

# Incremental rewrites with React, HTMX, and TanStack Start

let's talk rendering stuff

Hi I'm Swiz and we're using rewrites as an excuse to talk about modern web rendering shenanigans. We're talking about translating application state into HTML and the DOM.

---

## **wtf is this codebase!? we should rewrite**

*~ every new engineer ever*

Ok so you walk into a codebase and it's all wrong. Like it's not even full-stack JavaScript, there's a bunch of jQuery everywhere, there's no API, everything triggers a slow page reload, and it feels like a mess in there.

---

dirty little secret

## **everyone's always rewriting**

First, welcome to production software – it's trash. Reality is hard and every tutorial lied to you. You either die a prototype or become the code people complain about.


Second, a growing business changes fast. What worked for 3 engineers ain't gonna work for 30. The problems you had 6 months ago are different than the problems you're solving today. Even your customers aren't the same.


The secret of our industry is that you're always mid rewrite. Transitioning from one system to another. But you're never going to rebuild the whole thing from scratch. That would take too long


---

**The best companies don't build rock-solid systems that last forever, they learn how to change systems the fastest.**

The best companies aren't those that build rock-solid systems that last forever, it's the companies that can change systems the fastest. You can't predict the future and you can't build the perfect system so you better get good at fixing and rewriting your system bit by bit.







**Swizec Teller**   
@Swizec · [Follow](#)



Your code doesn't matter.  
But your architecture, domain modeling, data structures, and organization do.

this isn't poetry, it's a 12 book epic saga

7:24 AM · Jun 26, 2022 

 604  Reply  Copy link

[Read 12 replies](#)

So we're gonna talk about how you do that. The goal is to have a system built from independent components. The more composable your code, the easier it is to change and adapt to new requirements.

That's the real problem with jinja and jquery spaghetti. They encourage you to build monster pages that try to do everything and that makes them hard to change. Poke a thing and break everything else. The component model won for a reason.

## Step 1: React Islands

---

Most of this page comes from the server. Old skool MVC style. But what if we want a highly interactive area that follows all the modern best practices?

Here are 2 islands

Static prop from server: <b>root-1</b> Slow random number: 46 <button>Regenerate</button>	Static prop from server: <b>root-2</b> Slow random number: 46 <button>Regenerate</button>
--	--

React doesn't have to control your whole page. You can render tiny react apps side by side and leave the rest of your page alone. This is typical SPA stuff.

<https://stackblitz.com/edit/vitejs-vite-8tcyu52j?file=src%2Fmain.tsx>

---

## Standard SPA stuff

```
<p>Here are 2 islands</p>

<div style="display: flex; flex-direction: row">
  <div id="root-1"></div>

  <div id="root-2"></div>
</div>
</div>

<script>
  document.addEventListener('DOMContentLoaded', function () {
    window.exampleIsland(document.getElementById('root-1'), {
      serverValue: 'root-1',
    });
  });
</script>
```

This is pretty standard SPA stuff. Server returns HTML, JavaScript loads in the browser, renders React into a div. The only difference is that the html shows useful parts of your page instead of a blank skeleton.

---

## The islands can share memory

```
const queryClient = new QueryClient();

const Island: FC<PropsWithChildren> = ({ children }) => {
  return (
    <CssVarsProvider>
      <QueryClientProvider client={queryClient}>
        {children}
      </QueryClientProvider>
    </CssVarsProvider>
  );
};
```

You can give the islands a shared queryClient instance so they all share a cache. You get deduped API requests, coordinated reloads, etc. All the good stuff you're used to from react query and similar libraries. Any state management library that has a store should work here.

---

## You can even do form submits

```
<form action="/write-endpoint">
  {{ form.hidden_tag() }}
  <div id="react-island-root"></div>
</form>
```

Another nice thing is that you can do form submits. This lets you keep the existing backend for write requests. Just make sure your island form fields have the same names as before.

People forget that at the end of the day React renders DOM elements. If you show an input field it's an input field.

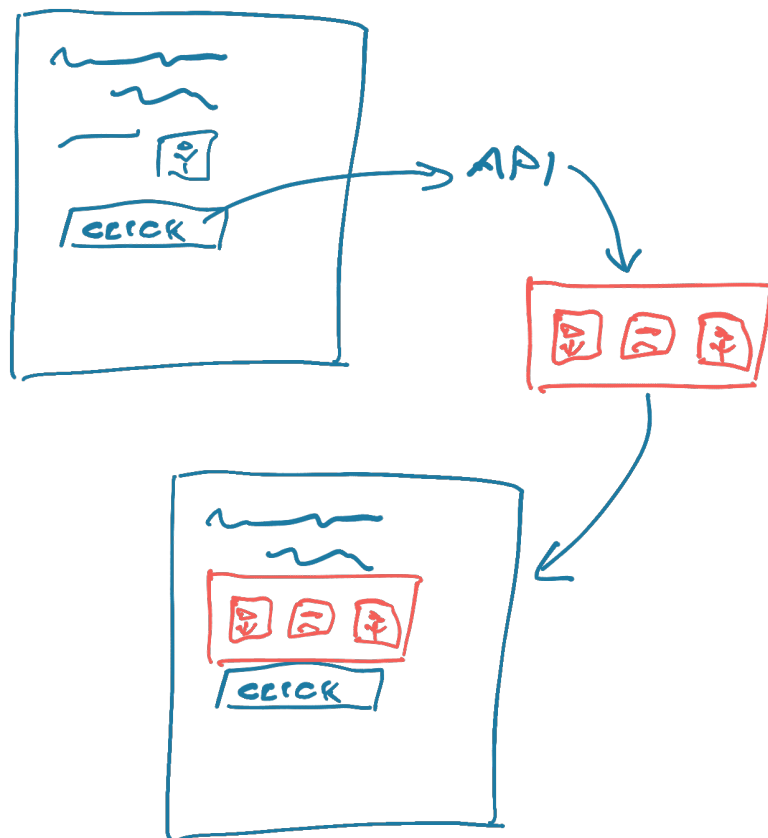
---

# What about HTML island inside React?

With islands you still have to rewrite a lot. What if you could include HTML islands inside your React components? That way you can keep more of your existing backend. Your app already knows how to hit a URL and return some UI

---

## React Server Components



React Server Components! Or React for 2 computers as Dan calls it. The basic idea is that you can render React on the server and use JSX as your API protocol instead of JSON.

Make a request, get back a component, insert into your tree, continue as normal. No need to write APIs, build JSON payloads, parse those payloads, and all the rest.

---

## Long-term there's no such thing as a decoupled client/server

Decoupling your client and server is a nice idea but I've never seen it work in practice. If you don't have 2 different clients using your API when it's built, you're gonna end up writing the perfect API for the current client. Then 6 months later it won't fit.

## RSC -> HTMX

dev.plasmidsaurus...

### Lab Management

#### Example

##### Users

Email	First Name	Last Name	Orders
swizec.teller+demo@plasmidsaurus.com	Demo	User	1
swizec.teller@plasmidsaurus.com	Swiz	T	24

cool.dino@example.com, wolfgang@example.com

Invite Users

##### Purchase Orders

PO Number	Initial Balance	Balance	Issued	Expiration
-----------	-----------------	---------	--------	------------

I couldn't get react server components to work. Not without a javascript backend, which we didn't have. The python ecosystem doesn't support this stuff.

But HTMX is built on the same idea! Make a request, get back some HTML, inject into the page, move on with your life.

The syntax is a little funny but it works great. We've been using this to modernize a lot of admin pages that follow a more traditional forms and updates pattern. You don't need a lot of reactivity, just a little.

# HTMX

```
<div hx-get="{{ url_for('settings.lab_users', lab_id=lab.id) }}"
      hx-swap="innerHTML"
      hx-trigger="load, usersUpdated from:body">
</div>

<form method="POST"
      hx-post="{{ url_for('settings.invite_users', lab_id=lab.id) }}">
  {{ form_invite_users.hidden_tag() }}
  {{ form_invite_users.emails(class="form-control") }}
  {{ form_invite_users.submit(class="btn btn-primary") }}
</form>
```

So that wasn't the prettiest example, but it was very quick to build. Add a few HTMX-y bits to the HTML and keep your whole backend the same. The GET request returns the whole table, the post request takes a regular form and returns some HTML.

---

```
{% for curr_user in lab.users | sort(attribute='first_name') %}
<tr>
<td>{{ curr_user.email }}</td>
<td>{{ curr_user.first_name }}</td>
<td>{{ curr_user.last_name }}</td>
<td>{{ curr_user.orders | count }}</td>
<td>
  <button type="button"
          class="btn btn-sm btn-danger"
          id="remove_user_btn"
          hx-delete="{{ url_for('settings.remove_user_from_lab', lab_id=lab.id,
user_id=curr_user.id) }}"
          hx-confirm="Are you sure you want to remove {{ curr_user.email }} from {{ lab.name }}"
Lab?>
    <i class="fa fa-close"></i>
  </button>
</td>
</tr>
{% endfor %}
```

The remove button is equally simplistic. You have a list of things, the button makes a delete request and your endpoint responds with a header that triggers a javascript event, which then re-fetches the table. Easy peasy.

---

# The ultimate dream

Now here's the ultimate dream – I want to write the same code and have my framework figure out the details based on its rendering environment. Return plain HTML and hydrate if we're on the server, act like an SPA if we're in the browser, give me images or pdfs if that's what I need.

**Tanner Linsley**   
@tannerlinsley · [Follow](#)



 Announcing [@Tan\\_Stack](#) Start v1 Release Candidate!

Upgrades 

 Unified Route Tree: no more server-specific files

 Type-safe middleware & server context upgrades

 CSP/nonce support

 Now works with any native Vite Env plugin

 Zero-JS: any server handler can render!

8:02 PM · Sep 23, 2025 

 1.6K  Reply  Copy link

[Read 64 replies](#)

We got that with TanStack Start. v1 just reached release candidate. We've been using it for a few things since early this year. It's been great.

## Let's try a live demo

basic react island

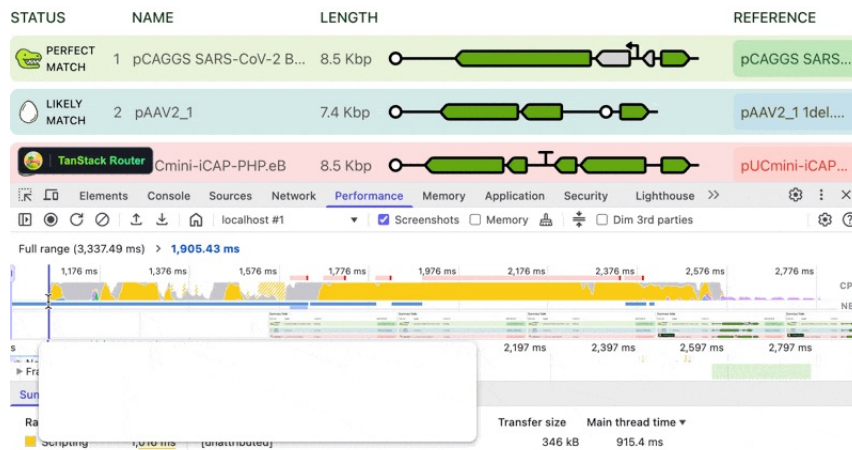
<https://plasmidsaurus.com/order/results/N76NH2>

tanstack start lambda

[https://lwnkivtnocgict3y2hf7qcvnhi0tidyq.lambda-url.us-west-2.on.aws/iframes/plasmid\\_demo](https://lwnkivtnocgict3y2hf7qcvnhi0tidyq.lambda-url.us-west-2.on.aws/iframes/plasmid_demo)



## Summary table



Here you can see what's going on this is from before I fixed a bug with lazy intersect observer. The page loads fully rendered as static HTML then without additional API requests, you get the visualizations after hydration runs and this becomes a normal React app.

It's all the same code as the react islands version. You don't need to worry about any of this when building, TanStack Start handles the details.

## Shared server/client query cache 🏆

Here's the magic: a server function that fetches data from an API, prewarms the query cache, and your UI components can behave as normal. No need to know what's going on.

```
// This function always runs on the node server
const getItemData = createServerFn()
  .validator((data: unknown) => ItemCode.parse(data))
  .handler(async (ctx) => {
    const itemCode = ctx.data.item_code;
    const response = await fetch(
      `${process.env.API_SERVER_URL}/api/v1/...`,
      {
        headers: {
          Authorization: `Bearer
${process.env.API_SERVER_TOKEN}`,
        },
      },
    );
    const data = await response.json();
    return Item.parse(data);
  });
```

```
export const Route = createFileRoute("/iframes/plasmid_demo")({
  component: RouteComponent,
  loader: async ({ context }) => {
    const item = await getItemData({
      data: { item_code: "N76NH2" },
    });

    context.queryClient.ensureQueryData({
      queryKey: ['item', 'N76NH2']
      queryFn: () => item
    })

    return { item };
  },
});
```

# Thank you

