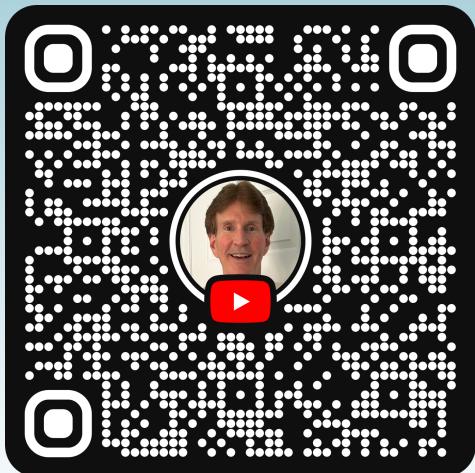




# Web Development Simplified



R. Mark Volkmann  
Object Computing, Inc.  
<https://objectcomputing.com>  
✉️ mark@objectcomputing.com  
🐦 @mark\_volkmann



—

“The single most important thing I have learned about software development over my career is that **if you do not aggressively fight complexity, it will eat you alive.**”

Vicki Boykis



**OBJECT COMPUTING**  
YOUR OUTCOMES ENGINEERED

Slides at <https://github.com/mvolkmann/talks/>



## Titanium Sponsors



### Gold Sponsors



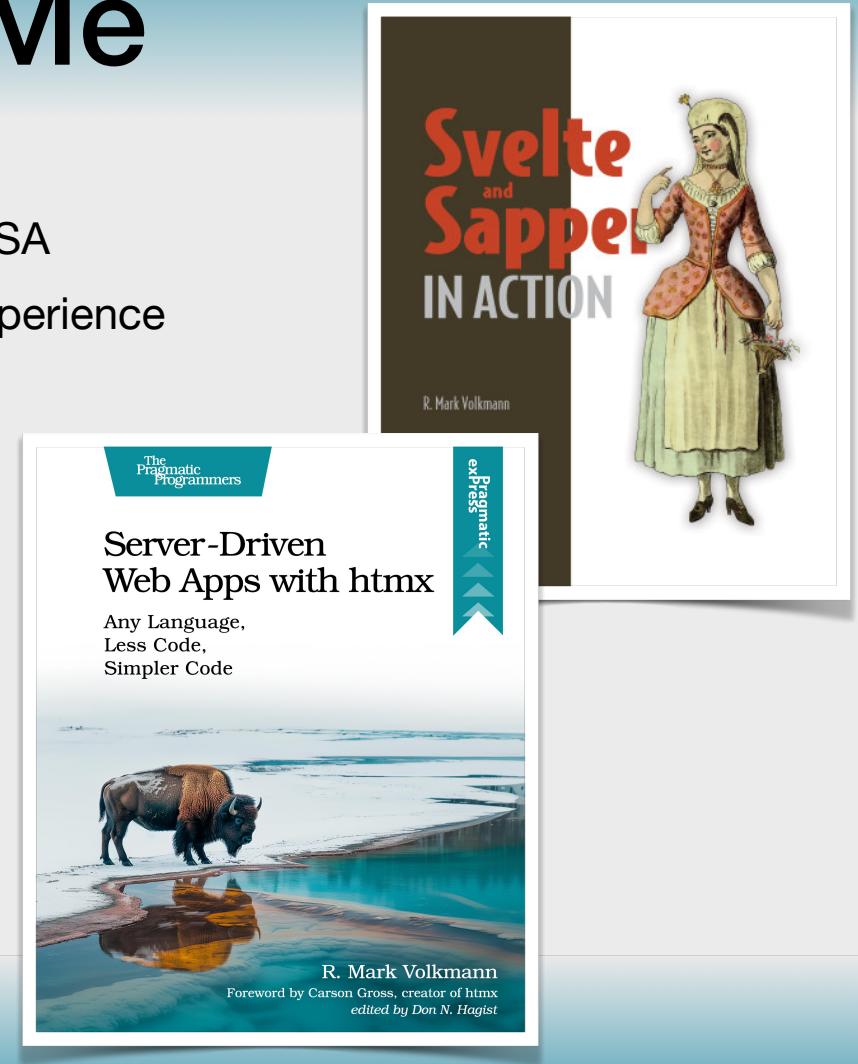
### Other Awesome Organizations



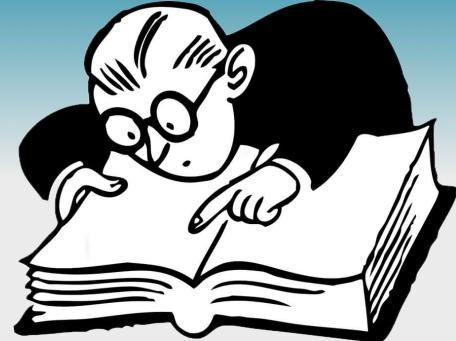


# About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 44 years of professional software development experience
- Writer and speaker
- **Blog** at <https://mvolkmann.github.io/blog/>
- Author of Manning book “**Svelte ... in Action**”
- Author of Pragmatic Bookshelf book “**Server-Driven Web Apps with htmx**”

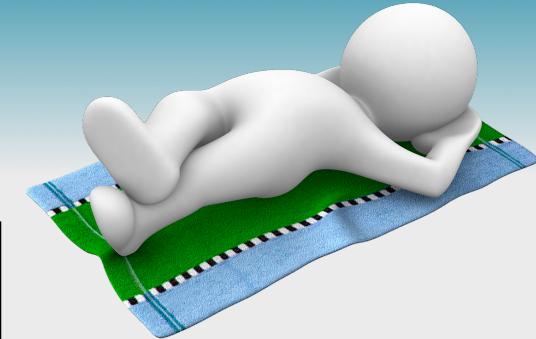


# Terminology



- **Hypermedia:** any data format that can describe branching from one “media” (e.g. a document) to another, like HTML
- **Hypermedia control:** an element that describes a server interaction, such as HTML anchor () and **form** elements
- **Hypermedia client:** software that understands and renders a hypermedia format, such as web browsers with HTML
- **HATEOAS:** Hypermedia As The Engine Of Application State pronounced “hay toss”
  - all allowed user actions are described by hypermedia controls found in endpoint responses that contain hypermedia
- **Hypermedia-Driven Application (HDA):** application that uses HATEOAS

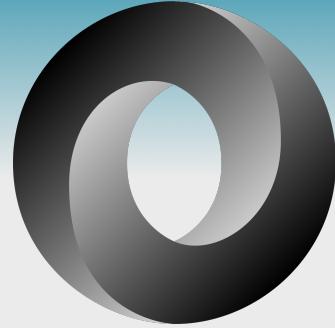
# What is REST?



- Defined by **Roy Fielding's PhD dissertation**
- A software architecture is **RESTful** if it
  - uses a client/server model
  - is stateless
  - can cache responses
  - supports a uniform interface ★
- A **uniform interface** is one where
  - requests identify a resource (e.g. good URLs)
  - resources are manipulated through their representations
  - messages are self-descriptive ★
  - **HATEOAS is used** ★

“Architectural Styles  
and the Design of  
Network-based  
Software Architectures”  
in 2000

# JSON-based APIs

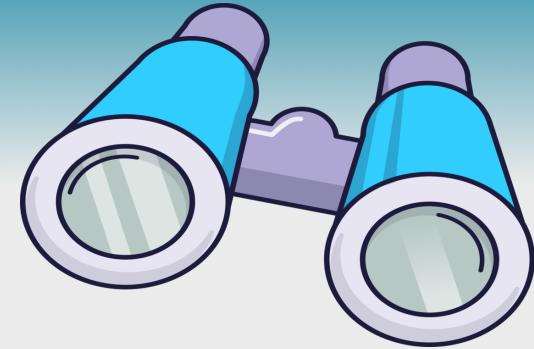


- Typically referred to as REST APIs
- Are **NOT REST** because JSON is not a hypermedia format
  - browsers know how to parse JSON, but don't know how to render it in a way that allows users to interact with it
  - even if JSON data described allowed user interactions, browsers wouldn't understand how to support them
  - requires custom, client-side JavaScript code → such as frameworks like React, Svelte, Vue, or Angular
- JSON-based APIs aren't bad, but they are not REST APIs

Roy Fielding said

“I am getting frustrated by the number of people calling any HTTP-based interface a REST API. ... That is RPC.”

# htmx Overview



- Client-side JavaScript library for implementing hypermedia-driven applications
  - adds support for new HTML attributes that make HTML more expressive
  - uses endpoints that return HTML rather than JSON
  - free and open-source [Zero-Clause BSD license](#)
- <https://htmx.org/>

# Tech Stacks ...

- Can use any programming language and framework that can implement an **HTTP server** whose endpoints return **HTML responses**
- Referred to as “Hypermedia On Whatever you’d Like” (**HOWL**)
- **Good choices make it easy to**
  - create new endpoints for any HTTP verb
  - specify type checking and validation of request data
  - get request data from headers, path parameters, query parameters, and bodies
  - send HTTP responses that include headers and bodies that contain text or HTML



# ... Tech Stacks



- **Good choices have tooling that supports**
  - **fast server startup** with no build process or a simple one
  - **automatic server restarts** after source code changes are detected
  - **good HTML templating** support (such as JSX) or my npm package js2htmlstr without relying on string concatenation
  - **syntax highlighting** of embedded HTML in code editors



# Creating a Project



- Let's walk through using one tech stack ... TypeScript, Bun, and Hono
  - install Bun from <https://bun.sh>
  - create a directory, cd to it, and enter **bun init**
  - install Hono with **bun add hono**
  - edit **tsconfig.json**  Add this inside **compilerOptions**:  
"jsxImportSource": "hono/jsx",
  - rename **index.ts** to **index.tsx** so JSX can be used to generate HTML
  - create **public/index.html** and **public/styles.css** as shown on next slide
  - modify **index.tsx** as shown on next slide
  - enter **bun run index.tsx**
  - browse localhost:3000

**Other options include:**

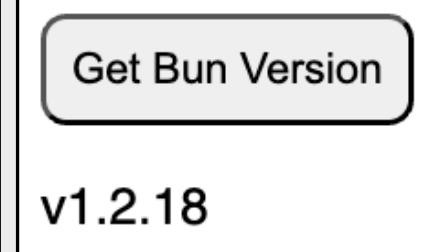
- Go with templ
- Python with Flask or Django
- TypeScript with Astro

# Demo Files

```
<html>                                              public/index.html
  <head>
    <title>htmx Demo</title>
    <link rel="stylesheet" href="styles.css" />
    <script
      src="https://unpkg.com/htmx.org@2.0.6"
    ></script>
  </head>
  <body>
    <button hx-get="/version" hx-target="#version">
      Get Bun Version
    </button>
    <div id="version"></div>
  </body>
</html>
```

```
body {                                              public/styles.css
  font-family: sans-serif;
}

button {
  border-radius: 0.5rem;
  margin-bottom: 1rem;
  padding: 0.5rem;
}
```



```
import {type Context, Hono} from 'hono';
import {serveStatic} from 'hono/bun';

const app = new Hono();

// Serve static files from public directory.
app.use('/*', serveStatic({root: './public'}));

app.get('/version', (c: Context) => {
  // Return a Response whose body contains
  // the version of Bun running on server.
  return c.text('v' + Bun.version);
});

export default app;
```

When this button is clicked, an HTTP GET request is sent to `/version`. The text it returns replaces `innerHTML` of element with id “`version`”.



# Hono



- Hono route methods are passed a **Context** object (`c` in previous code)
  - used to get request data and create responses
  - can use **Zod** (<https://zod.dev/>) to validate requests

Action	Code
get value of request header	<code>c.req.header('Some-Name')</code>
get value of path parameter	<code>c.req.param('some-name')</code>
get value of query parameter	<code>c.req.query('some-name')</code>
get value of text body	<code>const text = await c.req.text();</code>
get FormData from body	<code>const formData = await c.req.formData();</code>
get property from formData	<code>const value = (formData.get('property') as string)    '';</code>
get value of JSON body	<code>const object = await c.req.json();</code>

Action	Code
set value of response header	<code>c.header('Some-Name', 'some value');</code>
set status code	<code>c.status(someCode);</code>
return text response	<code>return c.text('some text');</code>
return JSON response	<code>return c.json(someObject);</code>
return HTML response	<code>return c.html(someHTML);</code>
return "Not Found" error	<code>return c.notFound();</code>
redirect to another URL	<code>return c.redirect('someURL');</code>

# Pros ...

5 Slides Worth!



- **Fixes HTML shortcomings**
  - any user interaction on any HTML element can trigger any kind of HTTP request and insert an HTML response without full page refresh
- **Improves startup time**
  - metrics such as “First Contentful Paint” and “Time to Interactive”
  - significant for users with old computers/phones or slow internet connections
- **Favors Locality of Behavior (LoB) over Separation of Concerns (SoC)**
  - places related code together which makes code easier to understand and modify

mostly refers to associating logic with HTML elements by adding new attributes



# ... Pros ...

- **Enables HOWL** Hypermedia On Whatever you'd Like
  - use any programming language that can implement an HTTP server whose endpoints return HTML
- **Encourages full-stack development**
  - often in SPA development one team implements JSON endpoints and another team implements UIs that use them
  - with htmx, developers implement complete features by defining endpoints that return HTML with htmx attributes
  - requires developers to know a programming language, HTML, and CSS, but not necessarily JavaScript

# ... Pros ...



- **Eliminates JSON as intermediate format**
  - in SPA applications
    - endpoints fetch data, serialize it to JSON, and return JSON
    - client code parses JSON and generates HTML from it
  - in htmx applications
    - endpoints fetch data, generate HTML from it, and return HTML
    - browser only has to insert returned HTML
    - no custom client-side code is required
  - results in simpler, faster code

# ... Pros ...



- **Eliminates need for API versioning**
  - JSON APIs
    - client-side code parses JSON and extracts data from it
    - if used by multiple clients, APIs must be versioned and each version must remain stable to avoid breaking clients
  - HTML APIs
    - client-side code does not parse HTML and extract data from it
    - intended for use by a single client application and can be specific to it
    - can be freely modified as long as the desire is for all users to get latest version on next site visit

Can still share code between endpoints that handles tasks such as database queries.

# ... Pros



- **Simplifies client-side code**
  - client-side code is mostly unnecessary because logic is embedded in HTML elements returned by server
  - fewer client-side dependencies are needed
- **Simplifies state management**
  - typical SPA applications manage state on both server and client
    - keeping state in sync in two places is tedious and error-prone
  - with htmx, non-UI state is only maintained on server
    - no state synchronization is needed and browser memory usage is reduced
    - returned HTML is a representation of state, not really state

# Cons



- htmx is a **new way of thinking** about web development
  - will take time to learn common patterns
- htmx is **not appropriate for apps that**
  - need UI updates on every mouse move or drag
    - too slow if each movement triggers a new HTTP request
    - examples include Google Maps and many games
  - require changes in one part of UI to trigger changes in many others
    - such as  
spreadsheet formulas
- Harder to **find examples** using your chosen tech stack
  - HOWL means apps are implemented using a wide variety of tech stacks



# htmx History



- Created by **Carson Gross**
  - principal software engineer at Big Sky Software
  - Computer Science instructor at Montana State University
- Work on predecessor **intercooler.js** began in 2013
- **First version** of htmx was released in May 2020
- **Latest version** 2.0.6 released June 2025 is **17.5K** minified and compressed
- Has **extensive** set of integration **tests** implemented in Mocha
- Interest in htmx exploded in 2023 after **YouTube videos** from **ThePrimeagen** and **Fireship** were released
- Strong showing in 2023 **JavaScript Rising Stars**
  - 2nd place in “Front-end Frameworks” behind React



# HTTP Verb Review

A blue hexagonal button with the word "HTTP" in white capital letters.

- **POST** - create and non-CRUD operations
- **GET** - read
- **PUT** - update all
- **PATCH** - update some
- **DELETE** - delete

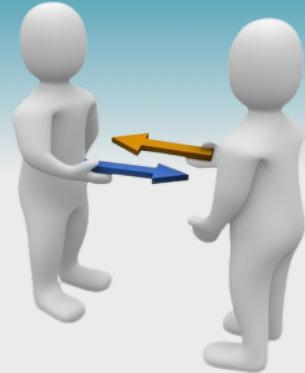
# htmx Basics



- Add htmx attributes to elements that trigger HTTP requests
- Specify events that trigger request
  - **hx-trigger** comma-separated list of event names with optional modifiers
- Specify HTTP verb to use and endpoint URL
  - **hx-get**, **hx-post**, **hx-put**, **hx-patch**, and **hx-delete**
- Specify element where response HTML will go
  - **hx-target** can be CSS selector and/or several keywords; defaults to **this**
- Specify where to place HTML relative to target
  - **hx-swap** see next slide

All elements have a **default trigger**.  
**form** elements trigger on **submit**.  
**input**, **textarea**, and **select** elements trigger on **change**.  
All other elements trigger on **click**.

# hx-swap



Assume **hx-target** refers to the **ul** element.

Options to  
insert content

**beforebegin**

**afterbegin**

**beforeend**

**afterend**

```
<p>before list</p>
<ul>
  <li>Red</li>
  <li>Green</li>
  <li>Blue</li>
</ul>
<p>after list</p>
```

Options to  
replace content

**outerHTML**

**innerHTML** (default)

Options that do not  
use response HTML

**delete**

removes target element

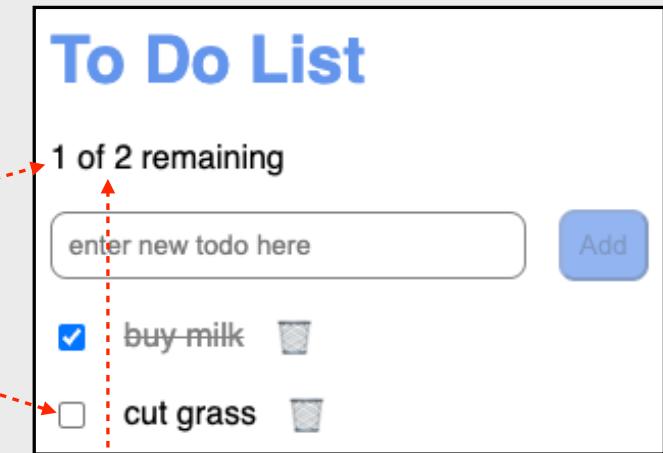
**none**

leaves target element as-is

# Endpoints

- Endpoints can return any combination of
  - one element to be placed **relative to target**
    - example: a new todo item
  - any number of elements to be placed **out-of-band**
    - example: updated status text
  - an **HX-Trigger** header to trigger **events** in browser
    - example of handling

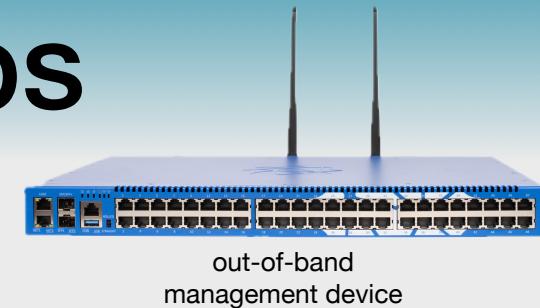
```
<p hx-get="/todos/status" hx-trigger="load, status-change from:body" />
```



**status-change** event is triggered when a new todo is added, a todo is deleted, or the done status of a todo changes.

The event bubbles up to **body** element.

# Out-of-band Swaps



- Returned HTML can target multiple elements
  - one primary element and any number of additional elements that replace existing elements with a **matching id**
- Example returned HTML

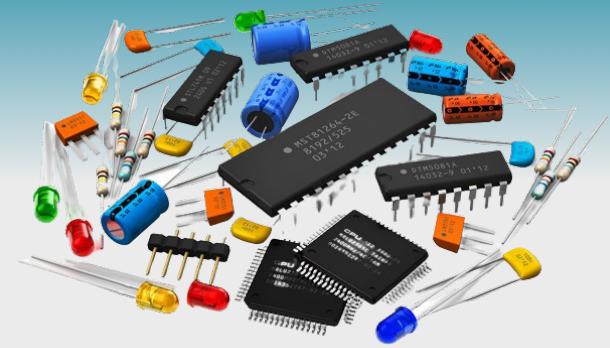
JSX fragment

```
<>
  <TodoItem todo={todo} />
  <p id="error" hx-swap-oob="true">
    {message}
  </p>
</>
```

- To place OOB element relative to target rather than replace it, use an **hx-swap** value for **hx-swap-oob** (e.g. **beforeend**)

# Components

- Bun endpoints can generate HTML with JSX
  - JSX is perhaps the best part of React
  - other parts like virtual DOM and hooks are not used by Bun
- JavaScript functions can return JSX
  - see `TableRow` in “Infinite Scroll” example later (5 more slides)
  - these functions can be thought of as UI components
- Returned elements can include `htmx` and `Alpine` attributes
  - see Alpine example later



# Input Validation



```
<div>
  <label for="email">Email</label>
  <input
    id="email"
    hx-trigger="keyup changed delay:200ms"
    hx-get="/email-validate"
    hx-target="#email-error"
    name="email"
    placeholder="email"
    required
    size="30"
    type="email"
  />
  <span class="error" id="email-error" />
</div>
```

not nested in  
a form, so  
includes value as  
a query parameter

<https://github.com/mvolkmann/htmx-examples/tree/main/input-validation>

```
app.get('/email-validate', (c: Context) => {
  const email = c.req.query('email') || '';
  const valid = validEmail(email);
  return c.text(valid ? '' : 'email in use');
});
```

## Sign Up

Email  email in use

Password  invalid password

# Lazy Loading

<https://github.com/mvolkmann/htmx-examples/tree/main/lazy-load>



Assume this `div` is part of a larger page and is not rendered until the user scrolls down the page.

```
<div  
    hx-trigger="revealed"  
    hx-get="/users"  
    hx-indicator=".htmx-indicator"  
/>  

```

`hx-target` defaults to **this**.  
`hx-swap` defaults to **innerHTML**.

## Users

ID	Name	Email	Company
1	Leanne Graham	Sincere@april.biz	Romaguera-Crona
2	Ervin Howell	Shanna@melissa.tv	Deckow-Crist
3	Clementine Bauch	Nathan@yesenia.net	Romaguera-Jacobson
4	Patricia Lebsack	Julianne.OConner@kory.org	Robel-Corkery
5	Chelsey Dietrich	Lucio_Hettinger@annie.ca	Keebler LLC
6	Mrs. Dennis Schulist	Karley_Dach@jasper.info	Considine-Lockman
7	Kurtis Weissnat	Telly.Hoeger@billy.biz	Johns Group
8	Nicholas Runolfsdottir V	Sherwood@rosamond.me	Abernathy Group
9	Glenna Reichert	Chaim_McDermott@dana.io	Yost and Sons
10	Clementina DuBuque	Rey.Padberg@karina.biz	Hoeger LLC

# Active Search



<https://github.com/mvolkmann/htmx-examples/tree/main/active-search>

```
<label for="name">Name</label>
<input
  hx-trigger="keyup changed delay:200ms"
  hx-post="/search"
  hx-target="#matches"
  name="name"
  size="10"
/>
<ul id="matches" />
```

Name ar  
Mark  
Richard

nested in a **form**, so  
includes value as form data

```
app.post('/search', async (c: Context) => {
  const data = await c.req.formData();
  const name = (data.get('name') as string) || '';
  if (name == '') return c.html('');

  const lowerName = name.toLowerCase();
  const matches = names.filter(n => n.toLowerCase().includes(lowerName));
  return c.html(
    <>
      {matches.map(name => (
        <li>{name}</li>
      ))}
    </>
  );
});
```

array defined  
outside this function

# Infinite Scroll ...

```
<table
  hx-trigger="load"
  hx-get="/pokemon-rows?page=1"
  hx-indicator=".htmx-indicator"
  hx-swap="beforeend"
>
  <tr>
    <td>ID</td>
    <td>Name</td>
    <td>Description</td>
  </tr>
</table>

```

<https://github.com/mvolkmann/htmx-examples/tree/main/infinite-scroll>

```
app.get('/pokemon-rows', async (c: Context) => {
  const page = c.req.query('page');
  if (!page) throw new Error(
    'page query parameter is required');

  const pageNumber = Number(page);
  const offset = (pageNumber - 1) * ROWS_PER_PAGE;
  const url = POKEMON_URL_PREFIX +
    `?offset=${offset}&limit=${ROWS_PER_PAGE}`;
  const response = await fetch(url);
  const json = await response.json();
  const pokemonList = json.results as Pokemon[];

  return c.html(
    <>
      {pokemonList.map((pokemon, index) => {
        const isLast = index === ROWS_PER_PAGE - 1;
        return TableRow(pageNumber, pokemon, isLast);
      })}
    </>
  );
});
```

on next slide

## Infinite Scroll

ID	Name	Description
1	bulbasaur	
2	ivysaur	
3	venusaur	
4	charmander	
5	charmeleon	
6	charizard	

# ... Infinite Scroll



function that acts like  
a UI component

```
function TableRow(page: number, pokemon: Pokemon, isLast: boolean) {
  const attributes = isLast
    ? {
        'hx-trigger': 'revealed',
        'hx-get': '/pokemon-rows?page=' + (page + 1),
        'hx-indicator': '.htmx-indicator',
        'hx-swap': 'afterend'
      }
    : {};
  const {name, url} = pokemon;
  const id = url.split('/')[6]; // 7th part of URL
  const imageUrl =
    `https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/${id}.png`;

  return (
    <tr {...attributes}>
      <td>{id}</td>
      <td>{name}</td>
      <td>
        <img alt={name} src={imageUrl} />
      </td>
    </tr>
  );
}
```

same attributes as  
on **table** element



# Boosting



- Consider anchor (`a`) and `form` elements that do not have `htmx` attributes like `hx-get` or `hx-post`
- Those send HTTP requests to a given URL and perform a **full page refresh**
- Can add `hx-boost="true"` to any element
  - changes descendant plain anchor and `form` elements to send HTTP requests using AJAX if JavaScript is enabled
- Results in **faster navigation** and **better user experience**
  - rather than full page refresh,  
`body` content in response replaces that in current page and `title` in `head` replaces that in current `head`
  - **avoids processing** `link` and `script` tags in response `head`
  - **assumes** current page has already loaded all required CSS and JavaScript

add to `body` to “boost” all plain anchor and `form` elements

# More to Investigate



- htmx has support for all these features that we didn't have time to cover
  - animation with CSS transitions
  - WebSockets
  - Server-Sent Events
  - History API
  - security through a Content Security Policy (CSP) or sanitizing HTML
  - htmx JavaScript API



# Hyperview



- Hypermedia approach to developing mobile apps
  - for Android and iOS
- Builds on React Native
- <https://hyperview.org/>

# Alpine



- “A lightweight JavaScript framework that uses custom HTML attributes to add dynamic behavior”
- Consider using this with htmx for client-side state and interactivity
- Example

similar to how htmx adds custom HTML attributes

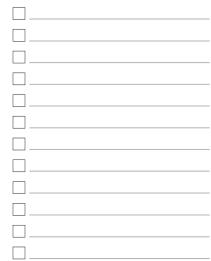
```
<div style="display: flex; gap: 1rem" x-data="{ count: 0 }">
  <button x-bind:disabled="count <= 0" x-on:click="count--">
    -
  </button>
  <div x-text="count"></div>
  <button x-on:click="count++">
    +
  </button>
</div>
```



See my **blog page on Alpine** at <https://mvolkmann.github.io/blog> and also covered in my htmx book.

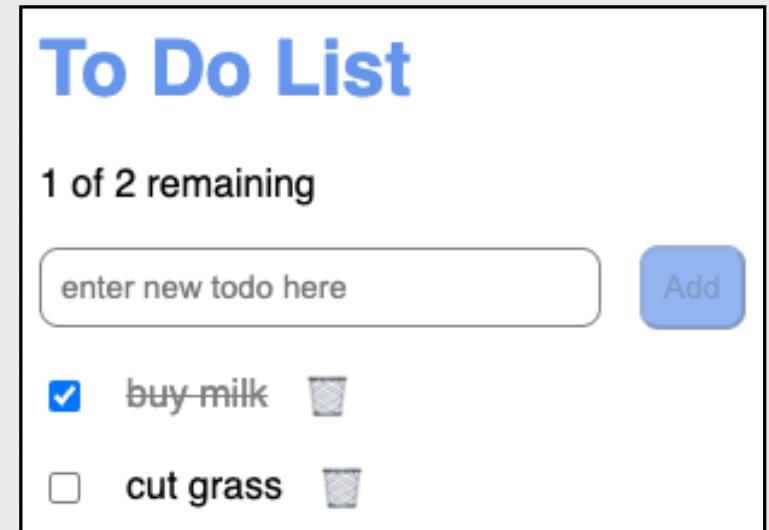
Include Alpine library in `head` element with:

```
<script src="//unpkg.com/alpinejs" defer></script>
```



# Todo App Example

- For a larger htmx example app, see <https://github.com/mvolkmann/htmxa-examples/tree/main/todo-hono>
- A todo app implemented with TypeScript, Bun, Hono, htmx, and Alpine
- Demonstrates many useful htmx patterns
- Persists data to a SQLite database for which Bun has built-in support





# fixi

term for a bicycle with a single gear that is attached in a fixed position to the rear wheel



- A minimal approach to hypermedia
  - “designed to be as lean as possible while still being useful for real world projects”
- Very small subset of htmx
  - 89 lines of JavaScript; 4K uncompressed and unminified; 3268 bytes

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script src="fixi.js"></script>
    <script>
      window.onload = () => {
        const form = document.querySelector("form");
        form.addEventListener("fx:after", () => {
          form.reset();
        });
      };
    </script>
  </head>
```

```
<body>
  <form
    fx-method="post"
    fx-action="/country"
    fx-target="#info"
    fx-swap="innerHTML"
  >
    <input name="name"
      placeholder="Country Name" required />
    <button>Submit</button>
  </form>
  <div id="info"></div>
</body>
</html>
```



# Triptych

- Means “artwork that consists of three panels”
- Set of three proposals to extend HTML, adding some features of htmx
- Written by Alexander Petros and Carson Gross
- <https://alexanderpetros.com/triptych/>
  1. Allow sending PUT, PATCH, and DELETE requests from HTML forms.
  2. Add **action** attribute to **button** element, allowing them to send HTTP requests.
  3. Support partial page replacement with HTML in HTTP responses.

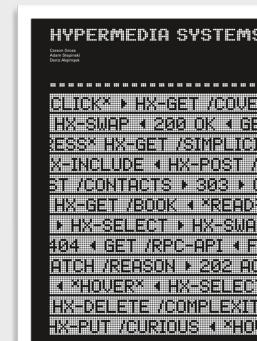


# Resources

- **htmx home page** - <https://htmx.org>
  - see docs, reference, examples, talk, and essays
- **My blog** - <https://mvolkmann.github.io/blog/> (select htmx)
- **My htmx example code** -  
<https://github.com/mvolkmann/htmx-examples/>
- **htmx Discord server** - <https://htmx.org/discord>
- **“Hypermedia Systems” book**
  - <https://hypermedia.systems/>
- **“Server-Driven Web Apps with htmx”**
  - my book from Pragmatic Bookshelf



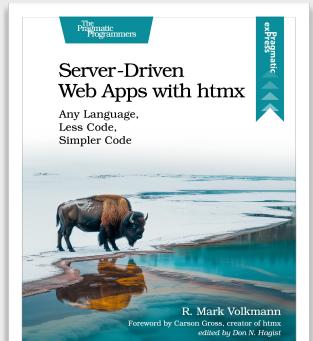
Use code  
**KCDC2025**  
for 35% off.



hard cover



soft cover



mine

# Wrap Up

- **htmx** provides a new way of implementing web applications that has many benefits
  - HTML becomes more expressive
  - code is easier to understand and you'll write less
  - state management is simplified
  - can implement with any programming language
  - faster app startup
    - due to downloading much less client-side JavaScript
  - faster client/server interactions
    - due to removal of JSON generation and parsing



Please Submit Talk Feedback



# My Latest Effort

- See npm package **wrec** which greatly simplifies creating web components
  - <https://www.npmjs.com/package/wrec>
- Name is acronym for **Web REactive Components**

```
import Wrec, {html} from './wrec.min.js';

class HelloWorld extends Wrec {
  static properties = {
    name: {type: String, value: 'World'}
  };

  static html = html`<div>Hello, <span>this.name</span>!</div>`;
}

HelloWorld.register();
<hello-world name="Mark"></hello-world>
```

