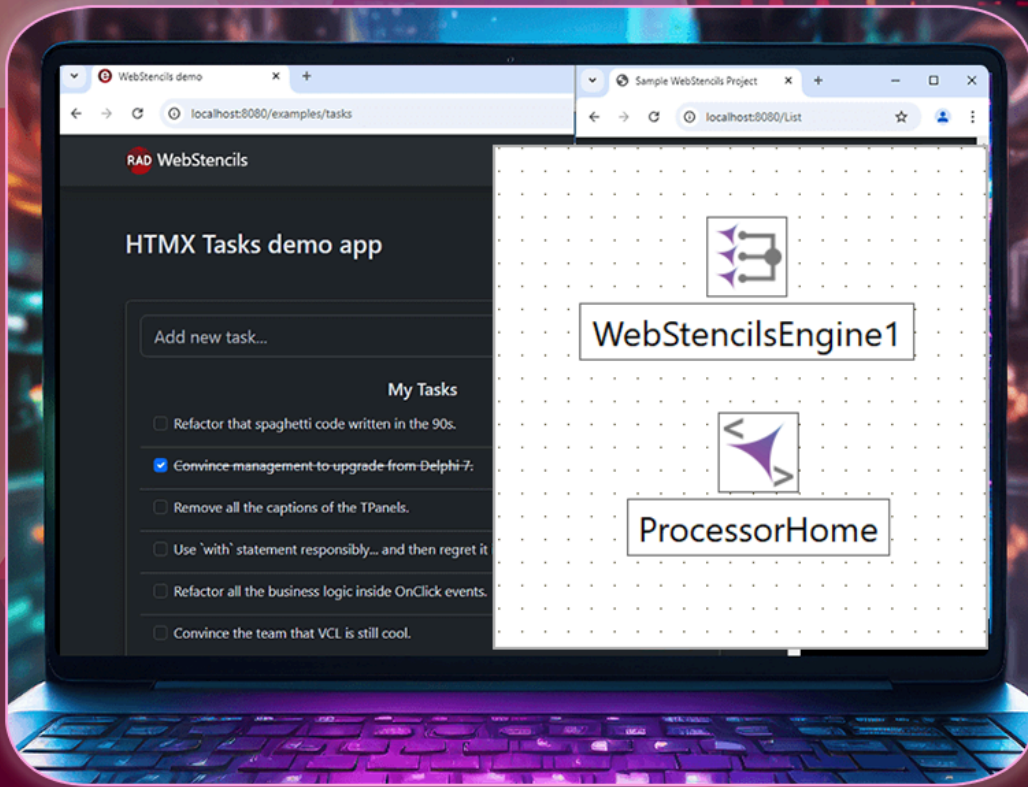


영어 원본 다운로드: <https://lp.embarcadero.com/HTMX-WebStencils>

HTMX & 웹스텐실즈

빠르게 웹 개발하기: RAD Studio 사용 (13.0 기준)



Author

Antonio Zapater

 **embarcadero**

©2026

목차

머릿말.....	7
01 - HTMX 소개.....	8
HTMX란?.....	8
간략 개요.....	8
전통적 자바스크립트와 AJAX 등과 비교.....	9
핵심 개념들.....	10
hx-get: 내용 가져오기(GET 요청을 통해).....	10
hx-target: 목표 엘리먼트 지정하기(그곳에 응답이 들어간다).....	10
hx-post: 데이터 제출하기(POST 요청을 통해).....	11
hx-put, hx-patch, hx-delete 요청들.....	11
hx-swap: 내용이 어떻게 뒤바뀌는지 그 방식을 제어하기.....	11
추가적인 핵심 개념들.....	12
02 - WebBroker(웹브로커) 소개.....	13
WebBroker(웹브로커) 란?.....	13
WebBroker의 주요 특징들.....	13
핵심 개념들.....	14
컴포넌트들 그리고 아키텍처.....	14
WebBroker 애플리케이션 만들기.....	14
요청들과 응답들을 다루기.....	15
배포 그리고 확장성 (Deployment and Scalability).....	15
세션 관리 (WebBroker 안에서).....	16
보안 고려사항들.....	16
CSRF 보호.....	16
데이터 검증(Data Validation).....	18
사이트-간 스크립팅(XSS, Cross-site Scripting).....	19
기타 보안 고려 사항들.....	19
03 - 여러분의 첫 웹 앱을 개발하기 (WebBroker를 사용).....	21
머릿말.....	21
“Hello World” 애플리케이션 만들기.....	21
기초적인 ‘할 일 목록’ 앱.....	22
04 - 고급 애트리뷰트들 및 보안 (HTMX를 사용).....	26
머릿말.....	26
고급 애트리뷰트(attribute)들.....	26
hx-put 그리고 hx-delete: 요청을 제출하기 (PUT 과 DELETE를 통해).....	26
hx-trigger: 이벤트 트리거들을 커스터마이징 하기.....	27
hx-select: 서버 응답의 일부를 선택하기.....	28
hx-include: 부가적인 데이터를 요청 안에 포함하기.....	28
hx-push-url: 브라우저의 URL을 업데이트하기.....	29
05 - 웹스텐실즈(WebStencils) 소개.....	30
WebStencils(웹스텐실즈)란?.....	30
핵심 개념.....	30
HTMX와 통합하기.....	31
CSS와 JS를 가리지 않음.....	31

WebStencils의 문장 구조(Syntax).....	31
@ 기호.....	31
중괄호 (블록용) {}.....	32
값을 접근하기 (점 표기법을 사용).....	32
WebStencils 키워드들 및 예시들.....	32
@page.....	32
@query.....	33
주석 (@* .. *@).....	33
@if 그리고 @else.....	34
@if not.....	34
@switch.....	35
@ForEach.....	36
맷음말.....	37
06 - 컴포넌트들, 배치(layout) 옵션들.....	38
머릿말.....	38
웹스텐실즈 컴포넌트들.....	38
웹스텐실즈 엔진(WebStencils Engine).....	39
웹스텐실즈 프로세서(WebStencils Processor).....	39
TWebStencilsEngine 그리고 WebBroker(웹브로커).....	39
데이터를 추가하기 (AddVar를 사용하기).....	40
직접 오브젝트를 할당하기.....	40
애트리뷰트(Attribute)를 사용하기.....	41
커스텀 데이터 액세스 (Lookup 함수들을 이용하기).....	41
중요한 고려사항들.....	42
배치(layout), 내용 자리표시자(Content Placeholder)들.....	43
@RenderBody.....	43
@LayoutPage.....	43
@Import.....	44
@ExtraHeader, @RenderHeader.....	44
중첩되는 배치(Nested Layout) 지원.....	46
템플릿 패턴들.....	47
표준 배치.....	47
Header/Body/Footer.....	47
재사용 구성요소들.....	48
맷음말.....	48
07 - ‘할 일 목록’ 앱을 WebStencils로 옮기기.....	49
머릿말.....	49
HTML 상수들을 템플릿들로 변환하기.....	49
주(Main) 레이아웃 템플릿.....	49
‘할 일 목록’ 템플릿.....	50
WebModule(웹모듈)을 업데이트하기.....	51
추가 기능들을 더하기.....	54
할 일 카테고리(들).....	54
할 일들을 필터링하기.....	56
맷음말.....	57

08 - WebStencils가 제공하는 고급 옵션들.....	58
머릿말.....	58
표현식 평가(Expression Evaluation): @()를 사용하기.....	58
@() 가 할 수 있는 것들.....	59
@Scaffolding.....	60
실용적인 스캐폴딩(Scaffolding) 예시.....	61
OnValue 이벤트 핸들러.....	62
기본적인 OnValue 예시.....	62
고급 OnValue 상황들.....	63
OnValue vs AddVar.....	65
인증(Authentication) 그리고 권한 부여(Authorization).....	65
맺음말.....	65
09 - 세션(Session) 관리 및 인증(Authentication).....	67
머릿말.....	67
세 가지 컴포넌트.....	68
TWebSessionManager.....	68
TWebFormsAuthenticator.....	68
TWebAuthorizer.....	68
인증 설정하기.....	68
컴포넌트 구성(Configuration).....	69
자격 증명 검증 (Credential Validation) 구현하기.....	69
로그인 폼(Login Form) 만들기.....	70
로그아웃 핸들러(Logout Handler) 만들기.....	71
@session 오브젝트.....	71
세션(Session)의 프로퍼티들.....	71
@session을 템플릿 안에서 사용하기.....	72
권한 부여 구역(Authorization Zone)들.....	73
보호된 영역(protected area)들 구성하기.....	73
구역 타입(Zone Type)들.....	73
다중 역할(Multiple Role)들.....	74
세션 구성(Session Configuration) 옵션들.....	74
세션 ID 저장소.....	74
세션 범위(Session Scope).....	75
세션 타임아웃.....	75
공유 비밀 키(Shared Secret).....	76
맺음말.....	76
10 - 데이터베이스-주도로 UI를 만들어 내기.....	77
머릿말.....	77
작동 방식.....	78
보안 : 화이트리스트(Whitelist) 체계.....	78
동적 폼(Dynamic Form)들을 구축하기.....	79
@switch 연산자와 필드 타입(Field Type)들.....	80
재사용 가능한 필드 컴포넌트(Component)들.....	82
하이브리드 접근 방식 (종종 최선의 선택이 된다).....	83
맺음말.....	84

11 - 최신 CSS 프레임워크를 가지고 작업하기.....	86
머릿말.....	86
CSS를 가리지 않는다(CSS-Agnostic)는 장점.....	87
프레임워크 옵션들(몇 가지 예시).....	87
Bootstrap(부트스트랩).....	87
Bulma(불마).....	88
PicoCSS(피코CSS).....	88
BeerCSS(비어CSS).....	88
DaisyUI(데이지UI).....	88
이 프레임워크들을 사용하기.....	88
Tailwind(테일윈드)라는 특수한 사례.....	89
Tailwind의 접근 방식을 이해하기.....	89
RAD 방식: 독립형 CLI.....	90
하이브리드 개발 작업 흐름.....	91
빌드 절차(Build Process)를 구성하기.....	92
지원을 하는 파일들.....	92
프레임워크 선택에 관한 참고 사항.....	93
맺음말.....	93
12 - 배포 옵션들 그리고 Docker(도커).....	94
머릿말.....	94
여러분의 옵션들을 이해하기.....	95
독립형 배포(Standalone Deployment).....	95
작동 방식.....	95
실제 운영 환경에서의 지속적 성공 가능성(Production Viability).....	96
배포 절차 (Deployment Process).....	97
FastCGI를 NGINX와 함께 사용하기 (RAD Studio 13.0 이상).....	97
NGINX 구성(Configuration).....	98
전통적인(traditional) 웹 서버 통합.....	99
‘상태 없음(statelessness)’의 문제.....	99
해결책: 외부(external) 세션 저장소.....	99
늘 생각해야 하는 기타 주의 사항들.....	100
전통적인 CGI에 관한 참고 사항.....	100
Docker(도커) 배포.....	100
Docker 이미지를 생성하는 방법.....	101
실제 운영급 Docker(도커) 이미지를 만들기.....	101
자동화된 빌드(Build) 접근 방식.....	102
사전 요구 사항들과 설정.....	103
완전한 예시가 제공된다.....	103
실제 운영 고려 사항들.....	104
SSL/TLS 구성(Configuration).....	104
로깅(Logging) 및 모니터링(Monitoring).....	105
환경별 구성(Configuration).....	105
맺음말.....	105
13 - RAD Server 통합을 통해 WebStencils를 사용하기.....	107
머릿말.....	107

웹스텐실즈를 RAD Server와 통합하기.....	107
웹스텐실즈 프로세서(WebStencils Processor)들을 사용하기.....	107
웹스텐실즈 엔진(WebStencils Engine)을 사용하기.....	108
할 일 목록 앱을 RAD Server 안에 다시 만들어 보기.....	110
데이터베이스 관리.....	110
컨트롤러 인자(argument)들.....	110
Action들을 Endpoint들로.....	110
요청에 들어 있는 데이터를 처리하기.....	111
정적인 리소스 다루기: JS, CSS, 이미지.....	111
프론트엔드 소스들.....	111
14 - 자료들 그리고 추가 학습.....	112
도움말 문서집 그리고 링크들.....	112
라이브 WebStencils 데모.....	112
엠바카데로 블로그 기고들.....	112
공식 HTMX 도움말 문서집 (HTMX.org).....	113
RAD Server 기술 가이드.....	113
HTMX를 MVC 패턴 방식으로 (HTMX.org).....	113
WebStencils (DocWiki).....	113
HTMX를 훨씬 더 확장하기.....	113
AlpineJS.....	114
Hyperscript.....	114
15 - 부록: 약어(Acronyms, Abbreviations).....	115

머릿말

이 책의 초점은 현대적이고 효율적인 접근 방식 웹 개발이다. HTMX와 WebStencils(웹스텐실즈)를 사용한다.

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 Studio의 개발 철학인 애플리케이션을 빠르게 개발하기와 완벽하게 일치한다. 즉, 개발자가 애플리케이션 로직에 더 집중할 수 있고, 복잡한 프론트엔드 코드와 씨름할 필요가 없도록 한다.

WebStencils의 아름다움은 템플릿-주도 아키텍처에 있다. 개발자가 처음부터 새로 만들지 않고, 기존의 비즈니스 로직을 노출할 수 있다. 재사용할 수 있고 사용자 정의가 가능한 템플릿들을 활용하기 때문이다. 이 템플릿들은 기존 애플리케이션들과 자연스럽게 통합된다. 그래서 오래된 프로젝트를 웹으로 가져오는 데 드는 거부감이 줄어든다. 즉, 개발 속도가 높아질 뿐만 아니라 개발 팀 간의 협업이 향상된다. 그 결과, 개발자들은 기존 코드 기반들을 가지고 더 긴밀하게 작업할 수 있다.

이 책은 HTMX와 WebStencils의 힘을 활용하는 방법을 알려준다. 그래서 현대적인 웹 애플리케이션을 여러분이 개발할 수 있다. 수고는 더 적고 유연성은 더 크다. 여러분이 기존 데스크톱 애플리케이션을 확대해 웹용으로 제공하든, 아니면 새로 동적 웹 프로젝트를 구축하든, 이 책은 실용적인 통찰력을 제공한다. 그래서 진화하고 있는 RAD Studio의 웹 개발 생태계를 여러분이 최대한 활용하도록 돕는다.

RAD Studio에 대해 더 알아보거나, 무료 평가판을 받아서 이 책의 예제들을 가지고 코딩을 해 볼 수 있다. <https://www.embarcadero.com/products/rad-studio>로 가면 된다.

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



Note

WebStencils와 HTMX 코드 예시들은 깃허브(GitHub) 저장소에서 받을 수 있다. 이 예시들은 이 가이드 문서에서 설명하고 있는 예시들과 비슷한 패턴을 따르고 있다:

<https://github.com/Embarcadero/WebStencilsDemos>

라이브 데모 사이트: <https://wsdemo.embarcadero.com>

01

HTMX 소개

HTMX란?

간략 개요

HTMX는 HTML을 확장한다. 그래서 하이퍼텍스트 매개체가 될 수 있도록 한다. 여러분은 AJAX 요청들을 만들고, CSS 전환 발동, WebSocket들 생성, 서버에서 보내는 이벤트(SSE, Server-Sent Event) 활용 등을 모두 HTML 엘리먼트 안에서 직접 서술할 수 있다. 이 방식은 사용자 지정 자바스크립트에 대한 필요를 줄인다. 그리고 개발은 보다 더 선언적이고, HTML-중심적으로 진행된다.

HTMX의 핵심 특징들:

- **단순성(Simplicity):** HTMX는 HTML 애트리뷰트를 사용해 행위를 서술한다. 그래서 이해와 관리가 쉽다.
- **강력함(Power):** HTMX는 단순하다. 그런데도, 정교한 상호 작용과 실-시간 업데이트가 가능하다.
- **유연성(Flexibility):** 어떠한 백-엔드 기술에서도 사용할 수 있다. 기존 시스템들과도 잘 통합된다.
- **성능(Performance):** HTMX는 가볍다 그리고 빠르다. 그래서 적재 시간과 런타임 성능이 향상된다.

전통적 자바스크립트와 AJAX 등과 비교

전통적인 웹 개발은 종종 상당한 양의 자바스크립트를 작성하게 된다. 그 자바스크립트들은 사용자 상호작용 처리, AJAX 요청 만들기, DOM 업데이트 수행 등을 한다. 이 방식은 복잡하고, 유지보수하기 어려운 코드를 유발한다. 특히, 애플리케이션이 규모와 복잡성이 커가면서 더 심해진다.

HTMX는 이와 다른 방식이다:

- **선언적 방식 vs 명령적 방식:** 내용을 가져와 업데이트 하는 방법을 서술하기 위한 자바스크립트 함수 따로 작성하지 않는다. HTML 안에서 애트리뷰트(attribute)들 안에 직접 선언하면 된다.
- **보일러플레이트가 줄어듬:** HTMX를 사용하면, 공통된 패턴(예: AJAX 호출의 결과를 사용해 div를 업데이트하기)들을 위한 코드가 최소화된다.
- **가독성 향상:** 엘리먼트의 동작들이 해당 HTML 안에 그대로 드러난다. 한눈에 쉽게 이해할 수 있다.
- **유지보수가 더 쉬움:** 동작이 HTML 엘리먼트에 함께 묶여있다. 따라서 대체로 HTMX-기반 코드는 수정이나 유지보수가 더 쉽다.

아래 간단한 예시는 그 차이를 잘 보여준다:

전통적인 자바스크립트/AJAX:

```
<button id="loadButton">내용 적재하기</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">내용 적재하기</button>
<div id="content"></div>
```

보다시피, HTMX 버전이 훨씬 더 간결하다. 그리고 그 의도가 HTML 안에 명확하게 드러난다.

핵심 개념들

효과적으로 HTMX를 사용하려면, 그 핵심 개념들 그리고 그 개념들이 어떻게 서로 함께 작동해 동적인 웹 애플리케이션을 만들어 내는지를 이해하는 것이 매우 중요하다.

hx-get: 내용 가져오기(GET 요청을 통해)

`hx-get` 애트리뷰트를 사용하면, GET 요청들을 서버에게 보내고, 그에 대한 응답을 사용해 페이지를 업데이트 할 수 있다. 사용자가 `hx-get` 애트리뷰트가 있는 엘리먼트와 상호작용(디폴트는 '클릭'이다)을 하면, HTMX는 명시된 URL로 가는 GET 요청을 만든다. 그리고 그 응답을 사용해 타겟 페이지를 업데이트한다.

예시:

```
<button hx-get="/api/user" hx-target="#user-info">
  사용자 정보 적재하기
</button>
<div id="user-info"></div>
```

이 예시에 있는 버튼이 클릭되면, HTMX는 GET 요청을 만들어 `/api/user` 로 보낸다. 그리고 받은 응답을 id가 `user-info`인 div 안에 넣는다.

hx-target: 목표 엘리먼트 지정하기(그곳에 응답이 들어간다)

앞의 예시에서 봤듯이, `hx-target` 애트리뷰트는 서버에서 받은 응답을 업데이트할 대상 엘리먼트가 무엇인지를 명시한다. 이것이 명시되지 않을 수 있다. 그러면, 디폴트 즉, 그 요청을 발동한 엘리먼트가 타겟이므로 그 엘리먼트를 업데이트한다.

예시:

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

위 경우, 응답을 `/api/notification` 로부터 받는다. 그 응답은 id가 `notification-area`인 div 안에 들어간다.

hx-post: 데이터 제출하기(POST 요청을 통해)

`hx-get`과 비슷하게, `hx-post` 애트리뷰트는 POST 요청을 만든다. 사용되는 전형적인 경우는 form 데이터를 제출할 때이다. 즉, 데이터를 서버에게 보내 그 서버의 상태를 변경하고자 할 때 사용된다.

예시:

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

사용자가 이 form을 제출하면, HTMX는 POST 요청을 만들고 `/api/submit`에게 그 form의 데이터를 담아 보낸다. 그리고 응답을 서버로부터 받으면 id가 `"response"`인 div 안에 넣는다.

hx-put, hx-patch, hx-delete 요청들

이 기타 요청들 역시 `hx-get`, `hx-post`와 같은 방식을 따른다. 즉, `hx-put`, `hx-patch`, `hx-delete`는 각각 PUT, PATCH, DELETE 요청을 보낸다. 즉, 이것들이 사용되면, 변경 또는 삭제를 해당 백엔드에 제출한다 (보다 자세한 사항은 뒤에 설명한다).

hx-swap: 내용이 어떻게 뒤바뀌는지 그 방식을 제어하기

`hx-swap` 애트리뷰트는 새 내용이 타겟 엘리먼트 안으로 어떻게 채워지는 지 그 방식을 여러분이 제어할 수 있도록 한다. 가장 흔한 옵션들은 다음과 같다:

- `innerHTML` (디폴트): 그 타겟 엘리먼트의 ‘안에 있는 HTML’을 교체한다
- `outerHTML`: 그 타겟 엘리먼트를 통째로 교체한다
- `beforebegin`: 그 타겟 엘리먼트가 시작되기 바로 전에 삽입한다
- `afterbegin`: 그 타겟 엘리먼트에서 가장 앞에 있는 자식이 되도록 삽입한다
- `beforeend`: 그 타겟 엘리먼트에서 가장 뒤에 있는 자식이 되도록 삽입한다
- `afterend`: 그 타겟 엘리먼트가 끝나고 난 바로 뒤에 삽입한다

예시:

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

위 코드는 이 list 중 맨 뒤에 새 항목을 덧붙인다. 이 list 전체를 교체하는 게 아니다.

추가적인 핵심 개념들

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

- **hx-trigger**: 무슨 이벤트가 그 AJAX 요청을 발동하지를 여러분이 명시할 수 있다. 디폴트는, 대부분의 엘리먼트들인 경우에는 Click 이벤트다. 또는 form인 경우에는 Submit 이벤트다.
- **hx-params**: 무슨 파라미터들을 그 요청과 함께 제출할 것인지를 여러분이 제어할 수 있다.
- **hx-headers**: 사용자 지정 헤더들을 그 AJAX 요청 안에 여러분이 추가할 수 있다.
- **hx-vals**: 추가적인 값들을 파라미터로 추가해 그 요청과 함께 제출할 수 있다.
- **hx-boost**: 일반 앵커(anchor)들과 폼(form)들을 강화하는 간단한 방식 (AJAX를 반영함으로써)
- **hx-push-url**: 새 URL을 이력 스택에 추가한다. 그래서 그 브라우저의 URL들을 업데이트 할 수 있다. 그래서 전체 페이지를 적재하지 않은 경우에도 스택 안에 넣을 수 있다.

이 핵심 개념들을 이해했다면, 여러분은 이제 HTMX를 사용해 동적으로 상호작용하는 웹 애플리케이션을 구축하는 견고한 기반을 갖추었다. 이 기초들에 더 익숙해질 수록, 보다 수준 높은 HTMX를 활용할 수 있다. 그래서, 정교하고 반응성이 좋은 UI를 제공할 수 있다. 자바스크립트 사용은 최소화된다.

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

02

WebBroker(웹브로커) 소개

.....

WebBroker(웹브로커) 란?

WebBroker는 강력한 프레임워크다. RAD Studio 안에 들어 있다. 이 프레임워크를 사용하면 개발자들이 견고한 웹 애플리케이션과 웹 서비스를 만들 수 있다. 이 프레임워크는 단단한 기반을 제공한다. 그래서 여러분은 서버-쪽 애플리케이션을 구축할 수 있고 RESTful 서비스, SOAP 서비스 등등을 개발할 수 있다.

WebBroker의 주요 특징들

- **다목적성:** WebBroker는 다양한 유형의 웹 서비스들을 지원한다(독립 실행, Apache, ISAPI, FastCGI 등등). 즉, 용도에 맞게 선택할 수 있다. 그래서 다양한 프로젝트 요구사항들을 맞출 수 있다.
- **통합:** RAD Studio와 통합된다. 그래서 익숙한 개발 환경을 델파이와 C++빌더 개발자들에게 제공한다.
- **확장성:** WebBroker 애플리케이션들은 쉽게 확장될 수 있다. 따라서 증가하는 부하와 복잡한 요구사항들을 다룰 수 있다.
- **성능:** 이 프레임워크는 고성능을 위해 고안되었다. 성능은 서버-쪽 애플리케이션에게 매우 중요하다.

핵심 개념들

WebBroker(웹브로커)의 핵심 개념들을 이해하는 것이 매우 중요하다. 그래야 웹 애플리케이션들과 웹 서비스들을 구축할 때 WebStencils(웹스텐실즈)를 효과적으로 사용할 수 있다.

컴포넌트들 그리고 아키텍처

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

- **TWebModule**: WebBroker 애플리케이션에서 중심이 되는 컴포넌트다. 다른 WebBroker 컴포넌트들을 담는 컨테이너 역할을 한다. 또한 그 애플리케이션의 전체 흐름을 다룬다.
- **TWebDispatcher**: 들어오는 HTTP 요청들을 알맞은 액션 항목(action item)들에게 전달하는 역할을 한다.
- **TWebActionItem**: 이 클래스는 명시된 URL 엔드포인트들이 어떻게 다루어져야 하는지 정의한다. 각 액션 항목(action item)은 저마다 특정 URL 패턴에 연계된다. 그래서 그 URL에 대한 요청이 오면 무슨 일을 수행해야 하는지를 정의한다.

WebBroker 애플리케이션의 아키텍처는 전형적으로 이런 흐름을 따른다:

1. HTTP 요청이 들어온다
2. **TWebDispatcher**는 그 요청을 알맞은 **TWebActionItem**에게 보내준다(routing)
3. **TWebActionItem**는 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 선택할 수도 있다.
3. 애플리케이션 유형을 고른다: standalone, Apache module 등등.
4. 8080 포트가 여러분의 컴퓨터 안에서 사용되고 있지 않다면, 디폴트를 그대로 두고 계속 진행한다.

WebModule 액션들을 여러분의 라우터라고 생각해도 좋다. 거기에 모든 엔드포인트들을 정의하게 된다.

여러분의 **TWebModule**을 구성하고 **TWebActionItem**들을 추가해 여러 URL 엔드포인트들을 다루도록 하자. 먼저, **TWebModule** 안 어느 곳이든 클릭한다. 그리고 Properties 메뉴에서 Actions를 선택한다. 메뉴가 나타나면 그 안에서 새 엔드포인트들과 거기에 연계되는 메서드들을 만들 수 있다. 대부분의 RAD Studio 컴포넌트들이 그렇듯이, 액션 항목은 여러 가지 이벤트들을 가지고 있다. 그대로 활용하면 된다.

아래의 간단한 예시는 TWebActionItem을 어떻게 설정하는지 그 방법을 보여준다.

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

이 액션 항목은 간단한 HTML 응답을 만들어 낸다. 그 시점은 자신에게 연계된 URL 요청이 왔을 때이다.

요청들과 응답들을 다루기

WebBroker는 간단명료한 매커니즘을 통해, HTTP 요청들과 응답들을 다룬다:

- **TWebRequest**: 들어오는 HTTP 요청에 관한 모든 정보에 접근한다. 해당되는 요청 관련 정보로는 파라미터들, 헤더들, 그 요청의 메서드 등이 있다.
- **TWebResponse**: 응답 데이터를 설정하는데 사용된다. 해당되는 응답 데이터로는 내용, 상태 코드, 헤더 등이 있다.

아래의 예시는 요청을 접근하고 응답 데이터를 구성하고 있다:

```
procedure TWebModule1.HandleGetUser(Sender: TObject;  
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);  
var  
    UserId: string;  
begin  
    UserId := Request.QueryFields.Values['id'];  
    // (UserId를 기반으로 그 사용자에게 대한) 사용자 데이터를 가져온다  
    Response.ContentType := 'application/json';  
    Response.StatusCode := 200;  
    Response.Content := '{ "id": "' + UserId + '", "name": "홍길동" }';  
    Handled := True;  
end;
```

배포 그리고 확장성 (Deployment and Scalability)

WebBroker 애플리케이션들은 유연한 배포 옵션들을 제공한다:

- **Standalone Executables**: 독립 웹 서버 안에서 실행될 수 있다(폼-기반 또는 CLI).
- **Apache/ISAPI/FastCGI Extensions**: 웹 서버들(IIS, Apache, NGINX)에 통합될 수 있다.

이런 유연성 덕분에 여러분은 다양한 배포 상황에 맞출 수 있다. 그래서 확장성과 성능을 다양한 웹 서비스 유형에서 보장할 수 있다. 처음 시작 단계에는 standalone executable를 선택해 개발과 테스트를 하고, 실제 운영용으로는 ISAPI 확장, FastCGI 애플리케이션, Apache 모듈 등으로 배포해 해당 웹 서버의 전체 기능을 사용하도록 할 수 있다.

세션 관리 (WebBroker 안에서)

세션 관리(Session management)는 역사적으로 개발자가 WebBroker(웹브로커) 애플리케이션들 안에서 직접 구현해야 했던 기능들 중 하나였다. 여러분은 코드를 작성해, 사용자들을 요청들 전반에 걸쳐 추적하고, 어딘가에(메모리, 데이터베이스, 파일들 안에) 그들의 데이터를 저장하고, 세션 만료를 다루야 했다.

좋은 소식이 있다. 이제 RAD Studio(라드 스튜디오)에는 세션 관리 컴포넌트들이 들어 있다. 그것들은 여러분을 위해 그 모든 것을 다뤄준다. 그래서 여러분은 세션 인프라를 구축하는 데 시간을 소비하는 대신, 애플리케이션의 실제 기능에 집중할 수 있다.

세션 관리를 자세히 다루는 곳은 [9장](#)이다. 새 `TWebSessionManager`, `TWebFormsAuthenticator`, `TWebAuthorizer` 컴포넌트에 대해 배울 것이다. 이 컴포넌트들은 WebStencils(웹스텐실즈)와 함께 작동한다. 그래서 완전한 인증 및 세션 시스템을 제공한다. 코드 작성은 최소화된다.

지금은 그냥, 세션 관리가 제공되고 있고, 여러분이 필요할 때 사용하면 된다는 정도만 알고 넘어가자.

보안 고려사항들

웹 애플리케이션들을 구축할 때 보안(Security)은 항상 최우선 순위여야 한다. WebBroker(웹브로커)는 여러분이 보안 모범 관행들을 구현하는 데 필요한 도구들을 제공한다. 하지만 그 도구들을 올바르게 사용하는 것은 여러분의 몫이다. 몇 가지 중요한 보안 고려 사항들을 살펴보자.

CSRF 보호

사이트-간 요청 위조(CSRF, Cross-Site Request Forgery)는 공격의 한 종류다. 그 공격은 악의적인 웹사이트가 사용자의 브라우저를 속여 원치 않는 요청을 여러분의 애플리케이션에게 보내도록 한다. 브라우저는 쿠키들(세션 쿠키들을 포함하여)을 자동으로 모든 요청에 담게 되어 있다. 따라서 여러분이 적절한 보호 조치들을 마련해 놓지 않으면 그 공격이 성공할 수 있다.

CSRF에 대한 표준적인 방어는 토큰(token)들을 사용하는 것이다. 작동 방식은 다음과 같다. 여러분은 폼(form)을 렌더링할 때, 고유하고 무작위적인 토큰(unique, random token)을 숨겨진 필드로 포함시킨다. 그 폼이 여러분의 서버로 제출되면, 여러분은 그 토큰이 예상하는 값과 짝이 맞는지 확인한다. 공격자는 그 토큰을 예측하거나 접근하지 못한다. 따라서, 유효한 요청으로 위장하지 못한다.

이 예시는 WebBroker 안의 구현을 보여준다:

```
function TWebModule1.GenerateCSRFToken: string;
var
    GUID: TGUID;
begin
    CreateGUID(GUID);
    Result := GUID.ToString;
end;

procedure TWebModule1.RenderForm(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
    CSRFToken: string;
begin
    CSRFToken := GenerateCSRFToken;
    // 토큰(token)을 세션(session) 안에 저장한다. 그래서 나중에 검증할 때 사용한다
    Request.Session.DataVars.AddPair('csrf_token', CSRFToken);

    Response.Content := Format('<form method="post" action="/submit">' +
        '<input type="hidden" name="csrf_token" value="%s">' +
        '<input type="text" name="data">' +
        '<button type="submit">Submit</button>' +
        '</form>', [CSRFToken]);
    Handled := True;
end;

procedure TWebModule1.HandleFormSubmission(Sender: TObject;
    Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var
    SubmittedToken, StoredToken: string;
begin
    SubmittedToken := Request.ContentFields.Values['csrf_token'];
    StoredToken := Request.Session.DataVars.Values['csrf_token'];

    if SubmittedToken <> StoredToken then
    begin
        Response.StatusCode := 403;
        Response.Content := '유효하지 않은 CSRF 토큰';
        Handled := True;
        Exit;
    end;

    // 그 폼(form) 데이터를 가지고 처리를 진행한다
    Response.Content := '폼(Form) 제출 성공';
    Handled := True;
end;
```

위 구현 안에서:

1. 우리는 고유한 CSRF 토큰을 만들어 낸다. 각 폼(form)이 제출될 때마다 그렇게 한다.
2. 그 토큰은 세션(session) 안에 저장된다. 그리고 숨겨진 필드(hidden field)로 폼 안에 담는다.
3. 폼이 제출되어 우리가 그 제출들을 처리할 때, 우리는 요청으로부터 온 그 토큰을 검증한다. 즉, 폼에서 제출된 토큰과 세션 안에 저장된 토큰을 대조한다.

데이터 검증(Data Validation)

데이터가 유효한지 검증(validate)하고 세척(sanitize)하는 것을 항상 서버 쪽에서 해야 한다. 클라이언트-쪽 유효성 검증(validation)이 마련되어 있다고 해도 그렇게 해야 한다. 클라이언트-쪽 유효성 검증은 사용자 경험 면에서는 훌륭하다. 하지만 그것은 우회될 수 있다. 서버는 여러분의 마지막 방어선이다.

예시:

```
procedure TWebModule1.ValidateUserInput(const Username: string);
begin
    if Length(Username) < 3 then
        raise Exception.Create('Username은 반드시 3 글자 이상이어야 합니다');

    if not TRegEx.IsMatch(Username, '^[a-zA-Z0-9_]+$') then
        raise Exception.Create('Username은 오직 영문자, 숫자, 밑줄로만으로 구성되어야 합니다');
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);
        // 등록 절차 진행...
        Response.Content := '등록(Registration) 성공';
    except
        on E: Exception do
            Response.Content := '등록(Registration) 실패: ' + E.Message;
        end;
    Handled := True;
end;
```

사이트-간 스크립팅(XSS, Cross-site Scripting)

XSS 공격을 막으려면, 항상 사용자-생성 콘텐츠를 인코딩하거나 이스케이프(escape)한 다음에 여러분의 HTML 응답 안에 담는다. `NetEncoding` 라이브러리는 HTML 문자열들을 이스케이프할 수 있는 다양한 기법들을 제공한다.

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

CSRF, 데이터 검증, XSS 보호를 넘어, 명심해야 할 다른 중요한 보안 관행들이 몇 가지 있다:

- **SQL 인젝션(Injection) 방지:** 데이터베이스와 상호작용할 때, 파라미터화된 쿼리(parameterized query) 또는 준비된 문장(prepared statement)을 사용한다. 절대로 사용자 입력을 SQL 문자열 안에 직접 이어붙이기(concatenate) 하지 않는다.
- **보안 통신(Secure Communication):** HTTPS를 사용해 전송 중인 데이터를 암호화 한다. 특히 중요한 경우는 로그인 폼들, 결제 정보, 기타 민감한 데이터들을 주고 받을 때이다.
- **인증 및 권한 부여(Authentication and Authorization):** 견고한 사용자 인증과 적절한 접근 제어를 구현한다. 사용자들은 오직 그들이 볼 수 있는 권한에 해당되는 리소스들에만 접근할 수 있다는 것을 확실하게 보장한다.
- **비밀번호 보안:** 비밀번호들을 평문으로 저장하거나 MD5와 같은 오래된 알고리즘들로 저장하지 않는다. 적절한 비밀번호 해싱 알고리즘들을 사용한다(Bcrypt, Argon2 등).
- **용량 제한(Rate Limiting):** 용량 제한 구현을 고려한다. 무차별 접근 공격(brute force attacks)과 API 남용을 방지하기 위해서다.
- **오류 처리:** 오류 메시지 안에서 어떤 정보를 노출하는지에 대해 주의한다. 상세한 오류 메시지들은 공격자들에게 여러분의 시스템에 대한 가치 있는 정보들을 제공할 수도 있다.

이 보안 조치들을 구현하는 것은 여러분의 WebBroker(웹브로커) 애플리케이션들의 보안을 상당히 향상할 수 있다. 명심할 점이 있다. 보안은 지속되는 절차(ongoing process)다. 또한, 최신 보안 모범 관행들과 취약점들에 대해 늘 최신 업데이트를 유지하는 것이 중요하다.

DocWiki에서 더 상세한 정보들을 확인할 수 있다: [Using WebBroker](#)

03

여러분의 첫 웹 앱을 개발하기 (WebBroker를 사용)

머릿말

이 장에서, 우리는 여러분의 첫 웹 애플리케이션을 차근차근 만든다. WebBroker를 사용한다. 그 시작으로 간단한 "Hello World" 예시를 만든다. 그리고 나서 기초적인 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 Studio 안에서 새 WebBroker Application을 만든다.
2. 여러분의 WebModule 안에서, 새 WebActionItem을 추가한다. 그리고 이름을 "ActionHello"라고 지정한다.
3. ActionHello 항목을 더블-클릭한다. 그러면 이벤트 핸들러가 만들어진다.
4. 그 이벤트 핸들러를 구현한다. 이렇게 하면 된다:

```

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>반가워, WebBroker 그리고 HTMX!</h1>
        <button hx-get="/greet" hx-target="#greeting">반갑다고 말하기</button>
        <div id="greeting"></div>
      </body>
    </html>
    ''';
  Handled := True;
end;

```

5. WebActionItem을 하나 더 추가한다. 이름은 “ActionGreet”로 한다. 그것의 “PathInfo” 프로퍼티의 값에 /greet를 지정한다. 그리고 그것의 이벤트 핸들러를 구현한다:

```

procedure TWebModule1.WebModule1ActionGreetAction(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<p>반가워! (서버에서 보내준 인사입니다)</p>';
  Handled := True;
end;

```

6. 여러분의 애플리케이션을 실행한다. 웹 브라우저에서 열고 “반갑다고 말하기” 버튼을 클릭한다. 그러면 인사말이 표시된다. 그렇지만, 페이지 전체가 새로 다시 적재되지는 않는다.

이 간단한 예시는 WebBroker가 어떻게 HTML 내용을 제공하는지 그리고 어떻게 HTMX를 사용하면, 동적 요청을 서버에 보낼 수 있는지를 보여준다.

기초적인 ‘할 일 목록’ 앱

더 복잡한 애플리케이션을 만들어 보자: 기초적인 할 일 목록 앱이다. 사용자가 할 일을 넣거나 볼 수 있다:

1. 새 WebBroker Application을 만든다. 또는 앞에서 만든 애플리케이션을 사용한다.
2. 여러분의 프로젝트 안에 새 유닛을 추가한다. 할 일 목록을 담기 위한 유닛이다:

```

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 안에, 필드를 하나 추가한다. To-Do list를 위한 필드이다:

```
FTodoList: TTodoList;
```

그 WebModule의 생성자 안에서 이것을 초기화(initialize)한다. 그리고 소멸자 안에서 해제(free)한다.

4. WebActionItem들을 추가한다. 할 일 목록 표시, 새 항목 추가, 목록 업데이트를 하는 것들이다. 그 이벤트 핸들러들을 구현한다. 이렇게 하면 된다:

```

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>할 일 목록</h1>
            <form hx-post="/add-todo" hx-target="#todo-list">
                <input type="text" name="todo-text" placeholder="새 할 일 항목">
                <button type="submit">추가하기</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. 사용자는 이제 새 할 일을 추가할 수 있다. 할 일 목록은 동적으로 업데이트 된다. 그렇다고 해서 페이지 전체가 다시 적재되지는 않는다.

이 기초적인 할 일 목록 앱은 WebBroker를 사용해 어떻게 여러 가지 요청 유형들(GET을 통한 목록 표시, POST를 통한 항목 추가)을 다루는 지 그 방법을 보여준다. 또한 HTMX 사용해, 어떻게 동적인 UI를 만들어 서버와 상호작용하는 지도 보여준다.

기억할 점이 있다. 실제 활용되는 애플리케이션이라면, 아마 에러 처리, 입력 유효성 검사, (아마도) 할 일 목록을 데이터베이스에 보존 등을 하고 싶을 것이다. 하지만, 이 예제는 여러분의 이해를 도와주는 좋은 시작점이다. WebBroker와 HTMX가 어떻게 함께 작동해, 대화형 웹 애플리케이션을 만들어 내는지를 이해할 수 있기 때문이다.

04

고급 애트리뷰트들 및 보안 (HTMX를 사용)



머릿말

이 장에서, 우리는 수준높은 애트리뷰트들 몇가지를 살펴본다. HTMX 안에 있는 것들이다. 또한 중요한 보안 사항들도 논한다. 이것들은 여러분이 HTMX와 WebBroker를 사용해 웹 애플리케이션을 구축할 때 고려해야 하는 것들이다.

고급 애트리뷰트(attribute)들

HTMX는 풍부한 애트리뷰트들을 세트로 제공한다. 그래서 AJAX 요청들과 DOM 업데이트들을 여러분이 정밀하게 제어할 수 있다:

hx-put 그리고 hx-delete: 요청을 제출하기 (PUT 과 DELETE를 통해)

hx-get 그리고 hx-post가 가장 흔하게 사용된다. 그렇지만, HTMX는 다른 HTTP 메서드들 즉 PUT, DELETE 등도 지원한다.

hx-put 사용 예시:

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

hx-delete 사용 예시:

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

여러분의 WebBroker 애플리케이션 안에서, 여러분이 이 요청들을 적절하게 처리해야 한다:

```
procedure TWebModule1.HandlePutRequest(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  if Request.MethodType = mtPUT then
  begin
    // PUT 요청 다루기
    Response.Content := '사용자를 업데이트했습니다';
    Handled := True;
  end;
end;

procedure TWebModule1.HandleDeleteRequest(Sender: TObject;
  Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
  if Request.MethodType = mtDELETE then
  begin
    // DELETE 요청 다루기
    Response.Content := '사용자를 삭제했습니다';
    Handled := True;
  end;
end;
```

hx-trigger: 이벤트 트리거들을 커스터마이징 하기

hx-trigger 애트리뷰트를 사용하면, 어떤 이벤트가 그 AJAX 요청을 발동하는지를 여러분이 지정할 수 있다. 디폴트는, 대부분의 엘리먼트들에서 Click 이벤트이다. form인 경우에는 Submit 이벤트이다.

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` 애트리뷰트를 사용하면, 서버의 응답 중 원하는 하위 집합을 여러분이 선택할 수 있다. 그래서 그 선택된 응답을 가지고 타겟을 업데이트 할 수 있다.

예시:

```
<button hx-get="/api/user"
      hx-target="#user-name"
      hx-select="#name">
  사용자 이름 적재하기
</button>
```

이 경우, 서버에서 온 응답 중에서 id가 "name"인 엘리먼트만 뽑아, 해당 타겟을 업데이트 할 때 사용한다.

이 방식을 사용하면, 여러분의 서버는 전체 HTML 페이지들을 반환하도록 허용(HTMX가 아닌 요청들 또는 디버깅을 할 때 유용)할 수 있으면서도, 여전히 HTMX에 의해 그 웹페이지의 일부만 골라서 업데이트 할 수 있다.

알아두면 좋은 점이 있다. HTML 문서 전체를 반환하는 것은 가능하다. 그 방식이 유용한 경우도 가끔 있다. 그렇지만, HTMX로 작업하는 경우 중 많은 상황에서, 여러분은 여러분의 서버가 꼭 필요한 특정 HTML 조각들만을 반환하도록 하는 방식을 더 선호하게 될 것이다.

hx-include: 부가적인 데이터를 요청 안에 포함하기

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

예시:

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

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

hx-push-url: 브라우저의 URL을 업데이트하기

`hx-push-url` 를 사용하면 브라우저에 전체 페이지 적재를 하지 않고도 그 브라우저의 URL을 여러분이 업데이트 할 수 있다. 이것은 단일-페이지 애플리케이션들 안에서 이동 능력을 유지하고 싶을 때 유용하다.

예시:

```
<button hx-get="/new-page"
        hx-push-url="true">
  다음 페이지로 이동하기
</button>
```

이 버튼이 클릭되면, 브라우저의 URL을 업데이트 한다. 그래서 앞/뒤로 이동이 올바르게 수행되도록 한다.

05

웹스텐실즈(WebStencils) 소개

.....

WebStencils(웹스텐실즈)란?

WebStencils(웹스텐실즈)는 서버-쪽에 있는 스크립트-기반 통합 기술이다. 그래서 HTML 파일들을 다룬다. 이 기술은 RAD Studio 12.2에 처음 도입되었다. 이를 통해, 개발자들은 현대적이고 전문적인 모양을 갖춘 웹사이트를 어떠한 CSS와 자바스크립트든 원하는 것을 기반으로 개발할 수 있다. 또한 RAD Studio로 만든 서버-쪽 애플리케이션에서 추출하고 처리하는 데이터를 그 웹사이트에 반영할 수 있다.

핵심 개념

WebStencils를 사용하면, 탐색용 웹사이트를 개발할 수 있다. 그 사례로는 블로그들, 온라인 카탈로그들, 온라인 주문 시스템들, 그리고 사전이나 위키(wiki) 등과 같은 참조 사이트들이 있다. WebStencils는 템플릿 엔진을 제공한다. 그 엔진은 ASP.NET Razor 처리 방식과 닮았다. 그런데, RAD Studio용으로 특별히 설계된 것이다.

WebStencils(웹스텐실즈)의 핵심 장점은 여러분에게 이미 친숙한 Delphi(델파이) 오브젝트들 및 데이터 구조들을 가지고 작업할 수 있고 HTML 템플릿들 안에 바로 반영할 할 수 있다는 점이다. 여러분은 완전히 새로운 패러다임을 배울 필요가 없다. 또는 여러분의 데이터를 특별한 포맷들로 변환할 필요도 없다. 만약 여러분이 Delphi 오브젝트들을 가지고 있다면, 그 오브젝트들의 프로퍼티(property)들을 여러분의 템플릿 안에서 접근할 수 있다. 단순하고 직관적인 문장을 사용하면 된다.

HTMX와 통합하기

WebStencils는 HTMX를 잘 보완한다. HTMX 페이지들은 서버-쪽 코드 생성이 주는 혜택을 누릴 수 있고, REST 서버들로 연결(hook)되어 내용을 업데이트 할 수도 있다. 한편, Delphi의 웹 기술들은 품질이 좋은 페이지 생성 그리고 REST API를 제공할 수 있다.

CSS와 JS를 가리지 않음

WebStencils(웹스텐실즈)의 핵심 기능 중 하나는 여러분에게 그 어떤 특정한 자바스크립트(JavaScript)나 CSS 라이브러리 사용도 강요하지 않는다는 점이다. 이것은 그저 서버-쪽 렌더링용 템플릿 엔진일 뿐이다. 그래서 여러분은 선호하는 그 어떤 프론트-엔드(front-end) 기술들을 얼마든지 사용할 수 있다.

Bootstrap(부트스트랩)을 사용하고 싶은가? 그렇게 하라. Tailwind CSS(테일윈드 CSS)를 선호하는가? 그것도 할 수 있다. 차트들이나 애니메이션들을 위해 특정한 자바스크립트(JavaScript) 라이브러리를 통합해야 하는가? 문제없다. WebStencils는 여러분의 방식을 방해하지 않는다. 그리고 여러분이 이미 알고 사랑하는 도구들을 가지고 구축할 수 있도록 내버려 둔다.

WebStencils의 문장 구조(Syntax)

WebStencils는 단순한 문장 구조(syntax)를 사용한다. 이 두 가지 요소가 그 기반이다:

1. @ 기호
2. 중괄호 (블록을 표시하기 위한 용도) { }

@ 기호

@ 기호는 특별한 표시자다. WebStencils 안에서 처리된다. 그 뒤에는 이런 것들이 따라 나올 수 있다:

- 오브젝트 이름 또는 데이터셋 이름
- 특별한 처리를 수행하는 키워드
- 또 다른 @

예시:

```
@object.value
```

위 문장은 `object`의 `value` 프로퍼티에 접근한다. 이 오브젝트 이름은 심볼로 사용되는 로컬 이름이다. 이 이름은 실제 서버 애플리케이션의 오브젝트 이름과 쌍(match)을 이룰 수 있다. 또는 코드 안에서 해석(resolved)될 수도 있다. `OnValue` 이벤트 핸들러 코드에서 안에서 처리하면 된다.

만약 여러분이 @ 기호 자체를 출력해야 한다면(예를 들어, 이메일 주소 안에 넣어야 한다면), 그저 이스케이프(escape)하면 된다. 즉, 두 개의 @ 기호를 연속으로 사용하면 된다:

```
<p>우리에게 연락하려면 support@example.com로 메일을 보내주세요</p>
```

중괄호 (블록용) {}

중괄호를 사용하면, 조건 블록 또는 반복 블록을 표시할 수 있다. 중괄호는 WebStencils의 특정 조건문 뒤에서 사용되었을 때만, WebStencils에 의해 블록으로 처리된다.

이것을 코드의 경계 표시라고 생각하기 바란다. 조건에 따라 포함되거나 반복되는 코드 블록을 묶는데 사용된다. 만약 중괄호 앞에 웹스텐실즈 키워드가 없다면, 그것은 그저 HTML 안에 있는 평범한 중괄호다.

값을 접근하기 (점 표기법을 사용)

이 예시는 값들이 WebStencils 안에서 어떻게 처리되는지 보여준다:

```
<h2>사용자 프로필</h2>
<p>이름: @user.name</p>
<p>이메일: @user.email</p>
```

점 표기법(dot notation)은 Delphi에서와 똑같이 작동한다. 만약 여러분의 오브젝트가 프로퍼티를 가지고 있다면, 여러분은 점 연산자를 사용해 그 프로퍼티에 접근할 수 있다. RAD Studio 12.3부터는, 심지어 여러분이 프로퍼티들을 계속 이어붙여 나갈(chain) 수도 있다:

```
@order.customer.address.city
```

WebStencils 키워드들 및 예시들

다양한 웹스텐실즈 키워드들을 살펴보자. 예시들도 함께 본다. 이 키워드들을 사용하면, 여러분의 템플릿들이 어떻게 처리되고 무슨 내용이 렌더링되는 지를 여러분이 제어할 수 있다.

@page

@page를 사용하면, 현재 페이지 요청에 있는 여러 프로퍼티들에 접근할 수 있다. 이것 유용한 경우로는 내비게이션 구축, 현재 페이지 이름 표시, 브레드크럼(breadcrumbs) 생성 등이 있다.

@page 오브젝트는 몇 가지 유용한 프로퍼티들을 제공한다:

- `pagename`: 현재 페이지의 이름 (확장자를 제외한 파일 이름)
- `filename`: 현재 템플릿 파일의 전체 파일 이름

- `request_path`: URL로부터 온 전체 요청 경로
- `request_segment`: 요청 경로의 마지막 세그먼트(segment)
- `referer`: 이 페이지를 참조한 페이지의 URL
- `browser`: 사용자의 브라우저에서 보낸 유저 에이전트(user agent) 문자열
- `address`: 클라이언트의 IP 주소

예시:

```
<p>현재 페이지: @page.pagename</p>
<p>요청 경로: @page.request_path</p>
<p>마지막 세그먼트: @page.request_segment</p>
```

`request_path` 그리고 `request_segment` 프로퍼티가 특히 유용한 경우가 있다. 동적인 내비게이션을 구축하거나 사용자가 여러분의 사이트 중 어느 구역을 보고 있는지를 결정할 때다. 예를 들어, 만약 누군가가 `/products/electronics`를 방문한다면, `request_path`는 `/products/electronics` 이다. `request_segment`는 `electronics`이다.

@query

`@query` 키워드를 사용하면, HTTP 쿼리 파라미터들을 여러분의 템플릿 안에서 바로 읽을 수 있다.

예시:

```
<p>당신이 검색한 것: @query.searchTerm</p>
```

이 예시에서, `searchTerm`은 URL 안에 들어있는 파라미터다:

`yourdomain.com?searchTerm=mySearch`

여러분은 어떤 쿼리 파라미터든 이 방식으로 접근할 수 있다. 만약 존재하지 않는 파라미터를 접근한다면, 그냥 비어 있게 된다. 그래서 검색 결과 페이지를 구축하기가 쉽다. 또한, 필터링된 리스트 제공하기 등 URL 파라미터들을 받아서 응답하는 그 어떤 다른 페이지든지 쉽게 구축할 수 있다.

주석 (@* .. *@)

주석은 WebStencils 안에서 `@* *@` 으로 감싼다. 그러면, 결과 HTML 안에 들어가지 않고 생략된다.

예시:

```
@* 이것은 주석이다 그리고 출력 결과 안에 나타나지 않을 것이다 *@  
<p>이것은 출력 결과 안에 나타나지 않을 것이다</p>
```

이 주석들은 HTML 주석들(<!-- -->)과는 다르다. HTML 주석들은 브라우저로 전송된다. 그래서 페이지 소스 안을 보면 드러난다. 웹스텐실즈 주석들은 서버에서 처리된다. 즉, 결코 클라이언트에게 전달되지 않는다. 이것이 유용한 경우는 여러분 자신이나 다른 개발자들에게 메모들을 남길 때이다. 그 메모는 사용자들에게 노출되지 않는다.

@if 그리고 @else

조건 실행을 다룰 때는 @if 그리고 @else를 사용한다.

예시:

```
@if (user.isLoggedIn) {  
  <p>환영합니다, @user.name!</p>  
}  
@else {  
  <p>계속 진행하려면, 로그인하세요.</p>  
}
```

@if 뒤에 오는 조건은 서버에서 평가된다. 그리고 오직 일치하는 HTML 블록만이 최종 출력에 포함된다. 그래서 여러분의 HTML을 깔끔하게 유지하도록 한다. 그리고 불필요한 마크업이 클라이언트에게 전송되지 않도록 한다.

@if not

부정 조건 실행은 @if not을 사용한다.

예시:

```
@if not(cart.isEmpty) {  
  <p>당신은 @cart.itemCount 개 항목을 카트 안에 담고 있습니다.</p>  
}  
@else {  
  <p>당신의 카트는 비어있습니다.</p>  
}
```

@switch

RAD Studio 13 부터는, `@switch` 연산자가 제공된다. 그래서 여러분은 서로 다른 코드 블록들을 실행할 때 표현식의 값을 기반으로 판단할 수 있다. 이 기능은 다중 if-else 문들의 대안이다. 그런데 여러분이 동일한 값을 가지고 여러 가능성들에 대해 확인하는 경우에 더 읽기 좋다는 장점이 있다.

예시:

```
@switch(user.role) {
  @case "admin" {
    <div class="admin-panel">
      <h2>관리자 대시보드</h2>
      <p>당신은 모든 기능들 전체에 완전하게 접근할 수 있습니다.</p>
    </div>
  }
  @case "moderator" {
    <div class="mod-tools">
      <h2>중재자 도구들</h2>
      <p>당신은 내용을 중재하고 사용자들을 관리할 수 있습니다.</p>
    </div>
  }
  @case "user" {
    <div class="user-content">
      <h2>환영합니다!</h2>
      <p>이 사이트를 즐겁게 탐색하세요.</p>
    </div>
  }
  @default {
    <div class="guest-content">
      <h2>환영합니다, 게스트님!</h2>
      <p>더 많은 기능들에 접근하려면 로그인하세요.</p>
    </div>
  }
}
```

`@switch` 연산자가 특히 유용한 경우는, 필드 타입에 기반해 폼을 생성할 때, 서로 다른 사용자 역할들을 처리할 때, 상태 코드들에 기반해 내용을 표시할 때 등이다. `@if` 문을 길게 늘어뜨리는 것보다 훨씬 더 읽기 좋다. 여러분이 동일한 프로퍼티를 여러 값들에 대해 확인하는 경우에 그렇다.

여러분은 또한 `@switch`를 데이터베이스 필드 타입들과 함께 사용할 수 있다. 그래서 동적 폼들을 만들어 낼 때 사용할 수 있다.

예시:

```
@switch(field.dataType) {
    @case "ftString" {
        <input type="text" name="@field.FieldName" maxlength="@field.Size">
    }
    @case "ftInteger" {
        <input type="number" name="@field.FieldName">
    }
    @case "ftDate" {
        <input type="date" name="@field.FieldName">
    }
    @case "ftBoolean" {
        <input type="checkbox" name="@field.FieldName">
    }
    @default {
        <input type="text" name="@field.FieldName">
    }
}
```

`@default` 케이스(case)는 선택 사항이다. 하지만 권장된다. 그것은 모든 기타 상황에 대처하는(catch-all) 역할을 한다. 즉 여러분의 특정한 케이스(case)들과 일치하지 않는 그 어떤 값이 와도 받아서 수행한다.

@ForEach

`@ForEach` 키워드를 사용하면, 열거자(enumerator) 안에 있는 엘리먼트들을 순환할 수 있다.

예시:

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

이것은 `productList` 안에 있는 각 항목을 루프(loop)로 순회한다. 그리고 각 항목마다 `` 엘리먼트를 하나씩 만든다. `var product`는 로컬 변수 선언이다. 그 변수는 루프 블록 안에서 사용될 수 있다.

여러분은 그 어떤 Delphi 컬렉션도 반복해서 순회할 수 있다. 그것이 `GetEnumerator` 메서드를 가지고 있다면 말이다. `TList<T>`, `TObjectList<T>`, `TArray<T>` 컬렉션들 그리고 데이터셋의 필드들도 그 대상에 해당된다. 즉, 여러분이 데이터셋들을 가지고 작업하고 있다면, 여러분은 해당 레코드들을 직접적으로 반복해서 순회할 수 있다.

또 다른 @ForEach 스타일

덜 선언적인 방식인 **@ForEach** 옵션이 있다. 여기에서는 var 선언을 하지 않을 수 있다. 반복 가능한 오브젝트만 직접 지정하면 된다. 그러면 각 반복의 항목은 자동으로 **@loop**라는 이름의 변수에 들어간다.

예시:

```
<ul>
  @ForEach productList {
    <li>@loop.name - @loop.price</li>
  }
</ul>
```

@ForEach (var object in objectlist) 문장이 일반적으로 더 선호된다. 가독성이 더 좋기 때문이다. 하지만 두 방식 모두 작동한다.

맺음말

WebStencils(웹스텐실즈)는 RAD Studio 애플리케이션들 안에서 동적인 웹 페이지들을 만들어 내는 강력한 방법이다. WebStencils 문장 구조를 사용하면, 서버-쪽 로직을 여러분의 HTML 템플릿들 안에 매끄럽게 넣을 수 있다. @ 기호, 중괄호들, 다양한 키워드들 등을 사용해, 여러분은 동적이고 상호작용적인 웹 페이지들을 쉽게 생성할 수 있다.

그 문장 구조는 Delphi 개발자들에게 직관적이고 친숙하도록 설계되었다. 여러분은 일반적인 Delphi 코드에서 하는 것과 똑같이 오브젝트들과 프로퍼티들을 가지고 작업하면 된다. 그런데 이제는 여러분의 HTML 템플릿들 안에서도 직접적으로 그렇게 할 수 있다. 그 덕분에 여러분의 기존 비즈니스 로직을 웹에 노출하는 것을 쉽게 할 수 있다. 모든 것을 다시 새로 작성할 필요가 없다.

여러분이 WebStencils에 더 익숙해질수록, 그 기능들을 함께 사용하는 더 많은 방법들을 알게 될 것이다. 조건문, 루프(loop), switch 연산자를 잘 섞어 쓰면 매우 유연한 방식으로 HTML을 만들어 낼 수 있다. 그 모든 것들은 서버에서 수행된다. 따라서, 여러분의 사용자들은 깔끔하고 빠르게 로딩되는 HTML을 얻게 된다. 불필요하게 클라이언트-쪽에서 처리되지 않는다.

이어지는 장들에서, 우리는 WebStencils의 더 고급 기능들을 탐구할 것이다. 즉, 템플릿들, 배치(layout)들, 그리고 여러분이 웹스텐실즈 컴포넌트들을 효과적으로 사용하는 방법 등을 배울 것이다. 또한 여러분의 데이터를 Delphi(델파이) 코드로부터 여러분의 템플릿들로 전달하는 방법을 알아본다. 그리고, 더 규모가 큰 애플리케이션들을 구조화하는 방법 즉, 레이아웃(layout, 배치) 페이지들과 재사용 가능한 컴포넌트들을 활용하는 방법도 살펴볼 것이다.

06

컴포넌트들, 배치(layout) 옵션들

머릿말

이 장에서, 우리는 WebStencils의 핵심 컴포넌트들을 알아본다. 템플릿들과 레이아웃(layout, 배치)들을 사용해본다. 그리고 흔하게 사용되는 템플릿 패턴들을 논의한다. 이 개념들을 이해하면, 여러분은 더 잘 정돈되고, 유지보수하기 좋고, 재사용하기 좋은 웹 애플리케이션을 WebStencils를 사용해 만들 수 있다.

웹스텐실즈 컴포넌트들

WebStencils 에는 두 가지 주요 컴포넌트들이 있다: 웹스텐실즈 엔진(WebStencils Engine) 그리고 웹스텐실즈 프로세서(WebStencils Processor, 처리기)이다. 이것들은 서로 함께 동작하면서 여러분의 템플릿들을 처리하고 사용자들에게 전달되는 최종 HTML 출력을 생성한다.

엔진(Engine)을 지휘자라고 생각하고 프로세서(Processor)를 작업자라고 생각하면 된다. 엔진은 모든 구성과 라우팅을 관리한다. 한편, 프로세서는 실제 작업 즉 템플릿을 변환해 최종 HTML을 만들어 낸다.

웹스텐실즈 엔진(WebStencils Engine)

웹스텐실즈 엔진은 중심 컴포넌트다. 여러분의 템플릿 처리를 전반적으로 관리한다. 엔진이 사용되는 경우는 크게 두 가지다:

1. **WebStencilsProcessor 컴포넌트들에게 연결되어 있는 경우:** 이렇게 설정한 경우, 엔진은 공유되는 설정들과 동작들을 여러 프로세서들에게 제공한다. 그래서 각 프로세서마다 사용자 정의를 해야 할 필요가 줄어든다.
2. **독립적으로 사용되는 경우:** 필요하면 엔진이 WebStencilsProcessor 컴포넌트들을 생성할 수 있다. 그래서 오직 엔진 컴포넌트만 여러분의 웹 모듈 안에 놓아두어도 된다.

`TWebStencilsEngine`의 주요 프로퍼티들과 메서드들:

- **Dispatcher:** 파일 디스패처 (`IWebDispatch`를 구현한 것)을 명시한다. 텍스트 파일들을 사후-처리할 수 있도록 한다.
- **PathTemplates:** 요청 경로 템플릿들의 모뎀이다. 요청들을 매칭하고 처리하는데 사용된다.
- **RootDirectory:** 그 파일 시스템의 최상위 경로를 명시한다. 상대 경로의 기준이 된다.
- **DefaultFileExt:** 디폴트 파일 확장자를 지정한다 (디폴트는 '.html' 이다).
- **AddVar:** 오브젝트들을 스크립트 변수 목록에 추가한다. 등록된 변수들은 프로세서들이 사용할 수 있다.
- **AddModule:** 오브젝트 하나를 스캔한다. 그 오브젝트의 멤버들 중에 [`WebStencilsVar`] 애트리뷰트 표기가 붙어 있는 것들을 찾고, 찾은 것들을 스크립트 변수 목록에 추가한다.

웹스텐실즈 프로세서(WebStencils Processor)

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

`TWebStencilsProcessor`의 주요 프로퍼티들과 메서드들:

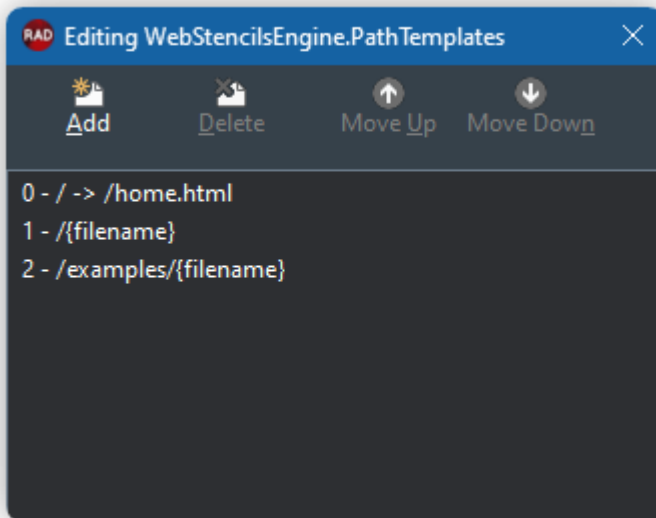
- **InputFilename:** 처리할 파일을 명시한다.
- **InputLines:** 처리할 내용을 직접 할당한다.
- **Engine:** 엔진을 명시한다 (선택 사항). 엔진은 데이터 변수들, 이벤트 핸들러들, 등등을 상속해 준다.
- **Content:** 최종 처리된 내용을 제공한다.
- **AddVar:** 오브젝트들을 스크립트 변수 목록에 추가한다. 등록된 변수들은 그 프로세서가 쓸 수 있다.

`TWebStencilsEngine` 그리고 `WebBroker`(웹브로커)

웹스텐실즈 엔진 컴포넌트를 `TWebFileDispatcher`와 쉽게 연결할 수 있다. 그러면 `WebBroker`가 자동으로 템플릿들을 제공한다.

일단 그렇게 구현을 해 놓은 후에는, 와일드카드를 PathTemplates 프로퍼티 안에서 사용할 수 있다. 그러면 들어오는 요청(request)들에 맞춰 자동으로 파일 매핑(file mapping)이 되도록 할 수 있다.

예시:



각 경로 정의를 분석해보자:

- **0 - / -> /home.html:** 그 웹사이트의 최상위 경로에 대한 접근을 home.html 템플릿에게 보낸다.
- **1 - /{filename}:** 엔진은 URI를 파일 이름에 매핑하려고 한다. 예를 들어, <https://localhost:8080/basics> 로 접근하는 경우, 파일 이름이 basics.html인 템플릿을 찾는다. 그 즉 그 파일이 '사전에 지정된 템플릿 해당 폴더' 안에 들어있으면 그 파일로 연결한다.
- **2 - /examples/{filename}:** 앞의 예시와 동작이 같다. 하지만 이 경우에는 템플릿들을 찾을 때/examples 경로 안을 들여다 본다.

데이터를 추가하기 (AddVar를 사용하기)

AddVar 메서드는 매우 중요하다. 여러분의 델파이 코드에서 여러분의 웹스텐실즈 템플릿에게 데이터를 보내기 위해 꼭 필요하다. AddVar를 사용하는 몇 가지 방식을 보자:

직접 오브젝트를 할당하기

가장 단순한 방식이다. 오브젝트를 직접 전달한다:

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

이렇게 하면 **user** 오브젝트를 여러분의 템플릿들 안에서 사용할 수 있다. 그리고 그 오브젝트의 **public**과 **published** 프로퍼티들에 접근할 수 있다. **@user.propertyName** 문장 구조를 사용하면 된다.

애트리뷰트(Attribute)를 사용하기

여러분의 클래스 정의는 그것의 멤버인 필드, 프로퍼티, 메서드에 **[WebStencilsVar]** 애트리뷰트를 붙일 수 있다. 그런 다음, 표시가 붙은 멤버들을 전체를 스크립트 변수로 추가하려면, **AddModule** 을 사용한다.

```
type
  TMyDataModule = class(TDataModule)
    [WebStencilsVar]
    FDMemTable1: TFDMemTable;
    [WebStencilsVar]
    Users: TObjectList<TUser>;
  end;

// 여러분의 WebModule(웹모듈) 안에서:
WebStencilsProcessor.AddModule(MyDataModule);
```

이것은 깔끔한 방식이다. 여러 오브젝트들을 한 번에 노출할 수 있다. 수많은 개별 **AddVar** 호출들을 작성하지 않아도 된다. 이것이 특히 유용한 경우는, 여러분이 데이터 모듈을 가지고 있고, 그 모듈 안에 있는 여러 개의 데이터셋 또는 오브젝트들을 템플릿들 안에서 사용할 수 있게 하고 싶을 때이다.

커스텀 데이터 액세스 (Lookup 함수들을 이용하기)

버전 13.0에서 추가된 더 강력한 기능들 중 하나는 커스텀 조회(lookup) 함수들을 제공하는 능력이다. 이것이 유용한 경우는, 전통적인 Delphi 오브젝트가 아니라 외부에서 온 데이터를 여러분이 노출하고 싶을 때이다. 또는 프로퍼티들이 해석되는 방식을 여러분이 더 많은 제어해야 하는 경우에도 유용하다.

이 예시는 딕셔너리(dictionary)를 사용해 환경 구성 값들을 제공하고 있다:

```
type
  TMap = TDictionary<string, string>;

var
  LDict: TMap;
begin
  LDict := TMap.Create;
  LDict.Add('APP_VERSION', '1.0.0');
  LDict.Add('APP_NAME', 'My Application');
  LDict.Add('DEBUG_MODE', 'True');
```

```
WebStencilsEngine.AddVar('env', LDict, True,
    function(AVar: TWebStencilsDataVar;
        const APropName: string;
        var AValue: string): Boolean
    begin
        Result := TMap(AVar.TheObject).TryGetValue(APropName.ToUpper, AValue);
    end);
end;
```

이렇게 해 놓으면, 여러분의 템플릿들 안에서, 이 값들에 접근할 수 있다. 일반 오브젝트처럼 사용하면 된다:

```
<h1>@env.APP_NAME - @env.APP_VERSION</h1>
@if env.DEBUG_MODE {
    <div class="alert alert-warning">Debug mode is active</div>
}
```

조회(lookup) 함수를 사용하면, 프로퍼티 이름들이 해석되는 방식을 여러분이 완전하게 제어할 수 있다. 이 예시에서, 우리는 프로퍼티 이름을 대문자로 변환한다. 그리고 나서, 그것을 딕셔너리에서 찾는다. 따라서, 템플릿 안에서 호출할 때는 대소문자를 구분하지 않아도 된다.

이 기법이 특히 유용한 경우:

- 구성 값(configuration value)들을 다른 원본에서 읽어올 때
- 계산된 프로퍼티들 즉, 실제 오브젝트 상에는 존재하지 않는 프로퍼티들
- 동적인 데이터, 즉 맥락에 따라 변하는 데이터
- 외부 시스템들 또는 외부 API들과의 통합할 때

이 ‘조회 함수’ 호출은 WebStencils가 오브젝트의 프로퍼티에 접근을 시도할 때마다 발생한다. 만약 그 함수가 **True**를 반환한다면, **AValue** 안의 값을 사용한다. 만약 **False**를 반환한다면, WebStencils는 일반적인 RTTI 프로퍼티 조회를 시도한다(fallback 즉 대비책이 수행되도록 되어있음).

중요한 고려사항들

명심하자. **@ForEach** 루프(loop)에서 사용하려면 그 오브젝트가 반드시 **GetEnumerator** 메서드를 가지고 있어야 한다. 그 에뮬레이터 함수가 오브젝트 값(object value)을 반환해 주어야 하기 때문이다. 레코드를 WebStencils에서 사용할 수는 없다. 레코드는 필요한 RTTI 기능을 지원하지 않기 때문이다.

여러분이 전달하는 오브젝트에 **AOwned**가 **True**로 설정되어 있는 경우, WebStencils가 소유권을 가진다. 그러면, 더 이상 필요하지 않을 때 그 오브젝트를 해제할 것이다. 만약 **AOwned**가 **False**로 설정되어 있다면, 그 오브젝트의 수명 주기를 여러분이 책임지고 관리해야 한다.

배치(layout), 내용 자리표시자(Content Placeholder)들

WebStencils는 강력한 배치 체계(layout system)를 제공한다. 이는 다른 템플릿 엔진들 즉 [Mustache](#), [Blade](#), [ERB](#), [Razor](#) 등과 비슷하다. 이 체계를 사용하면, 여러분의 페이지들을 위한 공통 구조를 여러분이 정의할 수 있고 그 구조 안에 특정 내용들을 끼워넣을 수 있다.

이 배치 체계 덕분에 여러분은 동일한 HTML 구조를 모든 페이지마다 반복하는 것을 피할 수 있다. 즉, 공통 구조를 하나의 배치 템플릿(layout template) 안에 한 번만 정의할 수 있다. 그런 다음, 그 배치 템플릿의 가변 부분들 안에 여러분이 넣고 싶은 내용 페이지들을 집어 넣으면 된다.

@RenderBody

`@RenderBody` 지시어는 ‘배치 템플릿’ 안에서 사용된다. 이것은 ‘특정 페이지에서 온 내용’이 삽입되어야 하는 위치를 표시한다. 예를 들어, 우리가 이런 공통 HTML 구조를 가지고 있다고 생각해보자. 그리고 이 파일 이름이 `BaseTemplate.html`이라고 가정하자:

```
<!-- 이 파일 이름은 the BaseTemplate.html 이다-->
<!DOCTYPE html>
<html>
<head>
  <title>나의 웹사이트</title>
</head>
<body>
  <header>
    <!-- 공통 header 내용 -->
  </header>

  <main>
    @RenderBody
  </main>

  <footer>
    <!-- 공통 footer 내용 -->
  </footer>
</body>
</html>
```

위에 있는 `@Renderbody` 키워드는 다른 것으로 교체될 것이다. 즉, 이 자리에는 다른 ‘자식 템플릿’ 안에 들어 있는 내용이 들어가게 된다.

@LayoutPage

`@LayoutPage` 지시어는 ‘내용 페이지’ 안에서 사용된다. 그 내용 페이지는 자신의 구조로 사용할 배치 템플릿이 무엇인지를 명시한다. 내용 파일의 맨 위에 명시하는 것이 전형적인 형태다:


```

<!-- BaseTemplate.html 파일이 기반 구조로 사용된다. 그리고 여기에 있는 나머지 내용들은
@RenderBody 태그가 있는 위치에 담긴다 -->
@LayoutPage BaseTemplate
<h2>나의 페이지에 온 것을 환영합니다</h2>
<p>이것은 나의 페이지 내용입니다.</p>

```

이 예시는, `BaseTemplate.html` 파일을 기반 배치로 사용하라고 명시하고 있다. `@LayoutPage`를 명시한 줄 아래의 내용들은 우리가 앞에서 본 예시인 `BaseTemplate.html` 파일 안 `@RenderBody`가 있는 위치에 렌더링된다.

이 방식을 사용하면, 여러분의 콘텐츠 페이지들은 실제 내용에만 집중할 수 있다. 주변 구조를 신경쓸 필요가 없다.

@Import

`@Import` 지시어를 사용하면, 외부 파일을 현재 템플릿 안의 특정 위치로 병합할 수 있다. 이것이 유용한 경우는 재활용할 수 있는 구성요소들을 만들어 사용할 때이다.

이 지시어를 활용하면, 여러분의 템플릿들을 중첩된 폴더들 안에 넣어두고 구조화할 수 있다. 또한 그 파일의 확장자를 생략할 수도 있다. 단, Engine 또는 Processor 안에서 디폴트 확장자를 미리 정의해 놓아야 한다.

```

@Import Sidebar.html

@* 동일한 행위 *@
@Import Sidebar

@* 중첩된 폴더를 활용하는 예시 *@
@Import folder/Sidebar

```

`@Import` 지시어는 가장 강력한 도구들 중 하나다. 여러분의 템플릿들이 DRY(Don't Repeat Yourself, 반복하지 마라) 원칙을 지킬 수 있도록 하기 때문이다. 만약 여러 템플릿들 안에 동일한 HTML 구조를 반복해 넣고 있다면, 그 동일한 구조를 뽑아 하나의 파일로 만들고, 임포트되는 컴포넌트로 사용하면 좋다.

@ExtraHeader, @RenderHeader

`@ExtraHeader` 지시어를 사용하면, 여러분의 HTML 문서의 header 구역 안에 넣을 추가 내용들을 정의할 수 있다. 특히 유용한 경우는, 그 페이지만을 위한 CSS 또는 자바스크립트 파일을 포함하고 싶을 때이다.

작동 방식은 이렇다:

1. ‘내용 페이지’ 안에서, `@ExtraHeader` 를 사용해 추가 헤더 내용을 정의한다.

2. ‘배치 템플릿’ 안에서, `@RenderHeader` 사용해 그 추가 헤더 내용이 삽입될 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>상품 정보</h1>
<div id="product-info">
    <!-- 상품 정보를 여기에 -->
</div>
```

배치 템플릿 (`BaseTemplate.html`):

```
<!DOCTYPE html>
<html>
<head>
    <title>나의 온라인 쇼핑몰</title>
    <link rel="stylesheet" href="/css/main.css">
    @RenderHeader
</head>
<body>
    <main>
        @RenderBody
    </main>
</body>
</html>
```

위 예시에서, ‘내용 페이지(`ProductPage`)’ 안에는 추가 CSS와 자바스크립트 파일을 정의한다. 그 페이지에서 사용하기 위해서이다. ‘배치 템플릿(`BaseTemplate`)’ 안에는 `@RenderHeader` 지시어가 있다. 이것은 내용 페이지에 적혀 있는 그 추가 리소스들이 최종 HTML 출력 안에 포함되도록 보장한다.

이 방식을 사용하면, 여러분이 ‘공통 기반 템플릿 하나’를 가질 수 있다. 그러면서도 개별 페이지에는 ‘각자에게 필요한 리소스 또는 메타 태그들을 추가’할 수 있다. 필요한 만큼 얼마든지 추가할 수 있다.

중첩되는 배치(Nested Layout) 지원

RAD Studio 13.0에는 ‘중첩되는 @ExtraHeader 블록’ 지원이 추가되었다. 이것이 유용한 경우는, 여러분이 복잡한 레이아웃 계층 구조들을 가지고 있을 때이다. 여러분은 기반 배치, 섹션-별 배치, 내용 페이지를 따로 가질 수 있는데, 그 각 구성 요소들은 저마다 자신만의 헤더 콘텐츠를 추가하고 싶을 수 있다.

예를 들어:

BaseLayout.html:

```
<head>
  <link rel="stylesheet" href="/css/base.css">
  @RenderHeader
</head>
<body>
  @RenderBody
</body>
```

ShopLayout.html:

```
@LayoutPage BaseLayout

@ExtraHeader {
  <link rel="stylesheet" href="/css/shop.css">
}

<div class="shop-container">
  @RenderBody
</div>
```

ProductPage.html:

```
@LayoutPage ShopLayout

@ExtraHeader {
  <link rel="stylesheet" href="/css/product.css">
}

<h1>상품 정보</h1>
```

위에 있는 `@ExtraHeader` 블록들 모두가 들어가는 위치는 `BaseLayout.html` 안 `@RenderHeader`가 있는 위치이다. 그리고 올바른 순서대로 들어간다. 그렇게 여러분의 페이지들 사이의 의존 관계를 계층식으로 쌓아나갈 수 있다.

명심할 점이 있다. 여러분은 '`@ExtraHeader` 블록들 여러 개'를 여러분의 콘텐츠 페이지 안에 담을 수 있다. 필요한 만큼 얼마든지 넣으면 된다. 그 모두는 배치 템플릿 안 `@RenderHeader`가 있는 위치에 렌더링된다.

`@LayoutPage`, `@RenderBody`, `@ExtraHeader`, `@RenderHeader` 를 함께 활용하면, 매우 유연하고 유지보수가 용이한 템플릿 구조들을 만들 수 있다.

템플릿 패턴들

`WebStencils`를 사용하면, 몇 가지 공통 템플릿 패턴들을 적용할 수 있다. 그래서 여러분의 애플리케이션의 모습들을 구조화 하는 것이 더 쉽다. 가장 흔한 패턴들을 살펴보자.

표준 배치

이 패턴은 내장된 `@LayoutPage`와 `@RenderBody`를 사용하는 방식이다. 이미 앞에서 본 것이다. 이 방식이 이상적인 경우는, 일관성 있는 구조 하나를 여러분의 사이트 전체에서 유지하면서도 개별 페이지들이 각자 고유한 내용을 제공할 수 있도록 하고 싶을 때이다.

이 패턴을 아마 가장 자주 사용하게 될 것이다. 일관성과 유연성 사이의 균형이 매우 좋기 때문이다.

Header/Body/Footer

이 패턴에서는, 페이지 각각이 개별 템플릿이다. 하지만, header, footer 등 모든 공통 부분들을 공유한다. '배치 페이지 하나'를 사용하는 방식이 아니다. 여러분의 각 페이지 템플릿마다 구조를 잡는 방식이다. 다음과 같다:

```
@Import Header.html

<main>
  <!-- 페이지-고유 내용을 여기에 -->
</main>

@Import Footer.html
```

표준 배치 패턴과 비교하면 이 방식이 더 유연하다. 하지만, 공통 요소들에 대한 관리에 더 신경을 써야 한다. 이것이 유용한 경우는 서로 다른 페이지들이 현저히 다른 구조가 필요하면서도 몇몇 공통 요소들을 여전히 공유해야 할 때이다.

이 방식이 포기하는 것(trade-off)이 있다. `@LayoutPage`가 제공하는 ‘자동화된 구조’라는 장점 일부를 잃는다. 그 대신 얻는 것이 있다. 원하는 컴포넌트들을 각 페이지마다 선택적으로 포함하거나 제외할 수 있다.

재사용 구성요소들

`@Import` 지시어는 매우 유용하다. 애플리케이션 전반에 걸쳐 재사용될 수 있는 개별 구성요소들의 세트를 여러분이 정의할 수 있다. 또한 순환될 수 있는 오브젝트들을 전달할 때도 사용할 수 있다. 심지어 별칭을 정의할 수도 있어서 사용할 구성요소들을 다른 이름으로 정의할 수도 있다. 그러면 원본 구성요소의 이름에 종속적이지 않게 (agnostic) 활용할 수 있다. 예시:

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

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

이 패턴을 사용하면, 여러분의 UI를 더 잘게 더 관리하기 좋은 조각들로 나누어 놓을 수 있다. 그래서 그 조각들을 유지관리하고 재사용하기가 쉽다. 또한 그 조각들을 서로 다른 페이지들에 걸쳐 사용하기에도 좋다.

여기서의 중요한 통찰이 있다. 재사용되는 컴포넌트가 반드시 정적(static)인 스크립트라야 하는 건 아니다. 컴포넌트 안에 데이터를 전달할 수 있다. 심지어 그것들을 루프(loop)들 안에 넣을 수도 있다. 그러면, 컴포넌트들의 유연성과 재사용성이 믿을 수 없을 정도로 커진다.

만약 여러 템플릿들 안에 동일한 HTML 구조를 복사해 붙여넣고 있다면, 비록 그 데이터가 다르더라도 그 동일한 구조를 뽑아 하나의 임포트되는 컴포넌트로 만들고, `@Import` 와 파라미터들을 함께 사용하면 된다.

맺음말

WebStencils 는 웹 애플리케이션 안에 템플릿들과 배치들을 만들 수 있는 유연하고 강력한 체계를 제공한다. WebStencils의 Engine과 Processor를 활용하기, `AddVar` 를 통해 데이터를 여러분의 템플릿에게 전달하기, 배치 지시어들(`@LayoutPage`, `@RenderBody`, `@Import` 등)을 활용하기를 통해, 여러분은 잘 구조화되고, 관리하기 좋고, 재사용할 수 있는 표현(view)들을 여러분의 웹 애플리케이션 안에 만들 수 있다.

우리가 앞에서 살펴본 템플릿 패턴들 – 즉, 표준 배치, header/body/footer, 재사용 구성요소들 – 은 여러분의 표현(view)의 구조를 잡는 방식이고 서로 조금씩 다르다. 여러분은 패턴 하나 또는 패턴의 조합을 선택하면 된다. 그래서 애플리케이션의 니즈와 팀의 작업흐름에 가장 잘 맞는 방식으로 구현할 수 있다.

07

‘할 일 목록’ 앱을 WebStencils로 옮기기

머릿말

이 장에서는, 앞에서 우리가 구축했던 할 일 목록 앱을 가져와 웹스텐실즈 템플릿으로 옮긴다. 그 때는 상수들 안에 순수한 HTML을 넣는 방식을 사용했다. 이 장의 마이그레이션 작업을 통해, WebStencils가 어떻게 여러분의 코드를 더 관리하기 좋고 확장하기 좋게 해주는지를 볼 수 있다. 또한 우리는 몇 가지 추가 기능들도 넣을 것이다. 그래서 이 새 방식이 얼마나 유연한지 보여준다.

이 [깃허브 저장소](#)에 있는 웹스텐실즈 데모를 받아서 프로젝트를 열어보면, 실제로 작동하는 할 일 목록 앱을 받을 수 있다. 그 프로젝트는 이 가이드에 나오는 코드들과 개념적으로 동일한 아이디어를 그대로 구현했다.

HTML 상수들을 템플릿들로 변환하기

HTML 상수들을 사용하지 않도록 웹스텐실즈 템플릿들로 바꾸는 작업부터 시작하자. 우리는 템플릿들을 작게 나누어 만들 것이다. 그래서 각 템플릿이 애플리케이션의 서로 다른 부분들을 담당하도록 한다.

주(Main) 레이아웃 템플릿

맨 먼저, 주 레이아웃 템플릿을 만들자. 이것은 우리가 구축하는 애플리케이션의 기반이 된다.

이름이 `layout.html`인 파일을 하나 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"
></script>
</body>
</html>

```



tip

위 예시에는, CDN들이 제공하는 여러 라이브러리들이 들어 있다. 그 링크들에는 각 고유 버전이 명시되어 있다. 이처럼 특정 버전을 명시할 때는, @ 기호로 이스케이프하는 것이 중요하다. 즉 @@라고 적어야 한다. 그래야 WebStencils가 변수라고 인식하지 않는다.

‘할 일 목록’ 템플릿

이제, ‘할 일 목록’ 페이지를 위한 템플릿 하나를 만들자. 이름이 `todo-list.html`인 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">완료처리</button>
            }
            <button class="btn btn-sm btn-danger"
                hx-delete="/delete/@todo.Id"
                hx-target="#todo-list">삭제하기</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">추가하기</button>
    </div>
</form>

```

WebModule(웹모듈)을 업데이트하기

이제, 새 템플릿을 사용하도록 우리의 WebModule을 업데이트하자. 아래 코드를 잘 보면, **TTodoList** 클래스가 확장되었다. 즉 **Delete**, **Complete**, **GetAllItems** 같은 메서드들이 더 추가되었다. 그 코드들은 여기에 적어 놓지 않았다. 순수한 델파이 코드이므로 굳이 이 가이드를 복잡하게 하고 싶지 않았기 때문이다. 깃허브에 있는 데모 프로젝트를 받아 보면, 아래와 비슷하게 코드가 작성된 것을 볼 수 있다.

```

unit TodoWebModule;

interface

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

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;
    TemplateFolder: string;
    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);
    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
    // 이것은 WebBroker 액션이다. 이것은 모든 요청들을 다룬다.
    // 유지관리성을 더 향상하려면, 각각 다른 액션들로 쪼개는 것이 좋다.
    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 := TemplateFolder + '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
    // ItemId은 요청의 PathInfo로부터 추출한다 그리고 int(정수)로 변환한다
    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;  
  
end.
```



Note

위 코드는 매우 단순화해 놓은 것이다. 읽어서 더 잘 이해할 수 있도록, 엔드포인트들을 연결하는 Action들을 사용하지 않았다. 데모는 깃허브에서 받을 수 있다. 그 데모는 더 관리하기 좋도록 [MVC](#) (Model-View-Controller) 방식을 사용하고 있고, 로직이 더 잘 추상화되어 있다.

추가 기능들을 더하기

이제 WebStencils로 마이그레이션하는 작업을 마쳤다. 이제, 추가 기능 몇 가지를 넣어 보자. 이 새 구조가 가지는 확장 능력과 유지보수 능력을 볼 수 있을 것이다.

할 일 카테고리(들)

할 일들을 카테고리로 나누는 기능을 추가하자. 먼저, `T_TODOItem` 클래스를 업데이트하자. 그것은 `T_TODOList` 유닛 안에 있다:

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

이제, `todo-list.html` 템플릿을 업데이트해 카테고리들을 포함시켜 보자:

`HandleAdd_TODO` 메서드를 업데이트 한다. 그것은 `WebModule` 안에 있다:

```

@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">완료처리</button>
          }
          <button class="btn btn-sm btn-danger"
            hx-delete="/delete/@todo.Id"
            hx-target="#todo-list">삭제하기</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">추가하기</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 일들을 필터링하기

할 일을 카테고리에 따라 필터링 하는 기능을 추가해 보자. 이름이 `category-filter.html` 인 파일을 새로 만든다:

```

<div class="mb-4">
  <h5>카테고리 별로 필터링</h5>
  <div class="btn-group" role="group">
    <button class="btn btn-outline-primary"
      hx-get="/" hx-target="#todo-list">전체</button>
    @ForEach (var category in Categories) {
      <button class="btn btn-outline-primary"
        hx-get="/filter/@category"
        hx-target="#todo-list">@category</button>
    }
  </div>
</div>

```

`todo-list.html` 템플릿을 업데이트 한다. 그래서 카테고리 필터를 포함하도록 한다:

```

@LayoutPage layout.html

@Import category-filter.html

<div id="todo-list">
  <!-- ... 이미 있는 할 일 목록 페이지의 내용 ... -->
</div>

<!-- ... 이미 있는 폼(form) ... -->

```

새 매서드를 추가한다. 그래서 이 WebModule이 필터링을 처리하도록 한다:

```

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;

```

WebModuleDefault 메서드를 업데이트한다. 새로 추가된 필터 경로를 처리하도록 한다:

```

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)
    // ... 이미 있는 경로들 ...
end;

```



warning

이 가이드의 초점은 WebStencils와 그것을 RAD 스튜디오에서 사용하는 방법이다. 위 코드, 즉 카테고리 필터링을 위해 필요한 추가 로직 등을 구현하는 코드는 그저 Delphi 코드다. 여기에는 WebStencils에 대한 코드가 아닌 Delphi 코드를 적어놓지 않았다. 왜냐하면 단순해야 그 코드 조각들을 이해하기가 더 좋기 때문이다.

맺음말

이 장에서, 우리는 ‘할 일 목록’ 앱을 마이그레이션했다. HTML 상수들을 웹스텐실즈 템플릿들로 옮겼다. 그래서 우리의 코드는 더 관리하기 좋고 읽기도 더 쉬워졌다. 또한 우리는 관심사들을 따로 분리했다 (Separation of Concerns). 그래서 이제는 쉽게 변경할 수 있다. 델파이 코드를 건드릴 필요가 없어졌다.

또한, 우리는 이 새 방식의 확장성을 볼 수 있었다. 즉, 작업 카테고리나 필터링 등 새 기능을 쉽게 추가할 수 있었다. 추가 기능 구현 과정은 명료했다. 웹스텐실즈 템플릿이 가진 유연성 덕분이다.

08

WebStencils가 제공하는 고급 옵션들

머릿말

이 장에서, 우리는 WebStencils가 제공하는 더 고급 기능들 몇 가지를 살펴본다. 이 기능은 여러분에게 능력과 유연성을 더해준다. 복잡한 웹 애플리케이션을 만들 때 유용하다. 이 기능들은 모든 프로젝트에서 필요한 것들이 아니다. 하지만, 이런 기능들이 있다는 것을 알고 있으면, 필요한 상황에 처했을 때 여러분의 시간을 크게 아낄 수 있다.

@ 키워드를 더 수준높게 이용하는 방법을 살펴보겠다. 표현식 평가(expression evaluation), 동적으로 HTML을 만들어 내기, OnValue 이벤트를 사용해 사용자 정의를 통해 값을 풀어내기 등이다.

표현식 평가(Expression Evaluation): @()를 사용하기

지금까지 우리는 프로퍼티를 접근하는 기본적인 방식 즉 `@object.property` 문장 구조를 사용했다. 이 방식은 대부분의 상황에서 사용된다. 하지만 때때로 더 복잡한 계산들, 메서드 호출들, 배열 인덱싱, 여러 값들을 결합하기 등이 필요할 수 있다. 이러한 상황에서 `@()` 문장 구조가 꼭 필요하다.

`@()` 문장 구조는 Delphi의 LiveBindings(라이브바인딩) 표현식 평가기를 사용한다. 그 평가기는 단순한 프로퍼티 접근을 훨씬 뛰어 넘는 복잡한 표현식들을 지원한다.

@() 가 할 수 있는 것들

메서드 호출:

표준 RTL 함수들을 사용하는 것이 가능하다. 즉 날짜 형식 지정, 문자열 조작, 수학 함수들 등을 사용할 수 있다. 뿐만 아니라, 여러분의 클래스들 안에 직접 정의해 놓은 public 함수들을 사용할 수도 있다:

```
<p>날짜에 형식 반영: @(FormatDateTime('yyyy-mm-dd', order.Date))</p>
<p>대문자로 출력하기: @(UpperCase(customer.FirstName))</p>
<p>반올림하기: @(Round(product.SalePrice))</p>

<!-- GetFullName() 은 고객 오브젝트 안에 있는 public 메서드다 -->
<p>Public 메서드 클래스: @(customer.GetFullName)</p>
```

산수(arithmetic) 그리고 계산(calculation):

```
<p>VAT 포함 가격: $@(product.Price * product.VatPercentage)</p>
<p>할인: $@(product.basePrice - product.salePrice)</p>
<p>이전 페이지: @(pagination.PageNumber - 1)</p>
```

배열과 리스트(list) 인덱스 처리:

```
<!-- TStringList -->
<p>First category: @(categories.Strings[0])</p>
<p>Selected color: @(colors.Strings[context.userChoice])</p>

<!-- TList<T> -->
<p>Product name: @(products.Items[2].Name)</p>

<!-- 고전적인 정적 배열 (indexed property와 getter가 있어야 함) -->
<p>First item: @(data.ClassicArray[0])</p>
```

중요:

- `TStringList` 를 위해서는, `.Strings[index]` 문장 구조를 사용하라
- `TList<T>` 를 위해서는, `.Items[index]` 문장 구조를 사용하라
- 고전적인 정적 배열 (`array[0..N] of Type`)을 위해서는, 프로퍼티(indexed property)와 게터(getter) 함수가 여러분의 Delphi 클래스 안에 만들어져 있어야 한다. 여러분이 직접 작성해 넣으면 된다.



Note

@() 문장 구조는 약간의 성능 오버헤드가 있다. 단순한 `@object.property` 접근보다 느리다. 왜냐하면, Delphi의 표현식 평가를 런타임에 사용하기 때문이다. 하지만, 이러한 오버헤드는 대부분의 애플리케이션들에서 무시할 수 있는 수준이다.

@Scaffolding

`@Scaffolding` 키워드는 매우 강력하다. 동적으로 HTML을 만들어 낼 때 여러분의 애플리케이션에 있는 데이터 구조들을 기반으로 할 수 있다. 이것이 특히 유용한 경우는 폼을 만들어 내거나 데이터 표를 표현할 때이다. 비슷한 HTML 패턴들을 반복적으로 만들어 내야 하는데, 그 안에 들어가는 HTML을 생성을 하는 비즈니스 로직이 매우 복잡해서 템플릿 하나 안에 담기에 너무 복잡할 때 사용하면 좋다.

하지만, 고유하고(unique) 수작업으로 제작되는 UI들을 위해서는, 일반적인 템플릿 문장 구조가 대체로 더 명확하고 유지보수가 쉽다. 즉, 그럴 때는 `@Import` 키워드를 사용하는 것이 좋다.

기본적인 문장 구조:

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

구조물(scaffolding)을 구현하기 위해서는, `WebStencilsProcessor`의 `OnScaffolding` 이벤트를 여러분이 다루어야 한다:

```
procedure TMyWebModule.ProcessorScaffolding(Sender: TObject;
  const AQualifClassName: string; var AReplaceText: string);
begin
  if SameText(AQualifClassName, 'User') then
  begin
    AReplaceText := '';
    // 폼 필드들을 만들어 낸다. 이때 User 프로퍼티들을 기반으로 만든다.
    AReplaceText := AReplaceText + '<input type="text" name="Username"
placeholder="Username">';
    AReplaceText := AReplaceText + '<input type="email" name="Email"
placeholder="Email">';
    // ... 더 많은 필드들을 추가한다 (필요한만큼)
  end;
end;
```

위 코드는 `Scaffolding` 키워드를 `AReplaceText` 값으로 교체한다.

실용적인 스캐폴딩(Scaffolding) 예시

위 코드 예시는, HTML 태그들을 하드코딩 했다. 단순함을 위해서다. 하지만, 실제 애플리케이션이라면 HTML을 하드코딩하는 게 아니라, 여러분이 반영하고 싶은 클래스 구조에 기반해 프로그램적으로 생성해 내고 싶을 것이다. 여기에 있는 더 정교한 접근 방식을 보자:

```
procedure TMyWebModule.ProcessorScaffolding(Sender: TObject;
  const AQualifClassName: string; var AReplaceText: string);
var
  LContext: TRttiContext;
  LType: TRttiType;
  LProp: TRttiProperty;
  LFieldHtml: string;
begin
  if SameText(AQualifClassName, 'User') then
  begin
    AReplaceText := '<div class="form-group">';

    LContext := TRttiContext.Create;
    try
      LType := LContext.FindType('TUser');
      if LType <> nil then
      begin
        for LProp in LType.GetProperties do
        begin
          // (외부에 공개되지 않는) 내부 프로퍼티들을 건너뛰다
          if not (LProp.Visibility in [mvPublic, mvPublished]) then
            Continue;

          // 알맞은 input 태그를 만들어 낸다. 프로퍼티 타입을 기반으로 만든다.
          case LProp.PropertyType.TypeKind of
            tkInteger:
              LFieldHtml := Format('<input type="number" name="%s" ' +
                'class="form-control">', [LProp.Name]);
            tkString, tkUString, tkLString, tkWString:
              LFieldHtml := Format('<input type="text" name="%s" ' +
                'class="form-control">', [LProp.Name]);
            tkEnumeration:
              if LProp.PropertyType.Handle = TypeInfo(Boolean) then
                LFieldHtml := Format('<input type="checkbox" name="%s" ' +
                  'class="form-control">', [LProp.Name]);
              else
                Continue;
            else
              Continue;
          end;

          AReplaceText := AReplaceText + Format(
            '<div class="mb-3">' +
            '<label class="form-label">%s</label>' +
            '%s' +
            '</div>', [LProp.Name, LFieldHtml]);
        end;
      end;
    end;
  end;
```

```

    finally
        LContext.Free;
    end;

    AReplaceText := AReplaceText + '</div>';
end;
end;

```

이 방식은 RTTI (Runtime Type Information)를 사용해 여러분의 클래스 안을 들여다본다. 그리고 자동으로 알맞은 폼 필드들을 만들어 낸다. 그 User 클래스가 변경되면, 그 폼에도 자동으로 반영된다. 수작업으로 업데이트를 할 필요가 없다.

OnValue 이벤트 핸들러

The **OnValue** 이벤트 핸들러를 사용하면 데이터를 여러분의 템플릿들에 동적으로 제공할 수 있다. 이것이 특히 유용한 경우는 여러분이 즉석에서 값들을 계산해야 할 때이다. 또는 프로퍼티 접근을 가로채서 특별한 처리를 하고 싶을 때에도 유용하다.

이 이벤트가 발동하는 시점은 WebStencils(웹스텐실즈)가 프로퍼티 참조를 만났는데 일반적인 RTTI 조회를 통해 해결할 수 없을 때이다. 그때마다 항상 발동한다. 여러분은 이 기회를 활용해 직접 값을 제공할 수 있다. 또한 여러분이 직접 커스텀 오브젝트들과 프로퍼티들의 계층 구조를 만들어 낼 때도 사용할 수 있다.

기본적인 OnValue 예시

```

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

```

그런 다음, 여러분의 템플릿 안에서, 이렇게 사용하면 된다:

```
<p>현재 시각: @CurrentTime</p>
```

고급 OnValue 상황들

OnValue 이벤트는 보이는 것보다 훨씬 더 강력하다. 여기에 몇 가지 실용적인 시나리오들이 있다:

1. 계산되는 프로퍼티(Computed Property)들

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
  var Handled: Boolean);
begin
  if SameText(ObjectName, 'stats') then
  begin
    Handled := True;
    if SameText(FieldName, 'TotalRevenue') then
      ReplaceText := FormatFloat('$#,##0.00', CalculateTotalRevenue)
    else if SameText(FieldName, 'ActiveUsers') then
      ReplaceText := IntToStr(GetActiveUserCount)
    else if SameText(FieldName, 'ConversionRate') then
      ReplaceText := FormatFloat('0.00%', CalculateConversionRate * 100)
    else
      Handled := False;
    end;
  end;
end;
```

템플릿 사용법:

```
<div class="dashboard">
  <div class="stat">전체 매출액: @stats.TotalRevenue</div>
  <div class="stat">활성 사용자들: @stats.ActiveUsers</div>
  <div class="stat">전환 비율: @stats.ConversionRate</div>
</div>
```

2. 지역화(Localization)/번역(Translation)

```
procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
  const ObjectName, FieldName: string; var ReplaceText: string;
```

```

    var Handled: Boolean);
begin
    if SameText(ObjectName, 'i18n') then
    begin
        ReplaceText := GetTranslation(FieldName, CurrentUserLanguage);
        Handled := True;
    end;
end;

```

템플릿 사용법:

```

<h1>@i18n.WelcomeMessage</h1>
<p>@i18n.IntroductionText</p>
<button>@i18n.GetStartedButton</button>

```

3. 복잡한 로직(logic)에 기반한 조건부 내용

```

procedure TMyWebModule.ProcessorOnValue(Sender: TObject;
const ObjectName, FieldName: string; var ReplaceText: string;
var Handled: Boolean);
begin
    if SameText(ObjectName, 'feature') then
    begin
        Handled := True;
        if SameText(FieldName, 'enabled') then
            ReplaceText := BoolToStr(IsFeatureEnabled(CurrentUser, FieldName), True)
        else if SameText(FieldName, 'available') then
            ReplaceText := BoolToStr(IsFeatureAvailable(CurrentSubscription, FieldName),
True)
        else
            Handled := False;
    end;
end;

```

템플릿 사용법:

```

@if (feature.enabled) {
    <div class="premium-feature">
        <h2>프리미엄 기능</h2>
        <p>이 기능이 제공되는 것은 귀하가 구독을 유지하고 있기 때문입니다.</p>
    </div>
}

```

OnValue vs AddVar

언제 **OnValue** 를 사용하고 언제 **AddVar**를 사용해야 하는지 궁금할 것이다. **AddVar** 를 사용하는 경우는 템플릿이 처리되기 전에 여러분이 이미 존재하고 있는 오브젝트와 그것의 프로퍼티들을 가지고 있어서 그것을 템플릿에서 사용하도록 할 때이다. 이런 상황에서는, 여러분의 비즈니스 도메인 오브젝트에 대해 일반적인 RTTI 프로퍼티 접근이 가능하기 때문이다.

OnValue 를 사용하는 경우는 코드를 통해 여러분이 값들을 계산해야 할 때, 데이터가 오브젝트로 존재하고 있지 않을 때, 프로퍼티 접근을 가로채 커스터마이징하고 싶을 때 (즉, 가상의 프로퍼티 또는 의사(pseudo) 오브젝트를 구현하고 싶을 때) 등이다.

OnValue 는 본질적으로 폴백(fallback, 대비책) 메커니즘(mechanism)이다. WebStencils는 먼저 RTTI를 시도한다. 그런데 만약 프로퍼티를 찾을 수 없다면 그 다음으로 **OnValue** 를 호출한다.

인증(Authentication) 그리고 권한 부여(Authorization)

마지막으로, @LoginRequired에 대해 말하고자 한다: WebStencils 초기 버전은, **@LoginRequired** 키워드 그리고 수작업 사용자 인증(authentication) 확인을 여러분이 직접 하도록 권장했다. 하지만, RAD Studio 13.0 부터, WebStencils에는 광범위한(comprehensive) 세션 관리 및 인증 체계가 내장되었다. 이 체계 덕분에 이제는 수동 인증 처리가 상당 부분 필요없다.

그 기능들은 제 9장에서 상세히 다룰 것이다. 만약 여러분이 새 애플리케이션을 구축한다면, 내장되어 있는 인증 체계를 사용하라. 그러면 관련 구현을 여러분이 직접 하지 않아도 된다. 내장되어 있는 체계는 더 안전하고, 더 사용하기 쉽다. 또한 커스텀 구현들에서 놓치기 쉬운 특이 케이스(edge case)들을 처리해 준다.

레거시(legacy, 오래되었지만 여전히 사용되고 있는) 코드에서는 **@LoginRequired** 를 사용하고 있을 것이다. 그런 경우, 기회가 있으면 이 새 인증 체계로 마이그레이션할 것을 권장한다. 마이그레이션 절차는 간결하다. 그리고 그 결과로 얻게 되는 이점들은 상당히 크다. 코드가 줄어들고, 보안과 세분화(granularity) 측면에서 향상되기 때문이다.

맺음말

WebStencils의 고급 기능들은 여러분이 복잡한 시나리오들을 처리할 때 필요한 추가적인 수단들을 제공한다. **Scaffolding** 키워드는 여러분이 코드를 사용해 ‘반복되는 HTML’을 생성할 수 있도록 한다. **OnValue** 를 사용하면 계산된 값들을 제공하거나 프로퍼티 접근을 가로챌 수 있다. **@()** 문장 구조를 사용하면 기본적인 로직을 템플릿들 자체 안에서 직접 적어 넣을 수 있다.

하지만, 가장 좋은 코드는 종종 가장 단순한 코드이다. 이 기능들을 사용하는 경우는 정말로 이것들이 여러분의 애플리케이션을 더 쉽게 유지보수할 수 있도록 할 때여야 한다. 즉, 이것들을 모든 곳에 사용해야 한다는 의무감을 갖지 않기를 바란다. 일반적인 `@if` 문들과 `@ForEach` 루프들을 가진 직관적인 템플릿이 때로는 정답이다.

09

세션(Session) 관리 및 인증(Authentication)

머릿말

세션(session) 관리는 역사적으로 WebBroker(웹브로커)로 웹 애플리케이션들을 구축할 때 가장 까다로운 부분들 중 하나였다. 여러 요청(request)들 전반에 걸쳐 여러분은 수작업으로 사용자들을 추적하고, 세션 데이터를 어딘가에 저장하고, 세션 만료를 처리하고, 인증 로직(logic)을 구현하고, 권한 부여를 관리해야 했다. 그것은 지금도 가능하다. 하지만 여러분이 많은 인프라(infrastructure) 코드를 작성하고 유지보수해야 한다는 뜻이다. 실제 애플리케이션에 집중하기 전에 그런 일부터 해야 했다.

RAD Studio(라드 스튜디오) 13.0은 광범위한 세션 관리 및 인증 체계를 도입했다. 그 체계는 여러분을 위해 그 모든 것을 다뤄준다. 여러분은 그저 WebModule(웹모듈)에 컴포넌트(component) 세 개를 올려놓고, 이벤트 하나를 다루면 된다. 그러면 역할 기반 접근 제어(role-based access control)를 갖춘, 완전하고 상용 수준인(production-ready) 인증 체계를 가지게 된다.

이것은 ‘최소화된 개념 증명(PoC, proof-of-concept)’ 수준이 아니다. 내장되어 있는 이 체계에는 여러분이 성숙한 인증 프레임워크에게 기대하는 모든 것들을 들어있다: 자동 세션 생애주기 관리, 구성-가능한 저장소 옵션들, 역할-기반 권한 부여, 미인증 사용자들에 대한 자동 리다이렉트(redirect), 암호화 서명이 된 세션 ID들 등을 갖추고 있다.

이 장에서, 우리는 이 새로운 컴포넌트들을 사용하는 방법을 탐색할 것이다 그래서, 코드 작성을 최소화하는 방식으로 여러분의 WebBroker 애플리케이션들 안에 인증(authentication)을 추가한다.

세 가지 컴포넌트

세션 관리 체계는 컴포넌트 세 개로 구성된다. 그것들이 함께 작동하면서 인증(authentication)과 권한 부여(authorization)를 처리한다:

TWebSessionManager

이것은 세션들 생애주기를 관리하는 기반 컴포넌트(foundation component)이다. 이것이 맡는 일은 세션 생성, 저장, 만료, 정리(clean up)이다. 요청이 들어오는 경우, 세션 매니저는 기존 세션을 읽어 오거나 새로운 세션을 생성한다.

세션 매니저는 구성 가능 수준이 매우 높다. 여러분은 이런 것들을 선택할 수 있다. 세션 ID가 어디에 저장되는지(쿠키들, 헤더들, 쿼리 파라미터들), 세션 범위가 어떻게 되는지(요청별, 사용자별, 사용자+IP별), 세션 만료 기간 등을 지정할 수 있다.

TWebFormsAuthenticator

이 컴포넌트는 ‘HTML 폼 기반’ 인증을 처리한다. 그리고 인증 흐름 전체를 관리한다: 미인증 사용자들을 로그인 페이지로 리다이렉트하기, 로그인 폼 제출들을 처리하기, 자격 증명(credentials)들을 여러분의 코드를 통해 검증하기, 로그인 시도의 성공 또는 실패 후에 사용자들을 리다이렉트하기 등을 담당한다.

이 Authenticator는 ‘보호되는 페이지’들에 대한 요청들을 가로채는 방식으로 작동한다. 만약 인증된 사용자가 아니면, 자동으로 그들을 로그인 URL로 보낸다. 사용자가 자격 증명을 제출하면, Authenticator는 이벤트를 발생시킨다. 그 이벤트를 구현해 여러분이 사용자 이름과 비밀번호를 검증하면 된다. 여러분의 대응에 따라, 사용자는 로그인되어 의도한 목적지로 리다이렉트되거나 실패 페이지로 전송된다.

TWebAuthorizer

Authenticator는 “여러분이 누구인가”를 처리한다. 반면, Authorizer는 “여러분이 무엇을 할 수 있는가”를 처리한다. 이 컴포넌트는 역할-기반 접근 제어를 제공한다. 그 방법으로, 권한 부여 구역(authorization zone)들을 사용한다. 여러분은 ‘사이트의 보호된 영역’들을 정의할 수 있다. 그리고 어떤 역할(roles)들이 있어야 그것들에 접근할 수 있는지를 지정할 수 있다.

예를 들어, 여러분은 `/admin` 구역을 두고 "admin" 역할을 가진 사용자들만이 접근하도록 할 수 있다. 한편, `/user` 구역들은 "user" 역할을 가진 누구나 이용하도록 할 수 있다. Authenticator는 이 규칙들을 자동으로 강제한다.

인증 설정하기

완전한 인증 체계를 설정하는 과정을 진행해보자. 좋은 소식이 있다. 필요한 코드의 양은 매우 적다.

컴포넌트 구성(Configuration)

먼저, 여러분의 WebModule 위에 다음 세 개의 컴포넌트들을 배치하라:

1. TWebSessionManager
2. TWebFormsAuthenticator
3. TWebAuthorizer

이제 그 authenticator의 프로퍼티들을 구성(configure)하라:

```
// Object Inspector 안에서 또는 코드 안에서:

// 미인증 사용자들을 어디로 보낼 것인가를 명시한다
WebFormsAuthenticator.LoginURL := '/login';

// 로그인 성공 후, 어디로 리다이렉트 할 것인가를 명시한다
WebFormsAuthenticator.HomeURL := '/';

// 로그인 실패 후, 어디로 리다이렉트 할 것인가를 명시한다
WebFormsAuthenticator.FailedURL := '/login?error=1';
```

이것이 기본적인 설정이다. 컴포넌트들은 WebModule(웹모듈)을 통해 자동으로 서로 연결된다.

자격 증명 검증 (Credential Validation)구현하기

여러분이 반드시 작성해야 하는 유일한 코드는 자격 증명(credential) 검증 코드 뿐이다. Authenticator의 OnAuthenticate 이벤트에 작성하면 된다:

```
procedure TWebModule1.WebFormsAuthenticatorAuthenticate(
  Sender: TCustomWebAuthenticator;
  Request: TWebRequest;
  const UserName, Password: string; var Roles: string; var Success: Boolean);
begin
  // 자격 증명(credential)을 검증한다: 여러분의 사용자 데이터베이스와 대조하기 등
  Success := False;
  Roles := '';

  // 예시: 사용자 데이터베이스와 대조해 확인(check)한다
  if ValidateUserCredentials(UserName, Password) then
  begin
    Success := True;
    Roles := GetUserRoles(UserName); // 예: 'user,admin'
  end;
end;
```

Roles 파라미터는 역할 이름들이 쉼표로 구분되어 나열되는 문자열이다. 예를 들어, 어느 사용자는 'user,moderator' 역할들을 가질 수 있다. 반면, 관리자는 'user,admin' 역할들을 가질 수 있다.



warning

절대 비밀번호를 일반 텍스트로 저장하지 마라! 적절한 비밀번호 해싱(hash)을 사용하라. 위의 예제는 `ValidateUserCredentials`가 내부적으로 해싱 비교를 처리한다고 가정한다. 시연 목적이거나 하드코딩된 자격 증명(credentials)들을 사용할 수 있겠지만, 실제 운영 애플리케이션에서는 항상 적절한 사용자 데이터베이스를 대조하고 해싱된 비밀번호들을 사용하는 방식을 통해 유효성을 검사해야 한다.

로그인 폼(Login Form) 만들기

단순한 로그인 폼 템플릿을 만든다 (login.html):

```
@LayoutPage layouts/baseLayout

<div class="login-container">
  <h1>로그인</h1>

  @if query.error {
    <div class="alert alert-danger">
      유저네임 또는 비밀번호가 유효하지 않습니다. 다시 시도하세요.
    </div>
  }

  <form method="post" action="/login">
    <div class="form-group">
      <label for="username">유저네임</label>
      <input type="text" id="username" name="username"
        class="form-control" required autofocus>
    </div>

    <div class="form-group">
      <label for="password">비밀번호</label>
      <input type="password" id="password" name="password"
        class="form-control" required>
    </div>

    <button type="submit" class="btn btn-primary">로그인</button>
  </form>
</div>
```

위 폼은 `/login`에게 포스트(post) 된다. 그러면 Authenticator가 자동으로 처리한다. 여러분은 이 login 엔드포인트를 위해 어떠한 액션 핸들러 코드도 직접 작성할 필요가 없다.

로그아웃 핸들러(Logout Handler) 만들기

로그아웃은 자동으로 Authenticator가 처리한다. 그저 설정해 놓은 **LogoutURL**로 POST를 보내면 된다:

```
// Object Inspector 안에서 또는 코드 안에서:  
WebFormsAuthenticator.LogoutURL := '/logout';
```

이게 전부다. 사용자들이 **/logout**으로 POST를 보내는 경우, 그들은 자동으로 로그아웃된다. 그리고 나서 **HomeURL**로 리다이렉트된다. 이 또한 어떠한 액션 핸들러 코드도 여러분이 작성할 필요가 없다.

여러분의 로그아웃 폼은 직관적이다. 이렇게 작성하면 된다:

```
<form method="post" action="/logout">  
  <button type="submit" class="btn btn-secondary">로그아웃</button>  
</form>
```

Authenticator가 로그아웃 흐름 전체를 처리한다: 세션 끝내기, 인증 상태 비우기, 사용자를 홈 페이지로 리다이렉트하기 등을 담당한다.

@session 오브젝트

인증 설정이 완료되면, 모든 템플릿은 자동으로 **@session** 오브젝트에 대해 접근할 수 있다. 이 세션 오브젝트는 '현재 세션' 그리고 '사용자'에 대한 정보를 제공한다.

세션(Session)의 프로퍼티들

@session 오브젝트는 몇 가지 유용한 프로퍼티들을 노출한다:

인증(Authentication) 상태:

- **@session.Authenticated** - 사용자의 로그인 여부를 나타내는 불리언(Boolean) 값
- **@session.UserName** - 인증된 사용자 이름
- **@session.UserRoles** - 심표로 구분된 사용자 역할(role)들의 문자열

세션(Session) 정보:

- **@session.SessionID** - 고유한 세션 식별자
- **@session.Timeout** - 세션 타임아웃(분 단위)
- **@session.CreatedTime** - 세션이 생성된 시점
- **@session.AccessedTime** - 마지막 접근 타임스탬프(timestamp)

- `@session.AccessedCount` - 이 세션 안에서 발생한 요청(request) 횟수

요청(Request) 정보:

- `@session.LastURL` - 마지막으로 요청된 URL
- `@session.LastIP` - 마지막 요청의 클라이언트(client) IP 주소
- `@session.LastReferer` - 리퍼러(referrer) 페이지
- `@session.LastUserAgent` - 브라우저의 유저 에이전트(user agent) 문자열

@session을 템플릿 안에서 사용하기

여러분이 템플릿 안에서 세션 오브젝트를 사용하는 전형적인 방법은 이렇다:

```
<!-- 네비게이션바는 인증(authentication)을 기반으로 서로 다른 내용을 보여준다 -->
<nav class="navbar">
  <a href="/" class="nav-brand">나의 앱</a>

  <div class="nav-menu">
    <a href="/home" class="nav-link">홈</a>
    <a href="/about" class="nav-link">About</a>

    @if session.Authenticated {
      <a href="/dashboard" class="nav-link">대시보드</a>

      @if session.UserHasRole('admin') {
        <a href="/admin" class="nav-link">어드민</a>
      }

      @if session.UserHasRole('moderator') {
        <a href="/moderate" class="nav-link">중재자</a>
      }

      <a href="/logout" class="nav-link">로그아웃 (@session.UserName)</a>
    } @else {
      <a href="/login" class="nav-link">로드인</a>
      <a href="/register" class="nav-link">가입하기</a>
    }
  </div>
</nav>
```

`UserHasRole()` 메서드는 특히 유용하다. 이것은 사용자가 특정 역할을 가지고 있는지를 확인한다. 즉, ‘심표로 구분된’ 역할 문자열을 내부적으로 처리해 해당 역할을 가지고 있는지 확인한다.

권한 부여 구역(Authorization Zone)들

권한 부여 구역(zone)들은 사이트의 전체 섹션(section)들을 사용자 역할들에 기반하여 보호하도록 해준다. 이 방식은 역할 확인을 모든 템플릿마다 그 안에서 직접 하는 것보다 더 편하다.

보호된 영역(protected area)들 구성하기

여러분은 권한 부여 구역(authorization zone)들을 구성할 수 있다. 코드 안에서 구성해도 되고, TWebAuthorizer 컴포넌트의 **Zones** 프로퍼티를 통해 구성해도 된다. 코드 안에서는 이렇게 하면 된다:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
var
  Zone: TWebAuthorizationZone;
begin
  // 전체를 보호한다 /admin/* URL 주소들 - 'admin' 역할 필요
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/admin*';
  Zone.Kind := zkProtected;
  Zone.Roles := 'admin';

  // 공개 접근을 허용한다 도움말 문서집 - 인증(authentication) 필요 없음
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/docs*';
  Zone.Kind := zkFree;

  // 건강도 점검 엔드포인트 - 인증과 세션 관리를 모두 우회
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/health*';
  Zone.Kind := zkIgnore;

  // 보호되는 사용자 대시보드 - 인증된 사용자 누구나
  Zone := WebAuthorizer.Zones.Add;
  Zone.PathInfo := '/dashboard*';
  Zone.Kind := zkProtected;
  Zone.Roles := 'user,admin'; // 둘 중 어느 역할(role)이든 효과 있음
end;
```

별표(*)는 와일드카드이다. 그 경로 뒤에 무엇이 나오든 모두 해당된다. 따라서 **/admin*** 이라고 되어 있으면, **/admin**, **/admin/users**, **/admin/settings**, 등등이 모두 해당된다.

구역 타입(Zone Type)들

세 가지 구역 타입이 있다:

- **zkProtected** - 인증 그리고 지정된 역할들이 필요하다 (기본값)

- **zkFree** - 인증 없이 접근을 허용한다. 하지만, 세션(session)은 여전히 관리된다
- **zkIgnore** - 인증과 세션 관리 모두를 완전히 우회한다



tip

zkIgnore를 건강도 확인(health check) 엔드포인트들에 사용하라. 그 엔드포인트들은 모니터링 시스템들이 빈번하게 호출한다. 또한 이처럼 세션 관리가 전혀 필요 없는 다른 엔드포인트들을 위해서도 사용하면 좋다. 매 요청마다 세션이 생성되는 것을 막아주기 때문이다. 그러면 메모리 사용량과 데이터베이스 부하(여러분이 지속성 세션 저장소를 사용하고 있을 경우)가 모두 줄어든다.

미인증 사용자가 **zkProtected** 구역에 접근을 시도하면, 그들은 자동으로 로그인 페이지로 보내진다. 성공적으로 로그인한 후에는, 그들이 원래 요청했던 페이지로 다시 리다이렉트된다.

만약 사용자가 로그인 상태이지만, 자신의 역할에 맞지 않는 보호된 영역에 접근하려고 시도하면, 그들은 **UnauthorizedURL** 프로퍼티에 구성해 놓은 엔드포인트로 리다이렉트될 것이다.

다중 역할(Multiple Role)들

여러분은 여러 역할들을 지정할 수 있다. 심표로 구분하면 된다. 사용자는 나열된 역할들 중 하나만 있어도 접근권을 가질 수 있다:

```
Zone.Roles := 'admin,moderator,support'; // 이 역할들 중 어느 것에든 접근 권한을 부여한다
```

만약 사용자가 역할들을 ‘모두’(union이 아닌 intersection) 보유해야 접근할 수 있는 구역을 구성하려면, 해당 검증을 여러분의 코드나 템플릿들 안에서 직접 구현해야 한다. 이 내장 시스템은 **OR** 로직을 사용하기 때문이다.

세션 구성(Session Configuration) 옵션들

TWebSessionManager 컴포넌트는 세션 동작을 커스터마이징 할 수 있도록 여러 구성 옵션들을 제공한다.

세션 ID 저장소

여러분은 세션 ID들이 저장되는 위치를 제어할 수 있다. **IdLocation** 프로퍼티를 사용하면 된다:

```
// 쿠키(cookie) 안에 저장 (기본 설정, 권장 방식)
WebSessionManager.IdLocation := ilCookie;

// HTTP 헤더 안에 저장 (API들용으로 유용함)
```

```
WebSessionManager.IdLocation := ilHeader;

// 쿼리(query) 파라미터들 안에 저장 (보안성이 가장 낮음, 꼭 필요할 때만 사용할 것)
WebSessionManager.IdLocation := ilQuery;
```

쿠키(cookie) 저장소는 기본 설정이다. 그리고 대부분의 웹 애플리케이션들에 잘 작동한다. 헤더(header) 저장소는 API 엔드포인트들에 유용하다. 그것들에서는 쿠키 사용이 실용적이지 않기 때문이다. 쿼리 파라미터(query parameter) 저장소는 실제 운영 환경이라면 피해야 한다. 보안 우려 때문이다 (그 이유: 세션 ID가 URL에 그대로 노출되고, 그 URL이 서버 로그에 기록된다. 기타 여러 취약점이 있다).

세션 범위(Session Scope)

세션 범위는 세션들이 사용자들과 연결되는 방식을 결정한다. **Scope** 프로퍼티를 사용하면 된다:

```
// 매 요청(request)마다 새 세션을 생성한다(특정한 사용자에게 대한 바인딩 없음)
WebSessionManager.Scope := ssUnlimited;

// 세션이 인증된 사용자에게 바인딩 된다.(사용자명과 역할들에 의해 식별됨)
WebSessionManager.Scope := ssUser;

// 세션이 사용자와 IP 주소 모두에 바인딩 된다 (더 안전하다. 하지만 IP 주소가 변경될 경우 끊어진다.)
WebSessionManager.Scope := ssUserAndIP;
```

기본 설정은 **ssUnlimited** 이다. 이 설정은 세션 ID가 없는 모든 요청(request)들마다 새 세션을 생성한다. 이는 익명 사용자들과 인증된 사용자들 모두에게 작동한다.

ssUser 를 사용하면, 세션 ID가 일정한 규칙에 따라 만들어 진다. 그것은 사용자명과 역할들을 바탕으로 **SharedSecret**을 키로 하는 HMAC-SHA2를 통해 만들어 진다. 따라서, 동일한 사용자가 얻게 되는 세션 ID는 항상 같다. 요컨데, 이 세션 범위는 세션 고정 공격(session fixation attacks)을 방지한다. 그리고, 동일한 사용자가 다른 디바이스들 또는 브라우저들을 사용해도 일관된 사용자 경험을 보장한다.

ssUserAndIP 는 HMAC 계산에 클라이언트의 IP 주소를 더 추가한다. 그래서 추가적인 보안을 제공한다. 즉, 세션을 ‘사용자’ 그리고 ‘그 사용자가 사용하는 IP 주소’ 모두에 바인딩한다. 하지만, 이 옵션은 모바일 네트워크 사용자 또는 로드 밸런서(load balancer) 뒤에 있는 사용자들에게는 문제가 될 수 있다. 그런 환경은 세션 도중에 IP 주소들이 변경될 수 있기 때문이다.

세션 타임아웃

세션이 얼마나 오래 지속되는지 구성한다. **Timeout** 프로퍼티(초 단위)를 사용하면 된다:


```
// 타임아웃을 30 분으로 지정한다 (1,800 초)
WebSessionManager.Timeout := 1800;

// 타임아웃을 2 시간으로 지정한다 (7,200 초)
WebSessionManager.Timeout := 7200;
```

기본 타임아웃은 3600초(1시간)이다. 타임아웃은 각 요청(request)마다 자동으로 재설정된다. 따라서 활성 사용자가 그 사이트를 사용하고 있는 한 그 사용자는 로그아웃되지 않을 것이다. `AccessedTime` 프로퍼티는 요청이 시작될 때와 완료될 때 모두 항상 업데이트된다. 따라서 모든 사용자 상호작용마다 타임아웃 카운트다운이 재시작되도록 보장한다.

공유 비밀 키(Shared Secret)

사용자-범위 세션들(`ssUser` 또는 `ssUserAndIP`)인 경우, 여러분은 반드시 공유 비밀 키(shared secret)를 구성해야 한다:

```
WebSessionManager.SharedSecret := 'your-secret-key-here';
```

공유 비밀 키(shared secret)는 ‘세션 ID들을 만들어 낼 때 적용하는’ HMAC-SHA2 해싱을 위한 키(key)로 사용된다.



warning

실제 운영 환경에서는, 길고 무작위인 문자열(최소 32자 이상)을 공유 비밀 키(shared secret)로 사용하라. 이 비밀 키를 구성 파일이나 환경 변수에 안에 안전하게 저장하라. 결코 버전 관리 시스템(version control)에 커밋(commit)하지 마라.

맺음말

RAD Studio(라드 스튜디오)에 내장된 세션 관리 및 인증 체계은 전통적으로 웹 인증을 위해 요구되었던 대부분의 보일러플레이트(boilerplate) 코드를 없앴다. 세 개의 컴포넌트와 단 하나의 이벤트 핸들러만으로 여러분은 상용 수준인(production-ready) 인증 체계를 얻게 된다. 그 체계는 암호화 서명된 세션들, 폼 기반 인증, 역할-기반 권한 부여 등을 이미 갖추고 있다.

이 체계는 충분히 유연하다. 그래서 일반적인 시나리오 대부분을 다룰 수 있다: 서로 다른 세션 범위(scope)들, 다양한 저장 위치들, 세분화된 권한 부여 구역(authorization zone)들 등등. `@session` 오브젝트를 사용하면, 여러분의 템플릿들에서 인증 상태에 쉽게 접근할 수 있다. 그래서 사용자 권한에 맞게 조정되는 UI(사용자 인터페이스) 구축을 쉽게 할 수 있다.

10

데이터베이스-주도로 UI를 만들어 내기

머릿말

폼(form)들은 웹 개발에서 가장 지루한 작업 부분들 중 하나이다. 데이터베이스 스키마(schema)가 변경될 때마다 여러분은 테이블을 업데이트(update)하고, IDE에서 데이터셋(dataset)의 필드(field)들을 다시 만들고, 그 필드들을 참조하는 모든 HTML 폼들을 수동으로 업데이트해야 한다. 새 컬럼(column)을 만들면? 다섯 개의 서로 다른 폼들을 업데이트해야 한다. 필드를 선택 사항에서 필수 사항으로 바꾸면? 여러분의 템플릿(template)들 안을 모두 뒤져서 해당되는 모든 인스턴스(instance)들을 찾아내야 한다.

RAD Studio(라드 스튜디오) 13.0은 ‘데이터베이스-기반 UI 생성’을 도입했다. 이것은 이런 모델을 완전히 바꿨다. 여러분의 데이터베이스에 맞추기 위해 수작업으로 폼을 구축하는 대신, 여러분이 구축하는 폼들이 자동으로 여러분의 데이터베이스에 맞춘다. 그리고 여러분의 FireDAC(파이어닥)의 쿼리(query)들은 ‘단 하나의 공식 기준점’이 된다. 그리고 여러분의 템플릿들은 데이터 필드의 메타데이터(metadata)들을 직접 읽는다.



warning

필드 메타데이터들을 템플릿들에서 접근하려면, 반드시 미리 화이트리스트(whitelist)를 구성해야 한다. TField 프로퍼티들은 노출되지 않는 것이 기본 설정이다. 보안 상의 이유 때문이다. 나중에 자세히 다루겠다. 지금은 이 점만 명심하자: ‘데이터베이스-기반’ 폼들을 만들려면, WebModule(웹모듈)의 초기화 안에서 필요한 화이트리스트(whitelist)를 명시적으로 미리 구성해 놓아야 한다.

작동 방식

WebStencils(웹스텐실즈)는 FireDAC 프로퍼티들에 접근할 수 있다. 템플릿 표현식들을 사용하면 된다. 예를 들어, 데이터셋의 `fields` 컬렉션을 참조하면, 모든 필드 정의에 접근할 수 있다. 그래서 여러분이 FireDAC 쿼리 안에 구성해 놓은 모든 메타데이터들에도 접근할 수 있다.

여기 간단한 예시가 있다:

```
@foreach(var f in customers.fields) {  
    <div>  
        <label>@f.DisplayLabel</label>  
        <input type="text" name="@f.FieldName" value="@f.Value">  
    </div>  
}
```

이 루프(loop)는 `customers` 데이터셋 안에 있는 모든 필드들을 반복하며, 각 필드를 읽는다:

- `DisplayLabel` - 사람이 읽기 편한 레이블 (여러분이 FireDAC 안에 지정해 놓는 것)
- `FieldName` - 실제 데이터베이스 컬럼(column) 이름
- `Value` - 해당 필드의 현재 값

보안 : 화이트리스트(Whitelist) 체계

데이터베이스-기반 UI는 강력하다. 하지만, 그와 동시에 보안 우려도 유발한다. ‘연결 문자열(connection string)’, 내부 데이터셋 상태 등 민감한 프로퍼티들을 여러분의 템플릿들이 실수로 노출하는 것을 원하는 개발자는 없다.

WebStencils는 화이트리스트 체계를 가지고 있다. 그래서 템플릿들을 통해 접근할 수 있는 프로퍼티들이 무엇인지를 통제한다. 기본 설정인 경우, 오직 최소한의 프로퍼티 세트들만이 화이트리스트 안에 포함된다:

TDataSet ('화이트리스트에 포함됨'이 기본 설정임):

- Active, FieldByName, First, Last, Next, Prior
- Bof, Eof, FieldCount, Fields, Found, RecordCount, RecNo

TStrings ('화이트리스트에 포함됨'이 기본 설정임):

- Contains, ContainsName, IndexOf, IndexOfName
- CommaText, Count, IsEmpty, DelimitedText
- Names, KeyNames, Values, ValueFromIndex, Strings, Text

TField ('화이트리스트에 포함되지 않음'이 기본 설정임):

이것은 의도적인 것이다. 여러분의 애플리케이션에서 필요한 TField 프로퍼티가 있다면, 여러분이 직접 명시적으로 화이트리스트에 넣어주어야만 한다. TField를 화이트리스트에 추가하지 않은 경우, 여러분의 템플릿들 안에서 `DisplayLabel`, `FieldName`, `Value` 와 같은 프로퍼티에 접근할 수 없다. 그리고 이러한 경우에는 데이터베이스-기반 폼들이 아예 작동하지 않는다.

여러분은 WebModule(웹모듈) 초기화 시점에 TField 화이트리스트를 구성해야만 한다:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
begin
    // TField 화이트리스트를 구성한다 - '데이터베이스-주소 폼들'을 위해 "필수"임
    // 이것이 없으면, 템플릿들은 필드의 프로퍼티들에 접근하지 못한다
    TWebStencilsProcessor.Whitelist.Configure(
        TField,
        ['DisplayText', 'Value', 'DisplayLabel', 'FieldName',
         'Required', 'LookupDataSet', 'LookupKeyFields',
         'Visible', 'DataType', 'Size', 'IsNull'],
        nil, // 어느 프로퍼티도 가로막지 않는다
        False // 부모 클래스의 제한 사항(restriction)들을 상속받지 않는다
    );
end;
```

이 구성이 없는 경우, `@field.DisplayLabel` 또는 이와 유사한 프로퍼티들을 템플릿들 안에서 접근하려고 하면 소리 없이 무시된다. 이는 '보안 우선 설계(security-first design)'에 따른 것이다. 즉, 무엇을 노출할지를 여러분이 명시적으로 선택해야 하는 방식이다.



warning

화이트리스트(whitelist)에 무엇을 넣을지 결정할 때는 보수적으로 하라. 확실하지 않은 경우라면, 화이트리스트에 추가하지 마라. 나중에 필요할 때 언제든지 추가할 수 있다.

동적 폼(Dynamic Form)들을 구축하기

가장 일반적인 사용 사례(use case)는 여러분의 데이터셋(dataset) 구조에 자동으로 적응하는 폼을 만들어 내는 것이다. 여기에 기본적인 패턴이 있다:

```
<form method="post">
  @forEach(var field in customers.fields) {
    @if(field.Visible) {
      <div class="form-group">
        <label for="@field.FieldName">
          @field.DisplayLabel
          @if(field.Required) {
```

```

        <span class="text-danger">*</span>
    }
</label>

    <input
        type="text"
        name="@field.FieldName"
        value="@field.Value"
        @if(field.Required) { required }
        @if(field.Size > 0) { maxlength="@field.Size" }>
</div>
}
}

<button type="submit">저장하기</button>
</form>

```

이 템플릿은 완전한 폼(form)을 만들어 낸다. 그 폼 안은 이런 것들이 반영된다:

- 레이블 (DisplayLabel에서 가져온다)
- 필수 필드임을 나타내는 표시
- HTML5의 required 애트리뷰트
- 최대 길이 유효성 검사
- 현재 값 채우기

하지만 약점이 있다. 위 템플릿은 모든 필드를 텍스트 입력창(text input)으로 만든다. 타입을 가리지 않고, 숫자, 날짜, 불리언(boolean), 메모(memo) 필드들을 모두 그렇게 한다. 이제 이 문제점을 해결해 보자.

@switch 연산자와 필드 타입(Field Type)들

@switch 연산자(RAD Studio 13.0에서 도입됨)은 서로 다른 필드 타입들을 처리하기에 완벽하다. 여러분은 `field.DataType` 을 검사하고 그에 적합한 HTML 입력(input)을 만들어 낼 수 있다:

```

@foreach(var field in customers.fields) {
    @if(field.Visible) {
        <div class="form-group">
            <label>@field.DisplayLabel</label>

            @switch(field.DataType) {
                @case "ftString" {
                    <input type="text"
                        name="@field.FieldName"
                        value="@field.Value"

```

```

        @if(field.Size > 0) { maxlength="@field.Size" }>
    }
    @case "ftInteger" {
        <input type="number"
            name="@field.FieldName"
            value="@field.Value"
            step="1">
    }
    @case "ftFloat" {
        <input type="number"
            name="@field.FieldName"
            value="@field.Value"
            step="0.01">
    }
    @case "ftDate" {
        <input type="date"
            name="@field.FieldName"
            value="@field.DisplayText">
    }
    @case "ftDateTime" {
        <input type="datetime-local"
            name="@field.FieldName"
            value="@FormatDateTime('yyyy-mm-dd'T'hh:nn', field.Value))">
    }
    @case "ftBoolean" {
        <input type="hidden" name="@field.FieldName" value="False">
        <input type="checkbox"
            name="@field.FieldName"
            value="True"
            @if(field.Value) { checked }>
    }
    @case "ftMemo" {
        <textarea name="@field.FieldName" rows="3">@field.Value</textarea>
    }
    @default {
        <input type="text" name="@field.FieldName" value="@field.Value">
    }
}
</div>
}
}

```

이 방식은 데이터베이스 필드의 타입에 기반하여 알맞은 HTML5 입력(input) 타입들을 만들어 낸다. 숫자들은 `<input type="number">`를, 날짜들은 `<input type="date">`를 얻는 식이다.



숨겨진 입력창(hidden input)을 체크박스 앞에 두는 것은 표준적인 HTML 패턴이다. 만약 폼(form)이 제출될 때 선택되지 않은 체크박스가 빠지는 경우 (선택되지 않은 체크박스들은 브라우저가 빠고 제출한다), 이 숨겨진 필드가 있으면, 여러분의 서버가 "False"를 받도록 보장한다. 그래서 여러분이 아무런 값도 받지 못하는 경우가 없다. 이는 서버 쪽 처리를 더 단순하게 한다. 또한 폴백(fallback)을 제공해 '정의되지 않은 반환 값(undefined return value)들'이 생기지 않도록 한다

재사용 가능한 필드 컴포넌트(Component)들

동적 폼들은 잘 작동한다. 하지만, 그 내용이 장황해질 수 있다. 일관된 스타일 반영, 유효성 검사 표시, 특별한 처리 등이 더 추가되어야 하는 경우가 많기 때문이다. 해결책은 '필드 렌더링' 부분만 따로 뽑아내는 것이다. 그래서 재사용 가능한 '부분 요소/컴포넌트'들로 만들어 놓는다.

dynamicInput.html 템플릿을 만든다:

```
@if(field.Visible) {
<div class="form-group">
  <label for="@field.FieldName">
    @field.DisplayLabel
    @if(field.Required) { <span class="text-danger">*</span> }
  </label>

  @switch(field.DataType) {
    @case "ftInteger" {
      <input type="number"
        class="form-control"
        name="@field.FieldName"
        value="@field.Value"
        @if(field.Required) { required }>
    }
    @case "ftDate" {
      <input type="date"
        class="form-control"
        name="@field.FieldName"
        value="@field.DisplayText"
        @if(field.Required) { required }>
    }
    @case "ftBoolean" {
      <div class="form-check">
        <input type="hidden" name="@field.FieldName" value="False">
        <input type="checkbox"
          class="form-check-input"
          name="@field.FieldName"
          value="True"
          @if(field.Value) { checked }>
      </div>
    }
  }
}
```

```

        <label class="form-check-label">@field.DisplayLabel</label>
    </div>
}
@default {
    <input type="text"
        class="form-control"
        name="@field.FieldName"
        value="@field.Value"
        @if(field.Required) { required }
        @if(field.Size > 0) { maxLength="@field.Size" }>
}
}
</div>
} @else {
    <input type="hidden" name="@field.FieldName" value="@field.Value">
}
}

```

그리고 나서, 그것을 여러분의 메인 폼 안에서 사용하라:

```

<form method="post">
    @forEach(var f in customers.fields) {
        @import partials/dynamicInput { @field = @f }
    }
    <button type="submit">저장하기</button>
</form>

```

이 패턴은 일관성을 모든 폼들에 걸쳐 제공한다. 그러면서도, ‘필드 특정’ 로직을 한곳에 보관한다. 날씨가 렌더링되는 방식을 바꿔야 한다면? 오직 그 파일 하나만 업데이트하면 된다.

하이브리드 접근 방식 (종종 최선의 선택이 된다)

모든 폼들을 자동으로 만들어 낼 수 있겠다는 생각을 할 수 있을 것이다. 하지만, 실제 애플리케이션들에서는 이보다 훨씬 더 복잡한 폼들도 많다. 그래서 특정 서식 적용(formatting), 직접 배치/구성하기, 등등이 필요할 수 있다.

그렇다고 해서 지금 본 모든 이점들을 누리지 못한다는 뜻이 아니다. 쓸만한 요령 하나를 제시하겠다. 필드(field)들 안에 있는 **Tag** 프로퍼티를 사용하는 것이다. 그 필드에 표시를 넣어서 자동으로 처리할 필드인지 수동으로 처리할 필드인지를 정의한다. 예를 들어, 수동으로 처리할 필드들에는 **Tag** 값을 1로 정의한다. 더 나아가, 필드들을 그룹화 할 수도 있다. 즉, 똑같은 **Tag** 값을 가진 필드들만 따로 모아서 특정 구역 안에 렌더링하도록 할 수 있다.

예시를 보자:

```
<form method="post">
  <!-- 사용자 정의 헤더(header) 구역-->
  <div class="form-header">
    <h2>고객 정보</h2>
    <p>정확한 연락 정보를 제공하세요.</p>
  </div>

  <!-- 동적으로 생성: 태그가 붙어 있는 필드(field)들을 다룬다 -->
  <div class="group-1">
    @forEach(var f in customers.fields) {
      @if(f.Visible and (f.Tag = 1)) {
        @import partials/dynamicInput { @field = @f }
      }
    }
  </div>
  <div class="group-2">
    @forEach(var f in customers.fields) {
      @if(f.Visible and (f.Tag = 2)) {
        @import partials/dynamicInput { @field = @f }
      }
    }
  </div>

  <!-- 사용자 정의 구역: 특별한 필드들을 다룬다 -->
</form>
```

이 방식은 동적 폼들이 가지는 유지보수성과 사용자 정의 배치(layout)들이 가지는 유연성을 결합한다. 즉, 두 방식의 장점을 모두 가질 수 있다: FireDAC(파이어닥)은 필드 정의들을 제어한다. 여러분은 사용자 경험(UX)을 제어한다.

핵심 통찰은 이것이다: **필드의 메타데이터는 유용하다. 폼들을 자동으로 생성하든 안하든 상관없이 그렇다.** 심지어 완전히 커스텀화된 배치 또는 읽기-전용 배치들에서도 `DisplayLabel`, `Required`, `DataType`, `Value` 등을 FireDAC으로부터 읽어서 사용하는 방식을 쓴다면, 여러분은 필드 정의들을 위한 ‘단 하나의 권위 있는 원본’을 가지게 된다. 그러면 각 템플릿들마다 그런 정보를 중복 작성하기를 반복할 필요가 없다.

맺음말

‘데이터베이스-기반 UI 생성’은 여러분의 데이터베이스와 여러분의 웹 애플리케이션 사이의 관계를 단순화한다. 스키마(schema)와 템플릿들 사이의 동기화를 수작업으로 유지하지 말고 그 대신, FireDAC에서 메타데이터를 한 번만 정의해 놓고, WebStencils가 그것들을 런타임(runtime)에 읽도록 할 수 있다.

핵심 통찰, 즉 필드 메타데이터 접근이 가치있다는 사실,은 이 두 가지 뚜렷하게 구분되는 방식들로부터 나온다:

1. **전체를 동적으로 생성** - 템플릿이 필드들을 반복하도록 한다. 그래서 내용을 자동으로 만들어 내도록 한다. 이것이 잘 맞는 경우는 관리자 인터페이스, 신속한 프로토타이핑이다. 또한 유사한 CRUD 폼들이 많이 있을 때에도 매우 적합하다.
2. **선택적으로 메타데이터를 접근** - 필드 프로퍼티들을 사용자 지정 배치(layout)들 안에서 사용한다. 그래서, 레이블들, 유효성 검사 규칙들, 타입 정보들을 한 곳에 중앙 집중화한다. 심지어 디자인 전체를 직접 정의하는 방식으로 페이지를 만들더라도, `@customers.EMAIL.DisplayLabel`이라는 ‘단 하나의 공식 기준점(source of truth)’을 가질 수 있다는 장점을 누릴 수 있다.

화이트리스트(whitelist) 체계는 여러분이 보안을 유지할 수 있도록 보장한다. 한편, `@switch` 연산자를 사용하면 서로 다른 필드 타입들이 렌더링되는 방식을 여러분이 정밀한 제어할 수 있다. 재사용 가능한 부분요소(partial)들과 함께 사용하면, 유연성과 유지보수성을 모두 확보할 수 있다.

11

최신 CSS 프레임워크를 가지고 작업하기



머릿말

WebStencils(웹스텐실즈)의 가장 실용적인 기능은 ‘이것이 대단한 것을 하지 않는다’는 것이다. 즉, 여러분이 무슨 CSS, 무슨 JavaScript를 쓰든 전혀 개의치 않는다. 학습해야 할 별도의 커스텀 컴포넌트 라이브러리도 없고, 독점적인 스타일 적용 체계도 없고, 인터페이스를 구축하기 위한 "WebStencils만의 방식"도 따로 없다. 그저 HTML 템플릿들일 뿐이다. 그리고 그 안에 동적인 내용을 넣을 수 있다. 그게 전부다.

‘CSS와 JavaScript를 가리지 않는다’는 특징은 여러분은 어떤 프레임워크든지 원하면 사용할 수 있도록 한다. 혹은 아무것도 사용하지 않아도 상관없다. [Bootstrap\(부트스트랩\)](#)을 원하는가? 그냥 넣으면 된다. [Tailwind\(테일윈드\)](#)를 선호하는가? 잘 작동한다. [Bulma\(불마\)](#)? 당연하다. [BeerCSS\(비어 CSS\)](#)? 하면 된다. 직접 CSS를 작성하기? 당연히 지원된다.

우리는 데모 프로젝트들을 작업할 때 Bootstrap 5를 사용해 왔다. 하지만, 이는 수많은 선택지들 중 하나일 뿐이다. 이 장에서는 WebStencils가 서로 다른 CSS 프레임워크들과 어떻게 함께 작동하는지 살펴본다. 특히 집중하는 대상은 Tailwind CSS이다. 왜냐하면 Tailwind CSS는 빌드 절차가 필요하고 그래서 흥미로운 도전 과제가 될 수 있기 때문이다.

CSS를 가리지 않는다(CSS-Agnostic)는 장점

많은 서버-쪽 프레임워크들은 종종 그것들 자체의 UI 컴포넌트들과 스타일적용 체계들을 묶어서 제공한다. 이는 처음에는 도움이 되는 것처럼 보인다(사전에 구축된 컴포넌트들은 초기 개발 속도를 높여줌). 하지만, 시간이 갈수록 제약이 된다. 여러분은 프레임워크가 제공하는 것들에 갇힌다. 그리고 커스터마이징은 종종 프레임워크들이 정해놓은 가정(assumption)들에 대해 싸운다는 의미가 된다.

WebStencils는 다른 방식을 취한다. 그저 순수한 템플릿 렌더링 엔진(template rendering engine)이다. 이것이 하는 일은 여러분의 HTML을 처리하고, 동적 데이터를 삽입하고, 조건부 루프(loop) 등을 처리한 뒤 그 결과물을 출력하는 것이다. 그 HTML 안에 무엇이 들어가는지는 전적으로 여러분에게 달려 있다.

이 방식은 여러 가지 실용적인 이점들을 제공한다:

- **여러분의 미학적 선택을 여러분이 한다.** 기업용 애플리케이션들은 그에 맞는 룩(look)이 필요하다. 소비자용 앱들은 또 다른 룩이, 내부 도구들은 완전히 다른 무언가가 필요하다.
- **여러분은 해당 생태계와 보조를 맞춘다.** 새 CSS 프레임워크가 출시되거나 기존 프레임워크를 위한 업데이트가 나왔을 때, 그것을 위해 WebStencils가 지원을 추가할 때까지 기다리지 않아도 된다. 그냥 해당 CDN 링크로 (또는 로컬 파일들을) 업데이트하면 된다.
- **여러분은 필요한대로 골라서 섞어 쓴다.** Bootstrap을 사용해 신속하게 프로토타입을 먼저 만들고, 그 후에 점진적으로 구역들을 하나씩 커스텀 CSS로 교체할 수 있다. 또는 유틸리티 클래스들을 위해서는 Tailwind를 사용하면서 다른 한편으로 몇몇 Bootstrap 컴포넌트들도 계속 유지할 수 있다.
- **어딘가에 갇히는 상황을 피한다.** 만약 나중에 프레임워크 변경을 결정하더라도, 이는 CSS의 변경일 뿐이다. 전체를 다시 작성하는 것이 아니다. 여러분의 서버 쪽 로직과 템플릿 구조는 그대로 유지된다.

프레임워크 옵션들(몇 가지 예시)

CSS 프레임워크의 지형은 방대하다. 하지만, 인기 있는 것들 중에 WebStencils와 잘 어울리는 옵션들 몇 개를 소개하겠다:

Bootstrap(부트스트랩)

가장 널리 사용되는 CSS 프레임워크 중 하나이다. Bootstrap은 광범위한 컴포넌트들, 반응형 그리드(grid) 체계, 방대한 도움말 문서집을 제공한다. 이는 특히 기업용 애플리케이션들과 신속한 프로토타이핑에 적합하다. 즉, 독특한 디자인보다는 일관되고 전문적인 외형이 더 중요한 경우에 좋다.

Bootstrap은 ‘컴포넌트-기반’이다: 즉 여러분은 미리 정의된 클래스를 사용한다. 그 클래스들은 버튼, 폼, 카드, 모달(modal) 및 기타 UI 요소들을 위해 미리 만들어져 있다. 또한 JavaScript도 포함하고 있다. 그래서 드롭다운이나 툴팁과 같은 대화형 컴포넌트들에서 사용된다.

Bulma(불마)

현대적인 CSS 프레임워크이고, 순수 CSS이다: JavaScript 의존성이 없다. Bulma는 Flexbox를 사용해 배치(layout)를 한다. 그리고 깔끔하고 현대적인 미학을 갖추고 있다. Bootstrap보다 커스터마이징하기가 더 쉽다. 그리고 학습 곡선이 더 완만하다.

불마는 모듈식(modular)이다. 그래서 여러분에게 필요한 부분들만 넣을 수 있다. 도움말 문서집이 훌륭하다. 그리고 추가 모듈들의 생태계가 성장하고 있다.

[더 자세한 정보 확인](#)

PicoCSS(피코CSS)

최소한의(minimal) CSS 프레임워크이다. 시맨틱(semantic) HTML에 집중한다. PicoCSS는 네이티브 HTML 요소들을 아름답게 스타일링한다. 수십 개의 유틸리티 클래스들이 필요없다. `<button>`이라고 쓰면 훌륭하게 표현된다: `class="btn btn-primary"`와 같은 코드는 필요 없다.

이 방식은 내용 중심의 사이트들에 신선함을 준다. 그리고 HTML의 복잡성을 크게 줄여준다.

[더 자세한 정보 확인](#)

BeerCSS(비어CSS)

Material Design(머티리얼 디자인) 프레임워크이다. 단순함을 염두에 두고 구축되었다. BeerCSS는 구글의 머티리얼 디자인 컴포넌트들과 스타일링을 가벼운 패키지 하나 안에 넣어 제공한다. 이것은 ‘모바일-우선’이다. 하지만 데스크톱에서도 훌륭하게 작동한다.

[더 자세한 정보 확인](#)

DaisyUI(데이지UI)

컴포넌트 라이브러리이다. Tailwind CSS 위에서 구축되었다. DaisyUI는 미리 스타일링된 컴포넌트들을 제공한다. 그러면서도 여러분이 Tailwind의 유틸리티 클래스들에게 접근할 수 있도록한다. 그래서 두 세계의 장점을 모두 누린다: 컴포넌트의 편리함 그리고 유틸리티의 유연성.

[더 자세한 정보 확인](#)

이 프레임워크들을 사용하기

대부분의 프레임워크의 경우, 통합은 매우 간단하다. 배치(layout) 템플릿에 CDN 링크를 추가하면 된다:

```
<!-- baseLayout.html -->
```

```

<!DOCTYPE html>
<html>
<head>
  <title>@page.title</title>

  <!-- Bootstrap -->
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css">

  <!-- 또는 Bulma -->
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/bulma@@1.0.4/css/bulma.min.css">

  <!-- 또는 PicoCSS -->
  <link rel="stylesheet"
        href="https://cdn.jsdelivr.net/npm/@picocss/pico@@2/css/pico.min.css">

  @RenderHeader
</head>
<body>
  @RenderBody
</body>
</html>

```



Note

@@ 문장 구성을 눈여겨 보라. CDN URL 안에 @ 기호가 있는 경우에는 그 기호를 두 번 반복해야 한다. 그래서 WebStencils 템플릿 문장 구성이 아니라고 구별할 수 있도록 한다.

이게 전부다. 이제 여러분의 템플릿들은 이 프레임워크가 제공하는 어떤 클래스들과 컴포넌트들이든 모두 사용할 수 있다. CDN 링크를 바꾸면, 프레임워크가 바뀐다. 여러분이 어떤 CSS를 사용하고 있는지 WebStencils는 전혀 알지 못한다. 신경 쓰지도 않는다

Tailwind(테일윈드)라는 특수한 사례

Tailwind CSS(테일윈드CSS)는 소규모 프로젝트나 빠른 실험인 경우에는, 다른 프레임워크들과 동일한 방식으로 작동한다. 그저 CDN 링크를 추가하기만 하면 끝이다. 하지만 실제로 운영되는 애플리케이션에게는 흥미로운 도전 과제를 제시한다. Tailwind가 작동하는 방식 때문이다.

Tailwind의 접근 방식을 이해하기

Tailwind는 ‘유틸리티 우선(utility-first)’ CSS 프레임워크이다. 인터페이스를 작성할 때, 여러분은 미리 구축된 컴포넌트들이 아니라, 단일 목적을 가진 작은 유틸리티 클래스들을 사용한다:

```

<!-- 전통적인 CSS의 접근 방식 -->
<button class="btn btn-primary">클릭하세요</button>

<!-- Tailwind 유틸리티의 접근 방식 -->
<button class="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600">
  클릭하세요
</button>

```

이 방식은 여러분에게 엄청난 유연성을 제공한다. 왜냐하면 커스텀 CSS를 작성하지 않고도 어떤 디자인이든 구축할 수 있기 때문이다. 하지만 한 가지 문제가 있다: Tailwind는 수천 개의 유틸리티 클래스들을 만들어 낸다. 만약 여러분이 이것들을 모두 포함한다면, 여러분의 CSS 파일은 3~4MB에 달할 것이다. 이는 실제 운영 사이트들에게 엄청난 초기 페이로드(payload)가 된다.

해결책은 빌드 절차이다. Tailwind는 여러분의 HTML 파일들을 스캔한다. 그래서 무슨 클래스가 실제로 사용되고 있는지 식별하고, 오직 그 클래스들만 담은 CSS 파일을 생성한다. "트리 셰이킹(tree-shaking)" 또는 "퍼징(purging)"이라고 하는 절차이다. 덕분에, CSS 파일 크기는 약 5~50KB 정도로 줄어든다.

전통적인 Tailwind 작업 흐름이 요구하는 것들:

- [Node.js](#) 가 설치되어 있어야 한다
- [npm \(Node Package Manager\)](#)
- 빌드 도구: [Vite](#) 또는 [Webpack](#) 등등
- `package.json` 파일 하나 (여기에 의존성들을 담는다)
- `npm install` 그리고 `npm run build` 명령어를 실행한다

RAD Studio(라드 스튜디오) 개발자들이 그저 웹 애플리케이션들에 스타일을 적용하고 싶은 경우라면, 이것은 상당히 과한 부담(overhead)이다. 여러분은 본질적으로 자바스크립트(JavaScript) 개발 환경을 구축하는 셈이다. 그저 CSS를 만들어 내기 위해서 말이다.

RAD 방식: 독립형 CLI

Tailwind는 독립형 CLI 바이너리를 제공한다. 그러면 Node.js를 설치할 필요가 없다. 이 단일 실행 파일은 Node.js 버전이 수행하는 모든 작업을 처리해 준다. npm 생태계가 없어도 된다.

이 실행 프로그램 덕분에, Tailwind를 RAD Studio와 함께 사용하는 것은 엄청나게 간단하다. Node.js, npm, package.json, 빌드 도구들이 없어도 된다. 그 대신, 여러분이 이렇게 하면 된다:

1. [독립형 CLI 바이너리](#) (파일 한 개)를 다운로드한다
2. Delphi 안에서, post-build event(빌드 후 이벤트) 하나를 구성한다
3. 빌드 시스템이 모든 것들을 자동으로 처리하게 내버려둔다

이 방식을 더 좋게 할 수도 있다. 여러분은 하이브리드 방식을 사용할 수 있다. 개발 중에는 빌드 절차를 빼고, 운영 시에는 최적화를 유지하는 방식이다.

하이브리드 개발 작업 흐름

영리한 해결책이다. CSS 소스들을 빌드 구성(build configuration)에 따라 전환하는 것이다:

Debug 빌드:

- Tailwind의 CDN을 사용한다
- 모든 유틸리티 클래스들을 즉시 사용할 수 있다
- 빌드 단계가 필요 없다
- 템플릿 변경 사항이 즉시 반영된다

Release 빌드:

- 최적화된 CSS를 만들어 낸다. post-build event를 사용하면 된다
- 오직 여러분이 사용하고 있는 클래스들만 포함한다
- 파일 크기가 매우 작다(일반적으로 5~50KB)
- RAD Studio에 의해 자동으로 처리된다

여러분의 애플리케이션이 무슨 모드(mode)로 실행되고 있는지를 감지해 알맞은 CSS를 적재(load)하도록 한다. 그러기 위해 템플릿에서는 WebStencils의 조건부 문장 구조를 사용한다:

```
<!-- baseLayout.html -->
<head>
  <title>@page.title</title>

  @if (env.debug) {
    <!-- 개발: 전체 Tailwind (CDN을 통해 받음) -->
    <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
  } @else {
    <!-- 실제 운영: 최적화되어 생성된 CSS -->
    <link rel="stylesheet" href="/static/css/output.css">
  }

  @RenderHeader
</head>
```

`env.debug` 변수는 여러분의 Delphi 코드 안에서 지정된다. 빌드 구성을 기반으로 정해지도록 한다:

```
// WebModule의 초기화(initialization) 안에서
procedure TWebModule1.SetupEnvironmentVariables;
begin
  // env.debug를 자동으로 템플릿들 안에서 사용할 수 있도록 한다
  WebStencilsEngine.AddVar('env', nil,
    function(AVar: TWebStencilsDataVar; const APropName: string;
```



```

    var AValue: string): Boolean
begin
    if SameText(APropName, 'debug') then
    begin
        AValue := {$IFDEF DEBUG}'True'{$ELSE}'False'{$ENDIF};
        Result := True;
    end;
end);
end;

```



Note

위 코드는 `env`의 값을 정의하고 있다. 여러 가지 방법이 있지만, 여기에서는 익명 메소드 (anonymous method)를 사용하고 있다. `env`의 값을 정의하는 방법으로는 `OnValue` 이벤트를 사용할 수도 있고, 커스텀 오브젝트를 생성해 환경 정보를 다루도록 할 수도 있다

빌드 절차(Build Process)를 구성하기

실제 CSS 생성은 자동으로 이루어진다. Delphi의 post-build events(빌드 후 이벤트)를 통해서 처리된다. 여러분의 프로젝트 옵션에서, Release(릴리스) 빌드를 구성해 Tailwind CLI를 실행하도록 만들면 된다:

빌드 후 이벤트 명령 (Post-build event command):

```

tools\tailwindcss.exe -i src\input.css -o templates\static\css\output.css --minify
--content "templates\**\*.html"

```

이 명령이 하는 일들:

- `src/input.css` 를 입력으로 받는다(이 파일은 그저 Tailwind를 임포트한다)
- `templates/` 디렉토리 안에 있는 모든 HTML 파일들을 스캔한다
- 최적화되고 축소된 CSS 파일을 만들어 낸다
- `templates/static/css/output.css`로 결과물을 출력한다

이 설정은 오직 Release(릴리스) 빌드들에 대해서만 구성하라. Debug(디버그) 빌드에서는 이를 완전히 건너뛰도록 하고 그 대신 CDN을 사용하도록 한다.

지원을 하는 파일들

이 절차가 작동하도록 하는 것은 이 두 개의 작은 구성 파일이다:

src/input.css (Tailwind를 임포트 한다):

```
@import "tailwindcss";
```

tools/tailwind.config.js (어디에서 클래스들을 찾을 것인지 그 위치를 Tailwind에게 알려준다):

```
module.exports = {
  content: ["templates/**/*.html"],
  safelist: [
    // Delphi(델파이) 코드 안에서 생성되는 모든 동적 클래스들을 추가하라.
    'bg-red-500',
    'text-green-600'
  ]
}
```

safelist는 여러분이 Delphi 코드 안에서 동적으로 생성하게 될 클래스들을 위한 목록이다 (예: 데이터에 따른 조건부 색상). 동적으로 생성되는 클래스들은 Tailwind가 HTML을 스캔하는 방식으로 감지할 수 없다. 그러니, 여러분이 그것들을 명시적인 목록으로 작성해 주어야 한다.

[GitHub 저장소](#) 안에는 완전하게 작동하는 예시가 들어 있다. 거기에는 모든 타일들이 구성되어 있다.

프레임워크 선택에 관한 참고 사항

CSS 프레임워크의 지형은 매우 혼잡하다. 그리고 꽤 정파적이다. 개발자들은 저마다 강한 의견들을 가지고 있다. 유틸리티 우선(utility-first) 방식과 컴포넌트 기반(component-based) 방식에 대해, Bootstrap의 보편성과 Tailwind의 유연성에 대해, 프레임워크들이 도움이 되는지 혹은 해로운지에 대해 생각이 다양하다.

WebStencils 사용자들에게 이 논쟁들은 대체로 핵심에서 벗어난 이야기이다. 여러분은 프레임워크를 선택할 때 여러분의 서버 쪽 아키텍처를 고려할 필요가 없다. 별개로 판단하면 된다. 프로젝트, 팀의 기술력, 디자인 요구 사항들에 적합한 것을 사용하면 된다. WebStencils는 이들 모두를 동등하게 잘 수용한다.

여러분이 선택하는 프레임워크는 여러분의 HTML에게 영향을 준다. 하지만, 여러분의 델파이 코드나 여러분의 WebStencils 아키텍처에게는 영향을 주지 않는다. 그리고 그렇게 되는 것이 옳다.

맺음말

WebStencils가 CSS를 가리지 않는 것은 하나의 기능이다. 한계(limitation)가 아니다. 즉, 스타일 적용이라는 논의로부터 분리된다. 그래서 여러분은 수단을 선택할 자유가 있다. 여러분의 프로젝트에 적합한 것들을 선택하면 되고, 언제든지 변경할 수 있다. 변경해도, 애플리케이션을 다시 작성할 필요가 없다.

Tailwind(테일윈드) 통합 사례는 복잡한 빌드 요구 사항들을 가진 프레임워크들마저도, WebStencils 생태계 안에서 훌륭하게 잘 작동한다는 것을 보여준다. ‘하이브리드 CDN/CLI’ 접근 방식은 개발 단계의 마찰을 제거한다. 그러면서도 실제로 운영될 결과물을 최적화 한다. 이 모든 것들을 표준 Delphi(델파이) 빌드 도구들을 통해 이룰 수 있다.

12

배포 옵션들 그리고 Docker(도커)

머릿말

여러분은 WebStencils(웹스텐실즈) 애플리케이션을 구축했다. 그리고 개발 과정에서 훌륭하게 동작했다. 이제 실무적인 질문이 뒤따른다: 이것을 실제로 어떻게 배포할 것인가?"

이에 대한 답변을 들으면 놀랄 수도 있다: 여러분의 애플리케이션을 독립형 실행 프로그램으로 실행하면 된다. 그것만으로도 실제 운영 환경에서 충분히 사용할 수 있다. 즉, 복잡한 인프라를 도입한 후에야 서비스를 시작할 수 있는 것이 아니다. 독립형 모델은 실제 운영 트래픽을 효과적으로 처리한다. 그리고 대부분의 애플리케이션들이라면 결코 이 수준을 넘을 정도로 커지지 않는다.

따라서, WebBroker(웹브로커)의 아키텍처는 여러분에게 ‘진짜 유연성’을 제공한다. 여러분은 가장 단순한 접근 방식부터 시작하면 된다. 그리고 늘어나는 요구에 맞춰 더 정교한 배포 방식으로 발전시켜가면 된다. 특정 인프라에 여러분을 가두는 프레임워크들과 달리, WebBroker 애플리케이션들은 여러 가지 서로 다른 방식으로 실행될 수 있다. 그리고 각 방식은 저마다 장단점이 있다.

이 장에서는 배포 옵션들을 살펴본다. 효과가 좋은 방식 세 가지에 집중하겠다: 즉, 독립형 애플리케이션, [FastCGI](#)와 [NGINX\(엔진엑스\)](#)를 함께 사용, [Docker](#) 컨테이너를 살펴본다. 또한 전통적인 웹 솔루션들인 [IIS](#)와 [Apache](#) 에 대해서도 설명하고, 왜 이 방식들은 신중한 고려가 필요한지를 알려준다.

이 장의 목표는 여러분을 배포 전문가로 만드는 것이 아니다. 목표는 상황에 가장 알맞은 접근 방식을 선택할 수 있도록 돕는 것, 그리고 각 옵션이 실제 실무에서 무엇을 의미하는지를 이해하도록 돕는 것이다.

여러분의 옵션들을 이해하기

WebBroker 애플리케이션들은 여러 가지 구성(configuration)으로 배포될 수 있다:

독립형 실행 프로그램(콘솔 또는 폼-기반)

여러분의 애플리케이션은 자체적인 웹 서버가 되어 실행된다. 여러분은 .exe(Windows용) 또는 실행 파일(Linux용)을 빌드한다. 그것은 특정 포트(port)를 감시(listen)하고 HTTP 요청(requests)들을 직접 처리한다. 이것은 가장 단순한 배포 모델이다.

FastCGI와 NGINX를 함께 사용 (RAD Studio 13.0부터 새로 도입됨)

여러분의 애플리케이션은 별도의 프로세스가 되어 실행된다. 그 프로세스는 FastCGI 프로토콜을 사용한다. NGINX는 HTTP 요청들을 받는다. 그리고 그 요청들을 여러분의 애플리케이션에게 프록시(proxy) 전달한다. 이는 더 높은 트래픽을 관리해야 하는 애플리케이션들을 위한 권장 실무 배포 방식이다. 이는 단순함과 산업 표준 수준의 요청 처리 능력이 결합된 방식이다.

Apache(아파치) 모듈 또는 IIS/ISAPI

여러분의 애플리케이션은 공유 라이브러리(Apache용 .so, IIS용 .dll)로 컴파일 된다. 그 라이브러리는 웹 서버의 프로세스 안에 적재(load)된다.

Docker(도커) 컨테이너

여러분의 독립형 애플리케이션은 ‘컨테이너 이미지’로 패키징된다. 거기에는 모든 의존성들이 포함된다. 그래서 일관성을 환경 전반에 걸쳐 제공한다. 그리고 배포 인프라를 단순화한다.

이 옵션들에서 핵심적인 차이점은 **상태 유지(stateful)**와 **상태 없음(stateless)**이다:

- **상태 유지** (독립형, FastCGI, Docker): 요청들 사이에서 여러분의 애플리케이션이 계속 실행을 유지한다. 메모리가 유지된다. 세션들이 자연스럽게 작동한다. 데이터베이스 연결들은 풀링(pooled)될 수 있다.
- **상태 없음** (Apache 모듈, IIS/ISAPI): 여러분의 코드는 웹 서버의 프로세스 공간 안에서 실행된다. 웹 서버는 여러분의 모듈을 내리거나(unload), 다시 적재(load)할 수 있다. 이 과정에서 메모리 상의 모든 상태는 없어진다. 그래서 세션들을 위해 데이터베이스나 파일 기반의 저장소가 필요하다. 또한 데이터베이스 연결 풀(pools)을 처리하는 과정이 더 복잡하다.

대부분의 WebStencils 애플리케이션들에게는 ‘상태 유지(stateful)’ 배포 방식이 실용적인 선택이다.

독립형 배포(Standalone Deployment)

독립형 모델은 WebBroker의 기본 설정이다. 여러분은 일반 실행 프로그램을 만든다. 그것은 자체 HTTP 서버를 포함하고 있다. 따라서 별도의 외부 웹 서버가 필요없다.

작동 방식

여러분이 RAD Studio(라드 스튜디오)에서 WebBroker 애플리케이션을 만들 때, 다음 중 하나를 선택한다:

- **콘솔 애플리케이션**: 명령줄 프로그램으로 실행된다. Linux 서버, Windows 서비스들에 이상적이다.
- **폼-기반 애플리케이션**: 단순한 GUI가 있어서 서버를 시작/중지 할 수 있다. 오직 Windows에서만 사용할 수 있다.

두 방식이 사용하는 기반 HTTP 서버는 똑같다(Indy의 `TIdHTTPWebBrokerBridge`). 화면만 다르다.

이것은 독립형 서버의 핵심 코드이다:

```
procedure RunServer(APort: Integer);
var
  LServer: TIdHTTPWebBrokerBridge;
begin
  LServer := TIdHTTPWebBrokerBridge.Create(nil);
  try
    LServer.DefaultPort := APort;

    // 연결 제한(limit)을 늘려라 그러면 동시 처리 능력(concurrency)이 더 좋아진다
    LServer.MaxConnections := 0; // 0 = 제한없음 (기본설정은 '제한있음' 이다)
    LServer.ListenQueue := 500; // 백로그(Backlog) 큐 크기 (기본값은 15)
    LServer.KeepAlive := False; // HTTP Keep-Alive를 비활성화: 단순성을 위해

    LServer.Active := True;
    WriteLn('Server started on port ', APort);
    WriteLn('Press Enter to stop...');
    ReadLn;

    LServer.Active := False;
  finally
    LServer.Free;
  end;
end;
```

이 가이드는 WebBroker를 심도 있게 다루지 않는다. 따라서 모든 프로퍼티들, 최적화를 위한 고급 방식들 등을 설명하지 않겠다. 하지만, 가장 일반적인 튜닝 파라미터들로는 `MaxConnections`, `ListenQueue`, `KeepAlive` 등이 있다. 이것들은 동시 요청 등을 서버가 어떻게 처리하는지 그 방식을 제어한다. 알아야 할 점이 있다. 이 설정들은 여러분의 애플리케이션이 자신의 HTTP 서버를 직접 제어할 때만 적용된다. 즉, Apache(아파치)나 IIS를 사용할 때는 적용되지 않는다.

실제 운영 환경에서의 지속적 성공 가능성(Production Viability)

이제 이런 생각이 들 수 있다: "단순한 독립형 실행 프로그램이 정말 실제 운영 트래픽을 처리할 수 있을까?"

물론 가능하다. 이 독립형 모델은 개발자 대부분이 생각하는 것보다 '실제 운영 환경'에 훨씬 더 적합하다. 평균적인 사양의 하드웨어에서 수행한 테스트 결과, 독립형 WebBroker 애플리케이션은 표준적인 워크로드에 대해 초당 1,000건 이상의 요청들을 처리할 수 있었다. 이것은 장난감이 아니다. 정식 실제 운영을 위한 배포 옵션이다.

주요 요구 사항: **NGINX(엔진엑스)를 앞단에 두어 SSL/TLS를 처리하라.** 이는 필수다. 선택이 아니다. NGINX는 SSL을 마무리한다. 정적 파일들을 효율적으로 처리한다. 실제 운영 등급의 HTTP 계층(layer)을 제공한다. 여러분의 Delphi 애플리케이션은 비즈니스 로직에만 집중하게 된다.

초기부터 과잉 설계(over-engineer)를 하지 마라. 독립형 배포에서 시작하라. 더 단순하고 더 쉽게 디버깅할 수 있기 때문이다. 다른 옵션으로 옮기는 것은 구체적인 근거가 있을 때 하면 된다. 단지 독립형 방식이 실제 운영용으로 "너무 단순해" 보인다는 이유만으로 이 옵션을 배제하지 마라.

배포 절차 (Deployment Process)

독립형 WebBroker 애플리케이션을 배포하는 절차는 Windows(윈도우)와 Linux(리눅스) 모두에서 매우 간단하다. 그리고 거의 동일하다: 애플리케이션을 64비트용 실행 파일 하나로 컴파일한다(Linux용 크로스-컴파일인 경우 [PAServer](#)를 사용). 그리고 그 실행 파일과 리소스(resources) 폴더를 복사해 여러분의 서버에 넣는다. 그런 다음, 기본적인 런타임 설정들을 구성하면 된다.

그 리소스 폴더(HTML 템플릿, CSS, JavaScript, 이미지 등을 담은 폴더)의 위치는 반드시 실행 파일을 기준으로 올바른 상대 위치여야 한다. 또는 절대 경로로 구성되어야 한다. 이 설정은 구성(config) 파일 또는 환경 변수들을 활용하면 된다.



실제 운영 환경을 위해서는, 리눅스 *systemd*(시스템디) 서비스 또는 윈도우 서비스 (*Windows Service*)를 생성하라. 그러면 서버가 부팅될 때 애플리케이션이 자동으로 시작된다. 그리고 만약 비정상적으로 종료되면 자동으로 재시작된다. 이는 실행 파일을 수작업으로 실행하는 것보다 훨씬 더 신뢰할 수 있는 방식이다.

FastCGI를 NGINX와 함께 사용하기 (RAD Studio 13.0 이상)

FastCGI(패스트 CGI) 지원은 RAD Studio 13.0에서 도입되었다. 이것은 WebBroker 애플리케이션들을 NGINX(엔진엑스) 뒷단에 배포하기 위한 현대적인 표준을 제공한다.

작동 방식

FastCGI는 웹 서버들과 애플리케이션 프로세스들 사이의 통신 프로토콜이다. 여러분의 애플리케이션은 컴파일되어 일반 독립형 실행 파일이 된다. 즉, 라이브러리나 모듈로 컴파일되는 게 아니다. 하지만, HTTP가 아니라 FastCGI를 사용해 NGINX와 통신한다.

그 아키텍처는 간단하다. 여러분의 애플리케이션은 백그라운드 프로세스로 실행된다. 그것은 특정 포트(전형적으로 9000번)를 감시한다. NGINX는 HTTP 요청들을 클라이언트들로부터 받고 이를 FastCGI 형식으로 변환해, 여러분의 애플리케이션에게 전달한다. 여러분의 애플리케이션은 그 요청을 WebBroker를 통해 받아서 처리한다. 그리고 그 응답을 다시 FastCGI를 통해 보낸다. 그러면 NGINX가 최종 HTTP 응답을 클라이언트에게 전달한다.

NGINX 구성(Configuration)

이것은 FastCGI를 위한 최소한의 NGINX 구성이다:

```
# 업스트림(upstream) FastCGI 서버를 정의한다
upstream fastcgi_backend {
    server 127.0.0.1:9000;
    keepalive 32;
}

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

    location / {
        fastcgi_pass fastcgi_backend;

        # 요구되는 FastCGI 파라미터들
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param QUERY_STRING $query_string;
        fastcgi_param REQUEST_METHOD $request_method;
        fastcgi_param CONTENT_TYPE $content_type;
        fastcgi_param CONTENT_LENGTH $content_length;
        fastcgi_param PATH_INFO $uri;
        fastcgi_param REQUEST_URI $request_uri;
        fastcgi_param SERVER_PROTOCOL $server_protocol;
        fastcgi_param REMOTE_ADDR $remote_addr;
        fastcgi_param SERVER_NAME $server_name;
        fastcgi_param SERVER_PORT $server_port;
    }
}
```

이 구성(configuration)은 모든 요청들을 여러분의 FastCGI 애플리케이션에게 전달하도록 NGINX에게 지시한다. 그리고 여러분의 애플리케이션이 요청들을 올바르게 처리하는데 필요한 HTTP 파라미터들을 담고 있다. 물론, 위 예시는 최소한의 구성을 예로 들었다. 여러분의 애플리케이션에는 다른 NGINX 구성이 더 필요할 수도 있다.



Note

기술적으로 볼 때, FastCGI는 Windows(윈도우)에서도 잘 작동한다. 하지만, NGINX와 Apache는 Linux(리눅스)에서 현저히 더 좋은 성능을 발휘한다. Windows 실제 운영 배포를 위해서라면, 독립형 방식을 채택하고 NGINX(리눅스에서 실행시킨다)를 리버스 프록시(reverse proxy)로 활용하는 방식을 고려하라. 또는 IIS 사용이 대안이 될 수 있다.

전통적인(traditional) 웹 서버 통합

Apache 모듈들과 IIS/ISAPI 익스텐션들은 웹 애플리케이션들을 위한 전통적인 배포 모델을 대표한다. 여러분의 Delphi 코드는 공유 라이브러리로 컴파일 된다. 그 라이브러리는 웹 서버의 프로세스 공간 안에 적재(load)된다.

‘상태 없음(statelessness)’의 문제

여기에서 ‘근본적인 주의 사항’이 생긴다. 여러분의 코드가 Apache 모듈이나 IIS/ISAPI DLL로 실행되기 때문에, 여러분은 그것의 프로세스 수명 주기를 제어하지 못한다: 그 수명 주기 제어는 그 웹 서버가 한다.

웹 서버가 하게 되는 작업들:

- 언제든지, 여러분의 모듈을 내리고(unload)하고 다시 적재(reload)한다.
- 여러분 모듈의 여러 독립 인스턴스들을 실행(run)한다.
- 애플리케이션 풀(IIS의 경우) 또는 프로세스들(Apache의 경우)을 재활용(recycle)한다.

이런 일들이 발생할 때마다, 모든 메모리 상의 상태(in-memory state)를 잃게 된다. 이런 것들이 해당된다:

- **세션 데이터:** 데이터베이스 또는 파일에 저장되어 있지 않다면 손실된다.
- **데이터베이스 연결 풀(pool):** 매번 다시 생성해야 한다
- **인메모리 캐시:** 다시 적재(reload)될 때마다 깨끗이 비워진다.

WebStencils에 내장된 세션 관리자(TWebSessionManager)는 세션들을 메모리 안에 저장한다. 이 방식은 독립형 또는 FastCGI 배포와 완벽하게 어울린다. 왜냐하면 해당 프로세스가 계속해서 실행되는 환경이기 때문이다. 반면 Apache나 IIS 모듈들을 사용하는 경우라면, 세션들은 예측이 불가능하다.

해결책: 외부(external) 세션 저장소

만약 여러분이 반드시 Apache 모듈이나 IIS/ISAPI를 사용해야 한다면, 세션 저장소를 구현하라. 즉 저장소가 여러분의 애플리케이션 밖에서 계속 유지되게 하라. 일반적인 접근 방식은 이 두 가지다:

- **데이터베이스 기반 세션:** 세션 데이터를 데이터베이스 테이블 (또는 redis) 안에 저장한다. 각 요청은 그 데이터베이스를 대상으로 세션 상태를 읽고 쓴다.
- **파일 기반 세션:** 세션 데이터를 디스크 상의 파일들에 저장한다. 웹 서버는 이 파일들에 접근할 수 있다. 프로세스가 다시 시작되는 경우에도 그렇다.

이 두 접근 방식 모두 복잡성과 지연 시간(latency)을 가중시킨다. 하지만, 웹 서버가 여러분의 프로세스 수명 주기를 제어하는 경우라면 꼭 필요하다(necessary). FastCGI 방식(프로세스가 계속 실행되는 상태로 유지되는 방식)이 모듈 기반 배포보다 일반적으로 선호되는 여러 가지 이유들 중 하나가 바로 이것이다.



Note

만약 여러분의 애플리케이션이 수평적 확장(scale horizontally)을 할 필요가 있다면, 인-메모리 세션 체계는 맞지 않다. 어떤 상황이든지, 데이터베이스(또는 redis)가 뒷받침하는 세션 체계를 반드시 구현해야 한다.

늘 생각해야 하는 기타 주의 사항들

상태 관리 이슈 외에도, Apache와 IIS 모듈들은 구성 상의 복잡성을 초래한다. 이런 것들이다:

경로(Path) 처리: 독립형 애플리케이션 안에서, `/login`은 `http://yourserver:8080/login`을 의미한다. 반면 Windows ISAPI 모듈 안에서는, `http://yourserver/WebApp.dll/login`을 의미할 수 있다. 여러분은 이 모듈 이름 즉 스크립트 이름을 고려해 모든 URL들과 리다이렉트들 안에 반영해 한다.

권한(Permission)들: 웹 서버의 사용자 계정은 권한이 필요하다. 여러분의 템플릿들에 대한 읽기 권한, 로그 디렉토리들에 대한 쓰기 권한, 적절한 데이터베이스 권한이 있어야 한다. 권한이 잘못되면, 파악하기 어려운 오류들이 가끔 발생할 수 있다.

디버깅: 여러분은 RAD Studio(라드 스튜디오) 안에서 독립형 버전을 사용해 애플리케이션을 개발할 것이다. 하지만, 이 방식을 선택하면, 배포하는 것은 모듈이 된다. 이 차이는 때때로 예기치 않은 동작들로 이어진다. 그것들은 디버깅하기 어렵다. 따라서 실제 운영 환경의 로깅(logging)에 크게 의존해야 한다. 그래서 이슈를 찾아낼 수 있어야 한다. 이런 이슈들을 개발 환경에서 재현하기는 어렵다.

전통적인 CGI에 관한 참고 사항

WebBroker는 전통적인 CGI([Common Gateway Interface](#))도 지원한다. CGI는 FastCGI의 전신이다.

CGI는 여전히 존재한다. 그래서 레거시 호환성을 위한 하나의 옵션이 된다. 하지만, 새로 배포하는 경우라면, FastCGI가 좋다. FastCGI는 CGI의 모든 이점들을 제공하면서도 (이 오래된 기술이 가진) 성능 손실, 유의점 등이 없다. 지금은 빠짐없이 소개하느라 CGI 옵션을 언급했다. 하지만, FastCGI 사용을 권장한다.

Docker(도커) 배포

Docker(도커)는 현대적인 배포 방식을 대표한다: 여러분의 애플리케이션 그리고 그것의 모든 의존성들은 하나의 컨테이너 이미지 안에 패키징된다. 그 이미지는 어떤 환경에서든 일관성있게 실행된다.

WebStencils 애플리케이션들을 위해 Docker 가 제공하는 이점들:

- **일관된 환경:** 개발, 테스트 수행, 실제 운영 모두 동일한 버전을 실행(run)한다
- **간편한 배포:** `docker run`은 여러분의 애플리케이션을 어디로든 배포한다
- **의존성 격리:** 여러분의 컨테이너 안에는 필요한 Linux 라이브러리들이 담긴다
- **확장성(Scalability):** 여러 개의 컨테이너들을 로드 밸런서 뒤에서 쉽고 빠르게 띄울 수 있다.

Docker 이미지를 생성하는 방법

프로젝트마다 다르지만, Delphi 바이너리들로 Docker 이미지를 생성하는 것은 놀라울 정도로 간단하다. 델파이는 네이티브 실행 파일을 생성하므로, 어떤 슬림 리눅스 기반 이미지도 여러분의 애플리케이션을 실행할 수 있다: 복잡한 런타임 의존성 없이 필요없다.

WebStencils 애플리케이션을 위한 최소한의 Dockerfile(도커파일)은 다음과 같다:

```
FROM debian:13-slim

WORKDIR /app

# 여러분이 컴파일한 Linux 실행 파일(executable)과 리소스들을 복사한다
COPY Linux64/Release/WebStencilsDemo /app/
COPY resources /app/resources

RUN chmod +x /app/WebStencilsDemo

EXPOSE 8080

CMD ["/app/WebStencilsDemo"]
```

기본적인 기능을 위한 설명은 이것으로 충분하다. 위 파일은 여섯 가지 필수 명령어들을 사용하고 있다: 기반 이미지, 작업 디렉토리, 파일 복사, 실행 파일 만들기, 포트 노출, 실행 순서로 진행된다.

실제 운영급 Docker(도커) 이미지를 만들기

앞선 예시는 기능적으로 잘 작동한다. 그렇지만, Docker 이미지를 빌드할 때의 몇 가지 모범 관행들이 빠져있다. 이제 이 데모의 [GitHub 저장소](#) 안에 들어 있는 Dockerfile을 살펴보자.

```
FROM debian:13-slim

# 보안 업데이트 설치와 정리 작업을 하나의 레이어에서 수행
RUN apt-get update && \
    apt-get upgrade -y && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

# non-root user 생성
RUN useradd -m -u 1001 appuser && \
    mkdir -p /app/logs /app/data /app/backup && \
    chown -R appuser:appuser /app

WORKDIR /app
```


1. **Docker 빌드 구성** - 도커를 위해 사용될 새로운 빌드 구성을 생성한다
2. **자동화된 스크립트** - PowerShell(파워셸) 스크립트. post-build event에 실행된다
3. **이미지 생성** - 그 스크립트는 여러분의 Docker 이미지를 자동으로 만들어 준다

작업 흐름: "Docker" 구성을 선택한다. 여러분의 프로젝트를 빌드한다 (Ctrl+Shift+F9). 그러면 Delphi가 실제 운영 급 컨테이너 이미지를 생성한다. 수동으로 도커 명령어들을 입력할 필요가 없다.

사전 요구 사항들과 설정

여러분에게 필요한 것들:

- [WSL2](#) 가 Windows에 설치되고 구성되어야 한다
- Docker CLI가 WSL2 안에 설치되어야 한다 (이 작업흐름에는 Docker Desktop을 권장하지 않음)
- Linux 플랫폼 구성 (Delphi 안에)
- PAServer (크로스-컴파일을 위해)

컨테이너화된 여러분의 애플리케이션을 실행하기:

```
# 기본적인 실행(run)
docker run -d -p 8080:8080 --name=myapp myapp:latest

# 영속적인 스토리지와 함께
docker run -d -p 8080:8080 \
  -v /host/logs:/app/logs \
  -v /host/data:/app/data \
  --name=myapp myapp:latest
```

볼륨(**-v** 플래그)들은 여러분의 데이터와 로그들이 살아남도록 보장한다. 컨테이너가 재시작 또는 업데이트 되는 경우에도 그렇다.



Note

위의 명령어는 로컬 폴더들을 컨테이너에 바인딩하고 있다. 하지만, 이것들은 Docker(도커) 볼륨들에 매핑될 수도 있다(이것이 더 나은 관행이라고 간주된다). 또한 두 개의 볼륨들이 매핑되어 있는데, 이는 이 Dockerfile 예시가 그렇게(로그 그리고 데이터로) 구성되어 있기 때문이다. 각 프로젝트마다 서로 다른 구성들과 요구 사항들을 가질 수 있다.

완전한 예시가 제공된다

WebStencils(웹스텐실) 데모 저장소에는 완전한 작동하는 Docker 설정(setup)이 들어 있다. 거기에는 모든 구성 파일들, 스크립트들, 도움말 문서집 등이 포함된다. 그 모든 내용을 다시 여기에 적어놓지는 않았다. 여러분이 그 저장소를 탐색해 구체적인 구현 세부 사항들을 직접 보기를 권한다:

GitHub 저장소: github.com/Embarcadero/WebStencilsDemos

이 저장소에 들어있는 것들:

- 완전한 전체 Dockerfile(도커파일)
- PowerShell(파워셸) 자동화 스크립트
- 빌드 구성(Build configuration) 설정
- 배포 (Deployment) 도움말 문서집
- 문제 해결(Troubleshooting) 가이드

이것은 훌륭한 시작점이다. 여기에서 시작한다. 그리고 여러분의 프로젝트들에 맞춰 조정해 가면 된다.

실제 운영 고려 사항들

여러분이 어떤 배포 방식을 선택하든, 실제 운영 관련 고려 사항들 몇 가지는 보편적으로 적용된다:

SSL/TLS 구성(Configuration)

실제 운영 애플리케이션들을 절대로 HTTP를 통해 서비스하지 마라. 내부용 애플리케이션일지라도 그렇다. SSL/TLS는 자격 증명 탈취와 중간자 공격(man-in-the-middle attack)들을 방지한다.

독립형 배포인 경우: NGINX를 앞단에 리버스 프록시로 두어 SSL을 처리하라. 대체로 가장 쉬운 방법이다:

```
server {
    listen 443 ssl;
    server_name myapp.example.com;

    ssl_certificate /etc/letsencrypt/live/myapp.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/myapp.example.com/privkey.pem;

    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

[Let's Encrypt\(렛츠 인크립트\)](#)는 certbot(서트봇)을 통해 무료 인증서(certificates)들을 제공한다.

FastCGI용: NGINX가 SSL을 직접 처리한다. 여러분의 애플리케이션은 인증서들을 전혀 건드리지 않는다.

Docker용: 별도의 컨테이너에서 NGINX를 실행하거나 또는 SSL을 마무리하는(terminate) 로드 밸런서를 사용하라.



만약 여러분이 NGINX에 익숙하지 않고 여러분의 요구 사항들이 엔터프라이즈 수준이 아니라면, [Swag](#) 또는 [NPM](#)(NGINX Proxy Manager)과 같은 프로젝트들을 사용하면 애플리케이션들을 보호하고 리다이렉트하는 과정이 훨씬 더 쉬울 수 있다.

로깅(Logging) 및 모니터링(Monitoring)

실제 운영 애플리케이션들의 동작은 관찰될 수 있어야 한다.

중앙 집중식 로깅을 고려하라. 단일 인스턴스 배포든 다중 인스턴스 배포든 상관없이 그렇다. RAD Studio는 시스템 로그들에 접근하는 다양한 방식들을 제공한다. 또한 훌륭한 써드-파티 로깅 솔루션들도 있다. 그것들은 로그 능력을 더 확장해준다. 예: [LoggerPro](#), [DataLogger](#), [Spring4D logging](#) or [QuickLogger](#).

앞에서 보여준 Docker 헬스 체크는 자동 모니터링과 자동 오케스트레이션(orchestration) 시스템들을 활성화한다. 그래서 상태가 건강하지 않은 컨테이너들을 자동으로 재시작할 수 있다.

환경별 구성(Configuration)

어떤 경우에도, 구성 데이터(configuration data)를 하드코딩하지 마라. 구성 파일들이나 환경 변수들을 사용하라. 모든 중요한 데이터에 대해서 그렇게 해야 한다.

```
// 환경 변수(environment variable)들로부터 읽는다
DatabaseHost := GetEnvironmentVariable('DB_HOST');
DatabasePassword := GetEnvironmentVariable('DB_PASSWORD');
ApiKey := GetEnvironmentVariable('API_KEY');

// 또는 구성 파일(config file)로부터
Config := TIniFile.Create(
    TPath.Combine(AppPath, 'config.ini')
);
```

이 방식은 개발(development), 스테이징(staging), 실제 운영(production) 환경에서 서로 다른 설정들을 적용할 수 있게 해준다. 그러면 코드를 변경할 필요가 없다.

맺음말

WebBroker(웹브로커)의 배포 유연성은 훌륭하다. 하지만, 그렇다고 모든 옵션들이 대등하다는 뜻은 아니다. 상태 유지(stateful) 옵션들(독립형, FastCGI, Docker)은 더 단순하고 신뢰할 수 있다. 왜냐하면 WebStencils(웹스텐실)의 세션 관리와 인증 방식에 자연스럽게 맞기 때문이다. 또한 그 방식은 여러분의 프로젝트들이 메모리 안에 정보를 캐싱/저장할 수 있도록 한다. 그리고 여러분의 애플리케이션 수명 주기를 여러분이 직접 다룰 수 있도록 해준다.

핵심 통찰은 이것이다: 독립형 배포는 그 자체로 실제 운영 환경 사용에 적합하다. 이것은 타협안이나 임시 해결책이 아니다. 충분히 타당하고 합리적인 아키텍처다. 그래서 실제 운영 트래픽을 효과적으로 처리한다. NGINX(엔진엑스)를 앞단에 두어 SSL를 처리하라. 그러면 여러분은 견고한 실제 운영 배포를 가지게 된다. 대부분의 애플리케이션들이라면 이보다 더 큰 것이 필요없다.

Docker 통합과 FastCGI 지원은 명확한 업그레이드 경로로 선택할 수 있다. 단, 그것들로 인한 복잡성을 감수할 만큼 여러분의 요구 사항이 커졌을 때 선택하기 바란다. Docker는 배포의 일관성 그리고 클라우드 플랫폼 통합을 제공한다. FastCGI는 NGINX의 고급 HTTP 기능들을 제공한다. 최소한의 구성 변경만으로 가능하다.

전통적인 웹 서버 모듈들(Apache, IIS)은 여전히 유효한 옵션이다. 그리고 성능 또한 훌륭하다. 하지만, 그것들은 상태 관리의 복잡성을 동반한다. 그 약점들을 감수해야 할만한 가치가 없는 상황들이 있다. 새 애플리케이션인 경우에는 꼭 그렇게 해야 할 필요가 있는지를 따져볼 필요가 있다.

13

RAD Server 통합을 통해 WebStencils를 사용하기

머릿말

RAD Server도 이 새 라이브러리인 WebStencils의 혜택을 받는다. 특별 통합이 개발되어 있다. 따라서 여러분은 RAD Server에서 웹 개발의 모든 잠재력을 활용할 수 있다. 문장 구조, 컴포넌트들 등 지금까지 배운 모든 것들은 RAD Server에서도 그대로 적용된다.

웹스텐실즈를 RAD Server와 통합하기

앞에서 본 예제들과 마찬가지로, RAD Server와 웹스텐실즈를 함께 사용하는 방법은 여러 가지다. 이제부터 그 중 가장 일반적인 패턴 두 가지를 보자:

웹스텐실즈 프로세서(WebStencils Processor)들을 사용하기

이것은 간단한 방식이다. 요청이 있을 때마다 Processor를 사용해 HTML을 만들어 낸다. (Processor는 디자인 타임에 만들 수 있고, 런타임에 프로그램적으로 생성할 수도 있다). 그리고 만들어진 HTML을 RAD Server의 응답 안에 넣어 직접 반환한다. 사용되는 템플릿들은 파일에서 읽어오도록 하거나 또는 상수, 변수 등에 넣어두면 된다. 그런 다음, 그 템플릿들을 프로세서가 처리한다. 그래서 HTML을 만든다.


```

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
  // 이 변수를 교체해 실제로 템플릿이 있는 경로를 가리키도록 할 것
  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> URL로 접근할 수 있을 것이다. 그러면 `LTemplateFile` 변수 안에 정의된 템플릿 파일의 내용이 렌더링되는 것을 볼 수 있다.

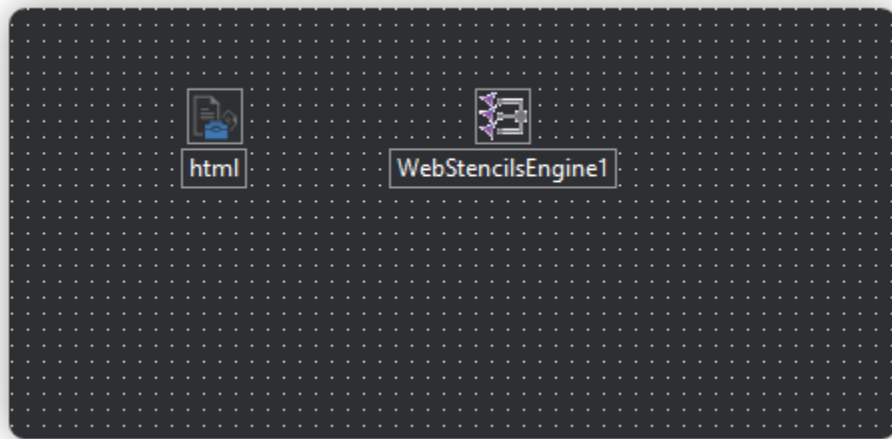


Note

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

웹스텐실즈 엔진(WebStencils Engine)을 사용하기

이 옵션을 선택한다면, 기존 `TEMSFileResource` 컴포넌트와 `TWebStencilsEngine` 컴포넌트를 조합하는 방식을 추천한다. 전자는 파일 시스템에 대한 매핑을 담당한다. 한편, 후자는 HTTP 매핑과 템플릿 처리를 담당한다. 이 두 컴포넌트를 서로 연결할 때는 ‘웹스텐실즈 엔진’의 Dispatcher 프로퍼티를 사용한다.



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

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

`{filename}` 라는 와일드카드를 유의해 보기 바란다. 이 와일드카드가 각 페이지들을 참조하도록 한다.

일단 컴포넌트들을 그렇게 구성해 놓았다면, `PathsTemplates` 를 Engine 안에서 정의할 수 있다. 그 방식은 앞에서 본 것처럼 와일드카드인 `{fileName}` 를 사용할 수 있다. 또는 템플릿 파일들의 이름을 직접 적어도 된다.

파일들을 올바르게 맵핑하려면, `FileResource`에 애트리뷰트들 몇 가지를 정의해야 한다. 앞의 예시에 했던 그대로 하면 된다:

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

이 예시에 따르면, 우리가 접근할 수 있는 엔드포인트는 다음과 같다:

`http://localhost:8080/testfile/{filename}`

위 엔드포인트에서 `{filename}` 에는 어떠한 템플릿이든 들어갈 수 있다. 단 그 템플릿은 `PathTemplate` 프로퍼티 안에서 우리가 미리 정의해 놓은 경로 안에 들어 있는 파일이어야 한다.



만약 동일한 *TWebStencilsEngine* 컴포넌트가 한 개보다 많은 *TEMSFileResource*를 필요로 한다면, 글로벌 메서드인 *AddProcessor*를 사용해 추가로 더 넣을 수 있다.

예시:

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

할 일 목록 앱을 RAD Server 안에 다시 만들어 보기

RAD Server는 WebBroker와 작동 방식이 조금 다르다. RAD Server는 REST를 준수하는 애플리케이션이 되어야 한다. 따라서 메모리 상태를 갖지 않는다. 따라서 몇 가지 사항들을 염두해야 한다.

여러분이 만약 [GitHub 저장소](#) 안에 있는 WebBroker 데모를 받아본다면, 그것이 MVC 방식을 따르고 있다는 것을 알 수 있을 것이다. 동일한 그 저장소 데모 안에는 똑같은 데모인데 RAD Server로 마이그레이션 되어 있는 데모가 있다. 그 둘은 기능이 완전히 똑같다. 그럼 이제, 우리가 그 마이그레이션을 하면서 무슨 리팩토링이 필요했었는지를 나열해 보겠다:

데이터베이스 관리

WebBroker 데모 안에서는, 할 일들이 저장되는 곳이 메모리 안이기 때문에, 싱글톤 패턴을 사용한다. 그래서 멀티-쓰레드 수행이 만들 수 있는 문제를 피했다. RAD 서버 안에서는, 그 할 일들이 메모리 안에 저장될 수 없다 (적어도 쉽지 않다). 따라서 우리는 InterBase 데이터베이스를 대신 사용했다.

그래서 우리의 모델은 데이터베이스에 접근해야 한다. 그 접근은 모델 안에서 직접 정의하지 않았다. 그 대신, 모델인 *TTasks*의 생성자 안에서 *TFDConnection* 컴포넌트를 할당했다. 데모를 단순화하기 위해, *TFDConnection* 컴포넌트 생성을 동일한 RAD Server DataModule 안에서 수행되도록 했다. 하지만, 실제로 사용할 앱이라면, *TFDManager* 컴포넌트 사용을 권장한다.

컨트롤러 인자(argument)들

WebBroker와 RAD Server는 요청들과 응답들이 모두 조금 다르다. 따라서 컨트롤러의 메서드들에게 전달되는 인자(argument)들 안에서 몇 가지 리팩토링을 해야 한다.

Action들을 Endpoint들로

WebBroker는 엔드포인트들을 정의한다. 그리고 그것들을 비즈니스 로직들과 연결하기 위해 Action들을 사용한다. RAD Server에서는 리소스 안에 있는 메서드들은 반드시 자신이 연결되는 특정 URI가 무엇인지를 명시해야 한다. 애트리뷰트를 사용해 명시하면 된다.

요청에 들어 있는 데이터를 처리하기

RAD Server는 JSON을 가지고 작업을 한다. 할 일 목록 앱에서 받은 데이터를 우리의 백엔드로 전달할 때도 그렇고, 그 데이터를 처리할 때도 그렇다. 그래서 몇 가지 변경이 필요하다:

1. **HTMX 확장인 `JSON-enc`를 사용한다:** 이 확장은 백엔드로 보내지는 요청들을 JSON 형식으로 인코딩한다. 즉 (기본 동작 방식인) 폼-데이터 형식 그대로 보내지 않는다. 이 확장을 사용하려면, 간단한 `<script>` 태그를 추가해야 한다. 그리고 우리가 `hx-post` 또는 `hx-put` 요청을 정의해 놓은 태그 안에 반드시 `hx-ext="json-enc"` 애트리뷰트를 명시해야 한다.
2. **데이터 처리 방식 변경:** RAD Server는 요청 데이터를 JSON으로 처리한다. 따라서, 몇 가지 변경을 해야 요청으로부터 값을 얻을 수 있다. 표준 JSON 라이브러리만 사용해도 데모 목적으로는 충분하다.

정적인 리소스 다루기: JS, CSS, 이미지

`TEMSFileResource` 컴포넌트를 사용하면 정적인 파일들을 제공할 수 있다. 독립적인 컴포넌트들을 만들고 그것들 각각을 해당 폴더에 매핑을 했다. 그 폴더들 안에는 알맞은 파일들(JS, CSS, 이미지 등)이 들어 있다.

프론트엔드 소스들

RAD Server는 리소스들에게 의존한다. 그래서 주 URL 뒤 맨 끝에 리소스 이름을 붙여서 요청을 보낸다. 예를 들면 - <https://localhost:8080/web>.

WebBroker 웹사이트에서 만들어진 엔드포인트들을 RAD Server로 이전했다. 그러기 위해서, 웹스텐실즈 템플릿들을 약간 변경해야 했다. 그래서 즉, 새로 생성되는 리소스를 가리키도록 했다.

14

자료들 그리고 추가 학습



아래에 유용한 자료들 몇 가지를 제시해 놓았다. 이 책이 다른 주제들의 잠재력을 훨씬 더 확장하는데 도움이 되는 것들이다:

도움말 문서집 그리고 링크들

라이브 WebStencils 데모

서비스되고 있는 라이브 데모가 있다. 이 데모는 WebBroker(웹브로커)와 Docker(도커)에서 실행된다. 주소는 wsdemo.embarcadero.com 이다. 다운로드가 필요없고, 설정도 필요없다: 그저 브라우저를 열고 바로 탐색을 시작하면 된다.

엠바카데로 블로그 기고들

WebStencils(웹스텐실즈)에 대해 설명하는 기고들 여러 개가 이미 있다. 우리는 정기적으로 더 많은 기고들을 계속 추가하고 있다. 기고를 읽으면, 가장 쉽게 WebStencils에 관한 새 소식과 기능들에 대한 모든 최신 정보를 얻을 수 있다.

[그 기고들은 여기에서 읽을 수 있다](#) (굵긴이: 일부는 한글로 번역되었으며, [여기](#)에서 볼 수 있음)

공식 HTMX 도움말 문서집 (HTMX.org)

공통 키워드들 중 대부분을 이 책이 다루고 있다. 하지만, HTMX는 훨씬 더 많은 것들을 여러분의 프로젝트에 더할 수 있다. HTMX 팀이 제공하는 도움말 문서집은 방대하다. 하지만, 이해하기 좋고, 부담스럽지 않다.

공식 도움말 문서 뿐만 아니라, htmx.org 웹사이트에서도 기고들과 예시들을 제공하고 있다:

- [공식 도움말 문서집](#)
- [예시](#)
- [기술 기고](#)

RAD Server 기술 가이드

만약 여러분이 RAD Server(라드 서버)에 익숙하지 않고 그 모든 가능성을 탐색하고 싶다면, 이 도서를 권한다. RAD Server의 주요 기능들을 안내할 뿐만 아니라, 더 구체적이고 더 수준높은 기능들도 볼 수 있다:

[이 도서 다운로드](#) (옮긴이: 한글 번역본은, [여기](#)에서 볼 수 있음)

HTMX를 MVC 패턴 방식으로 (HTMX.org)

이 웹 페이지는 HTMX를 어떻게 MVC-스타일 웹 애플리케이션에 맞추는지에 대해 요약 설명한다. 얇은 컨트롤러(thin controller)들의 예시를 보여주고, HTMX를 사용해 웹 개발을 할 때 MVC 흐름을 어떻게 구성하는지도 보여준다. 이 내용은 델파이-전용이 아니다. 하지만, 그 개념은 다양한 언어에서 MVC 디자인 패턴에 광범위하게 적용될 수 있다:

[여기로 가면 된다](#)

WebStencils (DocWiki)

우리의 공식 WebStencils 도움말 문서집은 DocWiki 안에 있다:

[여기로 가면 된다](#)

HTMX를 훨씬 더 확장하기

HTMX의 특징인 선언적 방식은 여러분의 프로젝트가 JS에 훨씬 덜 의존할 수 있도록 한다. 그렇지만, 상황에 따라 JS가 필요한 경우가 있다. 순전히 클라이언트-쪽에서 상호작용해야 하는 경우에 그렇다. 그 사례로, 웹사이트의 모습을 변경해 밝은 테마에서 어두운 테마로 적용하는 경우를 들 수 있다. 우리는 그 변경을 백엔드 안에서 다룰 수도 있다. 즉, 서버 쪽에서 다크 모드로 새로 HTML을 만들어 보내 줄 수 있다. 하지만, 쉽게 클라이언트 쪽에서 처리할 수도 있다. 현대식 CSS 라이브러리들 덕분이다.

추가 기능들이 필요하고 그 상호작용을 클라이언트 쪽에서 처리해야 하는 경우에 사용할 수 있는, 매우 작은 JS 라이브러리들이 있다. 이것들은 HTMX와 WebStencils에 매우 잘 통합된다.

AlpineJS

AlpineJS(알파인JS)는 가벼운 자바스크립트 프레임워크다. HTML에 간단한 상호작용을 추가하기 위해 고안되었다. 이것은 선언적 방식으로 DOM 엘리먼트들을 조작한다. 따라서 방대한 자바스크립트 코드를 작성할 필요가 없다.이 기술은 종종 [Vue](#) 또는 [React](#) 와 비교된다. 하지만 이것이 훨씬 더 작다. 그리고 더 쉽게 기존 HTML과 통합된다. AlpineJS가 이상적인 경우는 자바스크립트 행위를 웹 페이지에 추가할 때이다. 거대한 프레임워크로 인한 복잡성이 없기 때문이다:

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

도움말 문서집: [AlpineJS Docs](#)

Hyperscript

Hyperscript(하이퍼스크립트)는 일종의 스크립트 언어다. 이벤트 처리와 로직을 HTML 안에서 단순화하기 위해 고안되었다. 자바스크립트는 장황한 문장 구조가 필요하지만 Hyperscript는 HTML 애트리뷰트들 안에 직접 심어 넣도록 설계되었다. 그 초점은 이벤트-기반 프로그래밍을 더 직관적으로 하도록 만드는 것이다. 그러기 위해 자연어를 닮은 문장 구조를 사용한다. 따라서 개발자들이 복잡한 자바스크립트를 작성하지 않아도 동적 행위를 만들 수 있다:

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

도움말 문서집: [Hyperscript Docs](#)

영어 원본 다운로드: <https://lp.embarcadero.com/HTMX-WebStencils>

15

부록: 약어(Acronyms, Abbreviations)

A

AJAX - Asynchronous JavaScript and XML - 동적 웹 애플리케이션들을 만들기 위한 기법 중 하나. 이 기법은 페이지 전체를 다시 로드하지 않고 페이지의 일부분들을 업데이트할 수 있도록 한다

API - Application Programming Interface - ‘프로토콜들과 도구들의 집합’들 중 하나. 이 집합은 서로 다른 소프트웨어 애플리케이션들이 서로 통신할 수 있도록 한다.

ASP.NET - Active Server Pages .NET - Microsoft의 웹 애플리케이션 프레임워크. .NET 위에서 구축됨

C

CDN - Content Delivery Network - 분산된 서버들의 네트워크 중 하나. 웹 내용 전달을 지리적 위치에 기반해 제공한다

CGI - Common Gateway Interface - 표준 프로토콜 중 하나. 웹 서버들이 프로그램들을 실행하고 동적 내용을 만들어 낼 수 있도록 한다

CLI - Command Line Interface - 텍스트-기반 인터페이스 중 하나. 소프트웨어와 상호작용하기 위한 용도

CRUD - Create, Read, Update, Delete - 영속적 저장소(persistent storage)를 위한 네 가지 기본 연산들

CSRF - Cross-Site Request Forgery - 보안 취약점 중 하나. 인증되지 않은 명령들이 신뢰받는 사용자로부터 전송되도록 한다

CSS - Cascading Style Sheets - 스타일 시트 언어 중 하나. HTML 문서들의 표현을 기술하기 위해 사용

D

DLL - Dynamic Link Library - Windows 라이브러리 파일 유형 중 하나. 여기에 담긴 코드와 데이터는 여러 프로그램들에서 사용될 수 있다

DOM - Document Object Model - HTML 및 XML 문서용 프로그래밍 인터페이스 중 하나. 그 페이지의 구조를 오브젝트들의 트리로 표현한다

DRY - Don't Repeat Yourself - 소프트웨어 개발 원칙 중 하나. 목표는 반복을 줄이는 것이다

E

ERB - Embedded Ruby - 템플릿 체계 중 하나. Ruby 코드를 텍스트 문서들에 삽입한다

F

FastCGI - Fast Common Gateway Interface - CGI의 개선된 버전 중 하나. 프로세스들을 영속적으로 유지해 더 좋은 성능을 낸다

G

GUI - Graphical User Interface - 시각적 인터페이스. 사용자들이 소프트웨어와 상호작용할 수 있게 한다

GUID - Globally Unique Identifier - 128-비트 숫자. 정보를 고유하게 식별하기 위해 사용된다

H

HMAC - Hash-based Message Authentication Code - 메시지 인증용 메커니즘 중 하나. 암호학적 해시 함수들을 사용한다

HTML - HyperText Markup Language - 웹 페이지 생성용 표준 마크업 언어

HTMX - HTML Extensions - JavaScript 라이브러리 중 하나. 현대적인 브라우저 기능들을 여러분이 HTML로부터 직접 접근할 수 있다

HTTP - HyperText Transfer Protocol - 웹 을 통한 데이터 전송을 위해 사용되는 프로토콜

HTTPS - HyperText Transfer Protocol Secure - HTTP의 보안 버전. 암호화를 사용한다

I

ID - Identifier - 엘리먼트나 오브젝트를 가리키는 고유한 참조

IDE - Integrated Development Environment - 소프트웨어 애플리케이션 중 하나. 광범위한 종합 기능들을 소프트웨어 개발을 위해 제공한다.

IIS - Internet Information Services - Microsoft의 웹 서버 소프트웨어

IP - Internet Protocol - 네트워크 간 데이터를 라우팅(routing)하기 위한 가장 중요한 통신 프로토콜

ISAPI - Internet Server Application Programming Interface - Microsoft의 API. IIS 위에 웹 애플리케이션들을 만들 수 있도록 한다

J

JS - JavaScript - 프로그래밍 언어 중 하나. 웹 개발에 흔하게 사용된다

JSON - JavaScript Object Notation - 경량 데이터 교환 형식 중 하나

M

MVC - Model-View-Controller - 소프트웨어 아키텍처 패턴 중 하나. 애플리케이션 로직을 분리해 상호 연결된 그 세 가지 구성요소들 안에 나누어 넣는다

N

NGINX - Engine X (발음은 "엔진-엑스") - 고성능 웹 서버이자 리버스 프록시

NPM - Node Package Manager - JavaScript용 패키지 관리자

R

RAD - Rapid Application Development - 소프트웨어 개발 방법론 중 하나. 빠른 프로토타이핑과 반복적인 개발을 강조한다</

RTL - Run-Time Library - 미리 작성된 코드들의 모음. 프로그램들은 그 코드들을 실행 중에 사용할 수 있다

RTTI - Run-Time Type Information - 메커니즘 중 하나. 오브젝트의 데이터 타입에 대한 정보를 런타임에 노출한다

S

SOAP - Simple Object Access Protocol - 프로토콜 중 하나. 구조화된 정보를 웹 서비스들 간에 서로 교환하는 용도

SQL - Structured Query Language - 관계형 데이터베이스들을 관리하고 질의하기 위해 사용되는 언어

SSE - Server-Sent Events - 서버 푸시 기술 중 하나. 서버 하나가 클라이언트들에게 자동 업데이트들을 보낼 수 있도록 한다

SSL - Secure Sockets Layer - 암호화 프로토콜 중 하나. 보안 통신용 (현재는 TLS로 대체됨)

T

TLS - Transport Layer Security - 암호화 프로토콜 중 하나. 네트워크 상의 보안 통신을 제공 (SSL의 후속)

U

UI - User Interface - 사용자들이 컴퓨터 시스템과 상호작용하는 수단

URI - Uniform Resource Identifier - 리소스를 식별하는 문자열

URL - Uniform Resource Locator - 웹 리소스를 가리키는 참조 중 하나. 그것의 위치를 명시한다

UTF - Unicode Transformation Format - 문자 인코딩 방식 중 하나(예: UTF-8, UTF-16)

X

지금 **RAD Studio** 를 써보세요!

얼마나 쉽게 멀티 플랫폼을 대상으로 네이티브 앱을 구축할 수 있는지 보세요! 그저 단 하나의 코드 기반으로 할 수 있습니다!

www.embarcadero.com

