HyperionDev

# Responsive Web Design

## Task

Visit our website

## Introduction

You should now feel a bit more proficient at creating simple web pages using HTML and CSS. At this point, you may have noticed that the elements on the web page remain rigid and do not adapt comfortably to the dimensions of different screen sizes, or accommodate accessibility, such as the use of screen readers.

A site that looks good on a PC may become squashed and cluttered on a mobile device. Similarly, a site that looks good on both a PC and a mobile device also needs to accommodate users with poor eyesight, hearing, or mobility.

Responsive design aims to build web pages that detect the visitor's screen size and orientation and change the layout accordingly. This will be our focus for this task. Let's begin.

## What is responsive web design?

Responsive design is a method of creating a web application that is able to adapt to different screen resolutions while maintaining interactivity. In other words, the exact layout will vary, but the relationship between elements and the reading order remains the same. The responsive design approach combines the following components apart from HTML and CSS:

1. Flexible design layouts
2. Responsive images and units
3. Media queries

### Take note

It is important to note that responsive web design – **coined by Ethan Marcotte in 2010** – was a term used to describe an approach to a web design. Since then, responsive design has become the default approach with web frameworks, making it easier to design responsive sites.

# Responsive units

## The viewport

The viewport is the screen size where the web page is in view.

CSS has both absolute and relative units of measuring the viewport's dimensions. An example of an absolute unit of length is a **cm** (centimetre) or **px** (pixel). Relative units or dynamic values depend on the screen's size and resolution or the root element's font sizes.

The more common relative/responsive units are:

- `em`: relative unit based on the font size of the parent element
- `rem`: relative unit based on the font size of the root element
- `vh`; `vw`: percent of the viewport's height or width
- `%`: percentage of the parent element

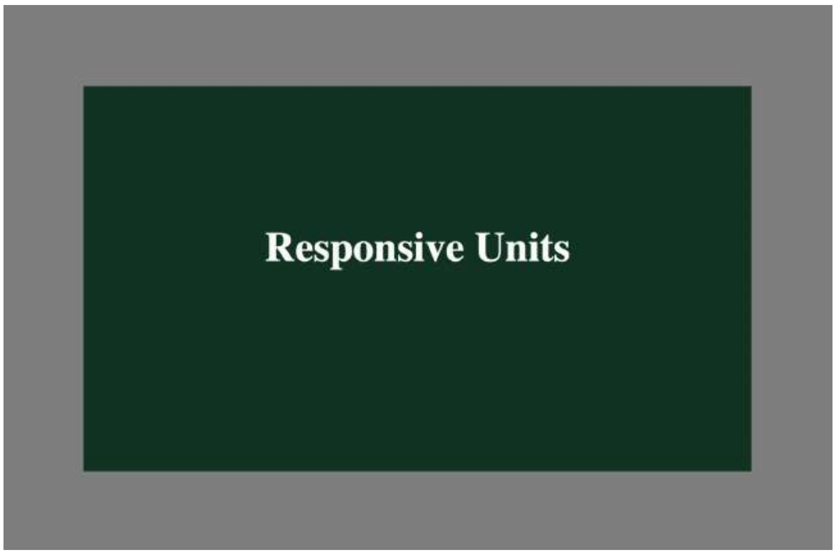
Viewport units are beneficial when an element's width and height must be specified relative to the viewport because they depend on the viewport dimensions.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Responsive Units Example</title>
    <style>
      * {
        box-sizing: border-box;
      }

      body {
        text-align: center;
        margin: 0;
        background-color: grey;
      }

      h2 {
        font-size: 5vw;
        color: white;
        padding-top: 25vh;
      }

      .centered {
        width: 80vw;
        height: 70vh;
        margin: 15vh auto;
        background: #123524;
      }
    </style>
  </head>
  <body>
    <div class="centered">
      <h2>Responsive Units</h2>
    </div>
  </body>
</html>
```

The example above uses a viewport unit to centre the element vertically. To do this, we need to know the element height. In this case, the element's height has been set to 70vh, which means that it will take up 70% of the viewport height.

To calculate the remaining space around the element, the following formula can be used [(100 – element height) / 2]vh. For example, with a height of 70vh, the formula becomes [(100 – 70) / 2]vh, which equals 15vh. The result of this formula will tell us how much space will be split between the top and bottom of the element.

This approach allows for an element to be vertically centred, as shown in the image below:



## Media queries

An adaptive web design approach uses media queries. These allow us to run a series of tests (e.g., whether the user's screen is greater than a certain width or resolution) and apply CSS selectively to style the page appropriately for the user's needs.

The different media types are:

- All: default, which matches all devices
- Print: used with printers
- Screen: fits devices with a screen
- Speech: fits devices with text-to-speech functionality

The `media screen` query allows the web page to respond to different screen sizes by applying specific styles based on the screen's viewport dimensions. It helps the page automatically adjust its layout to match the size of the device being used. See the code below:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Responsive Units Example</title>
    <style>
      * {
        box-sizing: border-box;
      }

      body {
        text-align: center;
        margin: 0;
        background-color: grey;
      }

      h2 {
        font-size: 5vw;
        color: white;
        padding-top: 25vh;
      }

      .centered {
        width: 80vw;
        height: 70vh;
        margin: 15vh auto;
        background: #123524;
      }

      @media screen and (max-width: 700px) {
        .centered {
          width: 80vw;
          height: 70vh;
          margin: 15vh auto;
          background: blue;
        }
      }
    </style>
  </head>
  <body>
    <div class="centered">
      <h2>Responsive Units</h2>
    </div>
  </body>
</html>
```

We've modified the code that centres the element in the earlier example. The `media` query checks the screen (the media type), and if the width dimensions are less than or equal to 700px, it changes the background colour of the centred element to blue. The `and` keyword combines a media feature with a media type or other media features.

You can have multiple queries within a stylesheet to monitor and adjust the page layout and elements to best suit various screen sizes.

The point at which the query triggers a change in the website layout is known as a breakpoint. You can declare breakpoints as a specific viewport width value.

You only need to add in a breakpoint and change the design when the content starts to look bad. It is important to note that a responsive layout might appear quite different on a desktop versus a tablet or smartphone.

A mobile-first design approach would be creating a single-column layout and implementing a multiple-column layout when there is sufficient screen width to handle it. The `media` query will check for the `min-width` parameter to trigger changes as the screen size increases. If you choose this approach, you would only need to include mobile breakpoints if you optimise the design for specific models.

With a desktop-first approach, we use the parameter `max-width` instead of width because the styles need to be constrained below a specific viewport size with decreasing screen width. The example above depicts a desktop-first approach, but a mobile-first approach takes precedence.

This approach ensures that the design remains adaptable and visually appealing across various devices.

## Flexible design layouts

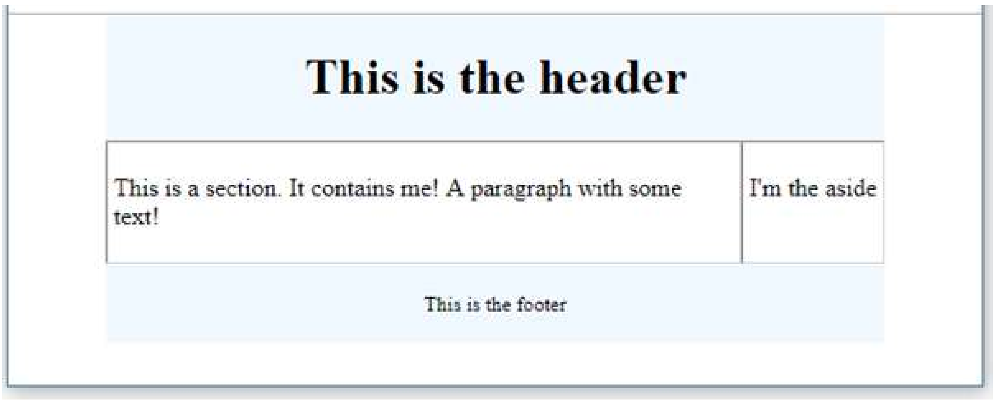
With a flexible design, the widths of page elements will be proportional to the width of the screen or browser window. Flexible design ensures that the layout remains consistent.

### Grid layout

Assume that we have created the following HTML:

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Position Example</title>
    <link rel="stylesheet" href="example_style.css" />
  </head>
  <body>
    <div id="container">
      <header><h1>This is the header</h1></header>
      <section>
        <p>This is a section. It contains me! A paragraph with some text!</p>
      </section>
      <aside><p>I'm the aside</p></aside>
      <footer>
        <p><small>This is the footer</small></p>
      </footer>
    </div>
  </body>
</html>
```

We want this HTML to be displayed in the browser as shown in the image below:

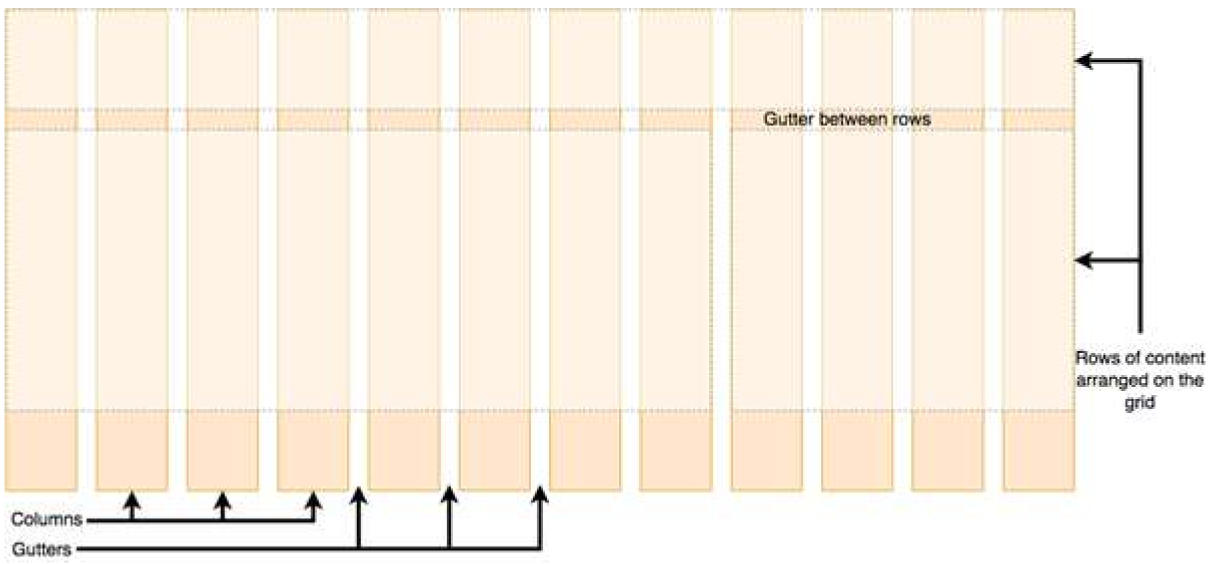

This is the header

This is a section. It contains me! A paragraph with some text!

I'm the aside

This is the footer

It would be difficult to get this layout, so we can use a CSS grid template to achieve this. A CSS grid is like a table that is designed to make it easier to position elements on a web page. The grid usually contains 12 columns, as depicted below:



*CSS grid framework (Mozilla, n.d.)*

The position of an element is described in terms of which row it is in and how many columns it takes up.

To illustrate, look at the CSS rules below:

```
body {
  width: 80%;
  margin: auto;
}

#container {
  display: grid;
}

header {
  grid-column: 1/13;
  grid-row: 1;
  background-color: aliceblue;
  text-align: center;
}

section {
  grid-column: 1/9;
  grid-row: 2;
  padding: 4px;
  border: 1px inset lightgrey;
}

aside {
  grid-column: 9/13;
  grid-row: 2;
  padding: 4px;
  border: 1px inset lightgrey;
}

footer {
  grid-column: 1/13;
  grid-row: 3;
  background-color: aliceblue;
  text-align: center;
}
```

We want the `<header>` element to span the full width of the whole container/grid. Thus, we specify that we want it to start at the first line (1) of the grid and end at the last line (13) of the grid. To position the `<header>` element at the top of the grid, put it in the first row of the grid.

Place the `<section>` element in the second row of the grid. The element is eight columns wide, so the section element starts at the first line of the grid and ends at the ninth line.

Fluid grids are a standard implementation of the fluid design approach. A fluid grid breaks down the width of the page into several equally sized and spaced columns. You then position the page content according to these columns. Each fluid column expands accordingly when the viewport expands horizontally, as does the content within the columns.

To define a grid we use the grid value of the `display` property. In the example below, we have added three columns to the grid using the `grid-template-columns` property. All child elements in the grid container become grid items.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Grid Example</title>
    <style>
      .grid-container {
        display: grid;
        grid-template-columns: auto auto auto;
        background-color: purple;
      }

      .grid-item {
        background-color: plum;
        text-align: center;
        border: 1px solid black;
        padding: 10vh;
        font-size: 3vh;
      }
    </style>
  </head>
  <body>
    <div class="grid-container">
      <div class="grid-item">1</div>
      <div class="grid-item">2</div>
      <div class="grid-item">3</div>
      <div class="grid-item">4</div>
      <div class="grid-item">5</div>
      <div class="grid-item">6</div>
      <div class="grid-item">7</div>
    </div>
  </body>
</html>
```

## Flexbox layout

Flexbox is a CSS module designed to more efficiently position multiple elements, even when the size of the contents inside the container is unknown. Items in a flex container expand or shrink to the available space.

Unlike grid layouts which use columns, a flexbox uses a single-direction layout to fill the container. A flex container expands items to fill the available free space or shrinks them to prevent overflow.

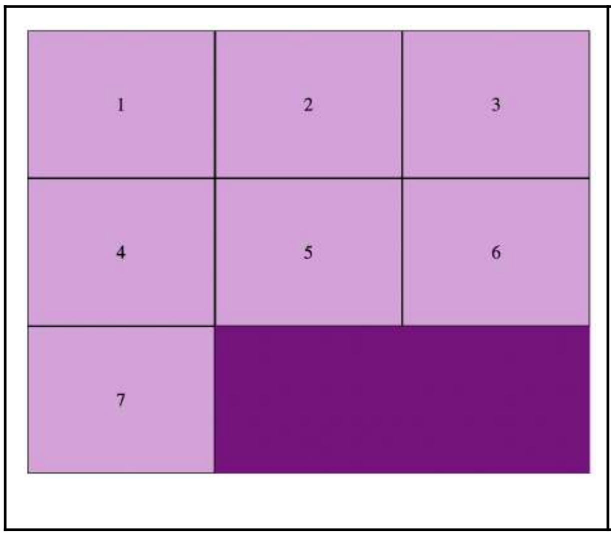
```
<!DOCTYPE html>
<html>
  <head>
    <title>FlexBox Example</title>
    <style>
      .flex-container {
        display: flex;
        flex-flow: row wrap;
        background-color: purple;
      }

      .flex-item {
        background-color: plum;
        text-align: center;
        border: 1px solid black;
        padding: 10vh;
        font-size: 3vh;
        flex: 1;
      }
    </style>
  </head>
  <body>
    <div class="flex-container">
      <div class="flex-item">1</div>
      <div class="flex-item">2</div>
      <div class="flex-item">3</div>
      <div class="flex-item">4</div>
      <div class="flex-item">5</div>
      <div class="flex-item">6</div>
      <div class="flex-item">7</div>
    </div>
  </body>
</html>
```

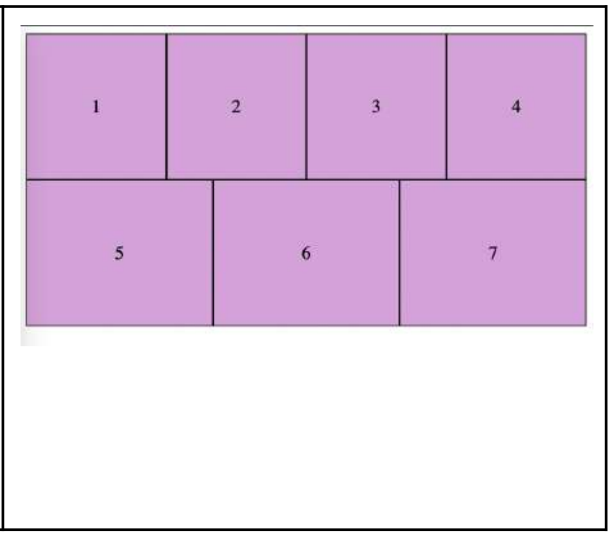
In the example above, we have defined a flexbox using the `flex` value of the `display` property. To specify the directional orientation of our flexbox items, we can use the `flex-direction` property, which establishes the main axis/direction by row or column. By default, `flex` items will all try to fit on one line, but we can change that using the `flex-wrap` property, which allows items to wrap as needed.

The `flex-flow` property is a combination of these two, and as you can see in the example below, it allows us to specify direction and wrapping.

We also added the rule `flex: 1` to our flex item above. This is a flex property that is a unitless proportion value that dictates how much space each flex item will take up along the central axis compared to other flex items. In this case, we're giving each `.flex-item` element the same value (a value of 1), which means they'll all take up an equal amount of the spare space left after properties like padding and margin have been set. An element with a flex value of 2 will take up twice as much of the available space.

**Grid container** **Flex container**





The image on the left is a grid container made up of seven grid items. We can see that the items are aligned with the three columns that have been declared. Take note of the available grid space, represented by a darker purple.

The image on the right is a flex container made of seven items. All items per row shared the available space equally and succeeded in filling the box regardless of the screen dimensions.

### Extra resource

Multi-column layout is a CSS module that adds support for multiple-column layout. Read here for more about **multi-column layout**.

## Responsive images

Responsive images follow the same concept as a fluid layout, using a dynamic unit to control the width or height.

One way to create a responsive image is by setting the `img width` property to a percentage value (remember the responsive unit?).

```
img {
  width: 100%;
  height: auto;
}
```

The snippet below ensures that the image is scalable but will never exceed its original using the `max-width` property.

```
img {
  max-width: 100%;
  height: auto;
}
```

The percentage unit approximates a single percentage of the viewport's width or height and ensures the image remains in proportion to the screen.

The problem with this approach is that every user has to download the full-sized image, even on mobile.

To serve different versions scaled for different devices, you need to combine the HTML `<picture>` tag with the `<source>` tag and the `srcset` attribute.

The `<source>` tag with the `srcset` attribute accepts several images with their respective widths in pixels. Used together with the `media` attribute (which accepts media queries), the image can be selected based on the viewport.

```
<picture>
  <source media="(min-width: 900px)" srcset="large-image.png" />
  <source media="(min-width: 700px)" srcset="medium-image.png" />
  <img src="small-image.png" alt="Responsive Picture" style="width: auto" />
</picture>
<p>
  Resize the browser width and the background image will change at 900px and
  700px
</p>
```

In the example above, the page can serve two different images depending on the viewport's dimensions. The larger of the images is set as the default. Should the width dimensions of the screen be equal to or less than 700px, the smaller image would be served.

## Accessibility

A good website is usable on devices with a variety of screen sizes. A great website is usable by people who use accessibility implements on their devices. Responsive design can greatly improve or hinder your website from an accessibility standpoint. Take, for example, the high contrast mode in Firefox. This mode makes content easier to view by stripping the page of background images and altering the background and text colours to make text more readable. If your website's content cannot be understood without the background image being present, then that is an accessibility issue as it locks out people with limited vision from being able to access your content.

Similarly, people who rely on screen readers may not be able to access your content if it relies on pictures with no alt text, and if your website structure does not allow for a screen reader to make sense of it, then you risk alienating more people with limited vision from accessing your content.

Here are some good practices to think about when designing any web page:

- Add descriptive alt text to all your images.
- Make use of HTML5 semantic tags, such as `<section>`, `<article>`, `<header>`, `<footer>`, `<aside>`, `<figure>`, and `<nav>`, and lean away from using non-descriptive tags like `<div>`.
- Use CSS to float your content around the page. Avoid relying on hacks such as invisible tables to position your content, as this can greatly confuse a screen reader.

Read this **web accessibility checklist** that goes into much more detail about accessibility.

## Take note

You can test your web page using multiple screen sizes with **Chrome DevTools**. You can select the mobile device or tablet of your choice to test the responsiveness of your design.

## Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.

## Auto-graded task

For this task, you will be recreating the **periodic table of elements**.

- The page should be responsive to changing screen dimensions until such a point where the table loses its structure, at which point it will be replaced by an image of the periodic table (making elements disappear can be done using the CSS `display` attribute).
- The table can be created using either the grid or flexbox layout. Both the flex and grid layouts have properties allowing one to manipulate the space between child items (i.e., `gap`, `row-gap`, and `column-gap`).
- Create at least one breakpoint that triggers a change in the styling of the web page.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

### Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Do you think we've done a good job or do you think the content of this task, or this course as a whole, can be improved?

Share your thoughts anonymously using this **form**.

## Reference list

Mozilla. (n.d.). *Grids*. MDN Web Docs. Retrieved June 18, 2024, from **https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Grids#A_CSS_Grid_grid_framework**