Chapter 15

Mobile Apps

What is in This Chapter ?

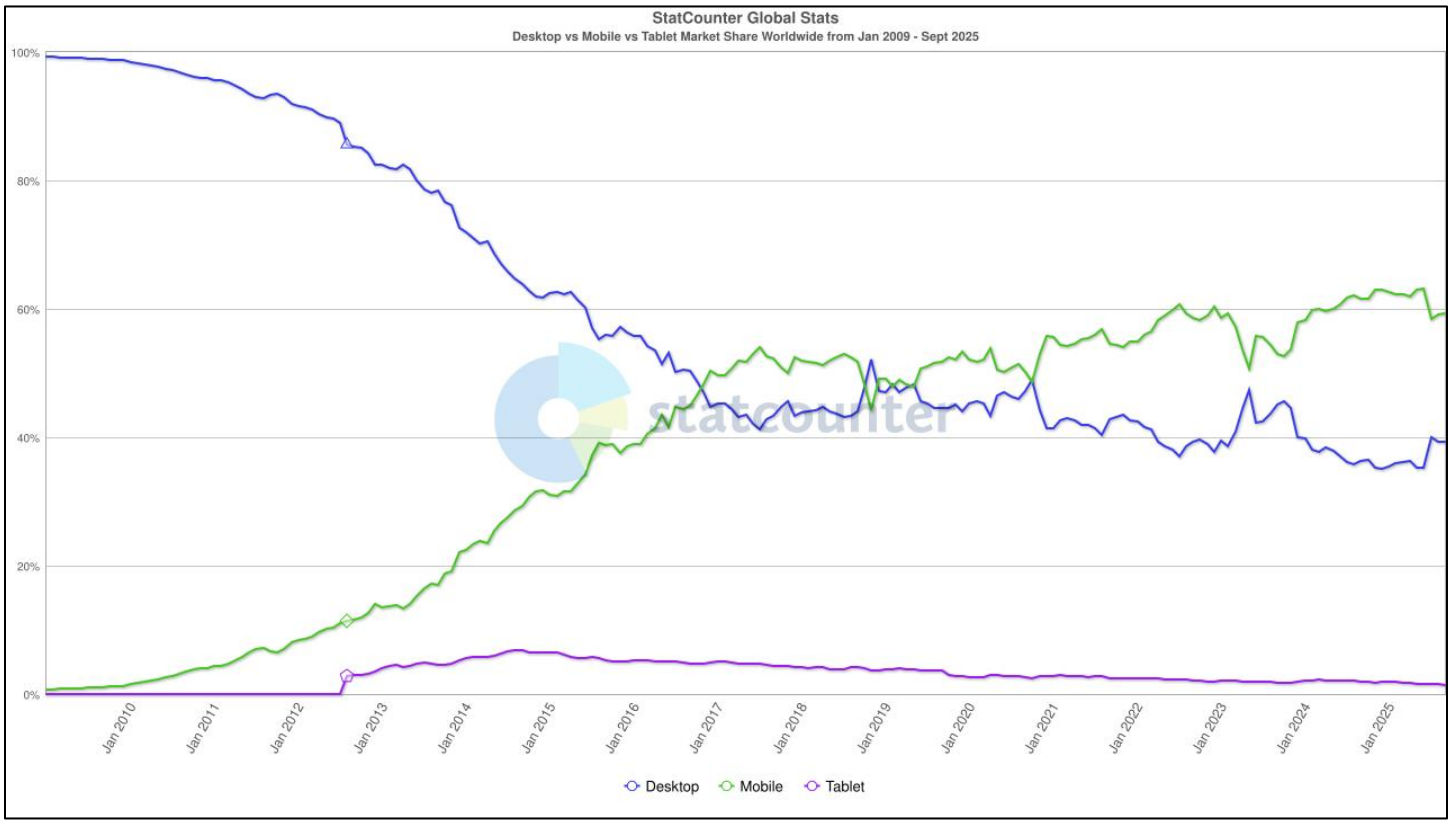
In this chapter, we discus ways to improve our website designs for various devices such as different sized **phones** and **tablets**. We first discus why it is necessary to do this by considering the web design issues that arise with mobile devices. We then get into a discussion of **responsive web design** and show why this is best in practice. Then we look at how **Chrome Dev Tool Device Simulation** can help us with the design of our websites that will look good on all devices. Then we discus how **viewports** play a role in this process and how **media queries** can help us ensure that our pages are accessible and appear nicely. We conclude with a discussion of ensuring **responsive images** on our pages that display nicely on all devices.

15.1 Mobile Vs. Desktop

Over the last 15 years, the way people use software has changed dramatically. Back in 2010, most internet activity happened on desktop/laptop computers, and mobile devices made up only a small share of web traffic. But as smartphones became more powerful and app stores grew, mobile use skyrocketed. By 2016, mobile web traffic had already passed desktop, and today **mobile** devices account for around **60%** of global traffic, leaving **desktop** at under **40%**.

At the same time, the app economy exploded … going from just a few hundred thousand apps in the early 2010s to millions available today, with billions of downloads every year. This shift shows why modern web development must think “mobile-first”.

The fact is that users are far more likely to interact with a service on their phone than on a traditional computer. Here is a chart taken from **https://gs.statcounter.com** that shows this trend:



As this shift toward mobile grew, it wasn’t just apps that changed … web browsing itself had to adapt to the new reality of smaller screens and touch-based interaction.

A ***mobile web browser*** is a browser designed for smartphones and tablets that can display web pages using **HTML**, **CSS**, and **JavaScript**. Modern mobile browsers behave much like their desktop counterparts, though they often lack support for certain desktop-only plugins. Because of their smaller screens and touch-based input, mobile devices encouraged developers to create websites tailored for these environments. These ***mobile-friendly websites*** are optimized for usability on phones through the use of larger buttons, more spacing, and simpler layouts … so that users can comfortably interact with the site using touch instead of a mouse.

Mobile and desktop websites are built using the same core technologies, but the way they are designed and optimized can differ significantly. The table below highlights the key differences between desktop and mobile websites, showing how design and functionality must adapt to the capabilities and limitations of each platform:

Aspect	Desktop Websites	Mobile Websites
Screen size	large, allows wide layouts	small, responsive layouts required
User interaction	mouse and keyboard	touch input (e.g., tap, swipe, pinch)
Connectivity	fast and stable	Slower, interrupted, limited
Data use	unconstrained	caps or metered data plans
Memory	abundant	limited, efficient coding needed
CPU	faster	slower, needs optimization
User intent	longer sessions, detailed tasks, research	quick tasks, on-the-go info, directions, immediate actions
Sensors	camera, mic	camera, mic, GPS, accelerometer,

Here is a quick “true or false” quiz to see if you understand how we need to think differently when developing mobile apps.

1. [T / F] Mobile websites often show less information than their desktop/laptop counterparts.
2. [T / F] Links (navigation) are generally larger on mobile websites.
3. [T / F] Mobile websites should **not** rely on mouse hovering to trigger actions on the website.
4. [T / F] Phone #s and addresses are of equal importance to desktop and mobile website users.
5. [T / F] Navigation links are often just as visible on mobile websites as on desktop websites.
6. [T / F] Video is often played automatically on mobile websites.
7. [T / F] **JavaScript** does not run as quickly on a mobile web browser.

How did you do? Here are the answers:

1. ✓ **True** - Designers must prioritize what information should be displayed because of the smaller screen size.

2. ✓ **True** - Fingers are much fatter than mouse pointers, so link hit targets need to be larger to avoid erroneous clicks.

3. ✓ **True** - Mouse hovering is only useful for desktop websites where the user is using a mouse.

4. × **False** - Mobile website users are more often in search of phone numbers and addresses, so phone numbers and addresses are often more prominent on mobile websites.

5. × **False** - Although some mobile websites prominently display navigation links, many mobile websites hide navigational links behind drop-down menus or hamburger (i.e. ☰) menus to save space. Desktop websites have more room to display navigation links.

6. × **False** - Video can consume a user's data plan quickly, so considerate mobile websites don't play video automatically.

7. ✓ **True** - The limited processor speed and memory of mobile devices cause the **JavaScript** engine to be slower. **JavaScript** engine speed can be a serious issue for websites that rely on intensive **JavaScript** processing like games.

Developers typically implement mobile websites using one of three main approaches:

1. **Separate websites:** Two entirely different sites are created … one optimized for desktop and another for mobile devices.

2. **Dynamic serving:** Both desktop and mobile browsers access the same **URL**, but the server detects the device type and sends either the desktop or mobile version accordingly.

3. **Responsive web design:** The server delivers the same **HTML** to all devices, but the browser adjusts the layout, images, and styling to fit the screen size, ensuring a consistent experience across desktops, tablets, and smartphones.

When doing dynamic serving … how does the server know whether it is a desktop or mobile device? It looks at the **User-Agent** attribute in the request header:

```
GET / HTTP/1.1
Host: code.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:119.0)
            Gecko/20100101 Firefox/119.0

GET / HTTP/1.1
Host: code.org
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 17_0 like Mac OS X)
            AppleWebKit/605.1.15 (KHTML, like Gecko)
            Version/17.0 Mobile/15E148 Safari/605.1.15
```

Of course, as with any choices, there are advantages and disadvantages:

Approach	✓ Advantages	✗ Disadvantages
Separate Websites	✓ full control over design & content ✓ optimized for performance ✓ can target mobile-specific features	✗ requires maintaining two sites ✗ higher development and maintenance cost ✗ must carefully manage Search Engine Optimization for two URLs ✗ may miss features or content on mobile ✗ relies on browser detection, which can be inaccurate
Dynamic Serving	✓ single URL ✓ device-specific optimization ✓ easier cross-device analytics	✗ server-side detection needed ✗ can be complex to implement ✗ risk of serving wrong version ✗ some features/content may be simplified on mobile
Responsive Web Design	✓ one site for all devices ✓ easier maintenance ✓ consistent **URL** structure ✓ adapts to all screen sizes	✗ can be heavier to load if not optimized ✗ less flexibility for device-specific content ✗ may require careful performance tuning

Making one website for all devices is the best approach because it's easier to maintain, keeps content consistent, and avoids problems like missing features or duplicate content.

15.2 Responsive Web Design

Responsive web design is a design approach where a website automatically adjusts its layout, images, and content to fit different screen sizes and devices for an "optimal" user experience. In responsive web design, the goal is to create one website that works well for all users, whether on a desktop, tablet, or smartphone.

To make this work effectively, a good website should be:

- **Minimal** - only include what's necessary
- **Functional** - everything works as expected
- **Intuitive** - easy to navigate and understand
- **Task-focused** - help users achieve their goals quickly

If our "desktop" site doesn't meet these standards, it's time to rethink the design before making it responsive.

Once we create a site like this, we can serve it to all users, regardless of device, and let responsive design ensure that it looks and works great on all devices.

There is a lot we can say about responsive web design. We will only discuss some basic things here but feel free to do research on your own to learn more. Here are some links you can start with:

https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design

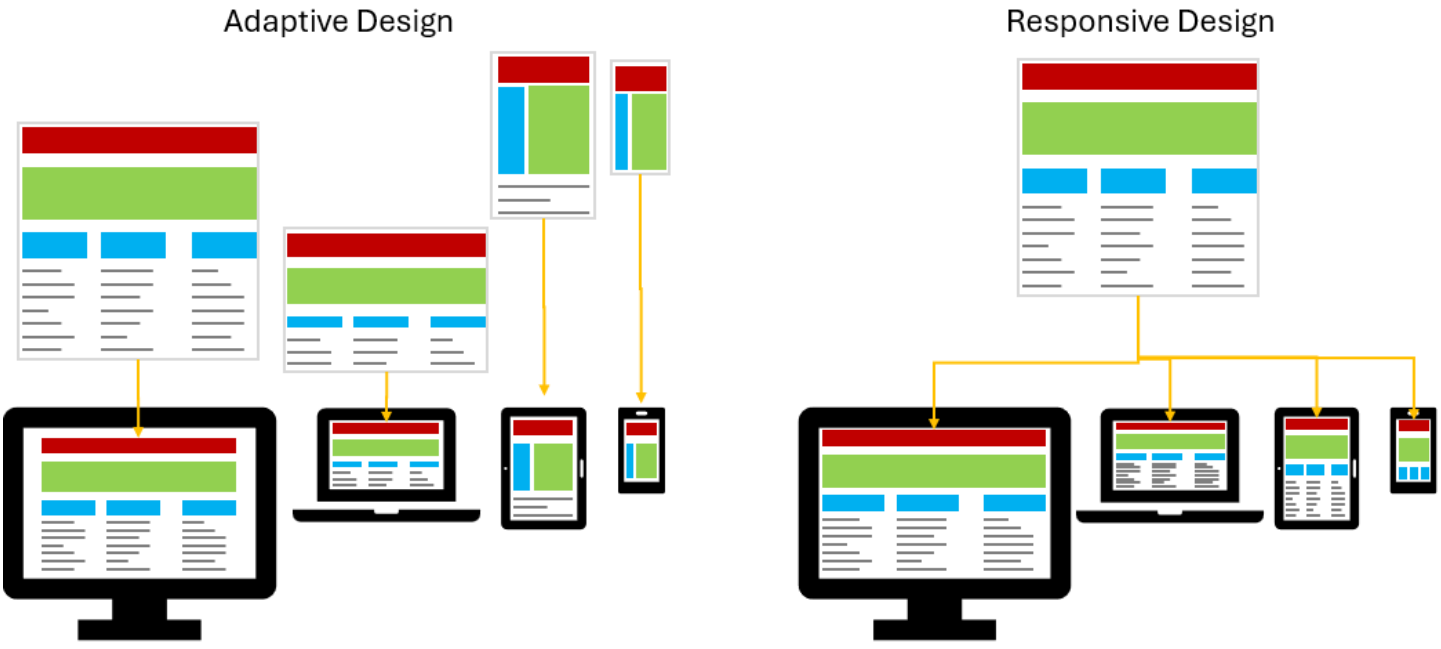
https://www.smashingmagazine.com/2011/01/guidelines-for-responsive-web-design/

Responsive Web Design (RWD) is characterized by three key features:

1. **Fluid grids:** Layouts use relative units (i.e., %, em) instead of fixed pixels, allowing elements to scale smoothly across screen sizes.
2. **Flexible media:** Images and videos scale within their containers, maintaining clarity and preventing overflow.
3. **CSS media queries:** CSS rules adjust styles based on screen width, height, or orientation, optimizing the layout for any device.

Effective responsive web design combines all three of these techniques.

While responsive design adapts fluidly to any screen size, some websites use ***adaptive design***, which relies on predefined layouts for specific device widths. Adaptive layouts "snap" to different designs at key breakpoints … for example, one layout for desktops, another for tablets, and a third for smartphones. This approach can be useful when designers want precise control over how content appears on different devices, ensuring that complex designs or features look exactly as intended at each screen width. It also allows optimization for performance, since only the assets needed for a particular layout are loaded.



Here are two sites that explain the differences between responsive and adaptive design:

https://www.browserstack.com/guide/adaptive-design-vs-responsive-design

https://www.uxpin.com/studio/blog/responsive-vs-adaptive-design-whats-best-choice-designers

While “pure” adaptive design is less common today, responsive sites often behave adaptively at certain breakpoints, which is why it can be hard to spot a true adaptive layout. To get an idea of the differences, it is good to check out a couple of sites and look for when the layout “jumps” or “snaps” to a certain size.

https://www.boston.com/
https://www.adidas.ca/en
https://www.dell.ca/

So, as you can probably guess … it can be a challenge to make our sites look good on all devices. As developers, we need to test our mobile websites on a variety of devices to ensure they work correctly for all users.

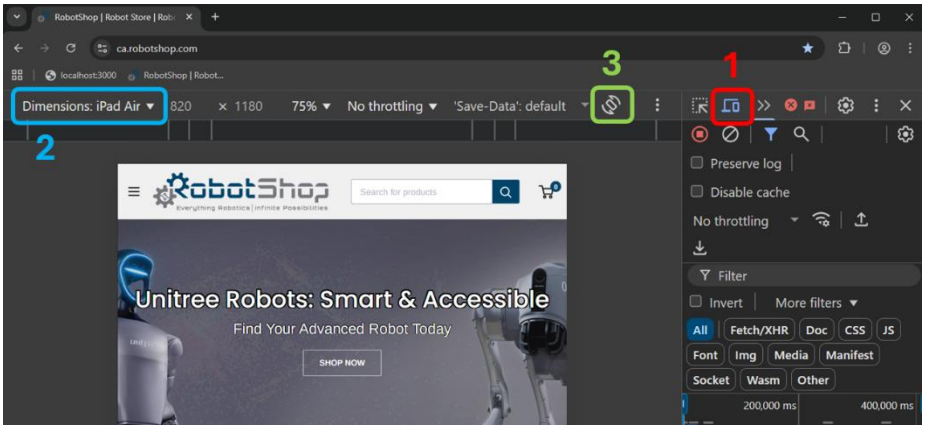
15.3 Chrome Dev Tool Device Simulation & Emulation

While testing on real devices is ideal, many desktop browsers include development tools that make mobile testing easier. For example, **Chrome’s DevTools** offers a screen emulator that can simulate a wide range of smartphones and tablets.

> ***A screen emulator is software that mimics how a mobile device’s screen behaves, including its size, resolution, and touch interactions.***

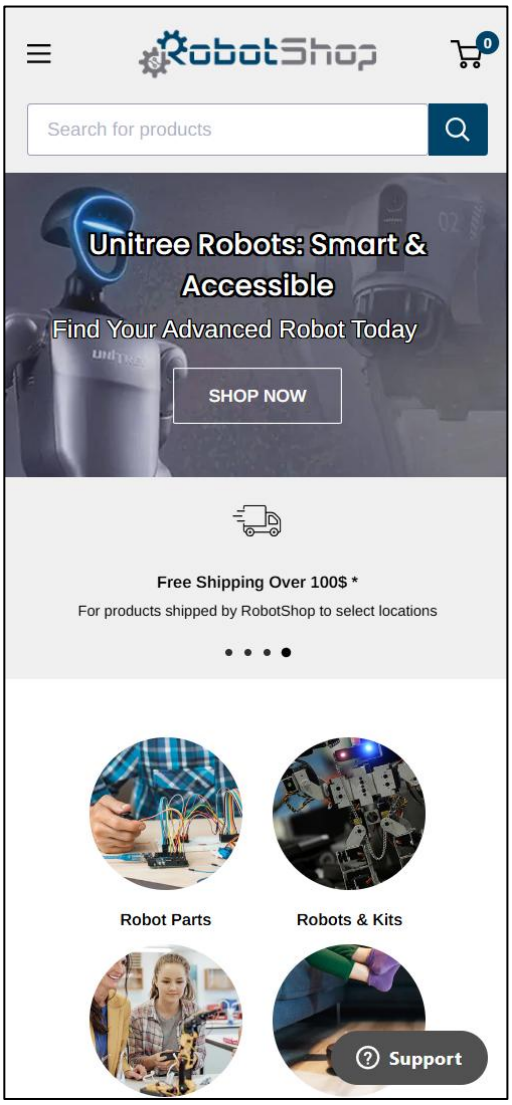
Although screen emulators let us simulate how a website looks and behaves on mobile devices from our desktop, they don’t perfectly replicate a real device. Some things, like CPU performance and certain hardware behaviors, can only be tested on an actual mobile device. When accuracy matters, it’s best to use remote debugging to test and debug directly on a real phone or tablet.

In Chrome … open the **Dev Tools** and then select the mobile button (see 1 →) and select the device from the dropdown list (see 2 →). Select the rotate button to switch from portrait to landscape (see 3 →). This will give us a close idea as to what our site will look like on that device.

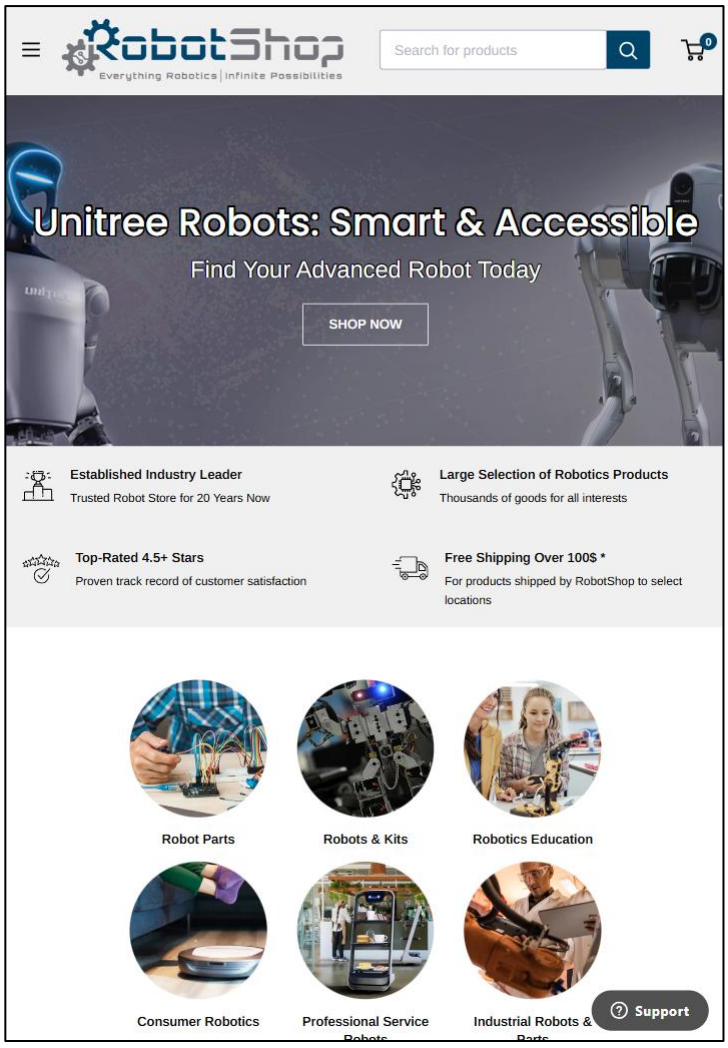


Here are some examples:

iPhone 14 Pro Max (landscape)

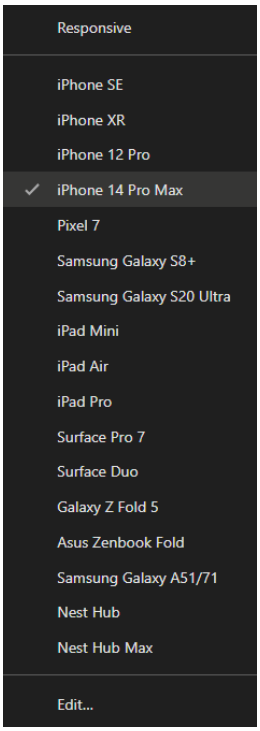


iPhone 14 Pro Max (portrait)



iPad Air (portrait)

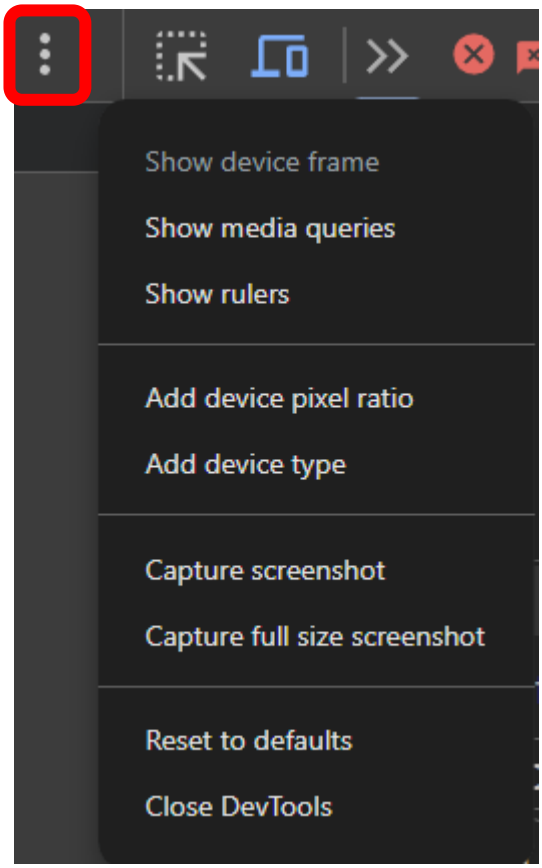
We can interact as we would with a webpage but the cursor turns into a round circle to simulate a finger press.



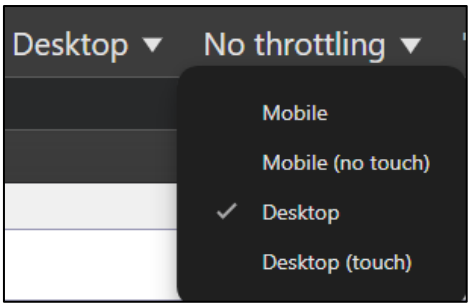
To simulate a pinch gesture, hold the shift key down and drag the mouse.

The dropdown list for the devices give a list as shown here on the right by default … but we can select **Edit…** to add/remove devices from the list →

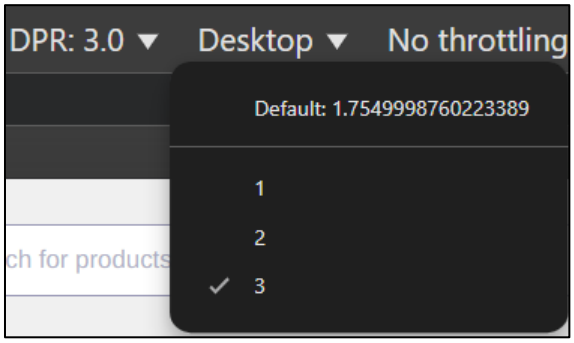
We can also select the **menu** option (i.e., three dots as shown below highlighted in red) to add a couple of more features.



Adding the "**Add device type**" option will add a menu item with a dropdown list that lets us select mobile vs desktop and allow touch or no touch options (see below).

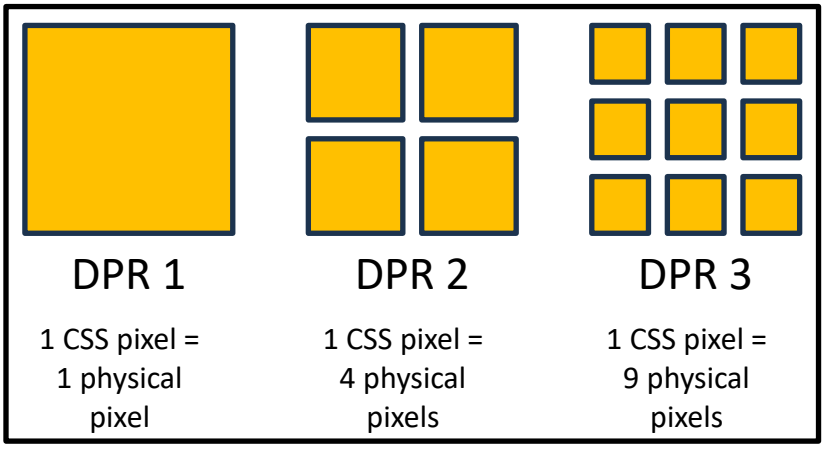


Adding the "**Add device pixel ratio**" option will add a menu item with a dropdown list that lets us select the DPR. See here →



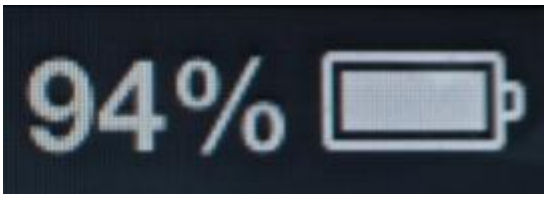
What is the ***Device Pixel Ratio*** (**DPR**)? It is the ratio between a device's **physical pixels** (the tiny dots on the screen) and its **CSS pixels** (the units web developers use in styling).

- A **DPR** of **1** means one **CSS** pixel = one physical pixel, found on older monitors & basic laptops.
- A **DPR** of **2** means one **CSS** pixel = four physical pixels (2×2 grid), found on Retina or high-resolution displays.
- A **DPR** of **3** means one **CSS** pixel = nine physical pixels (3×3 grid), found on some high-end smartphones.

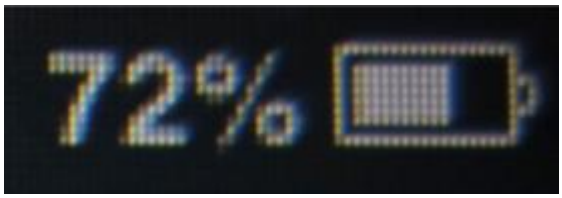


The **DPR** is set by the hardware and display of the device. For example, an iPhone with a Retina display might have a **DPR** of 2 or 3, while a standard desktop monitor is usually 1. However, some devices and browsers let users zoom or change display scaling, which can make the *effective* **DPR** appear different to websites. So, while the hardware sets the base **DPR**, the reported value can shift depending on user settings.

Higher **DPR** values mean sharper visuals, but also require higher-resolution images and icons to avoid looking blurry. Notice the sharpness difference:



2 DPR screen (e.g., iPhone 4)



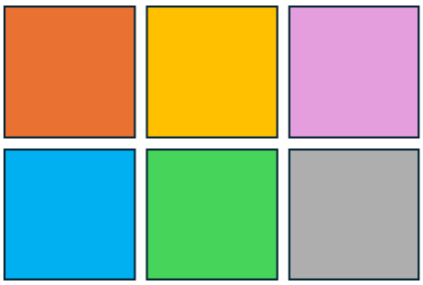
1 DPR screen (e.g., iPhone 3GS)

A ***device-independent pixel*** **(DIP)** is also known as a ***CSS pixel***. It is a virtual unit of measurement used in web and app design that stays the same size regardless of the screen's resolution or pixel density.

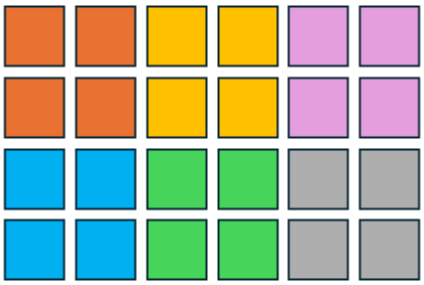
Consider this **CSS** stylesheet code:

```
.box {
  width: 3px;
  height: 2px;
  background-color: red;
}
```

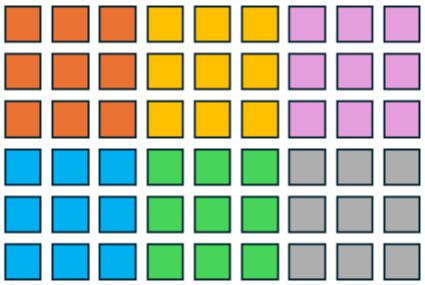
The box will always take up 3x2 logical **CSS** pixels but how many pixels will it take up on the screen to stay the same physical size?



DPR 1 screen
(old monitor)
3×2 physical pixels



DPR 2 screen
(Retina display)
6×4 physical pixels



DPR 3 screen
(high-end phone)
9×6 physical pixels

How will a website image that looks good on a standard 1 **DPR** screen look like on a 3 **DPR** screen?

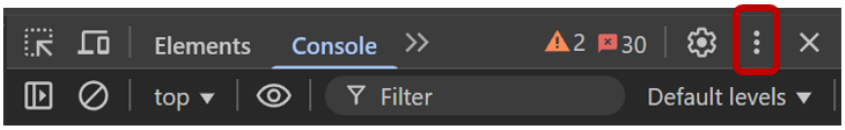
(a) Pixelated or blocky? (b) the same? (c) sharper?

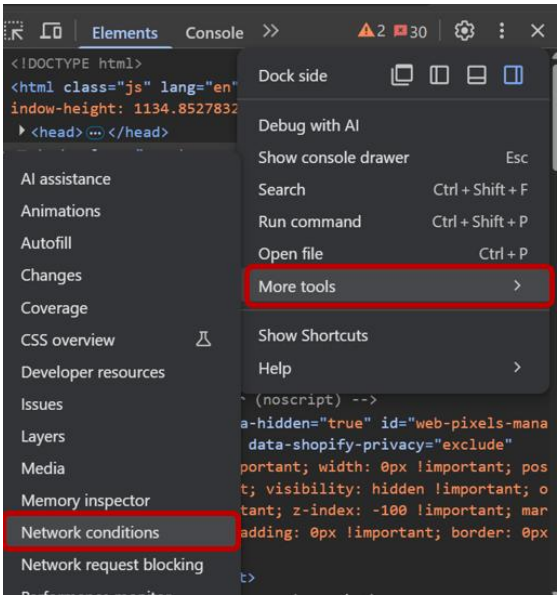
It will look blocky because a 3 **DPR** screen displays multiple pixels for each pixel in the image, giving the image a pixilated look.

For most coding tasks in **HTML/CSS/JavaScript**, we don't need to worry directly about **DPR**. We just work with **CSS** pixels (device-independent pixels), and the browser takes care of scaling them to the actual screen's physical pixels. That's why `width: 100px;` looks about the same size on a laptop and a phone, even though the phone may use many more real pixels to render it.

However, when working with images or graphics, we need to make sure our visuals look sharp on high-**DPR** displays by using higher-resolution images or scalable formats. This is why developers often provide **2** or **3** versions of each raster image for high-**DPR** screens, or alternatively use **Scalable Vector Graphics** (SVGs), which stay sharp at any resolution without needing multiple versions.

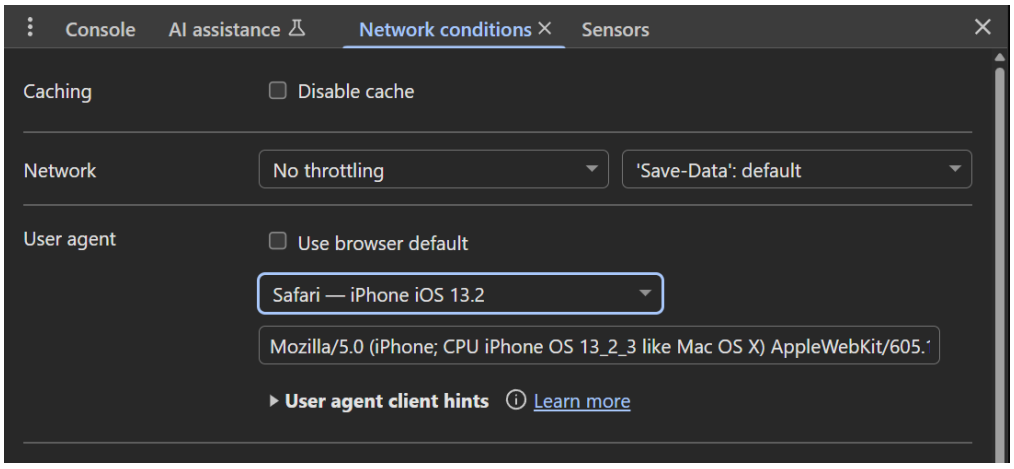
Now, looking at the Chrome **Dev Tools** tab, there is another menu option (with three dots):





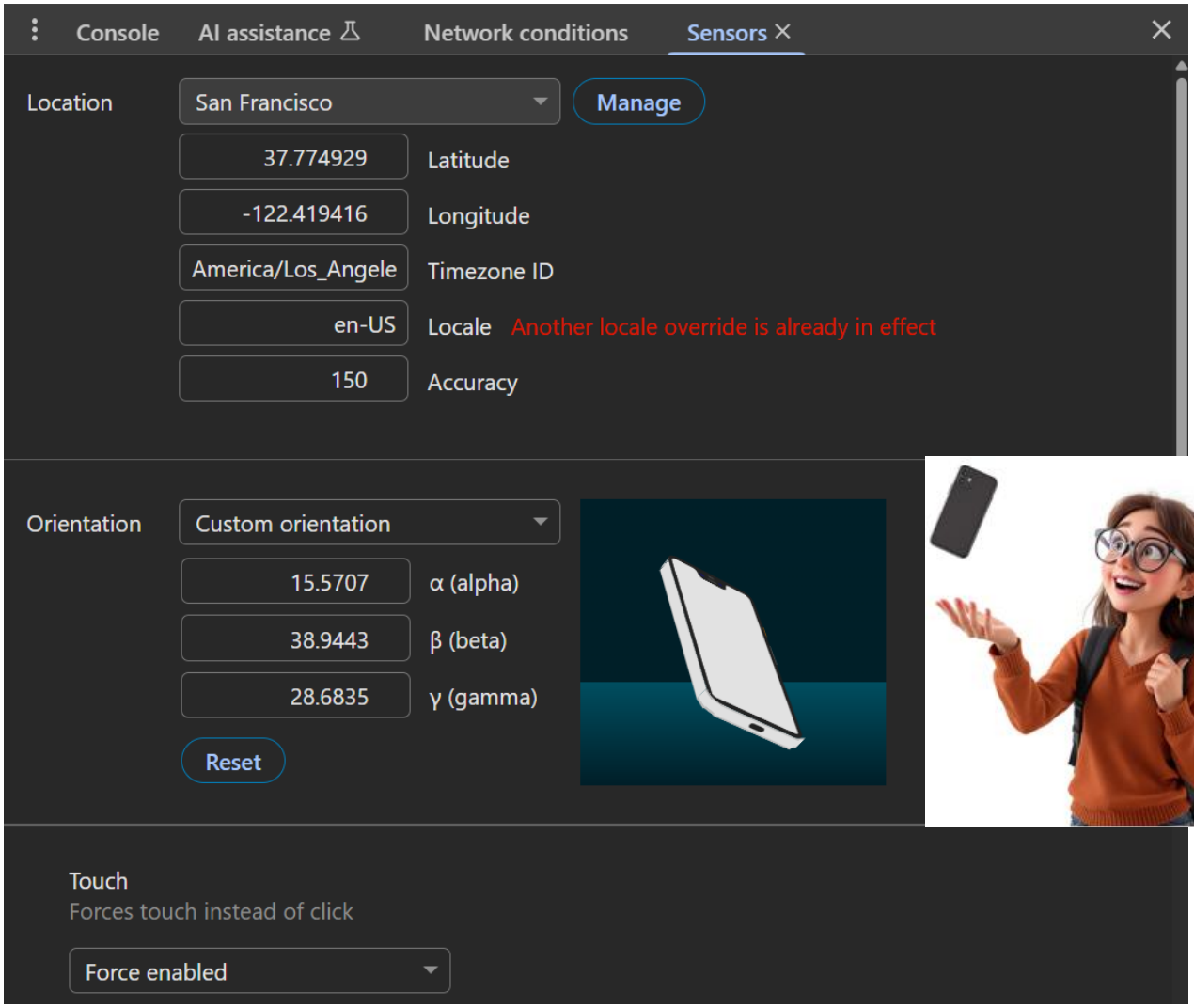
From the menu that appears, we can select **More tools** and then select **Network conditions** or **Sensors** from the menu. Two tabs will appear with those names (see below). For the **Network conditions**, we can …

- disable the cache (forcing the browser to download all web page resources every time),
- simulate slower networks like 3G (or simulate being offline) by using the network settings to see how our site responds, and
- change the user agent string to any number of browser user agents, so see how other pages respond as if we were using a different device.



With the **Sensors** tab, we can …

- emulate a GPS sensor by pretending we are in a certain location or time zone,
- emulate an accelerometer to react to different device orientations (i.e., to pretend that we are moving the device around), and
- simulate a force touch (3D Touch / pressure touch) instead of a simple click. This just simulates a force touch, but it doesn't allow us to indicate a variable pressure (although we can "fake this" with some **JavaScript** code by supplying a pressure value in **PointerEvents**).



1. [T / F] The geolocation (Latitude and Longitude) of the emulated device is modified by moving the desktop computer to a different location.

2. [T / F] A developer can drag the image that looks like a mobile device to modify the accelerometer's **alpha** (rotation around the z-axis), **beta** (left-to-right tilt), and **gamma** (front-to-back tilt) values.

How did you do?

1. × **False** - The developer types **Latitude** and **Longitude** values to change the emulated location of the device..

2. ✓ **True** - A developer can also enter the values directly into the textboxes.

As we can see, by using the Chrome Dev Tools mobile device simulation (which replicates screen sizes, resolutions, and touch interactions) we have a nice way to test and optimize our web apps without needing physical devices. Also, its network simulation feature allows us to test how our site will behave under various connection speeds and latencies. This will help us identify performance bottlenecks and optimize load times. Finally, its sensor emulation lets us simulate device features such as geolocation (gps), device orientation and movement (accelerometer), and touch pressure. That will help us with our site's interactive and context-aware functionality. Ultimately, these tools will help us to catch usability issues early in our website development process.

15.4 Viewports

When we open a web page on a phone, tablet, or desktop, the browser needs to decide how much of the page to show and how to scale it.

The viewport is the visible area of a web page inside a browser window.

Imagine holding up an **empty picture frame** in front of a large poster. The poster represents the **entire web page**, but we can only see the part that fits inside the frame at any given time. As we move the frame or zoom in and out, we reveal different parts of the poster. Similarly, the **viewport** is like that frame: it defines the portion of the page that's currently visible in the browser window, even though there's much more content beyond its edges.

The use of viewports is crucial for responsive web design. On a desktop, the viewport is usually large and wide, but on a smartphone it's much smaller. In the early days of smartphones, browsers would shrink desktop pages to fit tiny screens, which made text unreadable and forced users to pinch-zoom. To fix this, developers began using the `viewport` meta tag to control how pages are displayed on different devices. Without a proper viewport, a site designed for desktops might appear tiny on a phone, forcing constant zooming.

Here is an example of it's use in an **HTML** file:

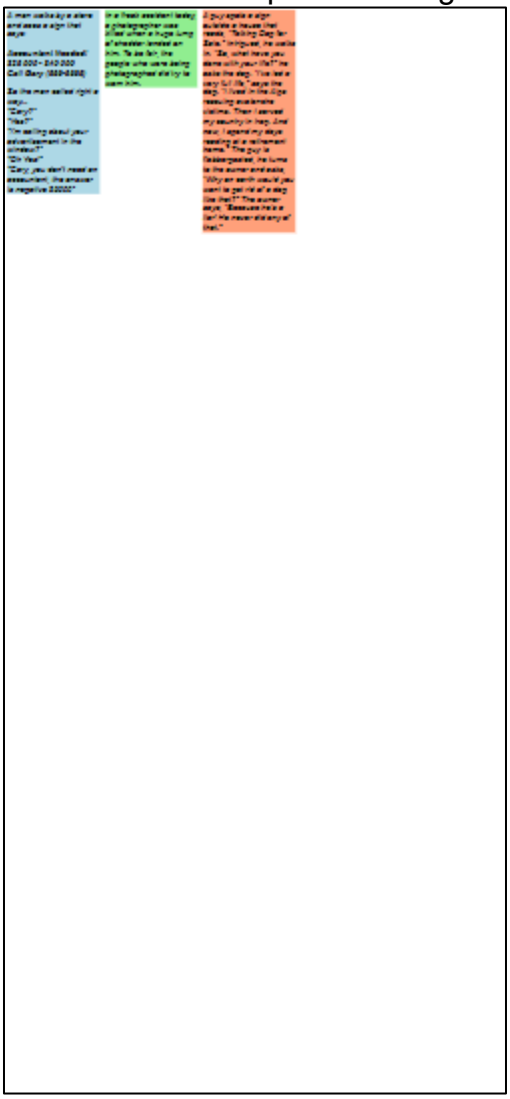
```
<meta name="viewport" content="width=device-width, initial-scale=1.0, user-scalable=yes">
```

This tells the browser to match the page's width to the device's screen width and to show the page at a **1:1** scale, without zooming out. By setting the `user-scalable=yes`, we are allowing the user to zoom in and out. Optionally, we can set it to `no` but then users cannot zoom the page.

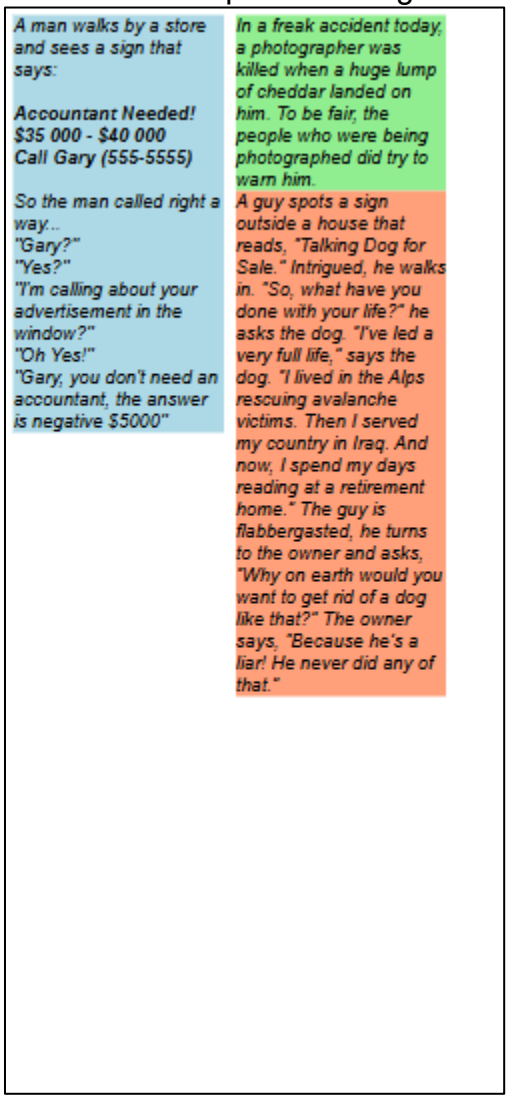
Allowing zooming improves accessibility (e.g., people with vision impairments can enlarge text) … and disabling it can violate accessibility guidelines. Since the default is `yes`, we don't even have to include this attribute.

Here is the difference (on an iPhone 14 Pro Max) between specifying the viewport meta and not specifying it when using our simple **HTML** file called **viewport.html**:

Without viewport meta tag



With viewport meta tag



Clearly, when the viewport meta tag is used, the text is much larger and easier to read. However, in some cases the web page must be scrolled horizontally to see the entire page if the web page hasn't been optimized for mobile use.

We often want a web page's content to take up a fixed percentage of the visible browser area. To make this easy, modern browsers support **CSS *viewport units***, which size elements relative to the browser's viewport.

> ***A viewport unit** (*vw ***and** *vh*) **is a percentage of the browser viewport's** *width* **or** *height* **where** *1vw* **= 1% of the viewport's width and** *1vh* **= 1% of the viewport's height.***

For example, a box styled as follows will automatically resize as the browser window changes size, keeping its specified proportions relative to the screen:

```
.box {
    width: 50vw;   /* 50% of viewport width */
    height: 30vh;  /* 30% of viewport height */
}
```

We can also specify minimum and maximum viewport units **vmin** and **vmax** where **1vmin** = the smaller of **1vw** and **1vh** and **1vmax** = the larger of **1vw** or **1vh**.

For example, the following **.square** uses **vmin** so its size is always based on the smaller dimension of the viewport. This keeps it perfectly square, even if the screen is very wide or very tall. And **.banner** uses **vmax** so its height is based on the larger dimension, making it stay prominent on both landscape and portrait screens:

```
.square {
    width: 50vmin;   /* 50% of the smaller viewport dimension */
    height: 50vmin;  /* stays square even if viewport is wide or tall */
    background-color: lightseagreen;
}

.banner {
    height: 20vmax;  /* 20% of the larger viewport dimension */
    background-color: coral;
}
```

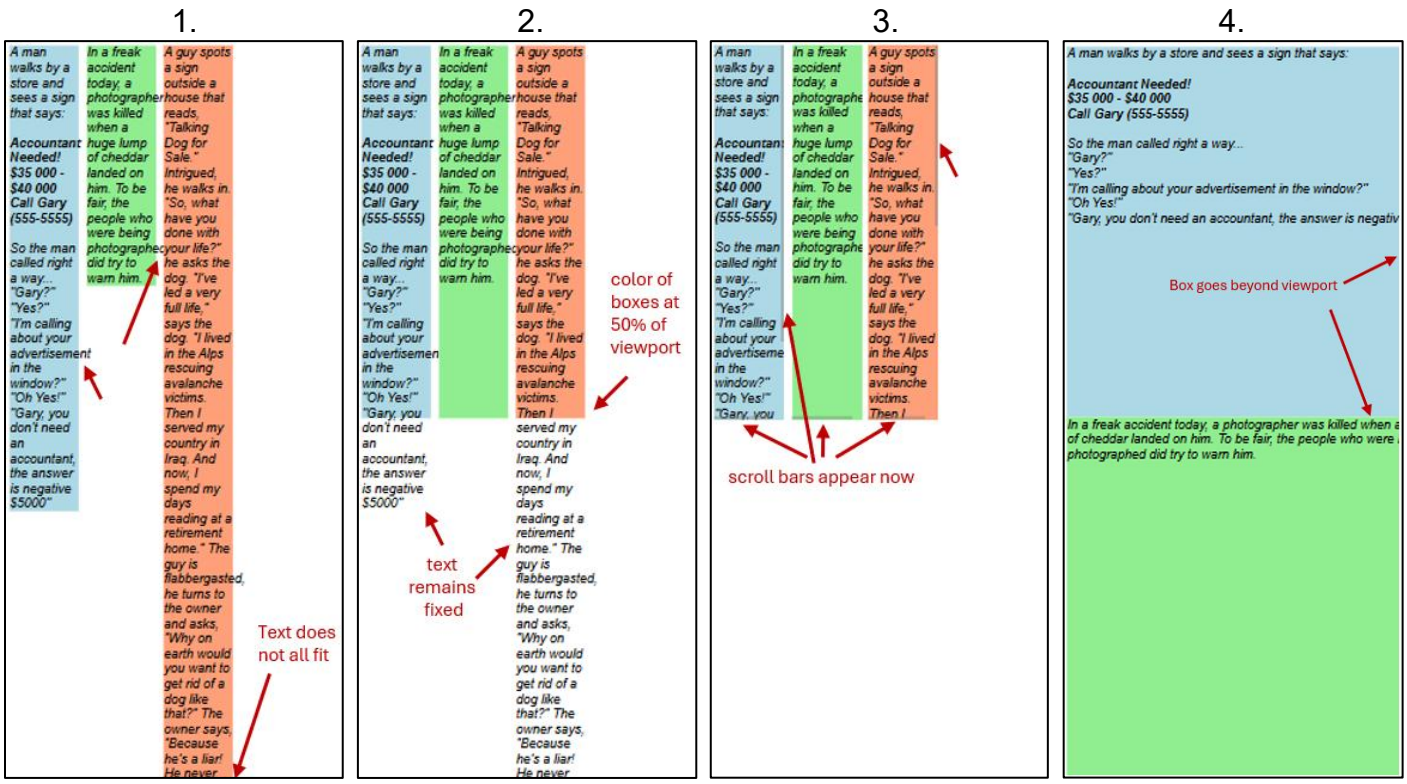
Our **viewport.html** file has this style definition:

```
.joke {
    font-family: arial;
    font-style: italic;
    width: 180px;
    padding: 5px;
    float: left;
    margin-right: 10px;
}
```

Let's try making a few adjustments to the viewport:

1. What would happen if we change the **width** to **20vw** instead of **180px**?
2. What would happen if we add **height: 50vh;** to the .joke class and re-render?
3. What would happen if we add **overflow: auto;** to the .joke class?
4. What would happen if we change the **width** to **120vw** ?

Let's see if we understand by looking at the results….



We should design our websites to make optimal use of the entire viewport.

To make the layout adapt naturally to different viewport sizes, developers use ***fluid layouts***, where elements scale proportionally rather than staying fixed.

> ***A fluid layout (also called a** liquid layout**) is a web page design approach where the widths of elements are set using percentages rather than fixed pixels.***

This allows the content to expand or shrink automatically to fill the browser window or viewport, making it flexible across different screen sizes.

While `vw` and `vh` units let elements scale based on the viewport width, sometimes we want elements to scale relative to their parent container instead. This is where ***percentage-based widths*** come in: using **%** allows a fluid layout that adapts naturally within its containing element, giving more control over multi-column or nested designs.

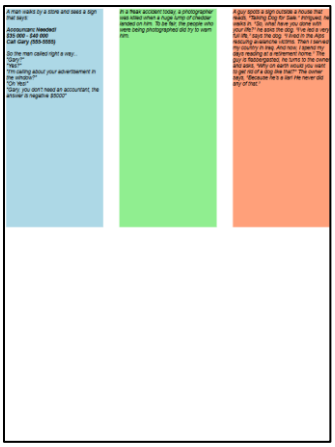
Here is a modified version of our **viewport.html** file called **viewport2.html** that makes use of percentages to have a fluid flow (although I took out the joke text to save space in these notes):

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>ViewPort Fluid Layout</title>
    <style type="text/css">
```

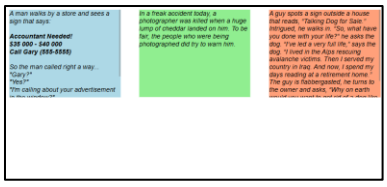
```
        /* This will fill the viewport width */
        .container {
            width: 100%;
            margin: 0 auto;
        }
        .joke {
            font-family: arial;
            font-style: italic;
            width: 30%;
            float: left;
            height: 50vh;
            margin-right: 5%;
            overflow: auto;
        }
        .joke:last-child {
            margin-right: 0;  /* remove margin on last column */
        }
        .joke1 { background: lightblue; }
        .joke2 { background: lightgreen; }
        .joke3 { background: lightsalmon; }
    </style>
</head>

<body>
    <div class="container"></div>
        <div class="joke joke1"> ... </div>
        <div class="joke joke2"> ... </div>
        <div class="joke joke3"> ... </div>
    </div>
</body>

</html>
```

Notice that all the jokes are in one container now. The container takes **100%** of the viewport. Each joke takes **30%** of their parent (which is the container). There is a **5%** (of the parent) margin on the right of each joke … except for the last joke … which has a margin of **0**. The total of the **3** jokes and the **2** margins in between is **100%**. Look at how everything looks now on various devices:

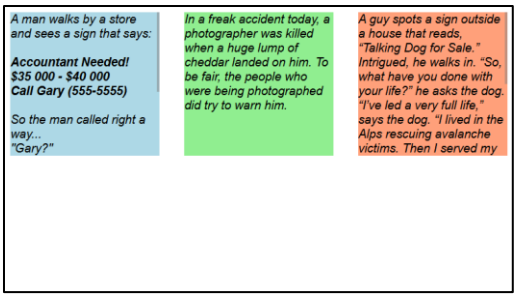
iPhone 14 (portrait)



iPad Air (portrait)



iPhone 14 (landscape)



iPhone SE (landscape)

What is the difference between using **vw** & **vh** versus using **%**?

> **vw** and **vh** size elements relative to the viewport (i.e., the browser window), while **%** sizes elements relative to their parent container.

Using relative units like **%** helps create a fluid layout, allowing elements to expand or shrink naturally with the container, and media queries can then fine-tune the design at specific breakpoints for a fully responsive page. Let's discus that next.

15.5 Media Queries

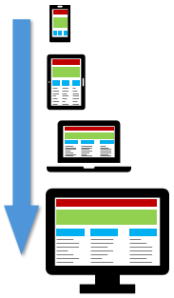
There are two general strategies for designing websites for all phones, tablets, desktops and laptops:

1. **Graceful Degradation (Desktop-First)**

   The idea is to begin by designing a rich, full-featured desktop version of the site. Then we simplify or adjust the design to work on tablets and mobile devices. This may be a good approach if our primary audience uses desktops, or if we already have a desktop design and need fast mobile compatibility. However, it can sometimes result in limited mobile functionality (i.e., fewer features) or performance issues on smaller devices, since the original design was not optimized for them.

2. **Progressive Enhancement (Mobile-First)**

   The idea is to begin by designing for the smallest screens, focusing on only the essential features. Then we scale up to larger devices, by adding enhancements such as additional layout components, richer interactions, and advanced styling. This approach prioritizes performance and usability on mobile devices, leads to cleaner **HTML** and better-organized **CSS** styles and often produces faster, more accessible websites. The trade-off is that it requires more upfront planning and forethought.

Progressive enhancement is often considered the best approach because it …

- ✓ Ensures everyone, regardless of device or browser, can access the core functionality of the website.
- ✓ Allows users with older devices, slower connections, or limited capabilities to still use the site effectively.
- ✓ Keeps the site fast and efficient by loading only essential features initially.
- ✓ Makes the website easier to maintain and scale by layering enhancements in a structured, organized way.

One way to implement progressive enhancement is through **CSS *media queries***, which let us adjust layouts (at specific viewport sizes or orientations) and styling based on screen size. This ensures the site adapts seamlessly from mobile to desktop while keeping all essential features accessible.

Media queries won't work correctly unless the viewport is properly set:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

A common mistake is forgetting to include the **<meta>** tag, which can cause the page to render incorrectly on mobile devices.

The general signature for using a **CSS** media query is as follows (this is just an example template):

```
@media
      <media-type>
      (<media-feature1>),
      (<media-feature2>) and (<media-feature3>),
      (<media-feature5>) and (<media-feature6>) and (<media-feature4>),
      {
            /* CSS rules go here */
      }
```

Note that we can use **and** to combine some features (i.e., to be more specific in our query) and we use the comma (i.e. **,**) to broaden our selection (i.e., to allow more possibilities).

The **<media-type>** is optional as well as all of the **<media-feature1>** expressions. Each **<media-feature1>** must evaluate to true or false.

The media types describe different output devices, not just screens and the media features refer to such things as device or viewport width & height, screen resolution and device orientation.

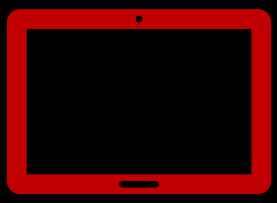
Let's consider first the various media types. There are three main media types: **screen**, **print** and **speech**. Each media type targets a different way that users experience content. The default is **screen**, in case we do not indicate it in our code.

We will look at these one at a time and discuss how some of the available features can be used for that type.

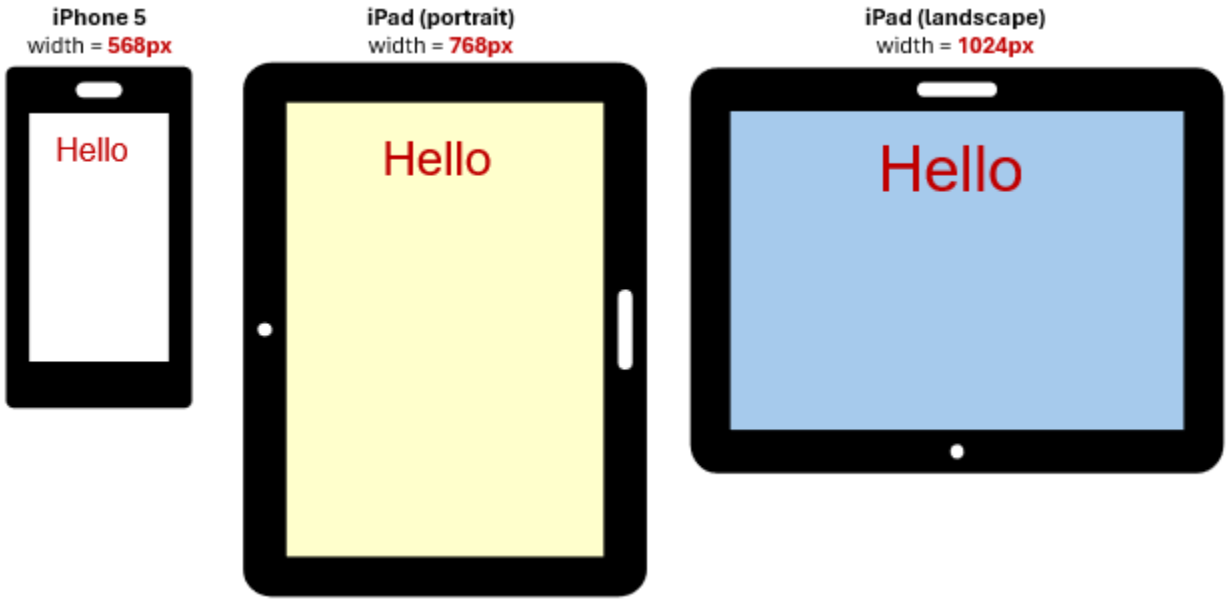
@media screen:

This indicates styles to be applied when the content is displayed on computer monitors, tablets, smartphones, or other screen devices. Screen media queries allow us to optimize layout, font sizes, colors, and spacing for different screen widths and resolutions, providing a better visual experience for users viewing content on electronic displays.

Here is an example of using this to adjust font size and color based on the viewport width:



```
/* Base styles for all screens */
body {
    font-size: 16px;
    background-color: white;
    color: red;
}

/* Styles that will apply for screens 600px wide and above */
@media screen (min-width: 600px) {
    body {
        font-size: 24px;
        background-color: #ffffcc; /* light yellow */
    }
}

/* Styles that will apply for screens 1024px wide and above */
@media screen (min-width: 1024px) {
    body {
        font-size: 32px;
        background-color: #a6caec; /* light blue */
    }
}
```



Note that the order of the last two is important because when two queries match, only the last one will be used. So, we should always list them in order from smaller screen sizes to larger ones.

A ***breakpoint*** is the screen width at which a media query takes effect. Best practice is to choose breakpoints based on the content layout rather than targeting specific devices. For example, it's better to hide a ** ** when the screen is wider than **700px** than to hide it only when viewed on an **iPhone X**.

Here are a couple of examples of when to use them for hiding content. Perhaps we want to hide sidebar navigation on small screens because they take up too much space on small devices. Instead, we might replace it with a hamburger menu instead. We can hide it like this:

```
@media (max-width: 600px) {
    nav.sidebar {
        display: none;
    }
}
```

Or maybe we have a large image that can crowd out content and slow page loads when using mobile devise. So, we could hide the banner on small screens:

```
@media (max-width: 600px) {
    .banner-image {
        display: none;
    }
}
```

Here is a table of the more commonly-used features:

Media Feature	Description	Example Usage
width	Viewport width	**(width: 800px)**
min-width	Min viewport width	**(min-width: 768px)**
max-width	Max viewport width	**(max-width: 480px)**
height	Viewport height	**(height: 600px)**
min-height	Min viewport height	**(min-height: 600px)**
max-height	Max viewport height	**(max-height: 1024px)**
orientation	Portrait or landscape	**(orientation: landscape)** or **portrait**
aspect-ratio	Width / height ratio	**(aspect-ratio: 16/9)**
min-aspect-ratio	Min width/height ratio	**(min-aspect-ratio: 4/3)**
max-aspect-ratio	Max width/height ratio	**(max-aspect-ratio: 16/10)**
resolution	Device pixel density	**(resolution: 2dppx)**
min-resolution	Minimum resolution	**(min-resolution: 2dppx)**
max-resolution	Maximum resolution	**(max-resolution: 300dpi)**
hover	device supports hover?	**(hover: hover)** or **none**
pointer	Type of pointing device	**(pointer: coarse)** or **fine**

The **`min-width`** and **`max-width`** features are by far the **most commonly used** for responsive layouts. Also, orientation is useful for tablets and phones.

Accessibility features are becoming more-and-more important. Here are a couple:

Media Feature and Options
`(prefers-color-scheme: light)` ... can be **`light`** or **`dark`** or **`no-preference`**
`(prefers-reduced-motion: reduce)` ... can be **`reduce`** or **`no-preference`**

There are also features like **`device-width`** and **`device-height`** … but these are not commonly used in modern responsive design because we typically focus on the viewport, not raw decide dimensions. A more comprehensive list can be found here:

https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_media_queries/Using_media_queries

Here is a typical example of how we might combine a lot of features so that we are properly handling various screen sizes and resolutions:

```
@media
    /* Small phones (portrait mode) */
    (max-width: 480px) and (orientation: portrait),

    /* Small phones (landscape mode) */
    (max-width: 767px) and (orientation: landscape),

    /* Tablets (portrait mode) */
    (min-width: 768px) and (max-width: 1024px) and (orientation: portrait),

    /* Tablets (landscape mode) */
    (min-width: 768px) and (max-width: 1024px) and (orientation: landscape),

    /* High-resolution screens (Retina or equivalent) with DPR = 2 */
    (min-resolution: 2dppx) {
        body {
            font-size: 16px;
            background-color: #f0f0f0;
            line-height: 1.5;
        }
        h1 {
            font-size: 2rem;
        }
    }
```

Notice how we have a set of media conditions so that the body and header will apply for any devices that meet any one of the specific widths/orientations/resolutions specified in the conditions.

`@media print:`

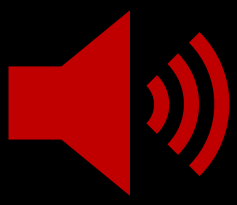
This indicates styles to be applied when the page is printed or viewed in print preview mode. When we print, the page layout, fonts, colors, and sizes might need to be different from a screen. Browsers will convert our **HTML** to a printable layout and the media query will apply to that rendering. Here is an example of what we may do to adjust the page style for printing:

```
@media print {
    body {
        font-size: 12pt;
        color: black;
        background: white;  /* remove screen backgrounds */
    }

    nav, .sidebar {
        display: none; /* hide navigation or sidebars when printing */
    }
}
```

`@media speech:`

This indicates styles to be applied when the content is read aloud by a screen reader or other text-to-speech device. Speech media queries let us adjust the reading order, hide non-essential visual elements, or modify content specifically for audio presentation, improving accessibility for users who rely on auditory output. Here is a typical way to use this to adjust for screen readers and text-to speech devices:

```
@media speech {
    /* hide non-essential visual elements */
    nav, .sidebar, .ads, .decorative {
        display: none;
    }

    /* improve text readability for screen readers*/
    body {
        font-size: 16pt;
        line-height: 1.6;
    }

    /* maintain list structure, so lists are read properly */
    ul, ol {
        margin-left: 1em;  /* keeps indentation */
    }

    /* emphasize headings visually for screen readers */
    h1, h2, h3 {
        font-weight: bold;
    }

    /* highlight important text for emphasis when read out loud */
    .important {
        font-weight: bold;
        text-decoration: underline; /* semantic cues help some screen readers */
    }
}
```



In this example, hiding navigation, sidebars, ads, and decorative elements makes the reading smoother and avoids unnecessary repetition. Adjusting font size and line height helps the speech engine interpret the content clearly. Keeping lists indented ensures screen readers read them correctly as sequences of items. Headings are emphasized to convey importance, and any particularly important items are styled with bold and underline cues, which some screen readers use to signal emphasis when reading aloud.

`@media all:`

This indicates styles to be applied to all media types, unless overridden by a more specific media query. It is the default media type if none is specified, so these styles will apply to screens, print, and other output devices. Using all ensures a consistent base style across all contexts.

We can also specify media queries inside of links. In this case, we can attach a stylesheet to a specific media query:

```
<link rel="stylesheet" media="(min-width: 1024px)" href="large-screen.css">
```

Here, the browser checks the media query when it parses the **<link>**. If the query matches (i.e., if it is **true** for the current viewport), the stylesheet is downloaded and applied. If the query does not match, the stylesheet is not applied (although it may still be downloaded by some browsers). So, the browser decides whether to use the external stylesheet.

15.6 Responsive Images

In web design, images are often the largest assets on a web page (i.e., they account for the most transmitted bytes). So, serving them correctly is essential for both performance and user experience.

> ***Responsive images** are images that automatically adjust to different screen sizes, resolutions, and device types.*

When dealing with images during responsive webpage design, we want to ensure that:

- images scale to various sizes without becoming pixelated or blocky.
- devices with higher **DPR**s receive higher-resolution images to maintain sharpness.
- devices with lower **DPR**s receive images at the minimum resolution necessary, avoiding unnecessarily large files that waste network bandwidth.

Why is it a concern? Because low **DPR** images appear terrible on high **DPR** screens. So, an image that looks nice and clear on a lower resolution device may look horrible (i.e., pixelated, blocky or blurry) on a higher-resolution device. Also, if we take a low resolution image and try to display it at the same size as a high resolution one, we will end up with an image that look pixelated, blocky or blurry.

Here is an example of what we could see when trying to display lower resolution images at the same size as higher-resolution images:


High res: 2560×2560

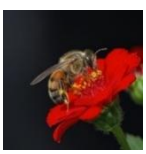

Medium res: 1080×1080


Low res: 480×480

To display raster images (e.g., JPEG, PNG, GIF) at the correct resolution on webpages, it's best to prepare multiple versions of the same image at different resolutions. This approach allows the browser to choose the most appropriate version:

- On retina or high-**DPR** devices, serve the highest-res image to keep it sharp.
- On standard or mid-range displays, serve a medium-res image to balance quality and file size.
- On low-resolution displays (or when using thumbnails), serve the lowest-res image to conserve bandwidth and improve load times.

For example, suppose we want to display an image at **100 x 75** logical pixels on the page. On a 3x **DPR** screen (e.g., iPhone 16 Pro Max, Samsung S24, Huawei P10), that space actually uses **300 x 225** physical pixels (i.e., (100 x 3) **X** (75 x 3)). If we only serve a **100 x 75** image, the browser would have to stretch it to fill **300 x 225** device pixels … which would make it look blurry or pixelated. To keep the image crisp, we should provide a **300 x 225** version for 3x **DPR** devices. Here are 3 images:


250 x 250


400 x 400

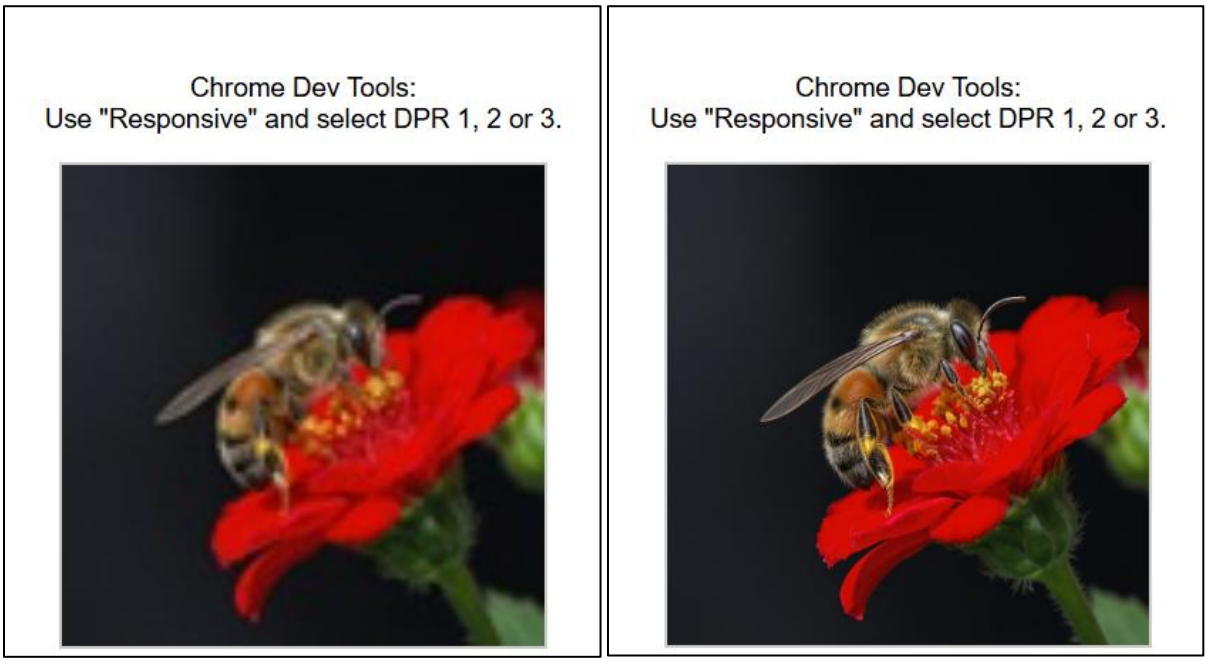

1024 x 1024

We can use the **<`img srcset`>** attribute specifies which image should be displayed for specific **DPR** values. Here is an example of how to include it in an **HTML** file:

```
<img
     src="bee-small.jpg"
     srcset="
         bee-small.jpg 1x,
         bee-medium.jpg 2x,
         bee-large.jpg 3x"
     alt="Bee on flower">
```

The first `src` attribute specifies the default image to use if the browser is unable to use `srcset` for some reason. The `srcset` attribute lists multiple versions of the same image along with their device pixel ratio (DPR) factors. This allows the browser to automatically select the most appropriate image: the small image for **DPR 1** devices, the medium image for **DPR 2** devices, and the large image for **DPR 3** devices.

We can test this by opening **test-resolutions.html** in **Chrome DevTools** by selecting a **Responsive device** in the device toolbar. Then, from the **DPR** drop-down menu, choose **1**. This effectively simulates a **DPR 1** device, forcing the browser to select the **1x (small) image** from the `srcset`. This is a simple way to demonstrate how different **DPR** images are chosen without needing an actual low-**DPR** device. We will be able to see what the low-res image would look like on a **DPR 3** device vs. a hi-res image on the device:



We can also use the `sizes` attribute to tell the browser how large the image will appear on the page in **CSS** (i.e., logical) pixels, depending on the viewport width. The browser uses this information to pick the best image from `srcset`. However, with **DPR**-based 1x/2x/3x descriptors that we just used, `sizes` is **optional**, because the browser primarily looks at **DPR**. But it can still help if we want the image to scale differently at different viewport widths.

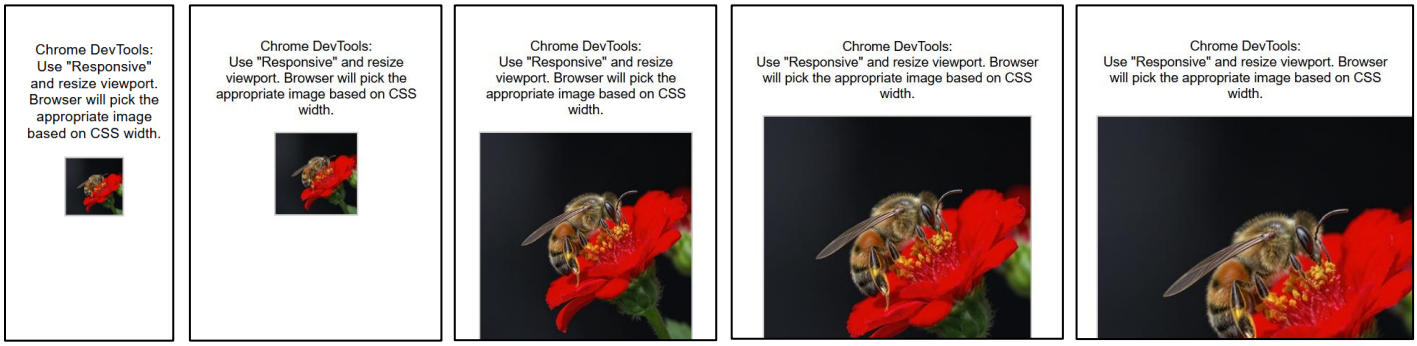
The `sizes` attribute is particularly useful when using width descriptors (w) in `srcset`. It tells the browser how wide the image will appear in logical **CSS** pixels for different viewport widths. The browser can then compare this display width with the widths listed in `srcset` and automatically choose the best image to show. Here is an example of using `sizes` in this way:

```
<img
    src="bee-small.jpg"
    srcset="
        bee-small.jpg 250w,
        bee-medium.jpg 400w,
        bee-large.jpg 800w"
    sizes="(max-width: 400px) 80vw, (max-width: 800px) 400px, 800px"
    alt="Bee on flower">
```

Here, the browser will look at the viewport width and if this with is …

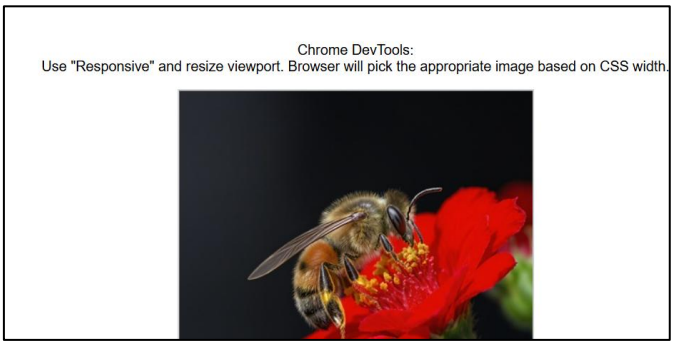
- <= 400 pixels … the image will occupy 80% of the viewport width.
- > 400 and <= 800 … the image will be 400 pixels wide.
- > 800 pixels … the image will be 800 pixels wide.

This only works if we remove **`width: 80vw;`** from the **img** tag style settings. We can try out the ***test-resolutions2.html*** file to see how it responds as we vary the screen size in Chrom Dev Tools. Make sure to select the **Responsive** mobile device at the top of the dropdown list of devices. Look at how the image size and resolution varies as we stretch the window to make it wider:

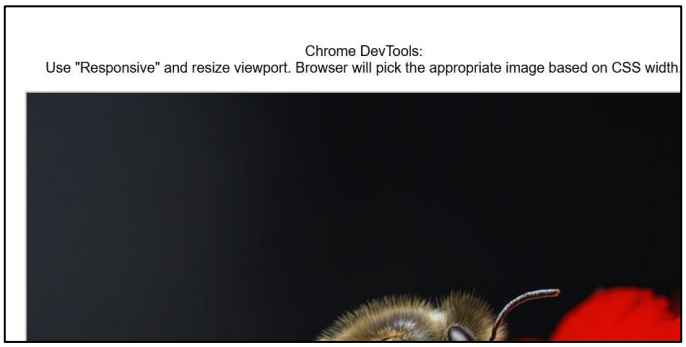


200 x 400	299 x 400	312 x 400	399 x 400	401 x 400
bee-small.jpg	bee-small.jpg	bee-medium.jpg	bee-medium.jpg	bee-medium.jpg

We will find that the image switches from small to medium when the width of the window reaches **312**. Why? Well, our code says to show the image at **80%** of the viewport width. **80%** of **312** is **249.6px**. Our **250** pixels small image will fit. But … the browser estimates how blurry each choice would look at a display width of **~250 px**. It prefers to err slightly on the *larger* side to keep images sharp even if scaling or rounding occurs, so it rounds up and adds a small ***fuzz factor*** (**~1.1×** or so). We will notice the image switch again when it it's a width of **800** …



799 x 400
bee-medium.jpg



804 x 400
bee-large.jpg

Notice that at **400**, the image stays a fixed size of **400px** (since we specified this in our `sizes` attribute). Then at **800** width it switches over to a fixed size of **800px** (also specified in our `sizes` attribute).

One question that may arise is … "Why does the small image not show at **80%** of the viewport … because isn't that what we specified in the `sizes` attribute?"

Well, when we use `(max-width: 400px) 80vw` in the `sizes` attribute … this is not a rendering instruction. It just tells the browser "Assume the image will be **80%** of the viewport when deciding which file to download." Once the file is chosen, **CSS** takes over to determine how wide it actually displays. Since we do not have a **CSS** width specified, the image just stays at **250 px** (or whatever file was picked).

If we want the small image to take up **80%** of the viewport, but have the other two images stay at a fixed size, we would need to add this to our styles:

```
/* Small viewports: scale image to 80% of viewport */
@media (max-width: 400px) {
    img {
        width: 80vw;      /* small image stretches/shrinks */
        max-width: none;
    }
}
/* Medium viewports: lock at 400px */
@media (min-width: 401px) and (max-width: 800px) {
    img {
        width: 400px;
    }
}
/* Large viewports: lock at 800px */
@media (min-width: 801px) {
    img {
        width: 800px;
    }
}
```

The HTML **<picture>** tag lets us to provide multiple image files, each defined in a **<source>** tag, so the browser can choose the best one for the situation (i.e., different screen sizes, pixel densities, or image formats).

The **<source>** tag provides a media condition (e.g., "only use this if the viewport is wider than **800px**") and a file to use if that condition is true. The browser checks each **<source>** in order and uses the first one that matches.

We must include a final **<img>** tag as a fallback. If none of the **<source>** conditions match, or if the browser doesn't understand **<picture>**, it will download and show the **<img>** file.

Here is a very common use of this tag. It uses the large image when the width is greater than **400**, the medium when between **250** and **400**, and the smaller image otherwise. The **test-resolutions3.html** has the width: **80%** set so that the image always takes **80%** of the viewport but now the browser chooses the best quality image only when the lower quality ones will look blurry (i.e., we set according to the file sizes):

```
    <picture>
        <source srcset="bee-large.jpg" media="(min-width: 400px)">
        <source srcset="bee-medium.jpg" media="(min-width: 250px)">
        <img src="bee-small.jpg" alt="Bee on a flower">
    </picture>
```

As a last thing to mention, it is ideal to use **Scalable Vector Graphics** (**SVG**) if they are available since they scale nicely to any resolution. It is an image/drawing define with curves, lines, points etc… that will always look crisp when resized.

Here is a **test-svg-scaling.html** file that displays an **svg** file, a corresponding **jpg** file for the same image and a manually-created **svg** drawing. Notice how they scale as we zoom in:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
    </head>
    <body>
        <!-- Resize the browser width to see the images grow or shrink -->
        <img src="responsive.svg" alt="Responsive Design Drawing" style="width:70vw">

        <!-- Circle with checkmark inside -->
        <svg viewBox="15 15 70 70" style="width:20vw">
            <circle cx="50" cy="50" r="30" stroke="orange" stroke-width="4" fill="lightblue" />
            <path d="M60 36l-15 15-7-7-5 5 12 12 20-20z" fill="green"></path>
        </svg>

        <!-- zoom in on this image to compare -->
        <img src="responsive.jpg" alt="Responsive Design Drawing" style="width:70vw">
    </body>
</html>
```

SVG file looks good

JPG file looks bad

Manual SVG looks good

There is a lot more that we can learn about responsive design images. Feel free to browse around…

Using the Viewport Meta Element:

https://developer.mozilla.org/en-US/docs/Web/HTML/Guides/Viewport_meta_element

Using Media Queries:

https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_media_queries/Using_media_queries

Using Responsive Images in HTML:

https://developer.mozilla.org/en-US/docs/Web/HTML/Guides/Responsive_images

Images in Markup:

https://web.dev/articles/responsive-images#images_in_markup

The Picture Element:

https://web.dev/learn/design/picture-element