</> htmx

htmx - past is now future

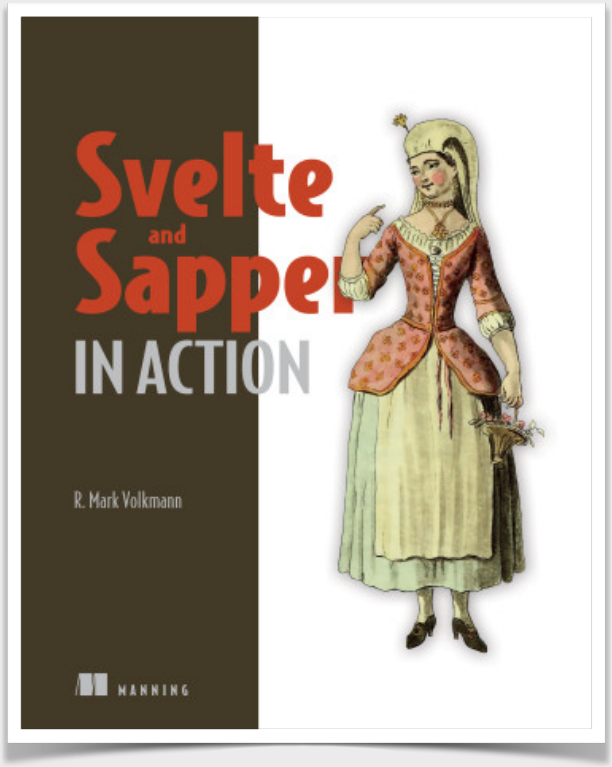
R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Slides at **https://github.com/mvolkmann/talks/**

About Me

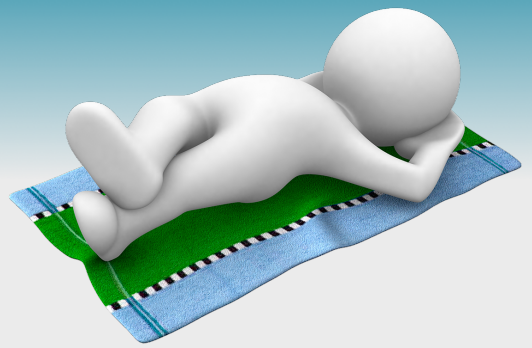
- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 43 years of professional software development experience
- Writer and speaker
- Blog at https://mvolkmann.github.io/blog/
- Author of Manning book “Svelte ... in Action”



Terminology

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

What is REST?



- Defined by **Roy Fielding's PhD dissertation**
- A software architecture is **RESTful if** it
  - uses a client/server model
  - is stateless
  - caches responses
  - supports a uniform interface ☆
- A **uniform interface** is one where
  - requests identify a resource
  - resources are manipulated through representations
  - messages are self-descriptive ☆
  - **HATEOAS is used** ☆

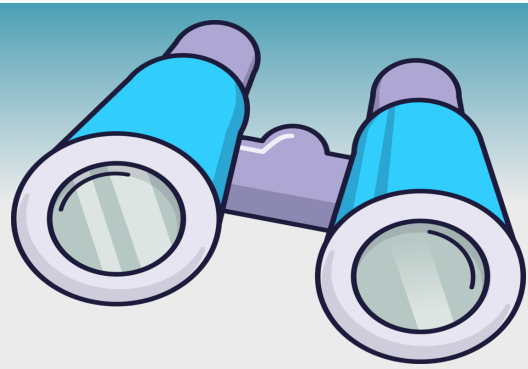
"Architectural Styles and the Design of Network-based Software Architectures"

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 a framework like React, Svelte, Vue, or Angular
- JSON-based APIs aren't necessarily 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 (HDAs)
  - adds support for new HTML attributes that make HTML more expressive
  - uses endpoints that return HTML rather than JSON
  - free and open-source
- Sponsored by at least 15 companies
  - including GitHub and JetBrains
- https://htmx.org/ Zero-Clause BSD license

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)
    without relying on string concatenation
  - **syntax highlighting** of 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
  - 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

index.tsx

```tsx
import { Hono } from "hono";
import type { Context } from "hono";
import type { FC } from "hono/jsx";

const app = new Hono();

const Layout: FC = ({ children }) => (
  <html>
    <head>
      <title>htmx Demo</title>
      <script src="https://unpkg.com/htmx.org@1.9.10">
      </script>
    </head>
    <body>{children}</body>
  </html>
);
```

FC is short for Functional Component

provides HTML boilerplate, possibly for many pages

```tsx
app.get("/", (c: Context) => {
  return c.html(
    <Layout>
      <button hx-get="/date" hx-target="#date">
        Get Data
      </button>
      <div id="date"></div>
    </Layout>
  );
});

app.get("/date", async (c: Context) => {
  return c.text(new Date().toLocaleDateString());
});

export default app;
```

browsing localhost:3000 hits this

Get Data

1/23/2024

Alternatively, get HTML from a static HTML file.

Pros ...

- **Fixes HTML shortcomings**
  - any user interaction on any HTML element
    can trigger any kind of HTTP request and
    insert an HTML response without a full page refresh
- **Improves startup time**
  - metrics such as “First Contentful Paint” and “Time to Interactive”
  - significant for users with older computers/phones and slower 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 ...

- **Eliminates JSON as intermediate format**
  - in SPA applications
    - endpoints fetch data, serialize it to JSON, and return JSON
    - browser 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 and render returned HTML
    - no custom client-side code is required
  - provides efficiency gains

.. 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 use latest version the next time they visit the site
    - initial application URL renders starting page
    - all other interactions are derived through URLs in that page and pages reached from it

crux of HATEOAS

... Pros ...

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

... Pros

- **Reduces learning curve**
  - learning htmx is significantly easier than learning a SPA framework
- **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, all non-UI state is only on server
    - no state synchronization is needed and browser memory usage is reduced
- **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

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
  - require offline support
    - “spirodonfl/htmx-offline-mode” is attempting to address this

htmx can be used in conjunction with other approaches

such as spreadsheets

htmx History

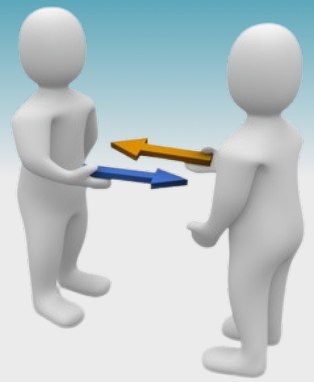
- Created by **Carson Gross**
  - principal software engineer at Big Sky Software
  - part-time Computer Science instructor at Montana State University
- Work on predecessor "**intercooler.js**" began in 2013
- **First version** of htmx was released in May 2020
- **Latest version** 1.9.10 released December 2023 is **less than 17K**
- 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

htmx Basics

- Add htmx attributes to elements that can trigger HTTP requests
- Specify events that trigger the 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 use 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`

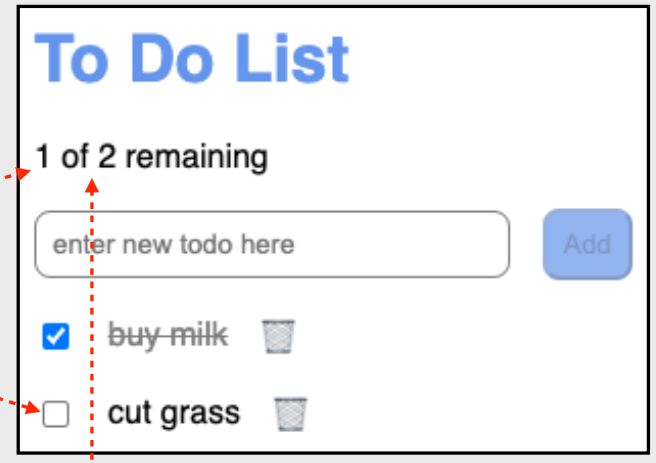
`none`

HTTP Verb Review

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

Endpoints

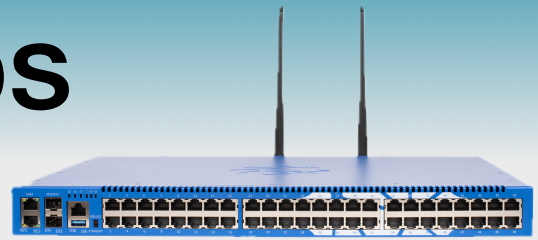
- Endpoints can return any combination of
  - one element to be placed **relative to target**
    - example: a new todo item
  - any number of elements to be placed **out-of-band**
    - example: updated status text
  - an **`HX-Trigger`** header to trigger an **event** in browser
    - example




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

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

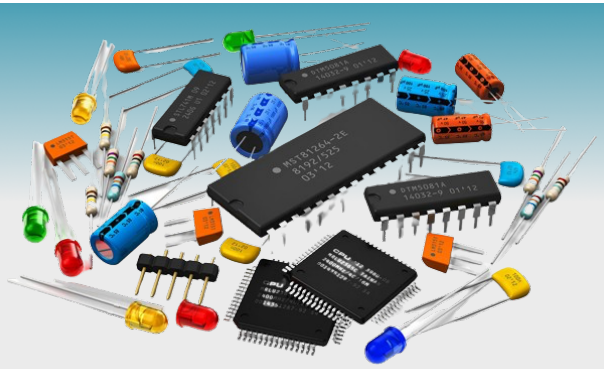
Out-of-band Swaps



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

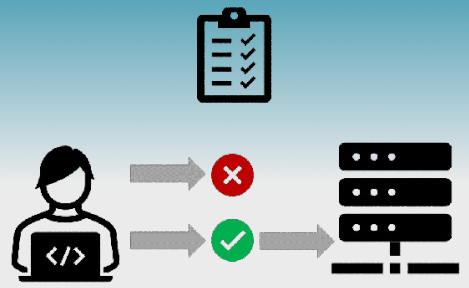
```
<>
  <TodoItem todo={todo} />
  <p id="error" hx-swap-oob="true">
    {message}
  </p>
</>
```

Components



- Can write functions that return HTML
  - see **`TableRow`** in “Infinite Scroll” example later
- HTML can be generated with JSX
  - JSX is perhaps the best part of React
  - questionable parts of React like virtual DOM and hooks are not used
- Returned elements can include htmx and Alpine attributes
  - see Alpine example later

Input Validation



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

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 old@aol.com **email in use**

Password ••••••••• **invalid password**

Submit

Lazy Loading

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

```
<div
  hx-get="/users"
  hx-indicator=".htmx-indicator"
  hx-trigger="revealed"
/>
<img
  alt="loading..."
  class="htmx-indicator"
  src="/spinner.gif"
/>
```

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"
  hx-swap="innerHTML"
  name="name"
  size="10"
/>
<ul id="matches" />
```


```
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>
      ))}
    </>
  );
});
```

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>
<img
  alt="loading..."
  class="htmx-indicator"
  src="/spinner.gif"
/>
```

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

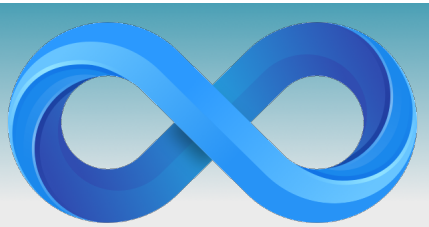
```
app.get('/pokemon-rows', async (c: Context) => {
  const page = c.req.query('page');
  if (!page) throw new Error(
    'page query parameter is required');

  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 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 the 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>
  );
}
```

Boosting

- Consider anchor (`a`) and `form` elements that do not have attributes like `hx-get` or `hx-post`
- These send HTTP requests to a given URL and perform a full page refresh

even when JavaScript is disabled

- 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

add to `body` to "boost" all plain anchor and `form` elements

- 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` replace that in current `head`
  - avoids processing `link` and `script` tags in response `head`
  - assumes current page has already loaded all required CSS and JavaScript

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

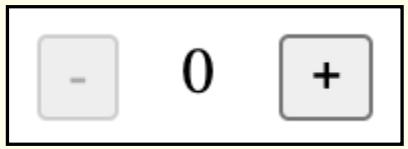
- Hypermedia approach to developing mobile apps
  - for Android and iOS
- Builds on React Native
- https://hyperview.org/

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

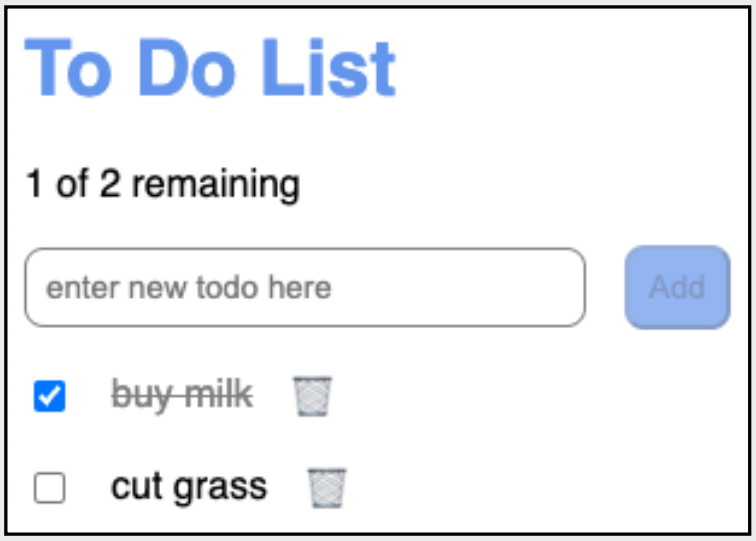
similar to how htmx adds custom HTML attributes

```
<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>
```



Todo App Example

- For a larger htmx example app, see https://github.com/mvolkmann/htmx-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



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/

Wrap Up

- **htmx** provides a new way of implementing web applications that has many benefits
  - HTML becomes more expressive
  - code is easier to understand
  - 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