Sold to
ramonvilafer@gmail.com

STEP BY STEP GUIDE

CSS in 44 minutes

Build your own webpage from scratch

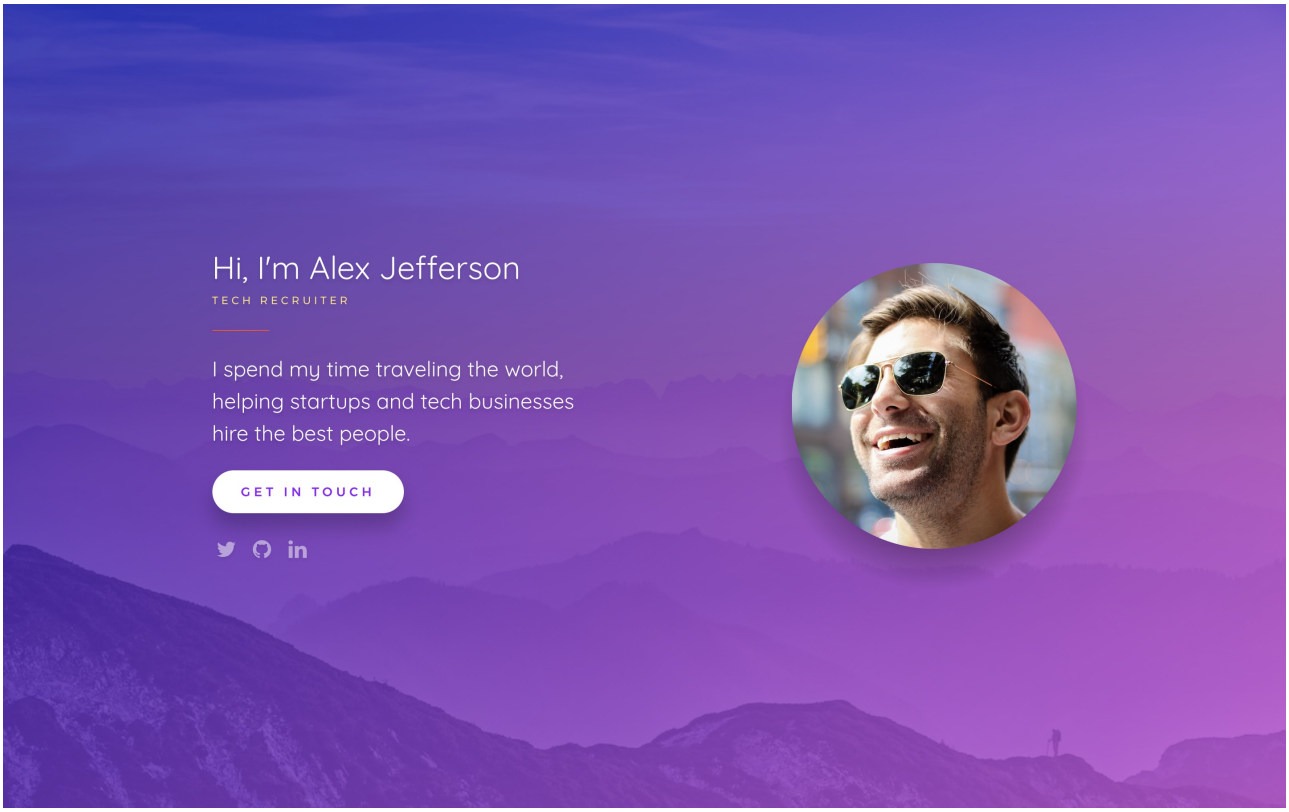


Written by
Jeremy Thomas

© Copyright 2018 – Jeremy Thomas

1 Introduction

This book is a **step by step** guide that will teach you how to build this webpage from scratch:



It's the webpage of a fictional tech recruiter called Alex Jefferson. All images come from Unsplash. As you follow this guide, feel free to change the styles and content the way you want.

Who is this book for

Anyone can read this book! No prior web development or programming knowledge is required, since I will tell you what to do, line by line.

What you will build

This webpage will use the following files:

- 1 HTML5 file
- 3 CSS files
- 4 images (provided)
- 1 JavaScript file

It will also make use of two third party services: Font Awesome and Google Fonts.

What you will learn

Through this guide, I will teach you how to:

- Set up a valid **HTML5** document
- Write **semantic** markup
- Insert responsive **Retina-friendly** images
- Learn how to **structure** your CSS correctly
- Design a 100% **responsive** page
- Understand how **media queries** work
- Use CSS **Flexbox** to lay out your components
- Style your text with **typography** properties
- Add visual feedback with CSS **Transitions**
- Bring your page to life with CSS **Animations**
- Make use of **Google Fonts**
- Include and style **Font Awesome** icons

What you need to start

You only need to install 2 programs:

- a decent **text editor** or IDE with syntax highlighting. I recommend Sublime Text but you can use Notepad++, Vim, Emacs, IntelliJ, Atom...
- a modern **web browser**. I use Google Chrome but you can use Firefox, Safari, Opera, or Edge.

You also need a few files that I have provided:

- 1 CSS file: `minireset.min.css`
- 7 images:
  - `alex.jpg`
  - `alex@2x.jpg`
  - `alex@3x.jpg`
  - `austria.jpg`
  - `1x.png`
  - `2x.png`
  - `3x.png`

Have them available and ready to be used.

2 Writing the HTML5 content

Open your text editor, create a new file and paste this code snippet:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Alex Jefferson – Tech Recruiter</title>
  </head>
  <body>
    <h1>Hello world!</h1>
  </body>
</html>
```

Save this file as `index.html`, open it in your browser, and you will see the following page:



> Make sure to have the `.html` file extension. Here's how to do it on Windows and Mac OS.

A valid and responsive HTML5 document

This page is a **valid responsive HTML5 page** because it satisfies the following requirements.

```
<!DOCTYPE html>
```

This line tells the browser that this webpage is an **HTML5 document** and should be interpreted as such.

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

This is the **responsive** meta tag. It tells the browser to set the *content* width to the *viewport* one, ensuring for the content to adapt automatically. It also tells the browser to set the zoom level to 1 initially, while still allowing the user to zoom in (especially on mobile phones).

Also, each HTML tag (`<html>` , `<head>` , `<body>` ...) is opened and closed in the right order. Only the `<meta>` tags need to be self-closing.

Adding the CSS reset

Before designing your page, you want to start with a **clean canvas**. Since each browser comes with its own default styles, you want to remove them first, otherwise these styles will clash with yours. This process is the purpose of a **CSS reset**.

> Best practice
>
> **Separate content and styling**
>
> You want to choose an HTML tag for its *semantic meaning*, not its *appearance*.
> If changing an element's tag in your HTML changes its appearance, then you're essentially styling your webpage in the HTML, which is what we want to avoid.

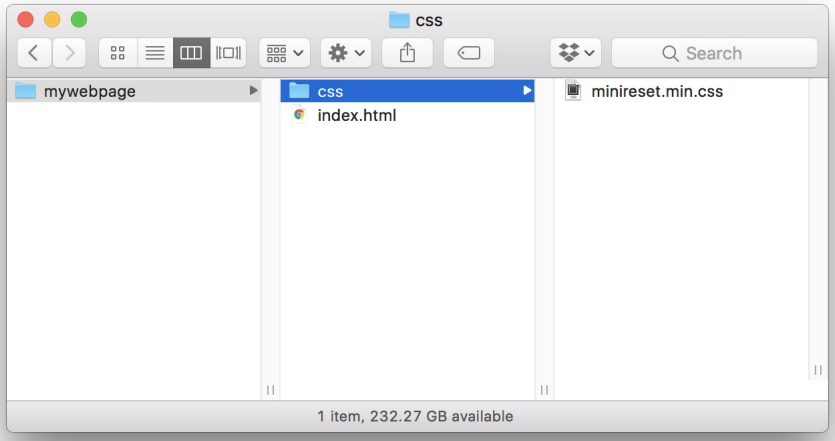
Most current CSS resets are either complicated or outdated, so I created a small one called minireset.css. I use it for all my projects, and it's included by default in Bulma too.

The only two real requirements are to:

- remove the margins and paddings from all block elements
- use `box-sizing: border-box`

Feel free to read through the source code.

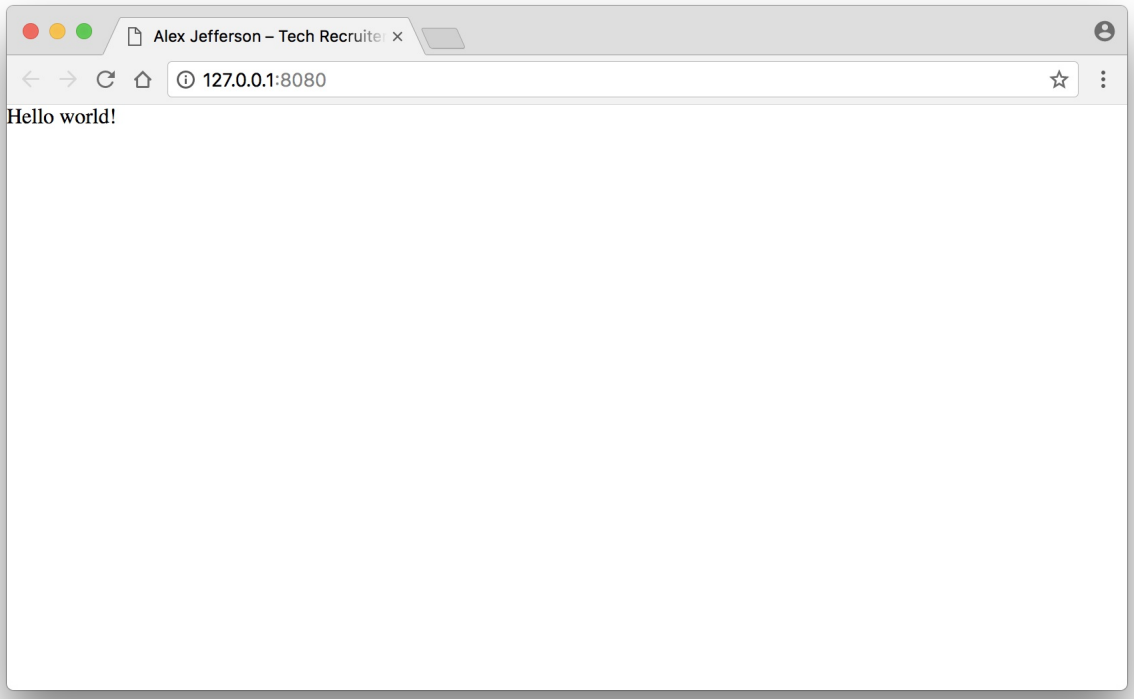
Alongside your `index.html` file, create a `/css` folder, and move the `minireset.min.css` file in it.



Let's **link** our CSS file *from* the HTML file. Just below the `<title>`, add this line:

```
<title>Alex Jefferson – Tech Recruiter</title>
<link rel="stylesheet" type="text/css" href="css/minireset.min.css">
```

The text `"Hello world!"` is now unstyled.



Your own CSS

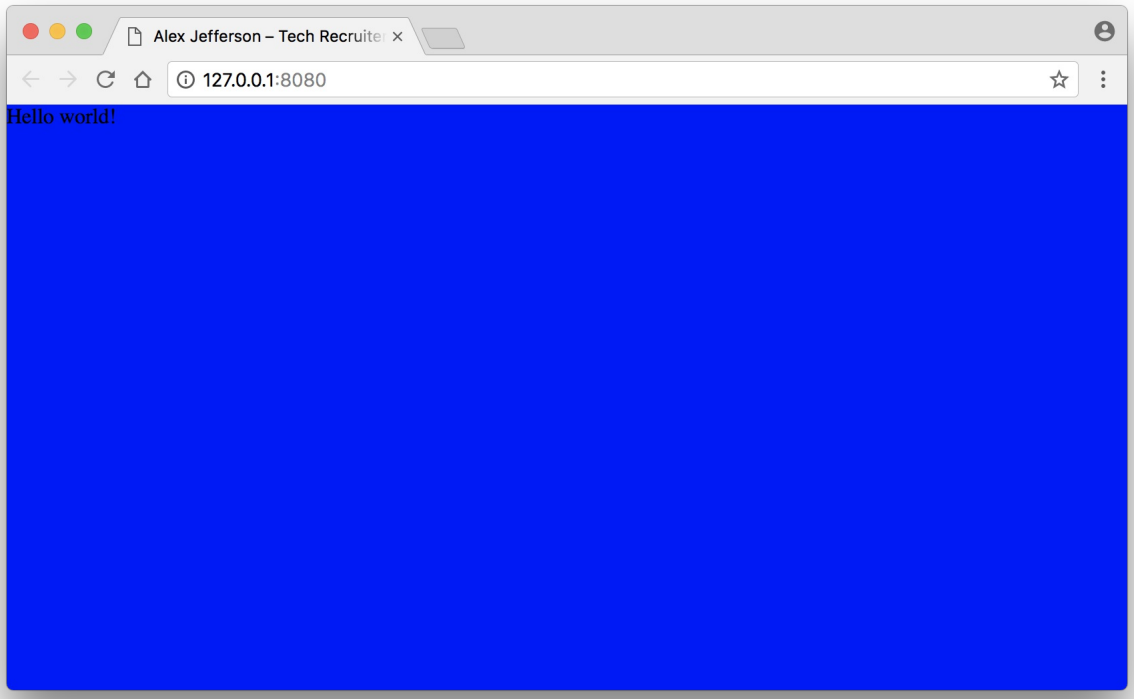
In the `/css` folder, create a new empty file called `main.css`.

You now have to include this CSS in your page. In `index.html`, right below the minireset, link your `main.css` file. You now have **2 CSS files**:

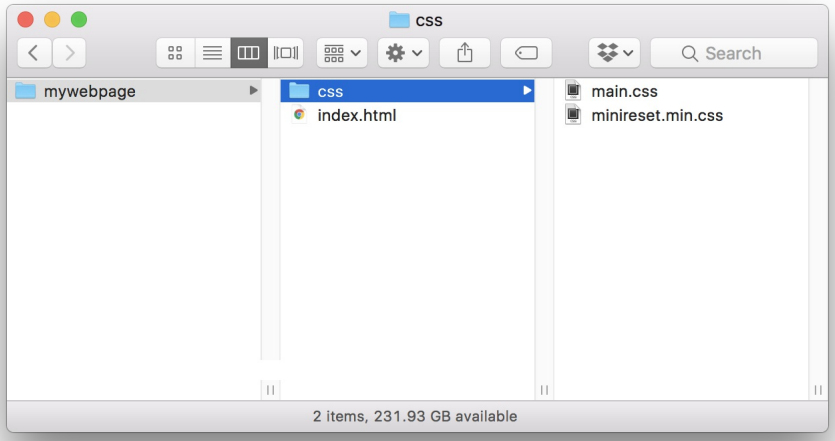
```
<link rel="stylesheet" type="text/css" href="css/minireset.min.css">
<link rel="stylesheet" type="text/css" href="css/main.css">
```

To see if the CSS was correctly included, add this to `main.css`:

```
html {
  background-color: blue;
}
```



If your screen doesn't turn blue, make sure your `index.html` file is alongside your `/css` folder:



The CSS now works so you can remove the blue background from the CSS. Before writing any CSS, you have to start with the most important of a webpage: the **content**.

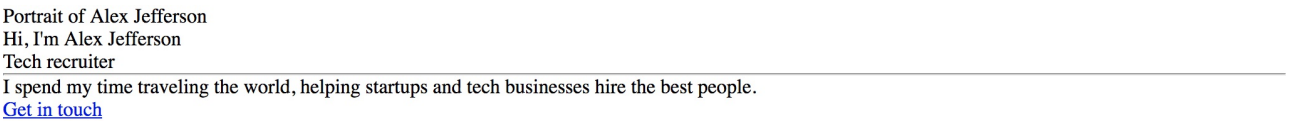
Writing content

The purpose of **design** is to enhance the presentation of the content it's applied to. It might sound obvious, but content being the primary element of a website, it should not be established as an afterthought. Written content makes up for more than 90% of the Web.

Remove `"Hello world!"` from your page, and put this code inside the `<body>`:

```
<div class="wallpaper"></div>
<div class="content">
  <aside class="side">
    <figure class="picture">
      <div class="picture-shadow"></div>
      <img id="pictureImage" class="picture-image"
           src="images/alex.jpg"
           srcset="images/alex.jpg 1x,
                   images/alex@2x.jpg 2x,
                   images/alex@3x.jpg 3x"
           alt="Portrait of Alex Jefferson"
           width="320"
           height="320">
    </figure>
  </aside>
  <main class="about">
    <h1 class="name">
      Hi, I'm Alex Jefferson
    </h1>
    <p class="job">
      Tech recruiter
    </p>
    <hr class="hr">
    <div class="description">
      <p>
        I spend my time traveling the world,
        helping startups and tech businesses
        hire the best people.
      </p>
    </div>
    <div class="contact">
      <a class="button" href="mailto:youremail@example.com">
        Get in touch
      </a>
    </div>
  </main>
</div>
```

As you can see, HTML is **noisy**: there is lots code for not much content on screen. But all these HTML tags are here for *semantic* reasons, and all these CSS class names are here for (future) *styling* purposes.



HTML structure

The `wallpaper` will display a transparent image in the background, covering the whole page.

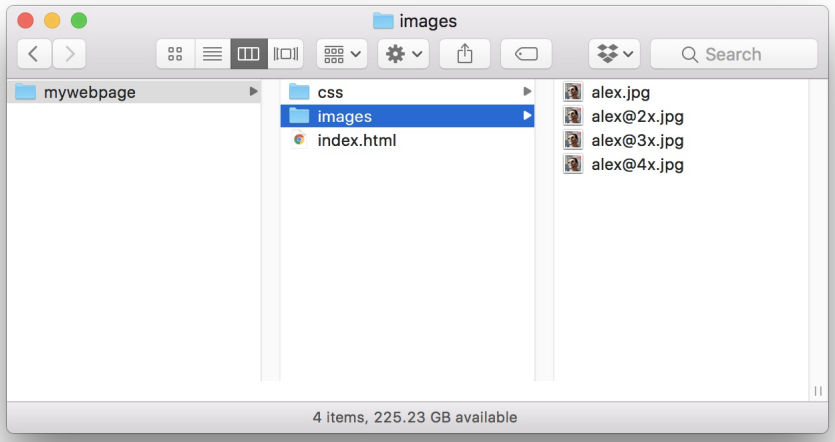
The `content` is the container for the readable part of the webpage, and will cover the whole page as well.

On the right, the `side` will display the portrait **image**.

On the left, the `about` section is a wrapper for all **text**. The `button` acts as a link for visitors to click, so make sure to replace `youremail@example.com` with your own email address.

Responsive and Retina-friendly images

As you can see, since the `alex.jpg` image is missing, the browser is showing the `alt` text "Portrait of Alex Jefferson" as a fallback. To fix this, create an `/image` folder alongside the `/css` one, and move the 4 images (`alex.jpg` , `alex@2x.jpg` , `alex@3x.jpg` and `alex@4x.jpg`) inside:



We want to display this image at a maximum of `320x320` pixels. But screens have different **pixel ratios**, especially mobile phones. For example, the Sony Xperia S has a pixel ratio of `2` , the Apple iPhone X `3` , and the Samsung Galaxy S8 `4` .

We could show the `1280x1280` version to everyone and resize it to `320x320` . But we would penalize users who own a device with a pixel ratio of `1` because they would end up downloading an image **10 times** the size that their device can actually display.

The solution is to:

- show the user the highest quality image possible their device can support
- prevent the browser from downloading the other images

This is possible thanks to the `srcset` attribute which tells the browser the list of image *alternatives* available for each pixel density, and let him figure out which image to display.

If the browser doesn't support `srcset` , it will use `src` as a fallback.



Hi, I'm Alex Jefferson
Tech recruiter

I spend my time traveling the world, helping startups and tech businesses hire the best people.
Get in touch

This is all the content that you need for now:

- a portrait image
- a name
- a job title
- a description
- a contact button

This content will be extended furthermore later. It's time to add your first lines of **CSS**.

3 Setting up a CSS base

Before styling individual elements, you need to set a global **base style** for your webpage.

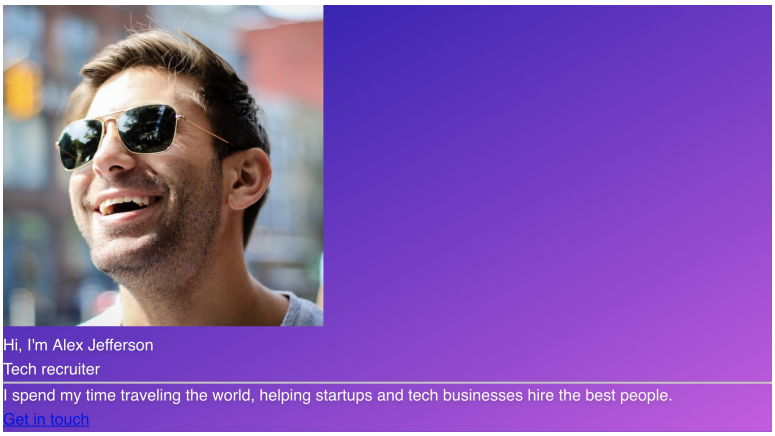
The html styles

The general styling of a webpage is done through styles applied to the `html` element because most of them are cascaded through all the other child elements.

In your empty `main.css` file, add this code:

```
/* 1. Base */

html {
  background-color: #5f45bb;
  background-image: linear-gradient(to bottom right, #180cac, #d054e4);
  color: #fff;
  font-family: "Quicksand", sans-serif;
  font-size: 16px;
  -moz-osx-font-smoothing: grayscale;
  -webkit-font-smoothing: antialiased;
  line-height: 1.5;
  min-height: 100vh;
  min-width: 300px;
  overflow-x: hidden;
  text-shadow: 0 3px 5px rgba(0, 0, 0, 0.1);
}
```



This rule-set is basic but does the job:

- the `background` and `color` declarations set the overall tone
- the `font` and `text-shadow` set the base typography
- the `line-height` and browser-specific `font-smoothing` declarations make the page more readable

- the dimensions and overflow values ensure that the page fills up the whole viewport

> Best practice
>
> **Order your CSS rules alphabetically**
>
> When editing a CSS file, finding the declaration to change can be time consuming.
> To reduce the **cognitive load** when both *reading* and *writing* rules, write them alphabetically.
> It's the only sorting rule that is future-proof because it's opinionless.
> When you will have more than 10 CSS declarations in a rule, you will be glad to know where to find what you were looking for in a split second!

The `background-color` usually acts as a fallback if the `background-image` doesn't load, but since we're using a `linear-gradient()` it will almost always appear. This gradient appears on top of the single-colored background, hence why we only see the gradient.

The font-family property takes a list of possible families. If the Quicksand font is not available, the browser will use the fallback font provided: `sans-serif`. This usually means Arial or Helvetica on Windows machines, San Francisco on Mac OS, and Ubuntu on Linux machines. Since your visitors will probably not have that font, we will include it using Google Fonts.

The `line-height` has a unitless value of `1.5`. It means each *line* of text will be 1.5 times the element's current font size. For most of the page, and combined with the `font-size: 16px` declaration, this will make each line of text `24px` high.

I always have a `min-width: 300px` that prevents the page from being too narrow to be readable.

The `min-height: 100vh` ensures the page to be at least as tall as 100% of the viewport height (`vh`).

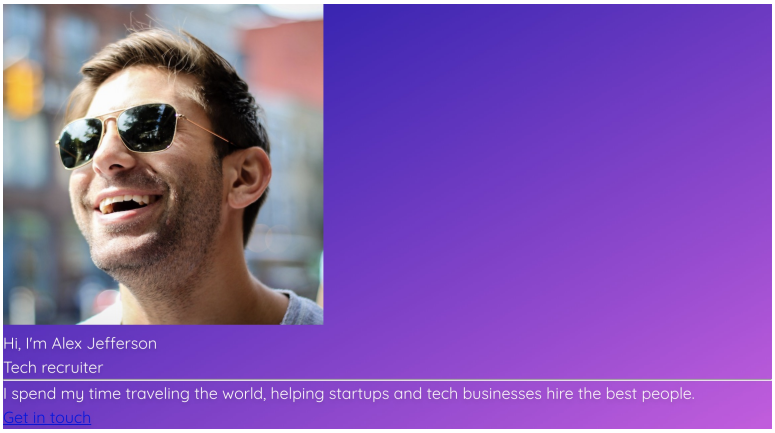
The `overflow-x: hidden` declaration prevents the page from scrolling horizontally while preserving the usual vertical scroll.

Adding the Google fonts

In `index.html`, before including `minireset.min.css`, add this CSS from Google Fonts. You now have **3 CSS files**:

```
<link href="https://fonts.googleapis.com/css?family=Montserrat|Quicksand" rel="stylesheet">
<link rel="stylesheet" type="text/css" href="css/minireset.min.css">
<link rel="stylesheet" type="text/css" href="css/main.css">
```

This adds two fonts to your page: Quicksand and Montserrat, both in regular font weight only (value of `400`). Reload the page to see the fonts in action:



Styling links

Links are by default blue and underlined. Add this in your CSS:

```
a {
  color: currentColor;
  cursor: pointer;
  text-decoration: none;
}
```

Using `currentColor` is prefered because it will pick up the color set by `html` before: `color: #fff`. Updating the color of both only requires a single line change.

Sometimes, the hand cursor doesn't show up when hovering a link. That's why I always add `cursor: pointer` to force the hand to show up.

"Get in touch" is now white as well:

Get in touch

4 Defining the layout

The wallpaper

At the end of `main.css` add this snippet:

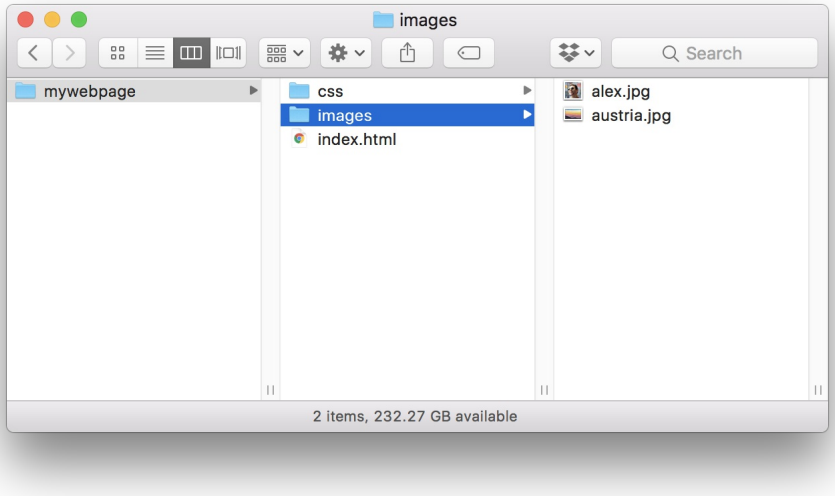
```
/* 2. Shared */

.wallpaper {
  display: block;
  height: 100%;
  left: 0;
  top: 0;
  width: 100%;
}
```

This is the `2. Shared` section of the CSS. Here, different selectors will share the *same set of rules*. We will extend this section as we need along the way.

For now, this rule ensures that the wallpaper covers the whole page.

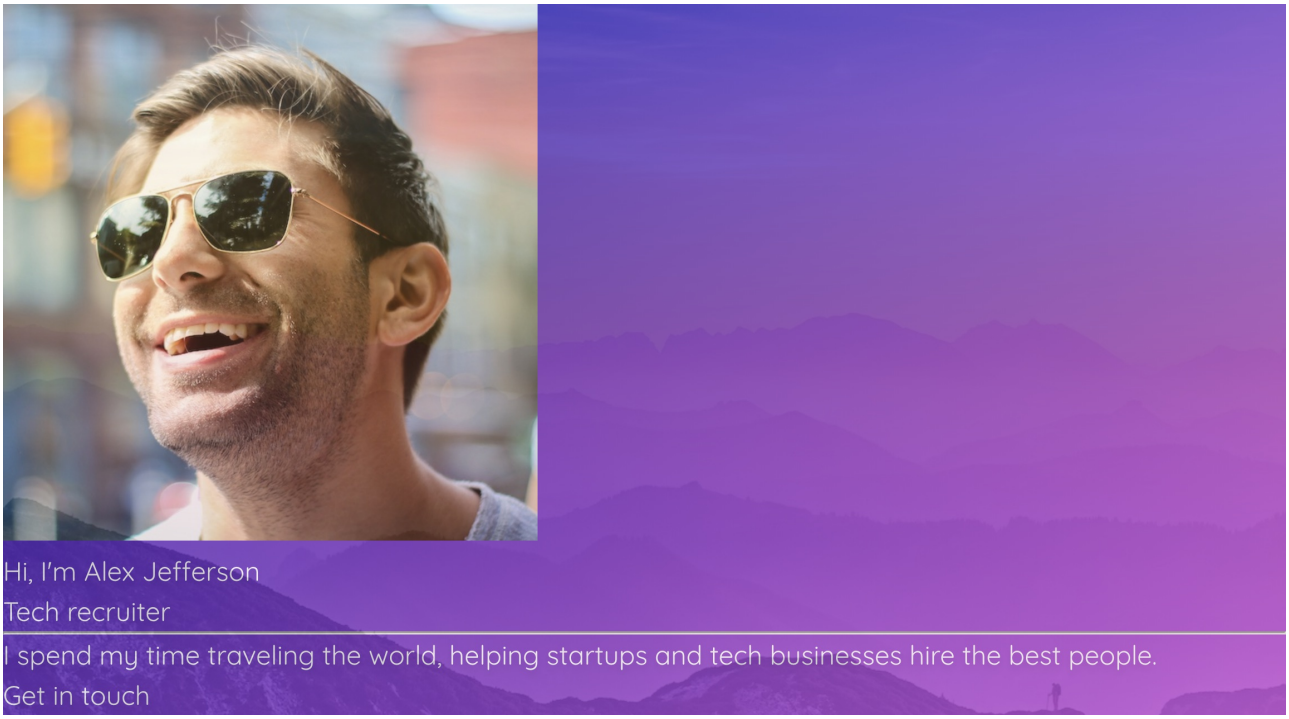
We now enter the `3. Specific` section where we will style each element individually. Add the `austria.jpg` file to the `/images` folder:



Then add this snippet at the end of `main.css` :

```css
/* 3. Specific */

.wallpaper {
  background-image: url("../images/austria.jpg");
  background-position: center;
  background-size: cover;
  opacity: 0.2;
  position: fixed;
}
```



> This image is not displayed with an `<img>` tag because its purpose is decorative and belongs in the CSS.

Thanks to the fixed positioning, the wallpaper will not scroll with the page.

> Best practice
>
> **Avoid the shorthand notation**
>
> We're using each `background-*` property instead of the shorthand `background` one.
> If you use `background: red`, you are essentially setting `background-color: red` and reseting all other properties to their `initial` value.
> It becomes a problem when you actually override a declaration you had previously written, and you wonder why your CSS line doesn't work!
> If `background-color` had been set earlier, it would have been undone here.
> The only useful shorthand properties are `margin`, `padding` and `border`.

Main layout

The `content` element is parent to both:

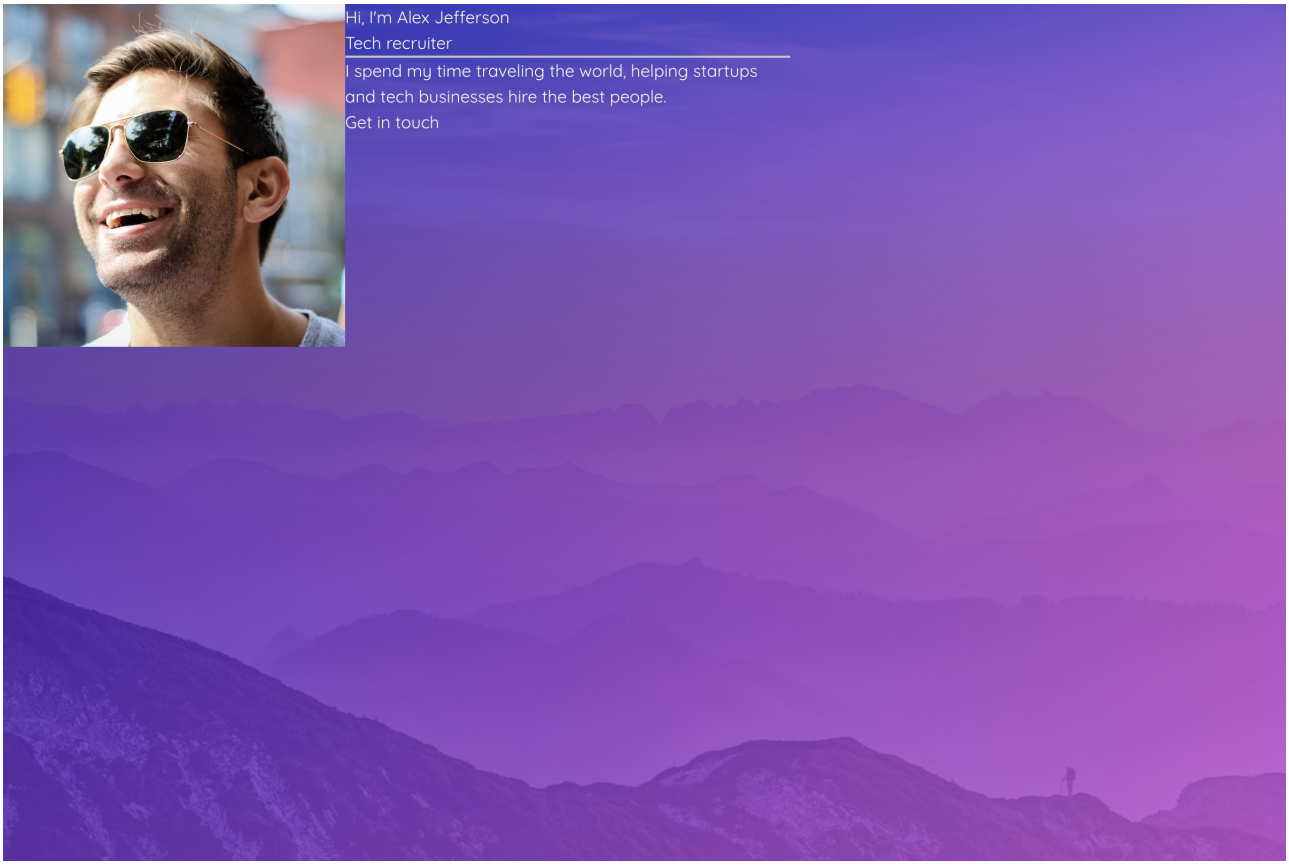
- `side` with the portrait image
- `about` with the text content

We're gonna use **CSS Flexbox** to set the global layout of the page.

```
.content {
  display: flex;
  position: relative;
  min-height: 100vh;
}

.side {
  max-height: 20rem;
  max-width: 20rem;
}

.about {
  max-width: 26rem;
}
```



We enable Flexbox on the `.content` by simply using `display: flex`. This makes both `.side` and `.about` Flexbox **items**.

The `position: relative` allows `.content` to appear above `.wallpaper`, and the `min-height` is here to make sure `.content` covers the whole page.

The use of `max-width` is very practical: it just means that at *any point* we don't want an element to by wider than a certain value. For readability reasons we want the `.about` section to never go beyond `26rem` in width (which is `26 x 16px = 676px`). Since it's a block element, it will use the whole width available up to a certain point. Depending on the length of your written content, you can play with the value here.

Best practice

Avoid z-index when possible

All positioned elements (`absolute` , `fixed` , or `relative`) can have a `z-index` value to stack them relatively to each other.
But elements are already stacked based on their location within the HTML code.
Because `.wallpaper` and `.content` are *siblings*, the second one `.content` will appear above (only if it's positioned too).
And child elements will appear *above* their parent.
It's better to place them as you want in your HTML, otherwise you'll have to keep track of all `z-index` values throughout your CSS.

Making the layout responsive

We want our layout to be **responsive**, meaning that it will be different on smaller viewports (below `800px`) than on wider ones (`800px` and above):

- on **mobile**, we're having a centered *vertical* layout
- on **desktop**, we're having a centered *horizontal* layout

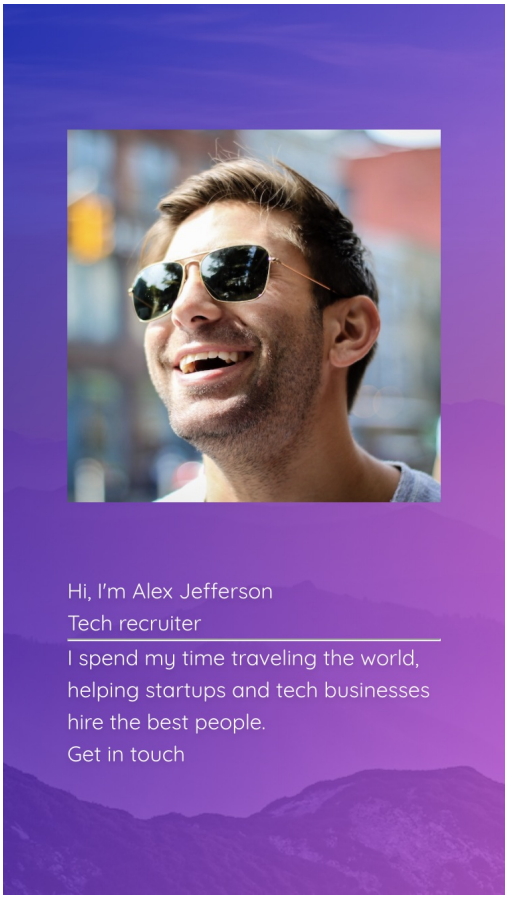
We're gonna write a **media query**: any CSS written inside that block will only be active if all the items in the media query list (`screen and (max-width: 799px)`) are **true**. In this case, we are targetting mobile devices who have a viewport narrower or equal to `799px` .

Luckily, even on your desktop you can simply resize your browser to see it in action.

```
/* 4. Responsiveness */

@media screen and (max-width: 799px) {
  .content {
    flex-direction: column;
    justify-content: center;
    align-items: center;
    padding: 5rem 3rem;
  }

  .side {
    margin-bottom: 3rem;
    width: 100%;
  }
}
```



> Resize your browser to see it in action.

We're using the following Flexbox properties:

- `flex-direction: column` makes the layout vertical
- `justify-content: center` makes the content centered on the **main** axis (vertical)
- `align-items: center` makes the content centered on the **cross** axis (horizontal)

The `padding` prevents the content from touching the viewport edges, giving it some space to breath.

The `.side` (which comes first) has a `margin-bottom` to separate it from the `.content` (which comes second).

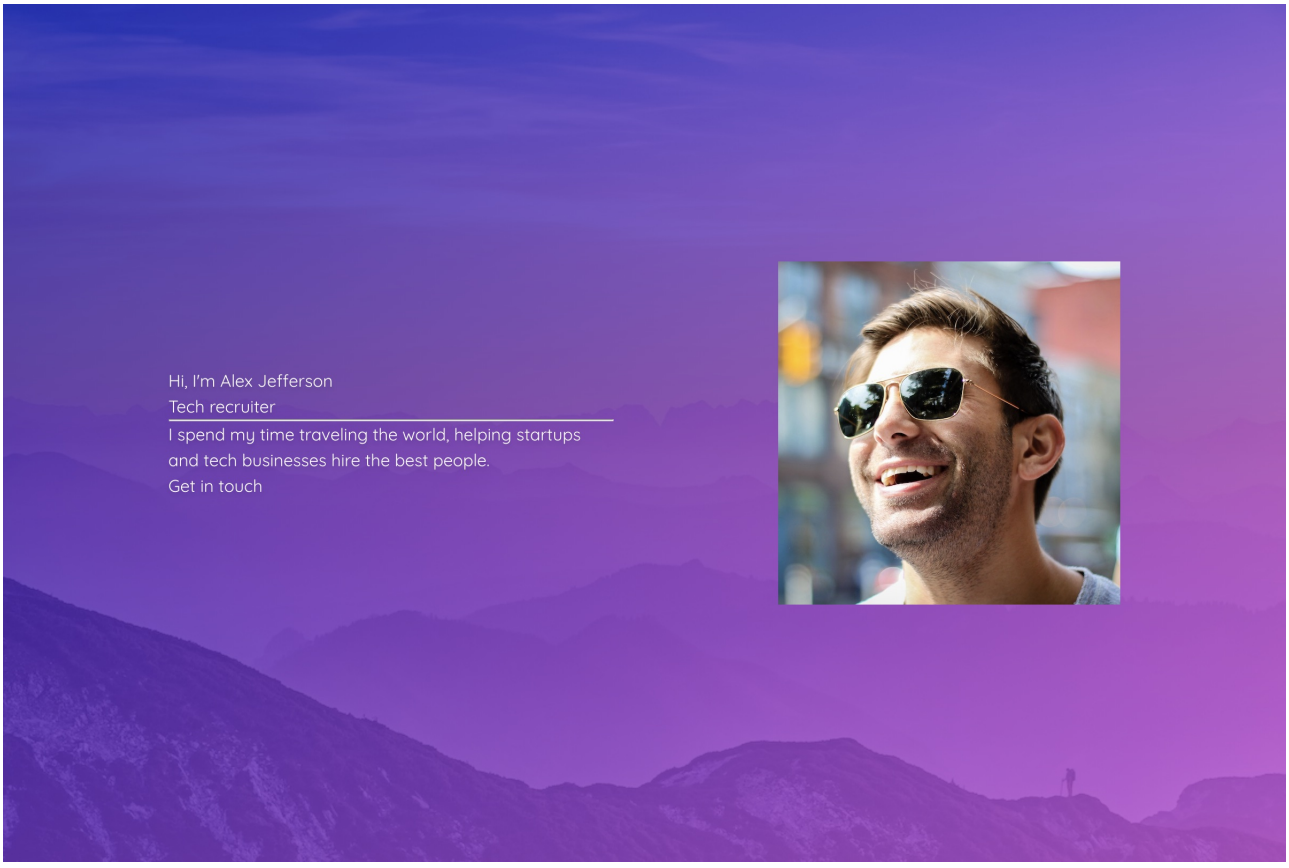
Desktop view

On desktop, we want the layout to be **horizontal** instead:

```css
@media screen and (min-width: 800px) {
  .content {
    align-items: center;
    justify-content: space-around;
    justify-content: space-evenly;
    padding: 4rem;
  }

  .side {
    flex-grow: 0;
    flex-shrink: 0;
    height: 20rem;
    margin-left: 4rem;
    order: 2;
    width: 20rem;
  }

  .about {
    flex-grow: 1;
    flex-shrink: 1;
  }
}
```



The `.content` uses the default value of `flex-direction` which is `row`, meaning the items (`.side` and `.about`) are spread out **horizontally**. `justify-content` appears twice because the `space-evenly` value is not available in all browsers, so we use `space-around` as a reasonable fallback.

The `.side` element (which contains the image), has fixed dimensions of a `20rem by 20rem` square. If there's more horizontal space available, we *don't* want it to grow, hence the `flex-grow: 0`. On the other hand, we also don't want it to shrink at all, otherwise, the image would be squased. That's why `flex-`

`shrink: 0` is used here too. We basically want the `.side` to be `20rem` by `20rem` at all times. We also want it to appear *after* the text content, which is why we use `order: 2`.

For the `.about`, we do want it to use the remaining space available, in both directions, which is why `flex-grow` and `flex-shrink` have both a value of `1`.

Best practice

In media queries, don't undo, just do

With mobile-first approaches, it's easy to make a layout work well on narrow screens, and then "undo" most of it on desktop. But that's tricky because you have to keep track of what has been done *outside* of media queries, and reset those values *inside* the desktop media query. You also end up writing a lot of CSS just to reset values, and you can end up leaving CSS like `margin-bottom: 0` you are not sure what.
The `margin-bottom` set for the `.side` element should **only** appear on *mobile*.
Instead of applying a margin by default on *all* screens, and removing it on *desktop*, we only apply it on *mobile*.

5 Styling all elements

The picture

The shadow is separated from the image because we are going to **animate** them separately later on.

In the `2. Shared` section of the CSS, update the list of **selectors** and add both `.picture-shadow` and `.picture-image`:

```
.wallpaper,
.picture-shadow,
.picture-image {
  display: block;
  height: 100%;
  left: 0;
  top: 0;
  width: 100%;
}
```

At the end of the `3. Specific` section of the CSS, add the following styles:

```
.picture {
  padding-top: 100%;
  position: relative;
  width: 100%;
}

.picture-shadow {
  border-radius: 290486px;
  background-image: radial-gradient(#000 0%, rgba(0, 0, 0, 0) 70%);
  position: absolute;
  top: 10%;
}

.picture-image {
  border-radius: 290486px;
  position: absolute;
}
```

The `padding-top: 100%` is a technique that makes the `.picture` as *high* as it is *wide* making it **square** at all times. It also has a `position: relative` so that it acts as a reference point for its two absolutely positioned children.

The shadow uses a semi-transparent `radial-gradient()`. It's slightly offset towards the bottom with `top: 10%`.

The `border-radius` is set to the very high value of `290486px` to ensure the elements to be **rounded**. You can use any extremely high value. I personaly use `290486px` as a trademark because it's my date of birth. It's quite interesting to see it show up in other people's code!



The name

The name is the most important information of the page. That's why it uses the `<h1>` HTML tag, which has the strongest semantic value. To reflect this prominence visually as well, we are gonna make it bigger:

```
.name {
  font-size: 2.25rem;
  line-height: 1.125;
  margin-bottom: 0.5rem;
}
```



The font size uses the `rem` unit: it's the **root** value, equal to the font size set on the `html` element, which we previously set to `16px`. So `2.25rem` is essentially `36px`. Using `rem` is useful because it references a common value, and we can update the `html` value to set *all* instances of `rem` values.

The `line-height` is set to `1.125` It's hard to see its purpose if the text is too short (resize your browser to reach two lines), but keeping the page value of `1.5` makes the two lines too spaced out.

> Best practice
>
> **Set line-height first, margin/padding second**
>
> Some developers use the line-height as a way to give space to an element. But as its name suggests, it's meant to define the height of a *single line*, not the space *between* each line.
> The line-height value should be set for **readability** purposes *only*.
> If you need to give more breathing space to an element, just use a bit of margin (or sometimes padding), which is what we're doing here with `margin-bottom`.

Job title

We want the job title to stand out a bit. That's why we are going to use our secondary font: **Montserrat**. This font works well when the letters are uppercase and slightly spaced out.

In the `2. Shared` section, add this snippet:

```
.job,
.button {
  font-family: "Montserrat", "Quicksand", sans-serif;
  letter-spacing: 0.3em;
  text-transform: uppercase;
}
```

This style is going to be used for our button as well, so let's add both selectors right now.

In the `3. Specific` section, add the following:

```
.job {
  color: #ffe479;
  font-size: 0.75rem;
}
```



The uppercase style makes the text quite "in your face", so we're reducing the font size a bit, and also applying a shade of yellow.

The hr line

The horizontal rule (`hr`) defines a semantic break between blocks of text. While keeping this semantic value intact, we want to make this line more subtle:

```
.hr {
  background-color: #ff470f;
  border: none;
  content: "";
  height: 1px;
  margin-bottom: 1.5rem;
  margin-top: 1.5rem;
  transform-origin: center left;
  width: 4rem;
}
```



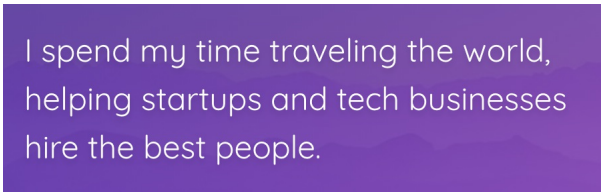
For more control, we're removing the `border` and using the `background-color` with a height of `1px` to define a thin short line.

The `transform-origin` will be used when we animate the width later on.

Description

The description only needs to be slightly more prominent. Let's increase the font size:

```
.description {
  font-size: 1.5rem;
}
```



Contact button

```
.contact {
  display: inline-block;
  margin-top: 1.5rem;
  vertical-align: top;
}
```

By using `display: inline-block` we combine two behaviors:

- `inline` makes sure the width is equal to its content (the button)
- `block` makes sure surrounding elements will appear above and below
- it also allows us to use `margin-top`

And `vertical-align: top` ensures the element to only use the vertical space required, and keeps the spacing tight.

We first need to add some CSS in the `2. Shared` section:

```
.button,
.social a {
  transform-origin: center;
  transition-duration: 100ms;
}
```

We're setting some transformation and transition values shared between the button and the social links (which are coming later).

The `transition-duration` is shared so we only need to replace the value in a single location if needed.

We can now focus on the `3. Specific` section for the button itself, which is the most elaborate element we have:

```css
.button {
  background-color: #fff;
  border-radius: 290486px;
  box-shadow: 0 1rem 2rem rgba(0, 0, 0, 0.2);
  color: #9013fe;
  display: inline-block;
  font-size: 0.875rem;
  line-height: 1;
  padding: 1.25em 2em;
  text-shadow: none;
  transition-property: box-shadow, transform;
  user-select: none;
  vertical-align: top;
  white-space: nowrap;
  will-change: box-shadow, transform;
}
```



The button uses the same `inline-block` technique as its parent.

The **height** of the button is **proportional** to the font size, and equal to `3.5` times the font size value:

- the unitless `line-height` is equal to `1` or `0.875rem`
- the vertical paddings (top and bottom) are set to `1.25em` each, or `1.25` times the font size

The `em` unit is equal to the *current* font size. In this case, it's similar to the unitless line height value.

If you play around with the `font-size` value, by setting `2rem` for example, you'll notice that the button will **resize proportionally**. This is very useful because we only need to update a *single value* to increase/decrease the size of the button, while maintaining a perfect ratio.

We also apply some other styles:

- because of the colored background and the darker text color, we remove the text-shadow
- we use `user-select: none` so the text content can't be selected, which can happen when clicking repeatedly on the button
- to make sure the text never is displayed on two lines, we use `white-space: nowrap`

The `transition-property` and `will-change` values are for the button states.

Button states

The button is the main interaction element of the page. Visitors can hover the button and click on it.

```css
.button:hover {
  box-shadow: 0 1.5rem 3rem rgba(0, 0, 0, 0.2);
  transform: scale(1.02) translateY(-4px);
}
```



The box shadow is transparent so it can work on any background color. For the transformation, we increase the button's size by 2% with `scale(1.02)` and move it upwards by a few pixels with `translateY(-4px)`.

When clicking the button, we want to make it look as if it was pressed downwards.

```
.button:active {
  box-shadow: 0 0.5rem 1rem rgba(0, 0, 0, 0.3);
  transform: scale(0.98) translateY(-2px);
}
```



We're just playing with the same properties but with different values. The shadow is stronger and smaller because the button appears closer to the "ground".

Play with the `transition-duration` value set in the `2. Shared` section to see it slower or faster.

6 Importing Font Awesome icons

Font Awesome is an icon library that allows you to easily include icons in your webpage. What it provides is essentially an **icon font** which simply is a text font where all the characters have been replaced with icons. So instead of typing **shapes** of *letters* you're essentially typing shapes of *icons*. This works because letters are vector shapes themselves. We just happen to be able to read them!

> We are including the whole Font Awesome library directly from the CDN. The benefit is that all icons are available. The disadvantage is that the files loaded are much bigger in size. You can use an icon font generator like IcoMoon or Fontello to only include the icons you actually need. This will reduce the size of the font files. One other option is to use SVG images directly, but that's trickier!

Importing the Font Awesome library

Just before the closing `</head>` tag in `index.html`, include the following script:

```
<script defer src="https://use.fontawesome.com/releases/v5.0.0/js/all.js"></script>
```

Right after the `<div class="contact">`, add these social icons:

```
<ul class="social">
  <li>
    <a>
      <i class="fab fa-twitter"></i>
    </a>
  </li>
  <li>
    <a>
      <i class="fab fa-github"></i>
    </a>
  </li>
  <li>
    <a>
      <i class="fab fa-linkedin"></i>
    </a>
  </li>
</ul>
```



> If the icons don't show up, make sure you are connected to the internet and that you have JavaScript enabled.

Styling the icons

If you reload the page repeatedly, you might notice that the layout "jumps" for a split second. That's because at first, the icons are not loaded, and the page only displays the text and the images. Then, when the icons finally load, the pop up, and cause the page to be redrawn, hence the jump.

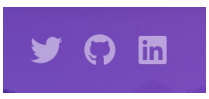
To avoid this jump, we're gonna set a square area of `2rem` by `2rem` for each `<li>` list item:

```
.social {
  display: flex;
  margin-top: 1.5rem;
}

.social li {
  height: 2rem;
  margin-right: 0.5rem;
  text-align: center;
  width: 2rem;
}
```

The `<ul>` list acts as a Flexbox container for each square list item. We can now style the inner links:

```
.social a {
  align-items: center;
  display: flex;
  font-size: 1.5rem;
  height: 2rem;
  justify-content: center;
  opacity: 0.5;
  transition-property: opacity, transform;
  width: 2rem;
  will-change: opacity, transform;
}
```



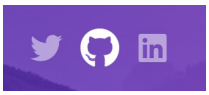
Each link item is a square as well, in which the Font Awesome icon is both vertically and horizontally **centered**.

Because the icons are text, we can use the `font-size` property to increase the icon's size. And since we've set all `<a>` tags to use the `currentColor`, it picks up the white color set for the page.

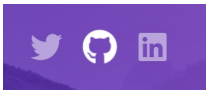
We're also setting the opacity to half transparent, which is convenient because it's valid for any color we would use.

As for the button, we're gonna add some hover and active styles:

```
.social a:hover {
  opacity: 1;
  transform: scale(1.25);
}
```



```
.social a:active {
  opacity: 1;
  transform: scale(1.1);
}
```



7 Creating CSS Animations

Animating in CSS is basically changing a set of *values* over *time*. For example, fading in an element is done by setting the `opacity` to zero `0`, and gradually incrementing it (`0.1`, `0.2`, `0.3` ...) until you finally reach the value of `1`.

It would take a while to figure all the *intermediate* values. Luckily in CSS, you only need to set **2 values**:

- the start values, with the keyword `from`
- the end values, with the keyword `to`

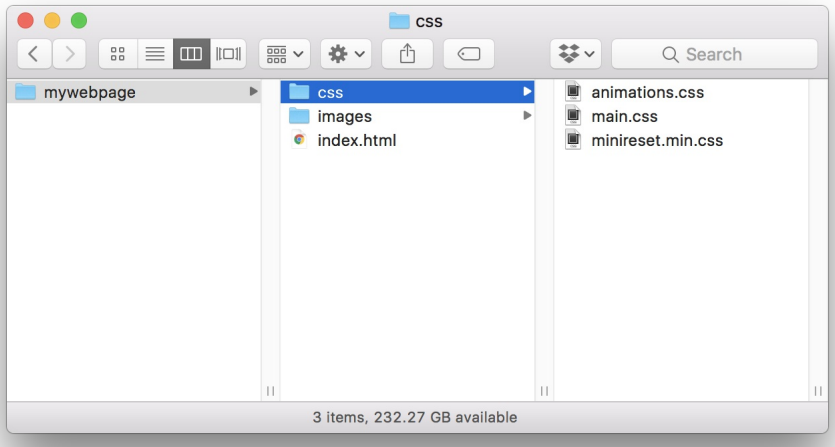
Everything in between will be determined by:

- the `animation-duration`: how long it takes to go from the start to the end
- the `animation-timing-function`: how the values in between are calculated (based on a curve)

CSS Animations are triggered when the page loads, or when a class name changes.

Adding a new CSS file

Writing CSS Animations takes quite a lot of space, so let's create a new file in the `/css` folder, and call it `animations.css`.



Add it to the list of CSS included. You should now have **4 stylesheets**:

```
<link href="https://fonts.googleapis.com/css?family=Montserrat|Quicksand" rel="stylesheet">
<link rel="stylesheet" type="text/css" href="css/minireset.min.css">
<link rel="stylesheet" type="text/css" href="css/main.css">
<link rel="stylesheet" type="text/css" href="css/animations.css">
```

Keyframes

A `@keyframes` statement contains a list of keyframes, which are all the intermediate steps of an animation.

We're gonna use the simple version where we only need to set two keyframes:

- `from` is the keyframe at the start of the animation
- `to` is the keyframe at the end of the animation

> You can have more than 2 keyframes, if you use **percentages** instead.

Here's the first keyframe we're gonna use. Add it to `animations.css`:

```
@keyframes bounceIn {
  from {
    opacity: 0;
    transform: scale(0.5);
  }
  to {
    opacity: 1;
    transform: scale(1);
  }
}
```

This `bounceIn` animation completes two changes simultaneously:

- fading in, from completely transparent to completely opaque
- scaling up, from half the size to the original size

We're gonna write 7 keyframe statements in total:

- `zoomOut` for the wallpaper
- `picImage` for the picture image
- `picShadow` for the picture shadow
- `slideDown` for the name
- `slideUp` for all other text elements
- `fillUp` for the hr line
- `bounceIn` for the contact button

Shared declarations

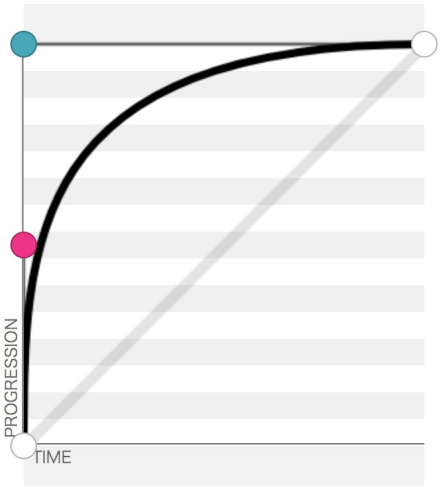
Like with our main CSS, we're gonna share a few declarations between multiple selectors.

```
.wallpaper,
.picture-shadow,
.picture-image,
.name,
.job,
.hr,
.description,
.contact,
.social li {
  animation-duration: 1s;
  animation-timing-function: cubic-bezier(0, 0.5, 0, 1);
  animation-fill-mode: both;
}
```

The duration is set to one second (`1s`).

The `animation-fill-mode` set to `both` tells the browser to use the `from` values *before* the animation, and to keep the `to` values *after* the animation has ended. Otherwise, the element would revert to their non-animated styles.

To make the animation more interesting and snappier, we're using a custom **cubic bezier** curve. See it in action!



Animating the contact button

For an animation to be triggered automatically, you only require the duration and the name. Let's add the name:

```
.contact {
  animation-name: bounceIn;
}
```

Click to see the Button Bouncein in action

Other keyframes

Here are the other 6 keyframe statements we're gonna need:

```
@keyframes fillUp {
  from {
    transform: scaleX(0);
  }
  to {
    transform: scaleX(1);
  }
}
```

```
@keyframes picImage {
  from {
    opacity: 0;
    transform: scale(1.2) translateY(-1rem);
  }
  to {
    opacity: 1;
    transform: scale(1) translateY(0);
  }
}
```

```
@keyframes picShadow {
  from {
    opacity: 0;
    transform: scale(1.2) translateY(4rem);
  }
  to {
    opacity: 1;
    transform: scale(1.1) translateY(0);
  }
}
```

```
@keyframes slideDown {
  from {
    opacity: 0;
    transform: translateY(-1rem);
  }
  to {
    opacity: 1;
    transform: translateY(0);
  }
}
```

```
@keyframes slideUp {
  from {
    opacity: 0;
    transform: translateY(1rem);
  }
  to {
    opacity: 1;
    transform: translateY(0);
  }
}
```

```
@keyframes zoomOut {
  from {
    opacity: 0;
    transform: scale(1.05);
  }
  to {
    opacity: 0.2;
    transform: scale(1);
  }
}
```

Other animations

Both the picture's shadow an image need a slightly *faster* animation, and a different animation curve:

```
.picture-shadow,
.picture-image {
  animation-duration: 750ms;
  animation-timing-function: cubic-bezier(0, 0.5, 0.25, 1.25);
}
```

We can now trigger all animations by using `animation-name` for all of them, while making some other adjustments too:

```
.wallpaper {
  animation-name: zoomOut;
  animation-timing-function: ease-out;
}

.picture-shadow {
  animation-name: picShadow;
}

.picture-image {
  animation-name: picImage;
}

.name {
  animation-name: slideDown;
}

.job {
  animation-name: slideUp;
}

.hr {
  animation-name: fillUp;
}

.description {
  animation-name: slideUp;
}
```

```
.social li {
  animation-duration: 500ms;
  animation-name: slideUp;
  animation-timing-function: cubic-bezier(0.5, 0, 0.5, 1.5);
}
```

The `.wallpaper` doesn't use a cubic bezier curve, but rather the `ease-out` keyword.

The social items have an even faster animation (500 milliseconds, or half a second) and different curve too.

Reload your page and see all animations triggered at once!

Click to see the Simultaneous Animations in action

Delaying each animation

Because all animations are happening *simultaneously*, it's difficult to see the effect of each of them. What we simply need to do, is to trigger them **in sequence**.

By using a trigger **delay**, we can achieve a lot!

```
.name {
  animation-delay: 100ms;
}

.job {
  animation-delay: 200ms;
}

.hr {
  animation-delay: 300ms;
}

.description {
  animation-delay: 400ms;
}

.picture-image {
  animation-delay: 500ms;
}

.picture-shadow {
  animation-delay: 500ms;
}

.contact {
  animation-delay: 600ms;
}
```

For the social icons, we want to delay them individually. We can use the `:nth-child` pseudo selector to select them:

```css
.social li:nth-child(1) {
  animation-delay: 800ms;
}

.social li:nth-child(2) {
  animation-delay: 900ms;
}

.social li:nth-child(3) {
  animation-delay: 1s;
}

.social li:nth-child(4) {
  animation-delay: 1.1s;
}

.social li:nth-child(5) {
  animation-delay: 1.2s;
}
```

> If you have more than 5 icons, just increment by `0.1s` each time.

Reload your page and see your whole page animated!

Click to see the Sequenced Animations in action

8 Triggering animations with JavaScript

Since you've been developing locally, both images (`alex.jpg` and `austria.jpg`) load **instantly**. But on a production server, these images will take a few seconds to load. Since the animation starts instantly, and only takes 1 second to complete, it will have ended *before* the image has loaded.

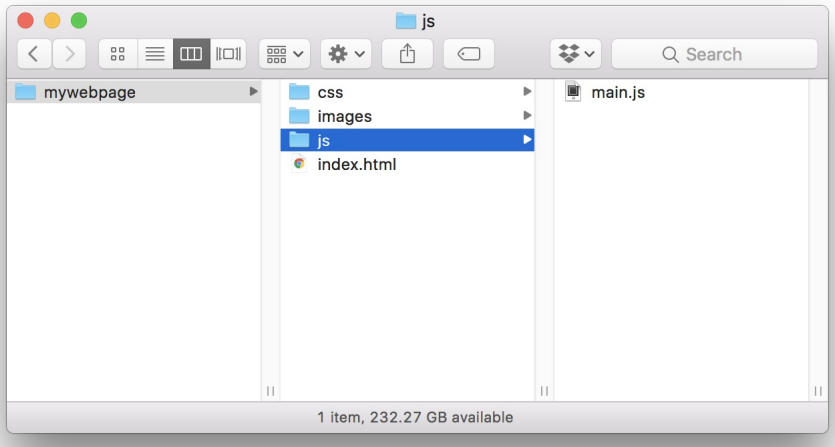
To see it in action on your local machine, simulate a slow connection using the Google Chrome Dev Tools:

- open the `"Developer Tools"` (with `Ctrl+Shift+I` or `Cmd+Opt+I` on Mac)
- go to the `"Network"` tab
- enable the `"Disable cache"` checkbox
- in the `"Online"` dropdown choose `"Slow 3G"`

Reload your page and see how the images take a while to load.

Adding a JavaScript file

Create a new folder called `/js` , and inside create a file called `main.js` .



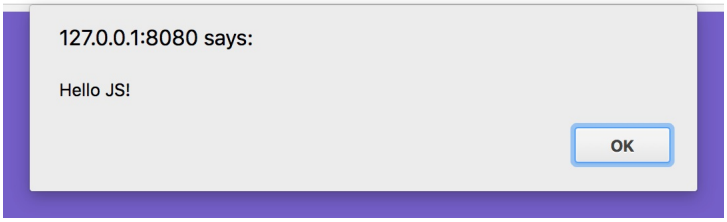
In `main.js` , type the following:

```
alert('Hello JS!');
```

In the `<head>` of your HTML, before the Font Awesome script, include your own script. You now have **2 scripts**:

```
<script defer type="text/javascript" src="js/main.js"></script>
<script defer src="https://use.fontawesome.com/releases/v5.0.0/js/all.js"></script>
```

Reload your page. You should see an alert box appear!



You can now **remove** the `alert()` function.

Editing the HTML

We need to prepare our HTML to interact with our JS, by using:

- **id** attributes, to select elements within our JS code
- **data** attributes, to pass data *from* the HTML to the JS

Locate the `wallpaper` element and add the `id` and `data-image` attributes:

```
<div id="wallpaper" class="wallpaper" data-image="images/austria.jpg"></div>
```

Add an `id` to the `picture` element:

```
<figure id="picture" class="picture">
```

Add another `id` to the `picture-image` element:

```
<img id="pictureImage"
     class="picture-image"
     src="images/alex.jpg"
     srcset="images/alex.jpg 1x,
             images/alex@2x.jpg 2x,
             images/alex@3x.jpg 3x"
     alt="Portrait of Alex Jefferson"
     width="320"
     height="320">
```

Appending a CSS class on load

We're gonna create a JS function that will:

- get an HTML element
- get the image source attached to that element
- create a new image from that source
- load that image
- when the image is loaded, add a new CSS class to the element

```
function loadImage(id, targetId) {
  var el = document.getElementById(id);
  var targetEl = targetId ? document.getElementById(targetId) : el;
  var imageToLoad;

  if (el.dataset.image) {
    imageToLoad = el.dataset.image;
  } else if (typeof el.currentSrc === 'undefined') {
    imageToLoad = el.src;
  } else {
    imageToLoad = el.currentSrc;
  }

  if (imageToLoad) {
    var img = new Image();
    img.src = imageToLoad;
    img.onload = function() {
      targetEl.classList.add('is-loaded');
    };
  }
}
```

Let's go through this function step by step. First, we define a function that can take two parameters called `id` and `targetId`:

```
function loadImage(id, targetId) {}
```

We then fetch the HTML element by using the value of `id`:

```
var el = document.getElementById(id);
```

We similarly fetch a target element to add a class to, but only if a `targetId` is provided. Otherwise, we revert to the element above:

```
var targetEl = targetId ? document.getElementById(targetId) : el;
```

We instantiate the `imageToLoad` variable:

```
var imageToLoad;
```

There are 3 (mutually exclusive) cases to set the value of `imageToLoad`:

- if `data-image` is provided, we use that value
- if the browser doesn't support `srcset` and `currentSrc`, we use the `src` value of the `<img>`
- otherwise we can simply use `currentSrc`

```
if (el.dataset.image) {
  imageToLoad = el.dataset.image;
} else if (typeof el.currentSrc === 'undefined') {
  imageToLoad = el.src;
} else {
  imageToLoad = el.currentSrc;
}
```

We check if `imageToLoad` holds a value:

```
if (imageToLoad)
```

We then create a new image from which the `src` is the value of the `imageToLoad`:

```
var img = new Image();
img.src = imageToLoad;
```

Finally, when the image is completely loaded by the browser, we add the CSS class `is-loaded` to the target element:

```
img.onload = function() {
  targetEl.classList.add('is-loaded');
};
```

We now need to add those `is-loaded` styles.

Setting the loaded styles

The animations for the `wallpaper` and the `picture-*` elements are still triggered automatically, so let's **remove** them first.

```
.wallpaper {
  animation-name: zoomOut;
  animation-timing-function: ease-out;
}

.picture-shadow {
  animation-name: picShadow;
}

.picture-image {
  animation-name: picImage;
}
```

And **replace** them with the following:

```
.wallpaper {
  animation-timing-function: ease-out;
}

.wallpaper.is-loaded {
  animation-delay: 1s;
  animation-name: zoomOut;
}

.picture.is-loaded .picture-shadow {
  animation-name: picShadow;
}

.picture.is-loaded .picture-image {
  animation-name: picImage;
}
```

We also need to set the **initial state** of those elements, by hiding them at the start. Add the following:

```
.wallpaper,
.picture-shadow,
.picture-image {
  opacity: 0;
}
```

If you now reload the page, the wallpaper and picture are **hidden**:

- the opacity is set to zero `0`
- the animation that fades them in is *not* triggered

Let's now trigger the animations with JS.

Using the function

We have set up the CSS to trigger the animation when the `is-loaded`. And we have defined a `loadImage()` function, but we haven't called it yet. Let's try it out!

At the end of `main.js`, call the function twice:

```
document.addEventListener('DOMContentLoaded', function() {
  loadImage('wallpaper');
  loadImage('pictureImage', 'picture');
});
```

We're waiting for the whole document to be loaded before calling the `loadImage()` twice, with different parameters.

Your page should now work perfecly! Try to set the network connection to `"Slow 3G"` again, and notice how the animations wait for the images to load before being triggered.

Edge case: no script

If for some reason the visitor doesn't have JS enabled, you still want to show them the images. The experience will not be as great, but it's better than showing no images at all.

In your HTML, just before the `</body>` closing tag, add this `<noscript>` tag:

```
<noscript>
  <style type="text/css">
    .wallpaper {
      animation-name: zoomOut;
    }

    .picture {
      animation-name: dropIn;
    }
  </style>
</noscript>
```

If JS is disabled, we simply trigger the animations. You now have the **best of both worlds**:

- if JS is *enabled*, the browser waits for the images to be loaded before trigger the animations
- if JS is *disabled*, the browser still shows the images by triggering the animations

9 Next steps

Explore HTML and CSS

While we've covered 90% of what you will need to build webpages with HTML and CSS, there are other tags and features available. You can find them on the Mozilla Developer Network. If you want a simpler version, try the free resources I've built: HTML Reference and CSS Reference.

Learn Sass

Sass is a CSS **preprocessor** that allows you to write CSS more easily. It comes with lots of features, but most notably **variables** (for setting values like colors, sizes, durations...), **mixins** (like JS functions but for your CSS) or **extends** (share declarations by extending another selector). You can read my free MarkSheet chapter on Sass as a start.

Learn JavaScript

While we've covered a bit of JS, there is a *lot* more to learn. There are tons of free resources online, like Eloquent JavaScript, JavaScript Garden or Codecademy.

For more complex interfaces, you can use a **JS framework** like React, AngularJS, Ember.js or Vue.js.

Learn backend development

We've only covered *client-side* development (or "frontend"). But for more complex websites, you will need to learn *server-side* development (or "backend"). This requires learning a **programming language** (like JavaScript, Python, Ruby, PHP...). Again, Codecademy is a great free resource.

Try out Bulma

Based on the knowledge gathered through 10 years of experience, I've created an open source CSS framework called Bulma.

You can use it directly to build interfaces simply by using the CSS classes provided. It's modular, so you can include only what you need. But since you probably want to write your own CSS after having read this book, it's also a good starting point for any type of work. I always use it in my personal and professional projects in some way.

It's also a good resource for learning. By going through the source code, you might end up implementing new features, writing extensions, or simply fixing bugs! It's a code base lots of developers have taken inspiration from.

Thank you for reading!