Blazor WebAssembly
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 5

主な対象者 5

Blazor のバージョンとホスティングモデル 5

開発環境 7

もっとも手軽な方法 – GitHub Codespaces 7

Visual Studio Code の使用 8

Visual Studio の使用 9

本自習書で使用している開発環境 9

この自習書で作成する Web アプリケーション 9

Step 1. Blazor アプリ開発を開始 - ボイラープレートのビルド 10

Windows + Visual Studio 2022 環境の場合 12

Visual Studio Code の場合 14

その他のエディタを使う場合 15

GitHub Codespaces の場合 15

証明書のエラーが出てしまったら 15

macOS または Windows の場合 ([管理者として実行] を選択して開いたターミナルから) 16

Linux の場合 16

それでも改善しない場合 16

補足 - プロジェクトの構造 16

BlazorWorldClock.Server 16

BlazorWorldClock.Client 16

BlazorWorldClock.Shared 17

補足 - Blazor WebAssembly アプリケーションが立ち上がるまでの流れ 17

index.html 17

Program.cs 17
App.razor 18
Step 2. CSS スタイルシートを実装 18
Step 3. タイトルの変更 – Razor 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 19
コードの書き換えと実行 20
ホットリロードについて 21
Step 4. モデルクラスの追加 23
概要 23
手順 24
Visual Studio の場合 24
Visual Studio Code + C# DevKit 拡張の場合 24
Step 5. 時計表示ページの実装 - コンポーネントの追加 25
概要 25
手順 26
Visual Studio の場合 26
Visual Studio Code + C# DevKit 拡張の場合 27
Step 6. 時計表示ページの実装 – App コンポーネント内への埋め込み 28
概要 28
手順 29
Step 7. 時計表示ページ - リスト化 (繰り返し) 30
概要 30
手順 30
Step 8. 時計情報の取得・登録を行うサービスの実装 – DI の使用 32
概要 32
手順 33
Step 9. 非同期処理化 35
概要 35
手順 35

Step 10.　時計追加フォームを追記 – 入力とイベントのバインディング 36
概要 36
手順 37
補足 - Visual Studio 2022 上での Blazor WebAssembly プログラムのデバッグ 41
Step 11.　入力内容のチェック (入力検証、バリデーション) 43
概要 43
手順 44
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 47
Step 12.　時計追加を独立した URL に切り出し - ルーティング 49
概要 49
手順 49
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 52
概要 52
手順 52
Step 14.　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 53
概要 53
手順 54
Step 15. 時計情報の編集 - ルーティング引数 57
概要 57
手順 57
Step 16. 時計情報編集ページの実装 59
概要 59
手順 60
Step 17. タイトルヘッダの追加 - レイアウト 62
概要 62
手順 63
Step 18. サーバー側実装の開始 - ASP.NET Core Web API の実装 66
概要 66

手順 66
Step 19. サーバー側 Web API の呼び出し - HttpClient の使用 70
概要 70
手順 70
Step 20. 時計の削除機能を実装 - JavaScript 相互運用 72
概要 72
手順 72
Step 21. 仕上げ - 時刻表示の自動更新 75
概要 75
手順 75
次のステップへ 78
日本語の書籍 79
日本語のサンプル 79
英語の情報 79
あとがき 80
追補 81
ライセンス 81
関連リソース 81

本自習書について

本自習書は、Single Page Web Application (SPA) を含めた様々な Web アプリケーションを C#で実装できるフレームワーク、"Blazor (ブレイザー)" の WebAssembly 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 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 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 のバージョンとホスティングモデル

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.8.0 です。また、Blazor には以下の "ホスティングモデル" があります。

- ブラウザ上で実行される **Blazor WebAssembly**
- 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 **Blazor Server**
- ASP.NET Core サーバー上で静的 HTML コンテンツを生成して返す **Blazor SSR** (**S**tatic **S**erver-side **R**endering)

さらには Blazor WebAssembly と Blazor Server を同一サイト上で混合して自動で切り替える **Auto レンダーモード**という動作形態も存在します。

このように様々な実行形態が存在する Blazor ですが、本自習書が対象とする Blazor のホスティングモデルは、典型的な SPA の構成に最も近い、実装したプログラムがブラウザ上で動作する **Blazor WebAssembly** としました。以降、特に明記なく本書に "Blazor" とだけ記してある場合は、Blazor WebAssembly を指すものとします。

なお、Blazor WebAssembly 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 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

Blazor アプリケーション開発にあたって最低限必要な環境は以下のとおりです。

- **.NET 8.0 SDK (8.0.100 かそれ以降)**
  https://dotnet.microsoft.com/download/dotnet-core/8.0
- **上記 SDK が対応しているデスクトップ OS (Windows、macOS、各種 Linux ディストリビューション)**
- **任意のテキストエディタ**
- **インターネット接続**

以上の環境があれば、テキストエディタでソースコードを記述し、"dotnet" コマンドをターミナル (コマンドプロンプト) 上から実行することで、Blazor アプリケーションの実装・ビルド・実行が可能です。

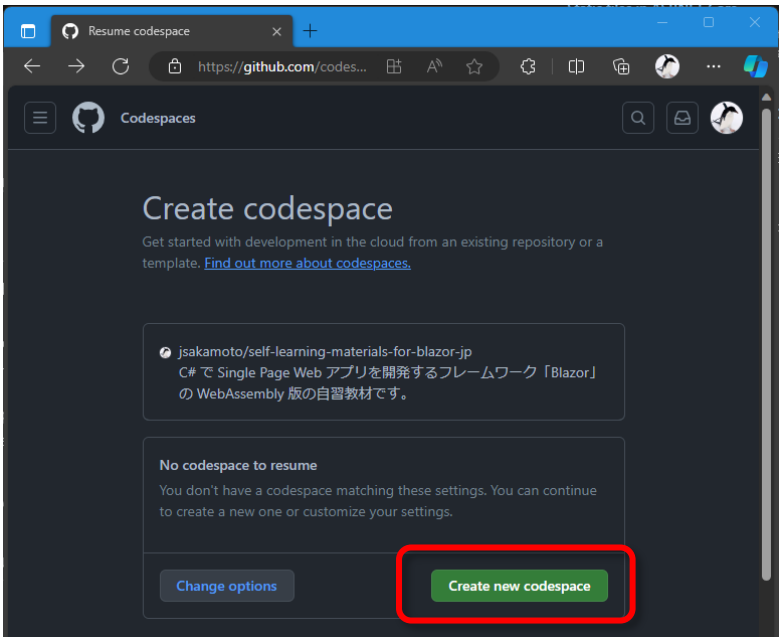
以下では、本自習書による Blazor WebAssembly アプリケーション開発を進めるにあたっての、推奨される開発環境について説明します。

もっとも手軽な方法 – GitHub Codespaces

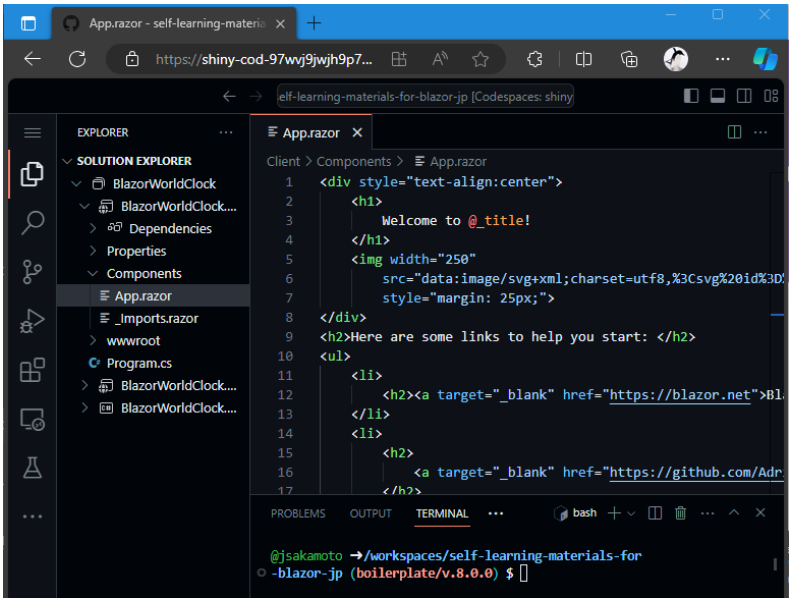
GitHub Codespaces は、GitHub が提供する仮想マシン上に構築される開発環境です。インターネット接続と Web ブラウザ、GitHub アカウントさえあればすぐに始められます。PC 上に .NET SDK をインストールする必要すらありません。Web ブラウザ上で Visual Studio Code 相当のコードエディタ環境が提供されるので、何となれば iPad をはじめとした PC 以外のデバイスでも作業できます。

GitHub アカウントをお持ちでサインイン済みであれば、下記 URL をブラウザで開くと、
https://codespaces.new/jsakamoto/self-learning-materials-for-blazor-jp/tree/boilerplate%2Fv.8.0.0?quickstart=1

下図のとおり、Codespaces の作成を確認するページが表示されます。



ここで [Create new codespaces] ボタンをクリックすると、GitHub 上で Linux ベースの仮想マシンが生成・起動され、ブラウザ上からは VSCode 相当のエディタおよびターミナルにて接続されます。



この仮想マシンには .NET SDK 8 が既にインストール済み、かつ、作業フォルダには本自習書の「Step 1. Blazor アプリ開発を開始-ボイラープレートのビルド」のソースコードがチェックアウト済みとなっています。
Codespaces 作成直後は、C#拡張のインストールなどの初期化処理の時間が少々かかりますが、それでもローカル PC 環境に手を入れることなく、すぐに本自習書に基づく開発作業を開始できます。

GitHub Codespaces は個人用アカウントですと、最長で 60 時間ぶんの無料枠で利用でき、超過分はもっとも安価な場合で 1 時間 $0.18 の料金で利用できます。詳細については下記リンク先を参照ください。
https://github.co.jp/features/codespaces

Visual Studio Code の使用

手元のローカル PC 上に、.NET SDK をインストールして開発作業を行なう場合は、**Visual Studio Code** の利用をお勧めします。特に、**C#** (v.2.10.28 以上)、および **C# Dev Kit** の Visual Studio Code 拡張を追加しての利用が推奨されます。これら拡張がインストールされていれば、構文ハイライトやインテリセンスをはじめとした、強力な開発支援が得られます。なお、C# Dev Kit 拡張の利用にあたっては、Visual Studio 相当の使用条件があります。

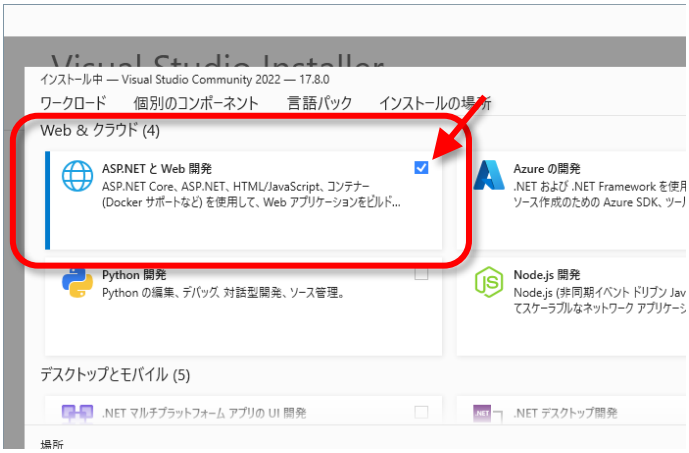
- **Visual Studio Code**
  https://code.visualstudio.com/
- **C# for Visual Studio Code**
  https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp
- **C# Dev Kit for Visual Studio Code**
  https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csdevkit

Visual Studio の使用

Windows PC をお使いの場合は、統合開発環境である Visual Studio の使用もお勧めです。

- **Visual Studio 2022 - 17.8.0 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/

なお、Visual Studio のインストールオプションとして、「**ASP.NET と Web 開発**」ワークロードが選択されている必要があります (下図)。



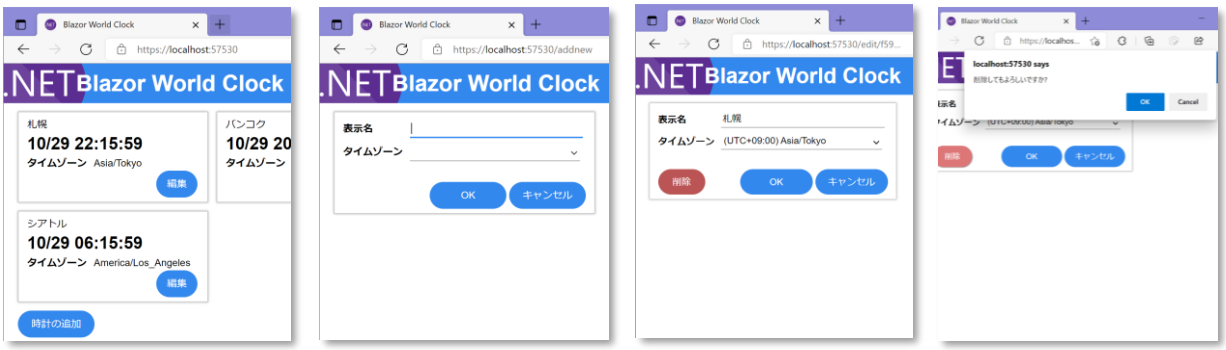
本自習書で使用している開発環境

本自習書では基本的には Windows 上で Visual Studio 2022 17.8.0 以降を使っての手順で説明いたします。いっぽうで、他の OS や Visual Studio Code をはじめとした各種エディタを利用されている場合に向けて、適宜、dotnet CLI で行なう場合の手順も付記します。

この自習書で作成する Web アプリケーション

この自習書では、ページ上に複数のタイムゾーンの現在時刻を一斉表示する「世界時計」を Blazor WebAssembly を使って実装します。

表示する時計の表示用の名称とタイムゾーンを、追加、変更、削除するページを備えます。



登録された時計は ASP.NET Core サーバー側で JSON 形式のテキストファイルに保存します。

なお、繰り返しになりますが、Blazor WebAssembly は、他の一般的な JavaScript 製 SPA フレームワーク類と同じく Web ブラウザ上で稼働するため、**サーバー側実装は本質的には必要ではありません**。

そのため、例えば本自習書で題材としている程度の "世界時計" Web アプリの時計情報の保存先としては、**実は Web ブラウザのローカルストレージでも充分**とも言えます。

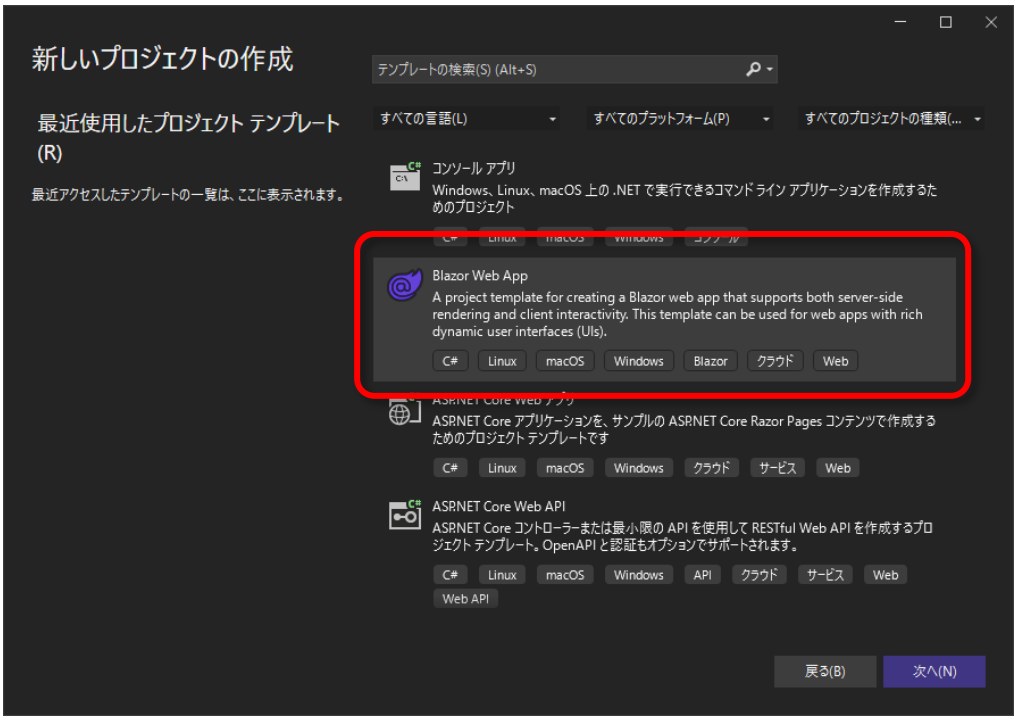
そしてまた、サーバー側実装を伴う場合であっても、Blazor WebAssembly は C#で実装するとはいえあくまでも Web ブラウザをプラットフォームとしているだけですから、サーバー側とのやりとりは一般的なブラウザアプリケーションと何ら違いはありません。つまり、**サーバー側実装が C# (ASP.NET Core) である必要もありません**。

しかしながら、本自習書を通して、Blazor WebAssembly 側の魅力や利点のみならず、**サーバー側実装に C# (ASP.NET Core) を採用した場合の連携の強力さ**をお伝えしたく、**あえて** C# (ASP.NET Core) によるサーバー側実装を絡めた内容としました。

Step 1. Blazor アプリ開発を開始 - ボイラープレートのビルド

ではいよいよここから、"Blazor World Clock" Web アプリの開発に着手していきましょう。

さて、前述の開発環境が整うと、Visual Studio 2022 をお使いの場合は、そのプロジェクトテンプレートにて、「Blazor Web App」を選んでプロジェクト新規作成することができます (下図)。



あるいは dotnet CLI で作業する場合は、下記コマンドを実行します。

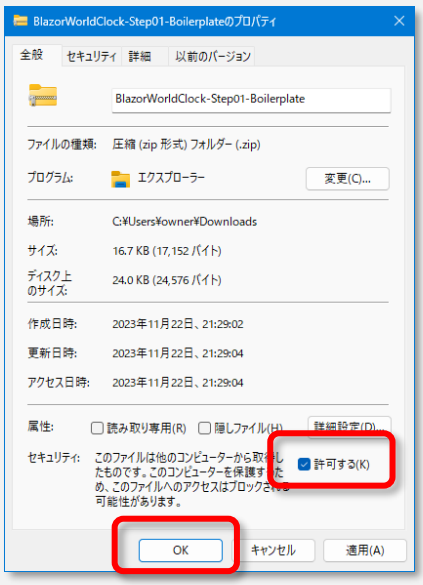
```
dotnet new blazor
```

しかし、これら .NET SDK に備え付けの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や、共有レイアウトの構成が実装済みであり、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。

また、これら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、サーバー側レンダリングを含む構成でプロジェクトが生成されます。これはこれでサーバー側からクライアント側まで一貫して C#で開発することの強力さをうかがえる実践的な構成となります。しかしながら、このテンプレートでは、静的コンテンツサーバーに配置可能な Blazor WebAssembly アプリにならないことと、「初めての Blazor プログラミングの習得」という目標においては学ぶべき事が少し多すぎます。

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、**Blazor としてはサーバー側実装を必要としない、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。つきましては、本自習書に同梱、又は下記リンク先からダウンロードした BlazorWorldClock-Step01-Boilerplate.zip を、お好みの作業フォルダーに解凍してください。

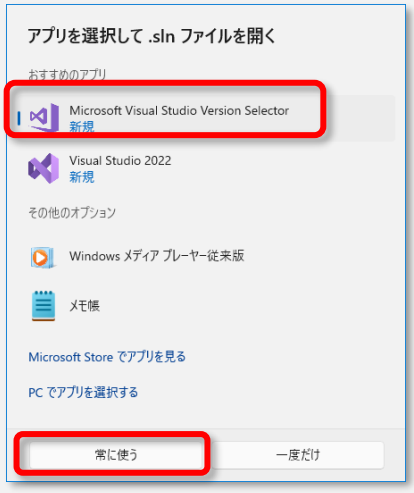
https://github.com/jsakamoto/self-learning-materials-for-blazor-jp/releases/download/doc%2F8.0.0/BlazorWorldClock-Step01-Boilerplate.zip



※注意: Windows 上で作業されている場合は、**解凍前**に、エクスプローラーでダウンロードした Zip ファイルのプロパティウィンドウを開き、[セキュリティ] の [...このファイルのアクセスはブロックされる可能性があります] の項目にて、**[許可する(K)]** のチェックを On にして [OK] をクリックする、**「ブロックの解除」を忘れずに**行っておいてください。

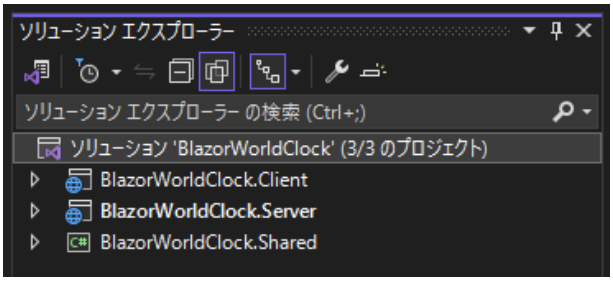
Windows + Visual Studio 2022 環境の場合

解凍すると、"BlazorWorldClock.sln" というソリューションファイルがあります。これをダブルクリックして Visual Studio 2022 で開きます。

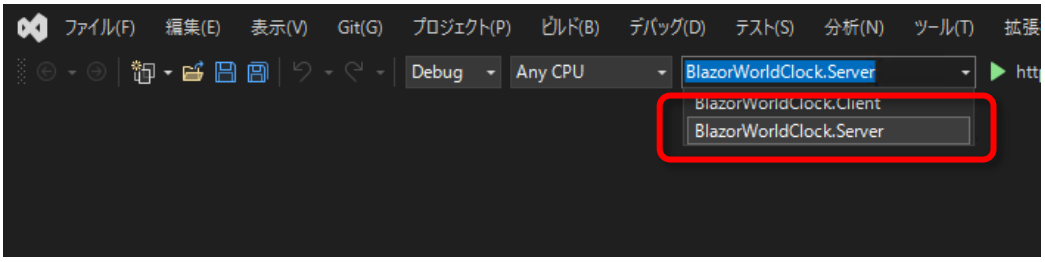


※ .sln ファイルをダブルクリックしたときに「アプリを選択して .sln ファイルを開く」ダイアログが開いた場合は、[Microsoft Visual Studio Version Selector] を選択して、[常に使う] ボタンをクリックしてください。

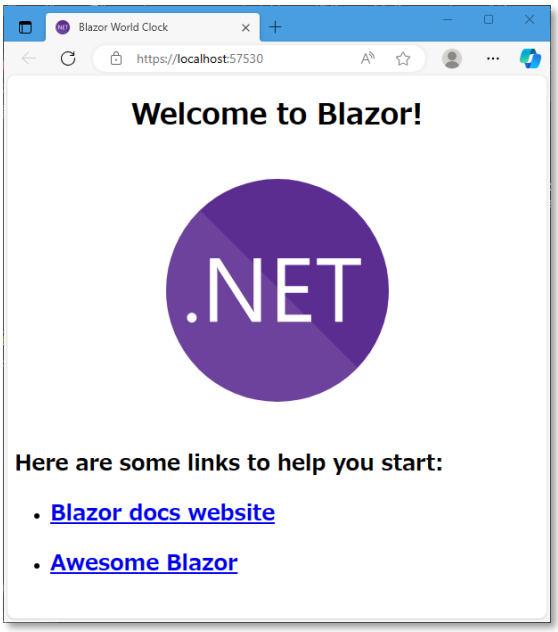
このソリューションファイルを開くと、Visual Studio のソリューションエクスプローラーウィンドウにて、3 つのプロジェクトが収録されているのがわかります (下図)。



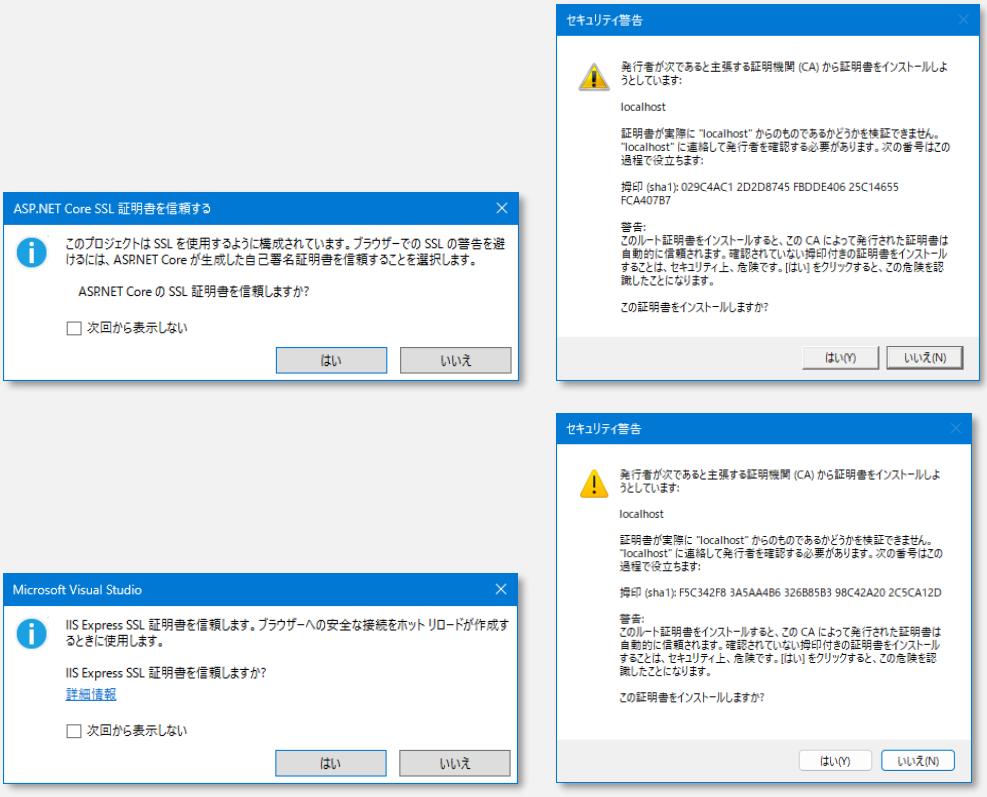
Visual Studio のツールバーにある [スタートアップ プロジェクト] のドロップダウンリストから **"BlazorWorldClock.Server" を選択**しておき、**常にこのプロジェクトを実行**するように固定します (下図)。



スタートアッププロジェクトを固定したら **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、下図のとおり表示されれば成功です。



※ 初めて起動するときに、下図のように**「ASP.NET Core SSL 証明書を信頼しますか?」「IIS Express SSL 証明書を信頼しますか?」**の確認メッセージボックスが表示されることがあります。

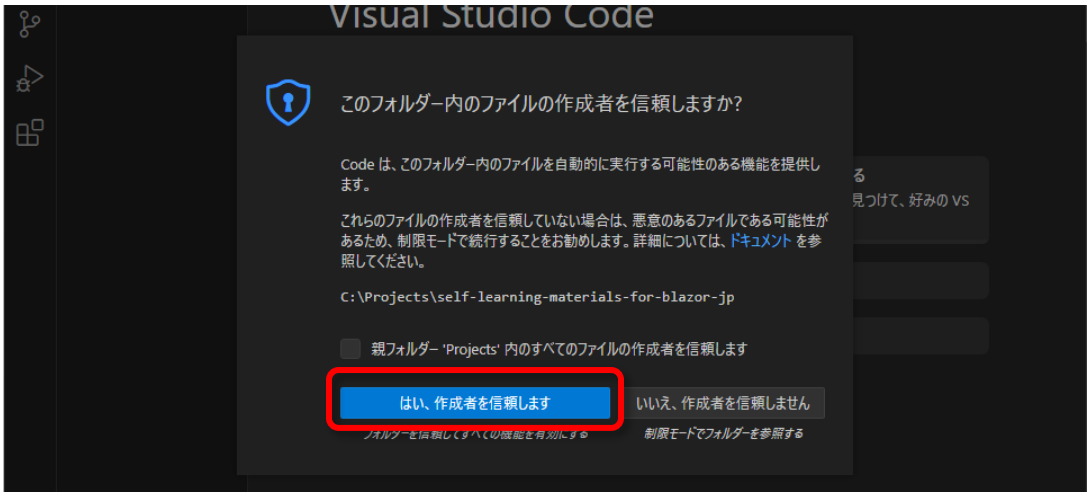


この場合は **[はい]** で回答して、**これら証明書を信頼**してください。

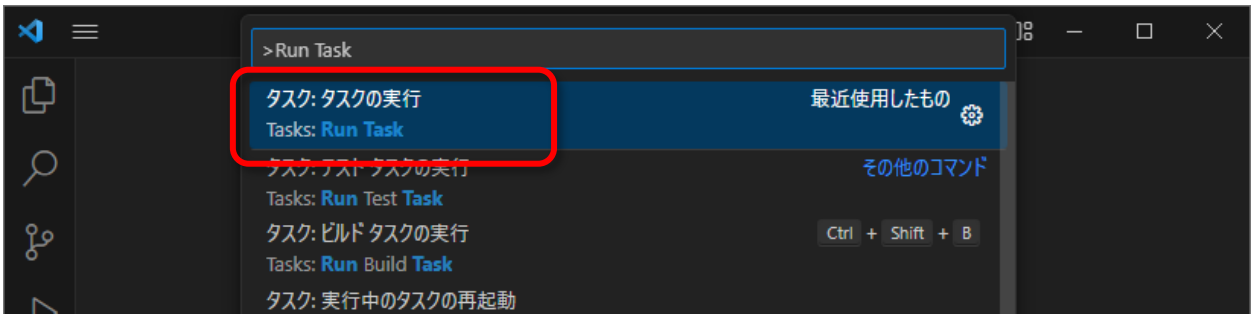
Visual Studio Code の場合

Visual Studio Code を使って作業する場合は、本自習書に同梱のボイラープレート BlazorWorldClock-Step01-Boilerplate.zip を解凍した先のフォルダを Visual Studio Code で開きます。

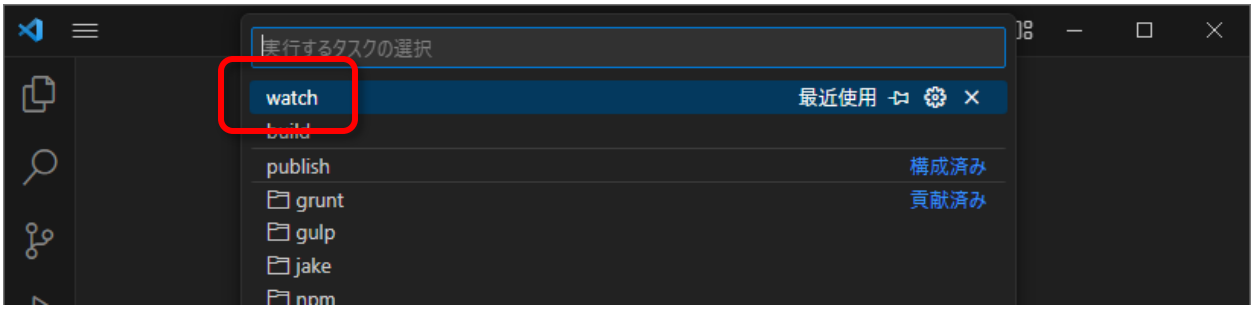
その際に、「このフォルダー内のファイルの作成者を信頼しますか?」という確認メッセージが表示されましたら (下図)、適宜ご判断の上、**[はい、作成者を信頼します]** をクリックいただくとよいかと思います。



なお、このボイラープレートには、Visual Studio Code 用のタスク設定も収録しています (".vscode" サブフォルダー)。これにより、**Ctrl + Shift + P** (Windows/Linux) または **⌘+ Shift + P** (macOS) で Visual Studio Code のコマンドパレットを開き、"Run Task" などと入力して絞り込まれた結果から **"タスク: タスクの実行 (Tasks: Run Task)"** を選択すると、



その中に **"watch"** タスクが用意されていますので、これを選択・実行してください。すると、前述の Visual Studio 2022 での開発時と同じように、ビルドとブラウザの起動が実行されます。



その他のエディタを使う場合

Visual Studio および Visual Studio Code 以外のエディタ環境で作業する場合は、dotnet コマンドを直接実行することで、ビルドとブラウザの起動を行ないます。ターミナル (コマンドプロンプト) を開き、カレントディレクトリを本自習書に同梱のボイラープレート BlazorWorldClock-Step01-Boilerplate.zip を解凍した先のフォルダ以下、./Server フォルダーにまで移動してから、**dotnet watch** コマンドを実行してください (下記例)。

```
cd ./Server
dotnet watch
```

そうすることで、ビルドとブラウザの起動が実行されます。

GitHub Codespaces の場合

GitHub Codespaces での作業中の場合、残念ながら本稿執筆時点では、dotnet watch によるファイル変更の監視と自動ビルドが機能しません。GitHub Codespaces 上では、ターミナルを開いて、カレントディレクトリを BlazorWorldClock.Server プロジェクトがあるフォルダー (./Server) に移動してから、**dotnet run** コマンドを実行してください (下記例)。

```
cd ./Server
dotnet run
```

そうすることで、ビルドと開発サーバーの起動が行なわれます。なお、そうして起動した Blazor アプリケーションに対してブラウザは自動では開きません。代わりに GitHub Codespaces の PORT タブから明示的にブラウザを開く必要があります。

証明書のエラーが出てしまったら

何らかの事情でサーバー証明書がうまく開発環境にインストールできていなかった場合、下図のように「ERR_CERT_INVALID」の証明書エラーが発生します。

この場合は、Visual Studio をお使いであればその Visual Studio を終了させ、それ以外の場合は実行中の dotnet コマンドとエディタ類をすべて終了させた上で、以下のコマンドを実行してから、再度改めてプログラムを起動しなおしてみてください。

macOS または Windows の場合 ([管理者として実行] を選択して開いたターミナルから)

```
dotnet dev-certs https
dotnet dev-certs https -clean
dotnet dev-certs https –trust
```

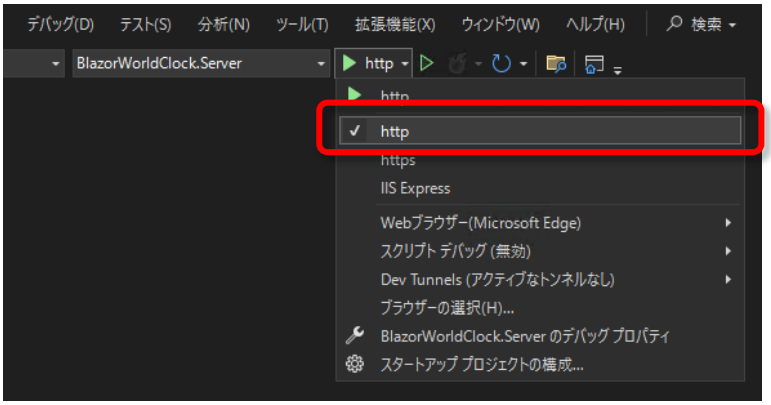
Linux の場合

下記リンク先の手順をお試しください。

https://learn.microsoft.com/ja-jp/aspnet/core/security/enforcing-ssl?view=aspnetcore-8.0&tabs=visual-studio%2Clinux-ubuntu#trust-https-certificate-on-linux

それでも改善しない場合

Windows 上で Visual Studio をお使いの場合、上記処置でもなお改善しない場合は、HTTPS プロトコルを使わずに HTTP プロトコルでとりあえず実行、学習を進めることも可能です。Visual Studio の起動プロファイルのドロップダウンリストを開き、[http] を選択してから実行してください (下図)。



補足 - プロジェクトの構造

この BlazorWorldClock 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。

BlazorWorldClock.Server

3 つあるプロジェクトのうち、BlazorWorldClock.Server は、従来からある ASP.NET Core Web アプリとほぼ相違ない、サーバー側実装です。静的コンテンツサーバーとして各種 HTML や CSS、Blazor WebAssembly アプリケーションとしてのアセンブリファイル (.wasm) を提供するほか、本自習書の後半にて、時計の情報を保存・取得する REST API サーバーとして実装します。

BlazorWorldClock.Client

この自習書でのいちばんの注目ポイントは、BlazorWorldClock.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 BlazorWorldClock.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 (いわゆる.dll ファイル、Blazor WebAssembly ではファイル拡張子は .wasm) 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。

BlazorWorldClock.Shared

BlazorWorldClock.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 8.0 クラスライブラリです。
このプロジェクトは、BlazorWorldClock.Client プロジェクトと BlazorWorldClock.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 (データ転送オブジェクト) 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 WebAssembly アプリケーションが立ち上がるまでの流れ

index.html

ブラウザに最初に読み込まれるのは、BlazorWorldClock.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">` タグの内容が、Razor コンポーネントによって生成される DOM コンテンツに書き換えられる仕組みです。

Program.cs

ここで BlazorWorldClock.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 (そして最終的には .wasm) 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 で、上から 1 行ずつ実行されます。
この Program.cs 内に下記の記述があります。

```
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 Razor コンポーネントのレンダリング結果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

BlazorWorldClock.Client プロジェクトには、Components サブフォルダー内に 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 Razor 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→.wasm 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 WebAssembly 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 標準の技術要素で動作する 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」フォルダー以下、BlazorWorldClock.Client プロジェクト内の wwwroot フォルダーの styles.css) を適用しておいてください。
styles.css は、下記 GitHub リポジトリのリンク先からも入手できます。

🌐 https://github.com/jsakamoto/self-learning-materials-for-blazor-jp/blob/v.8.0.0/Client/wwwroot/styles.css

> ※ 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 Razor 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://learn.microsoft.com/ja-jp/aspnet/core/blazor/components/css-isolation

> ※あらゆる外観を CSS を使ってすべて自分で実装するばかりではなく、Material Design や Fluent UI などの外観を持つコンポーネントを Blazor 向けにリリースした**「UI コンポーネントライブラリ」**も活用できます。
> 本自習書では割愛しますが、下記 URL などが参考になるかと思います。
> https://jsakamoto.github.io/awesome-blazor-browser/#libraries---extensions-component-bundles

Step 3. タイトルの変更 – Razor 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。

Razor コンポーネントのソースコード (.razor) には、**HTML タグ**の記述と、**C# のコードブロック (@code { … } で囲まれている部分)** 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 C# 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。App.razor には下記のとおり、C#コードブロックにて、_title という string 型のフィールド変数が定義されており、"Blazor" という文字列リテラルで初期化されているのがわかります。

```
...
@code {
    private string _title = "Blazor";
}
```

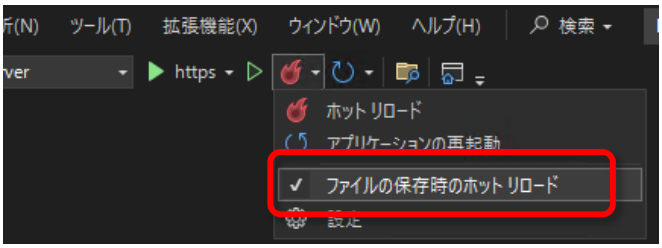
そして、 .razor 中の HTML 記述中に、**"@"** に続けて記述する C#コードで、この.razor 中のコードブロックで実装したメンバーを参照 (バインド) できます。App.razor では、下記のとおり、h1 タグの表示テキストとして、前述の _title フィールド変数がバインドされています。

```
...
  <h1>
    Welcome to @_title!
  </h1>
...
```

コードの書き換えと実行

試しに App.razor を書き換えて、フィールド変数がバインドされている様子を確認してみましょう。

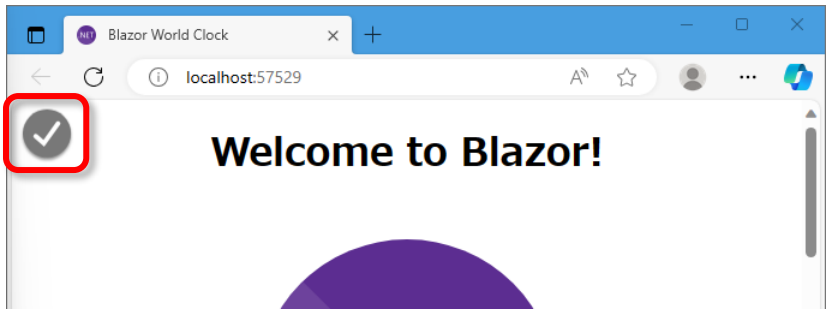
Visual Studio をお使いの場合は、ツールバー上の炎のアイコン横のドロップダウン記号をクリックし、[ファイル保存時のホットリロード] の項目にチェックが入っていなければこれをクリックしてチェックを On にしておきます (下図)。



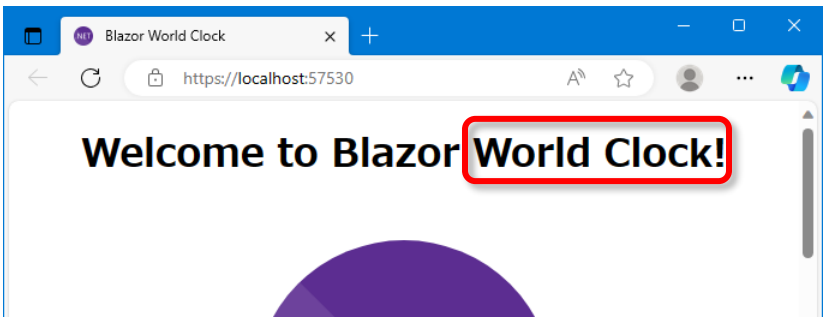
その上で、App.razor 中のコードブロックに記載のある、_title フィールド変数を初期化している文字列を、"Blazor" から "Blazor **World Clock**" と書き足して変更してみてください。

```
@code {
    private string _title = "Blazor World Clock";
}
```

App.razor を上記のとおり編集し保存すると、ホットリロード機能 (後述します) によって変更が適用され、そのインジケーター (丸いチェックマーク) がブラウザ上に表示されます。



その上でブラウザで再読み込みを実行すると、h1 要素のテキストにバインドしている _title フィールド変数の初期化内容の変更に応じて、表示も変更されているのが確認できます（下図）。



ホットリロードについて

Visual Studio での実行や、Visual Studio Code の watch タスク実行、および dotnet watch コマンドによる実行では、.razor ファイルや.cs ファイルを変更して保存すると、**"ホットリロード"** と呼ばれる機能によって、変更が実行中の Blazor アプリケーションに自動で適用されます。ホットリロードでは、コード変更の**差分**が実行中のプログラムに対して**動的に適用**されます。そのため、**プログラム全体を再起動することがなく**、基本的に**コンポーネントの状態 (フィールド変数やプロパティに格納されている値) はそのまま維持**されます。

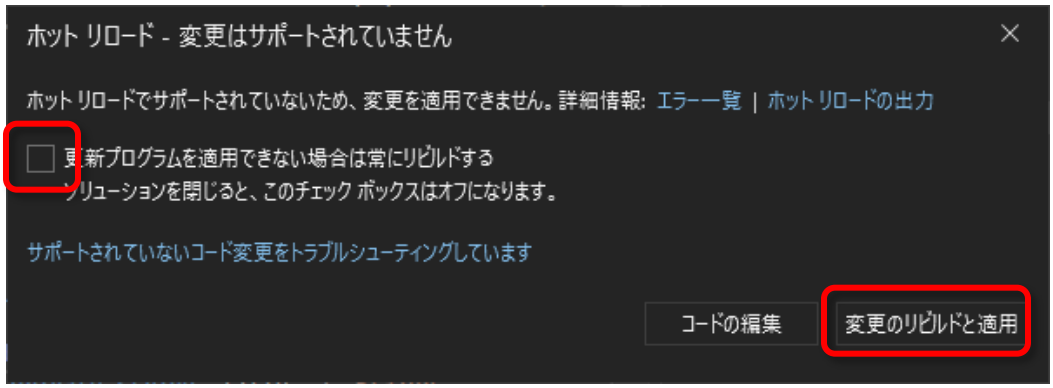
なお、先の _title フィールド変数を書き換えた例では、ホットリロードが適用されたにもかかわらず、ブラウザで再読み込みを行なうまで実際の表示が変更後の "Blazor World Clock" ではなく "Blazor" のままでした。これは、ホットリロードは成功しているのですが、コード変更の対象が「フィールド変数の初期化」処理であったためです。つまり、ホットリロードが適用されても App コンポーネントは既にインスタンス化されていて実行中であり、フィールド変数 _title の内容が勝手に書き換わることはありません。そのため、ブラウザを再読み込みして App コンポーネントを再度インスタンス化して、コード変更後の _title フィールド変数の初期化処理が行なわれるまでは、一見、変更が表示に反映されていないように見えた次第です。

なおホットリロードが有効になっていない環境、例えば GitHub Codespaces 上で dotnet run コマンドによる実行を行なっている場合は、.razor ファイルや .cs ファイルを変更したら、Ctrl + C を押して実行中の dotnet run コマンドを中止し、もういちど dotnet run コマンドを実行して再度ビルドして実行する必要があります。

> ※ .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→.wasm 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→.wasm ファイルを更新してからブラウザで再読み込みするか、あるいは**ホットリロード機能による変更の動的適用**が行なわれる必要があります。
> **.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。

しかしながら Blazor におけるホットリロードには制約・限界があり、ホットリロードによる変更適用ができない場面がしばしばあります。その場合は以下のように、再ビルドしてよいかどうか確認されることがあります。

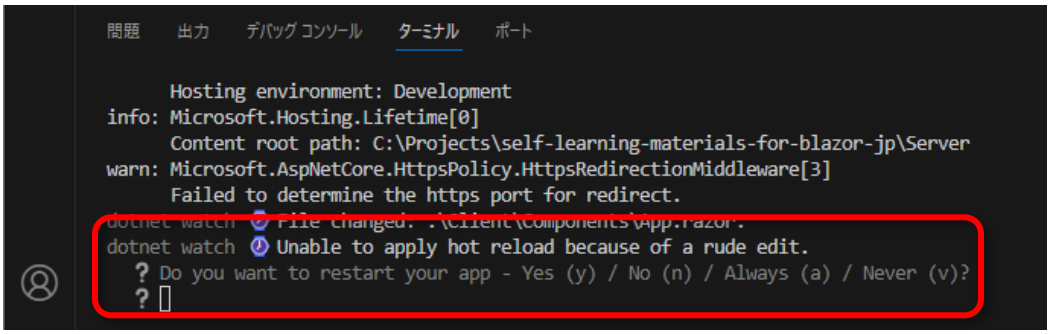
Visual Studio の場合



この場合はその都度判断の上、[コードの編集] を選択して変更を適用せずにコーディング作業を継続するか、[変更のビルドと適用] を選択して再ビルドを実行して変更を適用してください。

または、[更新プログラムを適用できない場合は常にリビルドする] のチェックを On にしてから [変更のリビルドと適用] をクリックすると、以後はこのダイアログを表示することなく、ホットリロードによる変更適用ができない場合は常に自動でリビルドして変更を適用するようになります。

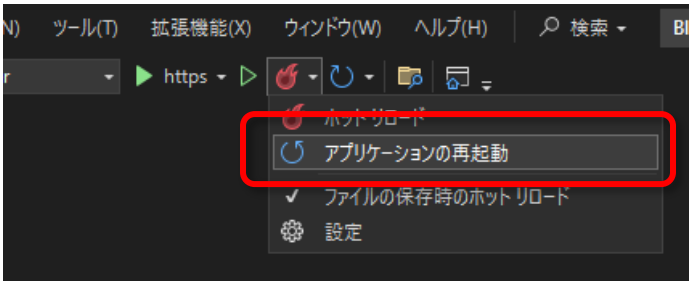
Visual Studio Code の watch タスクによる実行 または dotnet watch コマンド実行の場合



Visual Studio Code の watch タスクおよび dotnet watch コマンド実行の場合は、上図のとおり、dotnet watch を実行中のターミナルに表示されます。n キーを押して [No (いいえ)] で回答し変更を適用せずにコーディング作業を継続するか、y キーを押して [Yes (はい)] で回答し再ビルドを実行して変更を適用してください。

あるいは a キーを押して [Always (常に)] を回答して、以後、ホットリロードによる変更適用ができず再ビルドが必要になったら常に自動で再ビルドするように回答しておくとよいかと思います。

また、場合によっては、ホットリロードが途中から機能しなくなり、ブラウザ上でリロードしても変更が適用されなかったりエラーが表示されて実行できなかったりすることがあるかもしれません。その場合は、Visual Studio をお使いの場合は、ツールバー上の炎のアイコン横のドロップダウン記号をクリックし、**[アプリケーションの再起動]** をクリックして手動で再ビルドによる変更の適用を実施してください (下図)。



および Visual Studio Code の watch タスク実行か dotnet watch コマンドで実行中の場合は、dotnet watch コマンドを実行中のターミナルで **Ctrl + R** を押して、手動で再ビルドを指示してください。

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、**表示する時計を表現する、Clock クラス**を実装します。

Clock クラスは以下のプロパティおよびメソッドを持たせます。

- 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)
- 時計の表示用の名称である **Name プロパティ** (文字列型)
- 表示する時計のタイムゾーンを示す **TimeZoneId プロパティ** (文字列型)
- TimeZoneId プロパティで示されるタイムゾーンにおける現在時刻を返す **GetCurrentTime()** メソッド (DateTime 型)

本自習書で作成する BlazorWorldClock アプリケーションでは、この Clock 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。

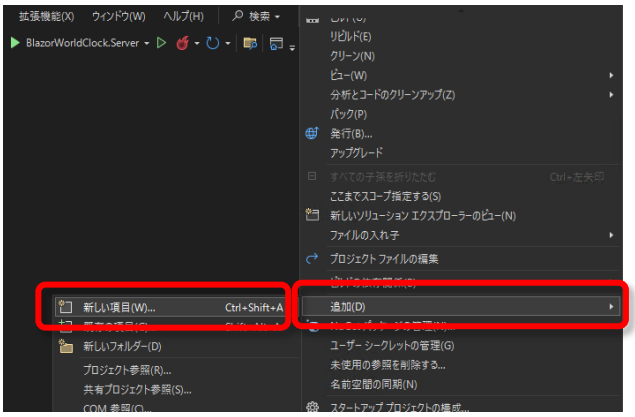
まずはクライアント側の実装を進めることにします。
そのため暫くは、Clock クラスは BlazorWorldClock.Client プロジェクト上でのみ使用します。
しかしいずれ、サーバー側 BlazorWorldClock.Server プロジェクトでも、永続化処理で Clock クラスを参照することになります。

このように **Clock 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、Clock 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、**BlazorWorldClock.Shared** プロジェクトに実装することにしましょう。

> ※ 世界時計を実装するにあたり、タイムゾーンの管理および各地のローカル時刻への変換は、.NET に備え付けの System.TimeZoneInfo クラスを使用します。Blazor WebAssembly アプリケーションは Web ブラウザ上で実行されますが、C# コードを JavaScript に変換しているのではなく、**.NET 実行環境がそのまま Web ブラウザ上で再現されている (MSIL インタプリタ)** ので、このように**普通に.NET 備え付けのクラスが使えたり**します。もちろん、Web ブラウザというプラットフォーム上で実行される以上、できることは Web ブラウザ内でできることに限られます。例えば System.Net.Sockets.TcpClient クラスを使って TCP ソケット通信をする、などといったことはできません。

手順

Visual Studio の場合

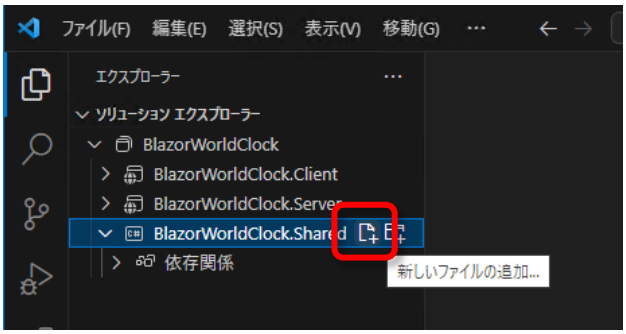
1. 

   Visual Studio のソリューションエクスプローラー上で BlazorWorldClock.Shared プロジェクトを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。
   (または、BlazorWorldClock.Shared プロジェクトを選択した状態で、キーボードから Ctrl + Shift + A、あるいは Shift + F2 のいずれかを打鍵します)

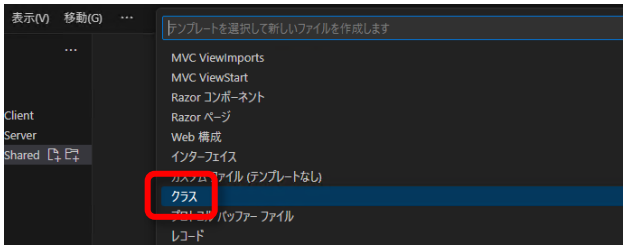
2. 

   すると「新しい項目の追加」ダイアログが現れるので、入力欄が "Clock.cs" となるように入力して、Enter キーを押すか [追加(A)] ボタンをクリックします

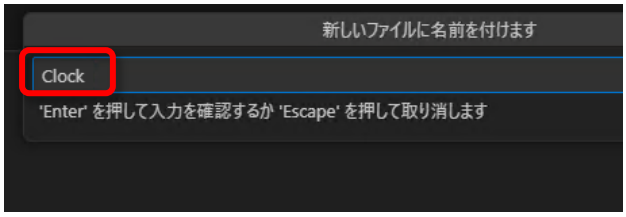
Visual Studio Code + C# DevKit 拡張の場合

1. 

   Visual Studio Code のソリューションエクスプローラー上で BlazorWorldClock.Shared プロジェクトを右にある [新しいファイルの追加] ボタンをクリックします。

2. 

   するとテンプレート一覧がコマンドパレットに現れるので、[クラス] を選択します。

3. 

   するとファイル名を指定する欄が開くので、ここに "Clock" と入力し Enter キーを押します。

> ※ 本自習書では基本、Visual Studio を使った手順で説明していますが、実のところ、上記ソリューションエクスプローラーでの右クリックから [追加(D)] - ～ でやっていることは、**単に空ファイルを追加しているのと変わりありません**。
> C# DevKit を使わない Visual Studio Code や dotnet CLI ベースで開発している場合は、以後、Visual Studio のソリューションエクスプローラーーの右クリックからの [追加(D)] - ～ の手順は、該当する空ファイルを追加する手順として読み替えてください。

4. Clock.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。

> ※Clock クラスのアクセス制御を "internal" から "public" に変更するのを忘れずに。
> ※Visual Studio または Visual Studio Code を使っている場合は、"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。

```
namespace BlazorWorldClock.Shared;

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

    public string Name { get; set; } = "";

    public string TimeZoneId { get; set; } = "";

    public DateTime GetCurrentTime()
    {
        var timeZone = TimeZoneInfo.FindSystemTimeZoneById(this.TimeZoneId);
        return TimeZoneInfo.ConvertTimeFromUtc(DateTime.UtcNow, timeZone);
    }
}
```

Step 5. 時計表示ページの実装 - コンポーネントの追加

概要

続けて、Clock クラスのオブジェクトを、その時刻とともに表示する手はずを進めていきましょう。
BlazorWorldClock 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 Clock クラスを表示する Razor コンポーネント "ClockList.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、Razor コンポーネント (.razor ファイル) は Components フォルダー内、特に固有のページとして表示するものはその下の **Pages フォルダー**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 Clock 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

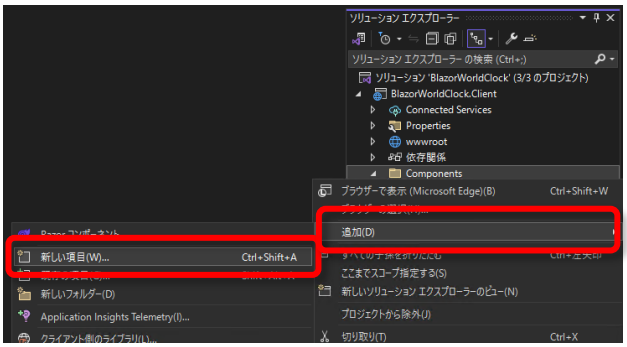
ところで、先のステップで作成した Clock クラスは、BlazorWorldClock.Shared プロジェクト内に作成しましたので、その名前空間が "BlazorWorldClock.Shared" となっています。そのため、クライアント側実装、すなわち、BlazorWorldClock.Client プロジェクト内の Razor コンポーネントのソースコード (.razor) 内からは、このままでは Clock クラスを使うのに名前空間付きの完全限定名 ("BlazorWorldClock.Shared.Clock") で指定する必要があり、少々煩わしいです。そこで、いずれの .razor ファイルからでも "Clock" のクラス名だけで参照できるように名前空間を既定で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、**Components/_Imports.razor** 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となっています。そのため、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 Components/_Imports.razor ファイルが含まれています。
エディタでこの _Imports.razor ファイルを開き、Clock クラスの名前空間 "BlazorWorldClock.Shared" を開いておく @using 節を追加してください。

```
...前半変更なし...
@using BlazorWorldClock.Client.Components
@using BlazorWorldClock.Shared
```

これで準備が整いましたので、引き続き ClockList.razor ファイルの新規作成へと進めていきましょう。

Visual Studio の場合

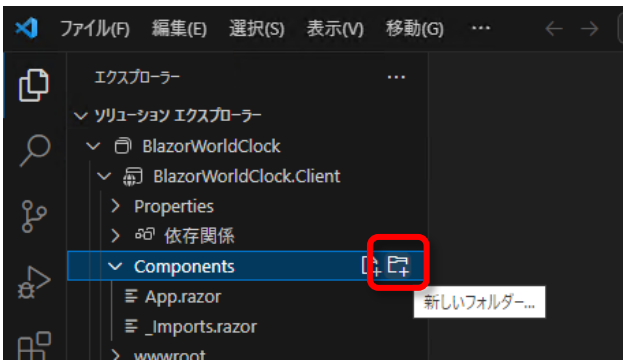
1. 

Visual Studio のソリューションエクスプローラー上で BlazorWorldClock.Client プロジェクトの Components フォルダーを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。
(または、Components フォルダーを選択した状態で、キーボードから Ctrl + Shift + A、あるいは Shift + F2 のいずれかを打鍵します)。

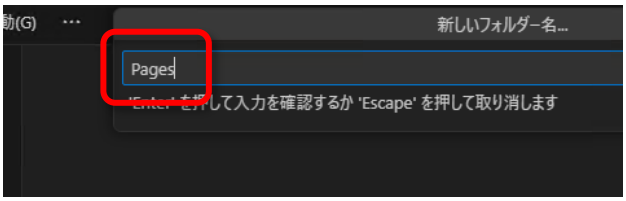
2. 

「新しい項目の追加」ダイアログが現れるので、入力欄が "Pages/ClockList" となるように入力して、Enter キーを押すか [追加(A)] ボタンをクリックします。

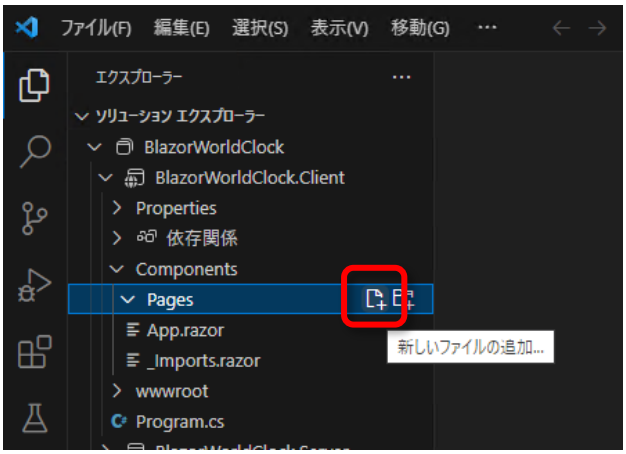
Visual Studio Code + C# DevKit 拡張の場合

1. 

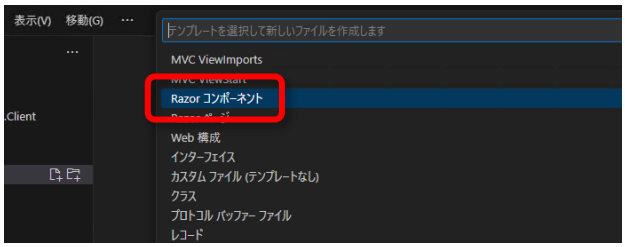
   Visual Studio Code のソリューションエクスプローラー上で BlazorWorldClock.Client プロジェクト以下、Components フォルダの右にある [新しいフォルダーの追加] ボタンをクリックします。

2. 

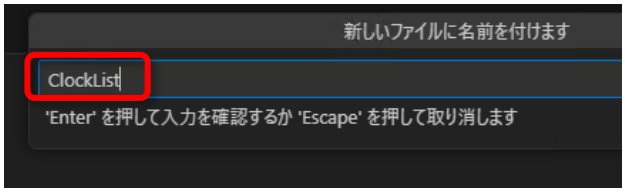
   するとフォルダー名を指定する欄が開くので、ここに "Pages" と入力し Enter キーを押します。

3. 

   すると Components フォルダーの下に Pages フォルダーが新規に作成されますので、この Pages フォルダーの右にある [新しいファイルの追加] ボタンをクリックします。

4. 

   するとテンプレート一覧がコマンドパレットに現れるので、[Razor コンポーネント] を選択します。

5. 

   するとファイル名を指定する欄が開くので、ここに "ClockList" と入力し Enter キーを押します。

> ※ 先にも述べましたが C# DevKit 拡張を含まない Visual Studio Code や dotnet CLI で開発の際は、単純に "ClockList.razor" というファイル名の空ファイルを、Components/Pages フォルダー内に touch コマンドなどで新規作成するだけでよいです。

1. ClockList.razor ファイルが Components/Pages フォルダー内に追加され、エディタ画面に開かれます。
   ClockList.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、Clock 型のフィールド "_clock" を定義します。
   - _clock フィールドには、ダミーデータとして適当な内容で Clock オブジェクトを new して割り当てます。
   - HTML で、_clock フィールドの内容 (表示名、現在時刻、タイムゾーン) をバインドします。

```
<div class="clock">
  <div class="name">
    <span class="value">@_clock.Name</span>
  </div>
  <div class="current-time">
    <span class="value">@_clock.GetCurrentTime().ToString("MM/dd HH:mm:ss")</span>
  </div>
  <div class="time-zone">
    <span class="caption">タイムゾーン</span>
    <span class="value">@_clock.TimeZoneId</span>
  </div>
</div>

@code {
    private Clock _clock = new()
    {
        Name = "札幌",
        TimeZoneId = "Asia/Tokyo"
    };
}
```

> ※TimeZoneId には、IANA tz データベースにおけるタイムゾーンの名前を指定します。
>
> 参考: *"List of tz database time zones - Wikipedia"*
>
> https://en.wikipedia.org/wiki/List_of_tz_database_time_zones

Step 6.　時計表示ページの実装 – 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、Clock オブジェクトを表示する新しい Razor コンポーネント "ClockList" が実装できました。
ですが、これだけではまだ、ClockList 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、ClockList 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 ClockList 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。

Razor コンポーネント (.razor ファイル) は、その .razor フォルダーが置かれている**サブフォルダー名の名前空間**、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、ClockList 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。そのためには先に説明した Components/_Imports.razor を再び使います。エディタで Components/_Imports.razor ファイルを開き、Components/Pages フォルダーに配置される .razor ファイルの名前空間 "BlazorWorldClock.Client.Components.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 BlazorWorldClock.Client.Components
@using BlazorWorldClock.Shared
@using BlazorWorldClock.Client.Components.Pages
```

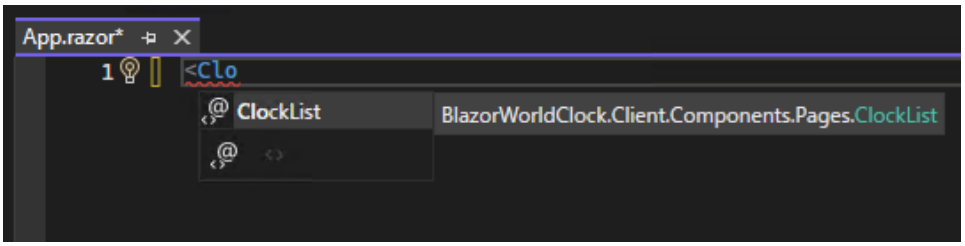
次は Components/App.razor です。

App.razor をエディタで開き、既存の内容をすべて削除します。

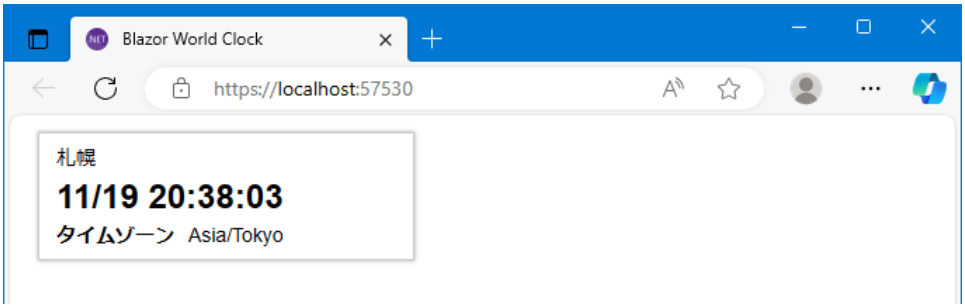
そして、下記のように ClockList コンポーネントのクラス名のタグを追記します。

```
<ClockList></ClockList>
```

なお、"Clo～" と、ある程度 Razor 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**。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 Clock オブジェクトがブラウザ上に表示されます (下図)。



> ※現在時刻は、今のところ、ページを表示した時点の時刻が表示されるのみです。
>
> 自動で現在時刻の表示が更新されるようにするのは、**本自習書の最後で実装**します。

Step 7. 時計表示ページ - リスト化 (繰り返し)

概要

ひとつの Clock オブジェクトを表示するところまではできました。

次は ClockList コンポーネントを改造し、複数の Clock 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 Clock 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. ClockList.razor を Visual Studio 内で開きます。
2. コードブロック中、Clock 型のフィールド_clock の定義をいったん削除します。
   代わりに、Clock 型の**配列**のフィールド_clock**s** に書き換えます。
3. _clocks フィールドに適当なダミーデータを割り当てます。

```
private Clock[] _clocks = [
    new() { Name = "札幌", TimeZoneId = "Asia/Tokyo" },
    new() { Name = "バンコク", TimeZoneId = "Asia/Bangkok" },
    new() { Name = "シアトル", TimeZoneId = "America/Los_Angeles" },
];
```

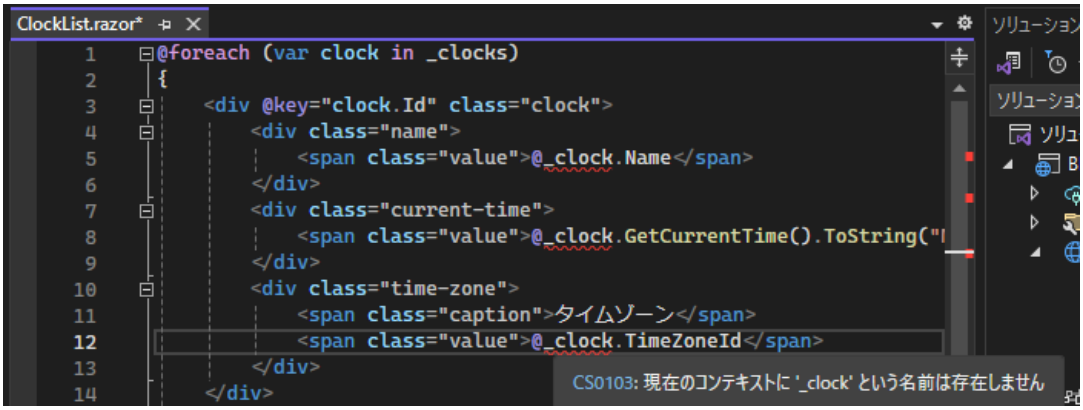
4. HTML全体を、@foreach (var clock in _clocks){ ～ } ブロックで囲みます。
5. foreach ループ直下のいちばん親の div 要素に、@key ディレクティブで、列挙するオブジェクトを一意に識別できるキーを指定します。
   列挙する個々の Clock オブジェクトを一意に識別できる因子は Id プロパティなので、これを指定します。

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

```
@foreach (var clock in _clocks)
{
  <div @key="clock.Id" class="clock">
    <div class="name">
      <span class="value">@clock.Name</span>
    </div>
    <div class="current-time">
      <span class="value">@clock.GetCurrentTime().ToString("MM/dd HH:mm:ss")</span>
    </div>
    <div class="time-zone">
      <span class="caption">タイムゾーン</span>
      <span class="value">@clock.TimeZoneId</span>
    </div>
  </div>
}
```

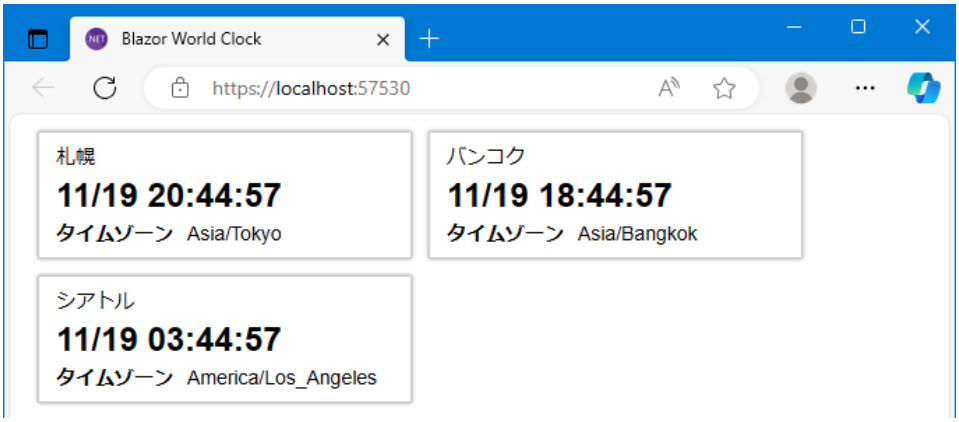
> ※ ループによるオブジェクトの列挙中に、@key ディレクティブを使って個々のオブジェクトを識別する必要については、下記、公式ドキュメントを参照ください。
>
> *" ASP.NET Core Blazor で要素、コンポーネント、モデルのリレーションシップを保持する"*
>
> https://learn.microsoft.com/ja-jp/aspnet/core/blazor/components/element-component-model-relationships

なお、上記編集中、Visual Studio や Visual Studio Code 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 Clock オブジェクトが表示されます (下図)。



Step 8. 時計情報の取得・登録を行うサービスの実装 – DI の使用

概要

引き続き、時計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、時計情報の蓄積および時計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。

サービスオブジェクトで時計情報を保持することにより、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 Razor コンポーネントが差し変わるようになっても、時計情報は失われずに Razor コンポーネント間で参照できるようになります。
また、さらにサーバー側の実装が進んで、時計情報をサーバー側で永続化・クライアント側と HTTP 通信で時計情報をやりとりするようになっても、そのために変更するのはこのサービスクラスのみで済むようになります。

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 Razor 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 Razor 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、時計情報を蓄え、時計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 **"ClockService" 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

> ※以降、ファイルの追加手順は、Visual Studio を使った場合の手順のみを掲載します。

1. Visual Studio のソリューションエクスプローラー上で BlazorWorldClock.Client プロジェクトを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。
   (または BlazorWorldClock.Client プロジェクトを選択した状態で、キーボードから Ctrl + Shift + A、あるいは Shift + F2 を打鍵します)
2. 「新しい項目の追加」ダイアログが現れるので、入力欄が "ClockService.cs" となるように入力して、Enter キーを押すか [追加(A)] ボタンをクリックします。
3. ClockService.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 Clock クラスのリストを持たせます。
   - この Clock クラスのリストに、とりあえず今はまだ、ダミーデータを初期設定しておきます。
   - 格納している Clock オブジェクトの集合を返す、"GetClocks()" メソッドを追加・実装します。
   最終的に ClockService.cs は下記のようになります。

```
using BlazorWorldClock.Shared;

namespace BlazorWorldClock.Client;

public class ClockService
{
  private readonly List<Clock> _clocks = [
    new() { Name = "札幌", TimeZoneId = "Asia/Tokyo" },
    new() { Name = "バンコク", TimeZoneId = "Asia/Bangkok" },
    new() { Name = "シアトル", TimeZoneId = "America/Los_Angeles" },
  ];

  public IEnumerable<Clock> GetClocks() => _clocks;
}
```

4. 次に、こうして実装した ClockService クラスを、Blazor の DI 機構に登録します。
   BlazorWorldClock.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<ClockService>();`」と追記して、DI 機構への ClockService クラスの登録処理を記載します。

```
using BlazorWorldClock.Client;
...
builder.Services.AddScoped(sp => new HttpClient { BaseAddress = ... });
builder.Services.AddScoped<ClockService>();
...
```

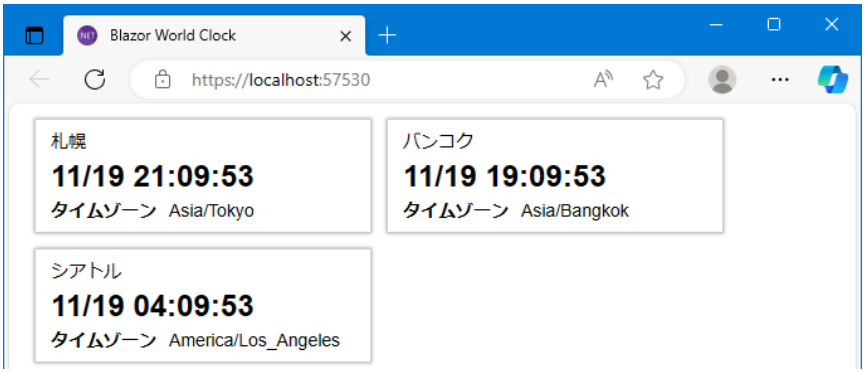
5. 次は、こうして DI 機構に登録された ClockService オブジェクトを ClockList で使用します。
   BlazorWorldClock.Client プロジェクトの ClockList.razor を Visual Studio で開き、行頭に「`@inject ClockService ClockService`」の行を追加します。
   この @inject ディレクティブにより、Razor コンポーネント ClockList のプロパティとして、**ClockService 型のプロパティ ClockService が追加**されます。この ClockService プロパティには、Blazor の DI 機構に登録された **ClockService 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 ClockService オブジェクトから、時計情報一覧を取得するように、ClockList.razor を変更します。
   まずは、ClockList.razor のコードブロック中、メンバーフィールド Clocks の型を `Clock[]`から `IEnumerable<Clock>` に変更し、初期設定していたダミーデータは空の列挙に置き換えます。
7. 次に、メンバーフィールド Clocks に、ClockService オブジェクトの GetClocks() メソッドで取得した時計情報一覧を設定する処理を足します。
   この処理は Razor 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 ClockService ClockService
...この部分は変更なし...
@code {
  private IEnumerable<Clock> _clocks = [];

  protected override void OnInitialized()
  {
    _clocks = this.ClockService.GetClocks();
  }
}
```

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 Clock 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 9. 非同期処理化

概要

さて、このまま時計情報の追加・編集へと邁進してもよいのですが、いずれ時計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るのですが、この時点で、時計情報サービス (ClockService) 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、ValueTask 型を使用できます**。

手順

1. ClockService.cs を Visual Studio で開き、GetClocks() 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、ValueTask<戻り値に返したい値の型>型に変更します。
   - メソッド名の末尾に "～Async" を追記します。
   - ダミーの時計情報の返し方は、ValueTask 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 GetClocks() メソッドは下記のようになります。

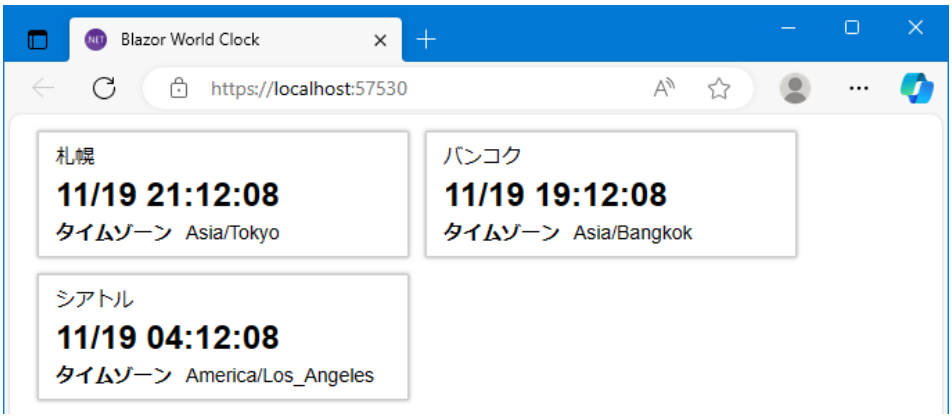
```
...
public async ValueTask<IEnumerable<Clock>> GetClocksAsync()
{
    return await ValueTask.FromResult(_clocks);
}
```

2. 次は、利用する側、ClockList 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 ClockService オブジェクトの GetClocksAsync() メソッドを await して呼び出し、結果に時計一覧が返ってきますからこれを Clocks 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()
{
  _clocks = await this.ClockService.GetClocksAsync();
}
```

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 Clock 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. 時計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

いよいよ時計の追加ができるようにしましょう。

時計表示 ClockList の HTML 末尾に、時計情報の入力欄 (input type=text 要素など) を設け、この入力欄と ClockList 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、時計情報サービス (ClockService) に時計情報の追加を行うようにします。
このために、時計情報サービス (ClockService) には、時計情報を追加するメソッドを実装します。

時計情報サービス (ClockService) を介して時計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、ブラウザ上の時計表示も更新されます。

手順

1. まずは時計情報サービス (ClockService) に時計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 AddClockAsync 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。
   BlazorWorldClock.Client プロジェクトの ClockService.cs を Visual Studio で開き、AddClockAsync 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 Clock 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、あえて非同期処理に仕立てます。
   AddClockAsync 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 ValueTask AddClockAsync(Clock clock)
   {
     _clocks.Add(clock);
     await ValueTask.CompletedTask;
   }
   ```

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。
   BlazorWorldClock.Client プロジェクトの ClockList.razor を Visual Studio で開き、コードブロック中に、新規時計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 "_newClock" 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。

   ```
   private Clock _newClock = new();
   ```

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_newClock を、時計情報サービス (ClockService) に先ほど実装した AddClockAsync() メソッドに引き渡して、時計情報の追加を行います。
   追加完了したらすかさず、再びの新規追加に備えて、メンバーフィールド_newClock に新しい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、AddClockAsync() 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。

```
private async Task OnOK()
{
    await this.ClockService.AddClockAsync(_newClock);
    _newClock = new();
}
```

4. あとは新規時計入力用のフォームの HTML をコーディングしましょう。
   入力用フォーム部分は、既存の HTML マークアップ部分にある @foreach ループで時計を一覧表示している直後から書き足していくことにします。
   レイアウトを整える目的から、まず、フォーム部分全体を構成する div 要素を、"clock" および "form" の CSS クラス名付きでマークアップします。続くマークアップ作業はこの div 要素の子要素として記述していきます、

```
@foreach (...)
{
    ...
}
<div class="clock form">
</div>
```

5. 次に、時計の表示名を入力する用に、レイアウト用の div 要素やラベル表示の span 要素などをマークアップして、その中に input type=text 要素を記述します。
   そしてこの input 要素に **@bind ディレクティブ**を記述し、フィールド変数 _newClock の Name プロパティとこの input 要素の入力内容とを結びつけ (バインド) ます。

```
<div class="clock form">
  <div class="name">
    <span class="caption">表示名</span>
    <span class="input-field">
      <input type="text" @bind="_newClock.Name" autofocus />
    </span>
  </div>
  ...
```

   このように **@bind ディレクティブ**を記述することにより、この input 要素には**フィールド変数 _newClock の Name プロパティが表示**され、かつ、この input 要素に**入力した内容はフィールド変数 _newClock の Name プロパティに書き戻され**るようになります。

6. 続く行には、タイムゾーンを一覧から選択する用に select 要素を記述します。
   select 要素においても同じ要領で、レイアウト用の div 要素やラベル表示の span 要素などをマークアップして、その中に select 要素をマークアップし、その select 要素に @bind ディレクティブを用いて、select 要素

で選択されるタイムゾーンの ID と、フィールド変数 _newClock の TimeZoneId プロパティとを結びつけ (バインド) ます。

```
...
<div class="time-zone">
  <span class="caption">タイムゾーン</span>
  <span class="input-field">
    <select @bind="_newClock.TimeZoneId">
    </select>
  </span>
</div>
...
```

7. さらには選択肢となるタイムゾーンの一覧を option 要素の集合として記述しなくてはなりません。
   さて、.NET プログラミングにおけるタイムゾーンの一覧は System.TimeZoneInfo.GetSystemTimeZones() 静的メソッドを呼び出すことで System.TimeZoneInfo オブジェクトの読み取り専用コレクションとして取得できます。
   そのようにして取得したタイムゾーンのコレクションを、C# の foreach 構文で option 要素の羅列へとマークアップしていきます。

```
<select @bind="this.NewClock.TimeZoneId">
  @foreach (var zone in TimeZoneInfo.GetSystemTimeZones())
  {
    <option value="@zone.Id">@zone</option>
  }
</select>
```

8. 最後に、OK ボタン (button 要素) を配置しましょう。"actions" の CSS クラス名を付与した div 要素をマークアップして、その中に OK ボタンの button 要素を配置します。OK ボタンのクリックイベントのハンドリングは、**@onclick** ディレクティブにハンドラメソッドを指定することで行います。このように、イベント名=”ハンドラ” の構文だと JavaScript によるイベントハンドリングとなるところを、**@イベント名="ハンドラ"** というように **@ 付きの構文**で記述すると、**そのイベントを C# のメソッドで捕捉**することができるようになります。

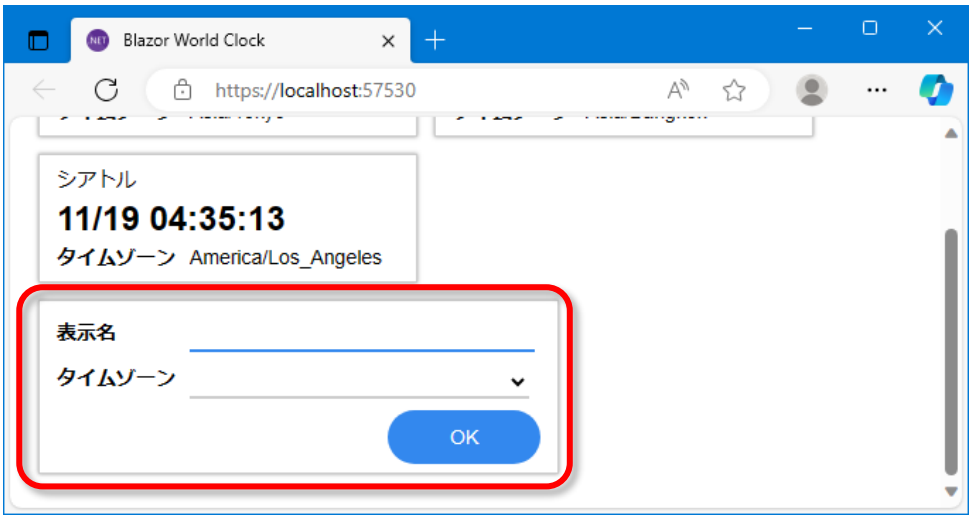
```
  ...
  <div class="actions">
    <button class="button" @onclick="OnOK">OK</button>
  </div>
</div>
```

9.

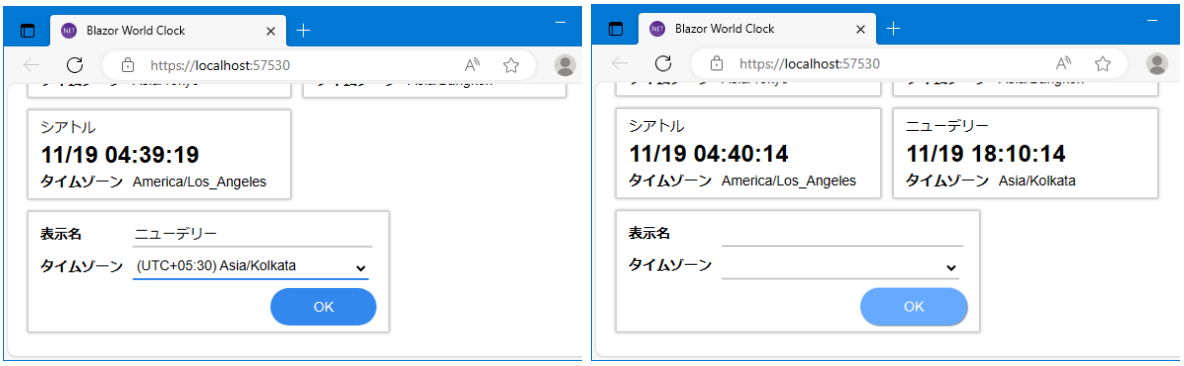
最終的に ClockList に追加される HTML は下記のようになります。

```
...
<div class="clock form">
  <div class="name">
    <span class="caption">表示名</span>
    <span class="input-field">
      <input type="text" @bind="_newClock.Name" autofocus />
    </span>
  </div>
  <div class="time-zone">
    <span class="caption">タイムゾーン</span>
    <span class="input-field">
      <select @bind="_newClock.TimeZoneId">
        @foreach (var zone in TimeZoneInfo.GetSystemTimeZones())
        {
          <option value="@zone.Id">@zone</option>
        }
      </select>
    </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、時計の一覧の下に、新規時計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時計として時計表示ページに追加されることが確認できます。



補足 - Visual Studio 2022 上での Blazor WebAssembly プログラムのデバッグ

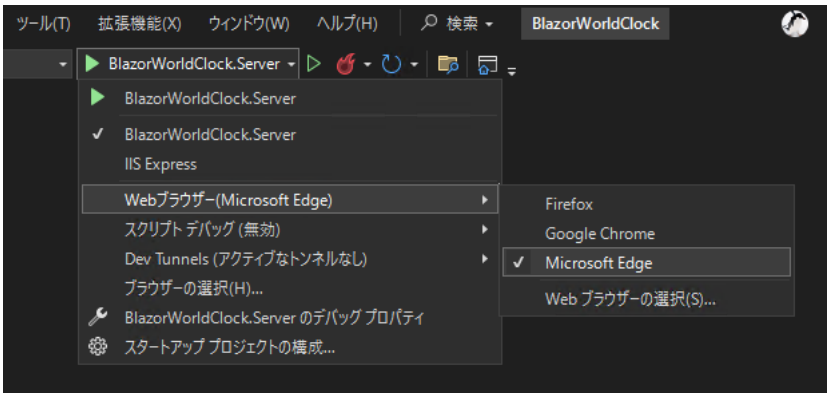
さてここで少し脱線して、Visual Studio 2022 上での Blazor WebAssembly プログラムのデバッグについて触れておきます。

Visual Studio 2022 では、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、変数のウォッチといったデバッガ機能が利用可能です。

但しブラウザ側の対応も必要で、Windows OS 上において 2023 年 11 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。

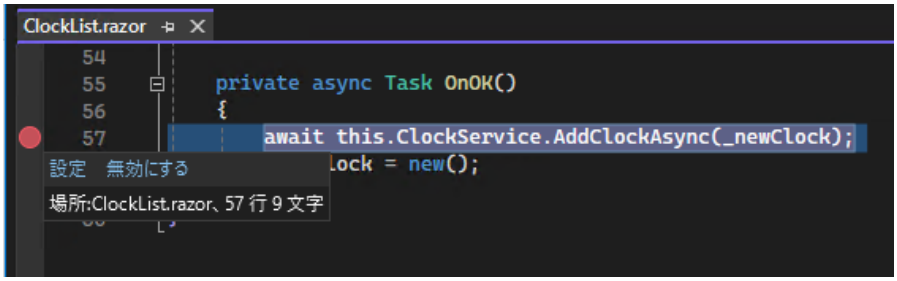
- Microsoft Edge
- Google Chrome

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)。



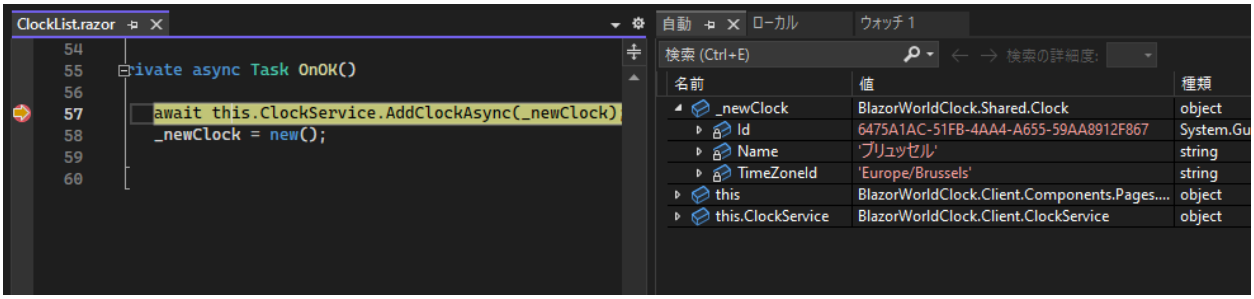
上記いずれかの Web ブラウザを起動するように Visual Studio 上で設定済みでしたら、試しに、このステップで実装した新規時計を追加する処理のところでブレークポイントの動作を試してみましょう。

まずは ClockList.razor ファイルを Visual Studio 内に開き、OnOK メソッド内で ClockService に新しい時計を追加し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 (上図)。

この状態で、キーボードの F5 キーを押して (ただし Ctrl キーは押さない!) 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時計表示ページが開きます。
時計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ます。且つ、各種変数の内容を見ることができるのがわかります（下図）。



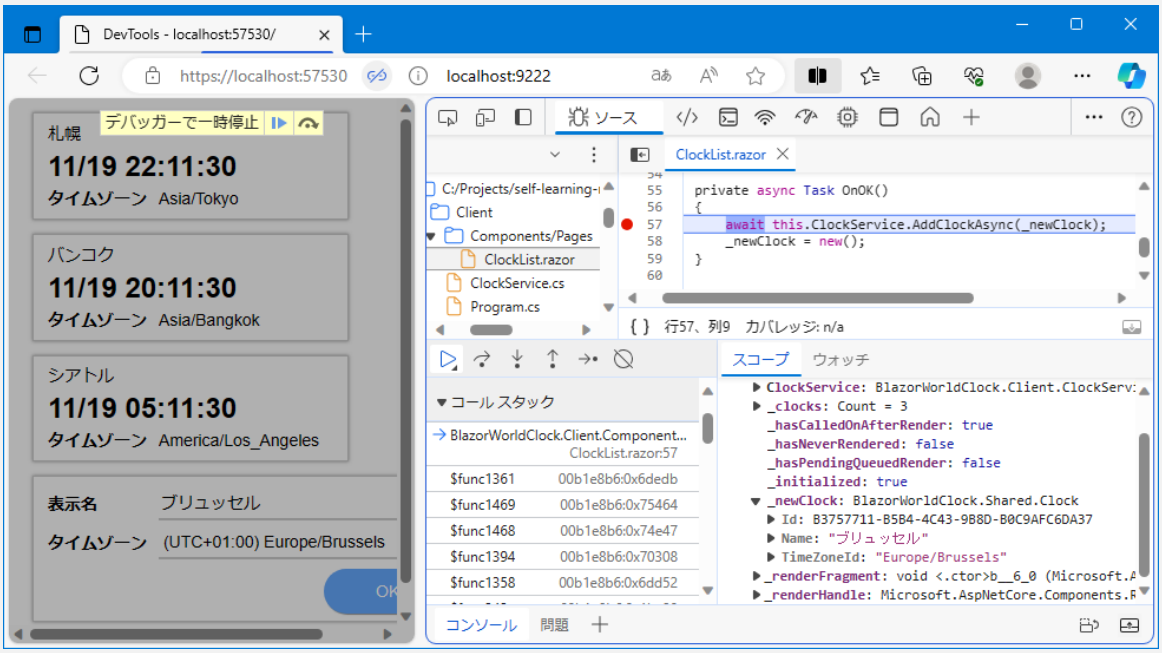
このように、Visual Studio 上で F5 実行するだけで、Blazor WebAssembly プログラムのデバッグが可能となっています。

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、ブレーク中に変数の書き換えができない、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 に限らず、Visual Studio Code、さらには**ブラウザの開発者ツール**からも、**Blazor WebAssembly プログラムのデバッグが可能**となっています (下図)。



詳細は下記公式ドキュメントサイトを参照ください。

https://learn.microsoft.com/ja-jp/aspnet/core/blazor/debug?tabs=visual-studio-code

Step 11. 入力内容のチェック (入力検証、バリデーション)

概要

ここまでで時計の新規追加ができるようになりました。

しかしながら、表示名が空欄、あるいはタイムゾーンが未選択のままでも [OK] ボタンを押せてしまいます。

ちなみに、タイムゾーンが未選択のまま [OK] を押すと、時計の追加は行なわれますが、そのようなタイムゾーン未選択の時計を表示するタイミングで、捕捉されない例外が発生してしまいます。

そこで、下記の入力チェックを実装してみます。

- 表示名が空欄でないこと
- 表示名は 20 文字まで
- タイムゾーンが選択されていること

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。

BlazorWorldClock アプリにおいては、"入力対象のオブジェクトのクラス" は Clock クラスです。すなわち、Clock 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

BlazorWorldClock.Shared プロジェクト内の Clock.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.ComponentModel.DataAnnotations;

namespace BlazorWorldClock.Shared;

public class Clock
{
  ...
```

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。
まずは時計の表示名を示す 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 Clock
{
    public Guid Id { get; set; } = Guid.NewGuid();

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

同じ要領で、タイムゾーンを示す TimeZoneId プロパティにも属性を付記します。
TimeZoneId プロパティについては、select および option 要素によるドロップダウンリストから選択する入力方式

なので、文字数上限を示す StringLength 属性は省略し、入力必須であることを示す Required 属性を付記するのみとします。

```
    [Required(ErrorMessage = "タイムゾーンを選択してください。")]
    public string TimeZoneId { get; set; }
    ...
```

以上で、Clock 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、時計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。
BlazorWorldClock.Client プロジェクトの ClockList.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 NewClock 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="clock form">
  <EditForm Model="_newClock" OnValidSubmit="OnOK">
    <div class="name">
      <span class="caption">表示名</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 コンポーネントと連動・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="Clock">
  <EditForm Model="_newClock" 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時計追加処理を呼び出すのをやめ、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。

そのため、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。

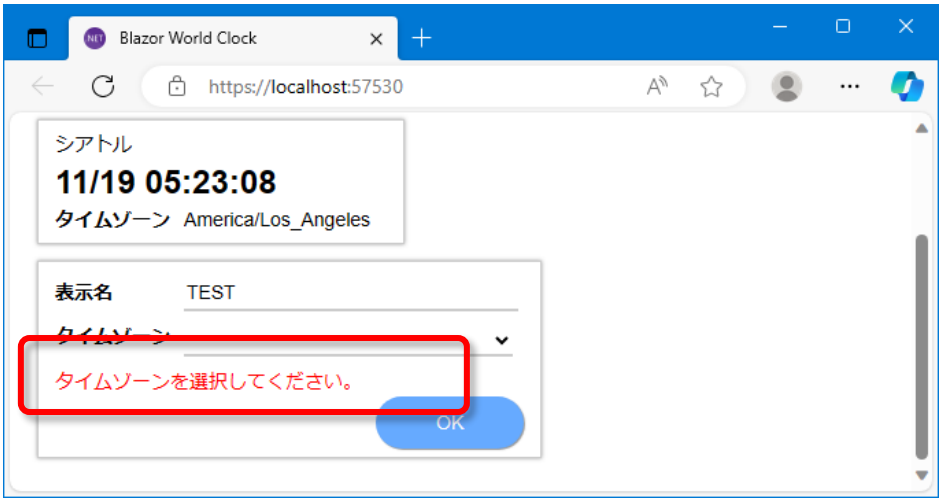
今回は、OK ボタンの上に配置することにします。

```
        ...
        </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、時計の新規追加欄について、**Clock 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 (下図)。



以上で、時計の新規追加フォームにおける入力内容のチェック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://learn.microsoft.com/ja-jp/aspnet/core/blazor/forms-and-input-components

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

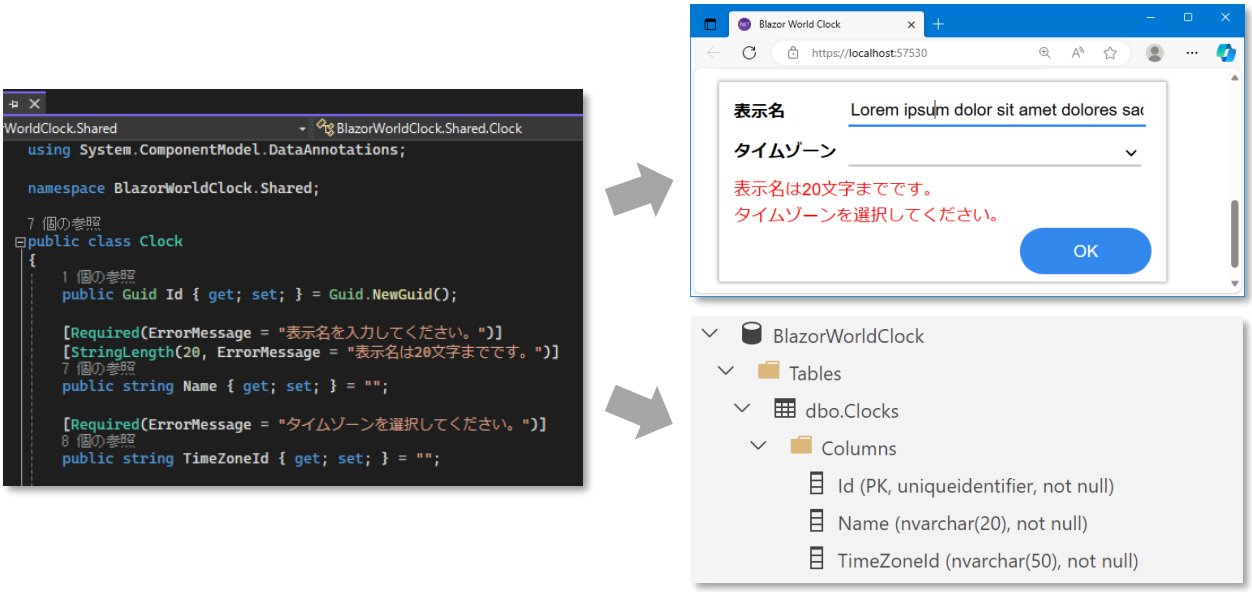
ASP.NET Core による Web アプリ開発においては、その要件として、リレーショナルデータベースに情報やデータを保存したり読み出したりすることがよくあります。

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 Microsoft SQL Server を採用した場合、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。

Step 12. 時計追加を独立した 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、時計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、時計の表示ページと追加のページを分け、これらページ間を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 Razor 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 Razor コンポーネントを割り当てるかは、各 Razor 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. BlazorWorldClock.Client プロジェクトの、いちばん根本の Razor コンポーネントである App.razor を開きます。現状では、時計表示ページである ClockList 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(App).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 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させます。

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 Razor 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、時計表示ページ ClockList を、ルート URL「/」に割り当てたいと思います。
   具体的には、BlazorWorldClock.Client プロジェクトの ClockList.razor を Visual Studio で開き、行頭に「`@page "/"`」の記述を追加することで、これを行ないます。

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、時計追加のユーザーインターフェースを独立した Razor コンポーネントに切り出し、「/addclock」の URL を割り当てましょう。
   新しい Razor 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ー上で BlazorWorldClock.Client プロジェクトの Components > Pages フォルダーを右クリックし、メニューから [追加(D)]-[新しい項目(W)...] をクリックします。

5. 「新しい項目の追加」ダイアログが現れるので、入力欄が "**AddClock.razor**" となるように入力してから [追加(A)] ボタンをクリックします。

6. AddClock.razor ファイルが Components/Pages フォルダー内に追加され、Visual Studio 内に開かれます。
   AddClock.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 "/addclock"`」を行頭に追記
   - 時計情報サービスを DI 経由で入手する「`@inject ClockService ClockService`」を次行に追記

7. さらに続けて、時計表示ページコンポーネント ClockList.razor から、以下の要素をカット (切り取り) してきて AddClock.razor に貼り付けます。
   - HTML パート中、foreach ループの下に追加した、 ～ 要素 (時計追加フォームのマークアップ)

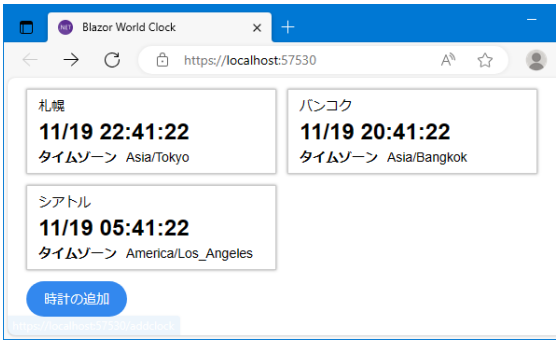
8. AddClock.razor に「`@code {～}`」コードブロックを作成し、ClockList.razor から以下のメンバーをカットして貼り付けます。
   - _newClock フィールド
   - OnOK メソッド

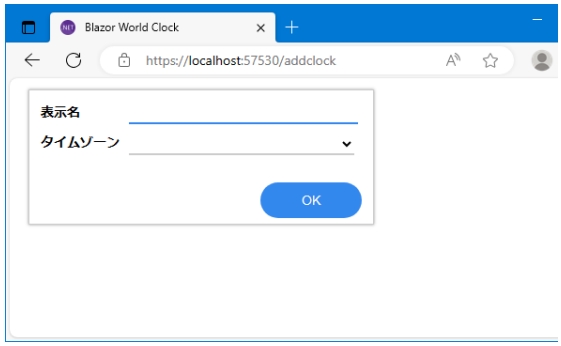
9. 以上で、ClockList.razor から、時計追加の機能に関する要素をひととおり、AddClock.razor へ移動することができました。
   最後に、時計表示ページ ("/") に、時計追加 ("/addclock") へのリンクを設けましょう。
   ClockList.razor を Visual Studio で開き、HTML パート部分の末尾に、"./addclock" へのリンクを下記のように追加します。

```
<div>
    <a class="button" href="./addclock">時計の追加</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。

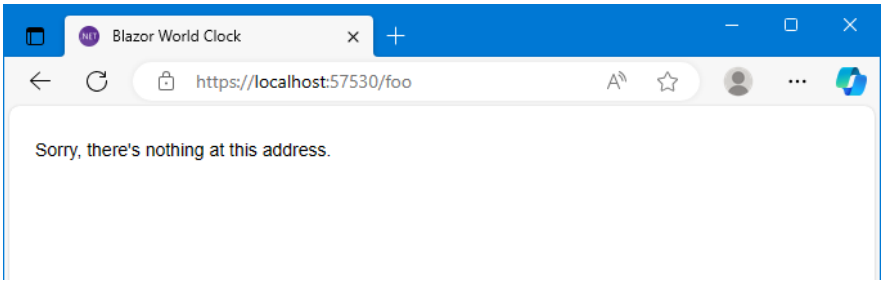
時計表示ページ ("/") に、「時計を追加」が増えています。これをクリックすると、URL が "/addclock" に遷移し、時計追加のユーザーインターフェースが表示されることが確認できます。





なお、時計追加のページで、表示名とタイムゾーンとを正しく入力して**「OK」ボタンをクリックしても、時計追加のページに居座ったまま**です。
これは、時計追加ページの OK ボタンクリック時に、**時計表示ページ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」で時計表示ページに戻ってみると、OK ボタンクリックして追加した時計情報が、たしかに時計表示ページ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（下図）。



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、時計追加ページで OK ボタンを押したら、時計表示ページの URL に戻るように実装していきましょう。
ついでに、時計追加ページにキャンセルボタンも実装しておきます。

Razor 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、時計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。
   BlazorWorldClock.Client プロジェクトの AddClock.razor を Visual Studio で開き、ClockService 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 "/addclock"
@inject ClockService ClockService
@inject NavigationManager NavigationManager
...
```

2. 次に、AddClock.razor のコードブロック中 OnOK メソッドの最後、時計情報サービスに新規時計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 "./" に遷移するように変更します。

```
private async Task OnOK()
{
    await this.ClockService.AddClockAsync(_newClock);
    this.NavigationManager.NavigateTo("./"); // ← _newClock = new() を置換
}
```

3. 以上で、時計追加ページで OK ボタンをクリックすると、無事時計を追加できたら、そのまま時計表示ページ ("/") に遷移するようになります。
   仕上げに、時計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。AddClock.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">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。

時計表示ページから「時計を追加」をクリックして時計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、時計表示ページに遷移して、かつ、追加した時計が時計表示ページに表示されていることを確認してください。

また、時計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、時計の追加を行わずに時計表示ページに戻れることを確認してください。

Step 14. 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、時計の "追加" までできるようになりました。次は時計の "編集" に取り掛かります。

ですがその前に、時計情報の入力ユーザーインターフェースを、**"時計情報フォーム" 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、時計の "追加" ページと、(このあと作成に着手する) 時計の "編集" ページの、**それぞれのコンポーネントに "時計情報フォーム" コンポーネントを埋め込む**ことで、コードの共有化が図れます。

"時計情報フォーム" コンポーネントでは、(時計情報の追加、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時計情報オブジェクトを受け取る必要があります。また、"時計情報フォーム" 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、Razor 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。
Razor 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。

さてこのような "時計情報フォーム" コンポーネントですが、ファイル名はその用途にあわせて "ClockForm.razor" としたいと思います。また、この ClockForm コンポーネントは @page ディレクティブを持たず、特定の URL にルーティングされるような "ページ" コンポーネントではないこと、むしろ "ページ" コンポーネント内で参照される共有部品としての扱いであることから、その保存先フォルダーは、Components/Pages ではなく、新たに Components/Shared というフォルダーを設けて、その中に配置したいと思います。

手順

1. まずは新しい Razor 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ー上で BlazorWorldClock.Client プロジェクトの Components フォルダーを右クリックし、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、入力欄が "**Shared/ClockForm.razor**" となるように入力してから [追加(A)] ボタンをクリックします。

3. Components フォルダー内に Shared フォルダーが新規作成され、さらにその中に ClockForm.razor ファイルが新規に追加され、Visual Studio のコードエディタ内に開かれます。

4. ClockForm.razor ファイル内に、いったん AddClock.razor の内容すべてをコピーして貼り付けます。

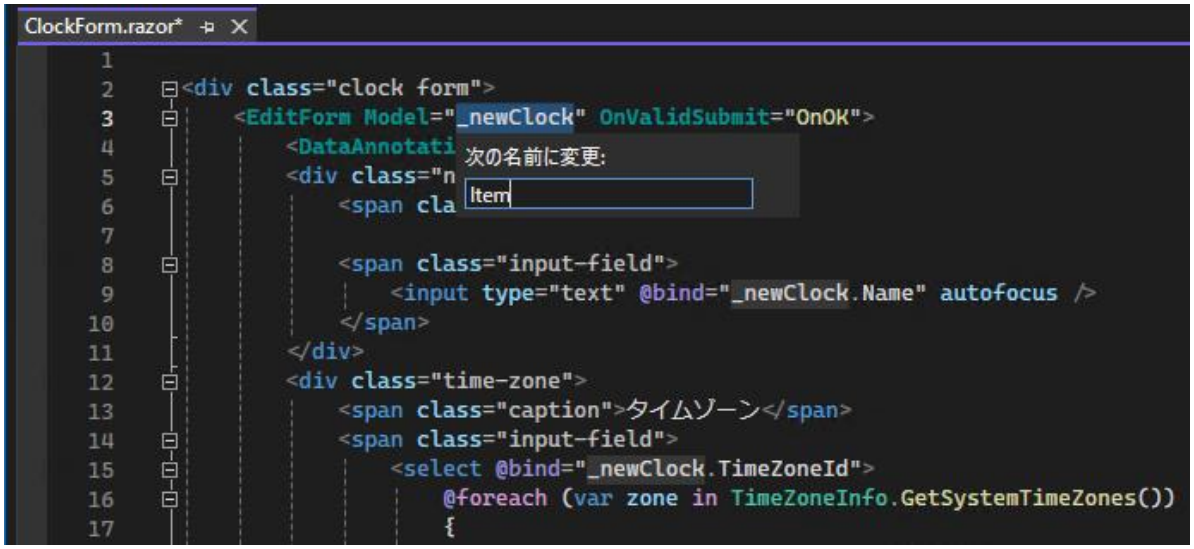
5. ただし ClockForm.razor はあくまでも子コンポーネントとして使うので、URL ルーティングへの割り当てである「@page ～」ディレクティブの行は削除します。

6. また、時計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、ClockForm.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。

7. 次に、編集対象の時計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_newClock を、[Parameter] 属性付きの public プロパティに実装を書き換えます。

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

8. ところで、このプロパティの名前 "**_newClock**" は、プライベートフィールドの命名規則である「アンダースコアから始まるキャメルケース」となっており、パブリックプロパティの命名規則である「パスカルケース」となっていません。さらに今後この ClockForm コンポーネントが時計情報の **"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全ての必要箇所で、プロパティ名が_newClock から Item に変更されます (上図)。

9. 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、イベントコールバック関数型の public プロパティを、ClockForm.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、編集対象の時計情報オブジェクトを引数にひとつ取るイベント関数として、Blazor プログラミング用途に用意されている EventCallback<T> 構造体を採用し、EventCallback<Clock> とします。

```
@code {
    [Parameter]
    Public Clock Item { get; set; } = new();

    [Parameter]
    public EventCallback<Clock> OnClickOK { get; set; }
```

10. ClockForm.razor の仕上げとして、OnOK メソッド内の末尾にて、入力チェック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。

```
private async Task OnOK()
{
    await this.OnClickOK.InvokeAsync(this.Item);
}
```

11. こうして追加した ClockForm 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きます。エディタで Components/_Imports.razor ファイルを開き、Components/Shared フォルダーに配置される .razor ファイルの名前空間 "BlazorWorldClock.Client.Components.Shared" を開いておく @using 節を追加してください。

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

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

```
<ClockForm OnClickOK="this.OnOK" />
```

13. 続けて、時計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した OnOK メソッドの実装を修正していきます。
まず動作としては、時計情報フォームコンポーネントから OnClickOK イベントコールバックが発生するとき、その引数として、同フォームの入力を反映した Clock オブジェクトが渡されます。
そこで OnOK メソッドでは、その引数に渡される Clock オブジェクトを、ClockService へ引き渡すように実装を書き換えます。その結果、フィールド変数 _newClock は不要になるので削除します。
最終的に AddClock.razor 内のコードブロックは下記のとおりとなります。

```
@code
{
    // private Clock _newClock = new(); ← これは削除

    private async Task OnOK(Clock clock)
    {
        await this.ClockService.AddClockAsync(clock);
        this.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、正常に時計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. 時計情報の編集 - ルーティング引数

概要

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

Razor 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 Razor コンポーネントの *identifier* という**同じ名前**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、編集ページコンポーネントの URL ルーティング定義は "/editclock/{**ClockId**}" とし、同コンポーネントに **ClockId** 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、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、"/editclock/{ClockId**: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://learn.microsoft.com/ja-jp/aspnet/core/blazor/fundamentals/routing#route-constraints-3

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

まずは時計表示ページから、編集ページに遷移できるように実装していきます。
BlazorWorldClock.Client プロジェクトの ClockList.razor を Visual Studio で開きます。
そして HTML パートの時計情報 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 clock in _clocks)
{
      ...
      <span class="value">@clock.TimeZoneId</span>
    </div>
    <div class="actions">
      <button class="button edit-button" @onclick="() => this.OnClickEdit(clock)">
        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 Clock 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時計情報オブジェクトを特定できます。

次に準備として、この時計表示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、時計情報編集ページの 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 ClockList.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 ClockService ClockService
@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。
引数に編集対象の時計情報オブジェクトが渡されて呼び出されますから、その編集対象時計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。

```
private void OnClickEdit(Clock clock)
{
    this.NavigationManager.NavigateTo($"./editclock/{clock.Id}");
}
```

時計表示ページの変更はこれで完了です。

続けて、時計情報編集ページ (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ー上で BlazorWorldClock.Client プロジェクトの Components/Pages フォルダーを右クリックし、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、入力欄が "**EditClock.razor**" となるように入力して [追加(A)] ボタンをクリックします。

EditClock.razor ファイルが Pages フォルダー内に追加され、Visual Studio 内に開かれます。
EditClock.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 "/editclock/{ClockId:guid}"」の行を追加します。

```
@page "/editclock/{ClockId:guid}"
```

そして EditClock.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 "{ClockId: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。

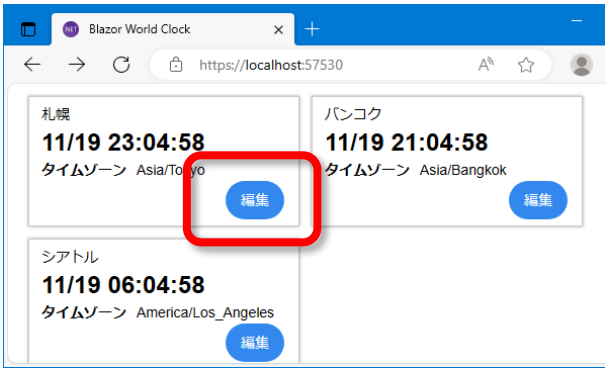
```
@code {
    [Parameter]
    public Guid ClockId { get; set; }
}
```

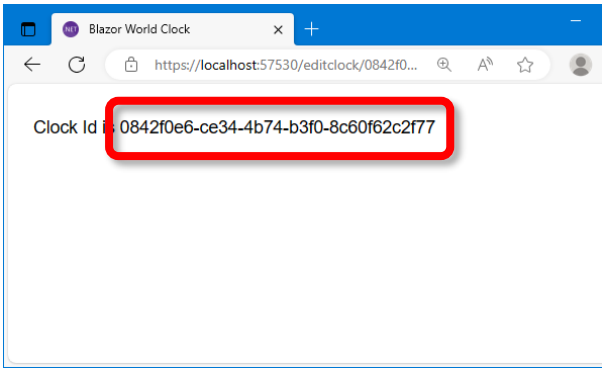
最後に、動作確認の目的で、EditClock.razor の HTML パートに、URL 引数を受け取った ClockId プロパティ値を表示するだけのマークアップを記述します。

```
<p>Clock Id is @this.ClockId</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。

時計表示ページの個々の時計情報に「編集」ボタンが追加されており、これをクリックすると時計情報の Id プロパティが表示されることを確認してください。





Step 16. 時計情報編集ページの実装

概要

それではいよいよ、時計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、時計情報サービスに、

- 指定された Id の時計情報の取得
- 及び、指定 Id の時計の更新

を行う機能を追加します。

そのうえで、時計情報の編集ページコンポーネントでは、先に作成した時計情報フォームコンポーネント (ClockForm) 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、編集ページコンポーネントでは追加のときと異なり、編集対象の時計情報の取得が非同期処理となります。すなわち、編集対象の時計情報オブジェクトが取得完了するまでの間、描画を抑止する制御が加わってきます。

手順

1. まずは時計情報サービス (ClockService.cs) に、GetClockAsync() メソッドと、UpdateClockAsync() メソッドとを、下記のように追加しておきます。

```
public async ValueTask<Clock?> GetClockAsync(Guid id)
{
   return await ValueTask.FromResult(_clocks.FirstOrDefault(c => c.Id == id));
}

public async ValueTask UpdateClockAsync(Clock clock)
{
    var index = _clocks.FindIndex(c => c.Id == clock.Id);
    if (index >= 0) _clocks[index] = clock;
    await ValueTask.CompletedTask;
}
```

2. 次に、時計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。
   EditClock.razor を Visual Studio で開き、時計情報サービスの DI 機構経由での注入を記述します。
   また、OK ボタンクリック時に時計表示ページへ遷移するために、NavigationManager も DI で注入します。

```
@page "/editclock/{ClockId:guid}"
@inject ClockService ClockService
@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。
   - 編集対象の時計情報オブジェクト
   - 編集対象の時計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、時計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、時計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 ClockId { get; set; }

    private Clock? _item;

    private bool _initialized = false;
```

4. 続けて、この時計情報編集ページコンポーネント EditClock 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 (及び、async キーワードを追加)、この中で時計情報サービスから編集対象の時計オブジェクトを取得します。
   取得できたら、その時計オブジェクトの複製を作って編集対象を指すフィールド変数 (_item) に代入します。
   最後に "初期化完了" のフラグのフィールド変数 (_initialized) を true にします。

```
protected override async Task OnInitializedAsync()
{
    var clock = await this.ClockService.GetClockAsync(this.ClockId);
    if (clock != null)
    {
        _item = new()
            {
                Id = clock.Id,
                Name = clock.Name,
                TimeZoneId = clock.TimeZoneId
            };
    }
    _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、(ClockForm 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、時計情報サービスに対し編集後の時計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。

```
private async Task OnClickOK(Clock item)
{
    await this.ClockService.UpdateClockAsync(item);
    this.NavigationManager.NavigateTo("./");
}
```

6. 最後に HTML を記述しましょう。

   まず、既存の HTML マークアップ (Clock Id is @this.ClockId) は削除しておきます。

   続けて、ページ初期化時、編集対象の時計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 フィールド変数に基づく @if ブロックを形成します。

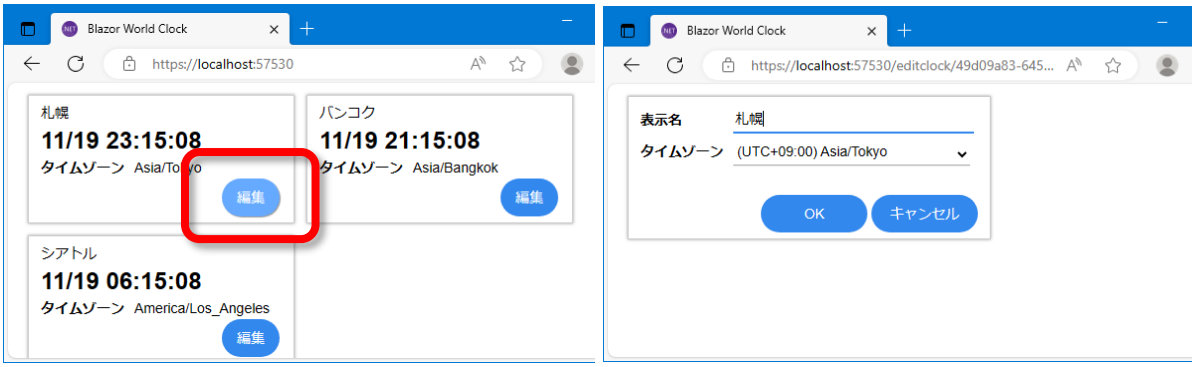
   さらに指定された ID 値で該当する時計情報が見つからなかった場合に備えた @if ブロックを重ねます。

   最後に、編集対象の時計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、時計情報入力フォームコンポーネント ClockForm をマークアップします。

```
@if (_initialized)
{
    @if (_item != null)
    {
        <ClockForm Item="_item" OnClickOK="this.OnClickOK" />
    }
    else
    {
        <p class="error-message">時計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。

時計表示ページから「編集」ボタンをクリックすると、その時計情報が編集できるページに遷移すること、かつ、実際に時計情報を変更して OK をクリックすると、変更が時計表示ページ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、表示・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 Razor 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** を任意の箇所で描画します。
この **Body プロパティを記述した部分に、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 Razor コンポーネントのクラスをパラメータに指定**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 プロパティを描画した箇所に、そのページコンポーネントが描画されるようになります。

なお、慣例的に、共通レイアウト用のコンポーネントは、Components フォルダーの下、"Layout" サブフォルダーに保存します。

手順

1. まずは共通レイアウトを実装する Razor 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。Visual Studio のソリューションエクスプローラー上で Components フォルダーを右クリックし、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、入力欄が "**Layout/MainLayout.razor**" となるように入力してから [追加(A)] ボタンをクリックします。

3. MainLayout.razor ファイルが Layout フォルダー内に追加され、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 World Clock</h1>
</header>
@Body
```

4. こうして作成した共通レイアウトコンポーネント "MainLayout" の型の名前空間は、"BlazorWorldClock.Client.Layout" になります。例によって、この名前空間を省略して MainLayout コンポーネントを参照できるよう、_Imports.razor でこの名前空間を @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。

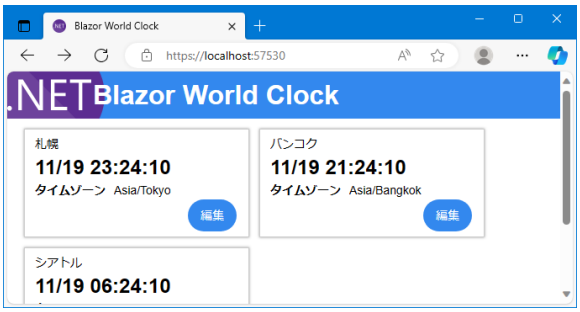
```
...前半変更なし...
@using BlazorWorldClock.Client.Components.Shared
@using BlazorWorldClock.Client.Components.Layout
```

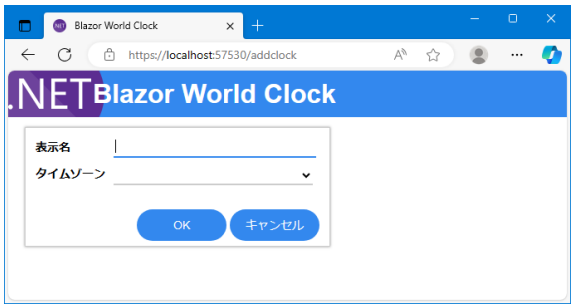
5. 以上で共通レイアウトコンポーネントは実装できました。
続けて各ページコンポーネントでこの MainLayout をレイアウトコンポーネントとして使用するよう、App.razor 中の Router コンポーネントに対して指定します。
Visual Studio で App.razor を開き、RouteView コンポーネントの **"DefaultLayout" パラメータ**に、共通レイアウトとして使うコンポーネントの型 (今回は MainLayout クラス) を指定するよう、追記します。

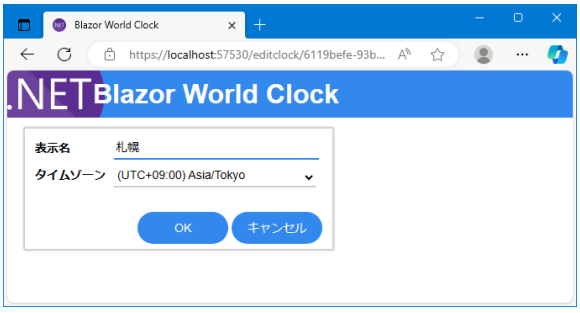
```
<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="typeof(MainLayout)" />
    </Found>
    <NotFound>
        <p>Sorry, there's nothing at this address.</p>
    </NotFound>
</Router>
```

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。

時計表示、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。





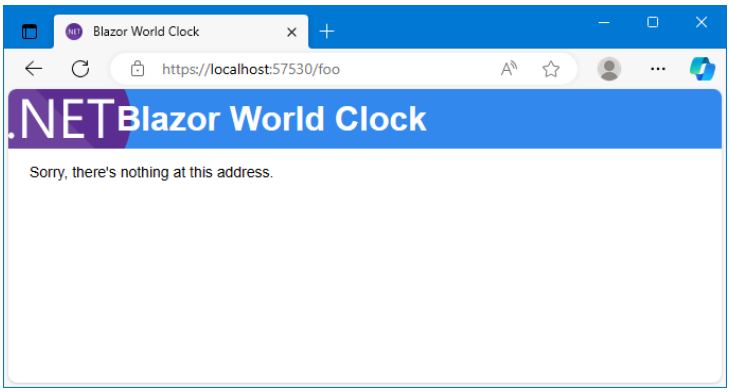


なお、ここまでの実装だけですと、URL に該当するコンポーネントが見つからない場合のエラー表示には、この共通レイアウトは適用されません。
それで構わなければ以上でよいのですが、URL に該当するコンポーネントが見つからない場合の表示にも共通レイアウトを適用したい場合は、App.razor 中の Router コンポーネントに以下の変更を行なうことで達成できます。

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(App).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

概要

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。
続いては、いよいよサーバー側を実装します。
すなわち、時計情報の保存、および、クライアント側 (Web ブラウザ上で稼働している Blazor WebAssembly アプリケーション) と時計情報を送受信する、そのようなサーバー側で稼働する C#プログラムを実装します。

クライアント-サーバー間の通信形態は、本自習書執筆時点で Web アプリにおけるクライアントサーバー間通信技法として割と一般的と思われる、標準な HTTP 要求で JSON 形式の本文を送受信する **"HTTP REST API" 方式**とします。

> ※Blazor によるクライアント-サーバー間の通信技法としては、本自習書で採用の HTTP REST 方式の他にも、WebSocket 等を基盤とした **SignalR** による双方向通信の方式や、通信インターフェースの型定義およびバイナリ形式による高効率なデータ転送が特徴的な **gRPC** 方式 (正確には gRPC-Web)、グラフ構造に対して高効率にクエリを行える **GraphQL** など、任意の方式を使えます。

一般的に、サーバー側の機能を HTTP REST 方式等による通信によって呼び出せるようにした機構を **"Web API"** と呼称したりします。本書でもこの "Web API" の呼称を使用します。

今回実装する Blazor World Clock のサーバー側 Web API の仕様として、HTTP 要求のメソッドおよび URL パスと対応する機能は、以下のとおりとします。

- HTTP POST /api/clocks ... 要求本文の JSON で指定した時計情報を追加
- HTTP GET /api/clocks ... すべての時計情報を JSON 形式で返却
- HTTP GET /api/clocks/*{id}* ... 指定の Id の時計情報を JSON 形式で返却
- HTTP PUT /api/clocks/*{id}* ... 指定の Id の時計情報を、要求本文の JSON の内容で更新
- HTTP DELETE /api/clocks/*{id}* ... 指定の Id の時計情報を削除

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、サーバー側の実装について詳細は一部割愛いたします。
詳細については本自習書に同梱のソースコードを併せて参照ください。

手順

1. 時計情報を JSON 形式でテキストファイルに保存・復元する機能を、**ClockStorage** というクラスに実装して実現することにします。
   サーバー側の機能ですから、Visual Studio のソリューションエクスプローラー上で BlazorWorldClock.Server 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、入力欄が "**ClockStorage.cs**" となるように入力してから [追加(A)] ボタンをクリックします。

3. BlazorWorldClock.Server プロジェクトに ClockStorage.cs が追加され、Visual Studio で開かれます。
   ClockStorage クラスの実装として、
   - Clock オブジェクトのリストをプライベートプロパティとして持ち、
   - コンストラクタの引数に指定された保存先ファイルに Clock オブジェクトのリストを JSON 形式で保存、
   - 及び、コンストラクタのタイミングでその JSON 形式ファイルを読み取り Clock オブジェクトのリストを復元

   とするよう、実装します。
   その上で、時計情報の追加変更を行なう public メソッドを公開します。
   前述のとおり ClockStorage クラスの実装詳細は本書面では割愛します。
   実際のコードは、本自習書に同梱のソースコード、又は下記 GitHub リポジトリのリンク先から入手ください。
   https://github.com/jsakamoto/self-learning-materials-for-blazor-jp/blob/v.8.0.0/Server/ClockStorage.cs
   大枠は以下の様になります。

```
using System.Text.Json;
using BlazorWorldClock.Shared;

namespace BlazorWorldClock.Server;

public class ClockStorage
{
    ...
    private readonly List<Clock> _clocks = [];

    public ClockStorage(string storagePath)
    {
      ...
    }

    public void AddClock(Clock clock)
    {
        ...
    }

    public List<Clock> GetClocks()
    {
        ...
    }

    public Clock GetClock(Guid id)
    {
        ...
    }

    public void UpdateClock(Guid id, Clock clock)
    {
        ...
    }

    public Clock DeleteClock(Guid id)
    {
        ...
    }
    ...
}
```

   なお、サーバー側実装でも、クライアント側実装と同じ、BlazorWorldClock.Shared プロジェクトに収録の**Clock クラスを共通して使えていること**に注目してください。

Blazor ではこのように、データ転送系のオブジェクト型 (DTO) の実装を、クライアント側・サーバー側で別々に持つことなく共有プロジェクト**一箇所で実装**できます。
そのため、データ転送オブジェクトの**実装の二度手間がない**ことは勿論、クライアント-サーバー間で**データ転送オブジェクト型の定義に齟齬が生じることがあり得ません**。

さらには、データ転送オブジェクトは**正真正銘の .NET オブジェクト**がシリアル化・逆シリアル化されているので、データ転送オブジェクトの**計算プロパティ**や**各種メソッド**が、**クライアント側・サーバー側のいずれであっても同じく参照できる・実行できる**のも注目ポイントのひとつです。

4. 次は、こうして実装した ClockStorage クラスの各メソッドを、HTTP 要求のメソッドおよび URL パスごとに割り当てていきます。
   まずは ClockStorage オブジェクトを、Web アプリケーション実行時にインスタンス化しましょう。
   Visual Studio で BlazorWorldClock.Server プロジェクトの Program.cs を開きます。
   そして、この後の実装を簡略化するために以下の 3 つの using 節を Program.cs 先頭に追記しておきます。

```
using BlazorWorldClock.Server;
using BlazorWorldClock.Shared;
using static System.Environment.SpecialFolder;
```

5. 次に、時計情報を JSON 形式で保存するファイルのフルパスを決定し、そのフルパスを引数に ClockStorage クラスのコンストラクタを呼び出して ClockStorage オブジェクトをインスタンス化します。
   保存先ファイルのフルパスは、
   - Windows OS では C:¥Users¥{ユーザー名}¥AppData¥Roaming¥Blazor World Clock¥clocks.json
   - Linux では /home/{ユーザー名}/.config/Blazor World Clock/clocks.json
   とします。

```
...
app.UseStaticFiles();

var storagePath = Path.Combine(Environment.GetFolderPath(ApplicationData),
"Blazor World Clock", "clocks.json");
var clockStorage = new ClockStorage(storagePath);

app.UseRouting();
...
```

6. ClockStorage オブジェクトをインスタンス化したら、その各メソッドを、HTTP メソッドと URL パスごとに割り当てしていきます。

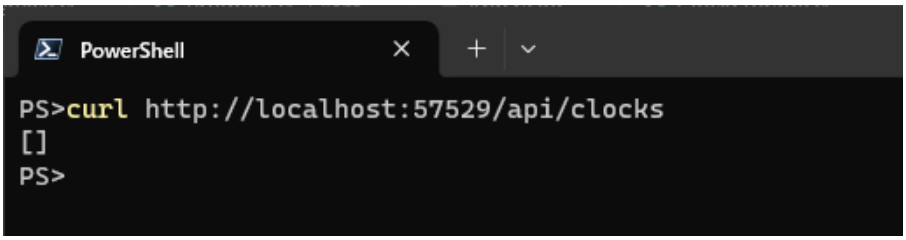
```
...
var clockStorage = new ClockStorage(storagePath);

app.MapPost("/api/clocks", (Clock clock) => clockStorage.AddClock(clock));
app.MapGet("/api/clocks", () => clockStorage.GetClocks());
app.MapGet("/api/clocks/{id}", (Guid id) => clockStorage.GetClock(id));
app.MapPut("/api/clocks/{id}", (Guid id, Clock clock) =>
                                        clockStorage.UpdateClock(id, clock));
app.MapDelete("/api/clocks/{id}", (Guid id) => clockStorage.DeleteClock(id));

app.UseRouting();
...
```

以上でサーバー側実装は完了です。
ビルドおよび実行して、エラーが発生しないことを確認ください。

また、ターミナル (コマンドプロンプト) を開いて curl コマンドを使って「http://localhost:57529/api/clocks」に対して HTTP GET 要求を送信してみると、いま実装した Web API によって、とりあえず空集合が JSON 形式で返ってくることが確認できるはずです (下図)。



> ※もちろん、curl コマンドのみならず、Postman であるとか、あるいは Visual Studio をお使いの場合は Visual Studio のエンドポイントエクスプローラーといった、HTTP REST 形式の Web API にアクセスするツールや機能を使って今実装した Web API をいろいろ試すことができます。

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、クライアント側の時計情報サービス (ClockService) 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 オブジェクトは、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、Razor コンポーネントではなく、時計情報サービス ClockService クラスです。
これまでの例では、Razor 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 ClockService 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 Razor 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 やその拡張メソッドなどをすぐに使えるようにするため、BlazorWorldClock.Client プロジェクトの ClockService.cs を Visual Studio で開き、ClockService.cs 先頭に以下の名前空間の使用を追加します。

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

続けて、ダミーデータ実装の _clocks 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、クラス名に続けてプライマリコンストラクタを記述し、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 ClockService(HttpClient httpClient)
{
    ...
```

続けて、時計情報サービスの実装を、ダミーデータ実装の _clocks プロパティを読み書きしていたものから、プライマリコンストラクタの引数で受け取った 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、時計情報サービスにはまだ備えていなかった、時計情報の削除のメソッド (DeleteClockAsync) も追加実装してしまいましょう。

```
public async ValueTask<IEnumerable<Clock>> GetClocksAsync()
{
    return await httpClient.GetFromJsonAsync<Clock[]>("api/clocks") ?? [];
}

public async ValueTask AddClockAsync(Clock clock)
{
    await httpClient.PostAsJsonAsync("api/clocks", clock);
}

public async ValueTask<Clock?> GetClockAsync(Guid id)
{
    return await httpClient.GetFromJsonAsync<Clock>($"api/clocks/{id}");
}

public async ValueTask UpdateClockAsync(Clock clock)
{
    await httpClient.PutAsJsonAsync($"api/clocks/{clock.Id}", clock);
}

public async ValueTask DeleteClockAsync(Guid id)
{
    await httpClient.DeleteAsync($"api/clocks/{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。

時計情報の永続化先がサーバー側となったので、初回は時計表示ページが空となっています。
以降、時計情報の追加や編集が機能していること、また、いまや時計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. 時計の削除機能を実装 - 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、時計の削除機能の実装に着手します。

モデル更新的には、既に実装してきた Web API (HTTP DELETE /api/clocks/{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://learn.microsoft.com/ja-jp/aspnet/core/blazor/javascript-interoperability

手順

1. BlazorWorldClock.Client プロジェクトの ClockForm.razor を Visual Studio で開きます。
   削除ボタンが押された時の confirm JavaScript 関数呼び出しを行うために、Blazor の JavaScript 相互運用サービス (IJSRuntime) を使用します。なので、JavaScript 相互運用サービス (IJSRuntime) 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 相互運用サービスインスタンスを受け取るフィールド変数名は "JSRuntime" としておきましょう。

```
@inject IJSRuntime JSRuntime
```

2. 次に ClockForm.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 EventCallback 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。

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

4. 残りは HTML マークアップです。

   ClockForm.razor の HTML パートにて、「削除」ボタンの button 要素を実装します。

   ただし、追加ページ (AddClock.razor) から使われる場合は、削除ボタンは非表示としておきたいところです。

   そこで、OnClickDelete イベントコールバックの **HasDelegate プロパティが true である** (すなわち OnClickDelete プロパティにコールバック関数が割り当てられている) 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 (this.OnClickDelete.HasDelegate)
  {
    <button class="button delete-button"
            type="button" @onclick="this.OnDelete">削除</button>
  }
  <button class="button">OK</button>
  <a class="button" href="/">キャンセル</a>
```

5. あとは、時計情報編集ページコンポーネント EditClock にて、削除ボタンが押された時の振る舞いを定義して、ClockForm の OnClickDelete プロパティにバインドしましょう。

   BlazorWorldClock.Client プロジェクトの EditClock.razor を Visual Studio で開き、時計情報サービスの削除処理を呼び出し後、URL "./" に遷移するメソッド、OnClickDelete メソッドを追加します。

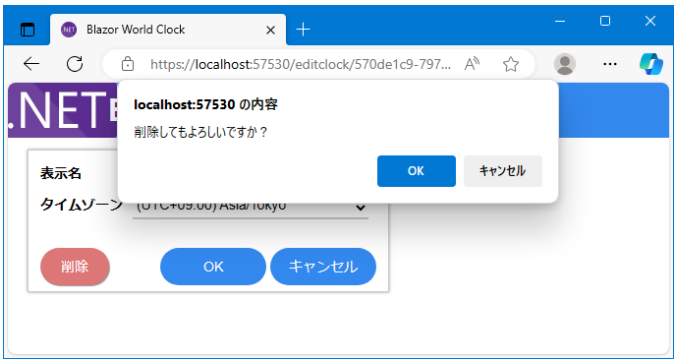
```
private async Task OnClickDelete()
{
    await this.ClockService.DeleteClockAsync(this.ClockId);
    this.NavigationManager.NavigateTo("./");
}
```

6. そしてこの OnClickDelete メソッドを、時計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。

```
<ClockForm Item="Item" OnClickOK="this.OnClickOK"
           OnClickDelete="this.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。

時計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、その時計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 (下図)。



Step 21. 仕上げ - 時刻表示の自動更新

概要

さていよいよ、これまで延ばし延ばしにしてきた、時刻表示の自動更新を実装して仕上げたいと思います。

Blazor (および一般的な SPA フレームワーク) では、基本的には何かユーザーによる操作をきっかけに再描画が行なわれます。
そのため、現時点での Blazor World Clock アプリケーションの時計表示ページは、**ページが開かれた時点の時刻を表示したまま、黙ってみていても表示が変わらず**、いわゆる "時計" として機能していません。

そこで.NET のタイマーを使い、1 秒周期で「状態が変更された」と Blazor ランタイムに知らせることで、**時刻表示を自動更新**するようにしましょう。
なお、.NET におけるタイマー実装は、歴史的理由から、同じ "Timer" というクラス名で名前空間違いで複数の実装があります。
今回は、いちばん汎用・無難に扱える、System.Timers 名前空間の Timer クラスを採用します。

> ※厳密には、タイマーによる 1 秒周期での表示更新では、秒の表示切り替わりタイミングが正確ではありません
> (現実世界での 1.9 秒になってから、Web 上の表示が 0 秒から 1 秒に表示更新される、のように)。
> ですが、そこは本自習書で習得する範疇外ということで、ご容赦ください。

手順

1. BlazorWorldClock.Client プロジェクトの ClockList.razor を Visual Studio で開きます。
   この後の実装を簡略化できるよう、名前空間 System.Timers を @using ディレクティブで開いておきます。

```
@page "/"
@using System.Timers
...
```

2. コードブロック中にて、1 秒 (すなわち 1,000 ミリ秒) 周期でイベント発火する System.Timers.Timer インスタンスのフィールド変数 "_timer" を用意します。

```
@code {
  private IEnumerable<Clock> _clocks = [];

  private readonly Timer _timer = new(interval: 1000);

  protected override async Task OnInitializedAsync()
  {
    ...
```

3. 引き続きコードブロック中にて、タイマーが発火したときに呼び出されるイベントハンドラメソッド " OnTick" を追記します。
   このメソッドの引数および戻り値は、System.Timers.Timer クラスの Elapsed イベントの仕様に合せます。
   このメソッド内では、Razor コンポーネントの **StateHasChanged メソッド**を呼び出し、そうすることで **「(状態に変更があったから) このコンポーネントを再描画せよ」**と Blazor のランタイムに知らせます。

```
@code {
  ...
  private void OnTick(object? sender, ElapsedEventArgs e)
  {
    this.StateHasChanged();
  }
  ...
```

4. コードブロック中 OnInitializedAsync メソッドのタイミングにて、上記 OnTick メソッドを Timer オブジェクトの Elapsed イベントのハンドラとして登録、続けて Timer オブジェクトを始動させます。

```
@code {
  ...
  protected override async Task OnInitializedAsync()
  {
    _clocks = await this.ClockService.GetClocksAsync();
    _timer.Elapsed += this.OnTick;
    _timer.Start();
  }
  ...
```

5. 以上で時計表示ページにて、時刻が刻一刻と表示更新されるようになりました。
   ただしここまでの実装では、時計追加ページや編集ページを開いて時計表示ページから離れたあとも Timer オブジェクトが破棄されず**メモリやタイマー資源を消費**し続けてしまいます。
   そこで時計表示ページを離れるときに、Timer オブジェクトを破棄する処理を実装します。

   この目的には、**.NET の破棄パターン**が使えます。
   つまり、Razor コンポーネントで System.IDisposable インターフェースを実装しておくと、その Razor コンポーネントが未使用となるタイミングで、System.IDisposable インターフェースの Dispose メソッドが呼び出されるようにできています。
   そこで時計表示ページコンポーネントでも System.IDisposable インターフェースを実装し、**Dispose メソッド内で Timer オブジェクトを破棄**することにします。
   Razor コンポーネントでインターフェースの実装を行なうには、**@implements ディレクティブ**を使います。

具体的には ClockList.razor の冒頭で以下の様に記述して System.IDisposable インターフェースを実装することを示します。

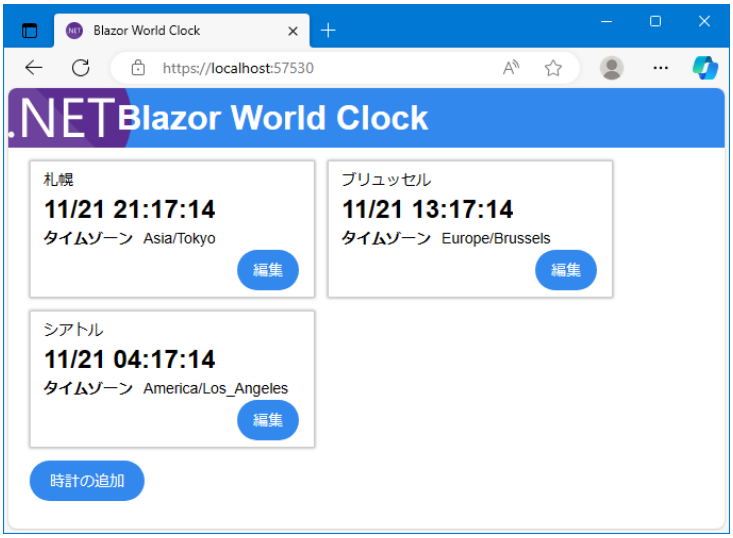
```
@page "/"
@using System.Timers
@implements IDisposable
...
```

6. 続けてコードブロック末尾に、Dispose メソッドを実装します。
   Dispose メソッド内では、まず、Timer オブジェクトの Elapse イベントからハンドラ登録を解除します。
   そして、System.Timers.Timer クラスも .NET 破棄パターン、すなわち IDisposable インターフェースを実装しているので、Timer オブジェクトの Dispose メソッドを実行することで破棄します。

```
@code {
  ...
    public void Dispose()
    {
        _timer.Elapsed -= this.OnTick;
        _timer.Dispose();
    }
}
```

以上で実装は完了です。

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てみてください。
時刻表示が刻一刻と自動更新されるようになったはずです。



これでようやく、"世界時計" としての機能を果たせるようになりました。

以上で Blazor プログラム "Blazor World Clock" の実装を通しての Blazor WebAssembly 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

本自習書の内容はあくまでも**「Blazor ってどんな感じなのだろう?」をお伝えするための入門レベル**であり、本自習書で触れていないテーマはまだまだたくさんあります。

状態管理はどうするのか、単体テストはどうやって書くのか、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の方法は、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、より込み入った JavaScript コードとの連携はどうするのか、Blazor の標準 API に用意されていないブラウザ機能や DOM を触るには、公開 Web サーバーに配置するには...

そこで以下では、さらに Blazor プログラミングの学びを深めていくのに役立つと思われる書籍や Web 上のリソースなどを紹介して、本自習書の締めくくりとしたいと思います。

日本語の書籍

以下では、本自習書の筆者が見つけた、日本語で書かれた Blazor の入門書や実践書を紹介します。

"猫でもわかる Blazor" - ねこじょーかー (著)

- PDF 版 - 入門編 (¥500), 実践編 (¥1,500), EC サイト編 (¥2,500)
- https://book.blazor-master.com/

"Blazor WebAssembly で即売会特化型レジアプリを作ってみた" - サークル きじのしっぽ / け (著)

- PDF 版 (¥800), PDF+書籍 (¥900)
- https://booth.pm/ja/items/1836404

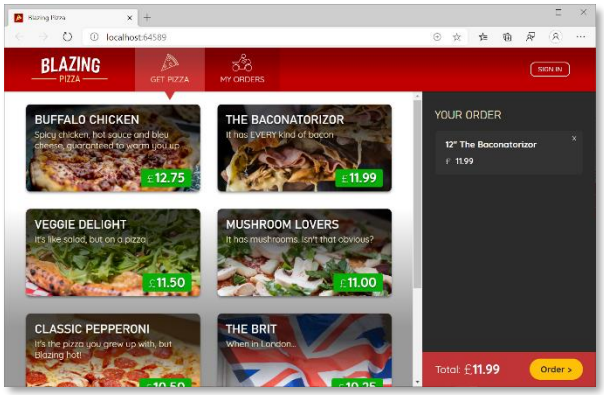
"Blazor 入門" – 日経 BP / 増田 智明 (著)

- Kindle 版 (¥2,722), 単行本 (¥3,080)
- https://www.amazon.co.jp/dp/482229692X/

日本語のサンプル

"Blazor - アプリケーション開発ワークショップ"

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



"BLAZING PIZZA" という架空のピザ屋さんの注文アプリを構築するというワークショップ教材が、.NET Foundation から GitHub リポジトリ上で公開されており、その日本語訳 fork がこちらです。

左図がその完成版のスクリーンショットなのですが、ここから窺い知れるとおり、**相当に本格的な Blazor による Web アプリケーション構築**を体験できます。

ただし、リポジトリに収録されているソースコードを見ますと、Blazor のバージョンが ver.3.2.0 Preview 2 のようです。ver.8.0 がリリースされている現在では、構文等は若干古いかもしれません。

英語の情報

"Blazor - app building workshop"

- https://github.com/dotnet-presentations/blazor-workshop

先に紹介した**「Blazor - アプリケーション開発ワークショップ」**の fork 元です。

こちらは、本稿執筆時点で Blazor のバージョンが 7.0 に更新されています。

"**BlazingChat**"

- https://github.com/CuriousDrive/BlazingChat

全 35 章からなる、相当に広範な内容を扱った、Blazor によるチャットアプリを構築する教材です。**章 (EP, エピソード) ごとに YouTube 動画 (英語) が用意**されており、動画を見ながら学習することができます。

"Awesome Blazor"

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

Blazor に関係するたくさんの "素晴らしい (Awesome)" 情報リソースのコレクションです。
豊富なライブラリ群の紹介 (例えば、**Material デザインや Fluent UI などの UI コンポーネントライブラリ**なども ここで見つかります) のほか、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。

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 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。

2023 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