# HTML Imports

#include for the web

**By** Eric Bidelman

**Published:** November 11th, 2013
**Updated:** December 18th, 2013
**Comments:** 72

**Your browser may not support the functionality in this article.**

## Why imports?

Think about how you load different types of resources on the web. For JS, we have

`<script src>`. For CSS, your go-to is probably `<link rel="stylesheet">`. For images it's `<img>`. Video has `<video>`. Audio, `<audio>`.... Get to the point! The majority of the web's content has a simple and declarative way to load itself. Not so for HTML. Here's your options:

1. `<iframe>` - tried and true but heavy weight. An iframe's content lives entirely in a separate context than your page. While that's mostly a great feature, it creates additional challenges (shrink wrapping the size of the frame to its content is tough, insanely frustrating to script into/out of, nearly impossible to style).
2. **AJAX** - I love `xhr.responseType="document"`, but you're saying I need JS to load HTML? That doesn't seem right.
3. **CrazyHacks™** - embedded in strings, hidden as comments (e.g. `<script type="text/html">`). Yuck!

See the irony? **The web's most basic content, HTML, requires the greatest amount of effort to work with**. Fortunately, Web Components are here to get us back on track.

## Getting started

HTML Imports, part of the Web Components cast, is a way to include HTML documents in other HTML documents. You're not limited to markup either. An import can also include CSS, JavaScript, or anything else an `.html` file can contain. In other words, this makes imports a **fantastic tool for loading related HTML/CSS/JS**.

## The basics

Include an import on your page by declaring a `<link rel="import">`:

```
<head>
  <link rel="import" href="/path/to/imports/stuff.html">
</head>
```

The URL of an import is called an *import location*. To load content from another domain, the import location needs to be CORS-enabled:

```
<!-- Resources on other origins must be CORS-enabled. -->
<link rel="import" href="http://example.com/elements.html">
```

*Fact!*

*The browser's network stack automatically de-dupes all requests from the same URL. This means that imports that reference the same URL are only retrieved once. No matter how many times an import at the same location is loaded, it only executes once.*

## Feature detection and support

To detect support, check if `.import` exists on the `<link>` element:

```
function supportsImports() {
  return 'import' in document.createElement('link');
}

if (supportsImports()) {
  // Good to go!
} else {
  // Use other libraries/require systems to load files.
}
```

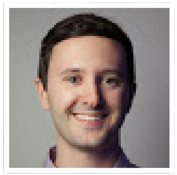
Browser support is still in the early days. Chrome 31 was the first browser to see an implementation. Since then, Chrome 36 was update with the latest spec. You can enable the flag by turning on **Enable experimental Web Platform features** in `about:flags` in Chrome Canary. For other browsers, Polymer's polyfill works great until things are widely supported.

Tip!

*Also **Enable experimental Web Platform features** to get the other bleeding edge web component goodies.*

## Bundling resources

Imports provide convention for bundling HTML/CSS/JS (even other HTML Imports) into a single deliverable. It's an intrinsic feature, but a powerful one. If you're creating a theme, library, or just want to segment your app into logical chunks, giving users a single URL is compelling. Heck, you could even deliver an entire app via an import. Think about that for a second.



*Using only one URL, you can package together a single relocatable bundle of web goodness for others to consume.*

A real-world example is Bootstrap. Bootstrap is comprised of individual files (bootstrap.css, bootstrap.js, fonts), requires JQuery for its plugins, and provides markup examples. Developers like à la carte flexibility. It allows them buy in to the parts of the framework *they* want to use. That said, I'd wager your typical JoeDeveloper™ goes the easy route and downloads all of Bootstrap.

Imports make a ton of sense for something like Bootstrap. I present to you, the future of loading Bootstrap:

```
<head>
  <link rel="import" href="bootstrap.html">
```

```
</head>
```

Users simply load an HTML Import link. They don't need to fuss with the scatter-shot of files. Instead, the entirety of Bootstrap is managed and wrapped up in an import, bootstrap.html:

```
<link rel="stylesheet" href="bootstrap.css">
<link rel="stylesheet" href="fonts.css">
<script src="jquery.js"></script>
<script src="bootstrap.js"></script>
<script src="bootstrap-tooltip.js"></script>
<script src="bootstrap-dropdown.js"></script>
...

<!-- scaffolding markup -->
<template>
  ...
</template>
```

Let this sit. It's exciting stuff.

## Load/error events

The `<link>` element fires a `load` event when an import is loaded successfully and `onerror` when the attempt fails (e.g. if the resource 404s).

Imports try to load immediately. An easy way avoid headaches is to use the `onload`/`onerror` attributes:

```
<script async>
  function handleLoad(e) {
    console.log('Loaded import: ' + e.target.href);
  }
  function handleError(e) {
    console.log('Error loading import: ' + e.target.href);
  }
</script>

<link rel="import" href="file.html"
      onload="handleLoad(event)" onerror="handleError(event)">
```

Tip!

*Notice the event handlers are defined before the import is loaded on the page. The browser tries to load the import as soon as it encounters the tag. If the functions don't exist yet, you'll get console errors for undefined function names.*

Or, if you're creating the import dynamically:

```
var link = document.createElement('link');
link.rel = 'import';
link.href = 'file.html'
link.onload = function(e) {...};
link.onerror = function(e) {...};
document.head.appendChild(link);
```

## Using the content

Including an import on a page doesn't mean "plop the content of that file here". It means "parser, go off an fetch this document so I can use it". To actually use the content, you have to take action and write script.

A critical `aha!` moment is realizing that an import is just a document. In fact, the content of an import is called an *import document*. You're able to **manipulate the guts of an import using standard DOM APIs**!

### link.import

To access the content of an import, use the link element's `.import` property:

```
var content = document.querySelector('link[rel="import"]').import;
```

`link.import` is `null` under the following conditions:

- The browser doesn't support HTML Imports.
- The `<link>` doesn't have `rel="import"`.
- The `<link>` has not been added to the DOM.
- The `<link>` has been removed from the DOM.
- The resource is not CORS-enabled.

### Full example

Let's say `warnings.html` contains:

```
<div class="warning">
  <style scoped>
    h3 {
      color: red;
    }
  </style>
  <h3>Warning!</h3>
  <p>This page is under construction</p>
</div>

<div class="outdated">
  <h3>Heads up!</h3>
  <p>This content may be out of date</p>
</div>
```

Importers can grab a specific portion of this document and clone it into their page:

```
<head>
  <link rel="import" href="warnings.html">
</head>
<body>
  ...
  <script>
    var link = document.querySelector('link[rel="import"]');
    var content = link.import;

    // Grab DOM from warning.html's document.
    var el = content.querySelector('.warning');

    document.body.appendChild(el.cloneNode(true));
  </script>
</body>
```

**LIVE DEMO:**

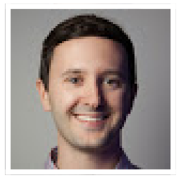
## Scripting in imports

Imports are not in the main document. They're satellite to it. However, your import can still act on the main page even though the main document reigns supreme. An import can access its own DOM and/or the DOM of the page that's importing it:

**Example -** import.html that adds one of its stylesheets to the main page

```
<link rel="stylesheet" href="http://www.example.com/styles.css">
<link rel="stylesheet" href="http://www.example.com/styles2.css">

<style>
  /* Note: <style> in an import apply to the main
     document by default. That is, style tags don't need to be
     explicitly added to the main document. */
  #somecontainer {
    color: blue;
  }
</style>
...

<script>
  // importDoc references this import's document
  var importDoc = document.currentScript.ownerDocument;

  // mainDoc references the main document (the page that's importing
us)
  var mainDoc = document;

  // Grab the first stylesheet from this import, clone it,
  // and append it to the importing document.
  var styles = importDoc.querySelector('link[rel="stylesheet"]');
  mainDoc.head.appendChild(styles.cloneNode(true));
</script>
```

Notice what's going on here. The script inside the import references the imported document (`document.currentScript.ownerDocument`), and appends part of that document to the importing page (`mainDoc.head.appendChild(...)`). Pretty gnarly if you ask me.



*A script in an import can either execute code directly, or define functions to be used by the importing page. This is similar to the way modules are defined in Python.*

Rules of JavaScript in an import:

- Script in the import is executed in the context of the window that contains the importing `document`. So `window.document` refers to the main page document. This has two useful corollaries:
    - functions defined in an import end up on `window`.
    - you don't have to do anything crazy like append the import's `<script>` blocks to the main page. Again, script gets executed.

- Imports do not block parsing of the main page. However, scripts inside them are processed in order. This means you get defer-like behavior while maintaining proper script order. More on this below.

## Delivering Web Components

The design of HTML Imports lends itself nicely to loading reusable content on the web. In particular, it's an ideal way to distribute Web Components. Everything from basic HTML `<template>`s to full blown Custom Elements with Shadow DOM [1, 2, 3]. When these technologies are used in tandem, imports become a `#include` for Web Components.

### Including templates

The HTML Template element is a natural fit for HTML Imports. `<template>` is great for scaffolding out sections of markup for the importing app to use as it desires. Wrapping content in a `<template>` also gives you the added benefit of making the content inert until used. That is, scripts don't run until the template is added to the DOM). Boom!

import.html

```
<template>
  <h1>Hello World!</h1>
  <!-- Img is not requested until the <template> goes live. -->
  <img src="world.png">
  <script>alert("Executed when the template is activated.");</script>
</template>
```

index.html

```
<head>
  <link rel="import" href="import.html">
</head>
<body>
  <div id="container"></div>
  <script>
    var link = document.querySelector('link[rel="import"]');

    // Clone the <template> in the import.
    var template = link.import.querySelector('template');
    var clone = document.importNode(template.content, true);

    document.querySelector('#container').appendChild(clone);
  </script>
</body>
```

## Registering custom elements

Custom Elements is another Web Component technology that plays absurdly well with HTML Imports. Imports can execute script, so why not define + register your custom elements so users don't have to? Call it..."auto-registration".

elements.html

```
<script>
  // Define and register <say-hi>.
  var proto = Object.create(HTMLElement.prototype);

  proto.createdCallback = function() {
    this.innerHTML = 'Hello, <b>' +
                     (this.getAttribute('name') || '?') + '</b>';
  };

  document.registerElement('say-hi', {prototype: proto});
</script>

<template id="t">
  <style>
    ::content > * {
      color: red;
    }
  </style>
  <span>I'm a shadow-element using Shadow DOM!</span>
  <content></content>
</template>

<script>
  (function() {
    var importDoc = document.currentScript.ownerDocument; // importee
    // Define and register <shadow-element>
    // that uses Shadow DOM and a template.
    var proto2 = Object.create(HTMLElement.prototype);

    proto2.createdCallback = function() {
      // get template in import
      var template = importDoc.querySelector('#t');

      // import template into
      var clone = document.importNode(template.content, true);

      var root = this.createShadowRoot();
      root.appendChild(clone);
    };

    document.registerElement('shadow-element', {prototype: proto2});
  })();
</script>
```

This import defines (and registers) two elements, `<say-hi>` and `<shadow-element>`. The first shows a basic custom element that registers itself inside the import. The second example shows how to implement a custom element that creates Shadow DOM from a `<template>`, then registers itself.

The best part about registering custom elements inside an HTML import is that the the importer simply declares your element on their page. No wiring needed.

index.html

```html
<head>
  <link rel="import" href="elements.html">
</head>
<body>
  <say-hi name="Eric"></say-hi>
  <shadow-element>
    <div>( I'm in the light dom )</div>
  </shadow-element>
</body>
```

**LIVE DEMO:**

**LIVE DEMO:**

( I'm in the light dom )

In my opinion, this workflow alone makes HTML Imports an ideal way to share Web Components.

## Managing dependencies and sub-imports

*Yo dawg. I hear you like imports, so I included an import in your import.*

## Sub-imports

It can be useful for one import to include another. For example, if you want to reuse or extend another component, use an import to load the other element(s).

Below is a real example from Polymer. It's a new tab component (`<polymer-ui-tabs>`) that reuses a layout and selector component. The dependencies are managed using HTML Imports.

polymer-ui-tabs.html

```
<link rel="import" href="polymer-selector.html">
<link rel="import" href="polymer-flex-layout.html">

<polymer-element name="polymer-ui-tabs" extends="polymer-selector"
...>
  <template>
    <link rel="stylesheet" href="polymer-ui-tabs.css">
    <polymer-flex-layout></polymer-flex-layout>
    <shadow></shadow>
  </template>
</polymer-element>
```

full source

App developers can import this new element using:

```
<link rel="import" href="polymer-ui-tabs.html">
<polymer-ui-tabs></polymer-ui-tabs>
```

When a new, more awesome `<polymer-selector2>` comes along in the future, you can swap out `<polymer-selector>` and start using it straight away. You won't break your users thanks to imports and web components.

## Dependency management

We all know that loading JQuery more than once per page causes errors. Isn't this going to be a *huge* problem for Web Components when multiple components use the same library? Not if we use HTML Imports! They can be used to manage dependencies.

By wrapping libraries in an HTML Import, you automatically de-dupe resources. The document is only parsed once. Scripts are only executed once. As an example, say you define an import, jquery.html, that loads a copy of JQuery.

jquery.html

```
<script src="http://cdn.com/jquery.js"></script>
```

This import can be reused in subsequent imports like so:

import2.html

```
<link rel="import" href="jquery.html">
<div>Hello, I'm import 2</div>
```

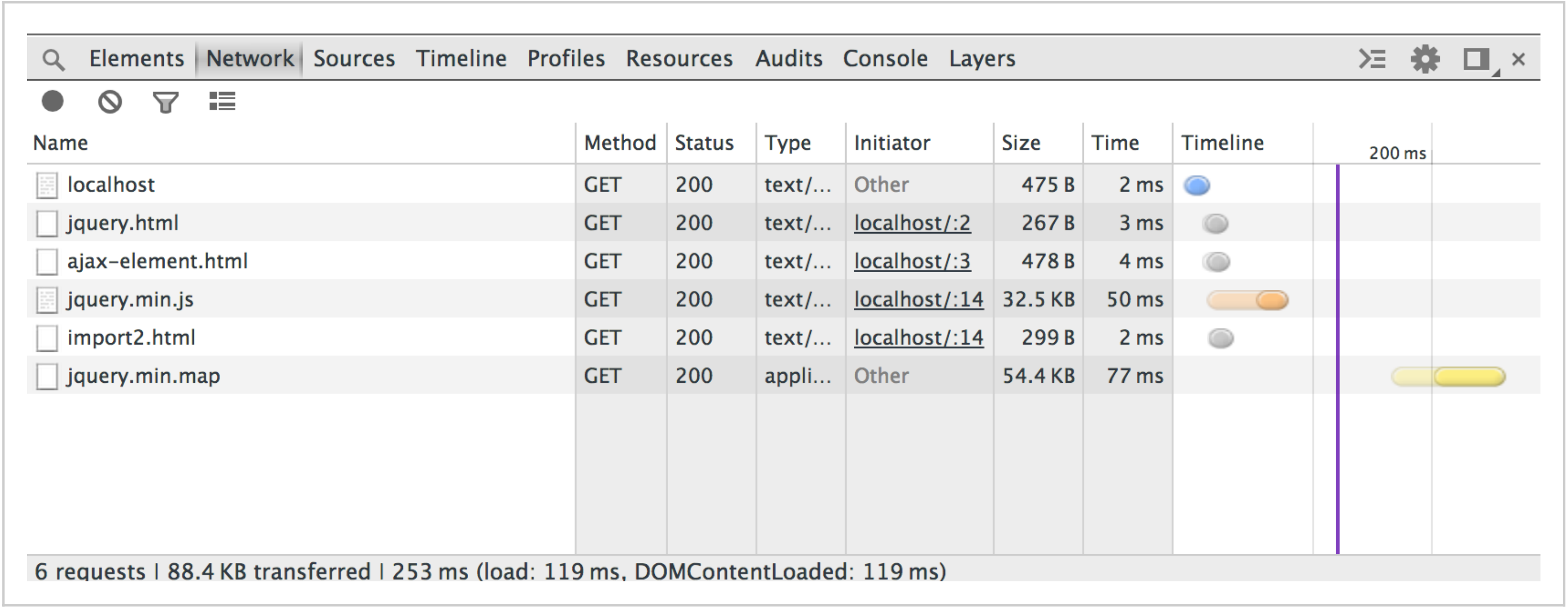
ajax-element.html

```
<link rel="import" href="jquery.html">
<link rel="import" href="import2.html">

<script>
  var proto = Object.create(HTMLElement.prototype);

  proto.makeRequest = function(url, done) {
    return $.ajax(url).done(function() {
      done();
    });
  };

  document.registerElement('ajax-element', {prototype: proto});
</script>
```

Even the main page itself can include jquery.html if it needs the library:

```html
<head>
  <link rel="import" href="jquery.html">
  <link rel="import" href="ajax-element.html">
</head>
<body>

...

<script>
  $(document).ready(function() {
    var el = document.createElement('ajax-element');
    el.makeRequest('http://example.com');
  });
</script>
</body>
```

Despite jquery.html being included in many different import trees, it's document is only fetched and processed once by the browser. Examining the network panel proves this:


Elements Network Sources Timeline Profiles Resources Audits Console Layers

| Name | Method | Status | Type | Initiator | Size | Time |
|---|---|---|---|---|---|---|
| localhost | GET | 200 | text/... | Other | 475 B | 2 ms |
| jquery.html | GET | 200 | text/... | localhost/:2 | 267 B | 3 ms |
| ajax-element.html | GET | 200 | text/... | localhost/:3 | 478 B | 4 ms |
| jquery.min.js | GET | 200 | text/... | localhost/:14 | 32.5 KB | 50 ms |
| import2.html | GET | 200 | text/... | localhost/:14 | 299 B | 2 ms |
| jquery.min.map | GET | 200 | appli... | Other | 54.4 KB | 77 ms |

6 requests | 88.4 KB transferred | 253 ms (load: 119 ms, DOMContentLoaded: 119 ms)

***jquery.html is requested once***

## Performance considerations

HTML Imports are totally awesome but as with any new web technology, you should use them wisely. Web development best practices still hold true. Below are some things to keep in mind.

### Concatenate imports

Reducing network requests is always important. If you have many top-level import links, consider combining them into a single resource and importing that file!

Vulcanize is an npm build tool from the Polymer team that recursively flattens a set of HTML Imports into a single file. Think of it as a concatenation build step for Web Components.

## Imports leverage browser caching

Many people forget that the browser's networking stack has been finely tuned over the years. Imports (and sub-imports) take advantage of this logic too. The `http://cdn.com/bootstrap.html` import might have sub-resources, but they'll be cached.

## Content is useful only when you add it

Think of content as inert until you call upon its services. Take a normal, dynamically created stylesheet:

```
var link = document.createElement('link');
link.rel = 'stylesheet';
link.href = 'styles.css';
```

The browser won't request styles.css until `link` is added to the DOM:

```
document.head.appendChild(link); // browser requests styles.css
```

Another example is dynamically created markup:

```
var h2 = document.createElement('h2');
h2.textContent = 'Booyah!';
```

The `h2` is relatively meaningless until you add it to the DOM.

The same concept holds true for the import document. Unless you append it's content to the DOM, it's a no-op. In fact, the only thing that "executes" in the import document directly is `<script>`. See scripting in imports.

## Optimizing for async loading

### Imports block rendering

*Imports block rendering of the main page*. This is similar to what `<link rel="stylesheet">` do. The reason the browser blocks rendering on stylesheets in the first place is to minimize FOUC. Imports behave similarly because they can contain stylsheets.

To be completely asynchronous and not block the parser or rendering, use the `async` attribute:

```
<link rel="import" href="/path/to/import_that_takes_5secs.html" async>
```

The reason `async` isn't the default for HTML Imports is because it requires developers to do more work. Synchronous by default means that HTML Imports that have custom element definitions inside of them are guaranteed to load and upgrade, in order. In a completely async world, developers would have to manage that dance and upgrade timings themselves.

## Imports do not block parsing

*Imports don't block parsing of the main page*. Scripts inside imports are processed in order but don't block the importing page. This means you get defer-like behavior while maintaining proper script order. One benefit of putting your imports in the `<head>` is that it lets the parser start working on the content as soon as possible. With that said, it's critical to remember `<script>` in the main document *still* continues to block the page. The first `<script>` after an import will block page rendering. That's because an import can have script inside that needs to be executed before the script in the main page.

```
<head>
```

```
  <link rel="import" href="/path/to/import_that_takes_5secs.html">
  <script>console.log('I block page rendering');</script>
</head>
```

Depending on your app structure and use case, there are several ways to optimize async behavior. The techniques below mitigate blocking the main page rendering.

**Scenario #1 (preferred): you don't have script in `<head>` or inlined in `<body>`**

My recommendation for placing `<script>` is to avoid immediately following your imports. Move scripts as late in the game as possible...but you're already doing that best practice, AREN'T YOU!? ;)

Here's an example:

```
<head>
  <link rel="import" href="/path/to/import.html">
  <link rel="import" href="/path/to/import2.html">
  <!-- avoid including script -->
</head>
<body>
  <!-- avoid including script -->

  <div id="container"></div>

  <!-- avoid including script -->
  ...

  <script>
    // Other scripts n' stuff.

    // Bring in the import content.
    var link = document.querySelector('link[rel="import"]');
    var post = link.import.querySelector('#blog-post');

    var container = document.querySelector('#container');
    container.appendChild(post.cloneNode(true));
  </script>
</body>
```

Everything is at the bottom.

### Scenario 1.5: the import adds itself

Another option is to have the import add its own content. If the import author establishes a contract for the app developer to follow, the import can add itself to an area of the main page:

import.html:

```
<div id="blog-post">...</div>
<script>
  var me = document.currentScript.ownerDocument;
  var post = me.querySelector('#blog-post');

  var container = document.querySelector('#container');
  container.appendChild(post.cloneNode(true));
</script>
```

index.html

```
<head>
  <link rel="import" href="/path/to/import.html">
</head>
<body>
  <!-- no need for script. the import takes care of things -->
</body>
```

### Scenario #2: you *have* script in `<head>` or inlined in `<body>`

If you have an import that takes a long time to load, the first `<script>` that follows it on the page will block the page from rendering. Google Analytics for example, recommends putting the tracking code in the `<head>`, If you can't avoid putting `<script>` in the `<head>`, dynamically adding the import will prevent blocking the page:

```
<head>
  <script>
```

```
    function addImportLink(url) {
      var link = document.createElement('link');
      link.rel = 'import';
      link.href = url;
      link.onload = function(e) {
        var post = this.import.querySelector('#blog-post');

        var container = document.querySelector('#container');
        container.appendChild(post.cloneNode(true));
      };
      document.head.appendChild(link);
    }

    addImportLink('/path/to/import.html'); // Import is added early :)
  </script>
  <script>
    // other scripts
  </script>
</head>
<body>
   <div id="container"></div>
   ...
</body>
```

Alternatively, add the import near the end of the `<body>`:

```

<head>
```

```
  <script>
    // other scripts
  </script>
</head>
<body>
  <div id="container"></div>
  ...

  <script>
    function addImportLink(url) { ... }

    addImportLink('/path/to/import.html'); // Import is added very
late :(
  </script>
</body>
```

***Note:** This very last approach is least preferable. The parser doesn't start to work on the import content until late in the page.*

## Things to remember

- An import's mimetype is `text/html`.
- Resources from other origins need to be CORS-enabled.

- Imports from the same URL are retrieved and parsed once. That means script in an import is only executed the first time the import is seen.
- Scripts in an import are processed in order, but do not block the main document parsing.
- An import link doesn't mean "#include the content here". It means "parser, go off an fetch this document so I can use it later". While scripts execute at import time, stylesheets, markup, and other resources need to be added to the main page explicitly. Note, `<style>` don't need to be added explicitly. This is a major difference between HTML Imports and `<iframe>`, which says "load and render this content here".

## Conclusion

HTML Imports allow bundling HTML/CSS/JS as a single resource. While useful by themselves, this idea becomes extremely powerful in the world of Web Components. Developers can create reusable components for others to consume and bring in to their own app, all delivered through `<link rel="import">`.

HTML Imports are a simple concept, but enable a number of interesting use cases for the platform.

## Use cases

- **Distribute** related HTML/CSS/JS as a single bundle. Theoretically, you could import an entire web app into another.
- **Code organization** - segment concepts logically into different files, encouraging modularity & reusability**.
- **Deliver** one or more Custom Element definitions. An import can be used to register and include them in an app. This practices good software patterns, keeping the element's interface/definition separate from how its used.
- **Manage dependencies** - resources are automatically de-duped.
- **Chunk scripts** - before imports, a large-sized JS library would have its file wholly parsed in order to start running, which was slow. With imports, the library can start working as soon as chunk A is parsed. Less latency!

  `<link rel="import" href="chunks.html">`:

```
<script>/* script chunk A goes here */</script>
<script>/* script chunk B goes here */</script>
<script>/* script chunk C goes here */</script>
...
```

- **Parallelizes HTML parsing** - first time the browser has been able to run two (or more) HTML parsers in parallel.

- **Enables switching between debug and non-debug modes** in an app, just by changing the import target itself. Your app doesn't need to know if the import target is a bundled/compiled resource or an import tree.

There are 72 comments. Want to add yours?

Share

Subscribe

## Next steps

Enjoyed this article? Grab the RSS feed and stay up-to-date.