rhtml_v2

RHTML Documentation

Introduction

RHTML is a Rust-first SSR framework that brings functional programming patterns to web development. Write real HTML with minimal directives, keep business logic in Rust, and compile everything to a single binary.

Core Philosophy:

- HTML files, not Rust macros
- Functional patterns over imperative code
- SSR-only (use HTMX/Alpine for client interactivity)
- Single binary deployment
- Tailwind CSS as first-class citizen

Quick Start

```
# Install RHTML CLI

cargo install rhtml-cli

# Create new project

rhtml new my-app

cd my-app

# Start development server

rhtml dev

# Build for production

rhtml build --release
```

Project Structure

```
my-app/
├── pages/
│   ├── _layout.rhtml          # Root layout
```

```
│   ├── index.rhtml            # Home page (/)
│   ├── about.rhtml            # About page (/about)
│   └── users/
│       ├── _layout.rhtml      # Users section layout
│       ├── index.rhtml        # Users list (/users)
│       ├── [id].rhtml         # User detail (/users/:id)
│       └── new.rhtml          # New user form (/users/new)
├── components/
│   ├── userCard.rhtml         # Reusable components
│   └── navbar.rhtml
├── static/                    # Static assets
│   └── favicon.ico
└── rhtml.toml                 # Configuration
```

Complete Directive Reference

Core Directives (11 Total)

```
<!-- 1. Conditional Rendering -->
r-if="expression"              <!-- Render if true -->
r-else-if="expression"         <!-- Chain conditions -->
r-else                         <!-- Fallback -->

<!-- 2. Pattern Matching -->
r-match="expression"           <!-- Match against value -->
r-when="pattern"              <!-- Match case -->

<!-- 3. Loops -->
r-for="item in collection"    <!-- Iterate over collection -->

<!-- 4. Interpolation -->
{expression}                   <!-- Evaluate Rust expression -->

<!-- 5. HTML Rendering -->
r-html="expression"            <!-- Render unescaped HTML -->

<!-- 6. Attributes -->
r-attr:name="{expression}"    <!-- Dynamic attribute -->
r-class:name="{boolean}"      <!-- Conditional CSS class -->

<!-- 7. Components -->
r-props="{...}"               <!-- Pass props to component -->
```

File Types & Functions

Layout Files (`_layout.rhtml`)

```
// pages/_layout.rhtml
// Required: layout() function
```

```
 cmp layout(slots: &Slots) {
    <!DOCTYPE html>
    <html>
    <head>
        <title>{slots.get("title").unwrap_or("RHTML App")}</title>
        <script src="https://unpkg.com/htmx.org@1.9.0"></script>
        <script src="https://cdn.tailwindcss.com"></script>
    </head>
    <body>
        <nav>
            <a href="/">Home</a>
            <a href="/users">Users</a>
        </nav>

        <main>
            {slots.content}  <!-- Page content inserted here -->
        </main>

        <footer>
            {slots.get("footer").unwrap_or("© 2024")}
        </footer>
    </body>
    </html>
}

css layout {
    nav {
        background: #333;
        padding: 1rem;
    }
    nav a {
        color: white;
        margin-right: 1rem;
    }
}
```

Page Files (`*.rhtml`)

```
// pages/users.rhtml
// Required: Page() function, one per file.
// Optional: data(), slots{}, cmp(), css

// Fetch data (runs on server)
data fn getUsers(query: &Query) -> Result<Vec<User>, Error> {
    let filter = query.get("filter");
    db::get_users(filter)
}

// Define layout slots

slots {
```

```
    title: "Users Directory",
    footer: "Total users: {props.data.len()}"
}

// Main page component
// Page - one per file.
cmp Page(props: &PageProps<Result<Vec<User>, Error>>) {
    <div class="container mx-auto p-4">
        <h1 class="text-3xl font-bold mb-6">Users</h1>

        <div r-match="props.data">
            <div r-when="Ok(users)">
                <div r-if="users.is_empty()">
                    <p>No users found</p>
                    <a href="/users/new" class="btn">Add First User</a>
                </div>
                <div r-else>
                    <div class="grid gap-4 md:grid-cols-2 lg:grid-cols-3">
                        <div r-for="user in users">
                            <userCard r-props="{user: user}" />
                        </div>
                    </div>
                </div>
            </div>
            <div r-when="Err(e)">
                <div class="alert alert-error">
                    Error loading users: {e}
                </div>
            </div>
        </div>
    </div>
}

// Local component (can also be in components/ folder)
cmp userCard(props: &UserCardProps) {
    <div class="card">
        <img src="{props.user.avatar}" alt="{props.user.name}" />
        <h3>{props.user.name}</h3>
        <p class="text-gray-600">{props.user.email}</p>

        <div r-class:active="{props.user.is_active}"
             r-class:inactive="{!props.user.is_active}">
            {if props.user.is_active { "Active" } else { "Inactive" }}
        </div>

        <!-- HTMX for interactivity -->
        <button hx-delete="/api/users/{props.user.id}"
                hx-confirm="Delete user?"
                class="btn btn-danger mt-2">
            Delete
        </button>
    </div>
}
```

```
css page {
    h1 {
        color: #333;
    }
}

css userCard {
    .card {
        border: 1px solid #e2e8f0;
        border-radius: 8px;
        padding: 1rem;
    }
    .active { color: green; }
    .inactive { color: red; }
}
```

Component Files (`components/*.rhtml`)

```
// components/userAvatar.rhtml
// Required: cmp() function with component name

cmp userAvatar(props: &UserAvatarProps) {
    <div class="avatar">
        <img r-if="props.user.avatar.is_some()"
             src="{props.user.avatar.unwrap()}"
             alt="{props.user.name}" />
        <div r-else class="avatar-placeholder">
            {props.user.name.chars().next().unwrap_or('?')}
        </div>
    </div>
}

css userAvatar {
    .avatar {
        width: 48px;
        height: 48px;
        border-radius: 50%;
        overflow: hidden;
    }
    .avatar-placeholder {
        background: #e2e8f0;
        display: flex;
        align-items: center;
        justify-content: center;
        font-weight: bold;
    }
}
```

Examples

Conditional Rendering

```
<!-- Simple if/else -->
<div r-if="user.is_premium">
    <span class="badge-premium">Premium Member</span>
</div>
<div r-else>
    <a href="/upgrade">Upgrade to Premium</a>
</div>

<!-- Chained conditions -->
<div r-if="score >= 90">Grade: A</div>
<div r-else-if="score >= 80">Grade: B</div>
<div r-else-if="score >= 70">Grade: C</div>
<div r-else>Grade: F</div>
```

Pattern Matching

```
<!-- Enum matching -->
<div r-match="user.subscription">
    <div r-when="Subscription::Free">
        <p>Free tier - Limited features</p>
    </div>
    <div r-when="Subscription::Pro { features, expires }">
        <p>Pro tier - {features.len()} premium features</p>
        <p>Expires: {expires.format("%Y-%m-%d")}</p>
    </div>
    <div r-when="Subscription::Enterprise { seats, .. }">
        <p>Enterprise - {seats} seats</p>
    </div>
    <div r-when="_">
        <p>Unknown subscription</p>
    </div>
</div>

<!-- Option matching -->
<div r-match="user.profile_image">
    <img r-when="Some(url)" src="{url}" class="avatar" />
    <div r-when="None" class="avatar-placeholder">
        No Image
    </div>
</div>

<!-- Result matching -->
<div r-match="api_result">
    <div r-when="Ok(data)">
        <successView r-props="{data: data}" />
    </div>
    <div r-when="Err(ApiError::NotFound)">
        <h2>404 - Not Found</h2>
    </div>
```

```
    <div r-when="Err(e)">
        <errorView r-props="{error: e}" />
    </div>
</div>
```

Loops

```
<!-- Basic iteration -->
<ul>
    <li r-for="item in items">
        {item.name}
    </li>
</ul>

<!-- With index -->
<div r-for="(item, index) in items.iter().enumerate()">
    <span>{index + 1}. {item.name}</span>
</div>

<!-- With filtering -->
<div r-for="user in users.iter().filter(|u| u.is_active)">
    <userCard r-props="{user: user}" />
</div>

<!-- Nested loops -->
<div r-for="category in categories">
    <h2>{category.name}</h2>
    <ul>
        <li r-for="product in category.products">
            {product.name} - ${product.price}
        </li>
    </ul>
</div>
```

Expressions & Interpolation

```
<!-- Simple interpolation -->
<h1>Welcome, {user.name}!</h1>

<!-- Rust expressions -->
<p>You have {messages.len()} new messages</p>
<p>Total: ${(price * quantity * (1.0 + tax_rate)).round()}</p>
<p>Joined {user.created_at.format("%B %d, %Y")}</p>

<!-- Method calls -->
<span>{user.full_name()}</span>
<span>{format_currency(order.total)}</span>
<span>{users.iter().filter(|u| u.is_active).count()} active users</span>
```

Dynamic Attributes & Classes

```
<!-- Dynamic attributes -->
<img r-attr:src="{user.avatar}"
     r-attr:alt="{user.name}"
     r-attr:title="{user.bio}" />

or
<img r-attr="{
      src:user.avatar,
      alt:user.name,
      title:{user.bio
}" />

<a r-attr:href="{format!('/users/{}/edit', user.id)}">
    Edit Profile
</a>

<!-- Conditional classes -->
<div class="user-status"
     r-class:active="{user.is_active}"
     r-class:premium="{user.is_premium}"
     r-class:admin="{user.role == Role::Admin}">
    {user.name}
</div>

<!-- Results in: class="user-status active premium" -->
```

Component Props

```
<!-- Passing props object -->
<userCard r-props="{
    user: current_user,
    show_actions: true,
    compact: false
}" />

<!-- Using struct syntax -->
<dashboard r-props="DashboardProps {
    user: current_user,
    stats: stats_data,
    ..Default::default()
}" />
```

HTML Rendering

```
<!-- Escaped by default -->
<p>{user.bio}</p>  <!-- HTML tags are escaped -->
```

```
<!-- Unescaped HTML (use carefully!) -->
<div r-html="{markdown_to_html(content)}"></div>
```

Integration Examples

With HTMX

```
<!-- pages/todos.rhtml -->
page(props: &PageProps<Vec<Todo>>) {
    <div class="max-w-2xl mx-auto p-4">
        <h1>Todo List</h1>

        <!-- HTMX form -->
        <form hx-post="/todos"
              hx-target="#todo-list"
              hx-swap="beforeend"
              class="mb-4 flex gap-2">
            <input name="title"
                   placeholder="New todo..."
                   class="flex-1 px-3 py-2 border rounded" />
            <button type="submit" class="btn btn-primary">
                Add Todo
            </button>
        </form>

        <!-- Todo list -->
        <ul id="todo-list">
            <li r-for="todo in props.data"
                id="todo-{todo.id}"
                class="flex items-center gap-2 p-2">

                <input type="checkbox"
                       hx-patch="/todos/{todo.id}/toggle"
                       hx-target="closest li"
                       hx-swap="outerHTML"
                       r-attr:checked="{todo.completed}" />

                <span r-class:line-through="{todo.completed}">
                    {todo.title}
                </span>

                <button hx-delete="/todos/{todo.id}"
                        hx-target="closest li"
                        hx-swap="outerHTML"
                        class="ml-auto text-red-500">
                    Delete
                </button>
            </li>
        </ul>
```

```
    </div>
}
```

With Alpine.js

```
<!-- Client-side interactivity with Alpine -->
page(props: &PageProps<ProductList>) {
    <div x-data="{
        search: '',
        category: 'all',
        priceRange: [0, 1000]
    }">
        <!-- Alpine handles client-side filtering -->
        <input x-model="search"
               placeholder="Search products..."
               class="px-3 py-2 border rounded" />

        <!-- RHTML handles server-side rendering -->
        <div r-for="product in props.data.products"
             x-show="product.name.toLowerCase().includes(search.toLowerCase())"
             class="product-card">
            <productCard r-props="{product: product}" />
        </div>
    </div>
}
```

Form Handling

```
// pages/users/new.rhtml

data fn handleForm(method: &Method, form: &FormData) -> FormResult {
    match method {
        Method::GET => FormResult::Empty,
        Method::POST => {
            let user = User {
                name: form.get("name")?,
                email: form.get("email")?,
                role: form.get("role").unwrap_or("user".to_string()),
            };

            match db::create_user(user) {
                Ok(user) => FormResult::Success(user),
                Err(e) => FormResult::Error(e.to_string()),
            }
        }
    }
}

page(props: &PageProps<FormResult>) {
    <div class="max-w-md mx-auto p-4">
        <h1>Create User</h1>
```

```
        <div r-match="props.data">
            <div r-when="FormResult::Success(user)">
                <div class="alert alert-success">
                    User {user.name} created successfully!
                    <a href="/users/{user.id}">View Profile</a>
                </div>
            </div>
            <div r-when="FormResult::Error(msg)">
                <div class="alert alert-error">{msg}</div>
            </div>
            <div r-when="_"></div>
        </div>

        <form method="post" class="space-y-4">
            <div>
                <label for="name">Name</label>
                <input name="name" required
                       class="w-full px-3 py-2 border rounded" />
            </div>

            <div>
                <label for="email">Email</label>
                <input name="email" type="email" required
                       class="w-full px-3 py-2 border rounded" />
            </div>

            <div>
                <label for="role">Role</label>
                <select name="role" class="w-full px-3 py-2 border rounded">
                    <option value="user">User</option>
                    <option value="admin">Admin</option>
                </select>
            </div>

            <button type="submit" class="btn btn-primary">
                Create User
            </button>
        </form>
    </div>
}
```

Configuration

```
# rhtml.toml

[project]
name = "my-app"
version = "0.1.0"
author = "Your Name"

[server]
```

```
port = 3000

host = "0.0.0.0"
workers = 4

[build]
output_dir = "dist"
static_dir = "static"
minify_html = true

minify_css = true

[dev]
hot_reload = true

port = 3000

open_browser = true

watch_paths = ["pages", "components", "static"]

[database]
url = "postgresql://localhost/myapp"
max_connections = 10

[tailwind]
enabled = true

config = "tailwind.config.js"  # Optional custom config
```

Tooling Requirements

Core Tools Needed

1. **RHTML Compiler**
   - Parser for `.rhtml` files
   - Code generator (RHTML → Rust)
   - Type checker integration
   - Error reporting with line numbers
2. **CLI Tool (`rhtml`)**

   ```
   rhtml new <project>      # Create new project
   rhtml dev                # Start dev server
   rhtml build              # Production build
   rhtml routes            # List all routes
   rhtml check             # Type-check without building
   ```

3. **Development Server**
   - Hot reload on file changes

    - Error overlay in browser
    - Request/response logging
    - Static file serving
4. **Build System**
    - File-based routing generator
    - CSS scoping/bundling
    - Asset optimization
    - Binary compilation
5. **IDE Support**
    - VS Code extension for `.rhtml` syntax highlighting
    - LSP for autocomplete and type hints
    - Format on save
    - Go to definition for components
6. **Testing Framework**

```
#[test]
fn test_user_page() {
    let props = PageProps {
        data: vec![test_user()],
    };
    let html = render_page(props);
    assert!(html.contains("test@example.com"));
}
```

Development Workflow

```
# 1. Create project

rhtml new my-app

cd my-app

# 2. Development

rhtml dev
# → Starts at http://localhost:3000
# → Hot reloads on changes
# → Shows errors in browser

# 3. Add pages

echo "page() { <h1>About</h1> }" > pages/about.rhtml
# → Automatically creates /about route

# 4. Build for production
```

```
rhtml build --release
# → Outputs single binary: target/release/my-app

# 5. Deploy
./target/release/my-app
# → Runs on port 3000
# → No Node.js, no dependencies
```

Project Template Structure

```
rhtml-template/
├── .gitignore
├── Cargo.toml
├── rhtml.toml
├── README.md
├── pages/
│   ├── _layout.rhtml
│   └── index.rhtml
├── components/
│   └── .gitkeep
├── static/
│   └── favicon.ico
├── src/
│   └── lib.rs          # Custom Rust functions
└── tests/
    └── pages_test.rs
```

Deployment

Single Binary

```
# Build

rhtml build --release

# Run anywhere
./my-app

# Docker

FROM scratch

COPY target/release/my-app /
EXPOSE 3000

CMD ["/my-app"]
```

Environment Variables

```
DATABASE_URL=postgres://... \
PORT=8080 \
RUST_LOG=info \
./my-app
```

Reverse Proxy (Nginx)

```
server {
    listen 80;
    server_name example.com;

    location / {
        proxy_pass http://localhost:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /static {
        alias /var/www/static;
        expires 1y;
    }
}
```

Performance Optimizations

1. **Compile-time Rendering**
    - Templates compile to Rust functions
    - Zero runtime parsing
    - Type-safe at compile time
2. **Automatic Optimizations**
    - CSS scoping and minification
    - HTML minification
    - Static asset hashing
    - Gzip compression
3. **Caching Headers**

    ```
    // Built-in caching for static assets
    Cache-Control: public, max-age=31536000
    ```

Why RHTML?

- **Simple**: Learn in 10 minutes
- **Fast**: Compiles to native code

- **Small**: ~5MB binary vs 200MB Node.js app
- **Type-safe**: Catch errors at compile time
- **Deployable**: Single binary runs anywhere
- **Familiar**: HTML + minimal directives
- **Functional**: Pattern matching, immutable data
- **Integrated**: Works with HTMX, Alpine, Tailwind

Summary

RHTML brings the elegance of modern frontend frameworks to Rust SSR development. With just 11 directives and a clear file structure, you can build production-ready web applications that compile to a single, deployable binary.

Start building with RHTML today!