Published in Better Programming

Hajime Yamasaki Vukelic Follow

Mar 26 · 6 min read · Listen

Poor man's islands architecture

How to implement a simple component islands loader in plain JS

Photo by Sam Deng on Unsplash

Sometimes you don't need much to enjo ice island surrounded by the ocean of static content.

The islands architecture is all the rave now. Some believe it's the answer to all questions and a solution to all issues. Right now, I'm working on a project of learning and comparing six of the more talked-about frameworks (I'll write about that once I'm done), and one of them is an implementation of the islands architecture. Since I'm new to that topic, I decided to read about it. As part of my effort to understand the concept, I wrote a simple component islands loader, and I'll share it with you here.

If you're new to the islands architecture, seeing the loader code may help you understand what it's about and how it works. Many authors make it sounds like it's rocket surgery, but it's really a simple (and effective) concept that shouldn't really lose its value just because it can be explained simply.

UPDATE: The loader code was made shorter and simpler by using the dynamic `import()`.

Islands of interactive UI bits

The idea behind the islands architecture is that you have a page with mixed static and highly interactive content, but the code for the interactive bits is loaded ad-hoc instead of all at once, speeding up the time-to-interactive.

As is common nowadays, developers use lots and lots of code for the interactive content, and the bundle sizes are getting close to half a megabyte *on average*. SSR alone does not address this issue because the bundle of JavaScript still needs to be shipped client-side to make the SSR'd content interactive again.

The islands architecture splits the page into blocks of interactive and non-interactive content, uses SSR to stitch them back together. Unlike micro-frontends or more traditional web apps, it leaves the code for the interactive components separate. The client-side *loader* code then loads the code for the interactive components *as needed*.

The "as needed" part is defined variably by different authors ranging from "visible in the viewport" to "having a chance of interaction". Regardless, the key takeaway is that it generally only works well if the page has lots of components that are *not* used immediately. In a truly desktop-application-style UI where everything is pretty much in the viewport, the benefits of using the islands architecture are slim to none.

Still, modern designs are really more like web pages than real apps, so there are plenty of scenarios where the islands architecture can shine.

Islands on the cheap

Sure there are frameworks for making islands-enabled pages. They do a lot more than just enable you to do the islands stuff. But if you just want to implement the islands architecture and you handle SSR using a more traditional SSR method (e.g., PHP, C#, Java, Python, NodeJS, WordPress, etc.) or even just a static site generator, the islands can actually be had for relatively cheap without any frameworks.

Let's write the loader script first. I create a file called `loader.js` and link it to the HTML like so:

```
<head>
  <!-- .... -->
  <script defer src="loader.js"></script>
</head>
```

The script itself starts off with the loader function:

```
{
let load = $ => import($.dataset.module || `/components/${$.tagName.toLowerCase()}.js`)
}
```

The braces around the code are just a block. It isolates variables defined within it from the global scope. There's nothing special about the `$` character in the variable name, it's just a habit I have had since the jQuery days to mark variables that point to DOM nodes.

The `load()` function loads the island code. It takes an element as its argument, and uses the element's lower-cased tag name as the name of the script. This is so that I can locate scripts using convention over configuration. If the element has a `data-module` attribute, it's used instead of the tag name.

UPDATE: In the old version of the code, the loader was creating a script tag and inserting into the `<head>`. I've later remembered that I can simply use the the dynamic `import()` to load the module. This makes the code a bit shorter which is a good thing for the loader, as it needs to be tiny. Note that the target file does not need to be an ES6 module. We are importing it for its side effects, as you will see later.

Next, we create an Intersection observer.

```
{
  // loader, etc. ...
let io = new IntersectionObserver(es => {
  for (let e of es) if (e.isIntersecting) {
    io.unobserve(e.target)
    load(e.target)
  }
})
}
```

This is the key part of the loader. It monitors registered nodes and as soon as they are within the viewport, they unhook the observer and load the code for the component.

The last bit of code scans the document for custom elements and adds them to the observer:

```
{
  // loader, observer, etc. ....
for (let $ of document.querySelectorAll('*'))
  if ($.tagName.includes('-')) io.observe($)
}
```

I used the `*` selector to have the script go through just about any element on the page. We could technically explicitly mark islands by using a custom attribute like `island` and select `[island]`. That would probably be a bit more efficient. This is good enough as a proof of concept.

It should be noted that the current version of the intersection observer does not track visibility. Firefox and Safari have still not adopted the v2, so we'll have to wait a bit before we can actually say "Don't load if not visible" (e.g., obscured by some other element or has opacity of 0).

Here's all of it in one piece. Under 0.4kb without any minification or gzipping.

```
{
let load = $ => import($.dataset.module || `/components/${$.tagName.toLowerCase()}.js`)
let io = new IntersectionObserver(es => {
  for (let e of es) if (e.isIntersecting) {
    io.unobserve(e.target)
```

```
      load(e.target)
    }
  }
  for (let $ of document.querySelectorAll('*'))
    if ($.tagName.includes('-')) io.observe($)
  }
```

Some parts of this are intentionally a bit terse. This is done to keep the payload small and avoid introducing a build step. The penalty we pay for this is a one-time effort to understand what the code does. Again, a *one-time* effort. I think it's worth it.

Creating components

Components are (by default) placed in the `components` directory. They should be named the same as the custom elements that they manage.

```
// components/test-island.js
{
  customElements.define('test-island', class extends HTMLElement {
    connectedCallback() {
      // Component logic goes here. `this` is the custom element.
      this.innerHTML = '<p>Hello, island!</p>'

      // Or maybe something like:
      //
      // ReactDOM.createRoot(this).render(<App/>)
    }
  }
}
```

The use of custom elements is not strictly intended for the actual component code. It's more about making it easier to acquire the node to which we want to attach the component logic. You can technically now render a React component into this node if you'd like.

The point is that, by doing it this way, we don't need to care about where on the page the component island is located, or how many *times* it is used, nor do we need to add initialization logic to the loader. It keeps things simpler and lighter.

The component script can also load its own stylesheets. This can be achieved by simply creating a `<link>` tag from within the script:

```
  document.head.append(Object.assign(document.createElement('link'), {
    rel: 'stylesheet',
    media: 'screen',
    href: 'components/test-island.css'
  }))
```

Alternatively, this logic can be moved to the loader. Moving the logic into the loader will be simpler in some cases because we can then avoid having to deal with relative URLs.

The SSR code or your static HTML files can now include the component anywhere in the page as a custom element:

```
<test-island>
  Some placeholder
</test-island>
```

The placeholder

The placeholder inside the custom element can simply be an indication that something will load (e.g., a spinner).

Or, even better, it could be a fully functional basic version of the functionality that doesn't require JavaScript and can work with the SSR server to facilitate the essential features. This is called progressive enhancement, and it's a natural fit for the SSR-enabled islands architecture. If the user has JavaScript disabled, the placeholder content is never replaced, so might as well make it work, too.

The beauty of the progressive enhancement approach is that, in a team setting, the basic version and the enhanced version can be worked on by different people or even different teams as they can be completely separate and independent.

Truly decoupled

Speaking of separate and independent, one advantage of the islands architecture is that the components can be developed and deployed completely separately from the main application. Different components can even have individual independent tech stacks. This is particularly useful for teams where people can't develop entire applications on their own for various reasons.

What about shared state or inter-component communication? Even when components need to share state or communicate with each other, this can be achieved using browser's built-in mechanisms such as sessionStorage, event buses, service workers, etc. This keeps the components truly decoupled and independent.

Software Architecture Web Development JavaScript Programming Software Development

---