parameter] 属性付きプロパティを設けて、この URL 引数を受け取るようにします。

なお、このように URL 引数を受け取るプロパティの型は、URL パターンの記述において、引数名の後ろにコロン (:) を続けて型名を記述することで int や datetime などの型のプロパティをバインド可能です。
今回は Guid 型を使いますので、"/edit/{DeviceId**:guid**}" とします。

> ※ URL パターンの引数に指定する型名に何が指定できるかについては、下記公式ドキュメントを参照ください。
> https://docs.microsoft.com/aspnet/core/blazor/routing?view=aspnetcore-3.0#route-constraints

まずはこの URL 引数の受け渡しがうまくいくか確認できるところまで進めます。

手順

まずはデバイス情報一覧ページから、編集ページに遷移できるように実装していきます。
BlazorWOL.Client プロジェクトの DevicesComponent.razor を Visual Studio で開きます。
そして HTML パートのデバイス情報 x 1件を表示するマークアップにて、下記のように編集ボタンを追加します。

```
@foreach (var device in Devices)
{
    ...
    <div class="mac-address">
      <span class="caption">MAC アドレス</span>
      <span class="value">@device.MACAddress</span>
    </div>
    <div class="actions">
      <button class="button edit-button" @onclick="() => OnClickEdit(device)">
        編集
      </button>
    </div>
  ...
```

OnClickEdit イベントハンドラはまだ未実装で、このあと実装を進めていきます。
ここでのポイントは、@onclick ディレクティブに指定するハンドラを、(メソッド名を直書きするのではなく) ラムダ式で指定することで、foreach 中のループ変数 device を、イベントハンドラ引数に渡しているところです。
これで、編集ボタンがクリックされた対象のデバイス情報オブジェクトを特定できます。

次に準備として、このデバイス情報一覧ページコンポーネントでは、編集ボタンのクリックによって、デバイス情報編集ページの URL への遷移を行いますから、Blazor 標準備え付けの NavigationManager サービスが必要です。
そこで DevicesComponent.razor に下記のとおり NavigationManager サービスの注入行を書き足します。

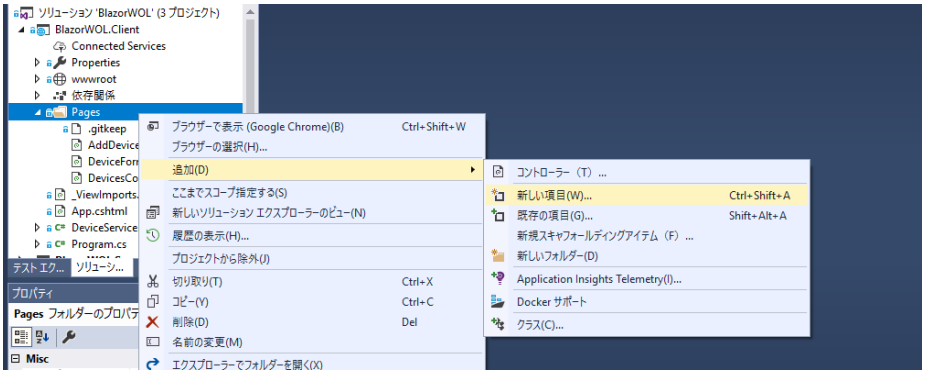
```
@page "/"
@inject DeviceService DeviceService
@inject NavigationManager NavigationManager
```

次にコードブロックにて、OnClickEdit イベントハンドラを実装します。
引数に編集対象のデバイス情報オブジェクトが渡されて呼び出されますから、その編集対象デバイス情報オブジェクトの Id プロパティ値をもとに遷移先の URL を組み立て、DI 機構経由で入手した NavigationManager サービスを使ってページ遷移を実行するように実装します。

```
void OnClickEdit(Device device)
{
    NavigationManager.NavigateTo($"/edit/{device.Id}");
}
```

デバイス情報一覧ページの変更はこれで完了です。

続けて、デバイス情報編集ページ (ただし、まだ実際の編集機能までは盛り込まず、URL 引数の確認ができる程度の実装) を作成します。
Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトの Pages フォルダを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。



「新しい項目の追加」ダイアログが現れるので、
- ダイアログ中央のアイテム一覧から「Blazor コンポーネント」をクリックして選択して、
- [名前(N)] に "**EditDevice**" と入力してから
- [追加(A)] ボタンをクリックします。

EditDevice.razor ファイルが Pages フォルダ内に追加され、Visual Studio 内に開かれます。
EditDevice.razor ファイル内に、概要のところで書いたとおり、URL ルーティング定義として「`@page "/edit/{DeviceId:guid}"`」の行を追加します。

```
@page "/edit/{DeviceId:guid}"
```

そして EditDevice.razor に「@code {}」コードブロックを追加し、上記 page ディレクティブで指定した "{DeviceId:guid}" URL 引数を受け取る、同名の Guid 型の [Parameter] 属性付きプロパティを追加します。

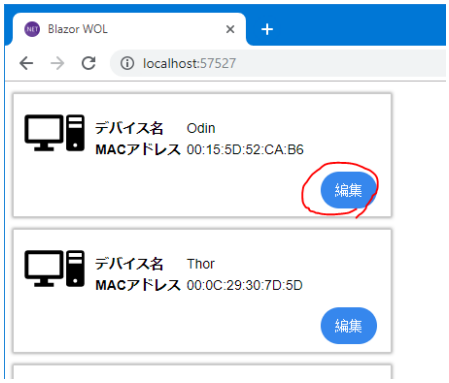
```
@code {
    [Parameter]
    public Guid DeviceId { get; set; }
}
```

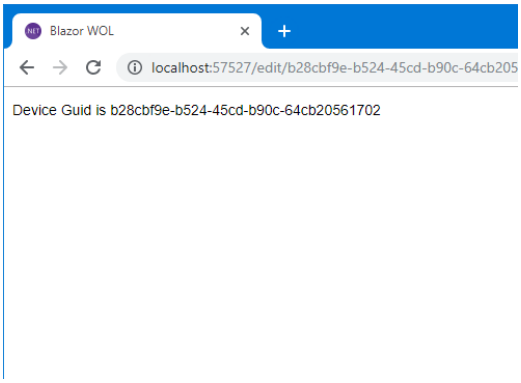
最後に、動作確認の目的で、EditDevice.razor の HTML パートに、URL 引数を受け取った DeviceId プロパティ値を表示するだけのマークアップを記述します。

```
<p>Device Id is @DeviceId</p>
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報一覧にて、個々のデバイス情報に「編集」ボタンが追加されており、これをクリックすると、デバイス情報の Id プロパティが動作確認表示されることを確認してください。





Step 16. デバイス情報編集ページの実装

概要

それではいよいよ、デバイス情報の編集機能を実際に作り込んでいきます。

まずはモデル更新ができるよう、デバイス情報サービスに、

- 指定 Id のデバイス情報の取得、
- 及び、指定 Id のデバイスの更新

を行う機能を追加します。

そのうえで、デバイス情報の編集ページコンポーネントでは、先に作成したデバイス情報フォームコンポーネント (DeviceForm) を利用してユーザーインターフェースを作りこんでいきます。

なお、デバイス情報編集ページコンポーネントではデバイス情報追加のときと異なり、編集対象のデバイス情報の取得が非同期処理となります。
すなわち、編集対象のデバイス情報オブジェクトが取得完了するまでの間の描画抑止の制御が加わってきます。

手順

1. まずはデバイス情報サービス (DeviceService.cs) に、GetDeviceAsync() メソッドと、UpdateDeviceAsync() メソッドとを、下記のように追加しておきます。

```
public async Task<Device> GetDeviceAsync(Guid id)
{
    return await Task.Run(() =>
        Devices.FirstOrDefault(dev => dev.Id == id));
}

public async Task UpdateDeviceAsync(Guid id, Device device)
{
    await Task.Run(() =>
    {
        var updateTarget = Devices.FirstOrDefault(dev => dev.Id == id);
        updateTarget.Name = device.Name;
        updateTarget.MACAddress = device.MACAddress;
    });
}
```

2. 次に、デバイス情報編集ページコンポーネントの変更に取り掛かります。
   EditDevice.razor を Visual Studio で開き、ファイル先頭のほうにデバイス情報サービスの DI 機構経由での注入を記述します。
   また、OK ボタンクリック時の一覧ページへの遷移も必要なので、NavigationManager も DI で注入します。

```
@page "/edit/{DeviceId:guid}"
@inject DeviceService DeviceService
@inject NavigationManager NavigationManager
```

3. 次にコードブロックの編集に移り、以下のメンバーフィールドを追加します。
   - 編集対象のデバイス情報オブジェクト
   - 編集対象のデバイス情報オブジェクトが取得できたかどうかを示す bool 型のフラグ
   とくに後者のフラグは、デバイス情報オブジェクトの取得が、最終的には HTTP 通信経由でのサーバー側からの取得で非同期処理となるため、サーバー側への問い合わせを行っているその間、デバイス情報編集のフォームを表示させないために必要です。

```
@code {
    [Parameter]
    public Guid DeviceId { get; set; }

    bool initialized = false;

    Device Item;
```

4. 続けて、このデバイス情報編集ページコンポーネント EditDeivce の初期化処理を実装します。
   OnInitializedAsync 仮想メソッドをオーバーライドし (async キーワードの追加も忘れずに)、この中でデバイス情報サービスへの編集対象デバイス情報オブジェクト取得を要請します。
   取得できたら、このデバイス情報オブジェクトの複製を作って、編集対象を指すメンバーフィールド (Item) に代入します。
   最後に "初期化完了" のフラグのメンバーフィールド (initialized) を true にします。

```
protected override async Task OnInitializedAsync()
{
    var device = await DeviceService.GetDeviceAsync(DeviceId);
    Item = new Device
    {
        Name = device.Name,
        MACAddress = device.MACAddress
    };
    initialized = true;
}
```

5. 次に、HTML パートはまだ記述していませんが、先に、OK ボタンがクリックされたときの処理をコードブロック内に記述してしまいます。
   コードブロック内に、(DeviceForm コンポーネントの OnClickOK プロパティにバインドする) OnClickOK メソッドを追加します。
   OnClickOK メソッド内では、デバイス情報サービスに対し編集後のデバイス情報で更新要請を行い、これが完了したら URL "/" にページ遷移する処理を記述します。

```
async Task OnClickOK(Device editedDevice)
{
    await DeviceService.UpdateDeviceAsync(DeviceId, editedDevice);
    NavigationManager.NavigateTo("/");
}
```

6. 最後にHTMLを記述しましょう。

   ページ初期化時に編集対象デバイス情報オブジェクトが取得できるまでの間は描画しないよう initialized メンバーフィールドに基づく @if ブロックを形成します。

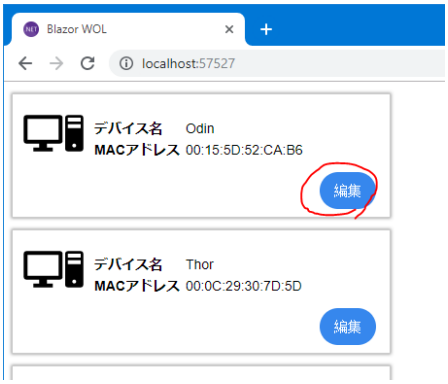
   さらに指定されたID値で該当するデバイス情報が見つからなかった場合に備えた @if ブロックを重ねます。

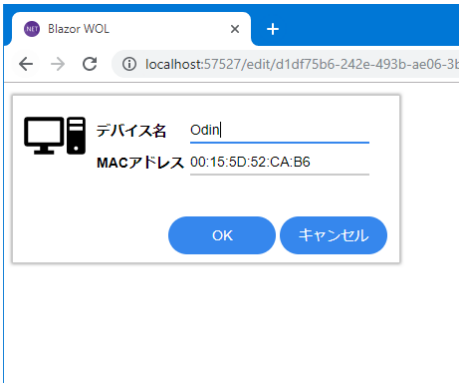
   最後に、編集対象のデバイス情報オブジェクトを引き渡しつつ、OKボタンがクリックされたときのイベントハンドラも指定して、デバイス情報入力フォームコンポーネント DeviceFormをマークアップします。

```
@if (initialized == true)
{
    @if (Item != null)
    {
        <DeviceForm Item="Item" OnClickOK="OnClickOK"></DeviceForm>
    }
    else
    {
        <p class="error-message">デバイス情報が見つかりません。</p>
    }
}
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報一覧から「編集」ボタンをクリックすると、そのデバイス情報が編集できるページに遷移すること、かつ、実際にデバイス情報を変更して OK をクリックすると、変更が一覧に反映されることを確認してください。





Step 17. タイトルヘッダの追加 - レイアウト

概要

ここまでで、一覧・追加・編集の 3 ページを作り込むことができました。

ここで、いずれのページでも共通の、アプリケーション名を表示するタイトルヘッダ部分を追加することにします。

Blazor では、このような "どのページでも共通のレイアウト" を単一コードで実現する仕組みが備わっています。

そのためには、まず、"共通レイアウト" の Blazor コンポーネントを実装します。
このコンポーネントはちょっとだけ特別で、@inherits ディレクティブを使用して **LayoutComponentBase 抽象クラスから派生**する必要があります。

あとは、この "共通レイアウト" の HTML パートを記述し、LayoutComponentBase 抽象クラスで提供されるプロパティ Body を任意の箇所でバインドすれば、その **Body プロパティをバインドした部分に、各ページコンポーネントのコンテンツが差し込まれる**仕組みです。

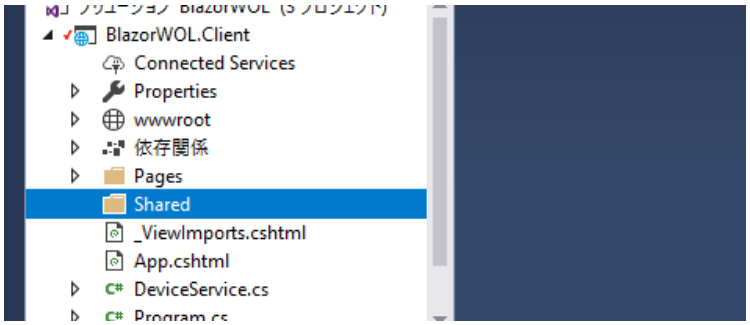
この共通レイアウトを使うには、App.razor 中に記述した Router コンポーネントにて、**共通レイアウトとして使う Blazor コンポーネントのクラスを指定**します。
このようにすることで、各ページコンポーネントの描画時には、この共通レイアウトコンポーネントの Body プロパティをバインドした箇所に、そのページコンポーネントが差し込まれて描画されるようになります。

手順

1. まずは共通レイアウトを実装する Blazor コンポーネントを追加します。
   コンポーネント名は MainLayout としましょう。
   慣例的に、いずれのページコンポーネントからも使用されるような類いのコンポーネントは、"Shared" サブフォルダに保存します。
   そこで共通レイアウトコンポーネントも Shared フォルダ内に配置することにします。

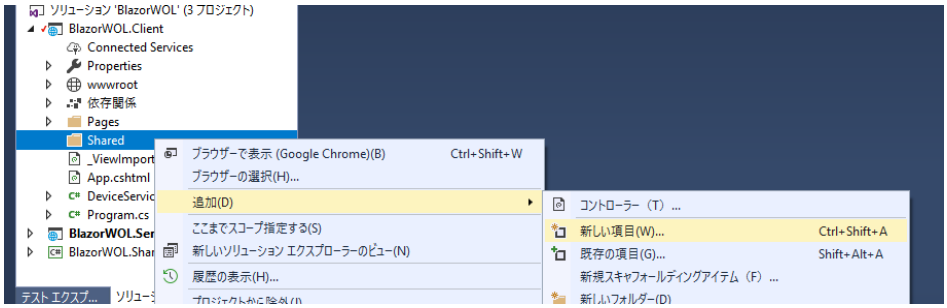
   Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトを右クリックし、メニューから [追加(D)]-[新しいフォルダー(D)...] をクリックします。

2. プロジェクトに新しいフォルダが追加されるので、フォルダ名を "Shared" と入力して Enter を押して確定します。



3. サブフォルダができましたので、次に、共通レイアウトコンポーネント "MainLayout" のファイルを追加していきます。

   Visual Studio のソリューションエクスプローラ上で BlazorWOL.Client プロジェクトの **Shared フォルダ**を右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。



4. 「新しい項目の追加」ダイアログが現れるので、

   - ダイアログ中央のアイテム一覧から「Blazor コンポーネント」をクリックして選択して、
   - [名前(N)] に "**MainLayout** " と入力してから
   - [追加(A)] ボタンをクリックします。

5. MainLayout.razor ファイルが Shared フォルダ内に追加され、Visual Studio 内に開かれます。

   MainLayout.razor ファイル内の既存のコードはいったんすべて削除し、以下のとおり実装します。

   - LayoutComponentBase クラスから派生することを示す @inherits ディレクティブを行頭に追加
   - HTML パートで、Body プロパティのバインドを含む、タイトルヘッダを表す HTML のマークアップ

```
@inherits LayoutComponentBase
<header>
    <h1>Blazor WOL</h1>
</header>
@Body
```

6. こうして作成した共通レイアウトコンポーネント "MainLayout" の型を、このあと参照することになります。

   しかしその際に長々と名前空間を含めて記述しなくてはならないのは、可読性が悪くなります。

   そこで名前空間を省けるよう、いずれのコンポーネントにも差し込まれる _Imports.razor 内にて、共通レイ

アウトコンポーネント "MainLayout" の名前空間を、@using ディレクティブで開いておきましょう。

_Imports.razor を Visual Studio のエディタ内に開き、下記のように最終行に追記します。

```
...前半変更なし...
@using BlazorWOL.Shared
@using BlazorWOL.Client.Pages
@using BlazorWOL.Client.Shared
```

7. 以上で共通レイアウトコンポーネントは実装できました。

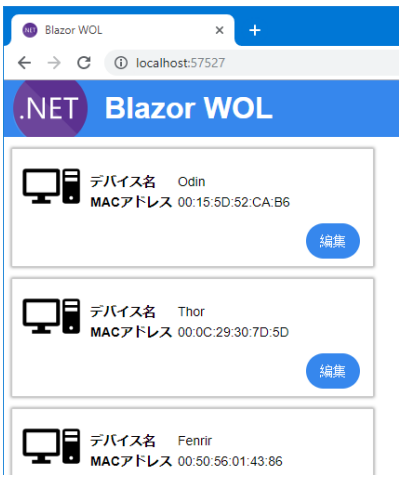
   続けて各ページコンポーネントでこの MainLayout をレイアウトコンポーネントとして使用するよう、App.razor 中の Router コンポーネントに対して指定します。

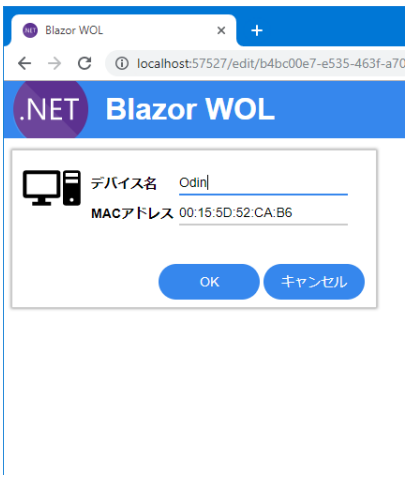
   Visual Studio で App.razor を開き、RouteView コンポーネントの **"DefaultLayout" パラメータ**に、共通レイアウトとして使うコンポーネントの型 (今回は MainLayout クラス) を指定するよう、追記します。

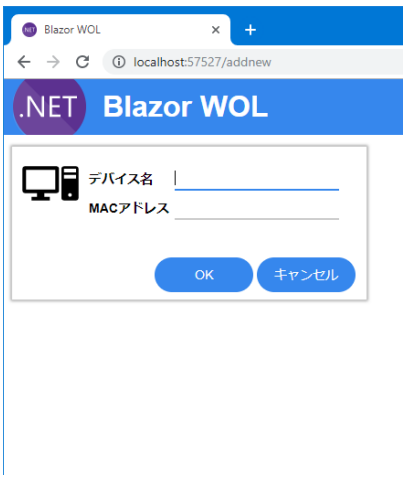
```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報一覧、追加、編集のいずれのページでも、タイトルヘッダが表示されることを確認してください。





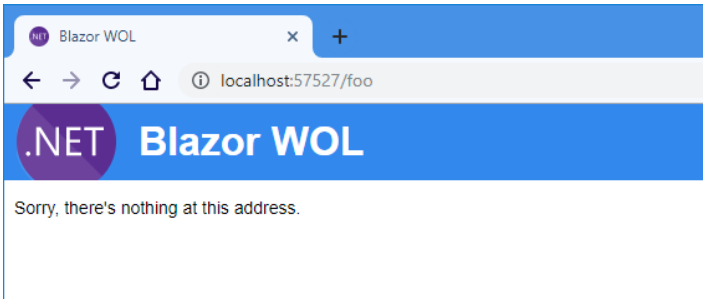


なお、ここまでの実装だけですと、該当するルート定義が無い場合のエラーメッセージ表示には、この共通レイアウトコンポーネントは適用されません。
それで構わなければ以上の実装でよいのですが、もしも、該当するルート定義がない場合のメッセージ表示にも共通レイアウトコンポーネントを適用したい場合は、App.razor 中の Router コンポーネントに対し、さらに以下の要領で追加の記述を行なうことで達成できます。

Visual Studio で App.razor を開き、該当するルート定義がない場合に描画される NotFound 描画要素について、素の HTML マークアップではなく、(Blazor に備え付けの) **LayoutView コンポーネント**でくるむようにします。
そしてこの LayoutView コンポーネントの **"Layout" パラメータ**に、共通レイアウトとして使うコンポーネントの型、MainLayout クラスを指定します。

```
<Router AppAssembly="typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="typeof(MainLayout)">
          <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

以上の実装で、LayoutView コンポーネントは、MainLayout クラスをレイアウトコンポーネントして使用して LayoutView コンポーネント自身の HTML マークアップを描画することになります。
結果、該当するルート定義がない場合のメッセージ表示にも、共通レイアウトが適用されます (下図)。



Step 18. サーバー側実装の開始 - ASP.NET Core Web API の実装

ここまでで、概ねユーザーインターフェースが形になりましたので、いよいよサーバー側を実装します。

なお、本自習書は Blazor の学習がねらいであり、ASP.NET Core については経験者を想定しています。
そのため、サーバー側実装の詳細手順は割愛いたします。

サーバー側 (BlazorWOL.Server プロジェクト) での実装作業の概要は以下のとおりです。

- デバイス情報を JSON 形式でテキストファイルに保存・復元する、DeviceStorage クラスを実装。デバイス情報の一覧取得・Id を指定しての取得・追加・更新・削除のひととおりの機能を実装。
- この DeviceStorage クラスを、Startup クラスの ConfigureServices メソッド内にて、シングルトンサービスとして ASP.NET Core の DI 機構に登録。
- DeviceStorage オブジェクトを DI 機構経由でコンストラクタ引数に受け取る、DevicesController Web API コントローラクラスを実装。DeviceStorage クラスの機能を、下記 URL で公開。
    - HTTP GET /api/devices ... すべてのデバイス情報を JSON 形式で返却
    - HTTP GET /api/devices/*{id}* ... 指定の Id のデバイス情報を JSON 形式で返却
    - HTTP POST /api/devices ... 要求本文の JSON で指定したデバイス情報を追加
    - HTTP PUT /api/devices/*{id}* ... 指定の Id のデバイス情報を、要求本文の JSON の内容で更新
    - HTTP DELETE /api/devices/*{id}* ... 指定の Id のデバイス情報を削除

サーバー側実装は、本自習書に同梱のソースコードから取得ください。

なお、サーバー側実装でも BlazorWOL.Shared プロジェクトを参照しているので、**クライアント側実装と同じ Device クラスを使用して実装できる**ところが注目ポイントです。

Step 19. サーバー側 Web API の呼び出し - HttpClient の使用

概要

ようやくサーバー側実装の Web API もできあがりましたので、いよいよ、クライアント側のデバイス情報サービス (DeviceService) クラスから Web API の呼び出しを行っていきます。

Blazor 上でのサーバー側との HTTP 通信には、他の.NET プログラミングでもおなじみの System.Net.Http.HttpClient クラスを使います。

ただし、**自分で HttpClient オブジェクトをインスタンス化してはいけません**。
Blazor プログラムはあくまでもブラウザ上で動作しているので、任意の TCP 通信はできませんから、そのような TCP 通信を行おうとする HttpClient は動作しません。

その代わり、**Blazor ランタイムによって DI 機構経由で提供される HttpClient オブジェクトを使ってください**。
この HttpClient オブジェクトは、ブラウザの XmlHttpRequest ないしは fetch API を使って HTTP 通信するように、内部のメッセージハンドラが差し替え済みとなっている特別なインスタンスです。
Blazor の DI 機構経由で入手した HttpClient オブジェクトであれば、普通にブラウザ上からサーバー側の Web API と通信できます。

ところで、Blazor コンポーネント内にて DI 機構経由でサービスオブジェクトを入手するには @inject ディレクティブを使いました。
しかし、今回、HttpClient サービスオブジェクトを必要としているのは、Blazor コンポーネントではなく、デバイ

ス情報サービス DeviceService クラスです。
これは .razor とは異なり、単純な C#ソースコード (.cs) で書かれた普通のクラスですから、@inject のようなディレクティブは使えません。

このようなサービスクラスで DI 機構経由でのオブジェクト入手が必要な場合は、コンストラクタの引数にて必要なオブジェクトを入手するように実装します。
そうしておくことで、DI 機構がそのクラスをインスタンス化するときに、コンストラクタ引数に応じて、その DI 機構で管轄しているオブジェクトを渡してくれます。

手順

まず、HttpClient、およびその拡張メソッドをすぐに使えるようにするため、BlazorWOL.Client プロジェクトの DeviceService.cs を Visual Studio で開き、DeviceService.cs 先頭に以下の 2 つの名前空間の使用を追加します。

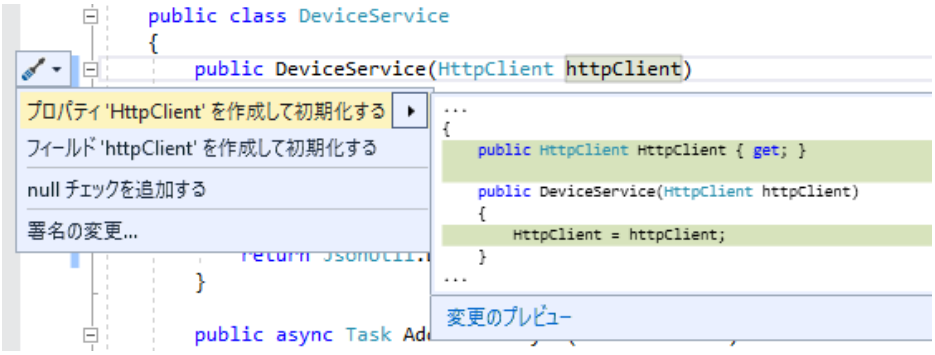
```
using System.Net.Http;
using System.Net.Http.Json;
```

続けて、ダミーデータ実装の Devices プロパティは削除しておきます。

次に、HttpClient オブジェクトを DI 機構から受け取れるようにするために、コンストラクタを追加します。
コンストラクタの引数にて、HttpClient オブジェクトを受け取るようにします。

```
public class DeviceService
{
    public DeviceService(HttpClient httpClient)
    {
    }
```

コンストラクタ引数に受け取った HttpClient オブジェクトを、プロパティに保存します。
なお、Visual Studio のクイックアクションを使うと、コンストラクタの引数から、これを保存するプロパティのコーディングを自動でおこなうことができます (下図)。



続けて、デバイス情報サービスの実装を、ダミーデータ実装の Devices プロパティを読み書きしていたものから、プロパティに保存しておいた HttpClient を使ってのサーバー側 Web API 呼び出しに書き換えます。

なお、このタイミングで、デバイス情報サービスにはまだ備えていなかった、デバイス情報の削除のメソッド (DeleteDeviceAsync) も追加実装してしまいましょう。

```
public async Task<IEnumerable<Device>> GetDevicesAsync()
{
    return await HttpClient.GetFromJsonAsync<Device[]>("api/devices");
}

public async Task AddDeviceAsync(Device device)
{
    await HttpClient.PostAsJsonAsync("api/devices", device);
}

public async Task<Device> GetDeviceAsync(Guid id)
{
    try
    {
        return await HttpClient.GetFromJsonAsync<Device>($"api/devices/{id}");
    }
    catch (HttpRequestException e) when (e.Message == "404 (Not Found)")
    {
        return null;
    }
}

public async Task UpdateDeviceAsync(Guid id, Device device)
{
    await HttpClient.PutAsJsonAsync($"api/devices/{id}", device);
}

public async Task DeleteDeviceAsync(Guid id)
{
    await HttpClient.DeleteAsync($"api/devices/{id}");
}
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報の永続化先がサーバー側となったので、初回はデバイス情報一覧が空となっています。
以降、デバイス情報の追加や編集が機能していること、また、いまやデバイス情報はサーバー側でファイルに永続化されるようになったので、ブラウザで再読み込みを繰り返しても、最後の保存結果が復元されることを確認してください。

Step 20. デバイス情報の削除機能を実装 - JavaScript 相互運用

概要

もう少しアプリケーションを仕上げていきましょう。
実装を先延ばしにしていた、デバイス情報の削除機能の実装に着手します。

モデル更新的には、ここまでで既に実装してきた Web API (HTTP DELETE /api/devices/{id}) を呼び出すだけです。

いっぽう、ユーザーインターフェースですが、「削除」ボタンを設けるのは当然として、削除ボタンを押したとたんに即時に削除処理が行われるのは好ましくありません。
そこで、とりあえず、ブラウザの confirm JavaScript 関数を呼び出して、本当に削除してよいかどうかの確認を取ることとします。

Blazor プログラムと JavaScript 間での関数呼び出しの相互運用機能が、Blazor には備わっています。

JavaScript の関数を Blazor 側から呼び出すのは簡単です。

まずは、JavaScript 相互運用を担う IJSRuntime インターフェースを備えたサービスを DI 機構経由で入手します。
そうして手に入れた JavaScript 相互運用サービスの InvokeAsync<T>() メソッドに、呼び出す JavaScript 関数名と引数を渡すだけです。

なお、JavaScript 側から Blazor プログラム内のコードを呼び出すこともできます。
本自習書では割愛しますので、詳細は Blazor 公式ドキュメントサイトの下記コンテンツなどを参照ください。

https://docs.microsoft.com/en-us/aspnet/core/blazor/javascript-interop?view=aspnetcore-3.0

手順

1. まずはデバイス情報入力フォーム DeviceForm に、削除ボタンを設けましょう。
   BlazorWOL.Client プロジェクトの DeviceForm.razor を Visual Studio で開きます。
   そして、削除ボタンが押された時の confirm 呼び出しを行うために Blazor の JavaScript 相互運用サービスを参照しますので、そのために IJSRuntime インターフェースを備えたサービスを DI 機構で注入してもらう @inject ディレクティブをファイル先頭のほうで記述しておきます。
   JavaScript 相互運用サービスのインスタンスを受け取るフィールド変数名は "JSRuntime" とでもしておきましょう。

```
@using System.Text.RegularExpressions
@inject IJSRuntime JSRuntime
```

2. 次に DeviceForm.razor のコードブロック内にて、削除ボタンが押された時のコールバック非同期関数をバインドする public プロパティ OnClickDelete を追加します。
   バインド用なので、Parameter 属性を付与するのを忘れないようにします。

```
[Parameter]
public Func<Task> OnClickDelete { get; set; }
```

3. そして、まだ HTML は未実装ですが、削除ボタンが押された時のイベントハンドラ OnDelete メソッドをコードブロック内に書き足します。
   OnDelete メソッドでは、Blazor の JavaScript 相互運用機能を介して、JavaScript 関数 "confirm" を呼び出し、その戻り値に応じて、削除ボタンが押された時のコールバック非同期関数 (親コンポーネントからバインドされる OnClickDelete プロパティ) を実行します。

```
async Task OnDelete()
{
    var yes = await JSRuntime.InvokeAsync<bool>("confirm","削除してもよろしいですか?");
    if (yes)
    {
        await OnClickDelete?.Invoke();
    }
}
```

4. 残りは HTML マークアップです。
   DeviceForm.razor の HTML パートにて、「削除」ボタンの button 要素を実装します。
   このとき、デバイス情報追加ページコンポーネント (AddDevice.razor) から使われる場合は、削除ボタンは非表示 (デバイス情報編集ページでのみ削除ボタンを表示する) にしましょう。
   そこで、OnClickDelete コールバック関数プロパティが非 null である場合にのみ、削除ボタンを表示するよう、@if ブロックを形成します。

```
<div class="actions">
    @if (OnClickDelete != null)
    {
        <button class="button delete-button" @onclick="OnDelete">削除</button>
    }
    <button class="button">OK</button>
    <a class="button" href="/">キャンセル</a>
```

5. あとは、デバイス情報編集ページコンポーネント EditDevice にて、削除ボタンが押された時の振る舞いを定義して、DeviceForm の OnClickDelete プロパティにバインドしましょう。
   BlazorWOL.Client プロジェクトの EditDevice.razor を Visual Studio で開き、コードブロック内に、デバイス情報サービスの削除処理を呼び出して処理完了したら URL "/" に遷移するメソッド、OnClickDelete メソッドを追加します。

```
async Task OnClickDelete()
{
    await DeviceService.DeleteDeviceAsync(DeviceId);
    NavigationManager.NavigateTo("/");
}
```

6. そしてこの OnClickDelete メソッドを、デバイス情報入力フォームのマークアップにて、OnClickDelete プロパティにバインドします。

```
<DeviceForm Item="Item" OnClickOK="OnClickOK" OnClickDelete="OnClickDelete"><
```

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行してみてください。

デバイス情報の編集ページに削除ボタンが現れ、これをクリックすると確認メッセージが表示され、さらにこの確認メッセージで OK をクリックすると、そのデバイス情報が削除されて一覧から消えることを確認してください。

> ※うまくいかない場合は、ソリューションをリビルドしてやり直してみてください。

Step 21. 仕上げ - 電源 ON ボタンの追加

これで BlazorWOL アプリケーションの実装はほぼ完成です。

ここまでで習得した内容に基づき、「電源 ON」ボタンを一覧ページに実装すれば完成です。

- サーバー側実装で Web API として HTTP POST /api/devices/{id}/wakeup エンドポイントを追加、このエンドポイントで WOL マジックパケット送信を実装

- クライアント側実装で、DeviceService クラスに WakeupAsync(id) メソッドを追加、上記 Web API エンドポイント呼び出しを実装
- クライアント側実装で、DevicesComponent コンポーネントに「電源 ON」の button 要素をマークアップ、このボタンの click イベントハンドラで DeviceService クラスの WakeupAsync メソッド呼び出し、さらに処理完了メッセージを JavaScript 相互運用機能を介して alert JavaScript 関数を使用して表示

なお、本自習書では、Wakeup On Lan マジックパケットの送信を行う実装として、WakeOnLAN NuGet パッケージを使用しました。

https://www.nuget.org/packages/WakeOnLAN/

以上を実装して、下図のとおり機能することを確認してください。

> ※この Blazor アプリをホストしている Windows OS 上に複数のネットワークカードがあると、期待したネットワークカード経由でマジックパケットが送信されない場合があります。これは今回採用した WakeOnLAN NuGet パッケージの実装上の制約で、いちばん優先順位の高いネットワークカードにのみマジックパケット送信することによるものです。
> ネットワークカードごとのメトリックを手動で設定して、マジックパケットを送信したいネットワークカードの優先順位を上げることで回避可能ですが、場合によってはご自身でマジックパケット送信処理をいちから実装されてもよいかと思います。

以上で Blazor プログラム "Blazor WOL" の実装を通しての、Blazor アプリケーションプログラミング自習書は完了です。

おつかれさまでした。

次のステップへ

本自習書の内容は入門レベルに留まっており、本自習書で触れていないテーマは多々あります。
状態管理はどうするのか、ダイアログのようなより高度な UI はどう実装するのか、ユーザーの認証と認可はどうすればよいのか、PWA への対応方法は...などなどです。

これらのさらなるテーマを学ぶためのひとつの方法として、「Blazor - アプリケーション開発ワークショップ」に取り組んでみる、という方法が考えられます。
ありがたいことに、日本語に翻訳されている fork 版が下記からアクセス可能です。
参考になれば幸いです。

https://github.com/kenakamu/blazor-workshop/tree/ja-jp

あとがき

本自習書に沿って実際に Blazor アプリケーションプログラミングをひととおりなぞることで、

- Blazor が提供・実現する実装形態のシンプルさ
- ASP.NET Core MVC 開発経験者に対する追加の学習コストの小ささ
- Visual Studio IDE や Visual Studio Code による開発支援の実際

などを身をもって体感いただき、Blazor が描く開発生産性向上の可能性を評価いただくことができれば、Blazor の 1 ファンとして冥利に尽きます。

本自習書にお付き合いいただきありがとうございました。

2020 年 5 月
坂本 純一

追補

ライセンス

本自習書、及び、ソースコードは、The Unlicense として提供します。

```
This is free and unencumbered software released into the public domain.

Anyone is free to copy, modify, publish, use, compile, sell, or
distribute this software, either in source code form or as a compiled
binary, for any purpose, commercial or non-commercial, and by any
means.

In jurisdictions that recognize copyright laws, the author or authors
of this software dedicate any and all copyright interest in the
software to the public domain. We make this dedication for the benefit
of the public at large and to the detriment of our heirs and
successors. We intend this dedication to be an overt act of
relinquishment in perpetuity of all present and future rights to this
software under copyright law.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.

For more information, please refer to <http://unlicense.org>
```

商用・非商用問わず、クレジット表示も不要で、本自習書及びソースコードを再利用・改変・再配布が可能です。

関連リソース

- Blazor 公式 GitHub リポジトリ
  - https://github.com/aspnet/AspNetCore/tree/master/src/Components
- Blazor 公式サイト - https://blazor.net/
  - "Get started with Blazor" - https://blazor.net/docs/get-started.html
- Blazor 関連リンク集 (英語) "Awesome Blazor"
  - https://github.com/AdrienTorris/awesome-blazor#awesome-blazor-