# **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。

A screenshot of a computer

Description automatically generated

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。

A screenshot of a computer

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

### 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。

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

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>

A screenshot of a computer

Description automatically generated ※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。

A screenshot of a computer

Description automatically generated ※ .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)。

A screenshot of a computer program

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

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。

A screenshot of a computer

Description automatically generated

※ 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。A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated  
A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated  
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。

A screenshot of a computer

Description automatically generated

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 **z+ 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、

A screenshot of a computer

Description automatically generated

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。

A screenshot of a computer

Description automatically generated

### 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。

A screenshot of a computer

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

## 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。

...

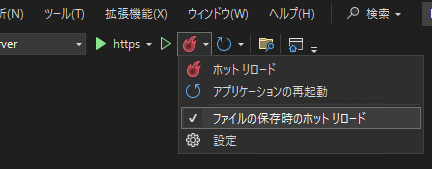
<h1>  
 Welcome to @\_title!  
 </h1>

...

### 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。

A screenshot of a computer

Description automatically generated

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）。

A screenshot of a computer

Description automatically generated

### 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**

A screenshot of a computer

Description automatically generated

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**

A screenshot of a computer

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

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

|  |  |  |
| --- | --- | --- |
|  |  | 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) |
|  |  | 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

|  |  |  |
| --- | --- | --- |
|  |  | 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。 |
|  |  | 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。 |
|  |  | 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。

1. 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

|  |  |  |
| --- | --- | --- |
|  |  | 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)。 |
|  |  | 「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

|  |  |  |
| --- | --- | --- |
|  |  | 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。 |
|  |  | 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。 |
|  |  | 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。 |
|  |  | 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。 |
|  |  | 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。  
A screenshot of a computer

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

※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" },  
];

1. HTML全体を、@foreach (var clock in \_clocks){ ～ } ブロックで囲みます。
2. foreach ループ直下のいちばん親の div 要素に、@key ディレクティブで、列挙するオブジェクトを一意に識別できるキーを指定します。  
   列挙する個々の 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。
3. 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。

A computer screen shot of a program code

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

## 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;

}

1. 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>();

...

1. 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。
2. (コンポーネント内で直に記載していたダミーデータではなく) DI経由で入手した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。
3. 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)。

A screenshot of a computer

Description automatically generated

## 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>> GetClocks**Async**()

**{**

**return** **await** **ValueTask.FromResult**(\_clocks);

**}**

1. 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 OnInitialized**Async**()

{

\_clocks = await this.ClockService.GetClocks**Async**();

}

以上の変更を保存し、プロジェクトをビルドしてブラウザで再読み込みを実行すると、引き続きまだ見た目は前回とまったく変わりませんが、非同期処理に改造しても、正しくダミーデータとして用意した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)。

A screenshot of a computer

Description automatically generated

## 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;

}

1. 続けて、ユーザーインターフェースの作りこみをします。  
   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();

1. また、このあとコーディングするOKボタンがクリックされたときの処理として、OnOKメソッドを同じくコードブロック内に追加します。  
   OnOKメソッド内では、メンバーフィールド\_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();

}

1. 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>

1. 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。

1. 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>

...

1. 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>

1. 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>

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)。

A screenshot of a computer

Description automatically generated

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。

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

### 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)。

A screenshot of a computer

Description automatically generated

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。

A screenshot of a computer program

Description automatically generated  
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）。

A screenshot of a computer

Description automatically generated

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)。  
A screenshot of a computer

Description automatically generated  
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)。

A screenshot of a computer

Description automatically generated

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

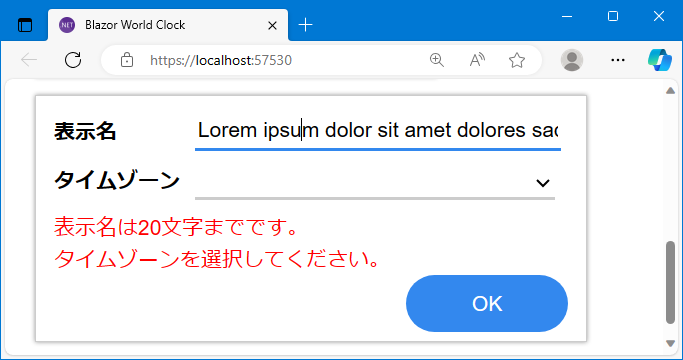
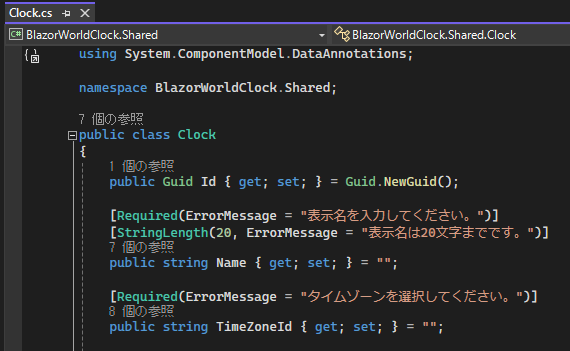
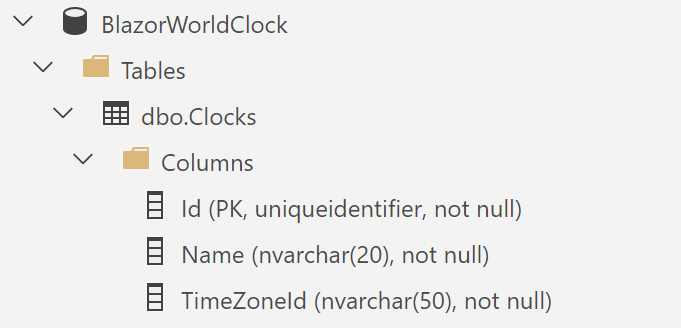
ASP.NET Core によるWebアプリ開発においては、その要件として、リレーショナルデータベースに情報やデータを保存したり読み出したりすることがよくあります。

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。

1. これでBlazorのルーティング機構が有効となりました。  
   しかし、このままでは、どのURLのときにどの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。
2. ここまでの変更を保存したら、いちどプロジェクトをビルドしてブラウザで再読み込みを実行し、とりあえず見た目上の振る舞いは変更前と変わりなく正常動作することを確認しておきましょう。
3. 続けて、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。
4. 「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。
5. 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
6. 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、<div class="clock form">～</div> 要素 (時計追加フォームのマークアップ)
7. 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
8. 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。

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

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）。

A screenshot of a computer

Description automatically generated

## 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

...

1. 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

}

1. 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();

1. 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) 。  
   A computer screen shot of a program

   Description automatically generated  
   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)。
2. 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; }

1. 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);

}

1. 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

1. 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" />

1. 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。

A screenshot of a computer

Description automatically generated　A screenshot of a computer

Description automatically generated

## 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;

}

1. 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

1. 次にコードブロックの編集に移り、以下のメンバーフィールドを追加します。  
   - 編集対象の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;

1. 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;

}

1. 次に、HTMLパートはまだ記述していませんが、先に、OKボタンがクリックされたときの処理をコードブロック内に記述してしまいます。  
   コードブロック内に、(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("./");

}

1. 最後にHTMLを記述しましょう。  
   まず、既存のHTMLマークアップ (<p>Clock Id is @this.ClockId</p>) 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。

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

## 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

1. こうして作成した共通レイアウトコンポーネント "MainLayout" の型の名前空間は、"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

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

<**Router** **AppAssembly**="@typeof(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。

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

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)。

A screenshot of a computer

Description automatically generated

## 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。

1. 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;

1. 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();

...

1. 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)。

A screenshot of a computer

Description automatically generated

※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

1. 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; }

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

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();

}

1. 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>

1. 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("./");

}

1. 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)。

A screenshot of a computer

Description automatically generated

## 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

...

1. 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()

{

...

1. 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();

}

...

1. 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();

}

...

1. 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

...

1. 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。

A screenshot of a computer

Description automatically generated

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

Graphical user interface, website

Description automatically generated**"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いおり、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>