

# B5 Cover Page

(182x257mm or 516x728pt)

# Rust と Github Pages で公開 する Web アプリ開発

— クラウドにお金を払いたくない人のための開発入門 —

[著] J-IMPACT

技術書典 18（2025 年夏）新刊

2025 年 5 月 31 日 ver 1.0

## ■免責

本書は情報の提供のみを目的としています。

本書の内容を実行・適用・運用したことで何が起こりようとも、それは実行・適用・運用した人自身の責任であり、著者や関係者はいかなる責任も負いません。

## ■商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。

また本書では、™、®、©などのマークは省略しています。

# はじめに

無料で運営し、使われ、成長する Web アプリケーションへ

本書で使用する技術と、その選定理由

♣ Github Pages

# 目次

|                               |           |
|-------------------------------|-----------|
| はじめに                          | i         |
| 無料で運営し、使われ、成長する Web アプリケーションへ | i         |
| 本書で使用する技術と、その選定理由             | i         |
| <b>第 1 章 環境構築と技術解説</b>        | <b>1</b>  |
| 1.1 Github Pages              | 1         |
| 1.2 WebAssembly (WASM)        | 2         |
| 1.3 Rust                      | 2         |
| ♣ Rust のインストール                | 2         |
| ♣ WASM 向け target のインストール      | 3         |
| 1.4 Yew                       | 3         |
| 1.5 Trunk                     | 4         |
| <b>第 2 章 Hello, Yew!</b>      | <b>5</b>  |
| 2.1 最も簡単な Web アプリ             | 5         |
| 2.2 作業の進め方                    | 5         |
| ♣ 空のプロジェクトを作って、必要なファイルを再現する   | 6         |
| ♣ リポジトリをフォークする                | 7         |
| 2.3 Web アプリを実装するためのファイルの解説    | 9         |
| ♣ Cargo.toml                  | 9         |
| ♣ src/main.rs                 | 10        |
| ♣ index.html                  | 12        |
| 2.4 ローカル環境での Web アプリの実行       | 15        |
| 2.5 Github Page のためのファイル解説    | 15        |
| ♣ Trunk.toml                  | 15        |
| <b>付録 A Trunk 実行時のエラーについて</b> | <b>17</b> |
| <b>あとがき / おわりに</b>            | <b>19</b> |

# 第 1 章

## 環境構築と技術解説

### 1.1 Github Pages

Github Pages（ギットハブ・ページズ）は、Github 上のリポジトリを使って Web サイトを無料で公開できるサービスです。Github のアカウントがあれば誰でも使用できる機能なので、アカウントをお持ちでない場合は Github のサイトでアカウント作成（サインアップ）してください。

[外部リンク] Github

<https://github.com/>

アカウント名は好きに設定していただいて構いません。本書では、読者のアカウント名に相当する箇所を `[your-user-name]` で記述します。

Github Pages のようなサービスをホスティングサーバといいます。一般に、ホスティングの形式には静的・動的の 2 種類が存在していますが、Github Pages は静的ホスティングにのみ対応しています。静的ホスティングとは、Web サイトを見にきたクライアント（ユーザ）に対して、常に同じ内容のページを送信するサービスです。それに対し動的ホスティングでは、クライアントからのリクエスト内容に応じて、送信するページの内容を「動的に」生成します。

これを聞いて、静的ホスティングしかできない Github Pages では、Web アプリを運営することはできないのではないかと感じた読者も多いと思います。なぜならば、私たちが日頃から利用している Web アプリの多くが、ユーザからの様々なリクエストに対して適切な応答を返すサービスであるからです。

しかし、ホスティングが静的であることと、Web ページの内容が常に固定されていることは同義ではありません。Web ページにスクリプトやバイナリを含めることによって、見た目には「動的な」応答を作り出すことができます。

確かに、動的ホスティングサービスを使って作れる Web アプリと比べると、Github Pages が提供できる機能には制限があります。たとえば、ログイン機能はホスティングサーバでの照合処理が

必要になるため、Github Pages では実装することができません。しかし、私たちが Web アプリに求める多くの機能は、実は静的ホスティングでも十分に対応が可能なのです。

本書では、静的ホスティングでも提供可能な様々な機能を紹介します。もし、読者の頭に「こういうサービスを気に入ってくれる人がいるのでは？」というアイデアが浮かんで、それが本書で紹介した機能で実現可能なのであれば、Github Pages を使って「サッ」と公開してしまいましょう。そして、実際にユーザに使ってもらってフィードバックを受け、サービスをさらに良いものにする…という流れが実現されたなら、著者としてこれ以上に嬉しいことはありません。

## 1.2 WebAssembly (WASM)

静的ホスティングで「動的な」機能を実現するためには、**JavaScript** (JS) を使用することが多いです。しかし本書では **WebAssembly** (WASM) を使用します。JS がスクリプトであるのに対し、WASM はバイナリであるという違いがあります。

私たちは人間が読める言語を使ってプログラミングをします。この人間語がスクリプトです。これを Web アプリとして実行するためには、インタプリタによって機械語へ翻訳する必要があります。一方、WASM 等のバイナリは機械語への翻訳後（コンパイル済み）のファイルなので、実行時に翻訳する必要がありません。この、実行時に翻訳の手間を省けることに加え、WASM が C/C++ や Rust などの高速な言語をコンパイルしたものであることから、JavaScript に比べて実行速度がかなり速くなります。本書では、Web アプリで WASM の機能が生きる例も紹介します。

## 1.3 Rust

WASM を生成するために、本書では **Rust** を使用します。Rust は安全性・高速性・並列性に優れた開発言語です。安全性とは、悪意を持ったユーザの攻撃対象となるようなバグを作り込みにくいということであり、高速性と並列性はそのままプログラムの軽快な動作に繋がります。このように Web アプリにとって嬉しい特徴を備えていることから、本書では Rust を採用しました。Rust は Web アプリに限らず、オペレーティングシステム (OS) 開発や組み込み開発などのコアな業界でも広く採用されており、今後もシェアを大きく伸ばしていくと予想されています。

### ♣ Rust のインストール

Rust をインストールする手順は、公式サイトを参考にしてください。

[外部リンク] Rust プログラミング言語 (公式)

<https://www.rust-lang.org/ja>

インストール作業後にリスト 1.1 のコマンドを実行し、Rust のバージョン情報が表示されたら、

インストール手順が正しく完了したことを確認できます。

▼ リスト 1.1: Rust のバージョン確認: shell

```
> rustc --version
```

## ♣ WASM 向け target のインストール

Rust のインストールが完了すると、Rust 言語で記述したコードをビルド（コンパイル）してバイナリを作ることができるようになります。ただし、このバイナリはどこで動かすか（実行環境）によって中身が変わります。たとえば、Windows 用・macOS 用・Linux 用では、それぞれ違う形式のバイナリが必要になります。

Rust では、どの環境向けにバイナリを作るかをターゲットによって指定します。このターゲットは 3 つの情報で表現されており、まとめてターゲットトリプル（target triple）といいます。

▼ リスト 1.2: ターゲットトリプルの構造

```
<アーキテクチャ（CPUの種類）>-<ベンダー（提供元）>-<OSの種類>
```

Rust は普段、自分が動いている PC 向けのターゲットでバイナリを作ります。しかし、任意のターゲットをインストールすることで、その環境向けのバイナリを生成することができます。この機能をクロスビルド（クロスコンパイル）といいます。

WASM 向けのターゲットは `wasm32-unknown-unknown` です。`wasm32` は WebAssembly の 32 ビットアーキテクチャを意味します。ベンダー情報・OS 情報がともに `unknown` であり、特定の提供元や OS に縛られていないことから、WASM は非常に広い環境で動作することがわかります。

本書を読み進めるにあたっては、このターゲットをリスト 1.3 のコマンドでインストールしておいてください。

▼ リスト 1.3: WASM 向けターゲットのインストール: shell

```
> rustup target add wasm32-unknown-unknown
```

## 1.4 Yew

Yew は Rust で Web フロントエンドアプリケーションを作るためのフレームワーク（ライブラリ）です。WASM を生成するためのコードを、安全性に優れた Rust で記述することができます。なお、Rust ではライブラリのことをクレートと呼びます。

Yew などのクレートを使用するためには、Rust のプロジェクト内で使うクレートとバージョンを宣言します。具体的な方法は 2 以降で解説します。



## 1.5 Trunk

Yew を使って記述した Rust のコードから WASM を生成するために、**Trunk** を使用します。リスト 1.4 のコマンドで、Trunk を Rust にインストールしてください。

### ▼ リスト 1.4: Trunk のインストール: shell

```
cargo install --locked trunk
```

Trunk は Rust から WASM へのビルドを実行するだけでなく、それを HTML ファイルと結合して WASM が動作する Web ページを生成します。また、PC にローカルなサーバ (localhost) を一時的に構築し、実際のページの動作を確認することもできます。



このあとのサンプルコードで、Trunk の動作時にエラーが生じた場合は公式ページ <https://trunkrs.dev/> の Getting Started を参照してください。たとえば、Apple M1 の環境では追加のコマンド実行が必要になる場合があります。

## 第 2 章

# Hello, Yew!

### 2.1 最も簡単な Web アプリ

新しい技術の最初の一步を踏み出すときには、「Hello, World!」を実行することが慣例です。ここでは、「Hello, Yew!」の文字列を表示するだけの Web アプリを作成し、Github Pages で公開するための基本を学びます。この Web アプリのサンプルは、以下の URL で公開されています。

[外部リンク] Hello, Yew!

<https://j-impact.github.io/hello-yew/>

文字を表示するだけでも立派な Web アプリです。この Web アプリを作るためのコード一式は、以下のリポジトリにまとめています。

[外部リンク] J-IMPACT/hello-yew

<https://github.com/J-IMPACT/hello-yew>

### 2.2 作業の進め方

以降の章では、このリポジトリに含まれるファイルの内容を 1 つずつ解説することで、Github Pages で Web アプリを公開するための最小構成を学習します。実際に作業を行いながら本書を読み進める場合には、以下の 2 通りの方法を推奨します。



ただし、どの方法を選択する場合であっても、Github アカウントが作成済みであり、自身の PC で `git` コマンドが使用可能になっていることを前提とします。`git` コマンドが使用できない場合には、Git の公式サイト <https://git-scm.com/> を参照してインストールしてください。

## ♣ 空のプロジェクトを作って、必要なファイルを再現する

Rust で空のプロジェクトを作成し、その中に `J-IMPACT/hello-yew` 内のファイルを作成しながらリポジトリを再現します。コードの写経を通して理解度を高めたい読者におすすめの方法です。コードを写し間違えると Web アプリが動作しなくなることがあるため注意が必要ですが、動作に不具合がある場合のバグ修正も含めて、非常に実践的な経験を積むことができると思います。

この方法で作業を進める場合には、シェルやコマンドプロンプトで Rust プロジェクトを作成したいディレクトリ（フォルダ）に移動し、リスト 2.1 のコマンドで、`hello-yew` という名前の空の Rust プロジェクトを作成します。

### ▼ リスト 2.1: Rust プロジェクトの新規作成: shell

```
> cargo new hello-yew
```

すると `hello-yew` ディレクトリが作成され、その中に Rust で開発を行うための必要最小限のファイルが生成されます。この `hello-yew` ディレクトリを VS Code などのエディタで開き、各ファイルの内容が `J-IMPACT/hello-yew` と同じになるようにコードを修正してください。

[外部リンク] Visual Studio Code (参考)

<https://code.visualstudio.com/>



いくつかのファイルは新規に作成する必要があります。

ただし、`Cargo.lock` ファイルは無視し、複製も削除もしないでください。このファイルは Rust のビルド時に、そのときのクレートの状況に合わせて自動で生成されますが、Web アプリ全体の動作に影響を与えません。



また、`J-IMPACT/hello-yew` に含まれていないからといって、リスト 2.1 の実行直後から存在する `.git` ディレクトリは絶対に削除しないでください。このディレクトリは Git ならびに Github でのプロジェクト管理に極めて重要です。

なお、`.git` ディレクトリは隠しフォルダに設定されているため、PC の設定によっては見えない場合があります。

この方法を採用した場合には、新規作成した `hello-yew` ディレクトリと、読者の Github リポ

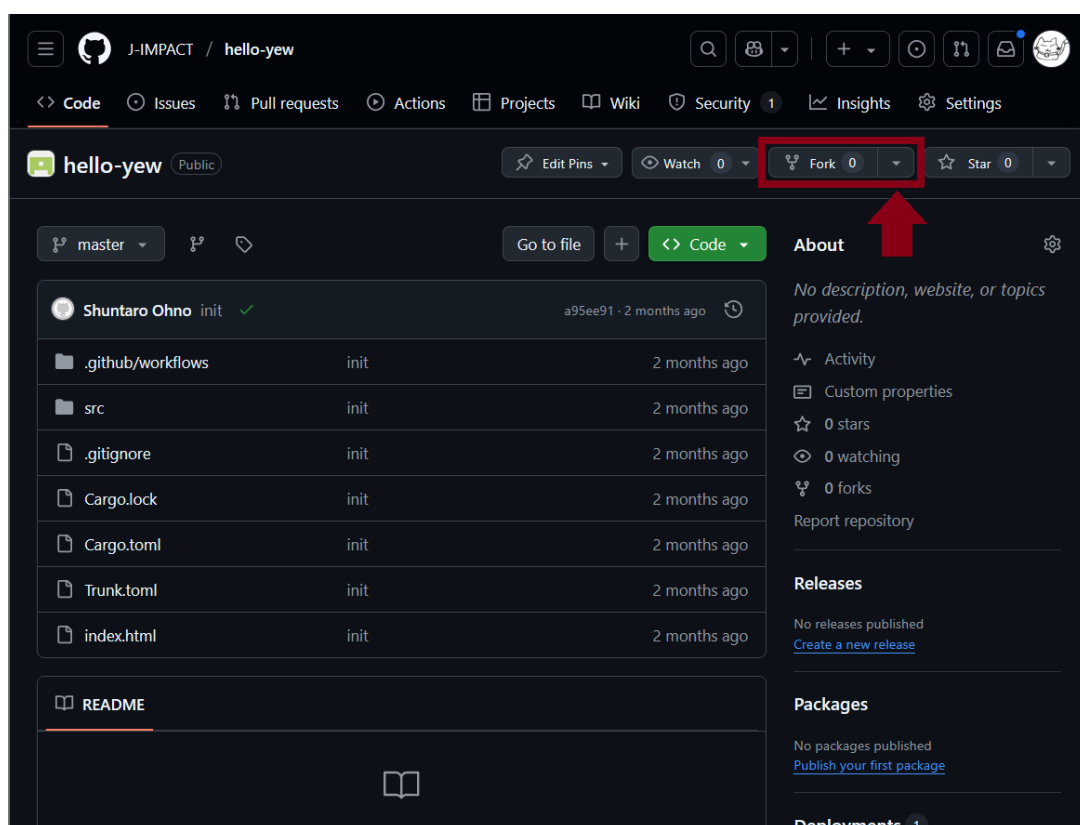
ジトリを結びつける作業が必要です。(作業内容は後ほど解説します)

## ♣ リポジトリをフォークする

コードを写経することに意味を感じない、またはサッと一通り本書を読み終えてしまいたい読者にはこちらの方法をおすすめします。

Github におけるリポジトリの「フォーク」とは、他のユーザが作成したリポジトリを自分のアカウントにコピーすることを意味します。コピー元のリポジトリを「本家」とし、自身のアカウントの管理下に「分家」を作成するイメージです。具体的には、本章の「Hello, Yew!」リポジトリは「J-IMPACT」のアカウントで作成したため `J-IMPACT/hello-yew` という名前になっていますが、これをフォークすることで `[your-user-name]/hello-yew` が作成されます。

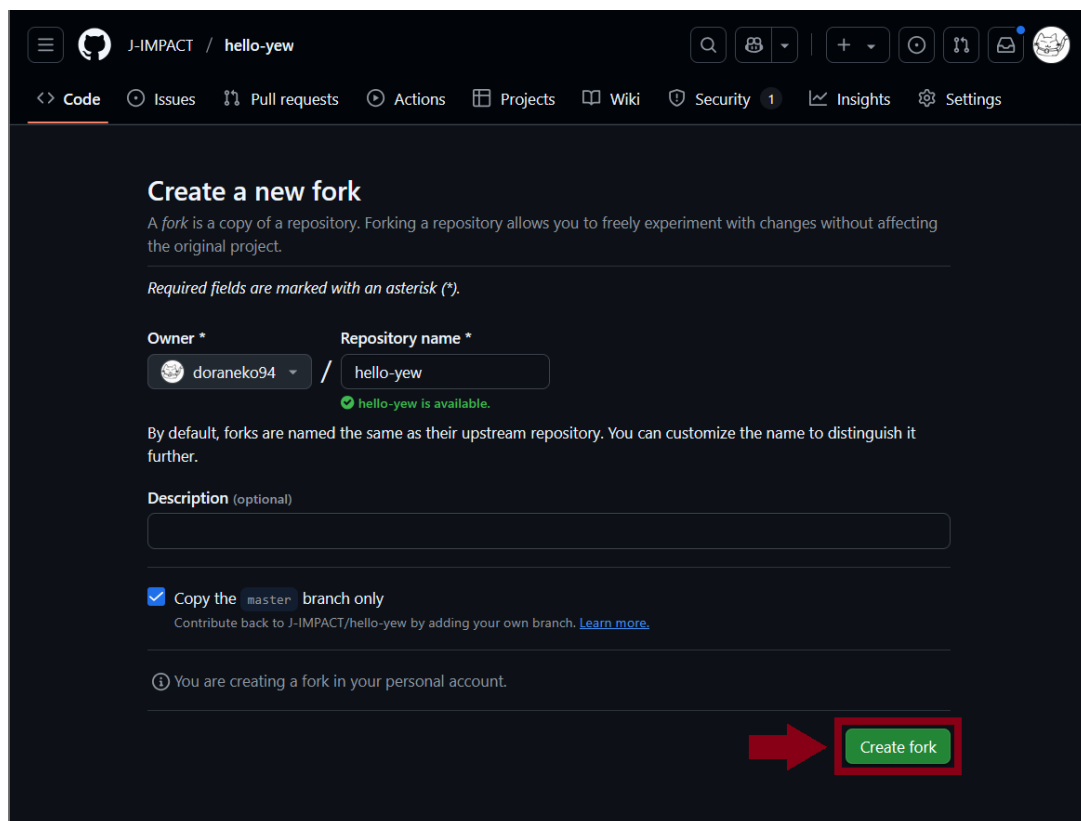
フォークの方法は非常に簡単で、Github にログインした状態で図 2.1 に示した「Fork」ボタンをクリックするだけです（なお、Fork ボタンをクリックする前に、その右横にある「Star」ボタンも押していただくと著者が泣いて喜びます）。



▲ 図 2.1: 「Fork」ボタンの位置

すると図 2.2 のような画面が表示されるので、「Create fork」ボタンをクリックするとフォークが完了し、読者のアカウントの管理下に置かれた `[your-user-name]/hello-yew` リポジトリの

ページに遷移します。



▲ 図 2.2: 「Create fork」 ボタンの位置



J-IMPACT/hello-yew と [your-user-name]/hello-yew の内容は独立しています。よって、分家である読者の hello-yew の内容を変更しても、本家のリポジトリの内容には影響を与えません。

この [your-user-name]/hello-yew を使って作業を行うため、リポジトリの内容を自身の PC にコピーしておきます。シェルやコマンドプロンプトでリポジトリのコピーを作成したいディレクトリ（フォルダ）に移動し、リスト 2.2 のコマンドでリポジトリをクローンしてください（[your-user-name] は、読者のアカウント名で置き換えてください。カッコ [] は削除してください。つまり、この部分は読者のリポジトリの URL に相当します）。

#### ▼ リスト 2.2: リポジトリのクローン: shell

```
> git clone https://github.com/[your-user-name]/hello-yew
```

## 2.3 Web アプリを実装するためのファイルの解説

ここからは、各ファイルの役割と内容を解説します。

### ♣ Cargo.toml

`Cargo.toml` は Rust プロジェクトに関する様々な設定を記述するファイルです。設定の中でも特に重要なのが、使用するクレートを宣言する `[dependencies]` の項目です。リスト 2.3 にこの部分を表示します。

#### ▼ リスト 2.3: dependencies の内容: Cargo.toml

```
[dependencies]
yew = { version = "0.21.0", features = ["csr"] }
```

ここでは `yew` クレートのみを使用することを宣言しています。 `version = "0.21.0"` は、これまで数多くリリースされた `yew` クレートの中でも、バージョン 0.21.0 を使用することを指定しています。異なるバージョンのクレートを使用するとビルドに失敗する場合があるため、指定するバージョンを吟味することは非常に重要です。

`features` では、`yew` クレートを使用する際のモード (feature) を指定しています。feature は配列として複数指定することができるため、`feature"s"` となっています。feature の指定を切り替えることによって、クレートが提供する関数や構造体の内容を変化させることができます。

ここで指定している `csr` は、**Client Side Rendering** を意味しています。これと対になるのが **Server Side Rendering (SSR)** です。レンダリング (Rendering) とは、コンピュータがデータを処理し、HTML 等による描画を行う処理のことです。つまり、「画面に何をどう表示するか」を決定する処理です。

以下、「幅が X、高さが Y の平行四辺形の面積を計算する Web アプリ」を例に、SSR と CSR の動作の違いを説明します。

### Server Side Rendering (SSR)

Web アプリにアクセスし、これを利用するユーザのことを**クライアント**といいます。対して、Web アプリの提供元を**サーバ**といいます。

SSR は、レンダリングをサーバ側で行う方式です。たとえばクライアントが Web アプリのフォームに「X=3, Y=4」と入力して「計算」ボタンを押したとします。SSR ではこの入力内容が**リクエスト**としてサーバに送られ、サーバでは「 $3 \times 4=12$ 」という計算を実行します。

そして計算結果「12」を含む HTML が生成され、それが**レスポンス**としてクライアントに返送されます。その結果、クライアントのブラウザに「面積は 12 です」と表示されます。

つまり SSR では、計算と描画を**サーバ側で実行**し、結果だけをクライアントに送る構造になっています。そのため、処理速度はサーバの性能に依存します。

## Client Side Rendering (CSR)

CSR は、レンダリングをクライアント側、つまりブラウザの中で行う方式です。この場合、クライアントが Web アプリにアクセスした時点で、サーバから HTML や CSS、WASM などの**アプリケーションのロジック**を含むファイル一式が送られます。この中には、

「幅が X、高さが Y の平行四辺形の面積を求めたかったら、 $X \times Y$  を計算してね」

という指示がすでに組み込まれています。

CSR では、クライアントがフォームに「 $X=3$ ,  $Y=4$ 」と入力した情報はサーバには送信されません。その代わりにクライアント側のブラウザ内で、すでに送られてきている指示に従って「 $3 \times 4=12$ 」という計算を実行し、結果の「12」という文字を画面に表示します。

つまり CSR では、計算と描画を**クライアント側で実行**します。そのため、処理速度はユーザの端末性能に依存しますが、一度ファイルを読み込めばページ遷移や操作が高速に行えるという利点もあります。

### 本書における基本戦略

実装したい Web アプリの機能によって、SSR・CSR それぞれに向き・不向きがあると考えられます。そのため、可能であればこれらを組み合わせて使用したいところです。しかし、静的ホスティングサーバである Github Pages では、サーバでの処理を実行できないため SSR は採用できません。よって、Github Pages は以下のような Web アプリを提供する用途には適していないことがわかります。

1. サーバの高性能な計算機能を提供する Web アプリ（物理シミュレータなど）
2. レンダリング時に、クライアントに秘匿すべき情報が必要となる Web アプリ（ログイン処理など）

これら以外の Web アプリの機能は、CSR でも十分に実装することができます。また、1. のように「高機能なサーバ処理」を提供することはできませんが、「高速なバイナリ（WASM）」や「高機能なアルゴリズム」を提供して、有用な物理シミュレータを作ることは可能です（ログイン処理に関しては、Web アプリへのアクセス時に全ユーザ名とパスワードをクライアントに送信してしまえば原理上は実装可能ですが、セキュリティの観点から現実的ではありません）。

本書では、CSR で実装可能な機能を数多く紹介し、それらを組み合わせて有意義な Web アプリを Github Pages 上で公開することを基本戦略とします。

## ♣ src/main.rs

`src` ディレクトリには Rust のソースコードが格納されます。その中で、`main.rs` が Web アプリの動作を定義する核となっています。

ここでは、`main.rs` の内容を解説しながら Rust ならびに Yew の使い方を解説します。

### クレート使用の宣言

## ▼ リスト 2.4: yew クレートの使用を宣言: src/main.rs

```
use yew::prelude::*;
```

リスト 2.4 では `yew` クレートを使用することを宣言しています。`prelude` は、`yew` クレートの中でも頻繁に利用される機能の一覧です。つまりリスト 2.4 の 1 行は、「`yew` クレートの頻繁に利用される機能（`prelude`）のすべて（`*`）を使用する」ことを宣言しています。

## コンポーネントの定義

Yew を使う主な目的は、描画する **コンポーネント** を定義することです。コンポーネントとは、複数の HTML 要素・CSS・JavaScript をまとめた **再利用可能な単位** です。リスト 2.5 では、`App` という非常に簡単なコンポーネントを定義しています。

## ▼ リスト 2.5: App コンポーネントの定義: src/main.rs

```
#[function_component(App)]
fn app() -> Html {
    html! {
        <h1>{ "Hello, Yew!" }</h1>
    }
}
```

コンポーネントは、`function_component` の記法を使用すると、`Html` 構造体を返す関数として定義することができます。`Html` 構造体は `html!` マクロを使用して記述することができます。ここでは `App` を、`<h1>Hello, Yew!</h1>` という HTML を表示するコンポーネントとして定義しています。コンポーネントは再利用可能な単位なので、これを `<App />` として HTML に挿入すると、その箇所では `<h1>Hello, Yew!</h1>` として表示されます。

なお、関数名の `app` には特に意味はありません。関数の返り値が `Html` であり、他の関数と被りがなければ関数名は何でも良いのですが、コードの可読性を高めるため、コンポーネントと同様の名前をつけておきましょう。

`html!` マクロの記法について補足しておきます。`<h1>Hello, Yew!</h1>` を表示したい場合に、リスト 2.6 のように書くのは正しくありません。

▼ リスト 2.6: 間違った `html!` マクロの書き方

```
html! {
    <h1>Hello, Yew!</h1>
}
```

`html!` マクロの内部には、HTML タグ、コンポーネント、`{ }` で囲んだ Rust の式や変数を書くことができます。よって、HTML タグである `<h1>` や `</h1>` を書くことは正しいのですが、その間にある `Hello, Yew!` は HTML タグやコンポーネントではなく、`{ }` にも囲まれていないので正しく認識できません。そのためリスト 2.5 では「Hello, Yew!」を Rust の文字列 `"Hello, Yew!"` として捉え、それを `{ }` で囲んで `h1` タグの内部に挿入しています。また、以



下のように変数に代入してから `html!` マクロに挿入することもできます。

▼ リスト 2.7: 変数を使用した App コンポーネントの定義

```
#[function_component(App)]
fn app() -> Html {
    let message = "Hello, Yew!";
    html! {
        <h1>{ message }</h1>
    }
}
```

また、`html!` マクロ内には `if` 分岐や `for` ループを含めることもできます。これらの記法については本書の後の章で解説します。

## Web アプリのレンダリング

Web アプリ全体を記述したコンポーネントを使用して、Web アプリのレンダリングを行うことができます。

▼ リスト 2.8: Web アプリのレンダリング: `src/main.rs`

```
fn main() {
    yew::Renderer::<App>::new().render();
}
```

`main` 関数内で `yew::Renderer::<App>::new()` と書くことで、Web アプリの全体像となるコンポーネントが `App` であることを指定して `Renderer` を定義できます。その後、`render` メソッドを使用して描画を実行します。

## ♣ `index.html`

Yew を使用して記述した Rust コードは、Trunk を使用して Web アプリへとビルドします。その際、`index.html` という HTML テンプレートを用意する必要があります。リスト 2.9 にその全体像を示します。

▼ リスト 2.9: HTML テンプレートの内容: `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Hello, Yew!</title>
</head>
<body>

</body>
```

```
</html>
```

`<head>` 内では、`<meta>` タグや `<title>` タグを使って Web アプリの設定をすることができます。一方、`<body>` 内は空白になっています。

Trunk を使用して Web アプリをビルドすると、Rust から WASM バイナリが作られ、この `<body>` の内部にはそのバイナリを読み込むための記述が挿入されます。`hello-yew` のディレクトリ内でリスト 2.10 のコマンドを使用して、実際に Web アプリをビルドしてみましょう。(1 から写経している読者も、現時点で `Cargo.toml` , `src/main.rs` , `index.html` があればビルドは成功するはずです)

#### ▼ リスト 2.10: Trunk で Web アプリをビルド: shell

```
> trunk build --release
```

すると `dist` というディレクトリが作成され、その中に以下のファイルが生成されます。

- `hello-yew-xxxxxxxxxxxxxxxxxx_bg.wasm`
- `hello-yew-xxxxxxxxxxxxxxxxxx.js`
- `index.html`

ビルド時に `WARN no rust project found` などの警告を発して、上記のファイルが生成されなかった場合は、[Appendix A "Trunk 実行時のエラーについて"](#)を参照してください。

ファイル名の「xxxxxxxxxxxxxxxxxx」の部分はビルドごとに割り振られるハッシュ値で、ビルドしたファイルの内容によって変化します。そのため、ファイル内容の更新を把握しやすくなっています。

`.wasm` ファイルは Rust コードから生成した WASM のバイナリです。この中に Web アプリの計算・描画処理が含まれていますが、スクリプト（コード）ではなくバイナリなので、これをエディタで開いても人間が読めるようにはなっていません。また、`.js` ファイルは `.wasm` ファイルを読み込むためのスクリプトです。

`dist` ディレクトリに生成された `index.html` の内容を示します。

#### ▼ リスト 2.11: WASM 読み込みが追加された HTML: `dist/index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Hello, Yew!</title>
  <link rel="modulepreload" href="/hello-yew/hello-yew-xxxxxxxxxxxxxxxxxx.js">
  <script src="/hello-yew/hello-yew-xxxxxxxxxxxxxxxxxx.js"
    crossorigin="anonymous" integrity="sha384-PU1JuRzrwT5mHB5G7NMxTXDiE/8R/
    qaDNB3QCJjXmg7WXgbnP9kAh5UjFQ7PIM2"></script>
  <link rel="preload" href="/hello-yew/
  hello-yew-xxxxxxxxxxxxxxxxxx_bg.wasm" crossorigin="anonymous" integrity="sh
  a384-pUY38i2Dst7ZceBh+URSvtB0J+noiPpJ8Q+GyaIsEnz7V6+5X5vcQfG6WaUFZczW" as>
```

```

>="fetch" type="application/wasm"></head>
<body>

<script type="module">
import init, * as bindings from '/hello-yew/hello-yew-xxxxxxxxxxxxxxxxx.js'
>';
const wasm = await init({ module_or_path: '/hello-yew/hello-yew-xxxxxxxxx'
>xxxxxxxx_bg.wasm' });

window.wasmBindings = bindings;

dispatchEvent(new CustomEvent("TrunkApplicationStarted", {detail: {wasm}}
>));

</script></body>
</html>

```

`hello-yew` ディレクトリの直下に用意した `index.html` に、`.wasm` ・ `.js` ファイルを通して WASM を使用するための記述が機械的に挿入されています。具体的には、`<head>` 内に `<link>`、`<body>` 内に `<script>` が追加された形です。これらの記述によって Web アプリ内での WASM の使用が可能になりますが、今回は `<h1>Hello, Yew!</h1>` を表示するだけなので、アプリの見た目はリスト 2.12 と同じになります。

#### ▼ リスト 2.12: hello-yew アプリと等価な HTML

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Hello, Yew!</title>
</head>
<body>
  <h1>Hello, Yew!</h1>
</body>
</html>

```

またリスト 2.9 とリスト 2.11 を比較すると、挿入された `<link>` の部分を除いて `<head>` 内の記述が保たれていることがわかります。このように、WASM に依存しない `<title>`、`<meta>` などの情報は、事前に `index.html` の `<head>` 内に記述しておくことができます。

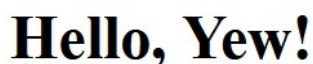
## 2.4 ローカル環境での Web アプリの実行

リスト 2.10 で確認したとおり、ここまでのファイルが正しく記述されていれば Web アプリをビルドすることができます。手元の環境（ローカル）で Web アプリの動作を確認するには、リスト 2.13 のコマンドを実行します。

▼ リスト 2.13: localhost で Web アプリを実行: shell

```
> trunk serve --release --open
```

するとブラウザが立ち上がり、図 2.3 のように「Hello, Yew!」が表示されます。



▲ 図 2.3: Hello, Yew!アプリの実行

## 2.5 Github Page のためのファイル解説

ここからは、作った Web アプリを Github Pages で公開するために必要なファイルを解説します。

### ♣ Trunk.toml

`Trunk.toml` は、Trunk でのビルドに関する設定を記述するファイルです。`hello-yew` の中では、リスト 2.14 の内容のみ記述しています。

▼ リスト 2.14: Web アプリの URL の指定: Trunk.toml

```
[build]
public_url = "/hello-yew/"
```

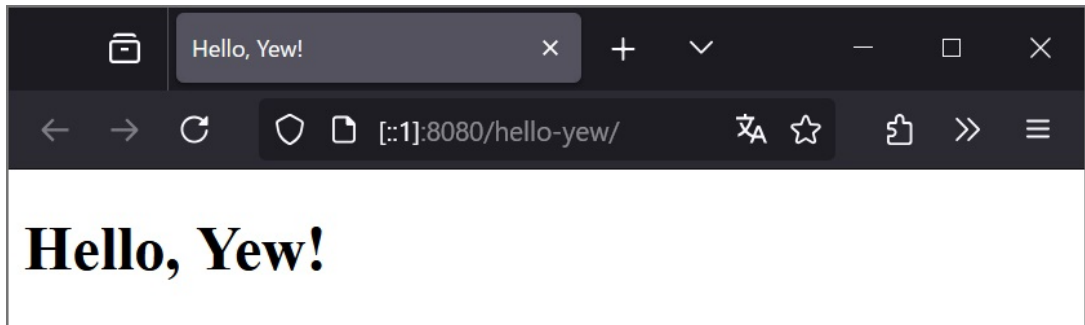
`public_url` を指定することにより、Web アプリの URL を変更できます。これを指定しない場合、デフォルトの URL は `"/"` です。

この機能を簡単に実験してみましょう。まず、`Trunk.toml` にリスト 2.14 の記述がある状態で、ローカル環境で Web アプリを起動します（リスト 2.15）。

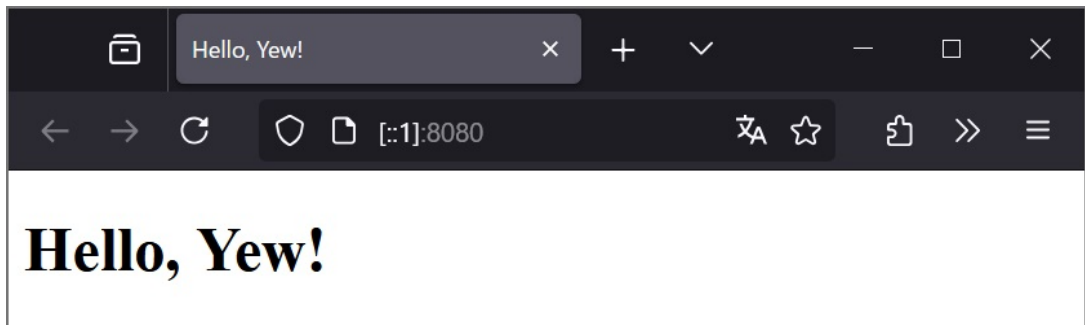
▼ リスト 2.15: localhost で Web アプリを実行（再掲）: shell

```
> trunk serve --release --open
```

すると図 2.4 のように、ブラウザの URL 欄には `a` と表示されているはずです。



▲ 図 2.4: `public_url = "/hello-yew/"` の場合の URL



▲ 図 2.5: `public_url = "/"` の場合の URL

# 付録 A

## Trunk 実行時のエラーについて

`trunk build` , `trunk serve` などのコマンド実行時に、リスト A.1 のような警告を発してビルドが不十分に終わることがあるかもしれません。

### ▼ リスト A.1: trunk build 実行時の警告: shell

```
> trunk build --release
20XX-XX-XXTXX:XX:XX.XXXXXXZ INFO starting build
20XX-XX-XXTXX:XX:XX.XXXXXXZ INFO spawning asset pipelines
20XX-XX-XXTXX:XX:XX.XXXXXXZ WARN no rust project found ←←←← 警告
20XX-XX-XXTXX:XX:XX.XXXXXXZ INFO applying new distribution
20XX-XX-XXTXX:XX:XX.XXXXXXZ INFO success
```

この場合、生成された `dist` ディレクトリには `index.html` しか存在せず、その中身には WASM に関する記述が追加されません。

リスト A.1 が生じる主要な原因の 1 つは、コマンドを実行するディレクトリの位置が間違っていることです。事前に `cd` コマンドを使って `Cargo.toml` ファイルが存在するディレクトリに移動してから `trunk build` , `trunk serve` などのコマンドを実行してください。

### ▼ リスト A.2: trunk build の実行 (hello-yew プロジェクトの場合) : shell

```
.../hello-yew> trunk build --release
```

コマンドを実行したディレクトリに `Cargo.toml` が存在するにも関わらず、リスト A.1 の警告 (「Rust プロジェクトが見つかりません」) が表示される場合もあります。この多くは Trunk のバージョンが古いケースで生じます。

従来の Trunk では、たとえばリスト A.3 のような記述を `Cargo.toml` に含める必要がありました。

### ▼ リスト A.3: Trunk v0.20.0 以前の lib の記述: Cargo.toml

```
[lib]
crate-type = ["cdylib"]
```

この記述は「Rust を他の言語のライブラリとして使用する」ことを宣言しています。他の言語とは、たとえば WASM のことです。

しかし、Trunk v0.21.0 からはこの記述を省略することができるようになりました。そのため、本書ではすべての `Cargo.toml` で `[lib]` の記述を省いています。

このように Trunk のバージョンが異なるとビルドに失敗することがあるため、リスト A.4 のコマンドで Trunk のアップデートを実行してください。

▼ リスト A.4: Trunk のアップデート: shell

```
> cargo install trunk --locked --force
```

その際、Rust 自体のアップデートを要求された場合はリスト A.5 のコマンドを実行してください。

▼ リスト A.5: Rust のアップデート: shell

```
rustup update stable  
rustup default stable
```

本書の刊行時のサンプルコードは、Trunk v0.21.14 でビルドできることを確認しています。将来的に v0.22, v0.23, ... がリリースされ、仕様変更によりサンプルコードのビルドができなくなった場合、リスト A.6 のコマンドを実行して Trunk のバージョンを v0.21.14 に戻すことを検討してください。

▼ リスト A.6: Trunk のバージョンを v0.21.14 に指定してインストール: shell

```
cargo install trunk --version 0.21.14 --force
```

## あとかき / おわりに



# Rust と Github Pages で公開する Web アプリ開発

クラウドにお金を払いたくない人のための開発入門

---

2025 年 5 月 31 日 ver 1.0 (技術書典 18)

著 者 J-IMPACT

---

© 2025 J-IMPACT

(powered by Re:VIEW Starter)