IDG
STRATEGIC
MARKETING SERVICES

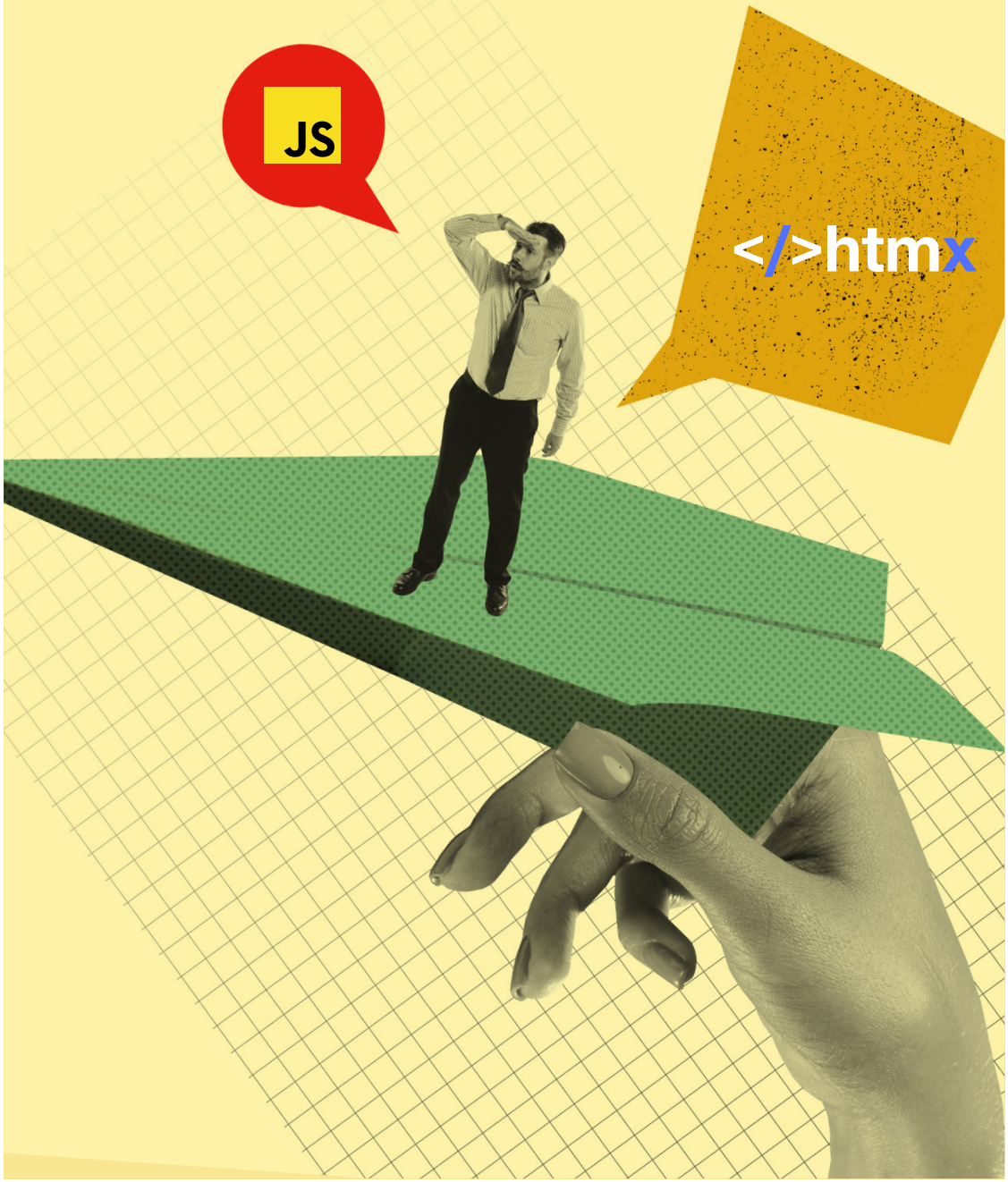
IT WORLD CIO

IT WORLD HOW TO

자바스크립트 없이 구현하는 동적 HTML

HTMX의 이해와 실전 가이드

> "더 단순했던 웹 시대로의 복귀" HTMX의 이해와 기본 활용법
> HTMX와 번을 활용한 풀 스택 웹 개발 1부. 엘리시아와 몽고DB
> HTMX와 번을 활용한 풀 스택 웹 개발 2부. 퍼그 템플릿
> "복잡성은 죄악이다" HTMX 창시자 카스 그로스 인터뷰



"더 단순했던 웹 시대로의 복귀"
HTMX의 이해와 기본 활용법

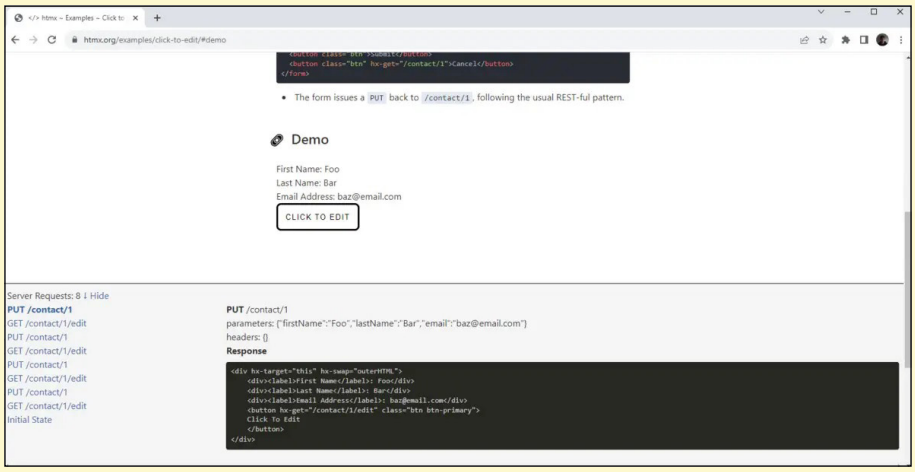
Matthew Tyson | InfoWorld

HTMX는 자바스크립트에 의존하지 않고 확장된 HTML 구문을 사용해 마크업에서 바로 HTTP 상호작용을 제공한다. 향후 웹 프론트 엔드의 작동 방식 전체에 영향을 줄 수 있는 매우 흥미로운 기술이다.

HTMX는 꽤 오래전에 나왔지만, 개발 프로젝트는 다소 뜸했다. 최근 깃허브 액셀러레이터(GitHub Accelerator)에 발탁된 만큼 분위기가 바뀔 가능성이 높다. 기본적인 개념은 상용구(boilerplate text) 자바스크립트-HTML 상호작용이 필요한 일반적인 상황에서 자바스크립트 없이 HTML 구문을 사용하는 것이다. 많은 상호작용이 HTMX를 통해 선언적이 된다. 여기까지만 들어도 귀가 솔깃하다. 웹 개발자라면 누구나 너무 많은 상용구에 고통받고 있기 때문이다. HTMX를 만든 카슨 그로스는 "하이퍼텍스트로 HTML을 완성해 더 진보적인 현대 웹 애플리케이션을 만드는 경쟁력 있는 대안이 되도록 표현력을 높이는 것이 목표다"라고 말했다.

HTMX에 대해 대략적인 감을 잡기 위해 데모를 살펴보자. 기본적으로 버튼을 클릭해 사용자 객체의 필드를 편집하는 예제다. 데이터는 백엔드 엔드포인트로 들어간다(PUT). <화면 1>을 보면 Show를 클릭한 후 하단 프레임의 네트워크 상호작용을 볼 수 있다.

일반적으로 어느 프레임워크를 사용하든 이런 작업을 하려면 자바스크립트가 필요하다. 하지만 HTMX는 다음과 같이 상호작용을 표시 UI를 위한 마크업과 편집 UI를 위한 마크업 등 2개의 마크업 덩어리로 바꾼다.



화면 1 | HTMX 폼 데모

```
<div hx-target="this" hx-swap="outerHTML">
    <div><label>First Name</label>: Joe</div>
    <div><label>Last Name</label>: Blow</div>
    <div><label>Email</label>: joe@blow.com</div>
    <button hx-get="/contact/1/edit" class="btn btn-primary">
    Click To Edit
    </button>
</div>
<!-- The edit: -->
<form hx-put="/contact/1" hx-target="this" hx-swap="outerHTML">
  <div>
    <label>First Name</label>
    <input type="text" name="firstName" value="Joe">
  </div>
  <div class="form-group">
    <label>Last Name</label>
    <input type="text" name="lastName" value="Blow">
  </div>
  <div class="form-group">
    <label>Email Address</label>
    <input type="email" name="email" value="joe@blow.com">
  </div>
  <button class="btn">Submit</button>
  <button class="btn" hx-get="/contact/1">Cancel</button>
</form>
```

소스를 보면 어떤 동작이 이뤄지는지 알 수 있다. hx-swap 속성은 편집되기 전 div 부분 HTML을 제공하고, outerHTML은 이것이 프레임워크 내부의 동적 콘텐츠와 어떻게 관련되는지 알려준다.

HTMX는 자바스크립트를 전혀 사용하지 않고 이 "swap"과 그 이후의 PUT을 어떻게 구현했을까? 답은 쉽다. 편집 마크업에 HTML의 서버 측 렌더링을 사용하고, 양식 변환 작업을 프레임워크로 추상화하는 것이다. 배후에서는 여전히 자바스크립트가 실행되지만, 우리는 전체 페이지 대신 세그먼트만 로드해 Ajax 요청을 보내는 더 세분화된 HTML 구문을 볼 뿐이다. 이는 DRY 원칙의 흥미로운 실제 예시이자, 리액트(React)와의 차이이기도 하다. 리액트를 사용해도 한 양식에서 다른 양식으로 정보를 섞을 때 상용구 코드를 일정 정도 써야 한다. 반면 HTMX는 이를 완전히 제거한 것은 아니지만, 상당수 작업을 서버로 옮긴다.

서버 측 HTMX

이제 서버 측 예제를 보자. 그로스는 "HTMX는 백엔드를 가리지 않는다. HTML만 사용하면 어느 백엔드를 쓰든 관계없다"라고 말했다. 이것이 가능한 것은 HTMX를 사용하는 서버 측 기술이 다양하기 때문이다. 어떤 식으로

작동하는지 알기 위해 익스프레스(Express)와 함께 퍼그(Pug) HTML 템플릿 엔진을 사용하는 to-do 예제를 보자. 이 예제는 일반적인 to-do 애플리케이션이다.

우선, 기존 to-do 항목은 다음 명령을 사용한 익스프레스의 출력이다.

```
res.render('index', { todos: filteredTodos, filter, itemsLeft: getItemsLeft() });
```

이 명령은 메모리 내 to-do 컬렉션을 사용하고 HTMX 상호작용을 이끄는 특수한 hx- 속성을 포함한다는 점을 제외하면, 일반적인 형식의 퍼그 템플릿을 사용해 렌더링한다. 예를 들어 새 to-do를 POST하는 양식은 다음과 같다.

```
form(hx-post="/todos", hx-target="#todo-list", hx-swap="afterbegin", _="on
htmx:afterOnLoad set #txtTodo.value to ''")
  input#txtTodo.new-todo(name="todo",placeholder='What needs to be done?',
autofocus='')
```

afterbegin 속성이 새 콘텐츠를 목록의 맞는 위치에 어떻게 넣는지는 여기서 볼 수 있다. on htmx 스크립트는 일종의 간소화된 스크립팅 언어인 하이퍼스크립트다. HTMX와 함께 자주 쓰이지만, HTMX의 일부이거나 HTMX를 사용하기 위해 필수적인 것은 아니다. 여기서 on htmx는 새 to-do가 만들어진 후 입력 양식의 값 설정을 처리하는 역할을 한다.

다음은 to-do 편집을 위한 퍼그 템플릿이다.

```
form(hx-post="/todos/update/" + todo.id)
  input.edit(type="text", name="name",value=todo.name)
```

편집된 to-do에 대한 JSON을 어디로 보낼지 표현하는 데는 hx-post 속성을 사용한다. 이 예제의 핵심은 다양한 상호작용을 위해 프론트 엔드에서 필요로 하는 화면의 여러 부분을 채우는 작업을 누가 담당하느냐다. 적절한 크기의 HTML(HTMX 태그로 데코레이션됨)을 제공하는 작업을 서버가 담당한다. HTMX 클라이언트는 속성에 따라 각 요소를 적절한 위치에 배치하고, 서비스에서 사용할 수 있도록 필요한 데이터를 전송한다.

데이터 수신을 담당하는 엔드포인트는 일반적인 엔드포인트처럼 작동하는데, 차이점은 필요한 HTMX가 응답해야 한다는 것이다. 다음에서 익스프레스 서버가 새 to-do를 생성하기 위해 POST를 어떻게 처리하는지 볼 수 있다.

```
app.post('/todos', (req, res) => {
  const { todo } = req.body;
  const newTodo = {
    id: uuid(),
    name: todo,
    done: false
  };
  todos.push(newTodo);
  let template = pug.compileFile('views/includes/todo-item.pug');
  let markup = template({ todo: newTodo});
  template = pug.compileFile('views/includes/item-count.pug');
  markup  += template({ itemsLeft: getItemsLeft()});
  res.send(markup);
});
```

이는 일반적인 POST 본문 핸들러로, 양식 데이터에서 값을 가져와 새 비즈니스 객체를 생성한다(newTodo). 그런 다음 이 값을 사용해 퍼그 템플릿을 채우고 이를 클라이언트로 돌려보내 프론트 엔드의 to-do 목록에 삽입한다. 다른 서버 측 기술도 있다. 예를 들어 자바 환경에서는 타임리프(Thymeleaf)-스프링 부트(Spring Boot) 조합과 함께 HTMX를 사용하고, 파이썬 환경에서는 장고(Django)-스프링 부트 조합으로 쓸 수 있다.

HTMX를 사용한 클라이언트 측 템플릿

HTMX가 유용한 다른 상황은 클라이언트 측 템플릿을 사용하는 경우다. 클라이언트에서 실행되고 서버로부터 JSON을 받아 마크업 해석을 수행한다. 필자는 그로스에게 RESTful 서비스를 JSON과 함께 사용하는 방법에 관해 물었는데, 클라이언트 측 템플릿을 사용하면 가능하지만 REST에 대한 일반적인 오해에 주의해야 한다고 말했다. 그렇다면 역으로, 기본 인코딩 양식 대신 JSON을 서버에 제출하려면 어떻게 해야 할까? 이때는 확장 기능인 JSON-ENC를 사용하면 된다.

매력적인 웹 인터렉션 접근법

HTMX는 여러 가지 점에서 흥미롭다. 앞으로 HTMX가 점점 성숙해지면서 지금과 같은 방식으로 작동하지 않을 가능성이 있지만, 이 개념의 유익한 영향력은 이미 충분히 입증됐다. 가장 매력적인 점은 일반적으로 fatch() 또는 이와 비슷한 것을 사용하는 매우 보편적인 Ajax 스타일의 온갖 다양한 요청을 HTML 속성 하나로 처리하는 것이다. 더 단순하고 깔끔하며 모든 요소를 한곳에 둘 수 있는 데다가, 마크업을 매우 명확하게 사용할 수 있다.

서버 측 마크업 생성도 마찬가지다. 사실 개발자는 이미 이런 목적으로 JSON을 다루는 데 익숙하다. 마크업을 가져오는 것은 클라이언트 생성에서 한 단계를 추가하는 것일 뿐이다. 그동안 수많은 서버 측 기술이 등장했고, HTML, 자바스크립트, CSS라는 강력한 3요소를 뒤로 잘 숨기는 것처럼 보였지만 결국은 모두 실패했다. 반면 HTMX는 다를 수 있다. 전체 판이 흔들리고 있다.

필자는 HTMX가 대규모 소프트웨어 프로젝트에서 어떻게 작동할지 상상해 봤다. 전체적인 복잡성을 낮춰줄까? 그로스는 이런 복잡성에 대해 나름의 생각을 갖고 있고, HTMX의 설계를 보면 이것이 반영됐음을 알 수 있다. HTMX는 웹 애플리케이션을 위한 상태 메커니즘으로, 하이퍼텍스트로 돌아가 작업을 단순화하는 데 목표를 둔다. 이 예제는 이런 개념이 실제로 어떻게 작동하는지 보여준다. JSON을 프로토콜로 사용한다는 것은 곧 클라이언트를 더 스마트하고 만들고, 아키텍처를 덜 자기 설명적으로 만드는 것을 의미한다.

만약 이런 이상이 실제로 구현된다면, 기반 언어인 HTML을 확장해 Ajax와 같은 최신 요구사항을 실제로 처리함으로써 내재한 복잡성을 피하고 웹 전체가 더 단순한 시대로 돌아갈 수 있다. 마크업은 다시 한번 중앙 데이터 설명자가 되어 UI는 물론 오가는 데이터도 충분히 처리할 수 있게 될 것이다.

HTMX와 번을 활용한 풀 스택 웹 개발
1부. 엘리시아와 몽고DB

Matthew Tyson | InfoWorld

HTMX를 앞에, 번(Bun)을 뒤에 두고 엘리시아(Elysia)와 몽고DB로 연결하면 쾌적하게 웹 앱을 개발할 수 있는 민첩한 기술 스택이 만들어진다. 번과 HTMX는 현재 소프트웨어 분야에서 가장 흥미로운 요소다. 번은 매우 빠른 올인원 서버 측 자바스크립트 플랫폼이고, HTMX는 단순하면서 강력한 인터페이스를 만드는 HTML 확장이다. 이 2가지 툴을 함께 사용해 데이터 저장에 몽고DB를, HTTP 서버로 엘리시아를 사용하는 풀 스택 애플리케이션을 만들어보자.

기술 스택

여기서는 번, HTMX, 엘리시아, 몽고DB 등 기술 스택의 4가지 주요 구성요소가 어떻게 상호작용을 하는지에 초점을 둔다. 이 스택은 쉽게 구성하고 민첩하게 변경할 수 있는 기민한 환경을 제공한다.

- 번은 자바스크립트 런타임, 번들러, 패키지 관리자, 테스트 러너다.
- 엘리시아는 익스프레스와 비슷하지만 번용으로 만들어진 고성능 HTTP 서버다.
- HTMX는 HTML에 상호작용성을 추가하는 새로운 접근 방법을 제공한다.
- 몽고DB는 대표적인 NoSQL 문서 지향 데이터 저장소다.

설치와 설정

일단 Bun.js를 설치해야 한다. 설치 방법은 간단하다. 개발 시스템에서 번과 함께 몽고DB를 서비스로 실행한다. 몽고DB 설치, 설정 방법은 여기를 참고한다. 패키지를 설치하고 명령줄에서 bun -v와 mongod -version 명령이 모두 정상 작동하는 것을 확인한다.

이제 본격적으로 새 프로젝트를 시작해 보자.

```
$ bun create elysia iw-beh
```

이렇게 하면 번은 엘리시아 템플릿을 사용해 새 프로젝트를 만든다. 이 템플릿을 이용하면 create 명령으로 프로젝트를 신속하게 시작할 수 있다. 번은 아무 구성 없이 노드와 함께 작동하지만 config 설정을 해 두는 것이 좋다. 번 템플릿에 대한 더 자세한 내용은 여기서 볼 수 있다.

이제 $ cd iw-beh 명령으로 새 디렉터리로 이동한다. $ bun run src/index.js. 명령을 사용해 현재 상태 그대로 프로젝트를 실행한다. 마지막으로 입력할 명령은 src/index.js이다. 번이 파일을 실행하도록 지시한다. src/index.js 파일에는 엘리시아 서버를 시작하는 간단한 코드가 포함된다.

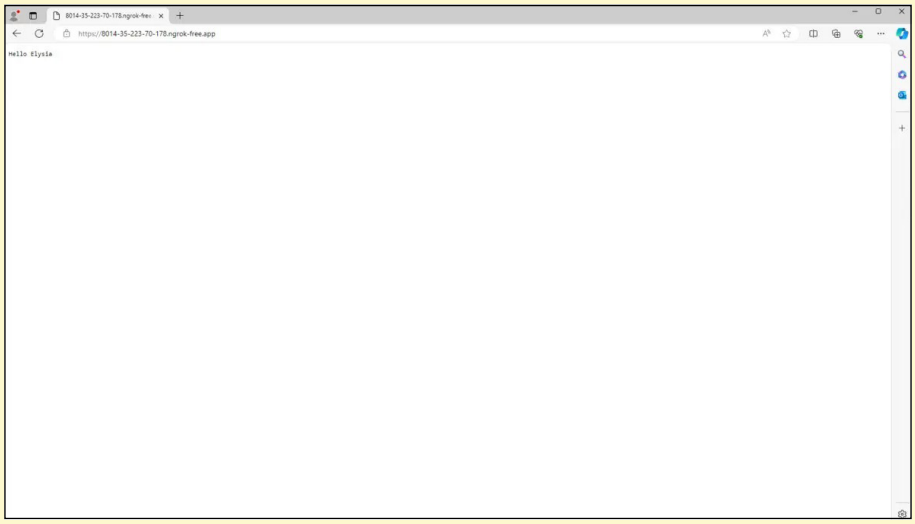
```
import { Elysia } from "elysia";

const app = new Elysia()
  .get("/", () => "Hello Elysia")
  .listen(3000);

console.log(
  `🦊 Elysia is running at ${app.server?.hostname}:${app.server?.port}`
);
```

파일에서 엘리시아를 가져온 후 포트 3000에서 수신하고 루트에 하나의 GET 엔드포인트가 있는 새 서버를 인스턴스화한다. 이 엔드포인트는 "Hello Elysia." 텍스트 문자열을 반환한다. 이 모든 작동 방식은 익스프레스(Express)와 비슷하다. localhost:3000을 방문하면 <화면 1>과 같은 간단한 인사말이 표시된다.

이제 엘리시아가 실행 중이니 static 플러그인을 추가해 보자. 엘리시아에는 다양한 플러그인이 있다. 여기서는 몇 가지 HTML만 다루므로 static 플러그인으로 충분하다.

```
$ bun add @elysiajs/static
```

이제 static 플러그인을 실행하는 엘리시아 서버가 iw-beh/public 디렉터리 내의 모든 항목을 서비스한다. 이 디



화면 1 | localhost:3000을 방문하면 간단한 인사말이 표시된다.

렉터리에 간단한 HTML 파일 하나를 두고 localhost:3000/ public을 방문하면 콘텐츠가 표시된다.

HTMX의 마법

index.html에 HTMX 페이지를 추가해 보자. 다음은 HTMX 홈페이지에서 가져온 간단한 예제다.

```
<script src="https://unpkg.com/htmx.org@1.9.10"></script>

<button hx-post="/clicked"
    hx-trigger="click"
    hx-target="#parent-div"
    hx-swap="outerHTML">
    Click Me!
</button>
```

이 페이지는 버튼 하나를 표시한다. 클릭하면 버튼이 서버에 /clicked 엔드포인트를 호출하고, 응답에 있는 항목으로 버튼이 대체된다. 아직 응답에는 아무것도 없으므로 어떤 작업도 실행되지 않는다. 모든 요소는 여전히 HTML이다. HTMX를 사용하면 자바스크립트 없이 API 호출을 수행하고 세분화된 DOM 변경을 처리한다. 실제로 이 작업은 방금 가져온 htmx.org 라이브러리의 자바스크립트에 의해 수행되지만 우리가 신경 쓸 필요가 없다는 점이 중요하다.

HTMX는 다음과 같은 단 3가지 요소 속성을 사용해 이런 작업을 수행하는 HTML 구문을 제공한다.

- hx-post는 AJAX 요청에 대해 트리거될 때 포스트를 제출한다.
- hx-target은 hx-post에 AJAX 요청을 실행하는 이벤트가 무엇인지 알려준다.
- hx-swap은 AJAX 이벤트가 발생할 때 무엇을 할지 알려준다. 여기서는 현재 요소를 응답으로 대체한다.

엘리시아와 몽고DB

이제 몽고DB에 뭔가를 쓰는 엔드포인트를 엘리시아에 만들어 보자. 먼저 번용 몽고DB 드라이버를 추가한다.

```
bun add mongodb
```

그다음 src.index.ts를 다음과 같이 수정한다.

```
import { Elysia } from "elysia";
import { staticPlugin } from '@elysiajs/static';
const { MongoClient } = require('mongodb');
```

```
const app = new Elysia()
  .get("/", () => "Hello Elysia")
  .get("/db", async () => {

    const url = "mongodb://127.0.0.1:27017/quote?directConnection=true&serverSelect
ionTimeoutMS=2000&appName=mongosh+1.8.0";

    const client = new MongoClient(url, { useUnifiedTopology: true });
try {

    await client.connect();

    const database = client.db('quote');
    const collection = database.collection('quotes');

    const stringData = "Thought is the grandchild of ignorance.";

    const result = await collection.insertOne({"quote":stringData});
    console.log(`String inserted with ID: ${result.insertedId}`);

  } catch (error) {
    console.error(error);
  } finally {
    await client.close();
  }
         return "OK";
  })
  .use(staticPlugin())
  .listen(3000);

console.log(
  `🦊 Elysia is running at ${app.server?.hostname}:${app.server?.port}`
);
```

이 코드에서는 몽고DB와 통신하는 /db 엔드포인트를 추가했다. 이번 예제에서는 위인의 유명한 격언을 모아 놓은 데이터베이스를 사용한다. localhost:3000/db로 가서 직접 테스트해 데이터가 몽고DB에 있는지 확인할 수 있다.

```
$ mongosh

test> use quote
```

```
switched to db quote
quote> db.quotes.find()
[
  {
    _id: ObjectId("65ba936fd59e9c265cc8c092"),
    quote: 'Thought is the grandchild of ignorance.',
    author: 'Swami Venkatesananda'
  }
]
```

HTMX 테이블 만들기

이제 프론트엔드에서 데이터베이스에 연결했으니, 테이블을 만들어 기존 격언을 출력해 보자. 간단한 테스트를 위해 다음과 같이 서버에 엔드포인트를 추가한다.

```
.get("/quotes", async () => {
    const data = [
      { name: 'Alice' },
      { name: 'Bob' },
    ];
    // Build the HTML list structure
  let html = '<ul>';
  for (const item of data) {
    html += `<li>${item.name}</li>`;
  }
  html += '</ul>';

    return html
  })
```

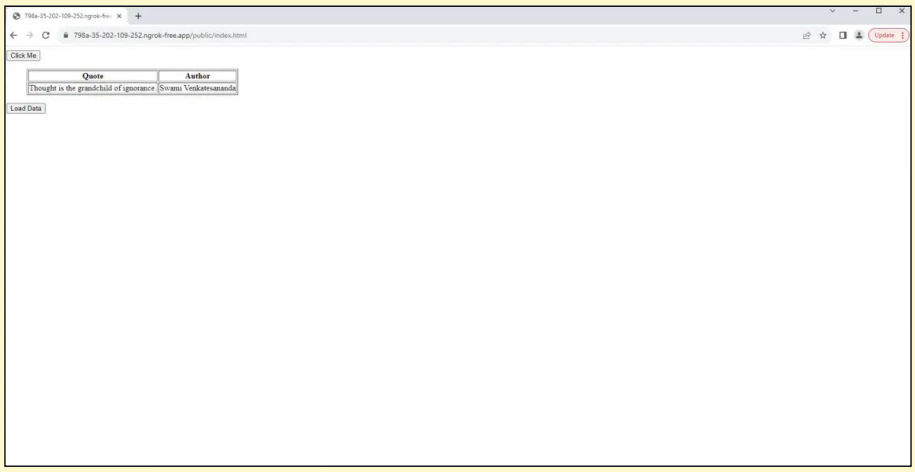
그런 다음 index.html에서 이 엔드포인트를 사용한다.

```
<ul id="data-list"></ul>
<button hx-get="/quotes" hx-target="#data-list">Load Data</button>
```

이제 /public/index.html을 로드하고 버튼을 클릭하면 서버에서 보낸 목록이 표시된다. 엔드포인트에서 HTML을 발행하는 것은 일반적인 JSON 패턴과 다르다는 것을 알 수 있다. 이는 설계상 RESTful 원칙을 따르는 것이다. HTMX에는 JSON 엔드포인트와 함께 사용하는 플러그인이 있지만 이 방법이 더 자연스럽다. 여기서는 엔드포인트에서 HTML을 수동으로 만들지만 실제 애플리케이션이라면 관리가 더 용이하도록 일종의 자바스크립트-HTML 템플릿 프레임워크를 사용하는 것이 좋다.

이제 데이터베이스에서 데이터를 불러올 수 있다.

```
.get("/quotes", async () => {
const url = "mongodb://127.0.0.1:27017/quote?directConnection=true&serverSelectionT
imeoutMS=2000&appName=mongosh+1.8.0";

      const client = new MongoClient(url, { useUnifiedTopology: true });
    try {
      await client.connect();

      const database = client.db('quote');
      const collection = database.collection('quotes');

      const quotes = await collection.find().toArray();

      // Build the HTML table structure
      let html = '<table border="1">';
      html += '<tr><th>Quote</th><th>Author</th></tr>';
      for (const quote of quotes) {
        html += `<tr><td>${quote.quote}</td><td>${quote.author}</td></tr>`;
      }
      html += '</table>';

      return html;
    } catch (error) {
      console.error(error);
      return "Error fetching quotes";
    } finally {
```



화면 2 | 데이터를 불러와 HTML 테이블로 반환

```
      await client.close();
    }

  })
```

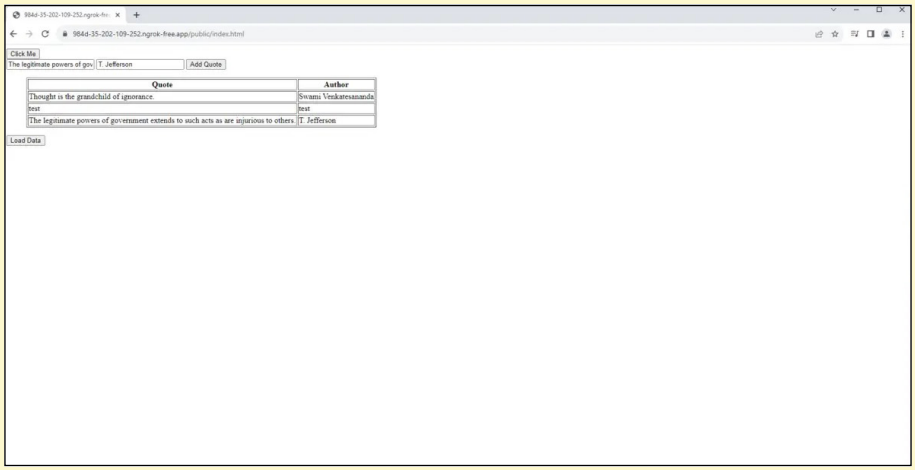
이 엔드포인트에서 데이터베이스의 모든 기존 격언 데이터를 불러와 간단한 HTML 테이블로 반환한다. 최종적으로 <화면 2>처럼 실행된다. 앞서 /db 엔드포인트를 다룰 때 삽입한 행이 보인다.

이제 새 격언을 만드는 기능을 추가해 보자. 다음은 백엔드 코드다(src/index.ts).

```
app.post("/add-quote", async (req) => {
    const url = "mongodb://127.0.0.1:27017/quote?directConnection=true&serverSelect
ionTimeoutMS=2000&appName=mongosh+1.8.0";

    try {
        const client = new MongoClient(url, { useUnifiedTopology: true });
        await client.connect();

        const database = client.db('quote');
        const collection = database.collection('quotes');

        const quote = req.body.quote;
        const author = req.body.author;

        await collection.insertOne({ quote, author });

        return "Quote added successfully";
    } catch (error) {
        console.error(error);
        return "Error adding quote";
    } finally {
        await client.close();
    }
})
```

프론트엔드는 다음과 같다(public/index.html).

```
<form hx-post="/add-quote">
    <input type="text" name="quote" placeholder="Enter quote">
    <input type="text" name="author" placeholder="Enter author">
```



화면 3 | 새 격언을 만드는 기능 추가

```
<button type="submit">Add Quote</button>
```

저자와 격언을 입력하고 Add Quote를 누르면 데이터베이스에 추가된다. Load Data를 클릭하면 <화면 3>처럼 갱신된 목록을 볼 수 있다.

정리하면 HTMX는 애플리케이션을 위한 서버와 클라이언트에서 최소한의 작업만 하고 있음을 알 수 있다. 특히 간소화된 부분은 양식 제출이다. 양식에서 데이터를 꺼내 JSON으로 변환하고 fetch() 또는 그와 비슷한 것을 사용해 제출하는 모든 작업을 hx-post 속성이 대체한다.

HTMX가 만능은 아니다

단, HTMX를 사용한다 해도 구조가 복잡해지면 어느 시점부터는 클라이언트에서 자바스크립트를 사용해야 한다. 예를 들어 인라인 행 편집 작업이라면 테이블에 직접 새 행 삽입하기와 같이 보통 자바스크립트로 할 만한 일부 작업은 HTMX 스와핑을 통해 할 수 있다. 그 이상의 복잡한 작업이 필요하면, 자바스크립트를 쓰면 된다.

프로그래밍 사고방식 측면에서 가장 큰 변화는 서버에서 HTMX를 생성하는 데 있다. 여러 하이엔드 HTML 또는 자바스크립트 템플릿 엔진을 사용해 이 작업을 더 쉽게 할 수 있다. 사실 HTMX에 익숙해지면 간단한 작업이다. 기본적으로 전체 기술 스택에서 JSON 변환 계층을 없앤 걸로 보면 된다. 필자의 깃허브 리포지토리에서 이 글에 사용된 코드를 내려받을 수 있다.

HTMX와 번을 활용한 풀 스택 웹 개발
2부. 퍼그 템플릿

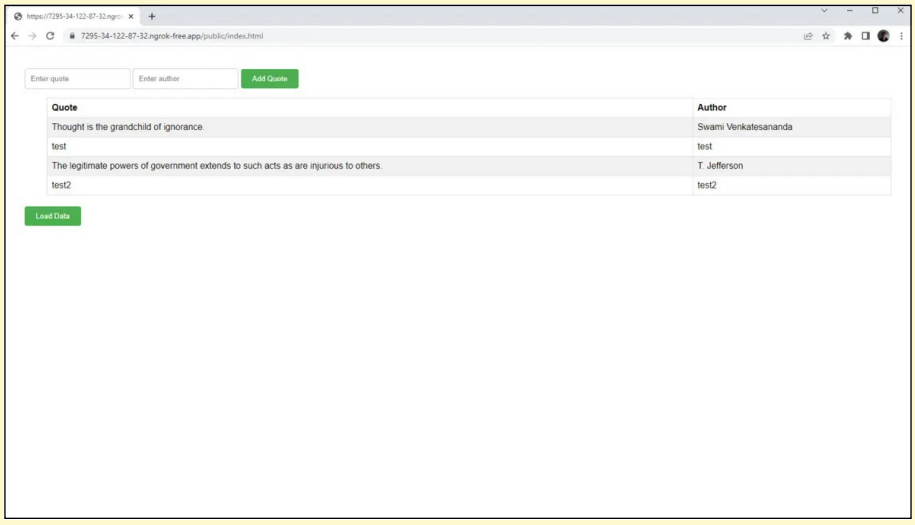
Matthew Tyson | InfoWorld

1부에서는 번, HTMX, 엘리시아, 몽고DB를 사용해 웹 개발 스택을 설정하고 간단한 예제 애플리케이션을 만들었다. 2부에서는 이 예제 애플리케이션의 데이터 액세스 계층을 추상화하고 더 복잡한 HTMX 상호작용을 추가해 보자. 또한 기술 스택에 퍼그(Pug)라는 새 구성 요소를 통합한다. 퍼그는 HTMX와 잘 맞는 인기 있는 자바스크립트 템플릿 엔진으로, DOM 상호작용 구성에 특히 유용하다.

1부에서 만든 예제 애플리케이션은 양식 하나, 테이블 하나로 구성된다. 사용자가 격언과 저자를 입력하면 애플리케이션의 사용자 인터페이스를 통해 검색, 표시된다. 너무 단순하므로, 조금 세련되게 보이도록 약간의 CSS를 추가한 것이 <화면 1>이다.

수정된 인터페이스의 프론트엔드 코드는 다음과 같다.

```
<script src="https://unpkg.com/htmx.org@1.9.10"></script>
  <form hx-post="/add-quote" hx-swap-oop="beforeend:#data-list" hx-trigger="every
time">
    <input type="text" name="quote" placeholder="Enter quote">
    <input type="text" name="author" placeholder="Enter author">
    <button type="submit">Add Quote</button>
```



화면 1 | CSS를 추가한 예제 애플리케이션

```
</form>
  <ul id="data-list"></ul>
  <button hx-get="/quotes" hx-target="#data-list">Load Data</button>
```

양식을 제출하고 데이터를 테이블로 로드하는 과정을 HTMX가 담당한다. 또한 여기서는 데이터베이스 연결이 공유되도록 애플리케이션의 백엔드도 수정했다. src/index.ts에서 바뀐 부분은 다음과 같다.

```
import { Elysia } from "elysia";
import { staticPlugin } from '@elysiajs/static';
const { MongoClient } = require('mongodb');

// Database connection details
const url = "mongodb://127.0.0.1:27017/quote?directConnection=true&serverSelectionT
imeoutMS=2000&appName=mongosh+1.8.0";
const dbName = "quote";
const collectionName = "quotes";
let client = new MongoClient(url, { useUnifiedTopology: true });

// Connect to the database (called only once)
async function connectToDatabase() {
  try {
    await client.connect();
  } catch (error) {
    console.error(error);
    throw error; // Re-throw the error to indicate connection failure
  }
  return { client, collection:  client.db(dbName).collection(collectionName) };
}

// Close the database connection
async function closeDatabaseConnection(client) {
  await client.close();
}
```

코드를 보면 데이터베이스 URL을 기본 몽고DB 로컬 호스트 주소로 정의하고 데이터베이스와 컬렉션 이름을 정의한다. 그런 다음 비동기 함수인 connectToDatabase()를 사용해 클라이언트를 연결하고 이렇게 컬렉션에 연결된 상태로 반환한다. 이후에는 데이터베이스에 액세스할 때마다 이 메서드를 호출하고, 작업이 완료되면 client.close()를 불러온다.

데이터베이스 연결 사용

서버 엔드포인트가 이 데이터베이스 지원을 어떻게 사용하는지 살펴보자. 여기서는 테이블을 구동하는 /quotes

엔드포인트를 중심으로 분석한다.

```
// Close the database connection
async function closeDatabaseConnection(client) {
  await client.close();
}

async function getAllQuotes(collection) {
  try {
    const quotes = await collection.find().toArray();

    // Build the HTML table structure
    let html = '<table border="1">';
    html += '<tr><th>Quote</th><th>Author</th></tr>';
    for (const quote of quotes) {
      html += `<tr><td>${quote.quote}</td><td>${quote.author}</td></tr>`;
    }
    html += '</table>';

    return html;
  } catch (error) {
    console.error("Error fetching quotes", error);
    throw error; // Re-throw the error for proper handling
  }
}

// Main application logic
const app = new Elysia()
  .get("/", () => "Hello Elysia")
  .get("/quotes", async () => {
    try {
      const { client, collection } = await connectToDatabase();
      const quotes = await getAllQuotes(collection);
      await closeDatabaseConnection(client);
      return quotes;
    } catch (error) {
      console.error(error);
      return "Error fetching quotes";
    }
  })
  .use(staticPlugin())
  .listen(3000);
```

```
console.log(
  ` Elysia is running at ${app.server?.hostname}:${app.server?.port}`
);
```

코드는 격언 데이터를 가져오기 위해 호출할 수 있는 백엔드 /quotes GET 엔드포인트다. 엔드포인트는 getAllQuotes() 메서드를 호출하고 이 메서드는 connectToDatabase()의 컬렉션을 사용해 격언과 저자 배열을 가져온 다음 행을 위한 HTMX를 생성한다. 마지막으로, HTMX로 행이 저장된 응답을 전송하면 테이블에 행이 삽입된다.

퍼그 템플릿 엔진 추가

행 HTMX를 수동으로 생성하면 번거롭기도 하지만, 무엇보다 오류가 발생하기 쉽다. 이때 템플릿 엔진을 사용해 깔끔한 구문으로 HTMX 구조를 정의할 수 있다.

이런 HTML 템플릿 엔진 중 자바스크립트용으로 가장 인기 있는 것이 퍼그(Pug)다. 퍼그를 사용하면 자바스크립트 코드로 인라인(inline)하는 것에 비해 서버에서 훨씬 더 쉽게 뷰를 만들 수 있고 확장성도 더 높다. 기본 개념은 데이터 객체를 가져와 템플릿으로 전달하면 템플릿이 이 데이터를 적용해 HTML을 출력하는 것인데, HTML이 아니라 HTMX를 생성하는 것이 특징이다. 이것이 가능한 이유는 HTMX가 기본적으로 '확장된' HTML이기 때문이다.

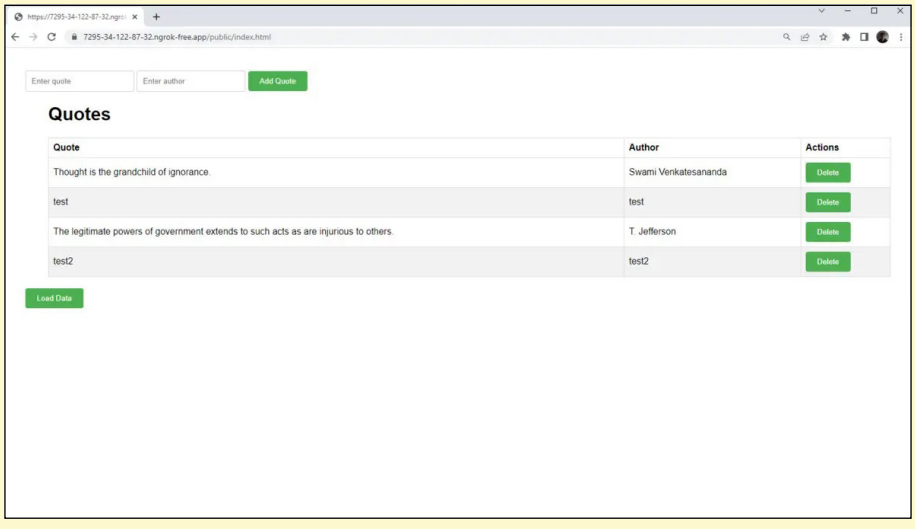
시작하려면 $ bun add pug를 사용해 프로젝트에 퍼그 라이브러리를 추가한다. 완료되면 프로젝트 루트에 /views라는 새 디렉터리를 만들고($ mkdir views) quotes.pug라는 파일을 추가한다.

```
doctype html
h1 Quotes
table
  thead
    tr
      th Quote
      th Author
      th Actions
  tbody
    each quote in quotes
      tr(id=`quote-${quote._id}`)
        td #{quote.quote}
        td #{quote.author}
        td
          button(hx-delete=`/quotes/${quote._id}` hx-trigger="click" hx-
swap="closest tr" hx-confirm="Are you sure?") Delete
          #{quote._id}
```

퍼그는 들여쓰기를 사용해 중첩된 요소를 처리한다. 속성은 괄호 안에 들어간다. 단어 Delete와 같은 일반 텍스트는 그대로 제공된다. 이 모든 기능을 HTML 또는 HTMX로 간결하게 기술할 수 있다. 구문에 대한 자세한 내용은 퍼그 홈페이지를 참고한다.

문자열 안에서는 ${}를 사용해야 하고, #{} 구문을 쓰면 템플릿에 삽입된 모든 데이터 객체를 참조할 수 있다. 리액트(React)와 같은 프레임워크의 토큰 보간과 비슷하다. 기본 개념은 전체적인 HTML/HTMX 구조를 정의한 다음 #{} 및 ${}로 참조되는 변수를 템플릿에 제공하는 것이다. 다음은 getAllQuotes()를 사용하는 서버 /quotes 엔드포인트에 변수를 보내주는 작업이다.

```
import pug from 'pug';
//...
async function getAllQuotes(collection) {
  try {
    const quotes = await collection.find().toArray();

    // Render the Pug template with the fetched quotes
    const html = pug.compileFile('views/quotes.pug')({ quotes });

    return html;
  } catch (error) {
    console.error("Error fetching quotes", error);
    throw error; // Re-throw the error for proper handling
  }
}
```



화면 1 | 퍼그 템플릿을 이용해 출력하기

데이터베이스에서 격언을 가져온 다음 퍼그 템플릿을 컴파일해서 격언을 전달한다. 그러면 퍼그가 HTML과 데이터를 조합하는 일을 알아서 해준다. 전체적인 흐름은 다음과 같다.

- GET /quotes에 요청이 도달한다.
- 몽고DB에서 격언을 검색한다.
- 퍼그 템플릿이 격언을 받는다.
- 퍼그 템플릿이 HTML 또는 HTMX로 이 격언을 렌더링한다.
- 내용이 채워진 HTML 또는 HTMX가 응답으로 전송된다.

결과는 <화면 2>와 같다.

DOM 상호작용 : 행 삭제

이제 Delete 버튼을 작동시켜 보자. 지금까지 살펴본 내용만으로도 삭제 요청을 구현하고 서버와 데이터베이스에서 이를 쉽게 처리할 수 있다. 하지만, 변경 사항을 반영하도록 테이블을 업데이트하는 부분은 어떻게 처리해야 할까? 방법은 여러 가지다. 단순히 전체 테이블을 새로 고치거나, 자바스크립트 혹은 HTMX를 사용해 테이블에서 행을 지울 수 있다. 이상적인 방법은 후자를 사용하고 모든 요소를 HTMX로 유지하는 것이다. views/quotes.pug 템플릿에서 다음과 같이 순수 HTMX를 사용해 행을 삭제할 수 있다.

```
tbody(hx-target="closest tr" hx-swap="outerHTML")
    each quote in quotes
      tr(id=`quote-${quote._id}`)
        td #{quote.quote}
        td #{quote.author}
        td
          button(hx-delete=`/quotes/${quote._id}` hx-trigger="click" hx-confirm="Are
you sure?") Delete
```

여기서 핵심적인 부분은 tbody의 hx-target=”closest tr” 그리고 hx-swap=”outerHTML”이다. x-target은 트리거 요소인 버튼에 가장 인접한 tr을 응답으로 대체하도록 지시한다. hx-swap의 outHTML은 내용만이 아니라 테이블 행 요소 전체를 제거한다. 서버 측에서 빈 본문과 함께 성공(HTTP 200)을 반환하면 HTMX가 행을 삭제한다. 참고로 hx-confirm은 확인 대화 상자를 제공하는 역할이다.

```
async function deleteQuote(collection, quoteId) {
  try {
    const result = await collection.deleteOne({ _id: new ObjectId(quoteId) });
    if (result.deletedCount === 1) {
      return "";
```

```
    } else {
      throw new Error( "Quote not found");
    }
  } catch (error) {
    console.error("Error deleting quote", error);
    throw error; // Re-throw the error for proper handling
  }
}
```

여기서 더 복잡한 DOM 상호작용도 추가할 수 있다. 예를 들면 행 삭제 시나리오에서 간단한 전환 효과를 추가할 수 있는데, HTMX 홈페이지에서 관련 예제를 확인할 수 있다.

애플리케이션 작동 방식의 근본적인 변화

지금까지 1~2부에 걸쳐 번, 엘리시아와 같은 비교적 새로운 기술을 살펴봤다. 하지만 가장 중요한 구성요소는 HTMX다. HTMX는 애플리케이션의 작동 방식을 전통적인 JSON API와 완전히 다르게 바꿔 놓는다.

HTMX를 퍼그와 같은 템플릿 엔진, 몽고DB와 같은 데이터베이스와 결합하면 UI를 생성하고 요청을 처리하는 작업을 더 간편하게 수행할 수 있다. 애플리케이션의 크기가 커지면 템플릿 상속과 같은 퍼그 기능도 요긴하다. 또한, HTMX는 hx-swap, hx-target을 통해 유연한 DOM 상호작용을 구현한다. 더 복잡한 환경이라면 언제든 자바스크립트를 사용하면 된다. 이렇게 스택을 구성한 후 명령줄로 들어가 종속성 추가와 같은 작업을 해보자. 실행할 때마다 번의 속도에 새삼 감탄하게 될 것이다. 여기서 사용한 모든 코드는 필자의 깃허브 리포지토리에서 내려받을 수 있다.

"복잡성은 죄악이다"
HTMX 창시자 카스 그로스 인터뷰

Matthew Tyson | InfoWorld

반응성을 대체하기 위해 노력하고, 자바스크립트와 경쟁하고, 진정한 REST를 복원하고, 웹을 더 개발자 친화적으로 만드는 작업까지, 모두 혼자 하고 있는 대학 교수가 있다. 바로 카슨 그로스다. 그로스는 HTMX와 하이퍼스크립트의 창시자이며, 몬타나 주립대학 소프트웨어 엔지니어링 교수이자 하이퍼미디어 시스템(Hypermedia Systems)의 공동 저자다. 그로스를 만나 HTMX, 하이퍼스크립트와 같은 프로젝트의 원동력, REST의 실패, 자바스크립트의 강한 생명력 등에 대해 들어봤다.

Q 일단 유명한 글인 '원시인의 뇌를 가진 개발자'부터 시작하자. 프로그래밍 관련해 궁금하다면 무조건 이것부터 읽으라고 해도 될 만큼 좋은 글이다.

A 감사하다. 젊은 세대 개발자를 생각하며 썼지만 대다수 개발자가 읽어볼 만한 글이라고 생각한다. 물론 완벽하지는 않다. 원시인 말투로 재미있게 쓰려고 해 격식 있는 글은 아니지만, 글에 나오는 대부분 조언은 여전히 유효하다.

Q 실제로 직면한 코딩 문제 대부분은 개발자 자신이 똑똑하다는 생각에서 비롯된다.

A 그렇다. 프로그래밍 경력이 어느 정도 되면 더는 머릿속에 시스템 전체를 담을 수 없음을 깨닫는 시점을 경험한다. 다들 항상 스스로 너무 똑똑하다고 생각하는 것인지, 일이 너무 복잡해지기 시작할 때를 제대로 인지하지 못하는 것인지는 모르겠다. 실제로 기술직은 "이건 내가 감당하기에 너무 복잡하다"라고 말하기가 쉽지 않은 것도 사실이다.

Q 공감한다. 전에 친구와 이야기하면서 "자바스크립트는 너무 유연해 무엇이든 코딩으로 해결할 수 있다"라고 했더니 그 친구는 "하하, 위험한 말이네"라고 했다. 자바스크립트 언어에 대해 어떻게 생각하나?

A 자바스크립트는 괜찮은 스크립팅 언어지만 매우 좋은 범용 언어는 아니다. 이제는 기능이 너무 많고 원하는 작업을 할 수 있는 방법이 너무 많다는 면에서 C++와 비슷하다. 반대로, 자바스크립트에는 한 가지 정말 놀라운 특징이 있다. 브라우저를 기반으로 한다는 것이다. 덕분에 앞으로도 한동안 웹용 핵심 스크립팅 언어 위치를 유지할 것이다.

Q 하이퍼스크립트에 깊은 인상을 받았다.

A 하이퍼스크립트는 프론트엔드 스크립팅을 두고 자바스크립트와 '경쟁하는' 언어다. 하이퍼카드(HyperCard)용 스크립팅 언어인 하이퍼토크(HyperTalk)를 기반으로 하며, 단순히 웹 페이지에서 스크립팅을 하려는 경우 더

개선된 작성 경험을 제공한다. 예를 들어 하이퍼스크립트 런타임은 프로미스(promise)를 마주치면 자동으로 해결해 스크립트 작성자가 따로 처리할 필요가 없다. 구문은 영어와 비슷한 자연스러운 구문인데, 좋아하는 사람도 있지만 싫어하는 이들도 많다. 확실히 열정을 갖고 추진 중인 프로젝트이며, 올해 HTMX 2가 나온 다음 하이퍼스크립트도 1.0 버전에 이르기를 기대한다.

하이퍼스크립트는 자바스크립트에서 렉서, 파서, 평가 기반 런타임으로 구현되는데, 놀랍게도 이렇게 해도 될 만큼 자바스크립트 런타임이 빠르다. 하이퍼스크립트로 비트코인 채굴기를 구현할 정도는 아니지만, 가벼운 DOM 스크립팅 용도로는 모자람이 없다. 하이퍼스크립트 파서와 런타임은 자바스크립트에서 내가 감내할 수 있는 복잡성의 끝자락에 있다.

Q HTMX는 매우 인상적인 언어다. 관련 기사에 대한 조회수를 보면 사람들이 HTMX에 상당한 관심이 있음을 알 수 있다. 또한 이 프로젝트는 진정한 REST 아키텍처의 가능성을 복원하려는 듯 느껴진다.

A 2023년에는 HTMX에 많은 일이 있었다. 아직 HTMX가 생소한 이들을 위해 잠시 설명하면, HTMX는 내가 만든 자바스크립트 라이브러리로, HTML에 특별한 속성을 추가한다. 개념만 보면 표준 HTML 링크와 양식의 href 및 액션 속성과 매우 비슷하다. 이런 속성을 활용해 AJAX 요청을 트리거한 다음 서버가 응답하는 HTML로 DOM의 일부를 대체한다.

Q HTMX는 현대적 사용, 특히 AJAX를 처리하도록 기존 HTML을 확장한다고 느꼈다.

A 매우 좋은 설명이다. 나는 종종 "하이퍼미디어로서 HTML을 완성하는 것이 HTMX다"라고 이야기한다. HTMX가 페이지의 어느 요소든 이벤트에 대한 응답으로 HTTP 요청을 발행한 다음 이 응답을 DOM의 어디에나 배치할 수 있다는 의미다. 단순하게 들릴 수 있지만 HTML을 조금만 일반화하면 기존에 자바스크립트로만 가능했던 UX 패턴을 구현할 수 있다. 예를 들어 무한 스크롤, 페이지 섹션의 지연 로딩, 입력과 동시에 수행되는 서버 측 유효성 검사 같은 것이다.

Q 그동안 REST가 무엇을 의미하는지 잘 안다고 생각했는데, 오랫동안 잘못 생각하고 있었던 것 같다. 당신은 'REST는 어떻게 REST와 반대되는 의미를 갖게 되었는가' 글에서 이 부분을 언급했다. 구체적으로 어떤 의미인가?

A 업계에서 REST라는 용어를 사용하는 방식은 바뀌지 않는다(웃음). 현재 이 용어는 "HTTP 위의 JSON API"를 의미하며, 이는 아마 우리가 은퇴할 때까지 바뀌지 않을 것이다. 그러나 REST가 원래 의도했던 것은 그게 아니다. REST는 JSON API가 존재하기 이전에 월드 와이드 웹의 작동 방식을 기술했다. 브라우저를 통해 상호작용을 이끄는 링크와 양식이 전부였다. REST라는 용어는 로이 필딩의 논문에서 처음 등장했는데, 그는 RESTful 시스템이 준수해야 하는 일련의 제약 조건을 정의했다.

흥미로운 것은, 현재 "RESTful"로 지칭되는 대부분의 JSON API는 이런 제약 조건을 충족하지 않는 반면, 서로

연결되는 HTML 페이지 몇 개로 구성된 간단한 정적 웹사이트가 이런 제약 조건을 모두 충족한다는 것이다. 간단한 정적 웹사이트를 만드는 사람이 이력서에 "RESTful" JSON API 엔지니어라는 화려한 직함을 달고 다니는 사람들보다 어느 면에서는 더 나은 "REST" 엔지니어인 셈이다. 언어는 바뀌지 않겠지만 웹이 이렇게 유연한 이유를 제대로 이해하기 위해서는 REST라는 용어의 원래 의미, 특히 REST의 '균일한 인터페이스 제약 조건'을 아는 것이 중요하다고 생각한다.

Q 개인적으로 지금까지 프로그래밍 강좌를 들어본 적이 없다. 아직도 대학에서는 책을 사용하는가?(웃음). 프로그래밍을 가르친다는 것은 어떤가? 누구나 CSCI 468 과정을 들을 수 있나?

A 468은 내가 가르치는 컴파일러 수업인데 밥 나이스트롬이 쓴 크래프팅 인터프리터(Crafting Interpreters)를 읽으면 수업 내용의 대략 50%를 습득할 수 있다. 수업에서는 JVM과 백엔드의 바이트코드를 주로 다루지만, 솔직히 말해 자바에 푹 빠진 사람이 아니라면 별로 재미있는 수업은 아니다. 나는 가르치는 것을 좋아한다. 대학 교수 연봉은 터무니없이 적고 관료주의도 감내해야 하지만 학생들이 어려움을 극복하고 스스로 프로그래밍할 수 있다는 자신감을 느끼는 과정을 지켜보는 것이 즐겁다.

Q 다시 REST에 대한 이야기로 돌아가서 더 높은 관점에서 보자. 당신은 하이퍼미디어 시스템의 공동 저자인데, 이 책을 보면 웹이 설계된 방식이 곧 애플리케이션 아키텍처이며, 제대로만 활용하면 이 아키텍처를 기반으로 애플리케이션을 만들 수 있다고 주장한다.

A 맞다. 그리고 HTMX는 원래의 아키텍처를 새로운 것, 예를 들어 고정 형식 JSON 데이터 API로 대체하는 것이 아니라 하이퍼미디어로서 HTML을 강화해 이 네트워크 아키텍처를 기반으로 쌓아 올리려는 시도다. 책에서는 제대로 기능하는 하이퍼미디어의 시스템적 속성을 개략적으로 설명하기 위해 노력했다. 예를 들어 나는 균일한 인터페이스가 제대로 작동하는 데 있어 브라우저 같은 하이퍼미디어 클라이언트가 얼마나 중요한지 오랫동안 제대로 알지 못했다. 전체를 의도한 대로 작동하도록 하려면 HTML과 같은 하이퍼미디어, HTTP와 같은 네트워크 프로토콜, 하이퍼미디어 서버, 하이퍼미디어 클라이언트까지 다양한 시스템이 필요하다.

Q 인터뷰를 정리하며 중요한 내용을 요약하려 했는데, 그 대답이 딱 들어맞는 것 같다. 앞서 언급한 '원시인의 뇌를 가진 개발자' 글은 표현의 복잡성, 클로저, 거절하기 등 모든 주제에 좋은 조언을 제시한다. 같은 맥락에서 방문자 패턴에 대해서는 어떻게 느끼는지는 궁금하다.

A 원시인의 뇌 에세이에서 언급한 것과 달리 나는 방문자 패턴에 100% 반대하지는 않는다. 사실 나는 대체로 소프트웨어 개발의 어떤 것에 대해서도 100% 찬성하거나 반대하지 않는다. 다만 트리를 통해 연산을 수행하는 다른 측면을 두는 것보다는 트리에 직접 연산을 인코딩하는 것이 더 낫다고 생각한다.

예를 들어 하이퍼스크립트에서 파싱 트리 요소에는 eval 메서드가 정의되는데, 파싱되는 바로 그 지점에서 직접 정의된다. 이렇게 되면 관심사(concern)가 혼합되지만 프로그래밍 자체는 더 명확해진다. "wait" 명령이 어떻게 작동하는지 확인할 때 어떻게 파싱되고 평가되는지 한 곳에서 볼 수 있기 때문이다. 이 부분이 매우 유용하다.

Q 모두가 복잡성이 나쁘다는 것을 안다. 그러면서도 끌리는 이유는 무엇일까?

A 그 부분에서도 업계의 압력이 작용한다고 생각한다. 소프트웨어는 냉혹한 산업이며, 남에게 똑똑하지 않은 것처럼 보이면 경력에 지장이 생길 수 있다. 즉, 우리는 모두 똑똑함을 입증해야 한다는 압박감, 그리고 다른 사람의 코드가 혼란스럽거나 지나치게 복잡해 보인다는 것을 인정하면 안 된다는 부담을 느낀다. 실제로 많은 IT 조직에서 "복잡하다"는 칭찬이고, "단순하다"는 비판으로 여겨진다.

나는 이런 알력이 있음을 이해하고 압박을 느끼는 엔지니어에게 동정심도 느낀다. 내가 항상 주장하는 것이 있다. 실무에서 어떤 것이 실제로 너무 복잡하게 보일 때, 엔지니어링 조직의 선임 엔지니어가 큰 소리로 "복잡하다"고 말해야 한다는 것이다. 그래야 젊은 엔지니어도 제 생각을 말할 수 있다. 조직 전체적으로 이런 표현에 대한 압박에서 어느 정도는 벗어날 수 있다.