# **Blazor WebAssembly (Client-side Blazor) 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 4

主な対象者 4

Blazor のバージョンとエディション 4

開発環境 5

この自習書で作成するWebアプリケーション 5

Step 1. ボイラープレートのビルド 6

補足 - プロジェクトの構造 7

BlazorWOL.Server 7

BlazorWOL.Client 7

BlazorWOL.Shared 7

補足 - Blazorアプリケーションが立ち上がるまでの流れ 8

index.html 8

Program.cs 8

App.razor 8

Step 2. CSSスタイルシートを実装 9

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 9

Step 4. モデルクラスの追加 10

概要 10

手順 11

Step 5. デバイス一覧ページの実装 - コンポーネントの追加 12

概要 12

手順 12

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 13

概要 13

手順 14

Step 7. デバイス一覧ページ - リスト化 (繰り返し) 15

概要 15

手順 15

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 17

概要 17

手順 17

Step 9. 非同期処理化 20

概要 20

手順 20

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 21

概要 21

手順 21

Step 11. 入力内容のチェックと正規化 24

概要 24

手順 24

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 29

補足 - Visual Studio 2019上での Blazor WebAssembly プログラムのデバッグ 30

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 31

概要 31

手順 32

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 35

概要 35

手順 35

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 36

概要 36

手順 37

Step 15. デバイス情報の編集 - ルーティング引数 39

概要 39

手順 40

Step 16. デバイス情報編集ページの実装 42

概要 42

手順 43

Step 17. タイトルヘッダの追加 - レイアウト 45

概要 45

手順 46

Step 18. サーバー側実装の開始 - ASP.NET Core Web API の実装 49

Step 19. サーバー側 Web APIの呼び出し - HttpClientの使用 50

概要 50

手順 51

Step 20. デバイス情報の削除機能を実装 - JavaScript相互運用 53

概要 53

手順 53

Step 21. 仕上げ - 電源ONボタンの追加 55

次のステップへ 57

あとがき 57

追補 58

ライセンス 58

関連リソース 58

## 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WebAssembly版 (client-side 版) を、1ステップずつ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.5.0.0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 **WebAssembly版** ("client-side" 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版** ("server-side" 版)

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

なお、WebAssembly版 ("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 (WebAssembly) 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 5.0 SDK (5.0.100かそれ以降)**<https://dotnet.microsoft.com/download/dotnet-core/5.0>
* **Visual Studio 2019 - 16.8.1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 5.0 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.1.23.6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 16.8.1 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 5.0 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、<div id="app"> というタグが記述されています。

この <div id="app"> タグが、Blazor が生成するコンテンツの受け入れ場所 (プレースホルダ) になります。

<div id="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 <div id="app"> タグ内が、Blazorコンポーネントが生成するDOMコンテンツに書き換えられる仕組みです。

### 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開始地点 (エントリポイント) 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 <div id="app"> 要素内に、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。

※ Blazorでは**「CSSアイソレーション (CSSの分離)」**という仕組みでスタイルシートを作成・適用できます。  
これは Blazor コンポーネント (拡張子 .razor のファイル。詳細は後程) と同じファイル名に、拡張子 .css を追加したスタイルシートを用意すると、そのスタイルシートで定義されたスタイルは**対応するコンポーネントにだけ適用される**、という便利な仕組みです。本手順書では「CSSアイソレーション」については割愛しますが、大変重要・便利な仕組みですので、ぜひ、Blazor 公式ドキュメントサイト (下記) を参照ください。  
<https://docs.microsoft.com/ja-jp/aspnet/core/blazor/components/css-isolation>

## 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。

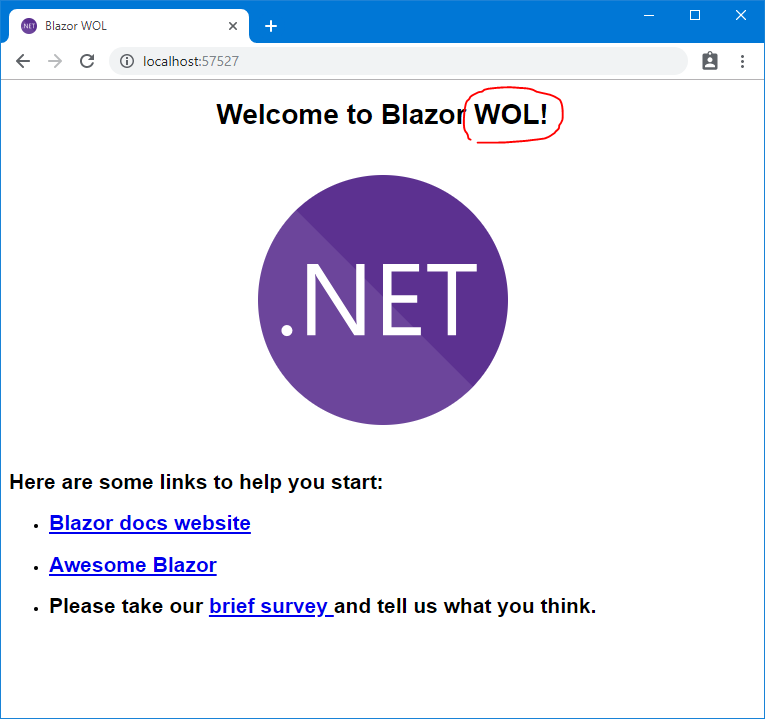
App.razorにはHTMLタグの記述と、C#のコードブロックが含まれています。

.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。  
   

|  |  |  |
| --- | --- | --- |
|  |  | 「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 |

1. 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。

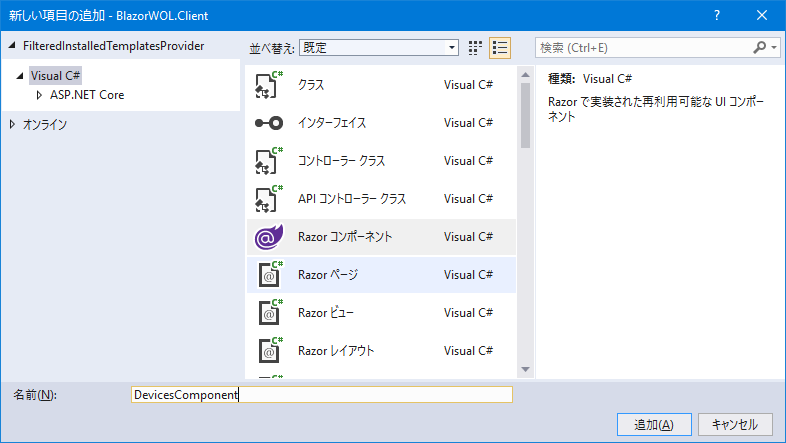
まずはDeviceクラスを表示するBlazorコンポーネント "DevicesComponent.razor" を新規作成します。  
慣例的に、Blazorアプリケーションプロジェクトにおいて、(App.razorを除く) コンポーネント = .razorファイルはPagesフォルダに配置します。

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。



1. 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

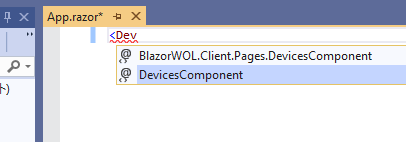
@using BlazorWOL.Shared

@using BlazorWOL.Client.Pages

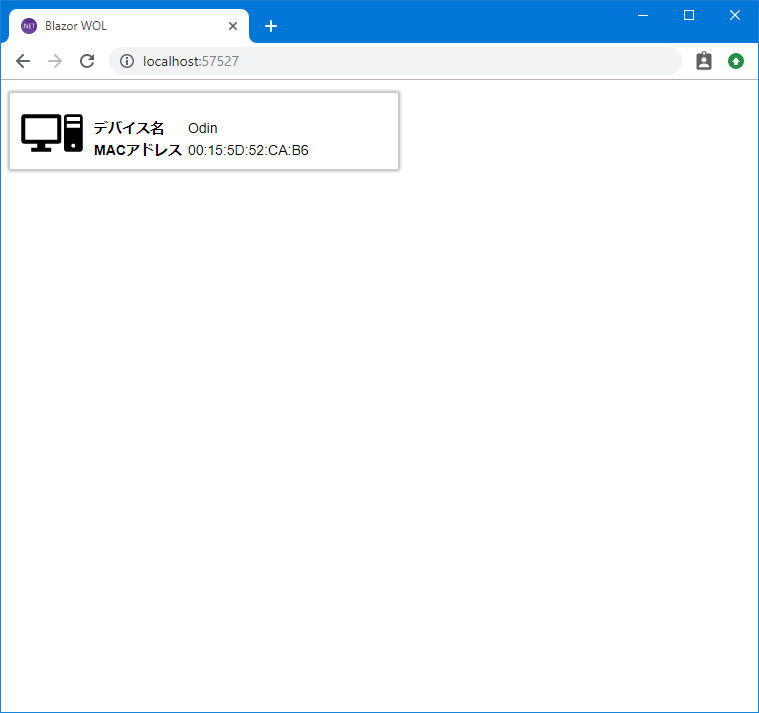
次はApp.razor です。  
App.razorをVisual Studioで開き、既存の内容をすべて削除します。  
そして、下記のように DevicesComponent コンポーネントのクラス名のタグを追記します。

<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"}  
};

1. HTML全体を、@foreach (var device in Devices){ ～ } ブロックで囲みます。
2. 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。
3. 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>

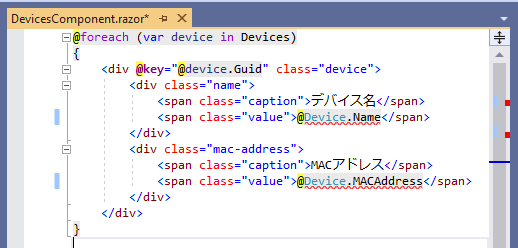
<span class="value">@device.MACAddress</span>

</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;

}

}

}

1. 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. AddScoped(sp => new HttpClient { BaseAddress = ...});」行の直後に「builder.Services.AddScoped<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.AddScoped(sp => new HttpClient { BaseAddress = ... });

builder.Services.AddScoped<DeviceService>();

...

1. 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。
2. (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。
3. 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>> GetDevices**Async**()

{

return **await** **Task.FromResult**(Devices);

}

1. 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 OnInitialized**Async**()

{

Devices = await DeviceService.GetDevices**Async**();

}

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));

}

1. 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();

1. 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();

}

1. 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>

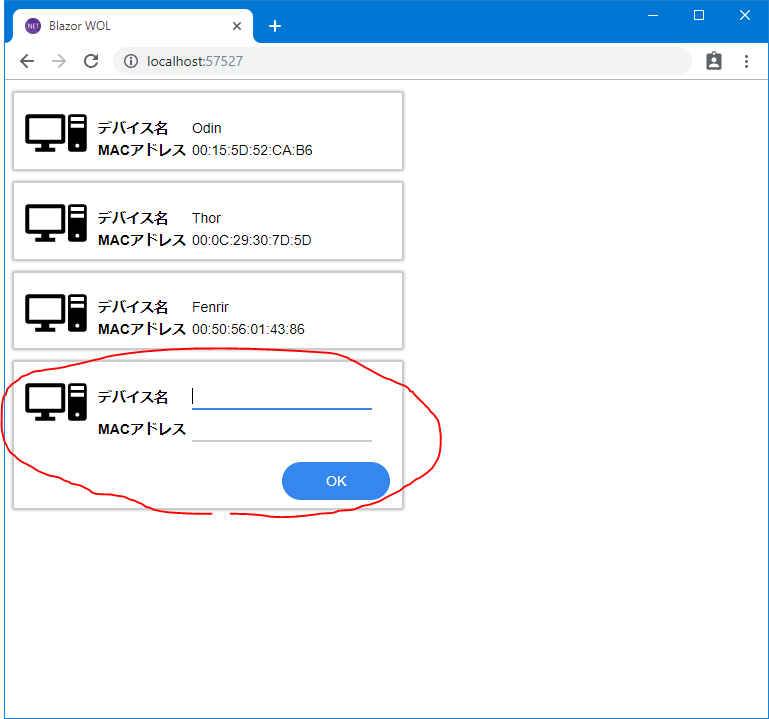
<div class="actions">

<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">

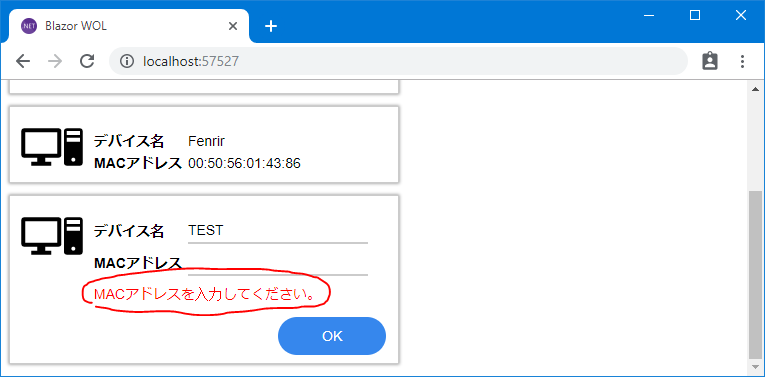
<**ValidationSummary**></**ValidationSummary**>

</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/ja-jp/aspnet/core/blazor/forms-validation>

### 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

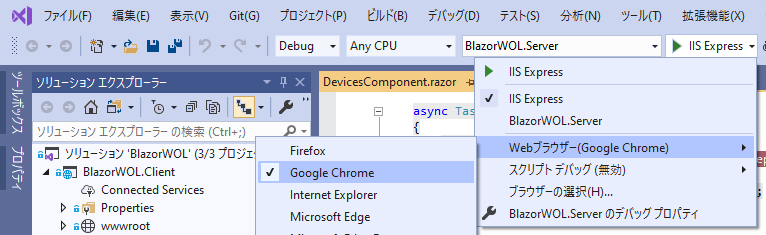
さてここで、もう少し脱線して、Visual Studio 2019上での Blazor WebAssembly プログラムのデバッグについて触れておきます。

2020年4月から、Visual Studio 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 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。

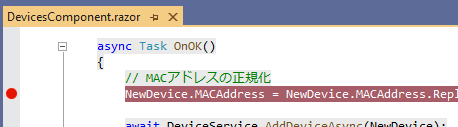
* Microsoft Edge (Chromium版)
* Google Chrome

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)。



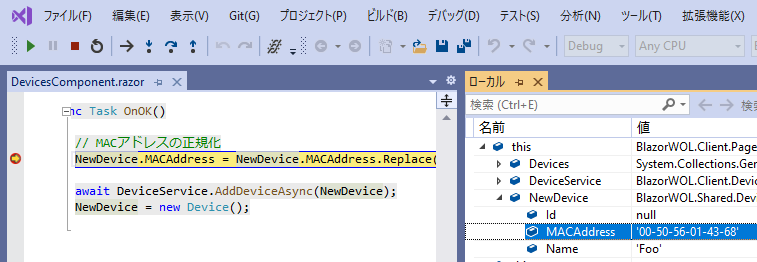
上記いずれかのWebブラウザを起動するようにVisual Studio上で設定済みでしたら、試しに、このステップで実装した入力内容正規化の部分で試してみましょう。

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Blazor WebAssemblyに対するデバッグ機能はまだ制約も少なくありません。  
具体的には、デバッガがアタッチされる前のOnInitializedでブレークできない、などの例があります。

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。

引き続き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。

1. 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。
2. 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。
3. 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)。
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「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。
5. 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
6. 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
7. 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
8. 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。

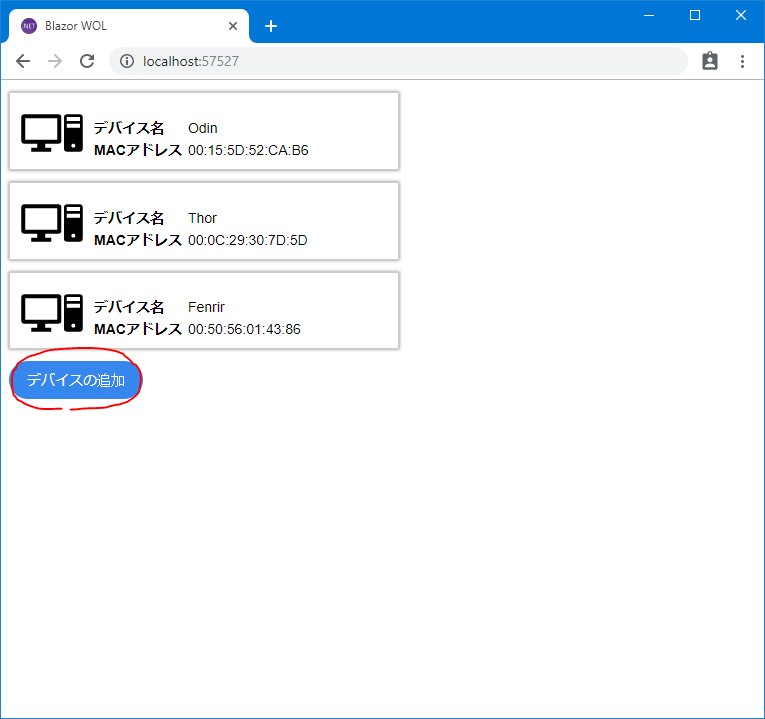
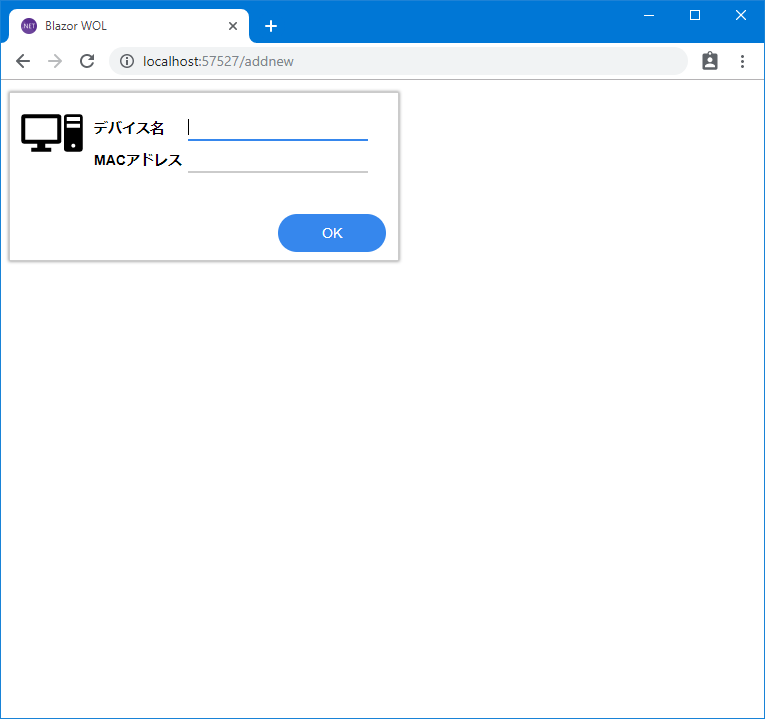
<div>

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

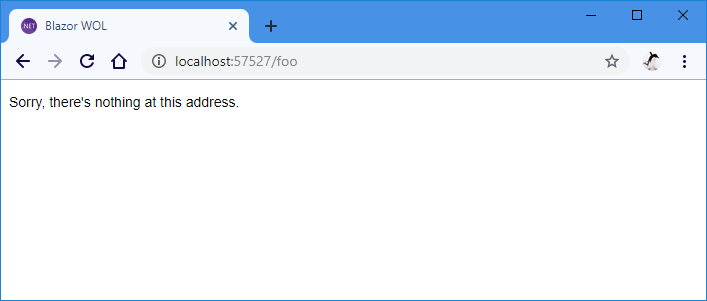
</div>

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

1. 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("/");**

}

1. 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; }

1. 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11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。

1. 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; }

1. 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);

}

1. 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。

1. 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/ja-jp/aspnet/core/blazor/fundamentals/routing#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; }

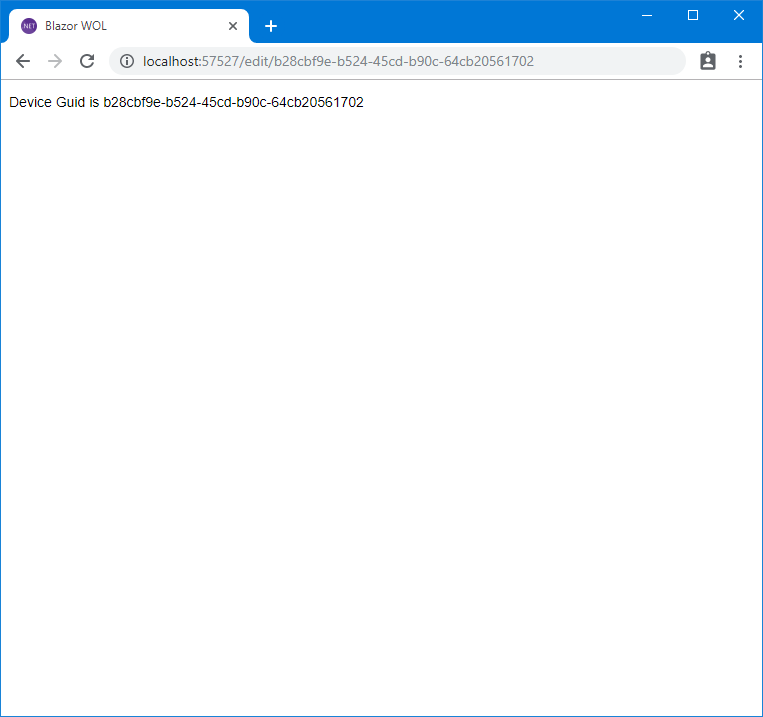
}

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;

});

}

1. 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

1. 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;

1. 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;

}

1. 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("/");

}

1. 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

{

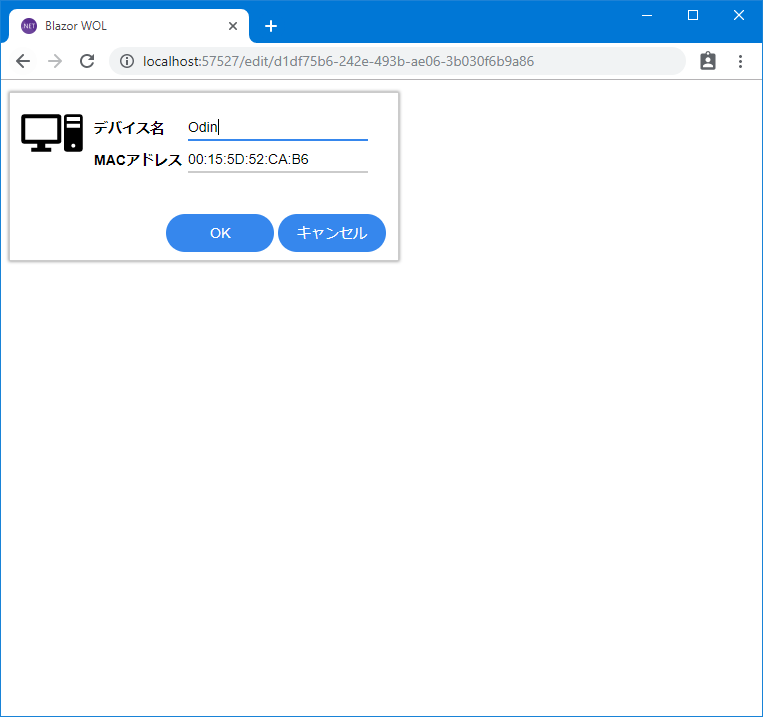
<p class="error-message">デバイス情報が見つかりません。</p>

}

}

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

1. 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

1. 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**>

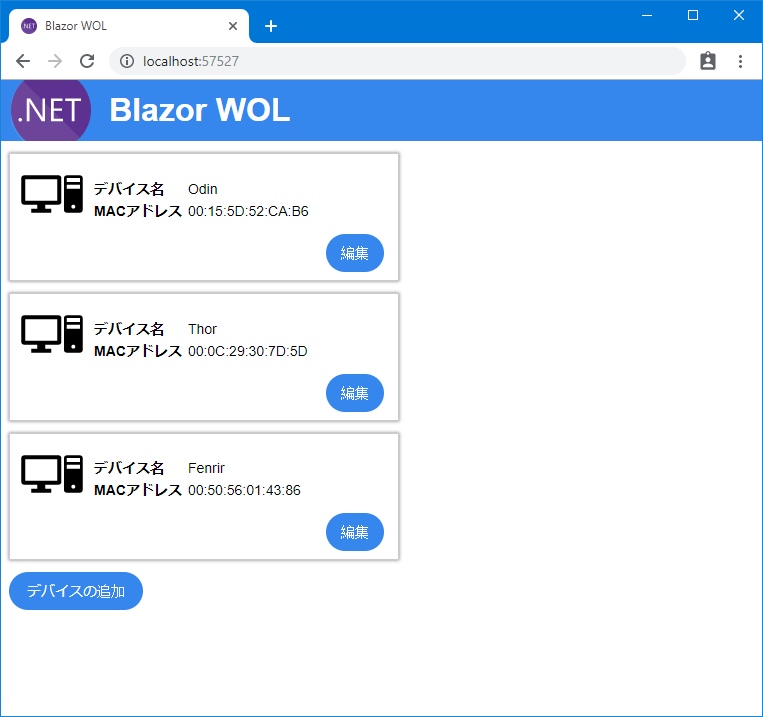
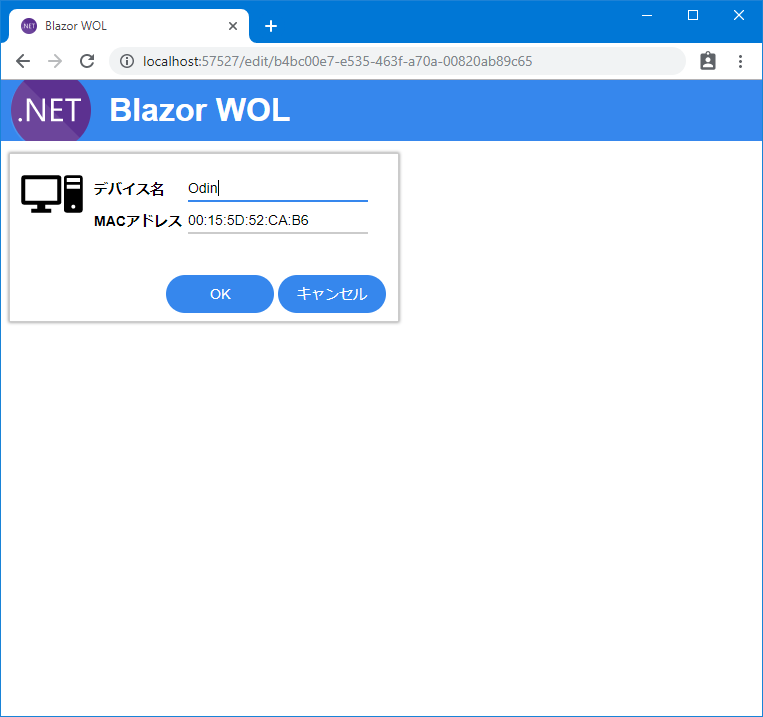
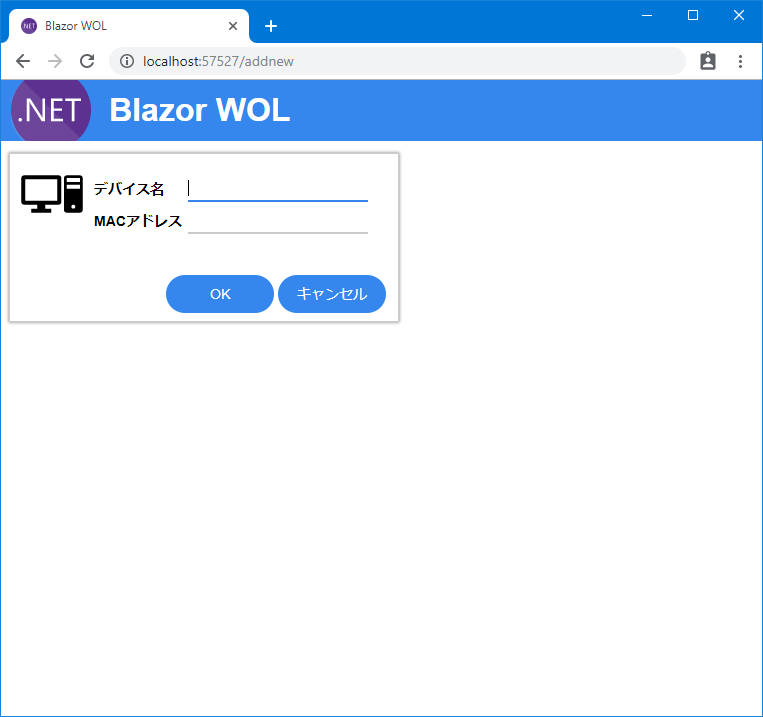
<p>Sorry, there's nothing at this address.</p>

</**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)">

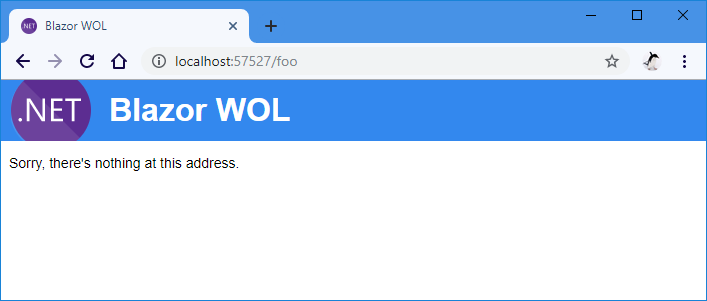
<p>Sorry, there's nothing at this address.</p>

</**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内の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。

なお、HttpClientオブジェクトを、必要になる都度、自分でインスタンス化 (new) して使うこともできなくはないのですが、ベースアドレスの設定などいくつかの考慮事項があります。

そのため通常は、Blazor WebAssemblyアプリケーションの開始時にDI機構に登録される、**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**。

さて今回、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。

これまでの例では、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。  
ところがDeviceServiceクラスは単純な C#ソースコード (.cs) で書かれた普通のクラスです。  
.razor で記述する Blazor コンポーネントとは異なり、@inject のようなディレクティブは使えません。

ではどうするかというと、このようにDI機構にそれ自体が登録されるクラスにて、DI機構経由での他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すればよいです。  
そうしておくと、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/ja-jp/aspnet/core/blazor/javascript-interop>

### 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

1. 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; }

1. 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();

}

}

1. 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" type="button" **@onclick**="OnDelete">削除</button>

}

<button class="button">OK</button>

<a class="button" href="/">キャンセル</a>

1. 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("/");

}

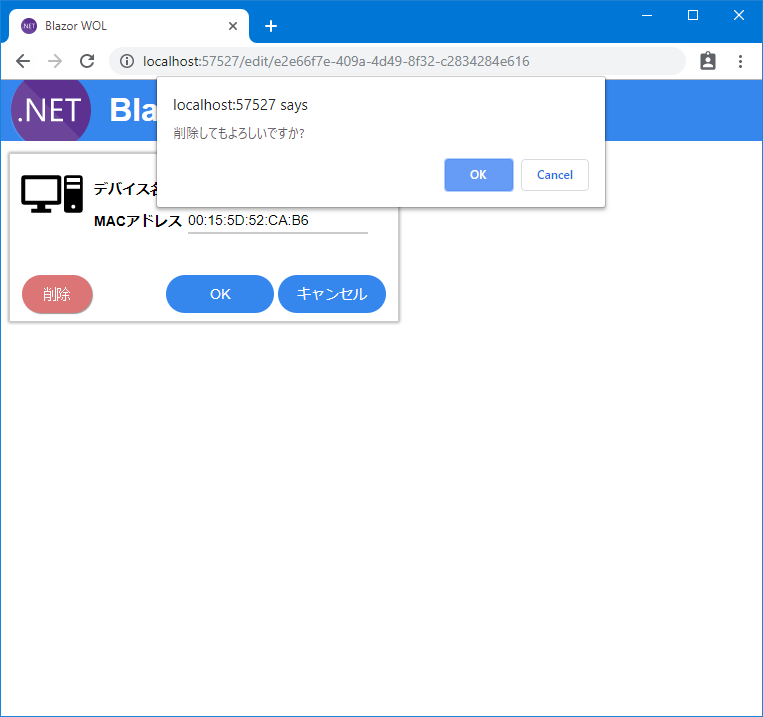
1. そしてこのOnClickDeleteメソッドを、デバイス情報入力フォームのマークアップにて、OnClickDeleteプロパティにバインドします。

<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、PWAへの対応方法は、コンポーネントをNuGetパッケージ化して再利用するには... 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。  
<https://github.com/kenakamu/blazor-workshop/tree/ja-jp>

また、Blazor 関連リンク集 (英語) **「Awesome Blazor」**(下記 URL ) には、豊富なライブラリ群の紹介 (例えば、マテリアルデザインのコンポーネント集などもここで見つかります) の他、英語とはなりますが、各種テキストやビデオ講習などもふんだんに掲載されています。

<https://github.com/AdrienTorris/awesome-blazor#awesome-blazor->

以上、参考になれば幸いです。

## 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11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->