Multimedia and Embedding

Images and the <img> element

Let's start with the humble `<img>` element, used to embed a simple image in a webpage. In order to put a simple image on a webpage, we use the `<img>` element. This is an empty element (meaning that it has no text content or closing tag) that requires a minimum of one attribute to be useful — `src`. The `src` attribute contains a path pointing to the image you want to embed in the page, which can be a relative or absolute URL, in the same way as `href` attribute values in `<a>` elements.

So for example, if your image is called dinosaur.jpg, and it sits in the same directory as your HTML page, you could embed the image like so:

```
<img src="dinosaur.jpg">
```

If the image was in an "images" subdirectory, which was inside the same directory as the HTML page, then you'd embed it like this:

```
<img src="images/dinosaur.jpg">
```

Note: Search engines also read image filenames and count them towards SEO. Therefore, you should give your image a descriptive filename; dinosaur.jpg is better than img835.png.

You could also embed the image using its absolute URL, for example:

```
<img src="https://www.example.com/images/dinosaur.jpg">
```

But this is pointless, as it just makes the browser do more work, looking up the IP address from the DNS server all over again, etc. You'll almost always keep the images for your website on the same server as your HTML.

Note: Elements like `<img>` and `<video>` are sometimes referred to as **replaced elements**. This is because the element's content and size are defined by an external resource (like an image or video file), not by the contents of the element itself.

Alternative text

The next attribute we'll look at is `alt`. Its value is supposed to be a textual description of the image, for use in situations where the image cannot be seen/displayed or takes a long time to render because of a slow internet connection. For example,

```
<img src="images/dinosaur.jpg"alt="The head and torso of a dinosaur skeleton;
          it has a large head with long sharp teeth">
```

The easiest way to test your alt text is to purposely misspell your filename.

The `alt` can come in handy for a number of reasons:

- The user is visually impaired, and is using a screen reader to read the web out to them. In fact, having alt text available to describe images is useful to most users.

- As described above, the spelling of the file or path name might be wrong.
- The browser doesn't support the image type. Some people still use text-only browsers, such as Lynx, which displays the alt text of images.
- You may want to provide text for search engines to utilize; for example, search engines can match alt text with search queries.
- Users have turned off images to reduce data transfer volume and distractions. This is especially common on mobile phones, and in countries where bandwidth is limited or expensive.

Width and height

You can use the `width` and `height` attributes to specify the width and height of your image.

```
<img src="images/dinosaur.jpg"
     alt="The head and torso of a dinosaur skeleton;
          it has a large head with long sharp teeth"
      width="400"
      height="341">
```

This doesn't result in much difference to the display, under normal circumstances. But if the user has just navigated to the page, and the image hasn't yet loaded, you'll notice the browser is leaving a space for the image to appear in.

However, you shouldn't alter the size of your images using HTML attributes. If you set the image size too big, you'll end up with images that look grainy, fuzzy, or too small, and wasting bandwidth downloading an image that is not fitting the user's needs. The image may also end up looking distorted, if you don't maintain the correct aspect ratio. You should use an image editor to put your image at the correct size before putting it on your webpage. Or else you should use CSS to alter an image's size.

Image titles

As with links, you can also add title attributes to images, to provide further supporting information if needed.

```
<img src="images/dinosaur.jpg"
     alt="The head and torso of a dinosaur skeleton;
          it has a large head with long sharp teeth"
     width="400"
     height="341"
     title="A T-Rex on display in the Manchester University Museum">
```

This gives us a tooltip on mouse hover, just like link titles:

Annotating images with figures and figure captions

The HTML5 `<figure>` and `<figcaption>` elements can be used to semantically link the image to its caption, which will not cause problems to screen readers.

For example, when you have 50 images and captions, the question of which caption goes with which image can be addressed by using `<figure>` and `<figcaption>`. The `<figcaption>` element tells browsers, and assistive technology that the caption describes the other content of the `<figure>` element.

```
<figure>
  <img src="images/dinosaur.jpg"
       alt="The head and torso of a dinosaur skeleton;
```

```
            it has a large head with long sharp teeth"
       width="400"
       height="341">

  <figcaption>A T-Rex on display in the Manchester University Museum.</figcaption>
</figure>
```

Note: From an accessibility viewpoint, captions and alt text have distinct roles. Captions benefit even people who can see the image, whereas alt text provides the same functionality as an absent image. Therefore, captions and alt text shouldn't just say the same thing, because they both appear when the image is gone. Try turning images off in your browser and see how it looks.

A figure doesn't have to be an image. It is an independent unit of content that:

- Expresses your meaning in a compact, easy-to-grasp way.
- Could go in several places in the page's linear flow.
- Provides essential information supporting the main text.

A figure could be several images, a code snippet, audio, video, equations, a table, or something else.

CSS background images

You can also use CSS to embed images into webpages (and JavaScript, but that's another story entirely). The CSS `background-image` property, and the other `background-*` properties, are used to control background image placement. For example, to place a background image on every paragraph on a page, you could do this:

```
p {
  background-image: url("images/dinosaur.jpg");
}
```

The resulting embedded image is arguably easier to position and control than HTML images. So why bother with HTML images? As hinted to above, CSS background images are for decoration only. If you just want to add something pretty to your page to enhance the visuals, this is fine. Though, such images have no semantic meaning at all. They can't have any text equivalents, are invisible to screen readers, and so on. This is where HTML images shine!

If an image has meaning, in terms of your content, you should use an HTML image. If an image is purely decoration, you should use CSS background images.

Video and Audio on the web

In the early days, native web technologies such as HTML didn't have the ability to embed video and audio on the Web, so proprietary (or plugin-based) technologies like Flash — and later, Silverlight (both of which are now obsolete) — became popular for handling such content. This kind of technology worked ok, but it had a number of problems, including not working well with HTML/CSS features, security issues, and accessibility issues.

The HTML5 specification provided native solution and added the `<video>` and `<audio>` elements would solve much of above problems if implemented correctly.

Note: There are quite a few OVPs (online video providers) like YouTube, Dailymotion, and Vimeo, and online audio providers like Soundcloud. Such companies offer a convenient, easy way to host and consume videos, so you don't have to worry about the enormous bandwidth consumption. OVPs even usually offer ready-made code for embedding video/audio in your webpages; if you use that route, you can avoid some of the difficulties we discuss further.

The <video> element

The `<video>` element allows you to embed a video very easily. A really simple example looks like this:

```
<video src="rabbit320.webm" controls>
  <p>Your browser doesn't support HTML5 video. Here is a <a href="rabbit320.webm">link  to the video</a> instea
d.</p>
</video>
```

The features of note are:

- `src`

  In the same way as for the `<img>` element, the `src` (source) attribute contains a path to the video you want to embed. It works in exactly the same way.

- `controls`

  For the users to be able to control video and audio playback, you must either use the controls attribute to include the browser's own control interface, or build your interface using the appropriate JavaScript API. At a minimum, the interface must include a way to start and stop the media, and to adjust the volume.

- **The paragraph inside the <video> tags**

  This is called **fallback content** — this will be displayed if the browser accessing the page doesn't support the `<video>` element, allowing us to provide a fallback for older browsers. This can be anything you like; in this case, we've provided a direct link to the video file, so the user can at least access it some way regardless of what browser they are using.

Using multiple source formats to improve compatibility

Different browsers support different video (and audio) formats. Things become slightly more complicated because not only does each browser support a different set of container file formats, they also each support a different selection of codecs. Fortunately, there are things you can do to help prevent this from being a problem. In order to maximize the likelihood that your web site or app will work on a user's browser, you may need to provide each media file you use in multiple formats. If your site and the user's browser don't share a media format in common, your media won't play.

Also, mobile browsers may support additional formats not supported by their desktop equivalents, just like they may not support all the same formats the desktop version does. On top of that, both desktop and mobile browsers may be designed to offload handling of media playback (either for all media or only for specific types it can't handle internally). This means media support is partly dependent on what software the user has installed.

So how do we do this?

```
<video controls>
  <source src="rabbit320.mp4" type="video/mp4">
  <source src="rabbit320.webm" type="video/webm">
```

```
  <p>Your browser doesn't support HTML5 video. Here is a <a href="rabbit320.mp4">link to the video</a> instead.</p>
</video>
```

Here we've taken the `src` attribute out of the actual `<video>` tag, and instead included separate `<source>` elements that point to their own sources. In this case the browser will go through the `<source>` elements and play the first one that it has the codec to support. Including WebM and MP4 sources should be enough to play your video on most platforms and browsers these days.

Each `<source>` element also has a `type` attribute. This is optional, but it is advised that you include it. The `type` attribute contains the media type of the file specified by the `<source>`, and browsers can use the type to immediately skip videos they don't understand. If type isn't included, browsers will load and try to play each file until they find one that works, which obviously takes time and is an unnecessary use of resources.

Other <video> features

There are a number of other features you can include when displaying an HTML video as shown:

```
<video controls width="400" height="400"
       autoplay loop muted preload="auto"
       poster="poster.png">
  <source src="rabbit320.mp4" type="video/mp4">
  <source src="rabbit320.webm" type="video/webm">
  <p>Your browser doesn't support HTML video. Here is a <a href="rabbit320.mp4">link to the video</a> instead.</p>
</video>
```

`width` and `height`

You can control the video size either with these attributes or with CSS. In both cases, videos maintain their native width-height ratio — known as the **aspect ratio**. If the aspect ratio is not maintained by the sizes you set, the video will grow to fill the space horizontally, and the unfilled space will just be given a solid background color by default.

`autoplay`

Makes the audio or video start playing right away, while the rest of the page is loading. You are advised not to use autoplaying video (or audio) on your sites, because users can find it really annoying.

`loop`

Makes the video (or audio) start playing again whenever it finishes. This can also be annoying, so only use if really necessary.

`muted`

Causes the media to play with the sound turned off by default.

`poster`

The URL of an image which will be displayed before the video is played. It is intended to be used for a splash screen or advertising screen.

`preload`

Used for buffering large files; it can take one of three values:

- `"none"` does not buffer the file
- `"auto"` buffers the media file
- `"metadata"` buffers only the metadata for the file

The <audio> element

he `<audio>` element works just like the `<video>` element, with a few small differences as outlined below. A typical example might look like so:

```
<audio controls>
  <source src="viper.mp3" type="audio/mp3">
  <source src="viper.ogg" type="audio/ogg">
  <p>Your browser doesn't support HTML5 audio. Here is a <a href="viper.mp3">link to the audio</a> instead.</p>
</audio>
```

Other differences from HTML video are as follows:

- The `<audio>` element doesn't support the `width`/`height` attributes — again, there is no visual component, so there is nothing to assign a width or height to.
- It also doesn't support the `poster` attribute — again, no visual component.

Other than this, <audio> supports all the same features as <video> — review the above sections for more information about them.

Displaying video text tracks

HTML can provide a transcript of the words being spoken in the audio/video.

To do so we use the WebVTT file format and the `<track>` element.

WebVTT is a format for writing text files containing multiple strings of text along with metadata such as the time in the video at which each text string should be displayed, and even limited styling/positioning information. These text strings are called **cues**, and there are several kinds of cues which are used for different purposes. The most common cues are:

- **subtitles:** Translations of foreign material, for people who don't understand the words spoken in the audio.
- **captions:** Synchronized transcriptions of dialog or descriptions of significant sounds, to let people who can't hear the audio understand what is going on.
- **timed descriptions:** Text which should be spoken by the media player in order to describe important visuals to blind or otherwise visually impaired users.

A typical WebVTT file will look something like this:

```
WEBVTT

1
00:00:22.230 --> 00:00:24.606
This is the first subtitle.

2
00:00:30.739 --> 00:00:34.074
This is the second.

  ...
```

To get this displayed along with the HTML media playback, you need to:

1. Save it as a `.vtt` file in a sensible place.

2. Link to the `.vtt` file with the `<track>` element. `<track>` should be placed within `<audio>` or `<video>`, but after all `<source>` elements. Use the `kind` attribute to specify whether the cues are `subtitles`, `captions`, or `descriptions`. Further, use `srclang` to tell the browser what language you have written the subtitles in. Finally, add `label` to help readers identify the language they are searching for.

Here's an example:

```
<video controls>
    <source src="example.mp4" type="video/mp4">
    <source src="example.webm" type="video/webm">
    <track kind="subtitles" src="subtitles_es.vtt" srclang="es" label="Spanish">
</video>
```

From object to iframe — other embedding technologies

Some HTML elements 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. `<embed>` and `<object>` are on their way to obsolescence.

Iframes

Say you wanted to include a PDF in your webpage.

```
<iframe
      src="media/mypdf.pdf"
      frameborder="0"
      width-="800"
      height="500"
      allowfullscreen
      frameborder="0"
      ><p>
        Here is a <a href="media/mypdf.pdf">link to the document.</a>
      </p>
</iframe>
```

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

- `frameborder` : The frameborder attribute allows set the border for the iframe. This can also be achieved by using CSS `border: none` attribute
- `allowfullscreen` : If set, the `<iframe>` is able to be placed in fullscreen mode
- `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.

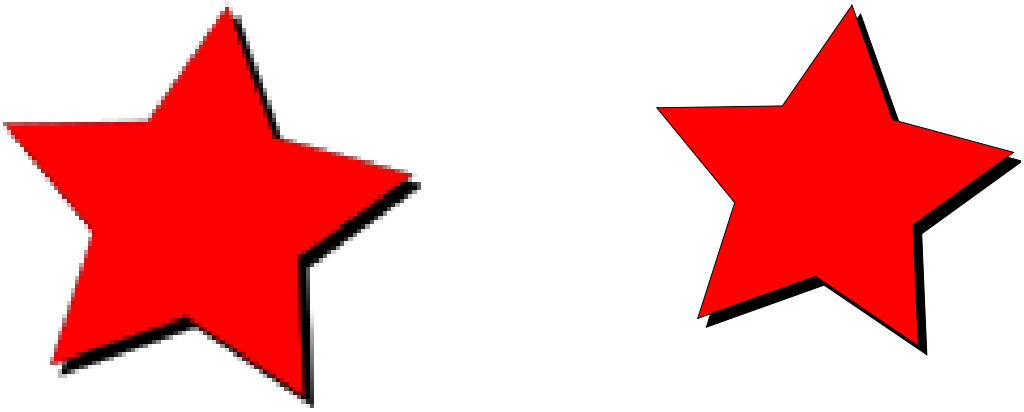
Adding vector graphics to the Web

Vector graphics are very useful in many circumstances — they have small file sizes and are highly scalable, so they don't pixelate when zoomed in or blown up to a large size.

Vector Graphics

- **Raster images** are defined using a grid of pixels — a raster image file contains information showing exactly where each pixel is to be placed, and exactly what color it should be. Popular web raster formats include Bitmap (`.bmp`), PNG (`.png`), JPEG (`.jpg`), and GIF (`.gif`.)
- **Vector images** are defined using algorithms — a vector image file contains shape and path definitions that the computer can use to work out what the image should look like when rendered on the screen. The SVG format allows us to create powerful vector graphics for use on the Web.

Let's look at an example



Moreover, vector image files are much lighter than their raster equivalents, because they only need to hold a handful of algorithms, rather than information on every pixel in the image individually.

What is SVG?

SVG is an XML-based language for describing vector images. It's basically markup, like HTML, except that you've got many different elements for defining the shapes you want to appear in your image, and the effects you want to apply to those shapes. SVG is for marking up graphics, not content.

As a simple example, the following code creates a circle centered inside a rectangle:

```
<svg version="1.1"
     baseProfile="full"
     width="300" height="200"
     xmlns="http://www.w3.org/2000/svg">
  <rect width="100%" height="100%" fill="black" />
  <circle cx="150" cy="100" r="90" fill="blue" />
</svg>
```

From the example above, you may get the impression that SVG is easy to handcode. Yes, you can handcode simple SVG in a text editor, but for a complex image this quickly starts to get very difficult. For creating SVG images, most people use a vector graphics editor like Inkscape or Adobe Illustrator. These packages allow you to create a variety of illustrations using various graphics tools, and create approximations of photos (for example Inkscape's Trace Bitmap feature.)

SVG has some additional advantages besides those described so far:

- Text in vector images remains accessible (which also benefits your SEO).
- SVGs lend themselves well to styling/scripting, because each component of the image is an element that can be styled via CSS or scripted via JavaScript.

SVG does have some disadvantages too:

- SVG can get complicated very quickly, meaning that file sizes can grow; complex SVGs can also take significant processing time in the browser.
- SVG can be harder to create than raster images, depending on what kind of image you are trying to create.
- SVG is not supported in older browsers, so may not be suitable if you need to support older versions of Internet Explorer with your web site (SVG started being supported as of IE9.)

Adding SVG to your pages

The quick way: img element

To embed an SVG via an <img> element, you just need to reference it in the src attribute as you'd expect. You will need a height or a width attribute (or both if your SVG has no inherent aspect ratio).

```
<img
    src="equilateral.svg"
    alt="triangle with all three sides equal"
    height="87"
    width="100" />
```

Pros

- Quick, familiar image syntax with built-in text equivalent available in the `alt` attribute.
- You can make the image into a hyperlink easily by nesting the `<img>` inside an `<a>` element.
- The SVG file can be cached by the browser, resulting in faster loading times for any page that uses the image loaded in the future.

Cons

- You cannot manipulate the image with JavaScript.
- If you want to control the SVG content with CSS, you must include inline CSS styles in your SVG code. (External stylesheets invoked from the SVG file take no effect.)
- You cannot restyle the image with CSS pseudo-classes (like `:focus`).

How to include SVG code inside your HTML

You can also open up the SVG file in a text editor, copy the SVG code, and paste it into your HTML document — this is sometimes called putting your **SVG inline**, or **inlining SVG**. Make sure your SVG code snippet begins with an `<svg>` start tag and ends with an `</svg>` end tag. Here's a very simple example of what you might paste into your document:

```
<svg width="300" height="200"><rect width="100%" height="100%" fill="green" /></svg>
```

Pros

- Putting your SVG inline saves an HTTP request, and therefore can reduce a bit your loading time.
- You can assign `class`es and `id`s to SVG elements and style them with CSS, either within the SVG or wherever you put the CSS style rules for your HTML document. In fact, you can use any SVG

presentation attribute as a CSS property.

- Inlining SVG is the only approach that lets you use CSS interactions (like `:focus`) and CSS animations on your SVG image (even in your regular stylesheet.)
- You can make SVG markup into a hyperlink by wrapping it in an `<a>` element.

Cons

- This method is only suitable if you're using the SVG in only one place. Duplication makes for resource-intensive maintenance.
- Extra SVG code increases the size of your HTML file.
- The browser cannot cache inline SVG as it would cache regular image assets, so pages that include the image will not load faster after the first page containing the image is loaded.

How to embed an SVG with an iframe

You can open SVG images in your browser just like webpages. So embedding an SVG document with an <iframe> can be done.

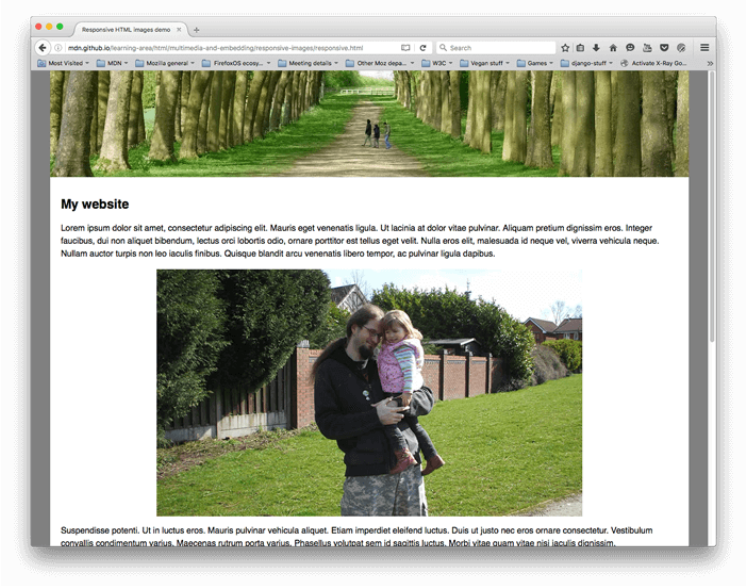
```
<iframe src="triangle.svg" width="500" height="500" sandbox>
    <img src="triangle.png" alt="Triangle with three unequal sides" />
</iframe>
```

Responsive images

Responsive images are images that work well on devices with widely differing screen sizes, resolutions, and other such features. This helps to improve performance across different devices. Responsive images are just one part of responsive design, a future CSS topic.

Why responsive images?

Let's examine a typical scenario. A typical website may contain a header image and some content images below the header. The header image will likely span the whole of the width of the header, and the content image will fit somewhere inside the content column. Here's a simple example(you may also visit the source code for below webpage):



This works well on a wide screen device, such as a laptop or desktop. We won't discuss the CSS much, except to say that:

- The body content has been set to a maximum width of 1200 pixels — in viewports above that width, the body remains at 1200px and centers itself in the available space. In viewports below that width, the body will stay at 100% of the width of the viewport.

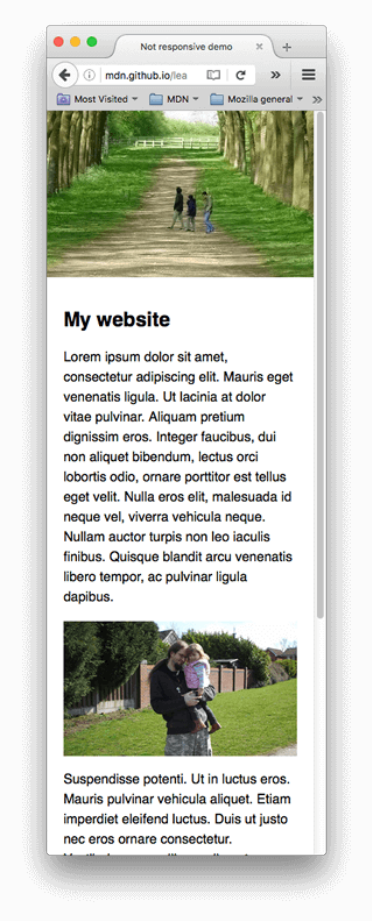
```
body {
        width: 100%;
        max-width: 1200px;
        margin: 0 auto;
        background-color: white;
      }
```

- The header image has been set so that its center always stays in the center of the header, no matter what width the heading is set at. If the site is being viewed on a narrower screen, the important detail in the center of the image (the people) can still be seen, and the excess is lost off either side. It is 200px high.

```
header {
        background: url(header.jpg) no-repeat center;
        height: 200px;
      }
```

- The content images have been set so that if the body element becomes smaller than the image, the images start to shrink so that they always stay inside the body, rather than overflowing it.

However, issues arise when you start to view the site on a narrow screen device. The header below looks ok, but it's starting to take up a lot of the screen height for a mobile device. And at this size, it is difficult to see the people within the first content image.



An improvement would be to display a cropped version of the image which displays the important details of the image when the site is viewed on a narrow screen. A second cropped image could be displayed for a medium-width screen device, like a tablet. The general problem whereby you want to serve different cropped images in that way, for various layouts, is commonly known as the **art direction problem**.

In addition, there is no need to embed such large images on the page if it is being viewed on a mobile screen. And conversely, a small raster image starts to look grainy when displayed larger than its original size. This is called the **resolution switching problem**.

Conversely, it is unnecessary to display a large image on a screen significantly smaller than the size it was meant for. Doing so can waste bandwidth; in particular, mobile users don't want to waste bandwidth by downloading a large image intended for desktop users, when a small image would do for their device. Ideally, you would have multiple resolutions available and serve the appropriate size depending upon the device accessing the data on the website.

To make things more complicated, some devices have high resolution screens that need larger images than you might expect to display nicely. This is essentially the same problem, but in a slightly different context.

This kind of problem didn't exist when the web first existed, in the early to mid 90s — back then the only devices in existence to browse the Web were desktops and laptops, so browser engineers and spec writers didn't even think to implement solutions. Responsive image technologies were implemented recently to solve the problems indicated above by letting you offer the browser several image files, either all showing the same thing but containing different numbers of pixels (resolution switching), or different images suitable for different space allocations (art direction).

How do you create responsive images?

Let's look at the two problems illustrated above and show how to solve them using HTML's responsive image features. You should note that we will be focusing on `<img>` elements for this section, as seen in the content area of the example above — the image in the site header is only for decoration, and therefore implemented using CSS background images.

Resolution switching: Different sizes

So, what is the problem that we want to solve with resolution switching? We want to display identical image content, just larger or smaller depending on the device — this is the situation we have with the second content image in our example. The standard `<img>` element traditionally only lets you point the browser to a single source file:

```
<img src="elva-fairy-800w.jpg" alt="Elva dressed as a fairy">
```

We can however use two new attributes — `srcset` and `sizes` — to provide several additional source images along with hints to help the browser pick the right one.

```
<img srcset="elva-fairy-480w.jpg 480w,
             elva-fairy-800w.jpg 800w"
     sizes="(max-width: 600px) 480px,
            800px"
     src="elva-fairy-800w.jpg"
     alt="Elva dressed as a fairy">
```

For the `srcset` and `sizes` attributes each value contains a comma-separated list, and each part of those lists is made up of three sub-parts. Let's run through the contents of each now:

`srcset` defines the set of images we will allow the browser to choose between, and what size each image is. Each set of image information is separated from the previous one by a comma. For each one, we write:

1. An **image filename** (`elva-fairy-480w.jpg`)
2. A space
3. The image's **intrinsic width in pixels** (`480w`) — note that this uses the `w` unit, not `px` as you might expect. This is the image's real size, which can be found by inspecting the image file on your computer

`sizes` defines a set of media conditions (e.g. screen widths) and indicates what image size would be best to choose, when certain media conditions are true — these are the hints we talked about earlier. In this case, before each comma we write:

1. A **media condition** (`(max-width:600px)`) — a media condition describes a possible state that the screen can be in. In this case, we are saying "when the viewport width is 600 pixels or less.
2. A space
3. The **width of the slot** the image will fill when the media condition is true (`480px`)

Note: For the slot width, you may provide an absolute length (px, em) or a length relative to the viewport (vw), but not percentages. You may have noticed that the last slot width has no media condition (this is the default that is chosen when none of the media conditions are true). The browser ignores everything after the first matching condition, so be careful how you order the media conditions.

So, with these attributes in place, the browser will:

1. Look at its device width.
2. Work out which media condition in the `sizes` list is the first one to be true.
3. Look at the slot size given to that media query.
4. Load the image referenced in the `srcset` list that has the same size as the slot or, if there isn't one, the first image that is bigger than the chosen slot size.

At this point, if a supporting browser with a viewport width of 480px loads the page, the (max-width: 600px) media condition will be true, and so the browser chooses the 480px slot. The elva-fairy-480w.jpg will be loaded, as its inherent width (480w) is closest to the slot size. The 800px picture is 128KB on disk, whereas the 480px version is only 63KB — a saving of 65KB. Now, imagine if this was a page that had many pictures on it. Using this technique could save mobile users a lot of bandwidth.

Note: In the <head> of the example linked above, you'll find the line `<meta name="viewport" content="width=device-width">`: this forces mobile browsers to adopt their real viewport width for loading web pages (some mobile browsers lie about their viewport width, and instead load pages at a larger viewport width then shrink the loaded page down, which is not very helpful for responsive images or design).

Resolution switching: Same size, different resolutions

If you're supporting multiple display resolutions, but everyone sees your image at the same real-world size on the screen, you can allow the browser to choose an appropriate resolution image by using `srcset` with `x`-descriptors and without `sizes`

```
<img srcset="elva-fairy-320w.jpg,
             elva-fairy-480w.jpg 1.5x,
             elva-fairy-640w.jpg 2x"
     src="elva-fairy-640w.jpg"
     alt="Elva dressed as a fairy">
```

In this example, the following CSS is applied to the image so that it will have a width of 320 pixels on the screen (also called CSS pixels):

```
img {
  width: 320px;
}
```

In this case, `sizes` is not needed — the browser works out what resolution the display is that it is being shown on, and serves the most appropriate image referenced in the `srcset`. So if the device accessing the page has a standard/low resolution display, with one device pixel representing each CSS pixel, the elva-fairy-320w.jpg image will be loaded (the 1x is implied, so you don't need to include it.) If the device has a high resolution of two device pixels per CSS pixel or more, the elva-fairy-640w.jpg image will be loaded. The 640px image is 93KB, whereas the 320px image is only 39KB.

Art direction

The art direction problem involves wanting to change the image displayed to suit different image display sizes. For example, a web page includes a large landscape shot with a person in the middle when viewed on a desktop browser. When viewed on a mobile browser, that same image is shrunk down, making the person in the image very small and hard to see. It would probably be better to show

a smaller, portrait image on mobile, which zooms in on the person. The `<picture>` element allows us to implement just this kind of solution.

Like <video> and <audio>, the `<picture>` element is a wrapper containing several `<source>` elements that provide different sources for the browser to choose from, followed by the all-important `<img>` element.

```
<picture>
  <source media="(max-width: 799px)" srcset="elva-480w-close-portrait.jpg">
  <source media="(min-width: 800px)" srcset="elva-800w.jpg">
  <img src="elva-800w.jpg" alt="Chris standing up holding his daughter Elva">
</picture>
```

- The `<source>` elements include a `media` attribute that contains a media condition — as with the first `srcset` example, these conditions are tests that decide which image is shown — the first one that returns true will be displayed. In this case, if the viewport width is 799px wide or less, the first `<source>` element's image will be displayed. If the viewport width is 800px or more, it'll be the second one.
- The `srcset` attributes contain the path to the image to display. Just as we saw with `<img>` above, `<source>` can take a `srcset` attribute with multiple images referenced, as well as a `sizes` attribute. So, you could offer multiple images via a `<picture>` element, but then also offer multiple resolutions of each one. Realistically, you probably won't want to do this kind of thing very often.
- In all cases, you must provide an `<img>` element, with `src` and `alt`, right before `</picture>`, otherwise no images will appear. This provides a default case that will apply when none of the media conditions return true (you could actually remove the second `<source>` element in this example), and a fallback for browsers that don't support the `<picture>` element.