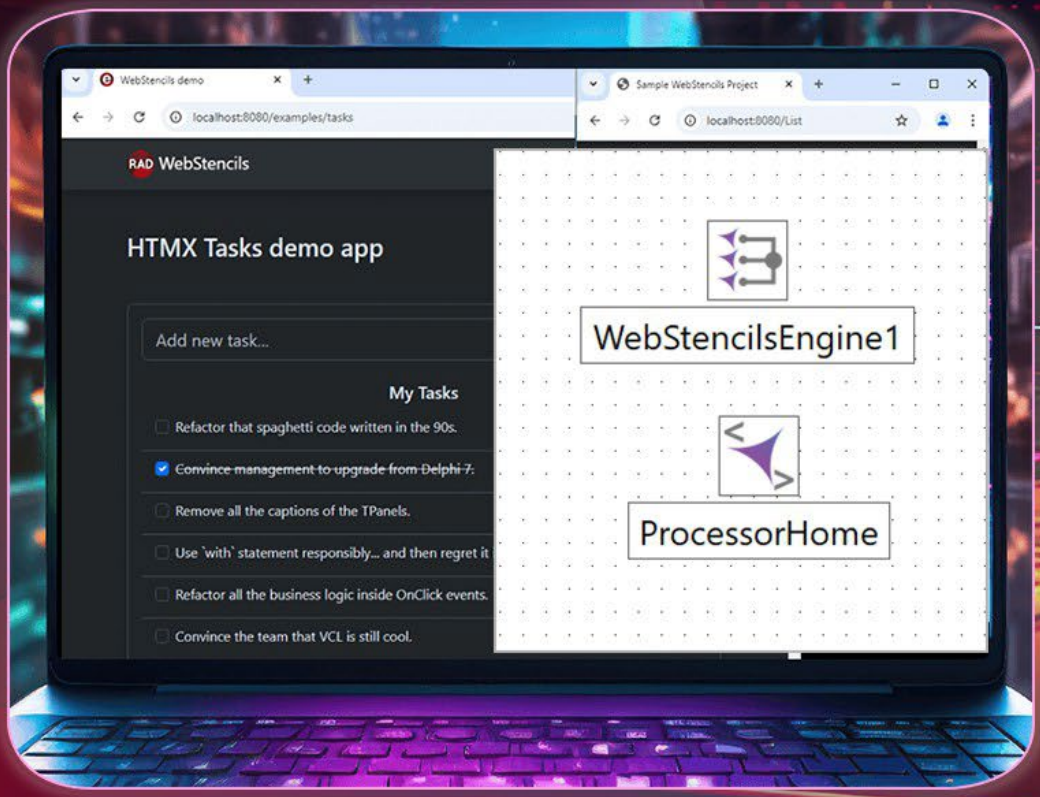
HTMX와 웹스텐실즈

빠르게 웹 개발하기 (RAD 스튜디오 사용)



Author
Antonio Zapater

embarcadero

HTMX와 웹스텐실즈

빠르게 웹 개발하기 (RAD 스튜디오 사용)

머리말

이 책의 초점은 최신식이고 간결한 웹 개발 방식을 위해 HTMX와 웹스텐실즈를 사용하는 방법을 알려주는 것이다.

HTMX는 동적인 웹 UI(사용자 인터페이스)를 구축하는데 사용되는 가벼운 자바스크립트 대안이다. 이것은 웹 개발자들에게 필수 솔루션이 되고 있다. 개발자들이 작성해야 하는 자바스크립트의 양을 크게 줄여주기 때문이다. 그래서 개발 과정이 더 빠르고 직관적이고, 코드 읽기 디버깅, 유지보수가 쉽다.

HTMX의 단순함은 RAD 스튜디오의 개발 철학인 애플리케이션을 빠르게 개발하기와 완벽하게 일치한다. 이것을 사용하면, 개발자는 애플리케이션 로직에 더 집중할 수 있다. 복잡한 프론트엔드 코드와의 씨름할 필요가 없기 때문이다.

웹스텐실즈의 아름다움은 템플릿-주도 아키텍처에 있다. 개발자가 새로 만들어 내지 않고 기존의 비즈니스 로직을 노출할 수 있도록, 재사용할 수 있고 사용자 정의가 가능한 템플릿들을 활용한다. 이 템플릿들은 기존 애플리케이션들과 자연스럽게 통합된다. 오래된 프로젝트를 웹으로 가져오는 데 드는 마찰이 줄어들기 때문에, 개발 속도가 높아질 뿐만 아니라 개발 팀 간의 협업이 향상된다. 그 결과, 개발자들은 기존 코드 기반들을 가지고 더 긴밀하게 작업할 수 있다.

이 책을 읽으면, HTMX와 웹스텐실즈의 힘을 활용하는 방법을 알게 된다. 그래서 수고가 더 적게 들고 유연성이 더 큰 현대적인 웹 애플리케이션을 개발할 수 있다. 여러분이 기존 데스크톱 애플리케이션을 웹으로 확대하든 아니면 새 동적 웹 프로젝트를 구축하든, 이 책을 통해 실용적인 통찰력을 얻을 수 있을 것이다. 이는 여러분이 RAD 스튜디오의 진화하는 웹 개발 생태계를 최대한 활용하는 데 도움이 된다.

RAD 스튜디오에 대해 더 자세히 알아보거나 또는 무료 평가판을 받아서 이 책의 예제들을 가지고 코딩을 해보려면 https://www.embarcadero.com/products/rad-studio를 방문하면 된다.

이제 이 기술들이 여러분의 작업 흐름을 어떻게 단순화하는지 그리고 여러분의 웹 개발 프로젝트를 어떻게 한 단계 끌어올릴 수 있는지 자세히 살펴보자!

WebStencils와 HTMX 코드 예시들은 이 가이드 문서에서 설명하고 있는 예시들과 비슷한 패턴을 따르고 있다. 그 예시들은 깃허브(GitHub) 저장소에서 받을 수 있다.
https://github.com/Embarcadero/WebStencilsDemos

01

HTMX 소개

HTMX는 무엇인가?

간략 개요

HTMX는 HTML을 확장한다. 그래서 하이퍼텍스트 매개체가 될 수 있도록 한다. 여러분은 HTMX를 사용해 AJAX 요청들을 만들고, CSS 전환을 발동하고, WebSocket들을 생성하고, 서버에서 보낸 이벤트(SSE, Server-Sent Event)들을 활용하도록 할 수 있다. HTML 엘리먼트 안에서 직접 서술하면 된다. 이 방식은 사용자 지정 자바스크립트에 대한 필요를 줄인다. 그리고 개발은 보다 더 선언적이고, HTML-중심적으로 진행된다.

HTMX의 핵심 특징들은 다음과 같다:

- **단순성**: HTMX는 HTML 애트리뷰트를 사용해 행위를 서술한다. 그래서 이해와 관리가 쉽다.
- **강력함**: 단순하면서도, HTMX는 정교한 상호 작용과 실-시간 업데이트가 가능하다.
- **유연성**: 어떠한 백-엔드 기술에서도 사용할 수 있고, 기존 시스템과도 잘 통합된다.
- **성능**: HTMX는 가볍고 빠르다. 그래서 적재 시간과 런타임 성능이 향상된다.

전통적 자바스크립트 및 AJAX와 비교

전통적인 웹 개발은 종종 상당한 양의 자바스크립트를 작성하는 방식으로 사용자 상호작용을 처리하고, AJAX 요청을 만들고, DOM을 업데이트한다. 이로 인해 복잡하고, 유지보수하기 어려운 코드가 생겨난다. 특히 애플리케이션이 규모와 복잡성이 커지면 더 심해진다.

HTMX는 이와 다른 방식을 사용한다:

- **선언 방식 vs 명령 방식**: 자바스크립트 함수를 작성하고, 내용을 가져오거나 업데이트하는 방식을 거기에 서술하는 것이 아니라, 여러분은 HTML 안에서 애트리뷰트를 사용해 직접 선언한다.
- **보일러플레이트가 줄어든다**: 공통된 패턴들 즉 AJAX 호출의 결과를 사용해 div를 업데이트하기와 같은 것들은 HTMX를 사용하는 경우, 최소한의 코드만 있으면 된다.
- **가독성 향상**: 엘리먼트의 동작을 해당 HTML 안에서 직접 볼 수 있다. 한눈에 더 쉽게 이해할 수 있다.
- **더 쉬운 유지보수**: 동작이 HTML 엘리먼트에 직접 묶여있기 때문에, HTMX-기반인 코드는 수정하거나 유지보수하기가 더 쉬운 경우가 많다.

아래 간단한 예시는 그 차이를 잘 보여준다:

전통적인 자바스크립트/AJAX:

```
<button id="loadButton">Load Content</button>
<div id="content"></div>

<script>
document.getElementById('loadButton').addEventListener('click', function()
    { fetch('/some-content')
        .then(response => response.text())
        .then(data =>
            { document.getElementById('content').innerHTML = data;
        });
});
</script>
```

HTMX:

```
<button hx-get="/some-content" hx-target="#content">Load Content</button>
<div id="content"></div>
```

보다시피, HTMX 버전이 훨씬 더 간결하다. 그리고 그 의도가 HTML 안에 그대로 드러나서 명확하다.

주요 개념들

효과적으로 HTMX를 사용하려면, 주요 개념들을 이해하고, 그것들이 서로 어떻게 함께 작동해서 동적인 웹 애플리케이션을 만들어 내는지를 알아야 한다.

hx-get: GET 요청을 통해 내용 가져오기

`hx-get` 애트리뷰트는 GET 요청들을 서버에게 보내고, 그에 대한 응답을 사용해 페이지를 업데이트 할 때 사용한다. `hx-get` 애트리뷰트가 붙은 엘리먼트에서 사용자가 상호작용을 하면 (디펄트는 on click임), HTMX는 명시된 URL로 가는 GET 요청을 만든다. 그리고 그 응답을 사용해 페이지를 업데이트한다.

예시:

```
<button hx-get="/api/user" hx-target="#user-info">
    Load User Info
</button>
<div id="user-info"></div>
```

이 예시에서, 버튼이 클릭되면 "`/api/user`"로 가는 GET 요청을 HTMX가 만든다. 그리고 해당 응답을 "`user-info`"라는 id를 가진 div 안에 넣는다.

hx-target: 응답이 들어갈 목표 엘리먼트 지정하기

앞의 예시에서 봤듯이, `hx-target` 애트리뷰트는 서버로부터 받은 응답이 어느 엘리먼트를 업데이트해야 하는 지를 명시한다. 지정되지 않은 경우, 기본 지정에 따라, 그 요청을 발동한 엘리먼트를 업데이트한다.

예시:

```
<button hx-get="/api/notification" hx-target="#notification-area">
    Check Notifications
</button>
<div id="notification-area"></div>
```

위 경우, "`/api/notification`"로부터 받은 응답은 id가 "`notification-area`"인 div 안에 삽입된다.

hx-post: POST 요청을 통해 데이터 제출하기

`hx-get`과 비슷하게, `hx-post` 애트리뷰트는 POST 요청들을 만들 때 사용된다. 전형적으로는, form 데이터를 제출할 때 즉 여러분의 서버에게 데이터를 보내서 그 서버의 상태를 변경하고자 할 때 쓰인다.

예시:

```
<form hx-post="/api/submit" hx-target="#response">
    <input type="text" name="username">
    <button type="submit">Submit</button>
</form>
<div id="response"></div>
```

사용자가 이 form을 제출할 때, HTMX는 "`/api/submit`"로 가는 POST 요청을 만들어 이 form 데이터를 전달한다. 그리고 서버에서 받은 응답을 #response div 안에 넣는다.

hx-put, hx-patch, hx-delete 요청들

이 기타 요청들 역시 `hx-get`, `hx-post`와 방식이 같다. 즉, `hx-put`, `hx-patch`, `hx-delete`는 각각 PUT, PATCH, DELETE 요청을 보낸다. 그래서 백엔드에 변경 또는 삭제를 제출하는데 사용된다. (보다 자세한 사항은 뒤에서 보여주겠다).

hx-swap: 내용을 뒤바꾸는 방식을 제어하기

`hx-swap` 애트리뷰트를 사용하면, 새 내용이 타겟 엘리먼트 안으로 들어가는 방식을 여러분이 제어할 수 있다. 가장 흔한 옵션들은 다음과 같다:

- `innerHTML` (디펄트): 그 타겟 엘리먼트의 내부 HTML을 교체한다
- `outerHTML`: 그 타겟 엘리먼트를 통째로 교체한다
- `beforebegin`: 그 타겟 엘리먼트 앞에 삽입한다
- `afterbegin`: 그 타겟 엘리먼트에서 가장 맨 앞에 있는 자식이 되도록 삽입한다
- `beforeend`: 그 타겟 엘리먼트에서 가장 뒤에 있는 자식이 되도록 삽입한다
- `afterend`: 그 타겟 엘리먼트 뒤에 삽입한다

예시:

```
<div id="list">
    <button hx-get="/api/item" hx-target="#list" hx-swap="beforeend">
        Add Item
    </button>
</div>
```

위 코드는 이 목록의 맨 뒤에 새 항목을 추가한다. 이 목록 전체를 교체하는 게 아니다.

추가적인 주요 개념들

위에서 본 네 가지 개념들이 HTMX의 기반이다. 하지만, 알아 두어야 할 몇 가지 중요한 기타 특징들이 있다:

1. `hx-trigger`: 그 AJAX 요청을 발동하는 이벤트가 무엇인지를 명시한다. 디펄트는, 대부분의 엘리먼트에서 Click 이벤트다. 또는 form에서는 Submit 이벤트다.
2. `hx-params`: 그 요청과 함께 제출되는 파라미터들이 무엇인지 제어한다.
3. `hx-headers`: 여러분은 그 AJAX 요청 안에 사용자 지정 헤더들을 추가할 수 있다.
4. `hx-vals`: 여러분은 요청과 함께 제출되는 파라미터들에게 추가적인 값들을 더할 수 있다.
5. `hx-boost`: 일반 앵커(anchor)들과 폼(form)들을 AJAX를 통해 강화할 수 있는 간단한 방식이다.
6. `hx-push-url`: 새 URL을 이력 스택에 추가한다. 그래서 브라우저에 전체 페이지 적재를 하지 않고도 그 브라우저의 URL을 업데이트 할 수 있다.

이 주요 개념들을 이해한다는 것은, 여러분이 HTMX를 사용해 동적이고 상호작용하는 웹 애플리케이션을 구축하는 견고한 기반을 갖추었다는 의미다. 이 기초들에 더 익숙해질 수록, 보다 수준 높은 HTMX를 활용해, 정교하고 반응성이 좋은 UI를 최소한의 자바스크립트 만으로 제공할 수 있게 된다.

HTMX에 대해 보다 깊이 있는 정보를 보려면, 공식 도움말 문서 참조: https://htmx.org/docs

02

WebBroker 소개

WebBroker는 무엇인가?

WebBroker는 RAD 스튜디오 안에 들어 있는 강력한 프레임워크다. 개발자들은 이 프레임워크를 사용해 견고한 웹 애플리케이션과 웹 서비스를 만들 수 있다. 이 프레임워크는 서버-쪽 애플리케이션을 만드는 단단한 기반을 제공한다. 또한 RESTful 서비스, SOAP 서비스 등등을 개발할 수 있도록 지원한다.

WebBroker의 주요 특징들

1. **다목적성**: WebBroker는 다양한 유형의 웹 서비스들 (stand-alone, Apache, ISAPI…)을 지원한다. 그래서 서로 다른 프로젝트 요구사항에 맞게 다양한 선택을 할 수 있다.
2. **통합**: RAD 스튜디오와 통합된다. 그래서 델파이와 C++빌더 개발자들은 자신들에게 익숙한 개발 환경을 사용해 구축할 수 있다.
3. **확장성**: WebBroker 애플리케이션들은 쉽게 확장될 수 있다. 따라서 증가하는 부하와 복잡한 요구 사항들을 다룰 수 있다.
4. **성능**: 이 프레임워크는 고성능을 위해 고안되었다, 성능은 서버-쪽 애플리케이션에서 매우 중요하다.

주요 개념들

WebBroker의 주요 개념들을 반드시 이해해야만, 효과적으로 웹스텐실즈를 사용해서 웹 애플리케이션과 웹 서비스를 구축할 수 있다.

컴포넌트들과 아키텍처

WebBroker 애플리케이션을 구축할 때는 핵심 컴포넌트 세트를 사용하게 된다. 이 컴포넌트들은 서로 함께 동작하면서 HTTP 요청들, 응답들, 라우팅, 미들웨어 등을 관리한다. 주요 컴포넌트들은 다음과 같다:

1. `TWebModule`: WebBroker 애플리케이션에서 중심이 되는 컴포넌트다. 이것은 다른 WebBroker 컴포넌트들의 컨테이너 역할을 한다. 그리고 그 애플리케이션의 전체 흐름을 처리한다.

2. `TWebDispatcher`: 이 컴포넌트는 들어오는 HTTP 요청들을 알맞은 액션 항목들에게 전달하는 것을 책임진다.

3. `TWebActionItem`: 이 클래스는 명시된 URL 엔드포인트들이 어떻게 다루어져야 하는지 정의한다. 각 액션 항목마다 특정 URL 패턴이 연계된다. 그리고 그 URL이 요청을 받았을 때 애플리케이션이 무슨 일을 수행해야 하는지를 정의한다.

WebBroker 애플리케이션의 아키텍처는 전형적인 흐름은 다음과 같다:

1. HTTP 요청이 들어온다
2. `TWebDispatcher`는 그 요청을 알맞은 `TWebActionItem`에게 보내준다
3. `TWebActionItem`은 자신에게 연계된 코드를 실행한다
4. 응답이 생성된다. 그리고 그 클라이언트에게 돌려 보내진다

WebBroker 애플리케이션 만들기

WebBroker 애플리케이션을 만들려면:

1. “File/New/Other…/Web/Web Server Application”을 선택한다.
2. 원한다면, Linux 호환성을 선택해도 된다.
3. 애플리케이션 유형을 고른다: standalone, Apache module 등등.
4. 8080 포트를 여러분의 컴퓨터 안에서 사용하고 있지 않다면, 디펄트를 그대로 두고 진행한다.

WebModule 액션들을 여러분의 라우터라고 생각해도 된다. 모든 엔드포인트들은 여기에 정의된다.

`TWebModule`을 구성하고 거기에 `TWebActionItem`들을 추가해 여러 URL 엔드포인트들을 다루도록 하려면, `TWebModule` 안 어느 곳이든 클릭하고 Properties 메뉴에서 Actions를 선택한다. 메뉴가 나타나면 그 안에서 새 엔드포인트들을 만들고 거기에 메서드를 연계한다. 대부분의 RAD 스튜디오 컴포넌트들이 그렇듯이, 이 액션 항목들 역시 사용할 수 있는 여러 가지 이벤트들을 가지고 있다.

아래의 간단한 예시를 통해 `TWebActionItem`을 어떻게 설정하는지 보자:

```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<html><body><h1>Hello, WebBroker!</h1></body></html>';
  Handled := True;
end;
```

이 액션 아이템은 자신에게 연계된 URL 요청이 오면, 간단한 HTML 응답을 생성한다.

요청들과 응답들을 처리하기

WebBroker는 HTTP 요청들과 응답들을 다루는 간단명료한 매커니즘을 제공한다:

- `TWebRequest`는 들어오는 HTTP 요청의 모든 정보에 접근한다. 여기에는 파라미터, 헤더, 요청 메서드 등도 포함된다.
- `TWebResponse`는 응답 데이터를 설정하는데 사용된다. 여기에는 내용, 상태 코드, 헤더 등도 포함된다.

요청을 접근하고 응답 데이터를 구성하는 예시

```
procedure TWebModule1.HandleGetUser(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  UserId: string;
begin
  UserId := Request.QueryFields.Values['id'];
  // Fetch user data based on UserId
  Response.ContentType := 'application/json';
  Response.StatusCode := 200;
  Response.Content := '{"id": "' + UserId + '", "name": "John Doe"}';
  Handled := True;
end;
```

배포와 확장

WebBroker 애플리케이션들은 유연한 배포 옵션들을 제공한다:

1. **Standalone Executables**: 독립 웹 서버 안에서 실행될 수 있다 (폼-기반 또는 CLI).
2. **Apache/ISAPI Extensions**: IIS, Apache와 같은 웹 서버에 통합될 수 있다.

이런 유연성 덕분에 여러분은 다양한 배포 상황에 맞출 수 있다. 그 결과 다양한 웹 서비스 유형에서 확장성과 성능을 보장한다. 처음에는 standalone executable로 시작해서 개발과 테스트를 하고, 실제 운영 환경용으로는 ISAPI 확장 또는 Apache 모듈로 배포해서 해당 웹 서버의 모든 기능을 사용할 수도 있다.

WebBroker 안 세션 관리

비록 WebBroker에는 세션 관리 기능이 내장되어 있지 않지만, 여러분의 WebBroker 애플리케이션 안에 견고한 세션 처리 시스템을 구현하는 것이 가능하다. 세션 관리는 사용자 상태를 여러 요청들에 걸쳐서 관리하는데 매우 중요하다. 그리고 종종 사용자 인증, 장바구니, CSRF 보호 등의 기능을 위해서도 필요하다.

세션 관리 방식들

세션 관리를 구현하는 방식은 몇 가지가 있다:

1. **인-메모리 세션**: 세션 데이터를 메모리 안에 저장한다. 이 방식은 빠르다. 하지만, 서버가 재시작되면 기존의 세션들이 유지되지 않는다. 또한 서버 인스턴스들 여러 개로 확장하기에 좋지 않다.
2. **데이터베이스 세션**: 세션 데이터를 데이터베이스 안에 저장한다. 이 방식은 지속성과 확장성을 제공한다. 하지만 시간 지연이 발생할 수 있다.
3. **파일-기반 세션**: 세션 데이터를 서버에 있는 파일 안에 저장한다. 이 방식은 지속성을 제공한다. 하지만, 성능 또는 잠금 이슈가 있을 수 있다. 많은 수의 동시 세션들이 있는 경우에 그렇다.
4. **분산 캐시 세션**: Redis와 같은 분산 캐시를 세션 저장소로 사용한다. 이 방식은 성능과 확장성의 좋은 균형을 제공한다.

인-메모리 세션 관리 구현하기

인-메모리 세션 관리의 기본적인 구현을 살펴보자:

```
unit SessionManagerU;

interface

uses
  System.Generics.Collections, System.SysUtils, Web.HTTPApp;

type
  TSessionData = class
  private
```

```
    FValues: TDictionary<string, string>;
  public
    constructor Create;
    destructor Destroy; override;
    property Values: TDictionary<string, string> read FValues;
  end;

  TSessionManager = class
  private
    FSessions: TDictionary<string, TSessionData>;
    function GenerateSessionId: string;
  public
    constructor Create;
    destructor Destroy; override;
    function GetSession(const SessionId: string): TSessionData;
    function CreateSession: string;
    procedure RemoveSession(const SessionId: string);
  end;

implementation

uses
  System.Hash;

{ TSessionData }

constructor TSessionData.Create;
begin
  inherited;
  FValues := TDictionary<string, string>.Create;
end;

destructor TSessionData.Destroy;
begin
  FValues.Free;
  inherited;
end;

{ TSessionManager }

constructor TSessionManager.Create;
begin
  inherited;
  FSessions := TDictionary<string, TSessionData>.Create;
end;

destructor TSessionManager.Destroy;
```

```
begin
  for var Session in FSessions.Values do
    Session.Free;
  FSessions.Free;
  inherited;
end;

function TSessionManager.GenerateSessionId: string;
begin
  var GUID := TGuid.NewGuid.ToString;
  Result := THashSHA2.GetHashString(GUID);
end;

function TSessionManager.GetSession(const SessionId: string): TSessionData;
begin
  if not FSessions.TryGetValue(SessionId, Result) then
    Result := nil;
end;

function TSessionManager.CreateSession: string;
var
  SessionId: string;
  SessionData: TSessionData;
begin
  SessionId := GenerateSessionId;
  SessionData := TSessionData.Create;
  SessionData.Values.AddOrSetValue('SessionId', SessionId);
  FSessions.Add(SessionId, SessionData);
  Result := SessionId;
end;

procedure TSessionManager.RemoveSession(const SessionId: string);
var
  SessionData: TSessionData;
begin
  if FSessions.TryGetValue(SessionId, SessionData) then
  begin
    SessionData.Free;
    FSessions.Remove(SessionId);
  end;
end;

end.
```

세션 관리를 WebModule 안으로 통합하기

세션 관리를 여러분의 WebBroker 애플리케이션 안에서 사용하려면, 기반 WebModule 클래스 하나를 만들고 그 안에 세션 처리를 넣으면 된다:

```
type
  TWebModuleWithSession = class(TWebModule)
    procedure WebModule1DefaultHandlerAction(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
  private
    FSessionManager: TSessionManager;
    function GetSessionId(Request: TWebRequest; Response: TWebResponse): string;
    function GetSession(Request: TWebRequest; Response: TWebResponse): TSessionData;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  end;

var
  WebModuleClass: TComponentClass = TWebModuleWithSession;

implementation

uses
  DateUtils;

{%CLASSGROUP 'System.Classes.TPersistent'}

{$R *.dfm}

constructor TWebModuleWithSession.Create(AOwner: TComponent);
begin
  inherited;
  FSessionManager := TSessionManager.Create;
end;

destructor TWebModuleWithSession.Destroy;
begin
  FSessionManager.Free;
  inherited;
end;

function TWebModuleWithSession.GetSessionId(Request: TWebRequest; Response:
TWebResponse): string;
const
  SessionCookieName = 'SessionId';
var
```

```
  LCookie: TCookie;
begin
  Result := Request.CookieFields.Values[SessionCookieName];
  if Result = '' then
  begin
    Result := FSessionManager.CreateSession;
    LCookie := Response.Cookies.Add;
    LCookie.Name := SessionCookieName;
    LCookie.Value := Result;
    LCookie.Path := '/';
    // For demo purposes, the Cookie is only valid for 1 minute
    LCookie.Expires := IncMinute(Now, 1);
  end;
end;

function TWebModuleWithSession.GetSession(Request: TWebRequest; Response: TWebResponse):
TSessionData;
var
  SessionId: string;
begin
  SessionId := GetSessionId(Request, Response);
  Result := FSessionManager.GetSession(SessionId);
end;
```

이렇게 설정하면, 세션 데이터를 WebModule의 액션 핸들러들 안에서 쉽게 접근할 수 있다:

```
procedure TWebModuleWithSession.WebModule1DefaultHandlerAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  LSession: TSessionData;
begin
  LSession := GetSession(Request, Response);
  // Use Session.Values to store or retrieve session data
  LSession.Values.AddOrSetValue('LastAccess', DateTimeToStr(Now));
  var HTMLResponse := '''
  <html>
  <head><title>Web Server Application</title></head>
  <body>Web Server Application</body>
  ''';
  HTMLResponse := HTMLResponse +
  '<p>Session ID: <strong>' + LSession.Values['SessionId'] + '</strong></p>' +
  '<p>Last Access: <strong>' + LSession.Values['LastAccess'] + '</strong></p>';
  HTMLResponse := HTMLResponse + '''
  </html>
  ''';
```

```
  Response.Content := HTMLResponse;
end;
```


Note

명심할 점이 있다. 이 SessionManager 구현은 PoC에 불과하다. 실제 운영을 위해서는, 구현이 더 필요하다. 예를 들어, 만료된 세션 삭제하기, TSessionData를 더 격리하기, SessionManage를 싱글톤으로 처리해 멀티-쓰레드 환경에 맞추기 등

보안 고려 사항들

CSRF 보호

크로스-사이트 요청 위조 (CSRF)는 승인되지 않은 명령을 웹 애플리케이션이 신뢰하는 사용자가 보내는 형태로 공격한다. CSRF로부터 보호하려면:

1. 요청-검증 토큰을 사용하라: 각 세션마다 고유한 토큰을 생성하고 그것을 폼 안에 포함한다.
2. GET이 아닌 모든 요청 각각에 대해 토큰에 대한 유효성 검사를 서버 상에서 수행한다.

WebBroker 안의 구현 예시:

```
unit CsrfProtection;

interface

uses
  System.SysUtils, Web.HTTPApp, SessionManager;

type
  TCsrfWebModule = class(TWebModule)
  private
    FSessionManager: TSessionManager;
    function GenerateCSRFToken: string;
  public
    procedure SendHTMLResponse(Sender: TObject;
      Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
    procedure ValidateCSRFToken(Request: TWebRequest);
    // The business logic related to session is the same as in the previous examples
```

```
    function GetSessionId(Request: TWebRequest; Response: TWebResponse): string;
    function GetSession(Request: TWebRequest; Response: TWebResponse): TSessionData;
  end;

implementation

uses
  System.Hash;

function TCsrfWebModule.GenerateCSRFToken: string;
begin
  var GUID := TGuid.NewGuid.ToString;
  Result := THashSHA2.GetHashString(GUID);
end;

procedure TCsrfWebModule.SendHTMLResponse(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  CSRFToken: string;
  Session: TSessionData;
begin
  Session := GetSession(Request, Response);
  CSRFToken := GenerateCSRFToken;
  LSession.Values.AddOrSetValue('CSRFToken', LCSRFToken);
  Response.Content :=
    '<form hx-post="/submit">' +
    '<input type="hidden" name="csrf_token" value="' + CSRFToken + '">' +
    // ... rest of your form ...
    '</form>';
  Handled := True;
end;

procedure TCsrfWebModule.ValidateCSRFToken(Request: TWebRequest);
var
  RequestToken, SessionToken: string;
  Session: TSessionData;
begin
  Session := GetSession(Request, Response);
  RequestToken := Request.ContentFields.Values['csrf_token'];
  SessionToken := Session.Values['CSRFToken'];

  if (RequestToken = '') or (SessionToken = '') or (RequestToken <> SessionToken) then
    raise Exception.Create('Invalid CSRF token');
end;

end.
```

이 구현 안에서:

1. 우리는 각 폼 제출 시 마다 고유한 CSRF 토큰을 생성한다.
2. 이 토큰은 해당 세션 안에 저장된다. 그리고 폼 안에 숨김 필드로 포함된다.
3. 폼 제출을 처리할 때, 여러분은 그 요청에서 보낸 토큰과 세션 안에 저장된 것을 대조해 유효성을 검증한다.

데이터 검증

서버 쪽에서 항상 데이터의 유효성을 검증하고 정제해야 한다. 클라이언트 쪽에 유효성 검사 있어도 말이다.

예시:

```
procedure TWebModule1.ValidateUserInput(const Username: string);
begin
  if Length(Username) < 3 then
    raise Exception.Create('Username must be at least 3 characters long');

  if not TRegEx.IsMatch(Username, '^[a-zA-Z0-9_]+$') then
    raise Exception.Create('Username can only contain letters, numbers, and
underscores');
end;

procedure  TWebModule1.HandleUserRegistration(Sender:  TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Username: string;
begin
  Username := Request.ContentFields.Values['username'];
  try
    ValidateUserInput(Username);
    // Process registration...
    Response.Content := 'Registration successful';
  except
    on E: Exception do
      Response.Content := 'Registration failed: ' + E.Message;
  end;
  Handled := True;
end;
```

크로스-사이트 스크립팅 (XSS)

XSS 공격을 방지하려면, 사용자가 생성한 내용을 항상 이스케이프 즉 인코딩한 후 여러분의 HTML 응답에 넣어야 한다. NetEncoding 라이브러리에는 HTML 문자열을 이스케이프 하는 다양한 방법들이 들어 있다.

예시:

```
function HTMLEncode(const S: string): string;
begin
  Result := TNetEncoding.HTML.Encode(S);
end;

procedure TWebModule1.DisplayUserComment(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  var UserComment := Request.QueryFields.Values['comment'];
  Response.Content := '<div class="comment">' + HTMLEncode(UserComment) + '</div>';
   Handled := True;
end;
```

기타 보안 고려 사항들

1. **SQL 주입 방지**: 데이터베이스와 상호작용할 때는 파라미터화된 쿼리 또는 미리 준비된 문장을 사용하라.
2. **안전한 통신**: HTTPS를 사용해 전송 중인 데이터를 암호화하라.
3. **권한 부여와 인증**: 견고한 사용자 인증 그리고 올바른 접근 제어를 구현하라.

이러한 보안 조치들을 구현하면, 여러분의 WebBroker 애플리케이션의 보안을 크게 향상시킬 수 있다. 보안은 지속적인 과정이며, 최신 보안 모범 사례와 취약점에 대해 계속 업데이트하는 것이 중요하다는 점을 잊지 말자.

더 자세한 정보는 docwiki 참고: Using WebBroker

03

여러분의 첫 웹 앱을 WebBroker를 사용해 개발하기

도입

이 장에서, 우리는 여러분의 첫 웹 애플리케이션을 WebBroker를 사용해 만드는 과정을 따라간다. 간단한 "Hello World" 예제부터 시작한다. 그리고 나서 간단한 할 일 목록 앱 만들기를 진행한다.

"Hello World" 애플리케이션 만들기

간단한 "Hello World" 예제부터 시작해 보자. HTMX를 WebBroker와 어떻게 함께 사용하는지 알 수 있다.

1. 가장 먼저, RAD 스튜디오 안에서 새 WebBroker Application을 만든다.
2. 여러분의 WebModule 안에서, 새 WebActionItem을 추가하고 이름을 "ActionHello"라고 지정한다.
3. ActionHello 항목을 더블-클릭한다. 그러면 이벤트 핸들러가 만들어진다.
4. 그 이벤트 핸들러를 아래와 같이 구현한다:

```
procedure TWebModule1.WebModule1ActionHelloAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '''
    <html>
      <head>
        <script src="https://unpkg.com/htmx.org@2.0.2"></script>
      </head>
      <body>
        <h1>Hello, WebBroker and HTMX!</h1>
        <button hx-get="/greet" hx-target="#greeting">Say Hello</button>
        <div id="greeting"></div>
      </body>
    </html>
    ''';
  Handled := True;
end;
```

5. WebActionItem을 하나 더 추가하고 이름을 "ActionGreet"라고 지정한 후 이벤트 핸들러를 구현한다:

```
procedure TWebModule1.WebModule1ActionGreetAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<p>Hello from the server!</p>';
  Handled := True;
end;
```

6. 이제 여러분의 애플리케이션을 실행한다. 웹 브라우저에서 열고 "Say Hello" 버튼을 클릭한다. 그러면 인사말이 표시되는 것을 볼 수 있다. 페이지 전체가 다시 적재되지는 않는다.

이 간단한 예제는 WebBroker가 어떻게 HTML 내용을 제공하는지 그리고 동적 요청을 서버에 보내려면 어떻게 HTMX를 사용하는지 보여준다.

간단한 할 일 목록 앱

이제 조금 더 복잡한 애플리케이션을 만들어 보자: 간단한 할 일 목록 앱을 만들어 사용자가 할 일을 넣거나 볼 수 있도록 해보자.

1. 새 WebBroker Application을 만든다. 또는 앞에서 만든 애플리케이션을 사용한다.
2. 여러분의 프로젝트 안에 새 유닛을 추가한다. 거기에 할 일 목록을 저장할 것이다:

```
unit TodoList;

interface

uses
  System.Generics.Collections;

type
  TTodoItem = record
    Id: Integer;
    Text: string;
  end;

  TTodoList = class
  private
    FItems: TList<TTodoItem>;
    FNextId: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    function AddItem(const Text: string): Integer;
    function GetItems: TArray<TTodoItem>;
  end;

implementation

{ TTodoList }

constructor TTodoList.Create;
begin
  FItems := TList<TTodoItem>.Create;
  FNextId := 1;
end;

destructor TTodoList.Destroy;
begin
  FItems.Free;
  inherited;
end;

function TTodoList.AddItem(const Text: string): Integer;
var
  Item: TTodoItem;
begin
  Item.Id := FNextId;
  Item.Text := Text;
  FItems.Add(Item);
```

```
  Result := FNextId;
  Inc(FNextId);
end;

function TTodoList.GetItems: TArray<TTodoItem>;
begin
  Result := FItems.ToArray;
end;

end.
```

3. 여러분의 WebModule 안에서, 이 TodoList를 위한 필드를 추가한다:

```
FTodoList: TTodoList;
```

이 필드를 WebModule의 생성자 안에서 초기화하고 소멸자 안에서 해제한다.

4. WebActionItem들을 추가한다. 할 일 목록을 표시하고, 새 항목을 추가하고, 목록을 업데이트하는 것들이다. 그 이벤트 핸들러들은 다음과 같다:

```
procedure TWebModule1.WebModule1ActionTodoListAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  Html: string;
  Item: TTodoItem;
Begin
  Html := '''
    <html>
      <head>
        <script src="https://unpkg.com/htmx.org@2.0.2"></script>
      </head>
      <body>
        <h1>Todo List</h1>
        <form hx-post="/add-todo" hx-target="#todo-list">
          <input type="text" name="todo-text" placeholder="New todo item">
          <button type="submit">Add</button>
        </form>
        <div id="todo-list">
    ''';

  for Item in FTodoList.GetItems do
  begin
    Html := Html + Format('<p>%d: %s</p>', [Item.Id, Item.Text]);
```

```
  end;

  Html := Html + '''
        </div>
      </body>
    </html>
    ''';

  Response.Content := Html;
  Handled := True;
end;

procedure TWebModule1.WebModule1ActionAddTodoAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
  TodoText: string;
  Html: string;
  Item: TTodoItem;
begin
  TodoText := Request.ContentFields.Values['todo-text'];
  FTodoList.AddItem(TodoText);

  Html := '';
  for Item in FTodoList.GetItems do
  begin
    Html := Html + Format('<p>%d: %s</p>', [Item.Id, Item.Text]);
  end;

  Response.Content := Html;
  Handled := True;
end;
```

5. 여러분의 애플리케이션을 실행한다. 여러분은 이제 새 할 일을 추가할 수 있다. 할 일 목록은 동적으로 업데이트 된다. 페이지 전체가 다시 적재되는 것이 아니라, 목록만 업데이트 된다.

이 간단한 할 일 목록 앱은 WebBroker를 사용해 여러 가지 요청 유형들(GET을 통해 목록 표시, POST를 통해 항목 추가)을 어떻게 다루는 지를 보여준다. 또한 HTMX 사용해 서버와 상호작용하는 동적인 UI를 어떻게 생성자는 지도 보여준다.

실전 애플리케이션 안에서라면 해야 할 것들이 더 있다는 점을 알아두자. 여러분은 에러 처리를 추가하고, 입력이 유효한지 검사하고, 아마도 할 일 목록을 데이터베이스에 보존하고 싶을 것이다.하지만, 이 예제는 여러분의 이해를 돕는 좋은 시작점이다. 대화형 웹 애플리케이션을 생성하기 위해 WebBroker와 HTMX가 어떻게 함께 동작하는지 이해할 수 있었을 것이다.

04

HTMX 사용시 고급 애트리뷰트(attribute)들과 보안

도입

이 장에서, 우리는 HTMX 안에 있는 보다 수준 높은 애트리뷰트들 몇 가지를 살펴본다. 그리고 HTMX와 WebBroker를 사용해 웹 애플리케이션을 구축할 때 고려해야 할 중요한 보안 사항들을 논한다.

고급 애트리뷰트(attribute)들

HTMX에는 풍부한 애트리뷰트들로 구성된 세트가 들어 있다. 그래서 AJAX 요청들과 DOM 업데이트들을 여러분이 정밀하게 제어할 수 있다:

hx-put 그리고 hx-delete: PUT과 DELETE를 통해 요청을 제출하기

`hx-get` 그리고 `hx-post`가 가장 많이 사용되긴 하지만, HTMX는 다른 HTTP 메서드들 즉 PUT, DELETE 등도 지원한다.

`hx-put`을 사용하는 예시:

```
<button hx-put="/api/user/1" hx-target="#user-info">
    Update User
</button>
```

`hx-delete`를 사용하는 예시:

```
<button hx-delete="/api/user/1" hx-target="#user-list">
    Delete User
</button>
```

여러분의 WebBroker 애플리케이션 안에서, 여러분은 이 요청들을 올바르게 처리해야 한다:

```
procedure TWebModule1.HandlePutRequest(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  if Request.MethodType = mtPUT then
  begin
    // Handle PUT request
    Response.Content := 'User updated';
    Handled := True;
  end;
end;

procedure TWebModule1.HandleDeleteRequest(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  if Request.MethodType = mtDELETE then
  begin
    // Handle DELETE request
    Response.Content := 'User deleted';
    Handled := True;
  end;
end;
```

hx-trigger: 이벤트 트리거들을 커스터마이징 하기

`hx-trigger` 애트리뷰트를 사용하면 여러분은 어떤 이벤트가 그 AJAX 요청을 발동하는지 지정할 수 있다. 디펄트는, 대부분의 엘리먼트에서 Click 이벤트다. 또는 form에서는 Submit 이벤트다.

예시:

```
<input type="text"
       name="search"
       hx-get="/search"
       hx-trigger="keyup changed delay:500ms"
       hx-target="#search-results">
```

이 문장은 검색 요청을 보낸다. 그런데, 사용자가 타이핑을 멈추고 500 밀리 초가 지난 후에 발동한다.

hx-select: 서버 응답의 일부를 선택하기

`hx-select` 애트리뷰트는 서버의 응답 중 하위 집합을 선택하는 기능으로써, 타겟을 업데이트 할 때 사용할 수 있다.

```
<button hx-get="/api/user"
        hx-target="#user-name"
        hx-select="#name">
    Load User Name
</button>
```

예시:

이 경우, 서버에서 온 응답 중 id가 "name"인 엘리먼트만 사용해 해당 타겟을 업데이트 한다.

이 방식을 사용하면 여러분의 서버가 전체 HTML 페이지들을 반환할 수 있게 해준다 (HTMX가 아닌 요청들인 경우이거나 디버깅을 하는 경우에 유용할 것이다). 그러면서도 동시에 HTMX를 사용해 페이지에서 원하는 부분들만 골라서 업데이트 할 수 있다.

알아 두어야 할 점이 있다. HTML 문서 전체를 반환하는 것이 가능하고 가끔 유용하기도 하지만, HTMX로 작업하는 많은 경우에, 여러분은 오직 필요한 특정 HTML 조각만을 여러분의 서버가 반환하는 방식을 선택하는 경우가 많을 것이다.

hx-include: 부가적인 데이터를 요청 안에 포함하기

`hx-include` 를 사용하면 다른 엘리먼트에 있는 값을 여러분의 요청 안에 포함시킬 수 있다.

예시:

```
<form hx-post="/api/submit" hx-include="#extra-data">
    <input type="text" name="username">
    <button type="submit">Submit</button>
</form>
<input type="hidden" id="extra-data" name="extra" value="some-value">
```

이 문장은 "extra-data"라는 HTML input 엘리먼트의 값을 폼 제출 안에 포함시킨다.

hx-push-url: 브라우저의 URL을 업데이트하기

`hx-push-url`를 사용하면 여러분은 브라우저에 전체 페이지 적재를 하지 않고도 그 브라우저의 URL을 업데이트 할 수 있다. 단일-페이지 애플리케이션에서 이동을 관리할 때 유용하다.

예시:

```
<button hx-get="/new-page"
        hx-push-url="true">
    Go to New Page
</button>
```

이것은 버튼이 클릭되면 브라우저의 URL을 업데이트 한다. 그래서 앞/뒤로 이동이 올바르게 되도록 한다.

05

웹스텐실즈(WebStencils) 소개

웹스텐실즈(WebStencils)는 무엇인가?

웹스텐실즈는 서버-쪽의 새 스크립트-기반 통합으로써, HTML 파일들을 처리하는 기술이다. RAD 스튜디오 12.2에서 도입되었다. 개발자들은 이를 통해 현대적이고 전문적인 모양을 갖춘 웹사이트를 개발할 수 있다. 그 웹사이트의 기반으로는 어떠한 자바스크립트든 사용할 수 있다. 또한 RAD 스튜디오로 만든 서버-쪽 애플리케이션을 통해 추출하고 처리한 데이터를 그 웹사이트에 반영할 수 있다.

핵심 개념

웹스텐실즈는 탐색 및 대화형 웹사이트를 개발할 수 있도록 한다. 블로그, 온라인 카탈로그, 온라인 주문 시스템, 사전과 같은 참고 자료 사이트, 위키 등이 모두 탐색 웹사이트의 예다. 웹스텐실즈는 템플릿을 제공하는데, ASP.NET의 Razor 처리와 닮았다. 하지만, 특별히 RAD 스튜디오용으로 고안된 것이다.

HTMX와 통합하기

웹스텐실즈는 HTMX와 서로 잘 도와준다. HTMX 페이지는 서버-쪽 코드 생성이 주는 혜택을 활용하고, REST 서버에 연결해 내용을 업데이트할 수 있다는 이득을 얻을 수 있다. 한편, 델파이 웹 기술들은 페이지 생성과 REST API 제공할 때 HTMX를 통해 그 품질을 높일 수 있다.

CSS와 JS를 가리지 않음

웹스텐실즈의 핵심 특징 중 하나는 특정 자바스크립트 또는 CSS 라이브러리를 사용하도록 강제하지 않는다는 점이다. 이것은 순전히 템플릿 엔진이다. 그리고 서버-쪽 렌더링만 담당한다. 따라서, 여러분은 어떠한 프론트-엔드 기술이든 선호하는 것을 사용하면 된다.

웹스텐실즈 문법 구조

웹스텐실즈는 간단한 문법 구조(syntax)를 가지고 있다. 그 기반은 이 주요 요소 두 개다:

1. @ 기호
2. 블록을 위한 중괄호 {}

@ 기호

@ 기호는 특별한 표시자로 웹스텐실즈 안에서 사용된다. 그 뒤에는 다음과 같은 것들이 붙을 수 있다:

- 오브젝트 이름 또는 필드 이름
- 특별한 처리를 하는 키워드
- 또다른 @

예를 들어:

```
@object.value
```

이 문법 구조를 사용하면, 이 'object'의 'value' 프로퍼티에 접근할 수 있다. 이 오브젝트 이름은 심볼로 사용되는 로컬 이름이다. 이것은 실제 서버 애플리케이션 안에서, 서버 오브젝트와 짝을 이룰 수 있다. 또는 서버- 쪽 코드 안에서 `OnValue` 이벤트 핸들러를 수행하는 동안 인식될 수 있다.

블록을 위한 중괄호 {}

중괄호는 조건 블록 또는 반복 블록을 표시할 때 사용된다. 이것들은 특정 웹스텐실즈 조건문 뒤에서 있는 경우에만 처리된다.

점 표기법을 사용해 값을 접근하기

이 예시는 값들이 웹스텐실즈 안에서 어떻게 처리되는지 보여준다:

```
<h2>User Profile</h2>
```

```
<p>Name: @user.name</p>
<p>Email: @user.email</p>
```

웹스텐실즈 키워드들과 예시들

다양한 웹스텐실즈 키워드들을 파악할 수 있도록, 예시들을 함께 보자:

@page

`@page` 키워드를 통해, 접근 연결 뿐만 아니라 해당 페이지의 여러 프로퍼티들에 접근할 수 있다.

예시:

```
<p>Current page is: @page.pagename</p>
```

@query

The `@query` 키워드를 통해, HTTP 쿼리 파라미터들을 읽을 수 있다.

예시:

```
<p>You searched for: @query.searchTerm</p>
```

이 예시에서, searchTerm은 URL 안에 포함된 파라미터다:
`yourdomain.com?searchTerm="mySearch"`

주석 (@* .. *@)

웹스텐실즈 안에서 주석은 `@* *@` 로 감싼다. 이것은 결과 HTML 안에 들어가지 않고 생략된다.

예시:

```
@* This is a comment and will not appear in the output *@
<p>This will appear in the output</p>
```

@if와 @else

조건 실행은 `@if`와 `@else`를 사용해 다룬다

예시:

```
@if user.isLoggedIn {
    <p>Welcome, @user.name!</p>
}
@else {
    <p>Please log in to continue.</p>
}
```

@if not

부정 조건 실행은 `@if not`을 사용한다.

예시:

```
@if not cart.isEmpty {
    <p>You have @cart.itemCount items in your cart.</p>
}
@else {
    <p>Your cart is empty.</p>
}
```

@ForEach

`@ForEach` 키워드는 열거자 안에 있는 엘리먼트들을 순환할 때 사용한다.

예시:

```
<ul>
@ForEach (var product in productList) {
    <li>@product.name - @product.price</li>
}
</ul>
```

맺음말

웹스텐실즈는 동적인 웹 페이지를 RAD 스튜디오 애플리케이션 안에 생성하는 강력한 방법을 제공한다. 그 문법 구조를 사용해 여러분은 서버-쪽 로직을 여러분의 HTML 템플릿들 안에 자연스럽게 통합할 수 있다. @ 기호, 중괄호, 기타 다양한 키워드들을 통해, 여러분은 동적이고 인터렉티브한 웹 페이지들을 쉽게 생성할 수 있다.

웹스텐실즈에 더 익숙해질 수록, 여러분은 그 모든 잠재력을 활용할 수 있게 된다. 그래서 복잡하고 반응성이 좋은 웹 애플리케이션을 만들 수 있다. 뒤에 이어지는 장들을 통해, 템플릿, 배치 등 웹스텐실즈의 보다 수준 높은 특징들을 알 수 있을 것이다. 또한 웹스텐실즈 컴포넌트들을 효과적으로 사용하는 방법도 알게 된다.

06

컴포넌트들 및 배치 옵션들

도입

이 장에서, 우리는 웹스텐실즈의 핵심 컴포넌트들을 알아본다. 그리고 템플릿들과 레이아웃들을 사용해보고 공통된 템플릿 패턴들을 논의한다. 이 개념들을 이해하면, 여러분은 보다 잘 정돈되고 유지보수하기 좋고, 재사용하기 좋은 웹 애플리케이션을 웹스텐실즈를 사용해 만들 수 있다.

웹스텐실즈 컴포넌트들

웹스텐실즈는 두 가지 주요 컴포넌트들을 도입한다: 웹스텐실즈 엔진 그리고 웹스텐실즈 처리기다. 이것들은 서로 함께 동작하면서 여러분의 템플릿을 처리하고 여러분이 원하는 최종 HTML 출력을 생성한다.

웹스텐실즈 엔진

웹스텐실즈 엔진는 중심 컴포넌트다. 이것은 여러분의 템플릿 처리를 전반적으로 관리한다. 주로 사용되는 상황은 두 가지다:

1. **WebStencilsProcessor 컴포넌트들에게 연결되어 사용된다**: 이렇게 설정하면, 엔진은 여러 처리기들에게 공유 설정들과 동작들을 제공한다. 그래서 각 처리기마다 개별적으로 사용자 정의를 할 필요를 줄여준다.

2. **독립적으로 사용된다**: 엔진은 WebStencilsProcessor 컴포넌트들을 필요할 때 생성할 수 있다. 그래서 여러분의 웹 모듈 안에는 엔진 컴포넌트만 있어도 된다.

`TWebStencilsEngine`에 들어 있는 핵심 프로퍼티들과 메서드들:

- `Dispatcher`: 텍스트 파일들의 사후-처리 위한 파일 디스패처(`IWebDispatch`를 구현한 것)를 명시한다.
- `PathTemplates`: 요청 경로 템플릿들의 모둠이다. 요청들을 알맞게 매칭하고 처리하는데 사용된다.
- `RootDirectory`: 파일 시스템의 최상위 경로를 명시한다. 이것을 기준으로 상대 경로가 파악된다.
- `DefaultFileExt`: 디펄트 파일 확장자를 지정한다 (디펄트는 '.html').
- `AddVar`: 처리기들이 오브젝트를 사용할 수 있도록 스크립트 변수들의 목록에 추가한다.
- `AddModule`: 오브젝트 하나를 스캔해서 그 안에 있는 멤버들 중에서 `[WebStencilsVar]` 애트리뷰트 표기가 있는 것들을 찾아 그것들을 스크립트 변수로 추가한다.

웹스텐실즈 처리기

웹스텐실즈 처리기는 개별 파일(전형적으로 HTML)들 그리고 그것들에 연계된 템플릿들을 처리하는 일을 담당한다. 처리기는 독립적으로 사용되거나 또는 웹스텐실즈 엔진에 의해 생성되고 관리될 수 있다.

`TWebStencilsProcessor`에 들어 있는 핵심 프로퍼티들과 메서드들:

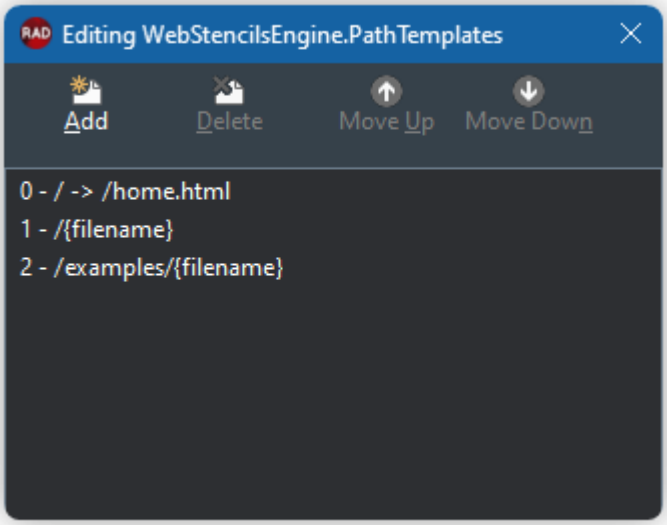
- `InputFilename`: 처리할 파일을 명시한다.
- `InputLines`: 처리할 내용을 직접 할당한다.
- `Engine`: 데이터 변수들, 이벤트 핸들러들, 등등을 상속해 줄 엔진을 명시한다 (선택 사항).
- `Content`: 최종 처리된 내용을 제공한다.
- `AddVar`: 처리기들이 오브젝트를 사용할 수 있도록 스크립트 변수들의 목록에 추가한다.

TWebStencilsEngine과 WebBroker

웹스텐실즈 엔진 컴포넌트는 `TWebFileDispatcher`와 쉽게 연결된다. 이를 통해 WebBroker는 자동으로 템플릿들을 제공할 수 있다.

이것을구현했다면, PathTemplates 프로퍼티 안에서 와일드카드를 사용할 수 있다. 이를 통해 자동으로 요청들을 파일에 맵핑할 수 있다.

예시:



정의된 각 경로를 분석해보자:

`0-/->/home.html`: 그 웹사이트의 최상위 경로에 대한 접근을 `home.html` 템플릿에게 보낸다.

`1-/{filename}`: URI를 파일 이름에 맵핑하려고 시도한다. 예를 들어, https://localhost:8080/basics 인 경우에는 사전에 정해진 템플릿 폴더 안에서 파일 이름이 `basics.html`인 템플릿을 찾는다.

`2-/examples/{filename}`: 바로 앞의 예시와 같은 동작을 한다. 하지만 이 경우에는 `/examples` 경로 안에서 템플릿을 찾는다.

AddVar를 사용해 데이터 추가하기

`AddVar` 메서드는 여러분의 델파이 코드에서 여러분의 웹스텐실즈 템플릿에게 데이터를 보내는데 매우 중요한 역할을 한다. AddVar를 사용하는 몇 가지 방식을 보자:

1. **오브젝트를 직접 할당하기**:

```
WebStencilsProcessor1.AddVar('user', UserObject);
```

2. **익명 메서드를 사용하기**:

```
WebStencilsProcessor1.AddVar('products',
```

```
  function: TObject
  begin
    Result := GetProductList;
  end);
```

3. **애트리뷰트를 사용하기**: 클래스 안에 있는 필드, 프로퍼티, 또는 메서드에 `[WebStencilsVar]` 애트리뷰트를 사용해 표시를 할 수 있다. 그리고 나서, `AddModule`을 사용하면 표시가 달린 모든 멤버들이 스크립트 변수로 추가된다:

```
type
  TMyDataModule = class(TDataModule)
    [WebStencilsVar]
    FDMemTable1: TFDMemTable;
    [WebStencilsVar]
    function GetCurrentUser: TUser;
  end;

// In your WebModule:
WebStencilsProcessor1.AddModule(DataModule1);
```

중요: 명심하자. 오직 `GetEnumerator` 메서드(오브젝트의 값을 반환한다)를 가진 오브젝트에서만 작동한다. 레코드는 웹스텐실즈에서 제공하지 못한다.

배치 및 내용 자리표시자들

웹스텐실즈는 강력한 배치 시스템을 제공한다. 이는 다른 템플릿 엔진들 즉 Mustache, Blade, ERB, Razor 들과 비슷하다. 이 시스템을 통해, 여러분은 여러분의 페이지들을 위한 공통 구조를 정의할 수 있다. 그리고 특정 내용을 그 구조 안으로 넣을 수 있다.

@RenderBody

`@RenderBody` 지시어는 배치 템플릿 안에서 사용된다. 그래서 특정 페이지에서 온 내용이 삽입되어야 하는 곳이 어디인지를 알려준다. 예를 들어 우리의 공통 HTML 구조가 다음과 같고 그 파일 이름이 BaseTemplate.html이라면:

```
<!-- This is the BaseTemplate.html -->
<!DOCTYPE html>
<html>
<head>
    <title>My Website</title>
</head>
<body>
    <header>
        <!-- Common header content -->
    </header>

    <main>
        @RenderBody
    </main>

    <footer>
        <!-- Common footer content -->
    </footer>
</body>
</html>
```

`@Renderbody` 키워드는 다른 자식 템플릿들 안에 들어가게 될 내용들에 의해 교체된다.

@LayoutPage

`@LayoutPage` 지시어는 내용 페이지 안에서 사용된다. 그래서 그 페이지의 구조가 어느 배치 템플릿을 따를 것인지를 명시한다. 이 문장은 전형적으로 내용 파일의 맨 위에 놓인다:

```
<!-- BaseTemplate.html will be used as a base and the rest of the content included where
the @RenderBody tag is located -->
@LayoutPage BaseTemplate
<h2>Welcome to My Page</h2>
<p>This is the content of my page.</p>
```

이 예시에서, 기반 배치는 BaseTemplate.html 파일이 된다. 그리고 `@LayoutPage` 키워드 아래에 있는 내용은 우리가 바로 앞에서 본 예시에 정의해 놓았던 `@RenderBody` 자리에 렌더링된다.

@Import

`@Import` 지시어를 통해 여러분은 외부 파일을 현재 템플릿 안의 특정 위치로 병합할 수 있다. 이것은 재활용할 수 있는 구성요소들을 만들어 쓸 때 유용하다.

이 지시어를 통해, 여러분은 중첩된 폴더들 안에 있는 템플릿들을 잘 구조화할 수 있다. 또한 그 파일의 확장자가 엔진 또는 프로세서 안에 디펄트로 정의되어 있는 경우에는 파일 확장자를 생략할 수도 있다.

```
@Import Sidebar.html

@* Same Behavior *@
@Import Sidebar

@* Nested folder example *@
@Import folder/Sidebar
```

@ExtraHeader와 @RenderHeader

`@ExtraHeader` 지시어를 통해, 여러분의 HTML 문서 구역 안에 넣어야 하는 추가 내용들을 정의할 수 있다. 이것은 그 페이지만을 위한 CSS 또는 자바스크립트 파일을 포함시켜야 하는 경우에 특히 유용하다.

동작 방식은 이렇다:

1. 내용 페이지 안에서, `@ExtraHeader`를 사용해 추가 헤더 내용을 정의한다.
2. 배치 템플릿 안에서, `@RenderHeader`를 사용해 이 추가 헤더 내용이 삽입되어야 하는 위치를 명시한다.

예시를 보자:

내용 페이지 (ProductPage.html):

```
@LayoutPage BaseTemplate
@ExtraHeader {
    <link rel="stylesheet" href="/css/product-page.css">
    <script src="/js/product-details.js"></script>
}

<h1>Product Details</h1>
<div id="product-info">
    <!-- Product information here -->
</div>
```

배치 템플릿 (BaseTemplate.html):

```
<!DOCTYPE html>
<html>
<head>
    <title>My Online Store</title>
    <link rel="stylesheet" href="/css/main.css">
    @RenderHeader
</head>
<body>
    <main>
        @RenderBody
    </main>
</body>
</html>
```

이 예시에서, 상품 상세 페이지는 추가 CSS와 자바스크립트 파일을 정의해서, 이 페이지에서 사용되도록 하고 있다. 배치 템플릿 안에 있는 `@RenderHeader` 지시어는 이 추가 리소스들이 최종 HTML 출력에 포함되도록 보장한다.

이 방식을 통해, 여러분은 공통되는 기반 템플릿을 가질 수 있다. 그러면서도 개별 페이지에 각자 필요한 리소스 또는 메타 태그들을 필요한 대로 추가할 수 있다.

알아둘 점이 있다. 여러분은 여러 개의 `@ExtraHeader` 블록들을 내용 페이지 안에 필요한 대로 넣을 수 있다. 그것들은 배치 템플릿 안에 있는 `@RenderHeader` 위치에 렌더링된다.

`@LayoutPage`, `@RenderBody`, `@ExtraHeader`, `@RenderHeader`을 함께 활용하면, 여러분은 매우 유연하고 관리하기에도 좋은 템플릿 구조를 만들 수 있다.

템플릿 패턴들

웹스텐실즈를 사용하면, 여러분이 몇 가지 공통 템플릿 패턴들을 적용할 수 있다. 그래서 여러분의 애플리케이션의 모습들을 더 쉽게 구조화할 수 있다.

표준 배치

이 패턴은 내장된 `@LayoutPage`와 `@RenderBody`를 사용하는 방식이다. 앞에서 본 것이다. 여러분의 사이트 전체에서 일관성 있는 구조 하나를 유지하면서도 개별 페이지들이 각자 고유한 내용을 제공할 수 있도록 하는데 이상적이다.

Header/Body/Footer

이 패턴에서는, 페이지 각각이 개별 템플릿이다. 하지만, header, footer와 같은 모든 공통 부분을 공유한다. 한 가지 배치 페이지를 사용하는 것이 아니라, 여러분의 템플릿마다 구조를 잡을 수 있다. 다음과 같다:

```
@Import Header.html

<main>
    <!-- Page-specific content here -->
</main>

@Import Footer.html
```

이 방식은 표준 배치 패턴에 비교하면 더 유연하다. 하지만, 공통 요소들에 대한 관리에 더 신경을 써야 한다.

재사용 구성요소들

`@Import` 지시어를 통해, 개별 구성요소들의 세트를 정의할 수 있다. 이 구성요소들은 여러분의 애플리케이션 전반에 걸쳐 재사용될 수 있는 것들이다. 또한 순환될 수 있는 오브젝트들을 전달하는 것도 이 지시어에서 할 수 있다. 심지어 별칭을 정의해 구성요소 정의가 더 독립적이도록 할 수 있다. 예를 들면:

```
<div class="product-list">
  @Import ProductList { @list = @ProductList }
</div>

<div class="tasks">
  @ForEach (var Task in Tasks.AllTasks) {
            @Import partials/tasks/item { @Task }
      }
</div>
```

이 패턴을 통해, 여러분의 UI를 더 잘게 더 관리하기 좋은 조각들로 나눌 수 있다. 그러면 쉽게 유지보수 할 수 있으며 서로 다른 페이지들에 걸쳐 재사용할 수도 있다.

맺음말

웹스텐실즈는 여러분의 웹 애플리케이션에 템플릿들과 레이아웃들을 만들 수 있는 유연하고 강력한 시스템을 제공한다. `AddVar`를 통해, 여러분의 템플릿에게 데이터를 전달할 수 있다. 또한 `@LayoutPage`, `@RenderBody, @Import` 와 같은 지시어들을 활용하면, 잘 구조화되고, 관리하기 좋고, 재사용할 수 있는

표현(view)들을 웹 애플리케이션 안에서 사용할 수 있다.

앞에서 살펴본 템플릿 패턴들 – 즉, 표준 배치, header/body/footer, 재사용 구성요소들 – 은 여러분의 표현(view)의 구조를 잡을 수 있는 다른 방식을 각각 제공한다. 여러분의 애플리케이션의 니즈에 따라 그리고 팀의 작업 흐름에 맞추어 알맞은 패턴(또는 패턴들의 조합)을 선택하면 된다.

다음 장에서, 우리는 웹스텐실즈의 보다 수준 높은 특징들을 알아본다. 여기에는 폼을 가지고 작업하기, 사용자 입력 다루기, 동적으로 내용 업데이트 구현하기 등이 해당된다.

07

할 일 목록 앱을 웹스텐실즈로 옮기기

도입

이 장에서는, 우리가 앞에서 구축한 앱 즉 평범한 HTML을 상수들 안에 넣어 구축했던 할 일 목록 앱을 웹스텐실즈 템플릿으로 옮겨본다. 이 마이그레이션을 통해 여러분의 코드를 웹스텐실즈가 어떻게 더 관리하기 좋고 확장하기 좋게 해주는지를 볼 수 있다. 또한 몇 가지 추가 지능들을 넣어서 이 새 방식의 유연성도 보여줄 것이다.

깃허브 저장소(GitHub Repo)에 있는 웹스텐실즈 데모를 받아서 프로젝트를 열어보면, 작동하는 할 일 목록 앱이 들어 있음을 알 수 있다. 그것은 이 가이드에서 본 코드들과 똑 같은 아이디어를 구현한 것이다.

HTML 상수들을 템플릿으로 변환하기

우리가 만든 HTML 상수들을 웹스텐실즈 템플릿들로 변환하는 것부터 시작하자. 우리는 템플릿들을 따로따로 나누어 만들 것이다. 그래서 애플리케이션의 서로 다른 부분들을 각각 담당하도록 한다.

주 레이아웃 템플릿

맨 먼저, 주 레이아웃 템플릿을 만들자. 이것은 우리가 만드는 애플리케이션의 밑바탕이 된다.

`layout.html`이라는 파일을 만든다:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@@5.1.3/dist/css/bootstrap.min.css"
rel="stylesheet">
    <script src="https://unpkg.com/htmx.org@@1.9.2"></script>
    @RenderHeader
</head>
<body>
    <div class="container">
        @RenderBody
    </div>
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@@5.1.3/dist/js/bootstrap.bundle.min.js"></scr
ipt>
</body>
</html>
```

앞에서 본 예시는, CDN들이 제공하는 다양한 라이브러리들을 포함하고 있었다. 그것들에 대한 링크에는 각 고유 버전이 명시되어 있다. 이때, @@를 사용해 @ 기호를 이스케이프하는 것이 중요하다. 그래야 웹스텐실즈가 변수로 인식하지 않는다.

할 일 목록 템플릿

이제, 할 일 목록에서 사용할 템플릿을 만들자. `todo-list.html`이라는 파일을 만든다:

```
@LayoutPage layout.html

<div id="todo-list">
    @ForEach (var todo in Todos) {
        <div class="card mb-2">
            <div class="card-body d-flex justify-content-between align-items-center">
                <span>@todo.Description</span>
                <div>
                    @if not todo.Completed {
```

```
                        <button class="btn btn-sm btn-success"
hx-post="/complete/@todo.Id"  hx-target="#todo-list">Complete</button>
                    }
                    <button class="btn btn-sm btn-danger" hx-delete="/delete/@todo.Id"
hx-target="#todo-list">Delete</button>
                </div>
            </div>
        </div>
    }
</div>

<form hx-post="/add" hx-target="#todo-list" class="mt-4">
    <div class="input-group">
        <input type="text" name="description" class="form-control" placeholder="New todo
item" required>
        <button type="submit" class="btn btn-primary">Add</button>
    </div>
</form>
```

WebModule을 업데이트하기

이제, 새 템플릿을 사용하도록 우리의 WebModule을 업데이트하자. 아래 코드에서 사용되는 `TTodoList` 클래스는 확장되었다. 그래서 `Delete`, `Complete`, `GetAllItems` 등의 메서드들이 더 들어 있다. 그 코드는 여기에 적어 놓지 않았다. 순수한 델파이 코드이므로 굳이 이 가이드를 복잡하게 할 필요가 없기 때문이다. 깃허브에 있는 데모 프로젝트를 받아 보면, 아래와 비슷한 코드 작성 방식을 볼 수 있다.

```
unit TodoWebModule;

interface

uses
  System.SysUtils, System.Classes, Web.HTTPApp, TodoList, WebStencils;

type
  TTodoWebModule = class(TWebModule)
    procedure WebModuleCreate(Sender: TObject);
    procedure WebModuleDestroy(Sender: TObject);
    procedure WebModuleDefault(Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
  private
    FTodoList: TTodoList;
    FWebStencilsProcessor: TWebStencilsProcessor;
```

```
    procedure HandleGetTodoList(Response: TWebResponse);
    procedure HandleAddTodo(Request: TWebRequest; Response: TWebResponse);
    procedure HandleCompleteTodo(Request: TWebRequest; Response: TWebResponse);
    procedure HandleDeleteTodo(Request: TWebRequest; Response: TWebResponse);
  public
    { Public declarations }
  end;

var
  TodoWebModule: TTodoWebModule;

implementation

{%CLASSGROUP 'System.Classes.TPersistent'}

{$R *.dfm}

procedure TTodoWebModule.WebModuleCreate(Sender: TObject);
begin
  FTodoList:= TTodoList.Create;
  FWebStencilsProcessor := TWebStencilsProcessor.Create(Self);
  FWebStencilsProcessor.TemplateFolder := ExtractFilePath(ParamStr(0)) + 'templates\';
end;

procedure TTodoWebModule.WebModuleDestroy(Sender: TObject);
begin
  FTodoList.Free;
  FWebStencilsProcessor.Free;
end;

procedure TTodoWebModule.WebModuleDefault(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
Begin
// This is a WebBroker action that handles all the 요청들. For better maintainability, it
should be split into different actions.
  if Request.PathInfo = '' then
    HandleGetTodoList(Response)
  else if (Request.PathInfo = '/add') and (Request.MethodType = mtPost) then
    HandleAddTodo(Request, Response)
  else if Request.PathInfo.StartsWith('/complete/') and (Request.MethodType = mtPost)
then
    HandleCompleteTodo(Request, Response)
  else if Request.PathInfo.StartsWith('/delete/') and (Request.MethodType = mtDelete)
then
    HandleDeleteTodo(Request, Response)
  else
  begin
```

```
    Response.Content := 'Not Found';
    Response.StatusCode := 404;
  end;

  Handled := True;
end;

procedure TTodoWebModule.HandleGetTodoList(Response: TWebResponse);
begin
  FWebStencilsProcessor.AddVar('Todos', FTodoList.GetAllItems);
  FWebStencilsProcessor.InputFileName := 'todo-list.html';
  Response.Content := FWebStencilsProcessor.Content;
end;

procedure TTodoWebModule.HandleAddTodo(Request: TWebRequest; Response: TWebResponse);
var
  Description: string;
begin
  Description := Request.ContentFields.Values['description'];
  FTodoList.AddItem(Description);  HandleGetTodoList(Response);
end;

procedure TTodoWebModule.HandleCompleteTodo(Request: TWebRequest; Response:
TWebResponse);
var
  ItemId: Integer;
Begin
  // The ItemId is extracted from the PathInfo of the Request and converted to int
  ItemId := StrToIntDef(Request.PathInfo.Substring('/complete/'.Length), -1);
  if ItemId <> -1 then
    FTodoList.CompleteItem(ItemId);
  HandleGetTodoList(Response);
end;

procedure TTodoWebModule.HandleDeleteTodo(Request: TWebRequest; Response: TWebResponse);
var
  ItemId: Integer;
Begin
  ItemId := StrToIntDef(Request.PathInfo.Substring('/delete/'.Length), -1);
  if ItemId <> -1 then
    FTodoList.DeleteItem(ItemId);
  HandleGetTodoList(Response);
end;

end.
```

Note

위 코드는 매우 단순화해 놓은 것이다. 눈으로 보고 더 잘 이해할 수 있도록, 엔드포인트 연결을 위한 Action들을 사용하지 않고 있다. 깃허브에서 받을 수 있는 이 데모는 더 관리하기 좋도록 MVC 방식을 사용하고 있으며, 로직을 더 잘 추상화해 놓았다.

기타 기능 추가하기

이제 우리 앱을 웹스텐실즈로 마이그레이션을 했으니, 추가 기능 몇 가지를 넣어서 이 새 구조가 가지는 확장 능력과 유지보수 능력을 보자.

할 일 카테고리

할 일에 카테고리를 추가하자. 먼저, `TTodoList` 유닛 안에 있는 `TTodoItem` 클래스를 업데이트하자:

```
TTodoItem = class
public
  Id: Integer;
  Description: string;
  Completed: Boolean;
  Category: string;
  constructor Create(AId: Integer; const ADescription, ACategory: string);
end;

constructor TTodoItem.Create(AId: Integer; const ADescription, ACategory: string);
begin
  Id := AId;
  Description := ADescription;
  Completed  :=  False;
  Category := ACategory;
end;
```

이제, `todo-list.html` 템플릿을 업데이트해서 카테고리를 포함시켜 보자:

WebModule 안에 있는 `HandleAddTodo` 메서드를 업데이트 한다:

```
@LayoutPage layout.html

<div id="todo-list">
    @ForEach (var todo in Todos) {
        <div class="card mb-2">
            <div class="card-body d-flex justify-content-between align-items-center">
                <div>
                    <span>@todo.Description</span>
                    <small class="text-muted ms-2">[@todo.Category]</small>
                </div>
                <div>
                    @if not todo.Completed {
                        <button class="btn btn-sm btn-success"
                                hx-post="/complete/@todo.Id"
                                hx-target="#todo-list">Complete</button>
                    }
                    <button class="btn btn-sm btn-danger"
                            hx-delete="/delete/@todo.Id"
                            hx-target="#todo-list">Delete</button>
                </div>
            </div>
        </div>
    }
</div>

<form hx-post="/add" hx-target="#todo-list" class="mt-4">
    <div class="input-group">
        <input type="text"
               name="description"
               class="form-control"
               placeholder="New todo item" required>
        <input type="text" name="category" class="form-control" placeholder="Category">
        <button type="submit" class="btn btn-primary">Add</button>
    </div>
</form>
```

```
procedure TTodoWebModule.HandleAddTodo(Request: TWebRequest; Response: TWebResponse);
var
  Description, Category: string;
begin
  Description := Request.ContentFields.Values['description'];
  Category := Request.ContentFields.Values['category'];
  FTodoList.AddItem(Description, Category);
  HandleGetTodoList(Response);
```

```
end;
```

할 일 필터링

할 일을 카테고리 별에 따라 필터링 하는 기능을 추가해 보자. `category-filter.html` 라는 파일을 새로 만든다:

```
<div class="mb-4">
    <h5>Filter by Category</h5>
    <div class="btn-group" role="group">
        <button class="btn btn-outline-primary"
                hx-get="/" hx-target="#todo-list">All</button>
        @ForEach (var category in Categories) {
            <button class="btn btn-outline-primary"
                    hx-get="/filter/@category"  hx-target="#todo-list">@category</button>
        }
    </div>
</div>
```

위에 보이는 카테고리 필터를 `todo-list.html` 템플릿 안에 넣어 업데이트한다:

```
@LayoutPage layout.html

@Import category-filter.html

<div id="todo-list">
    <!-- ... existing todo list content ... -->
</div>

<!-- ... existing form ... -->
```

필터링을 다루는 새 메서드를 해당 WebModule 안에 넣어 업데이트한다:

```
procedure TTodoWebModule.HandleFilterTodos(Request: TWebRequest; Response: TWebResponse);
var
  Category: string;
  FilteredTodos: TArray<TTodoItem>;
begin
  Category := Request.PathInfo.Substring('/filter/'.Length);
```

```
  FilteredTodos := FTodoList.GetItemsByCategory(Category);
  FWebStencilsProcessor.AddVar('Todos', FilteredTodos);
  FWebStencilsProcessor.AddVar('Categories', FTodoList.GetAllCategories);
  FWebStencilsProcessor.InputFileName := 'todo-list.html';
  Response.Content := FWebStencilsProcessor.Content;
end;
```

새 필터 경로를 다룰 수 있도록 `WebModuleDefault` 메서드를 업데이트한다:

```
procedure TTodoWebModule.WebModuleDefault(Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  if Request.PathInfo = '' then
    HandleGetTodoList(Response)
  else if Request.PathInfo.StartsWith('/filter/') then
    HandleFilterTodos(Request, Response)
  // ... existing routes ...
end;
```

이 가이드의 초점은 웹스텐실즈와 그것을 RAD 스튜디오에서 사용하는 방법이다. 앞에서 본 코드에서, 카테고리 필터링을 위해 필요한 추가 로직을 구현하는 코드 등은 그저 델파이 코드다. 웹스텐실즈에 대한 코드가 아니므로 이 가이드에 포함시키지 않았다. 이 코드 조각들을 이해하기가 더 간단하도록 하기 위해서다.

맺음말

이 장에서, 우리는 할 일 목록 앱을 이전했다. HTML 상수들을 웹스텐실즈 템플릿들로 옮겼으므로 우리의 코드는 더 관리하기 좋고 읽기도 더 쉬워졌다. 또한 우리는 관심 사항들을 따로 분리했다. 즉 HTML을 별도의 템플릿 파일들 안으로 옮겼다. 그래서 델파이 코드를 건드릴 필요가 없이 쉽게 변경할 수 있게 되었다.

또한 이 새 방식의 확장성을 볼 수 있었다. 작업 카테고리와 필터링 등 새 기능을 쉽게 추가했다. 추가 방식은 구현이 간략하고 분명했다. 웹스텐실즈 템플릿이 가지는 유연성 덕분이다.

웹스텐실즈를 사용하면, 우리는 다음과 같은 몇 가지 이점이 있다:

1. **관심사 분리**: 우리의 HTML은 이제 델파이 코드로부터 따로 분리되었다.

2. **재사용성**: 우리는 카테고리 필터링과 같은 구성요소들을 서로 다른 페이지들에 걸쳐 재사용할 수 있다.
3. **유지보수성**: UI를 변경할 때 델파이 코드를 수정할 필요가 없다.
4. **확장성**: 새 기능 추가하기가 더 간단하다. 그러면서도 코드 중복을 가능한 최소로 줄일 수 있다.

다음 장에서, 우리는 웹스텐실즈의 더 많은 고급 특징들을 살펴본다. 그리고 그것들을 우리의 애플리케이션 아키텍처 안으로 통합하는 방법을 알아본다.

08

웹스텐실즈에서 제공하는 고급 옵션들

도입

이 장에서, 우리는 웹스텐실즈 안에 있는 더 많은 고급 기능들과 옵션들을 살펴본다. 이 도구들을 통해 여러분은 더욱 동적이고, 효율적이고, 정교한 앱 애플리케이션들을 만들 수 있다.

@query 키워드

`@query` 키워드를 통해, 여러분은 HTTP 쿼리 파라미터를 직접 여러분의 템플릿 안에서 읽을 수 있다.이는 URL 파라미터들을 기반으로 동적으로 내용을 생성할 때 유용하다.

```
<!--
Example url: https://example.com?searchTerm="searchingString"&page=9&totalPages=34
-->

<h1>Search Results for: @query.searchTerm</h1>
<p>Page @query.page of @query.totalPages</p>
```

@Scaffolding

`@Scaffolding` 키워드는 여러분의 애플리케이션에 있는 데이터 구조들을 기반으로 해서, 동적으로 HTML을 생성하는 강력한 방법을 제공한다. 폼을 생성할 때 또는 데이터 테이블을 표현할 때 특히 유용하다.

```
<form>
  @Scaffolding User
</form>
```

구조물(scaffolding)을 구현하려면, WebStencilsProcessor의 `OnScaffolding` 이벤트를 처리해야 한다:

```
procedure TMyWebModule.ProcessorScaffolding(Sender: TObject;
  const AQualifClassName: string; var AReplaceText: string);
begin
  if SameText(AQualifClassName, 'User') then
  begin
    AReplaceText := '';
    // Generate form fields based on User properties
    AReplaceText := AReplaceText + '<input type="text" name="Username"
placeholder="Username">';
    AReplaceText := AReplaceText + '<input type="email" name="Email"
placeholder="Email">';
    // ... add more fields as needed
  end;
end;
```

위 코드는 Scaffolding 키워드를 AReplaceText의 값으로 교체한다.

이 `@Scaffolding` 코드 예시에서는, 이 키워드를 교체하는 HTML 태그들을 하드코딩 방식으로 적어 놓았다. 이해를 더 잘 수 있도록 그렇게 한 것이다. 하지만, 실전에서는 다른 클래스, 레코드, 상수 등에 저장되어 있는 코드를 사용해 교체하는 방식이 더 유용하다. 그 방식은 HTML 코드 조작의 구조화와 재사용성을 향상시킨다.

@LoginRequired

`@LoginRequired` 키워드를 사용하면, 사용자 인증 상태를 기반으로 여러분의 템플릿 중 특정 부분에 대한 접근을 제한할 수 있다. 이 키워드를 만나면, 특별한 이벤트가 발동된다. 그곳에서 여러분은 그 사용자가 로그인 했는지를 확인할 수 있다. 이 키워드의 목표는 승인(authorization)이 필요한 페이지들을 넣을 수 있게 하는 것이다.

WebStencilsProcessor는 불리언 프로퍼티인 `UserLoggedIn`을 제공해 사용자의 로그인 여부를 지정할 수 있도록 한다. 사용자 로그인의 성공 여부를 여기에 저장하면 된다.

```
@LoginRequired
<div class="protected-content">
  This content is only visible to logged-in users.
</div>
```

또한 역할을 명시해서 더 세밀하게 접근을 제어할 수도 있다:

```
@LoginRequired(admin)
<div class="admin-panel">
  Admin-only content goes here.
</div>
```

현재 사용자의 역할들을 명시하려면, WebStencilsProcessor의 `UserRoles` 프로퍼티를 사용하면 된다.사용자가 여러 가지 역할을 가지고 있는 경우에는, 이 역할들을 쉼표로 구분해 정의하면 된다. 예를 들어: `sales,management`와 같이 적어주면 된다.

처리기 안에 있는 내부 이벤트에서 그 사용자의 역할들을 평가한다. 그리고 그 사용자가 로그인하지 않아서 명시된 역할을 가지고 있지 않은 경우에는 `EWebStencilsLoginRequired` 예외를 일으킨다.

WebStencils는 내장 인증 시스템을 제공하지 않는다. 이 프로퍼티들과 키워드들은 오직 사용자의 요청이 외부 서비스를 통해 올바르게 승인된 경우에 사용할 수 있도록 하기 위해 있는 것들이다.

OnValue 이벤트 핸들러

`OnValue`이벤트 핸들러는 여러분의 템플릿에게 데이터를 공급할 때 사용된다. 이것이 특히 유용한 경우는 여러분의 값을 즉석에서 계산하거나 또는 여러분의 데이터 접근 로직을 여러분의 템플릿 안에 두지 않고 따로 빼내어 다른 곳에 두고 싶을 때다.

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
  var Handled: Boolean);
begin
  if SameText(ObjectName, 'CurrentTime') then
  begin
    ReplaceText := FormatDateTime('yyyy-mm-dd hh:nn:ss', Now);
    Handled := True;
  end;
end;
```

사용할 때는 이렇게 하면 된다:

```
<p>Current time: @CurrentTime</p>
```

템플릿 패턴들

웹스텐실즈는 공통 템플릿 패턴들 몇 가지를 제공한다. 그래서 애플리케이션 표현(view)의 구조를 여러분이 효과적으로 구성할 수 있도록 돕는다.

표준 배치

이 패턴은 내장된 `@LayoutPage`와 `@RenderBody`를 사용하는 방식이다. 여러분의 사이트 전체에서 일관성 있는 구조 하나를 유지하면서도 개별 페이지들이 각자 고유한 내용을 제공할 수 있도록 하는데 이상적이다. 지금까지 보여 준 데모들은 이 패턴을 따르고 있다.

Header/Body/Footer

이 패턴에서는, 페이지 각각이 개별 템플릿이다. 하지만, 이 페이지들은 header, footer와 같은 모든 공통 부분을 공유한다. 한 가지 배치 페이지를 사용하는 것이 아니라, 여러분의 템플릿마다 구조를 잡을 수 있다. 다음과 같다:

```
@Import Header.html

<main>
    <!-- Page-specific content here -->
</main>

@Import Footer.html
```

이 방식은 표준 배치 패턴에 비교하면 더 유연하다. 하지만, 공통 요소들에 대한 관리에 더 신경을 써야 한다.

재사용 구성요소들

`@Import` 지시어를 통해, 개별 구성요소들의 세트를 정의할 수 있다. 이 구성요소들은 여러분의 애플리케이션 전반에 걸쳐 재사용될 수 있는 것들이다. 또한 순환될 수 있는 오브젝트들을 전달하는 것도 이 지시어에서 할 수 있다. 심지어 별칭을 정의해 구성요소 정의가 더 독립적이도록 할 수 있다. 예를 들면:

```
<div class="product-list">
    @Import ProductList { @list = @ProductList }
</div>

<div class="tasks">
  @ForEach (var Task in Tasks.AllTasks)
        { @Import partials/tasks/item { @Task }
    }
</div>
```

이 패턴을 통해, 여러분의 UI를 더 잘게 더 관리하기 좋은 조각들로 나눌 수 있다. 그러면 쉽게 유지보수 할 수 있으며 서로 다른 페이지들에 걸쳐 재사용할 수도 있다. 여러분은 이 패턴의 몇 가지 예시들을 이 데모의 코드에서 볼 수 있다.

맺음말

웹스텐실즈의 이 고급 기능들은 여러분이 동적이고, 효율적이고, 정밀한 웹 애프릴케이션을 만들 수 있도록 하는 강력한 도구가 된다. 이 능력들을 활용하면, 여러분의 코드는 더 관리하기 좋고 유연하다. 또한 여러분의 관심사들을 더 효과적으로 분리할 수도 있다. 그리고 견고한 웹 애플리케이션을 만들어서 요구사항들에 맞추면서 키워갈 수 있다.

다음 장에서, 우리는 웹스텐실즈를 RAD Server에 어떻게 통합하는지 알아본다. RAD Server는 확장성있는 웹 애플리케이션을 구축하는 더 많은 가능성들을 열어준다.

09

RAD Server 통합을 웹스텐실즈와 함께 사용하기

도입

RAD Server 역시 이 새 라이브러리인 웹스텐실즈의 혜택을 받는다. 특별 통합을 개발해 놓았으므로, 여러분은 RAD Server에서 웹 개발의 모든 잠재력을 활용할 수 있다. 문법 구조, 컴포넌트들 등등, 지금까지 배운 모든 것들은 RAD Server에서도 그대로 적용된다.

웹스텐실즈를 RAD Server와 통합하기

앞에서 본 예제들과 마찬가지로, 여러 가지 방식을 통해 RAD Server와 웹스텐실즈를 함께 사용할 수 있다. 가장 일반적인 패턴 두 가지를 보자:

웹스텐실즈 처리기들을 사용하기

이것은 간단한 방식이다. 이 방식에서는 각 요청마다 처리기를 사용한다. 그래서 필요한 HTML을 생성한다 (디자인 타임 또는 코드 작성 방식으로 구현). 그리고 RAD Server의 응답 안에 직접 반환한다. 처리기에서 사용하는 템플릿은 파일 또는 상수, 변수 등등에 들어 있을 수 있으며, 처리기는 이것을 읽은 후에 처리한다.

```
type
  [ResourceName('testfile')]
  TTestResource = class(TDataModule)
    [ResourceSuffix('get', './')]
    [EndpointProduce('get', 'text/html')]
    procedure Get(const AContext: TEndpointContext; const ARequest: TEndpointRequest;
const AResponse: TEndpointResponse);

...

procedure TTestResource.Get(const AContext: TEndpointContext; const ARequest:
TEndpointRequest; const AResponse: TEndpointResponse);
var
  LTemplateFile, LHTMLContent: string;
Begin
  // replace this variable with the real path to the template
  LTemplateFile := 'C:\path\to\your\file.html';
  WebStencilsProcessor.InputFileName := LTemplateFile;
  LHTMLContent := WebStencilsProcessor.Content;
  AResponse.Body.SetString(LHTMLContent);
end;
```

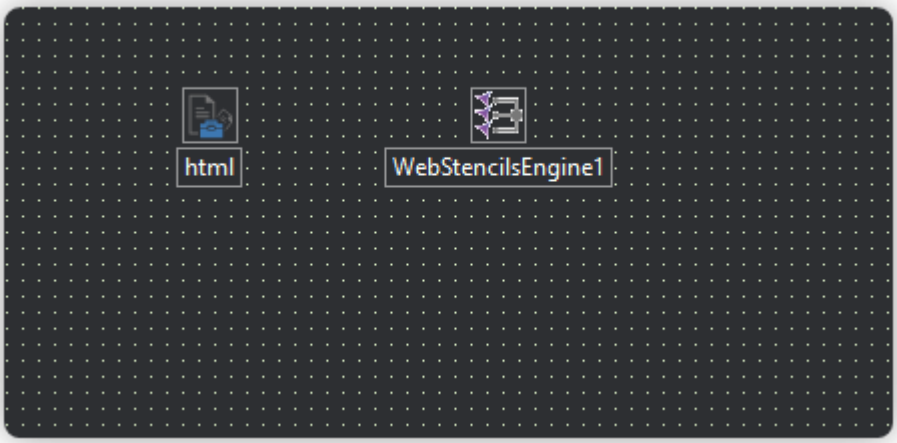
이 RAD Server 프로젝트를 실행하고 브라우저를 열면, http://localhost:8080/testfile URL에 접근할 수 있을 것이다. 그리고 `LFilePath`. 변수 안에 정의된 템플릿에 있는 내용이 렌더링되는 것을 볼 수 있을 것이다.


Note

RAD Server용인 경우, 템플릿 경로는 반드시 절대 경로여야 한다. 상대 경로를 사용할 수 없다. 그 이유는 RAD Server가 Apache 또는 IIS를 사용해 운영 환경에 배포되는 방식 때문이다.

웹스텐실즈 엔진을 사용하기

이 옵션을 선택한다면, 기존 `TEMSFileResource` 컴포넌트와 `TWebStencilsEngine` 컴포넌트를 조합하는 방식을 추천한다. 전자는 파일 시스템에 대한 맵핑을 담당한다. 한편, 후자는 HTTP 맵핑과 템플릿 처리를 담당한다. 이 두 컴포넌트를 서로 연결할 때는 웹스텐실즈 엔진의 Dispatcher 프로퍼티를 사용한다.



`TEMSFileResource`를 올바르게 구성하려면, 템플릿이 있는 위치를 명시해야 한다. `PathTemplate` 프로퍼티를 사용하면 된다.

`C:\path\to\your\templates\{filename}`

`{filename}` 라는 와일드카드에 특히 주의해야 한다. 이것은 각 페이지들을 참조한다.

컴포넌트들을 일단 구성했다면, `PathsTemplates`를 Engine 안에서 정의할 수 있다. 그 방식은 앞에서 본 것처럼 와일드카드인 `{fileName}` 또는 템플릿 파일의 이름을 명시하면 된다.

파일들을 올바르게 맵핑하려면, FileResource에 애트리뷰트들 몇 가지를 정의해야 한다. 앞의 예시에 있는 것처럼 하면 된다:

```
type
  [ResourceName('testfile')]
  TTestfileResource1 = class(TDataModule)
    [ResourceSuffix('./')]
    [ResourceSuffix('get', './{filename}')]
    [EndpointProduce('get', 'text/html')]
    html: TEMSFileResource;
```

이 예시에 따르면, 우리는 http://localhost:8080/testfile/{filename} 엔드포인트에 접근할 수 있다. 여기에서 `{filename}`은 우리가 `PathTemplate` 프로퍼티 안에서 정의한 경로 안에 저장되어 있는 모든 파일들을 가리킨다.

동일한 `TWebStencilsEngine` 엔진이 하나보다 많은 `TEMSFileResource`를 필요로 하는 경우, 글로벌 메서드인 `AddProcessor` 를 사용해 추가할 수 있다.
예시:

```
AddProcessor(FileResourceResource, WebStencilsEngine1);
```

이 할 일 목록 앱을 RAD Server 안에 다시 만들어 넣기

RAD Server는 WebBroker와 작동 방식이 조금 다르다. REST-호환 애플리케이션이 되기 위해, 메모리 상태를 가질 수 없기 때문이다. 따라서 몇 가지 사항을 고려해야 한다.

여러분이 만약 깃허브 저장소 안에 있는 WebBroker 데모를 받아본다면 MVC 방식을 따르고 있다는 것을 알 수 있었을 것이다. 동일한 그 저장소 데모 안에서는 RAD Server로 마이그레이션 된 동일한 데모도 들어 있다. 그 둘의 기능은 모두 똑같다. 이전을 하기 위해 리팩토링이 필요했던 것들을 나열해 보겠다:

데이터베이스 관리

WebBroker 데모 안에서는, 할 일들이 메모리 안에 저장되기 때문에, 싱글톤 패턴을 사용해서 멀티-쓰레드 수행에 따른 문제를 피했다. RAD 서버 안에서는, 그 할 일들을 메모리 안에 저장할 수 없다 (적어도 쉽지 않다). 따라서 우리는 InterBase 데이터베이스를 대신 사용했다.

우리의 모델은 데이터를 직접 안에서 정의하는 대신, 데이터베이스에 접속해야 한다. 그것을 위해, 모델인 `TTasks` 생성자 안에서 `TFDConnection` 컴포넌트가 할당된다. 데모가 간단하도록, 동일한 RAD Server DataModule 안에서 `TFDConnection` 컴포넌트를 생성했다. 실전 앱에서는, `TFDManager` 컴포넌트를 사용하는 것을 권장한다.

컨트롤러 인자(argument)들

WebBroker와 RAD Server의 요청들과 응답들은 서로 조금 다르다. 따라서 컨트롤러 메서드들에게 전달되는 인자(argument)들에 대한 리팩토링 몇 가지가 필요하다.

Action들을 Endpoint들로

WebBroker는 Action들을 사용해 엔드포인트들을 정의하고 그것들을 비즈니스 로직들과 연결한다. 한편 RAD Server에서는 리소스 안에 있는 메서드들을 정의할 때 반드시 애트리뷰트를 사용해 특정 URI를 지정해야 한다.

요청에 들어 있는 데이터를 처리하기

RAD Server는 JSON을 사용한다. 할 일 목록 앱에서 받은 데이터를 우리의 백엔드로 전달할 때 뿐만 아니라 그 데이터를 처리할 때도 그렇다. 그래서 몇 가지 변경이 필요하다.

1. **HTMX 확장인 JSON-enc를 사용한다**: 이 확장은 백엔드로 보내지는 요청들을 JSON 형식으로 인코딩한다. (기본 동작 방식인) 폼-데이터 형식으로 그냥 보내지 않는다. 이 확장을 사용하려면,

간단한 `<script>` 태그를 추가해야 한다. 그리고 `hx-post` 또는 `hx-put` 요청이 정의되어 있는 태그 안에 반드시 `hx-ext="json-enc"` 애트리뷰트를 명시해야 한다.

2. **데이터 처리 변경**: RAD Server는 요청 데이터를 JSON으로 처리하기 때문에, 몇 가지 변경을 해야 요청으로부터 데이터를 얻을 수 있다. 이 데모 목적으로는 표준 JSON 라이브러리만 사용해도 충분하다.

정적인 JS, CSS, 이미지 등을 다루기

`TEMSFileResource` 컴포넌트는 정적인 파일들을 전달하기 위해 사용된다. 각 폴더마다 독립적인 컴포넌트를 만들어서 맵핑을 했다. 그 폴더들 각각에는 파일들(JS, CSS, 이미지 등)이 들어 있다.

프론트엔드 원본들

RAD Server는 리소스들에게 의존한다. 즉, 우리의 주 URL에는 해당 리소스의 이름이 뒤에 붙는다. 예를 들면 - https://localhost:8080/web.

WebBroker 웹사이트에서 만들었던 엔드포인트들을 RAD Server로 이전하기 위해서는, 웹스텐실즈 템플릿들을 약간 변경해서 새로 생성되는 리소스들을 가리키도록 해야 한다.

10

자료 및 추가 학습

이 책에서 설명된 주제들의 잠재력을 더 확장하는데 유용한 자료들 몇 가지들을 정리해 놓았다:

도움말 문서 및 링크들

공식 HTMX 도움말 문서(HTMX.org)

공통 키워드들 대부분을 이미 이 책에서 다뤘다. 하지만, HTMX는 훨씬 더 많은 것들을 여러분의 프로젝트에 넣어 줄 수 있다. HTMX 팀에서 제공하는 도움말은 방대하다. 하지만, 이해하기 좋고, 부담스럽지 않다.

공식 도움말 문서 뿐만 아니라, htmx.org 웹사이트에는 기고들과 예시들도 있다.

- 공식 도움말 문서
- 예시
- 기고

델파이와 HTMX (엠바카데로 블로그)

이 기고 시리즈에서, HTMX와 WebBroker에 대한 접근을 맨 처음 소개했었다. 그 개념들 대부분은 이미 이 책에서 논의된 것들이다. 하지만, 더 많은 코드 예시들 그리고 기타 통합에 관련된 내용들이 있다: 예를 들어, WordPress에 HTMX와 통합하기 등이 있다.

- 웹의 강력함을 다루기: 델파이와 HTMX를 사용 (한글 번역)
- 웹의 강력함을 다루기: 델파이와 HTMX를 사용 - 2 편 (한글 번역)
- 델파이와 HTMX를 워드프레스 안에서 실행하기: HTMX 시리즈 3 편 (한글 번역)

RAD Server 기술 가이드

RAD Server에 익숙하지 않다면 그리고 그 모든 가능성들을 알아보고 싶다면, 이 책을 보면 된다. 주요 기능들을 여러분이 쭉 따라갈 수 있을 것이다. 뿐만 아니라 더 많은 고유하고 수준 높은 기능들에 대해서도 설명되어 있다.

RAD 서버 가이드 보기 (한글 번역)

HTMX를 MVC 패턴 방식으로 (HTMX.org)

어떻게 HTMX를 MVC-스타일 웹 애플리케이션에 맞추는지에 대해서는 아래 페이지에 잘 요약되어 있다. 여기에서는 얇은 컨트롤러들의 예시를 보여주고 HTMX를 사용해 웹 개발을 할 때 MVC 흐름을 어떻게 구성하면 되는지를 보여준다. 델파이에 맞춰진 내용은 아니지만, 이 개념은 MVC 디자인 패턴에서 광범위하게 적용할 수 있으며, 다양한 언어로 적용할 수 있다.

알아 보기

웹스텐실즈 (DocWiki)

우리의 공식 웹스텐실즈 도움말 문서는 DocWiki 안에 있다. 알아보기

UI/CSS 라이브러리들

이 공간은 방대하다. 그리고 여러 가지 라이브러리들이 들어 있다. 웹스텐실즈와 HTMX는 UI를 가지지 않는 성질을 가지고 있으므로, 사실 상 어떤 라이브러리든 사용할 수 있다. 특히 아래에 있는 것들과 잘 작동한다. 그리고 도움말도 잘 되어있다.

- Bootstrap
- BeerCSS
- PicoCSS
- Bulma
- DaisyUI
- TailwindCSS

TailwindCSS는 규모 면에서 매우 무거운 라이브러리다. 그리고 이것을 CDN을 통해 추가하는 경우에는 초기 적재량이 너무 크다. 제공되는 CLI 도구를 사용해 사용하지 않는 클래스들을 모두 제거해야 실전 개발에서 활용할 수 있다.

HTMX를 더 확장하기

HTMX의 특징인 선언 방식 덕분에, 여러분의 프로젝트는 JS에 훨씬 덜 의존할 수 있다. 하지만, 상황에 따라 JS가 필요한 경우가 있다. 순전히 클라이언트-쪽에서 상호작용해야 하는 경우에 그렇다. 예를 들어 웹사이트의 모습을 밝은 테마에서 어두운 테마로 바꾸기가 여기에 해당된다. 우리는 백엔드 안에서 이것을 처리해서 어두운 테마용 HTML을 만들고 나서 다시 클라이언트에게 보내줘도 된다. 하지만, 현대식 CSS 라이브러리들 덕분에 서버에 보내지 않고 클라이언트 쪽에서 쉽게 처리할 수 있다.

추가 기능들과 상호작용을 클라이언트 쪽에서 처리해야 하는 경우에 사용할 수 있는, 매우 작은 JS 라이브러리들이 있다. 이것들은 HTMX와 웹스텐실즈에 굉장히 잘 통합된다.

AlpineJS

AlpineJS(알파인JS)는 가벼운 자바스크립트 프레임워크다. HTML에 간단한 상호작용을 추가할 수 있도록 하기 위해 고안되었다. 이것은 선언적 방식으로 DOM 엘리먼트들을 조작한다. 방대한 자바스크립트 코드를 작성할 필요가 없다.이 기술은 종종 Vue 또는 React와 비교된다. 하지만 이것이 훨씬 더 작다. 그리고 이미 작성한 HTML에 통합하기가 더 쉽다. AlpineJS는 자바스크립트 행위를 웹 페이지에 추가하는데 이상적이다. 거대한 프레임워크가 가지는 복잡성이 없기 때문이다.

- 사용 사례: 모달, 드롭다운, 토글, 폼의 유효성 검사, 기타 UI 상호작용.
- 핵심 특징: 선언 방식 문법 구조, 반응형 데이터, DOM 조작을 위한 지시어들.

도움말 문서: AlpineJS Docs

Hyperscript

Hyperscript(하이퍼스크립트)는 일종의 스크립트 언어다. 이벤트 처리와 로직을 HTML 안에서 단순화할 수 있도록 고안되었다. 자바스크립트는 장황한 문법 구조가 필요하지만 Hyperscript는 HTML 애트리뷰트들 안에 직접 끼워 넣도록 설계되었다. 그 초점은 이벤트-기반 프로그래밍을 더 직관적으로 할 수 있도록 하는 것이다. 그러기 위해 자연어 같은 문법 구조를 사용한다. 따라서 개발자는 복잡한 자바스크립트를 쓰지 않아도 동적 행위를 만들 수 있다.

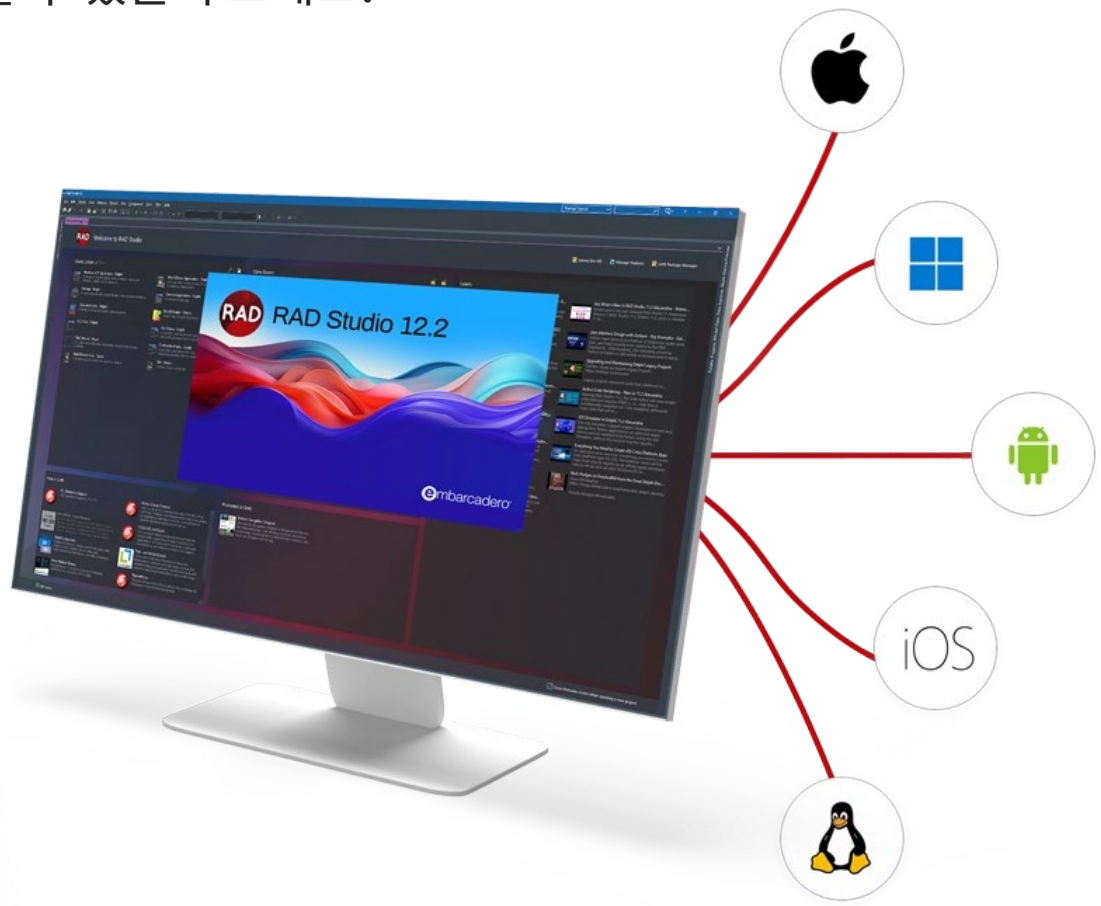
- 사용 사례: 대화형 버튼, 폼 제출 다루기, 엘리먼트 보여주기/숨기기 토글.
- 핵심 특징: 자연어 문법 구조, 선언 방식 이벤트, 일반적인 상호작용을 단순하게 처리.

도움말 문서: Hyperscript Docs

지금 RAD 스튜디오를 써보세요!

얼마나 쉽게 단 하나의 코드베이스를 사용해 멀티 플랫폼을 대상으로 네이티브 앱을 구축할 수 있는지 보세요!

www.embarcadero.com



embarcadero