From object to iframe — other embedding technologies

Previous Overview: Multimedia and embedding Next

By now you should really be getting the hang of embedding things into your web pages, including images, video and audio. At this point we'd like to take somewhat of a sideways step, looking at some elements that allow you to embed a wide variety of content types into your webpages: the `<iframe>`, `<embed>` and `<object>` elements. `<iframe>`s are for embedding other web pages, and the other two allow you to embed PDFs, SVG, and even Flash — a technology that is on the way out, but which you'll still see semi-regularly.

Prerequisites:	Basic computer literacy, basic software installed, basic knowledge of working with files, familiarity with HTML fundamentals (as covered in Getting started with HTML) and the previous articles in this module.
Objective:	To learn how to embed items into web pages using `<object>`, `<embed>`, and `<iframe>`, like Flash movies and other webpages.

A short history of embedding

A long time ago on the Web, it was popular to use **frames** to create websites — small parts of a website stored in individual HTML pages. These were embedded in a master document called a **frameset**, which allowed you to specify the area on the screen that each frame filled, rather like sizing the columns and rows of a table. These were considered the height of coolness in the mid to late 90s, and there was evidence that having a webpage split up into smaller chunks like this was better for download speeds — especially noticeable with network connections being so slow back then. They did however have many problems, which far outweighed any positives as network speeds got faster, so you don't see them being used anymore.

A little while later (late 90s, early 2000s), plugin technologies became very popular, such as Java Applets and Flash — these allowed web developers to embed rich content into webpages such as video and animations, which just weren't available through HTML alone. Embedding these technologies was achieved through elements like `<object>`, and the lesser-used `<embed>`, and they were very useful at the time. They have since fallen out of fashion due to many problems, including accessibility, security, file size, and more; these days most mobile devices don't support such plugins anymore, and desktop support is on the way out.

Finally, the `<iframe>` element appeared (along with other ways of embedding

content, such as `<canvas>`, `<video>`, etc.) This provides a way to embed an entire web document inside another one, as if it were an `<img>` or other such element, and is used regularly today.

With the history lesson out of the way, let's move on and see how to use some of these.

Active learning: classic embedding uses

In this article we are going to jump straight into an active learning section, to immediately give you a real idea of just what embedding technologies are useful for. The online world is very familiar with Youtube, but many people don't know about some of the sharing facilities it has available. Let's look at how Youtube allows us to embed a video in any page we like using an `<iframe>`.

1. First, go to Youtube and find a video you like.
2. Below the video, you'll find a *Share* button — select this to display the sharing options.
3. Select the *Embed* button and you'll be given some `<iframe>` code — copy this.
4. Insert it into the *Input* box below, and see what the result is in the *Output*.

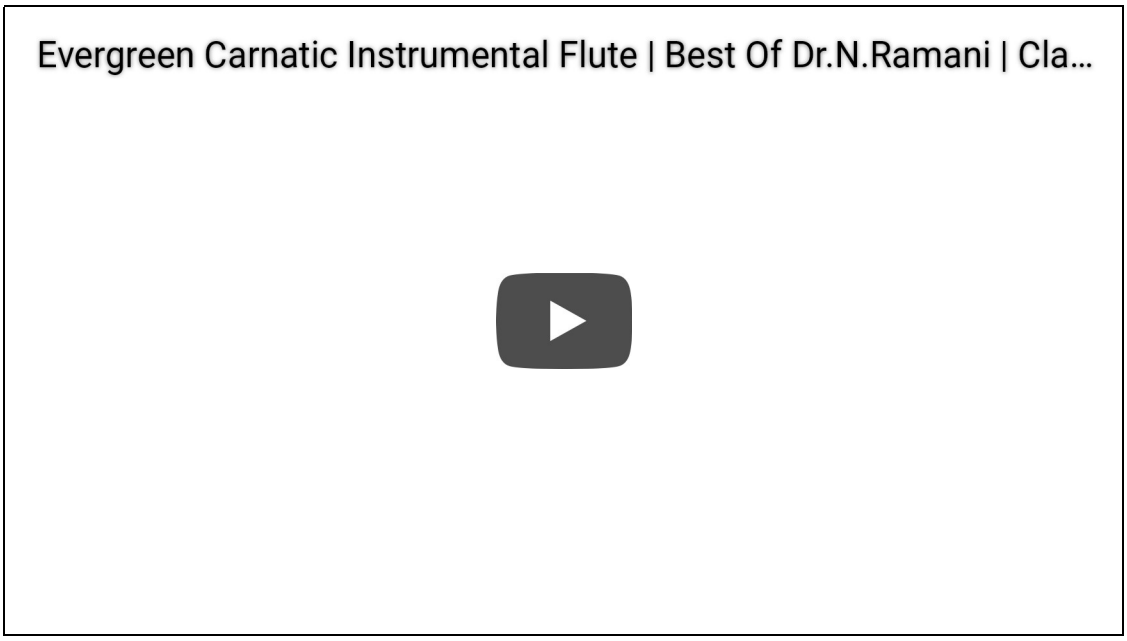
For bonus points, you could also try embedding a Google Map in the example:

1. Go to Google Maps and find a map you like.
2. Click on the "Hamburger Menu" (three horizontal lines) in the top left of the UI.
3. Select the *Share or embed map* option.
4. Select the Embed map option, which will give you some `<iframe>` code — copy this.
5. Insert it into the *Input* box below, and see what the result is in the *Output*.

If you make a mistake, you can always reset it using the *Reset* button. If you get

really stuck, press the *Show solution* button to see an answer.

Live output



Editable code

Press Esc to move focus away from the code area (Tab inserts a tab character).

```
<iframe width="560" height="315" src="https://www.youtube.com/embed
/IhfsKWLDf7A" frameborder="0" allow="autoplay; encrypted-media"
allowfullscreen></iframe>
```

Reset Show solution

Iframes in detail

So, that was easy and fun, right? `<iframe>` elements are designed to allow you to embed other web documents into the current document. This is great for incorporating third-party content into your website that you might not have direct control over and don't want to have to implement your own version of — such as video from online video providers, commenting systems like Disqus, maps from

online map providers, advertising banners, etc. The live editable examples you've been using through this course are implemented using `<iframe>`s.

There are some serious Security concerns to consider with `<iframe>`s, as we'll discuss below, but this doesn't mean that you shouldn't use them in your websites — it just requires some knowledge and careful thinking. Let's explore the code in a bit more detail. Say you wanted to include the MDN glossary on one of your web pages — you could try something like this:

```
<iframe src="https://developer.mozilla.org/en-US/docs/Glossary"
        width="100%" height="500" frameborder="0"
        allowfullscreen sandbox>
  <p> <a href="https://developer.mozilla.org/en-US/docs/Glossary">
    Fallback link for browsers that don't support iframes
  </a> </p>
</iframe>
```

This example includes the basic essentials needed to use an `<iframe>`:

`allowfullscreen`
If set, the `<iframe>` is able to be placed in fullscreen mode using the Full Screen API (somewhat beyond scope for this article.)

`frameborder`
If set to 1, this tells the browser to draw a border between this frame and other frames, which is the default behaviour. 0 removes the border. Using this isn't really recommended any more, as the same effect can be better achieved using `border: none;` in your CSS.

`src`
This attribute, as with `<video>`/`<img>`, contains a path pointing to the URL of the document to be embedded.

`width` and `height`
These attributes specify the width and height you want the iframe to be.

Fallback content

In the same way as other similar elements like `<video>`, you can include fallback content between the opening and closing `<iframe></iframe>` tags that will appear if the browser doesn't support the `<iframe>`. In this case, we have included a link to the page instead. It is unlikely that you'll come across any browser that doesn't support `<iframe>`s these days.

`sandbox`

This attribute, which works in slightly more modern browsers than the rest of the `<iframe>` features (e.g. IE 10 and above) requests heightened security settings; we'll say more about this in the next section.

> **Note**: In order to improve speed, it's a good idea to set the iframe's `src` attribute with JavaScript after the main content is done with loading. This makes your page usable sooner and decreases your official page load time (an important SEO metric.)

Security concerns

Above we mentioned security concerns — let's go into this in a bit more detail now. We are not expecting you to understand all of this content perfectly the first time; we just want to make you aware of this concern, and provide a reference to come back to as you get more experienced and start considering using `<iframe>`s in your experiments and work. Also, there is no need to be scared and not use `<iframe>`s — you just need to be careful. Read on...

Browser makers and Web developers have learned the hard way that iframes are a common target (official term: **attack vector**) for bad people on the Web (often termed **hackers**, or more accurately, **crackers**) to attack if they are trying to maliciously modify your webpage, or trick people into doing something they don't want to do, such as reveal sensitive information like usernames and passwords. Because of this, spec engineers and browser developers have developed various security mechanisms for making `<iframe>`s more secure, and there are also best practices to consider — we'll cover some of these below.

> Clickjacking is one kind of common iframe attack where hackers embed an

> invisible iframe into your document (or embed your document into their own malicious website) and use it to capture users' interactions. This is a common way to mislead users or steal sensitive data.

A quick example first though — try loading the previous example we showed above into your browser — you can find it live on Github (see the source code too.) You won't actually see anything displayed on the page, and if you look at the *Console* in the browser developer tools, you'll see a message telling you why. In Firefox, you'll get told *Load denied by X-Frame-Options: https://developer.mozilla.org/en-US/docs/Glossary does not permit framing*. This is because the developers that built MDN have included a setting on the server that serves the website pages to disallow them from being embedded inside `<iframe>`s (see Configure CSP directives, below.) This makes sense — an entire MDN page doesn't really make sense to be embedded in other pages unless you want to do something like embed them on your site and claim them as your own — or attempt to steal data via clickjacking, which are both really bad things to do. Plus if everybody started to do this, all the additional bandwidth would start to cost Mozilla a lot of money.

Only embed when necessary

Sometimes it makes sense to embed third-party content — like youtube videos and maps — but you can save yourself a lot of headaches if you only embed third-party content when completely necessary. A good rule of thumb for web security is *"You can never be too cautious. If you made it, double-check it anyway. If someone else made it, assume it's dangerous until proven otherwise."*

Besides security, you should also be aware of intellectual property issues. Most content is copyrighted, offline and online, even content you might not expect (for example, most images on Wikimedia Commons). Never display content on your webpage unless you own it or the owners have given you written, unequivocal permission. Penalties for copyright infringement are severe. Again, you can never be too cautious.

If the content is licensed, you must obey the license terms. For example, the content on MDN is licensed under CC-BY-SA. That means, you must credit us

properly when you quote our content, even if you make substantial changes.

Use HTTPS

HTTPS is the encrypted version of HTTP. You should serve your websites using HTTPS whenever possible:

1. HTTPS reduces the chance that remote content has been tampered with in transit,
2. HTTPS prevents embedded content from accessing content in your parent document, and vice versa.

Using HTTPS requires a security certificate, which can be expensive (although Let's Encrypt makes things easier) — if you can't get one, you may serve your parent document with HTTP. However, because of the second benefit of HTTPS above, *no matter what the cost, you must never embed third-party content with HTTP.* (In the best case scenario, your user's Web browser will give them a scary warning.) All reputable companies that make content available for embedding via an `<iframe>` will make it available via HTTPS — look at the URLs inside the `<iframe> src` attribute when you are embedding content from Google Maps or Youtube, for example.

> **Note**: Github pages allow content to be served via HTTPS by default, so is useful for hosting content. If you are using different hosting and are not sure, ask your hosting provider about it.

Always use the `sandbox` attribute

You want to give attackers as little power as you can to do bad things on your website, therefore you should give embedded content *only the permissions needed for doing its job.* Of course, this applies to your own content, too. A container for code where it can be used appropriately — or for testing — but can't cause any harm to the rest of the codebase (either accidental or malicious) is called a sandbox.

Unsandboxed content can do way too much (executing JavaScript, submitting forms, popup windows, etc.) By default, you should impose all available restrictions by using the `sandbox` attribute with no parameters, as shown in our previous example.

If absolutely required, you can add permissions back one by one (inside the `sandbox=""` attribute value) — see the `sandbox` reference entry for all the available options. One important note is that you should *never* add both `allow-scripts` and `allow-same-origin` to your `sandbox` attribute — in that case, the embedded content could bypass the same origin security policy that stops sites from executing scripts, and use JavaScript to turn off sandboxing altogether.

> **Note**: Sandboxing provides no protection if attackers can fool people into visiting malicious content directly (outside an `iframe`). If there's any chance that certain content may be malicious (e.g., user-generated content), please serve it from a different domain to your main site.

Configure CSP directives

CSP stands for **content security policy** and provides a set of HTTP Headers (metadata sent along with your web pages when they are served from a web server) designed to improve the security of your HTML document. When it comes to securing `<iframe>`s, you can *configure your server to send an appropriate `X-Frame-Options` header.* This can prevent other websites from embedding your content in their web pages (which would enable clickjacking and a host of other attacks), which is exactly what the MDN developers have done, as we saw earlier on.

> **Note**: You can read Frederik Braun's post On the X-Frame-Options Security Header for more background information on this topic. Obviously, it's rather out of scope for a full explanation in this article.

The <embed> and <object> elements

The `<embed>` and `<object>` elements serve a different function to `<iframe>` — these elements are general purpose embedding tools for embedding multiple types of external content, which include plugin technologies like Java Applets and Flash, PDF (which can be shown in a browser with a PDF plugin), and even content like videos, SVG and images!

> **Note**: A **plugin** is a software that provides access to content the browser cannot read natively.

However, you are unlikely to use these elements very much — Applets haven't been used for years, Flash is no longer very popular, due to a number of reasons (see The case against plugins, below), PDFs tend to be better linked to than embedded, and other content such as images and video have much better, easier elements to handle those. Plugins and these embedding methods are really a legacy technology, and we are mainly mentioning them in case you come across them in certain circumstances like intranets, or enterprise projects.

If you find yourself needing to embed plugin content, this is the kind of information you'll need, at a minimum:

	`<embed>`	`<object>`
URL of the embedded content	`src`	`data`
accurate media type of the embedded content	`type`	`type`
height and width (in CSS pixels) of the box controlled by the plugin	`height` `width`	`height` `width`
names and values, to feed the plugin as parameters	ad hoc attributes with those names and values	single-tag `<param>` elements, contained within `<object>`

	`<embed>`	`<object>`
independent HTML content as fallback for an unavailable resource	not supported (`<noembed>` is obsolete)	contained within `<object>`, after `<param>` elements

> **Note**: `<object>` requires a `data` attribute, a `type` attribute, or both. If you use both, you may also use the `typemustmatch` attribute (only implemented in Firefox, as of this writing). `typemustmatch` keeps the embedded file from running unless the `type` attribute provides the correct media type. `typemustmatch` can therefore confer significant security benefits when you're embedding content from a different origin (it can keep attackers from running arbitrary scripts through the plugin).

Here's an example that uses the `<embed>` element to embed a Flash movie (see this live on Github, and check the source code too):

```
<embed src="whoosh.swf" quality="medium"
       bgcolor="#ffffff" width="550" height="400"
       name="whoosh" align="middle" allowScriptAccess="sameDomain"
       allowFullScreen="false" type="application/x-shockwave-flash"
       pluginspage="http://www.macromedia.com/go/getflashplayer">
```

Pretty horrible, isn't it. The HTML generated by the Adobe Flash tool tended to be even worse, using an `<object>` element with an `<embed>` element embedded in it, to cover all bases (check out an example.) Flash was even used successfully as fallback content for HTML5 video, for a time, but this is increasingly being seen as not necessary.

Now let's look at an `<object>` example that embeds a PDF into a page (see the live example and the source code):

```
<object data="mypdf.pdf" type="application/pdf"
        width="800" height="1200" typemustmatch>

```

```
    <p>You don't have a PDF plugin, but you can <a href="mypdf.pdf">download
</object>
```

PDFs were a necessary stepping stone between paper and digital, but they pose many accessibility challenges and can be hard to read on small screens. They do still tend to be popular in some circles, but it is much better to link to them so they can be downloaded or read on a separate page, rather than embedding them in a webpage.

The case against plugins

Once upon a time, plugins were indispensable on the Web. Remember the days when you had to install Adobe Flash Player just to watch a movie online? And then you constantly got annoying alerts about updating Flash Player and your Java Runtime Environment. Web technologies have since grown much more robust, and those days are over. For most applications, it's time to stop delivering content that depends on plugins and start taking advantage of Web technologies instead.

- **Broaden your reach to everyone.** Everyone has a browser, but plugins are increasingly rare, especially among mobile users. Since the Web is largely usable without plugins, people would rather just go to your competitors' websites than install a plugin.
- **Give yourself a break from the extra accessibility headaches that come with Flash and other plugins.**
- **Stay clear of additional security hazards.** Adobe Flash is notoriously insecure, even after countless patches. In 2015, Alex Stamos, chief security officer of Facebook, even requested that Adobe discontinue Flash.

So what should you do? If you need interactivity, HTML and JavaScript can readily get the job done for you with no need for Java applets or outdated ActiveX/BHO technology. Instead of relying on Adobe Flash, you can use HTML5 video for your media needs, SVG for vector graphics, and Canvas for complex images and animations. Peter Elst was already writing some years ago that Adobe Flash is rarely the right tool for the job, except for specialized gaming and business applications. As for ActiveX, even Microsoft's Edge browser no longer supports it.

Summary

The topic of embedding other content in web documents can quickly become very complex, so in this article, we've tried to introduce it in a simple, familiar way that will immediately seem relevant, while still hinting at some of the more advanced features of the involved technologies. To start with, you are unlikely to use embedding for much beyond including third-party content like maps and videos on your pages. As you become more experienced, however, you are likely to start finding more uses for them.

There are many other technologies that involve embedding external content besides the ones we discussed here. We saw some in earlier articles, such as `<video>`, `<audio>`, and `<img>`, but there are others to discover, such as `<canvas>` for JavaScript-generated 2D and 3D graphics, and `<svg>` for embedding vector graphics. We'll look at SVG in the next article of the module.

← Previous ↑ Overview: Multimedia and embedding Next →

In this module