

# **For the Love of God, Don't Try To Implement That Yourself**

**A frustrated guide to web development, because hope has been gone for a very long time.**

Morgan Saville

July 6, 2021



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	The Client-Server Model . . . . .	1
<b>2</b>	<b>HTML, CSS and JavaScript — Making a simple clicker game</b>	<b>3</b>
<b>3</b>	<b>Setting up a Node.js server</b>	<b>45</b>
<b>4</b>	<b>Painting By Numbers</b>	<b>55</b>



# 1 Introduction

It's a very badly kept secret that making websites from scratch is a low-key nightmare. This simple fact of life is what creates the market for intuitive out-of-the-box website builders such as Wix<sup>1</sup>, Blogger<sup>2</sup> and the proud sponsor of every podcast ever: Squarespace<sup>3</sup>. This list is by no means comprehensive, and all of the above provide an excellent service — ideal for people who have some information in their brain, and want to get it up there online for the public's viewing pleasure.

## 1.1 The Client-Server Model

Nevertheless, a website builder doesn't fit everybody's needs. If you want your website to be more than a glorified Word document, you might need to get a little bit more hands-on, and do some programming. To understand how to go about this, we need to understand what a website actually is.

Most websites comprise of two main parts: A client, which is the device which wants to see the website (e.g. your laptop when you want to go to <https://google.com>), and a server, which is some other computer located somewhere in the magical land of it doesn't really matter. When the client tries to navigate to a URL, such as google or whatever website it wants to visit, it sends a request to the server saying something along the lines of "Hello there, I am a client and I would like to view *[some page on your website]* k thanks bye". The client may also say something like "Oh and by the way here's some other information I want you to know" but we'll talk about exactly what these requests look like and what information might be bundled up with them. The server is listening out for these requests, and when it receives one it might run some code, and then it will send some data back to the client. Usually this data will be the webpage the client was asking for. Alternatively it could be some other kind of data such as an error message (e.g. the infamous "404 Page Not Found" error). Either way, this data gets sent back to the client where it gets displayed by the web browser as a web page.

Let's briefly review some terminology, as it can get quite confusing.

- The client: The computer which is viewing the webpage. The term "client" can also be used to refer to the web browser running on that computer, or in some cases, the person using the computer.

---

<sup>1</sup><https://www.wix.com/>

<sup>2</sup><https://www.blogger.com/about/>

<sup>3</sup><https://www.squarespace.com/>

## 1 Introduction

- The server: The computer which listens out for requests from the client, potentially runs some code, and sends back the webpage it requested. The term “server” can also be used to describe the piece of software running on this computer which is listening out for requests, running the code, and returning the data.

The important thing to note here is that for the most part, once the client has received the website and is viewing it, the server’s role is done. This raises the question, how is it that a website can change the way it looks even after it’s been loaded. For example, imagine a webpage consisting of nothing but a dot moving across the screen. How is the dot moving if the server isn’t sending any more data to the client? The answer is that the “data” sent from the server back to the client contains some code in a language called JavaScript. This JavaScript code isn’t run by the server, but it is rather sent to the client to be run there.

This is, I believe, the key dichotomy of websites. There is some code that is run by the server when it receives a request from the client. This is called “back-end” code. There is then different code that is meant to be run by the client’s web browser itself. This code is sent to the client by the server. This is called “front-end” code. The difference is incredibly important, and one key reason for this is that the front-end code can’t communicate directly with the back-end server.

Making complicated websites can be incredibly difficult. There are lots of crucially important but very boring details that a developer has to keep in mind such as making a user account system which stores passwords securely, making sure that electronic payments are secure, protecting the website from various common attacks and so on. Furthermore, aside from the essential stuff like that, there are also little user experience tweaks such as nice animations that can massively improve your website, but are a pain to implement. Luckily, for common problems such as these, libraries exist to help us out. These are little bundles of code which are made by somebody else, and solve some of these problems for us. This abstracts away the boring stuff and allows the modern developer to focus on the important aspects of their website.

In this book however, we don’t want to make life easier for ourselves. We don’t really care about the finished project. We care about learning, and to learn, we must dive into the weeds, and handle all of the messy complicated business ourselves. The projects we build together in this book will be made from scratch.

## 2 HTML, CSS and JavaScript — Making a simple clicker game

Now, we’ve covered the fact that the server will send some JavaScript code to be run on the front-end. What else will it send? For the most part we only need to worry about three types of data: HTML, CSS and JavaScript. These are the building blocks of websites as the client sees them, and together they provide all the information the client needs to render the webpage correctly. Their basic functions are described below:

- HTML (Hypertext markup language): Tells you what is in the webpage (e.g. “there is a dot”).
- CSS (Cascading stylesheet): Tells you what things look like (e.g. “the dot is red and has a black outline. The dot is 20 pixels by 20 pixels. The page has a light blue background”).
- JavaScript: Tells things what to do (e.g. “the dot is moving across the screen”).

If a webpage were a human body, the HTML would be the bones, the CSS would be the skin and the JavaScript would be the muscles. There is occasionally some crossover between these three (for example, putting something in the centre of the webpage can technically be done with HTML, even though it should be done with CSS) but the best practice is usually to keep these three in their respective lanes.

I’m now going to give you a brief introduction to all three of these languages, as we will be making very heavy use of them throughout this book. To follow along, all you’ll need is some basic knowledge of programming concepts (variables, functions etc.), but you don’t need to have ever used any of these specific languages before.

HTML (Hypertext markup language) is a tag-based language used to tell your browser what elements are on the webpage. Most elements require an “opening” tag and a “closing” tag. All of the HTML between the opening tag and the closing tag are then part of this element. For example, if you wanted to have multiple paragraphs of text, each paragraph would go inside its own *p* tag (short for paragraph). The opening *p* tag looks like this:

<p>

The closing *p* tag looks like this:

```
</p>
```

So the HTML for our paragraphs might look like this:

```
<p>
  This is the first paragraph. As you can see, it is sandwiched between
  the opening p tag and the closing p tag, so the browser knows that this
  is a paragraph.
</p>

<p>
  This is the second paragraph, inside its own p tag.
</p>
```

You could just slap that code into a file and open it with your web browser, and you would indeed see two paragraphs on an otherwise empty webpage. However, there are some more things that every HTML document should have. All of the HTML for the page should be inside an `<html>` tag. Inside the `<html>` tag, there should be two more tags: `<head>` and `<body>`. The `<head>` tag should contain information which won't be directly displayed as an element on the webpage itself, such as the icon and title which will show in your web browser tab, and links to CSS and JavaScript files. For now let's just set the title in the browser tab (set using the `<title>` tag). The `<body>` tag will contain the elements to be displayed on the page. Our HTML file now looks like this:

```
1  <html>
2    <head>
3      <title>Two paragraphs</title>
4    </head>
5    <body>
6      <p>
7        This is the first paragraph. As you can see, it is sandwiched
        between the opening p tag and the closing p tag, so the browser
        knows that this is a paragraph.
8      </p>
9      <p>
10       This is the second paragraph, inside its own p tag.
11     </p>
12   </body>
13 </html>
```



HTML as a language doesn't care about whitespace. This means that you can add a bunch of new lines or tabs or spaces into your HTML and the end result will look the same — the browser will just ignore it. This means that we can use tab indentation to make the structure of the HTML much clearer to the developer. As you can see in the above example, we tend to indent everything inside a tag. This makes it very obvious that for example, the `<p>` tag on line 6 is inside the `<body>` tag which is in turn inside the `<html>` tag.

Note that for shorter tags such as the `<title>` tag on line 3, we can write both the opening tag, content, and closing tag all on one line.

Additionally, it is good practice to add the line `<!DOCTYPE html>` to the top of our document. This is called a doctype declaration, and it is not in fact an HTML tag itself (although it looks like one). It is just a bit of information which tells the browser what type of file it's looking at.

Our HTML now looks like this:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Two paragraphs</title>
5      </head>
6      <body>
7          <p>
8              This is the first paragraph. As you can see, it is sandwiched
              between the opening p tag and the closing p tag, so the browser
              knows that this is a paragraph.
9          </p>
10         <p>
11             This is the second paragraph, inside its own p tag.
12         </p>
13     </body>
14 </html>
```

If you put this code into a file (for example called *index.html*) and open it in your web browser, you should see this:

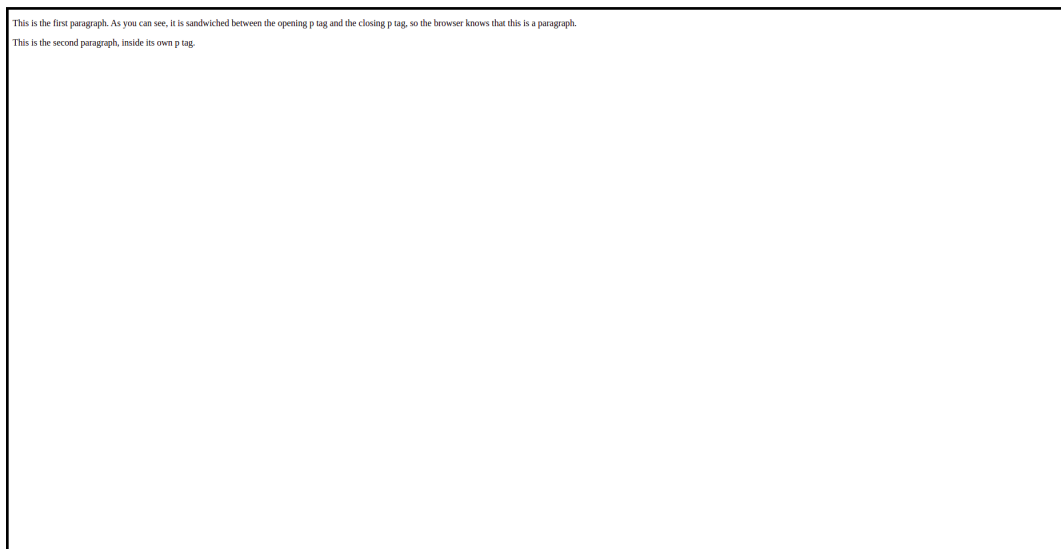


Figure 2.1: The HTML has been rendered by the browser

Now that we’ve got a basic understanding of the syntax of HTML, let’s try our hand at a slightly more interesting example. Let’s make a simple clicker game. There will be a picture of a cat on the screen, and a counter which tells you how many times you’ve clicked it. We’ll also have a reset button.

There’s one problem to address quickly before we start. Most modern web browsers will throw a bit of a fit when you’re loading HTML from a local file. For example, if you have some HTML which loads an image from a local source (such as our picture of a cat), then your browser very likely won’t let you for security reasons. Because of this, we need to load our webpage from a server somewhere. In the next section, we will indeed cover how to set up your own local development server for testing and how to deploy your websites to the internet. For now if you want to follow along with the examples, I recommend using a platform such as CodePen<sup>1</sup> which allows you to create HTML, CSS and JavaScript projects and view them from within the site itself.

Now we can move on to actually building the game. First we can create a file called *index.html* (it can actually be called anything as long as the extension is *.html*. Calling the entry point to the website *index* is a convention which will become clear in the next section). We can put this file in our project folder (which I’m going to call *clicker*). In this file, we will dump in our boilerplate HTML with nothing but a `<title>` tag and an empty `<body>`.

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Clicker game</title>
5      </head>
6      <body>
```

---

<sup>1</sup><https://codepen.io/>

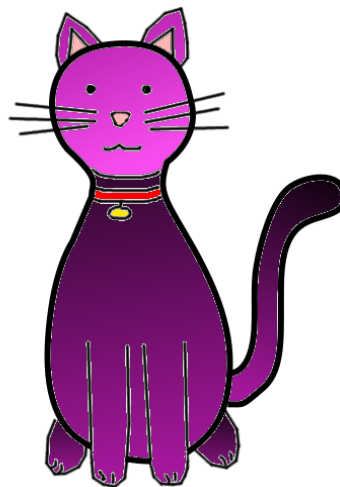
```
7     </body>
8 </html>
```

Now we can start adding some content to the `<body>`. We need three things: an image (the cat), a counter, and a button. Let's tackle these one at a time.

We can add in an image using an `<img>` tag, but how does it know which image to use? We can answer this question using HTML attributes. An attribute is a piece of information specified inside an opening tag itself. For example, in the `<img>` tag, we can specify where to find the image by writing for example `</img>`. This sets the "src" attribute of the `<img>` tag equal to the string `./path/to/image.png`, which we'll say is the path to the image file. There are many attributes available for each type of tag, and over the course of this book we will discuss a few of them, and what they're for.

Another important thing to note is that there's no content between the opening `<img>` tag and the closing `</img>` tag. We can therefore replace both of them with a single `<img />` tag, which acts as both an opening and closing tag. This is what it known as a "self-closing tag".

For the clicker game, I'm going to use this image:



This is Wafflecone the cat. My girlfriend drew him, and since then I've included my little buddy in every project I can. He is absolutely free to use, so if you want to use him, you can find the image at <https://wafflecone.co.uk/img/wafflecone.png>. I'm going to save him in a folder called `img` under the file name `wafflecone.png`. My project folder structure now looks like this:

```
clicker
├── index.html
├── img
│   └── wafflecone.png
```

Therefore, I can use the tag `` to embed the image into the webpage. Our HTML now looks like this:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Clicker game</title>
5      </head>
6      <body>
7          
8      </body>
9  </html>
```

If we open this up in a web browser, we'll see this:

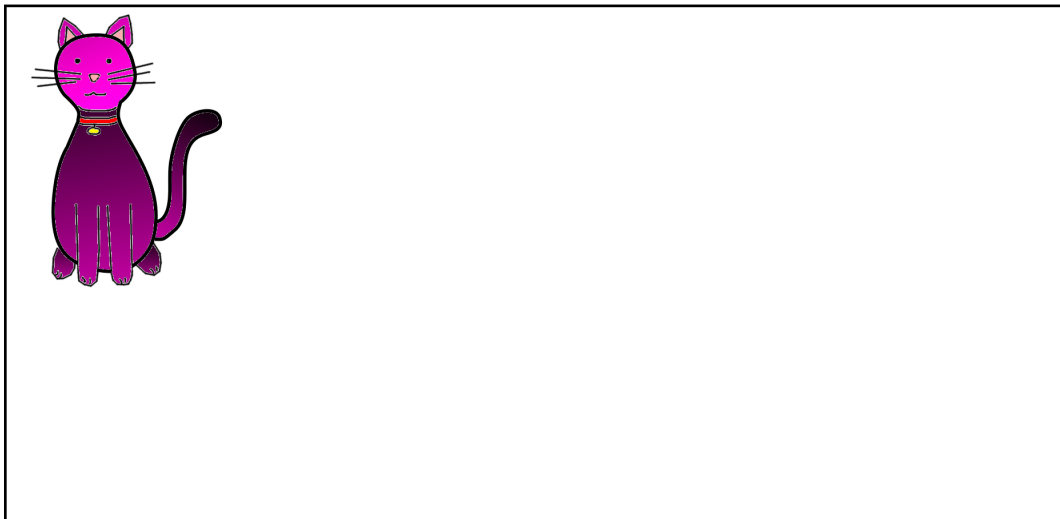


Figure 2.2: Our image is embedded in the page

If you're wondering why we put `./` at the start of the file path, it is because when writing file paths, `."` means the current directory (i.e. the folder in which *index.html* exists). Similarly, `.."` means the parent directory (one further up from the current directory).

You may also be wondering how the browser actually gets the image into the page. We talked earlier about how the client makes requests to the server which sends back data. Here, two requests are being made. The first is the client saying "Hi server, could I please have *index.html*?" and the server will send back the contents of *index.html*. The client will look through this data and realise that it has an image it needs to find. It will look at the *src* attribute to find out where to get the image from, and then it will send another request to the server, asking for the image. The server will send back the image, and the browser can then display it in the page. This is important to note. Whenever you link to an external resource from within your HTML, the browser makes an additional request in order to fetch that resource. This also means that the HTML will be received before all of the images are

necessarily received. This is why if you have a slow internet connection, the majority of a page might load first, and then the images will load afterwards as they get fetched by the browser.

Another thing to note is that the resources we use (such as the image) can even be from external websites. We could have just as easily used a tag like

```

```

and the page would look exactly the same. One thing to consider is that you are now trusting that the external website won't go down, and the image won't be deleted/altered/moved by the owner of that website.

Now that we've got the image in place, we can add the counter. We could do this in a `<p>` tag, but I'm instead going to introduce a new tag to use: `<div>`. This is short for "division" and is typically used as a container for other HTML elements. `<div>` elements can usually be thought of as a section of the page. We will insert a `<div>` tag into the page with the text "0" inside it (the initial value of the counter).

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Clicker game</title>
5      </head>
6      <body>
7          
8          <div>
9              0
10         </div>
11     </body>
12 </html>
```

Next we can add in the reset button. This can be done using the `<button>` tag. The content of the tag will displayed inside the button, so we can use the following tag:

```
<button>Reset</button>
```

Our HTML now looks like this:

```
1  <!DOCTYPE html>
```

```
2 <html>
3   <head>
4     <title>Clicker game</title>
5   </head>
6   <body>
7     
8     <div>
9       0
10    </div>
11    <button>Reset</button>
12  </body>
13 </html>
```

If we now open this up in the browser, we'll see this:



Figure 2.3: The cat image, the counter, and the reset button

We've now got the building blocks of the site. They are, however, sinfully ugly. Luckily, we can style all of the elements with CSS.

We can now create a directory in the project folder called *css* and inside it, create a file called *styles.css*. Our project folder structure now looks like this:

```
clicker
├── index.html
├── img
│   └── wafflecone.png
└── css
    └── styles.css
```

Before we can set any styles, we need to tell the HTML file where to find the CSS. We can do this with a `<link>` tag inside the `<head>` tag. The `<link>` tag looks like this:

```
<link rel="stylesheet" href="./css/styles.css">
```

As you can see, we are setting two attributes of the `<link>` tag: the “rel” attribute (short for “relationship”) is set to the value “stylesheet” which tells the browser that the resource we are linking to is CSS. The “href” attribute specifies the location of the linked document. The order in which we define these attributes does not matter.

Also note that the `<link>` tag does not require a closing tag. Our HTML now looks like this:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Clicker game</title>
5      <link rel="stylesheet" href="./css/styles.css">
6    </head>
7    <body>
8      
9      <div>
10         0
11      </div>
12      <button>Reset</button>
13    </body>
14  </html>
```

We can now start writing some CSS inside the file *styles.css*.

CSS works by applying specific styles to groups of elements. We specify which elements to which to apply which styles using “selectors”. To demonstrate this, let’s first try to style the counter. There are multiple ways we could write a CSS selector which includes the counter. The first would be to select all the `<div>` elements. The selector for that is simply “`div`”. After the selector, we enclose all of the styles in curly brackets. For example, if we wanted to set the font size to be bigger (let’s say 100 pixels), then we could use the property “font-size” and set it to a value of 100px.

```
div {
  font-size: 100px;
```

```
}
```

As shown above, to set a property, you put a colon after its name, and then you put the value you want. After each property, put a semicolon. If we reload the website in the browser, we can see the result.

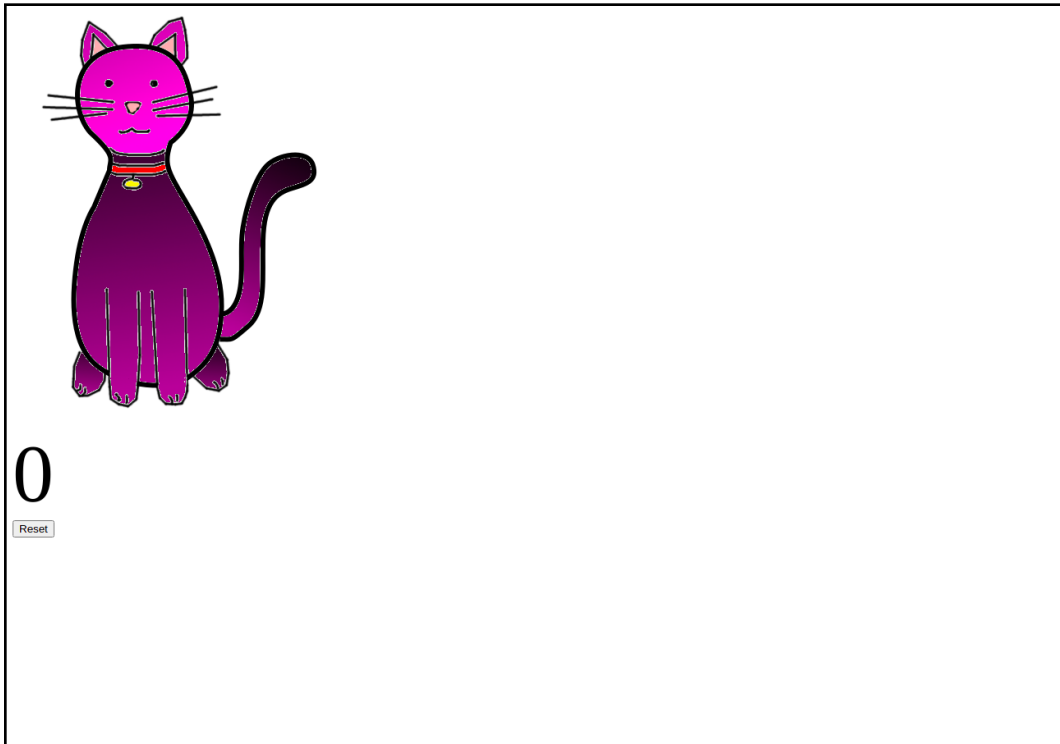


Figure 2.4: The font size of the counter is 100px

We will talk more about CSS properties later. For now, let's focus on the selector. Using the selector “`div`” is not ideal, because as the website becomes more complex, we might have many `<div>` elements, and our current CSS would apply the style to all of them. What we want is to only select this one specific `<div>`. We can achieve this by setting its “id” attribute.

```
9      <div id="counter">
10        0
11      </div>
```

We can now select only for the element with this ID. ID selectors in CSS work by writing a hash sign “#” followed by the ID, for example: “`#counter`”.

```
#counter {
  font-size: 100px;
```



```
}
```

Our site looks exactly the same, but now if we were to add more `<div>` elements to it, they would not be affected by the style.

Note that there should only ever be one element with a given ID, and one element can only have at most one ID. If, let's say, we wanted to have two counters, we would need the CSS selector to apply to both of them, but not to the rest of the `<div>` elements. We could do this by using classes. Classes are a bit like IDs, except an element can have multiple classes, and each class can be shared by multiple elements. For example, we could say

```
9      <div class="counter">
10          0
11      </div>
```

And we could have multiple elements with that class. The CSS selector for a class is a dot followed by the class name.

```
.counter {
    font-size: 100px;
}
```

If we wanted to give our `<div>` multiple classes, you can separate them with a space. For example, `<div class="class1 class2">`. This would mean that the `<div>` has two classes; “class1” and “class2”. We could use either `.class1` or `.class2` to select it. If we wanted to select elements which have both of these classes, we can combine the selectors into `.class1.class2` to achieve this. Furthermore, if we wanted to select only `<div>` elements with the class “class1” then we could use the selector `div.class1` to achieve this.

CSS selectors can get quite complicated<sup>2</sup>, but this is all we need for now, and we will talk through more complicated usage as and when we start to use it.

For now, let's go back to using IDs. We will give the image and ID of “cat”, we will give the counter an ID of “counter”, and we will give the button an ID of “reset”. Our HTML now looks like this:

```
1  <!DOCTYPE html>
2  <html>
```

---

<sup>2</sup>For more information, visit [https://www.w3schools.com/cssref/css\\_selectors.asp](https://www.w3schools.com/cssref/css_selectors.asp)

```

3     <head>
4         <title>Clicker game</title>
5         <link rel="stylesheet" href="./css/styles.css">
6     </head>
7     <body>
8         
9         <div id="counter">
10             0
11         </div>
12         <button id="reset">Reset</button>
13     </body>
14 </html>

```

In our CSS, we can continue styling the counter. Let's change the font to Arial. We can do this using the `font-family` property. We can provide multiple font family names, and the browser will try to render the first one. If this fails, it will try the second one and so on.

```

1  #counter {
2      font-size: 100px;
3      font-family: Arial, Helvetica, sans-serif;
4  }

```

Now, even though the text itself is on the left-hand side of the page, the `<div>` itself actually spans the entire width of the window. You can see this if you open up the developer tools in your web browser and hover over the element in the hierarchy. If we want the text to appear in the centre of the screen, we don't have to move the `<div>` itself, we just need to say that all the text inside the `<div>` should be aligned with the centre. This is done with the CSS property `text-align`.

```

1  #counter {
2      font-size: 100px;
3      font-family: Arial, Helvetica, sans-serif;
4      text-align: center;
5  }

```

Note the Americanisation of the word "center". This is required as part of the CSS syntax, and similar spellings are required for properties involving "color". For consistency's sake, I will therefore use Americanisations throughout all the code, even when it is not required by the syntax (though I will continue to use the British spelling when describing the concepts).

If we refresh the browser, we will see the counter in Arial in the middle of the page.

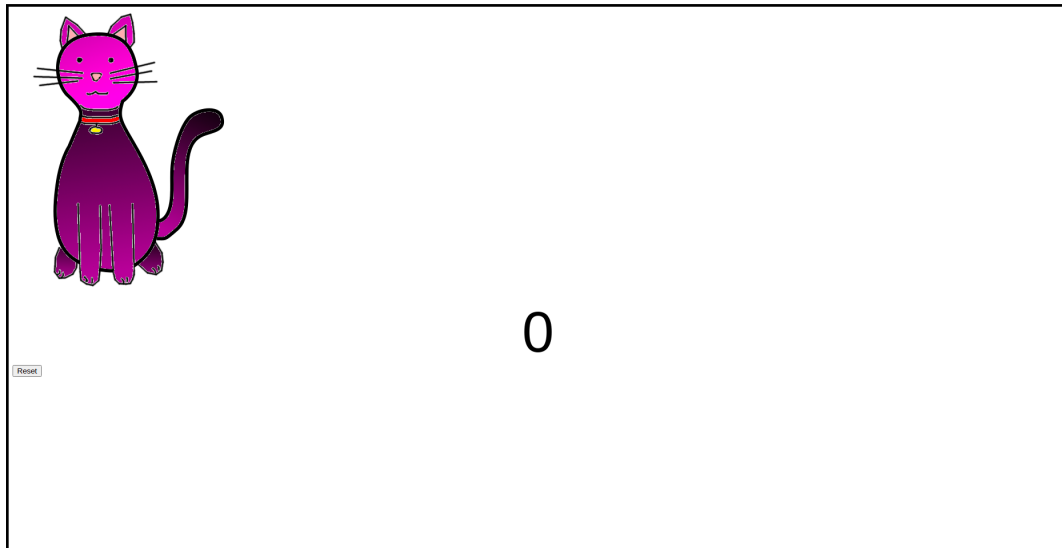


Figure 2.5: The counter is in Arial, and is in the middle of the page

Now let's style the reset button. We can select it by its ID using the `#reset` selector, and we can make the text bigger – let's say 50px.

```
7 #reset {  
8     font-size: 50px;  
9 }
```

We can make the border darker by using the `border` property. This typically takes three values: the border width, the border style, and the border colour. For example:

```
7 #reset {  
8     font-size: 50px;  
9     border: 3px solid black;  
10 }
```

This will result in a solid black border around the button which is three pixels wide, like this:

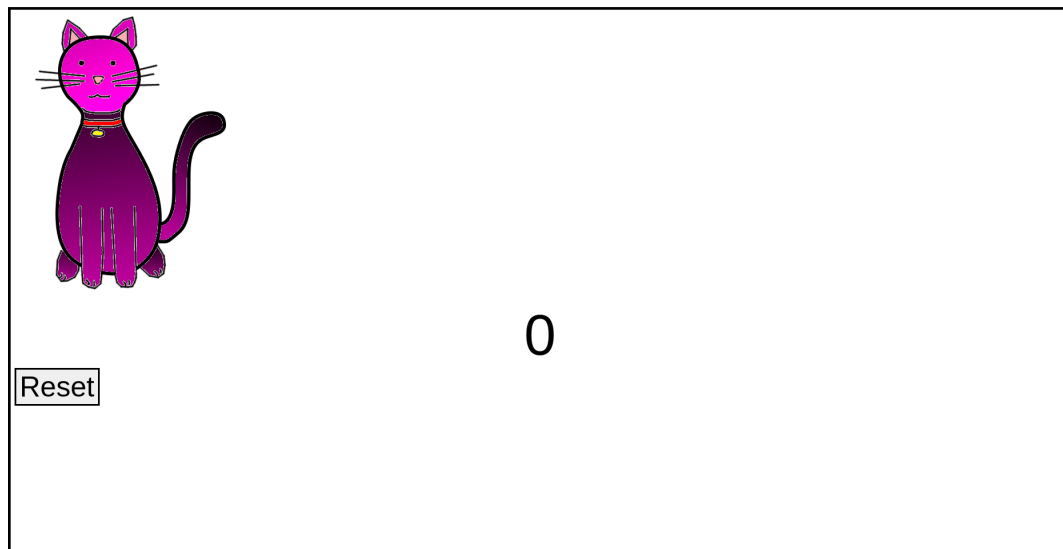


Figure 2.6: `border: 3px solid black;`

The same result could have been achieved using three separate properties like so:

```
7 #reset {  
8     font-size: 50px;  
9     border-width: 3px;  
10    border-style: solid;  
11    border-color: black;  
12 }
```

We can now set the colour of the button using the `background-color` property. In CSS, there are several ways to specify a colour. One is by name, like we did with the border (we just used the word “black”). CSS has many built-in colours which you can refer to by name. At the time of writing, these include:

	lightsalmon		salmon		darksalmon
	lightcoral		indianred		crimson
	firebrick		red		darkred
	coral		tomato		orangered
	gold		orange		darkorange
	lightyellow		lemonchiffon		lightgoldenrodyellow
	papayawhip		moccasin		peachpuff
	palegoldenrod		khaki		darkkhaki
	yellow		lawngreen		chartreuse
	limegreen		lime		forestgreen
	green		darkgreen		greenyellow
	yellowgreen		springgreen		mediumspringgreen
	lightgreen		palegreen		darkseagreen
	mediumseagreen		seagreen		olive
	darkolivegreen		olivedrab		lightcyan
	cyan		aqua		aquamarine
	mediumaquamarine		paleturquoise		turquoise
	mediumturquoise		darkturquoise		lightseagreen
	cadetblue		darkcyan		teal
	powderblue		lightblue		lightskyblue
	skyblue		deepskyblue		lightsteelblue
	dodgerblue		cornflowerblue		steelblue
	royalblue		blue		mediumblue
	darkblue		navy		midnightblue
	mediumslateblue		slateblue		darkslateblue
	lavender		thistle		plum
	violet		orchid		fuchsia
	magenta		mediumorchid		mediumpurple
	blueviolet		darkviolet		darkorchid
	darkmagenta		purple		indigo
	pink		lightpink		hotpink
	deeppink		palevioletred		mediumvioletred
	white		snow		honeydew
	mintcream		azure		aliceblue
	ghostwhite		whitesmoke		seashell
	beige		oldlace		floralwhite
	ivory		antiquewhite		linen
	lavenderblush		mistyrose		gainsboro
	lightgray		silver		darkgray
	gray		dimgray		lightslategray
	slategray		darkslategray		black
	cornsilk		blanchedalmond		bisque
	navajowhite		wheat		burlywood
	tan		rosybrown		sandybrown
	goldenrod		peru		chocolate
	saddlebrown		sienna		brown
	maroon				


However, if we want to use custom colours, we can specify them with RGB values.

One way to describe colours digitally is by splitting them into red, green and blue components. A colour has some amount of all three of these components, which each range from 0 (none) to 255 (maximum). The colour black can be represented by the triplet of values (0, 0, 0) which means 0 red, 0 green, and 0 blue. The colour white can be represented by (255, 255, 255) which means full red, full green, and full blue. Yellow can be written as (255, 255, 0) which means full red, full green, and no blue.

Instead of writing `black` in the CSS, we equally could have written `rgb(0, 0, 0)`.

Another equivalent way of specifying colours is with a hex code. Hex (short for hexadecimal) is the name for base-16 positional notation, and it a way of writing numbers. Instead of going from 0 to 9 and wrapping back around to 10 as we normally do when using base-10 numbers in every day life, in hex we count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f and then wrap around to 10 – a number equivalent to sixteen rather than ten. Here, instead of having a tens' place and a units' place, we have a sixteens' place and a units' place. This is a convenient way to write numbers for computers, because it is easy to convert between hex and binary.

Therefore, as well as each of the red, green, and blue values being represented by a base-10 number between 0 and 255, we can also represent them each as a two-digit hex number ranging from 00 to ff. This is because ff is the hexadecimal way of writing the number two hundred and fifty-five. We can therefore write a colour as a hash symbol “#” (to signify that this is a hex colour) followed by 6 hexadecimal digits – the first two are the red component, the third and fourth are the green component, and the last two are the blue component. Instead of writing `black` in the CSS, we could have instead written `#000000`. Instead of `white` we could write `#ffffff`, and instead of yellow we could write `#ffff00`, etc. You can perform a Google search for the term “colour picker” and you will be given a nice widget wherein you can select a colour, and you will be told the hex code for it, and the RGB values.

At last, we can pick a colour for the button. I will go with `#f09ae9` . Our CSS now looks like this:

```

1  #counter {
2      font-size: 100px;
3      font-family: Arial, Helvetica, sans-serif;
4      text-align: center;
5  }
6
7  #reset {
8      font-size: 50px;
9      border: 3px solid black;
10     background-color: #f09ae9;
11 }
```

If we look at the page, we can see that it worked.

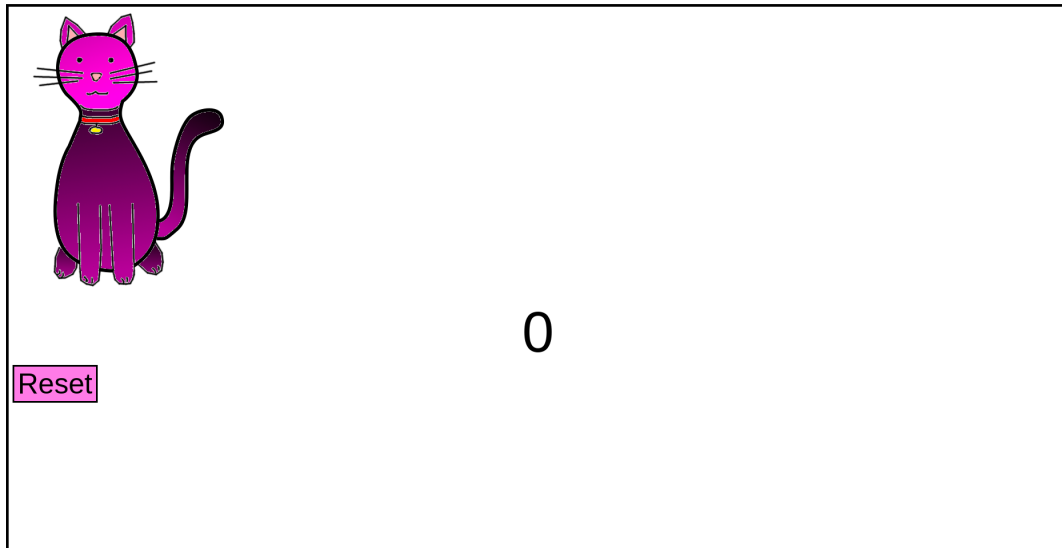




Figure 2.7: The button has a pink background

However, you can hardly even tell this is a button. A good way to make buttons more responsive is to make them change colour slightly when you hover your mouse over them, and when you click. We can do this using CSS pseudo-classes. Pseudo-classes allow us to create CSS selectors which only apply to elements in a certain state. For example, we can add `:hover` to the end of a selector to make the style only apply to those elements when the mouse is hovering over them. Similarly, we can use the `:active` pseudo-class to apply styles to an element when the mouse is currently clicking on it. We will make each of these a slightly darker than the default pink – `#9e6299`  for hover and `#694165`  for active.

```
7 #reset {
8     font-size: 50px;
9     border: 3px solid black;
10    background-color: #f09ae9;
11 }
12
13 #reset:hover {
14     background-color: #9e6299;
15 }
16
17 #reset:active {
18     background-color: #694165;
19 }
```

Now when you hover over the reset button, it turns darker, and darker still when you click on it. When you stop clicking on it and move the cursor away, it will go back to its default pink colour. Another trick to make buttons feel more responsive is to use the `cursor` property. This allows you to specify what the mouse cursor should look like when it's hovering over the element. We can set the value to be `pointer` for the button, so that when you hover over the

button, your mouse cursor turns into a pointer, and the user can clearly tell that the button is clickable.

```
7  #reset {
8      font-size: 50px;
9      border: 3px solid black;
10     background-color: #f09ae9;
11     cursor: pointer;
12 }
13
14 #reset:hover {
15     background-color: #9e6299;
16 }
17
18 #reset:active {
19     background-color: #694165;
20 }
```

If we want to place the button directly underneath the counter, we can't simply use the same `text-align: center` technique that we used for the counter itself. This is because the counter `<div>` stretched across the whole screen, and giving it `text-align: center` meant that it should put its contents in the centre of itself. The button on the other hand does not stretch across the whole screen. The button's width is defined by its content, so its content is in the centre of the button by default. Instead, one approach would be to set `text-align: center` for the `<body>` element, so that the button would be placed in the centre of it. This is quite a messy solution though, as that would affect every single element inside the `<body>`, not just the button<sup>3</sup>. Instead, we need to talk about margins and padding.

The margin of an element is the space around the outside of an element. The padding of an element on the other hand is the space on the inside of the element, between itself and its content. These are very often conflated by people who are new to CSS, so make sure you fully understand the difference. If we wanted to make the button further away from the left edge of the screen, we would increase the button's margin (or the `<body>`'s padding). If we wanted to add some space between the edges of the button and the text "Reset" inside it, then we would increase the button's padding. To demonstrate this, let's add some padding to the button. Let's give it 30 pixels of padding on all four sides.

```
7  #reset {
8      font-size: 50px;
9      border: 3px solid black;
10     background-color: #f09ae9;
11     cursor: pointer;
12     padding: 30px;
13 }
```

---

<sup>3</sup>Even though we are eventually going to centre the cat as well, this still isn't great practice for CSS in general.



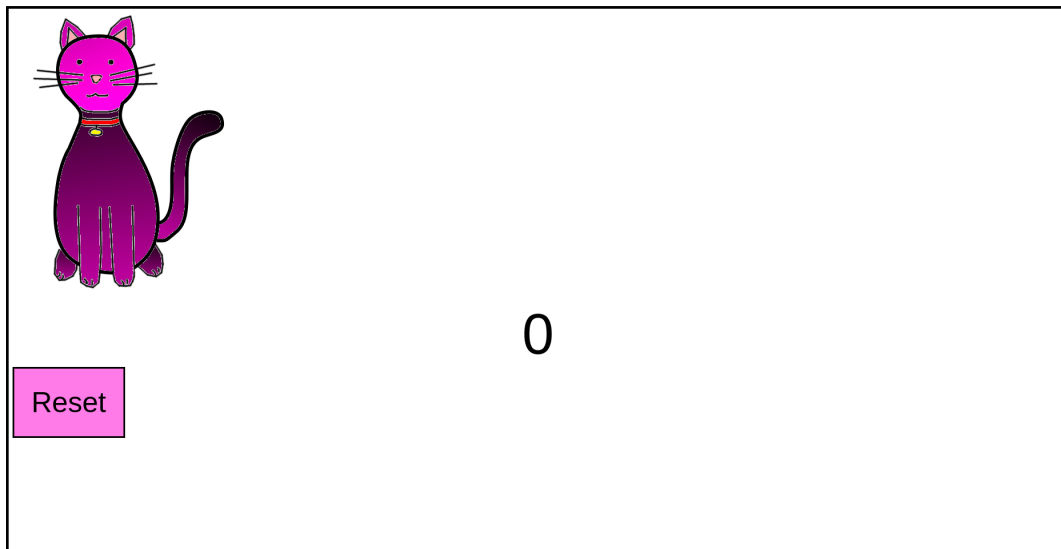


Figure 2.8: 30 pixels of padding in the button

The same action could be achieved by setting the top, right, bottom and left padding separately like so:

```
7  #reset {
8      font-size: 50px;
9      border: 3px solid black;
10     background-color: #f09ae9;
11     cursor: pointer;
12     padding-top: 30px;
13     padding-right: 30px;
14     padding-bottom: 30px;
15     padding-left: 30px;
16 }
```

Alternatively, if you provide four values for the `padding` property, then they will set the top, right, bottom, and left padding respectively. To demonstrate, let's give 20px of padding for the top and the bottom, and 40px of padding for the right and the left.

```
7  #reset {
8      font-size: 50px;
9      border: 3px solid black;
10     background-color: #f09ae9;
11     cursor: pointer;
12     padding: 20px 40px 20px 40px;
```

13 }

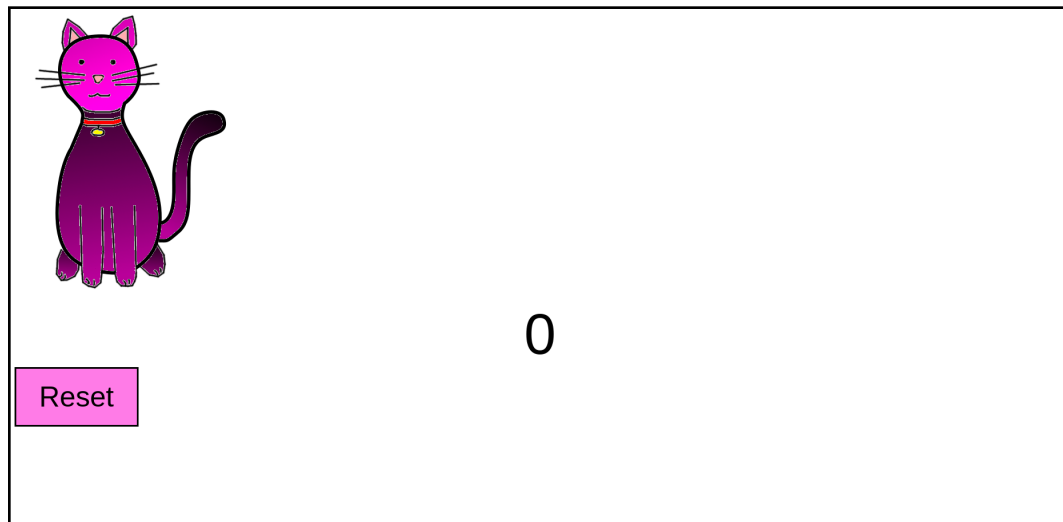


Figure 2.9: 20 pixels of padding on the top and bottom of the button; 40px of padding on the right and left

For brevity's sake, we will use the shorthand of only providing two values. The first one will be used as the padding for the top and the bottom, and the second value will be for the right and the left. Our CSS now looks like this:

```
7 #reset {  
8     font-size: 50px;  
9     border: 3px solid black;  
10    background-color: #f09ae9;  
11    cursor: pointer;  
12    padding: 20px 40px;  
13 }
```

Now let's move on to the margin. We could give the button 100px of margin in all four directions by setting the property `margin: 100px`.

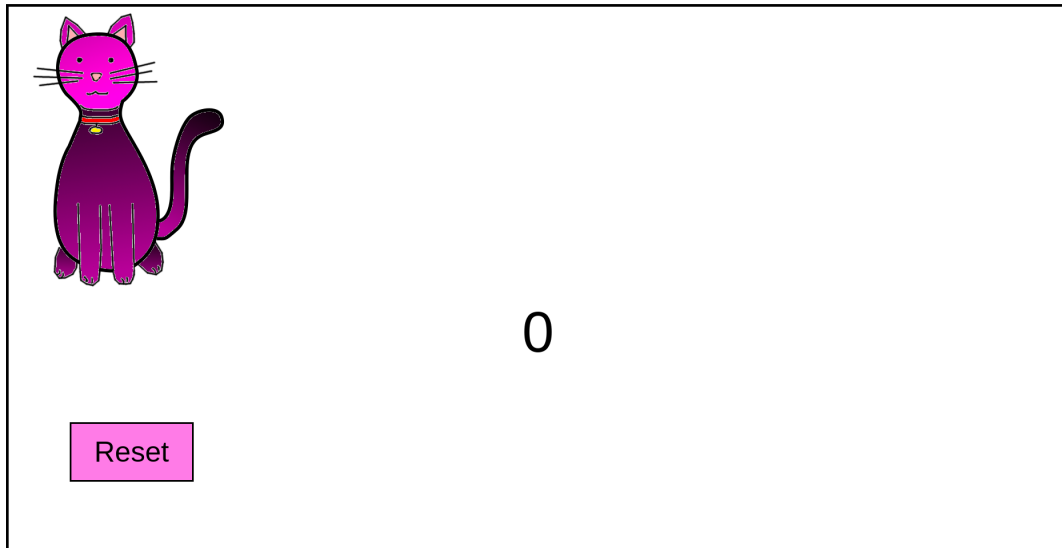


Figure 2.10: 100px of margin around the button

Similarly to padding, we could set the margin on all four of the edges separately using the `margin-top`, `margin-right`, `margin-bottom`, and `margin-left` properties, or by passing four values to the `margin` property. Again, passing only two values will result in the first being used for vertical margins and the second being used for horizontal margins.

Remember, what we want to achieve is putting the button in the centre. With some elements, you can achieve this by setting the horizontal margin to a value of `auto`. For example, if we wanted to keep a vertical margin of 100px but centre the button horizontally, we could set `margin: 100px auto` like so:

```
7  #reset {
8      font-size: 50px;
9      border: 3px solid black;
10     background-color: #f09ae9;
11     cursor: pointer;
12     padding: 20px 40px;
13     margin: 100px auto;
14 }
```

However, this does not work for our button because of the CSS `display` property. The `display` property controls how elements flow onto the page. By default, buttons have a `display` value of `inline-block`. The “inline” part of this means that you could put this button in the middle of some text and it won’t disturb the flow. This might be useful if we wanted our page to look like this:

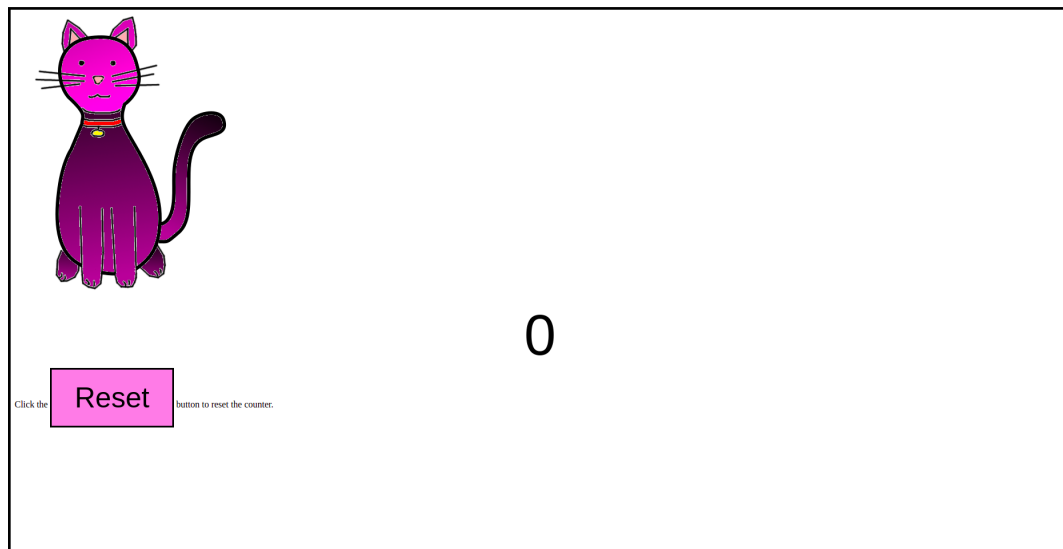


Figure 2.11: The button flows inline with the text

The “block” part means that the button functions like a container. You could set its width and height separately from its contents. However, the “inline” nature of the button means that giving it a margin of `auto` no longer centres it. Luckily, we can change its `display` property to `block`.

```
7  #reset {  
8      font-size: 50px;  
9      border: 3px solid black;  
10     background-color: #f09ae9;  
11     cursor: pointer;  
12     padding: 20px 40px;  
13     margin: 100px auto;  
14     display: block;  
15 }
```

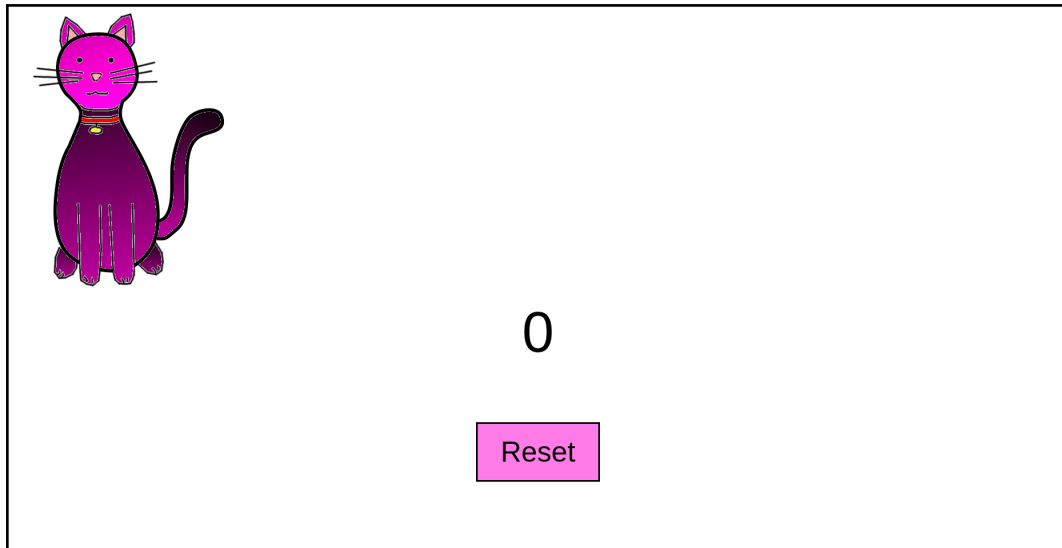


Figure 2.12: The button is centred

One final piece of styling we'll do to the button is give it rounded corners. We can do this with the `border-radius` property. Somewhat arbitrarily, we will give it a value of 25px.

```
7  #reset {  
8      font-size: 50px;  
9      border: 3px solid black;  
10     background-color: #f09ae9;  
11     cursor: pointer;  
12     padding: 20px 40px;  
13     margin: 100px auto;  
14     display: block;  
15     border-radius: 25px;  
16 }
```

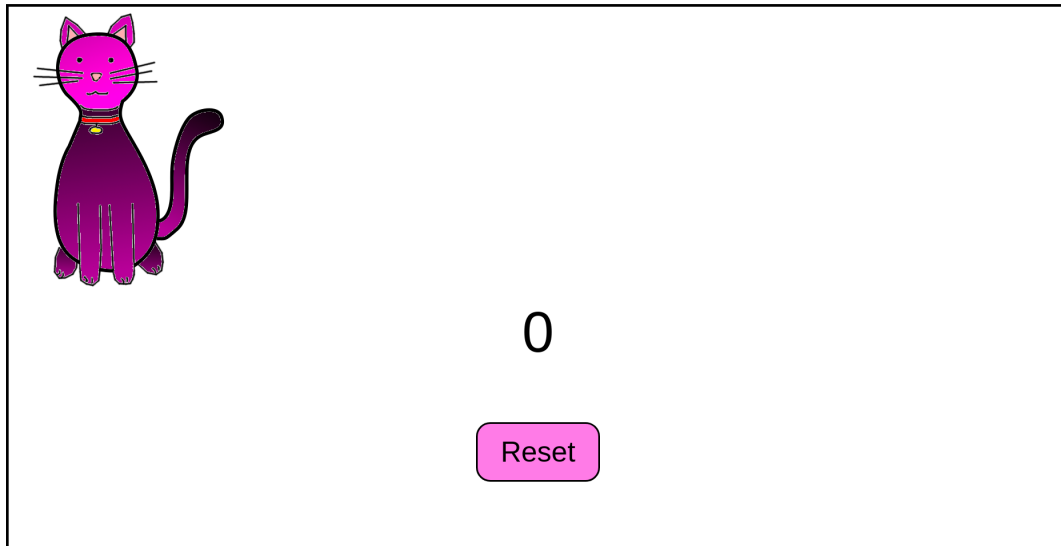


Figure 2.13: The button is has rounded corners

You might notice that if you click the reset button, an ugly blue or orange outline appears around it. This is an artefact of Chrome or Firefox trying to show you that a button is focused. They also do this with text boxes, and frankly it’s disgusting. We can remove it with CSS by stating that any element which is focused should have no outline. The selector for “any element” is `*` and we can use the `:focus` pseudo-class to select any focused element.

```
1  *:focus {  
2      outline: none;  
3  }
```

Now let’s move on to styling the cat. We can of course select the image using its ID with the `#cat` selector. By default, `<img>` tags have a `display` property of `inline`, so we can change it to `block` and give it a vertical margin of `50px` and a horizontal margin of `auto` to centre it.

```
30  #cat {  
31      display: block;  
32      margin: 50px auto;  
33  }
```

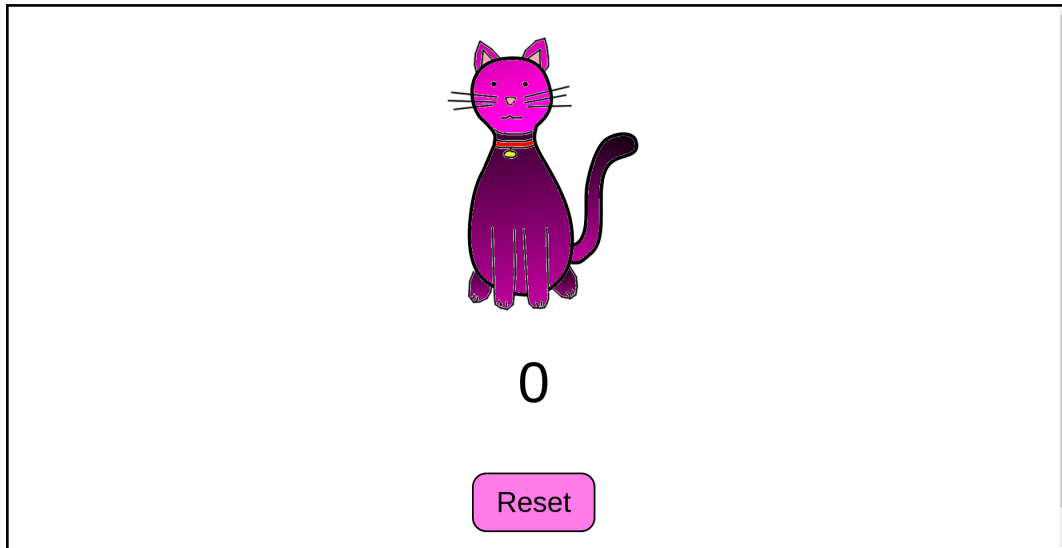


Figure 2.14: The cat is centred

Now you might find that the content of the page is too tall, and a vertical scroll-bar appears in your web browser. Now we could take more care to make sure that everything is going to fit in the page, but since this is just a toy project, let's fix it quickly by reducing the margin on the top edge of the reset button from 100px to 50px (which moves it upwards slightly), and removing the margin on the bottom edge since we don't really need it.

```
11 #reset {
12     font-size: 50px;
13     border: 3px solid black;
14     background-color: #f09ae9;
15     cursor: pointer;
16     padding: 20px 40px;
17     margin: 50px auto 0 auto;
18     display: block;
19     border-radius: 25px;
20 }
```

That takes care of the scroll-bar<sup>4</sup>. Now as a user, I would not know that the cat is even clickable at this point. Let's make it more responsive by giving it a pointer cursor.

```
30 #cat {
31     display: block;
32     margin: 50px auto;
33     cursor: pointer;
34 }
```

---

<sup>4</sup>Of course, this will vary depending on the size of your screen

Let's also make it get slightly bigger when the mouse hovers over it. We can do this by giving it a default width and a height, and using the `:hover` pseudo-class to give it a slightly bigger width and height when the mouse is over it. In fact, we only need to specify either a width *or* a height, not both. The other one will be calculated by the browser such that the image maintains its proper aspect ratio. Let's give the cat a height of 450px by default, and 500px when the mouse is over it.

```
30 #cat {
31     display: block;
32     margin: 50px auto;
33     cursor: pointer;
34     height: 450px;
35 }
36
37 #cat:hover {
38     height: 500px;
39 }
```

There are two problems with this. The first is that when we hover over the cat, it does indeed get 50 pixels taller, but that forces the counter and the reset button 50 pixels downwards. Luckily we can fix this by changing the cat's vertical margin when the mouse is over it to be 25px each on the top and the bottom, rather than 50px. This will offset the cat's growth, leaving the counter and the reset button exactly where they are.

```
37 #cat:hover {
38     height: 500px;
39     margin: 25px auto;
40 }
```

The other problem is that the growth of the cat is very sudden and uncomfortable to look at. We can make the cat grow smoothly using CSS transitions. We will cover the `transition` property in more detail in later chapters, but for now we can simply give it three values: a property to transition, the amount of time it takes to transition, and the timing curve. The property to transition could be `height`. Let's say it takes about 150 milliseconds to transition, and we can give it a timing function of `ease`. This timing function means that the transition will start off slow, get faster in the middle, and finish slowly<sup>5</sup>.

```
30 #cat {
31     display: block;
32     margin: 50px auto;
```

---

<sup>5</sup>More details about timing functions can be found at [https://www.w3schools.com/cssref/css3\\_pr\\_animation-timing-function.asp](https://www.w3schools.com/cssref/css3_pr_animation-timing-function.asp)



```

33     cursor: pointer;
34     height: 450px;
35     transition: height 150ms ease;
36 }

```

This is the equivalent of setting the following three properties individually:

```

30 #cat {
31     display: block;
32     margin: 50px auto;
33     cursor: pointer;
34     height: 450px;
35     transition-property: height;
36     transition-duration: 150ms;
37     transition-timing-function: ease;
38 }

```

If we try to hover over the cat now, we can see that the size does indeed increase gradually, but the movement still appears jerky somehow. This is because the cat's margin does not transition, but rather changes suddenly. This results in the counter and the results button jumping upwards, and then gradually moving back to where they were as the cat gets bigger, pushing them back downwards. One solution to this is to change the transition property from `height` to `all`, so that every property of the image transitions together.

```

30 #cat {
31     display: block;
32     margin: 50px auto;
33     cursor: pointer;
34     height: 450px;
35     transition: all 150ms ease;
36 }

```

However, setting the transition property to `all` is generally not best practice because it can be much slower to compute. An alternative is to use the three separate properties mentioned before, and simply pass both `height` and `margin` as transition properties.

```

30 #cat {
31     display: block;
32     margin: 50px auto;
33     cursor: pointer;
34     height: 450px;
35     transition-property: height, margin;

```

```

36     transition-duration: 150ms;
37     transition-timing-function: ease;
38 }

```

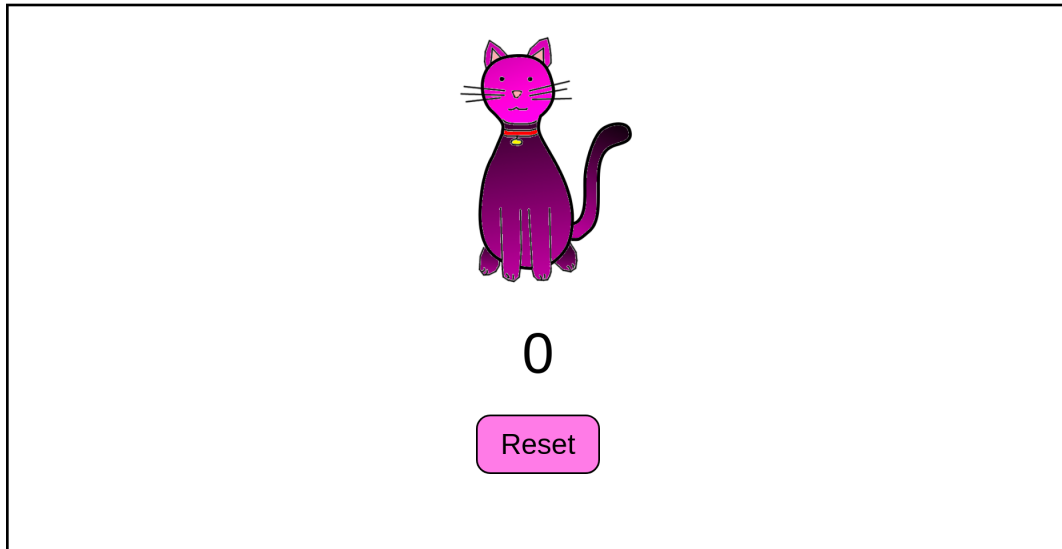


Figure 2.15: The cat now gets smoothly bigger when you hover over it

We can make the cat even more responsive by making it rotate in a circle when you click on it. We can do this by using the `transform` property. We can give it a default value of `rotate(0)` to indicate that by default it should not be rotated. Then, we can use the `:active` pseudo-class to change its `transform` to `rotate(360deg)` when it gets clicked on. This is one full rotation. Then, we need to specify that we also want to transition the `transform` property too.

```

30 #cat {
31     display: block;
32     margin: 50px auto;
33     cursor: pointer;
34     height: 450px;
35     transition-property: height, margin, transform;
36     transition-duration: 150ms;
37     transition-timing-function: ease;
38     transform: rotate(0);
39 }
40
41 #cat:hover {
42     height: 500px;
43     margin: 25px auto;
44 }
45

```

```

46 #cat:active {
47     transform: rotate(360deg);
48 }

```

This isn't quite perfect though, as 150ms is far too fast for a 360 degree rotation. It would hurt your eyes to look at. Instead, we can go back to using only one `transition` property, and give it the following value:

```

35     transition: height 150ms ease, margin 150ms ease, transform 500ms ease;

```

As you can see, the commas separate three distinct transitions. The first is for the `height`, the second for the `margin`, and the third — a much longer 500 millisecond transition — for the `transform`.

One side effect of this method of making the cat spin in a circle is that if you press down your mouse button on the cat and then release it while the cat is still mid-rotation, it will stop its rotation and smoothly rotate back. This means that if you spam the mouse button quickly, the cat just fidgets in place, which I think is rather cute.

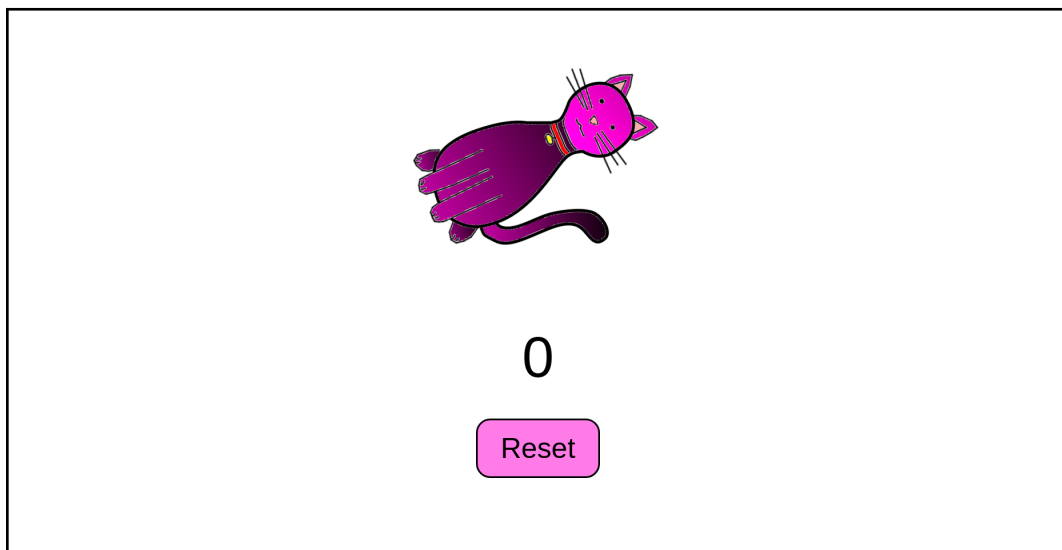


Figure 2.16: The cat spins in a circle when you click on it

We're almost done with the styling, but before we move on, let's just set a nice background colour for the page. We could do this with the `background-color` property of the `<body>` element like so:

```

1  body {
2      background-color: #ffe78f;

```

```
3 }
```

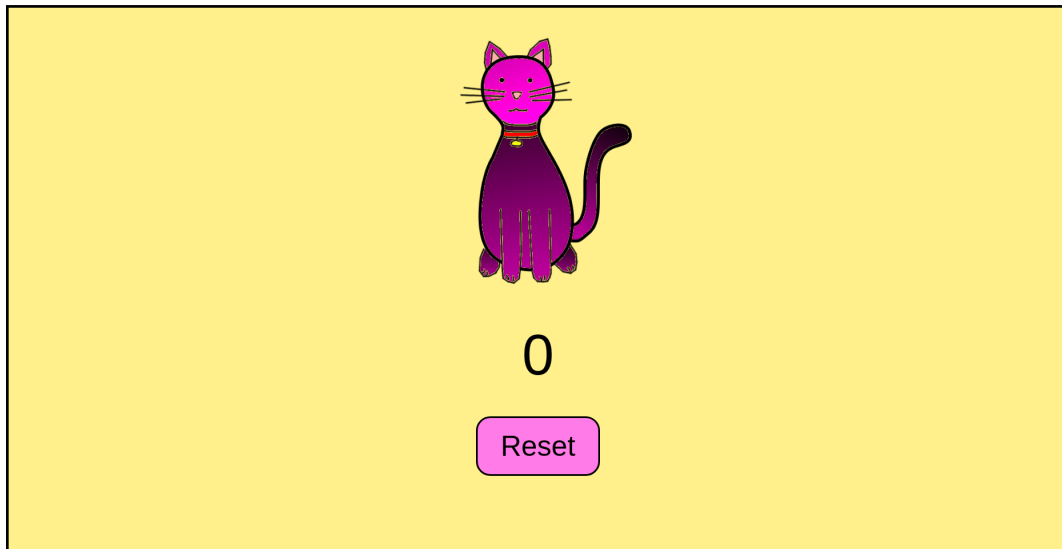


Figure 2.17: A solid background colour

However, I think it would be nice to have a smooth gradient background colour. We can do this by setting the `background-image` property. We can generate a value for this property using the `linear-gradient()` function, which takes several parameters. The first parameter is a direction for the gradient. The rest of the arguments are the colour stops. I want the gradient to go from the top downwards, so we can give the direction as `to bottom`. Alternatively, we could give it as `180deg`, or, since top-to-bottom is the default, we could omit this parameter entirely. For completeness' sake though, let's give it as `to bottom`. I also want to fade from `#ffd36b` at the top to the slightly lighter `#ffe78f` at the bottom. We can therefore write:

```
1 body {
2     background-image: linear-gradient(to bottom, #ffd36b, #ffe78f);
3 }
```

As you can see below, this leaves us with a slight problem

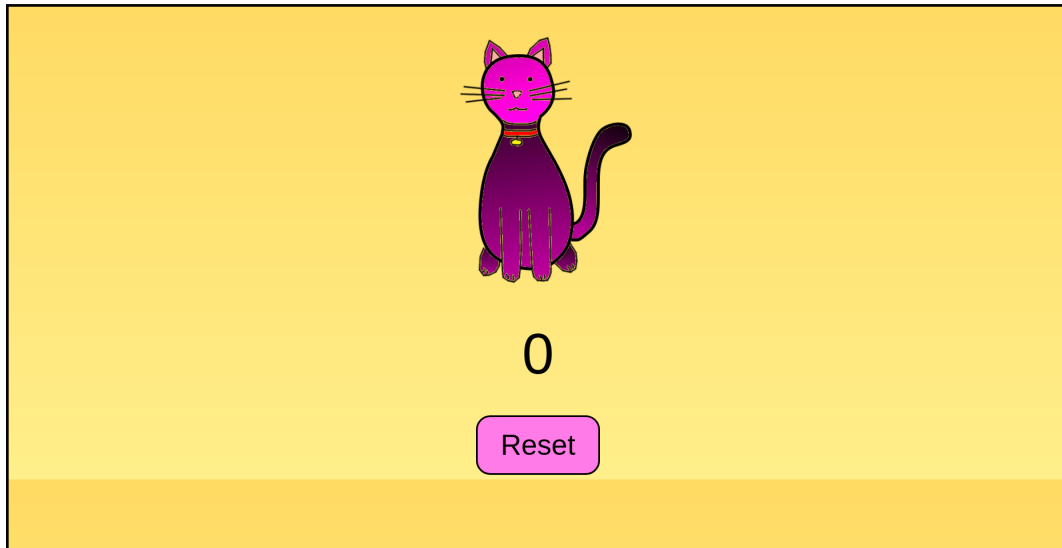


Figure 2.18: The gradient repeats when it gets to the bottom of the page's content

This is because the `<body>` element doesn't stretch down to the bottom of the page. We can fix this by simply ensuring that the heights of both the `<body>` element and the `<html>` element is 100% of the available space. We can select both of these elements using the CSS selector `body, html` and while we're here we might as well set the widths too, just for fun.

```
1 body, html {  
2     width: 100%;  
3     height: 100%;  
4 }
```

This very almost solves the problem, except now we have these ugly scroll-bars again.

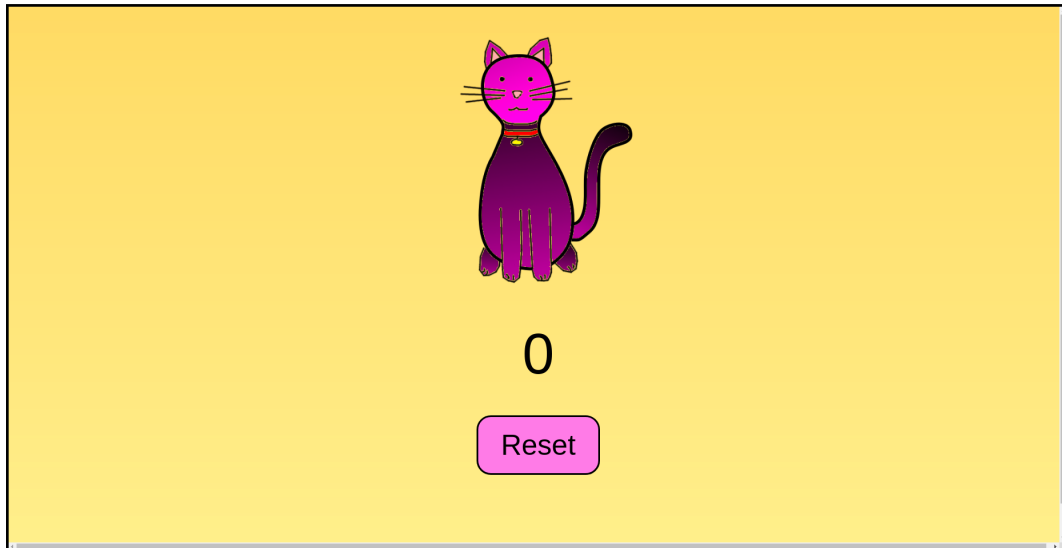


Figure 2.19: Ugly scroll-bars

This is due to the margin on the cat element. The `<body>` is responding strangely and is being offset from the top of the screen for some reason. We could remove the top margin of the cat, and instead put top padding on the `<body>`, but instead we can simply glue the body to the top-left of the screen. We do this by setting its `position` property to `absolute` which means that its position is relative to the `<html>` element, and then we can set that offset to 0 by setting the `top` and `left` properties to 0, meaning that the top edge of the `<body>` is 0 pixels away from the top edge for the screen, and likewise for the left edge. We can also get rid of the margin on the `<body>`.

```
6  body {  
7      background-image: linear-gradient(to bottom, #ffd36b, #ffe78f);  
8      margin: 0;  
9      position: absolute;  
10     top: 0;  
11     left: 0;  
12 }
```

Finally, the styling is complete.

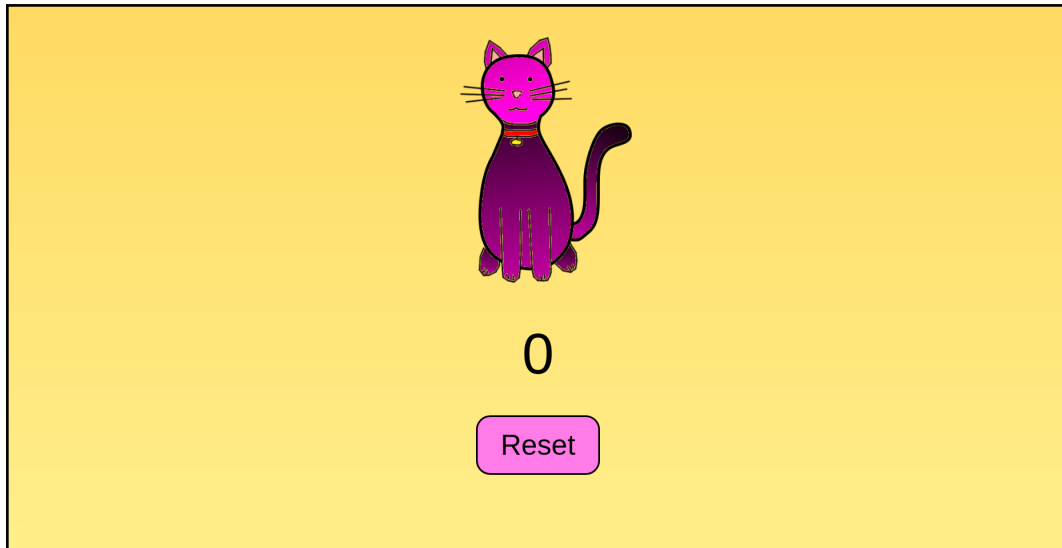


Figure 2.20: Pretty website

The finished CSS looks like this:

```
1  body, html {
2      width: 100%;
3      height: 100%;
4  }
5
6  body {
7      background-image: linear-gradient(to bottom, #ffd36b, #ffe78f);
8      margin: 0;
9      position: absolute;
10     top: 0;
11     left: 0;
12 }
13
14 *:focus {
15     outline: none;
16 }
17
18 #counter {
19     font-size: 100px;
20     font-family: Arial, Helvetica, sans-serif;
21     text-align: center;
22 }
23
24 #reset {
25     font-size: 50px;
26     border: 3px solid black;
27     background-color: #f09ae9;
28     cursor: pointer;
```

```
29     padding: 20px 40px;
30     margin: 50px auto 0 auto;
31     display: block;
32     border-radius: 25px;
33 }
34
35 #reset:hover {
36     background-color: #9e6299;
37 }
38
39 #reset:active {
40     background-color: #694165;
41 }
42
43 #cat {
44     display: block;
45     margin: 50px auto;
46     cursor: pointer;
47     height: 450px;
48     transition: height 150ms ease, margin 150ms ease, transform 500ms ease;
49     transform: rotate(0);
50 }
51
52 #cat:hover {
53     height: 500px;
54     margin: 25px auto;
55 }
56
57 #cat:active {
58     transform: rotate(360deg);
59 }
```

Now that we've got the content of the page and the style of the page, let's move on to the functionality – it's time for some JavaScript.

The first step is to create a JavaScript file. Let's call it *script.js* and put it into a folder named *js* in our project directory, which now looks like this:

```
clicker
├── index.html
├── img
│   └── wafflecone.png
├── css
│   └── styles.css
└── js
    └── script.js
```



Inside *index.html*, let's add in this script. We do so with a `<script>` tag like so:

```
<script src="./js/script.js"></script>
```

If you're wondering why we don't use a self-closing tag, such as

```
<script src="./js/script.js" />
```

It's because that doesn't work. Why? Absolutely no reason at all. It makes no sense whatsoever, and the sooner you internalise that feeling of hopelessness, ambiguous confusion, and badly-documented despair, the better you'll become at web development, so don't worry about it too much.

We do however have a choice. Do we put the `<script>` tag into the `<head>` or the `<body>`? Well if we put the script in the `<head>`, then it gets run before the cat, the counter, and the reset button are actually loaded into the page. If we try to access them directly from the script immediately, then they won't exist yet and our script won't work. We should therefore put the script at the bottom of the `<body>` such that our HTML looks like this:

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Clicker game</title>
5      <link rel="stylesheet" href="./css/styles.css">
6    </head>
7    <body>
8      
9      <div id="counter">
10         0
11      </div>
12      <button id="reset">Reset</button>
13      <script src="./js/script.js"></script>
14    </body>
15  </html>
```

Inside our *script.js* file, we can perform a little test to make sure that the script is running.

```
1  console.log("The script is running!");
```

The `console.log` function is built-in when you're running JavaScript code from a web browser. It will output data to the developer console, which you can access by opening the developer tools in your web browser. If you open up the page in your browser and take a look in the console, you'll see the message

```
The script is running!
```

Now we need to detect when the cat is clicked. To get a hold of the `<img>` element, we can run the handy dandy function `document.getElementById`, passing it a value of `"cat"`, which is the ID we gave to the `<cat>` element. We can see that function working by passing its result into `console.log`.

```
1 console.log(document.getElementById("cat"));
```

You should see the output:

```

```

This shows that it found the element successfully. In JavaScript, DOM elements (elements in the Document Object Model<sup>6</sup>) have a property called “onclick”, which we can set equal to a function which takes zero parameters. Inside this function, we can run a `console.log` to test it out. The syntax for this is as follows:

```
1 document.getElementById("cat").onclick = function () {  
2     console.log("click");  
3 }
```

This function should get run every time the element is clicked. If you refresh the webpage and then click on the cat, you should see that message appear in the console.

```
click
```

Now, once the cat has been clicked, we need to find out the value of the counter. We can do that

---

<sup>6</sup>[https://www.w3schools.com/js/js\\_htmlDOM.asp](https://www.w3schools.com/js/js_htmlDOM.asp)

by first finding the `<div>` element we're using for the counter by calling `document.getElementById("counter")`. We can find out the text in the counter by accessing the `innerText` property of this element. We can `console.log` this to see what we get.

```
1 document.getElementById("cat").onclick = function () {
2     console.log(document.getElementById("counter").innerText);
3 }
```

If we now click on the cat, you should see this appear in the console:

0

Let's store this value in a variable called `currentCount`. The syntax for that is as follows:

```
1 document.getElementById("cat").onclick = function () {
2     let currentCount = document.getElementById("counter").innerText;
3 }
```

We can then increase the counter by setting the `innerText` property of the counter element equal to `currentCount + 1`.

```
1 document.getElementById("cat").onclick = function () {
2     let currentCount = document.getElementById("counter").innerText;
3     document.getElementById("counter").innerText = currentCount + 1;
4 }
```

There's actually no need to call `document.getElementById("counter")` twice in this code. Instead we can just store the element in a variable like so:

```
1 document.getElementById("cat").onclick = function () {
2     let counter = document.getElementById("counter")
3     let currentCount = counter.innerText;
4     counter.innerText = currentCount + 1;
5 }
```

Now if we click on the cat, we see something a little unexpected. Instead of increasing to “1”, the counter increases to “01”. Weird, but still technically correct I guess so let’s see what happens if we click it again. “011”. Yeah, that’s not right.



Figure 2.21: This is what happens when you click the cat 5 times

You can sort of see what’s going on here. It’s taking the character “0”, and adding the character “1”, ending up with the string “01”. JavaScript is what is known as a dynamically typed language. That means that the datatype (integer, string, etc.) is decided based on the context in which the variable is used. When we store the “innerText” of the counter in the variable `currentCount`, it stores it as text (a string), and a weird quirk of JavaScript is that when you add an int (short for integer) to a string, it treats them both as strings.

We need a way of telling the program to treat the counter value as an int. Luckily, JavaScript has a `parseInt` function which does the trick.

```
1 document.getElementById("cat").onclick = function () {
2     let counter = document.getElementById("counter")
3     let currentCount = parseInt(counter.innerText);
4     counter.innerText = currentCount + 1;
5 }
```

Now when we click on the cat, the counter increases by 1.

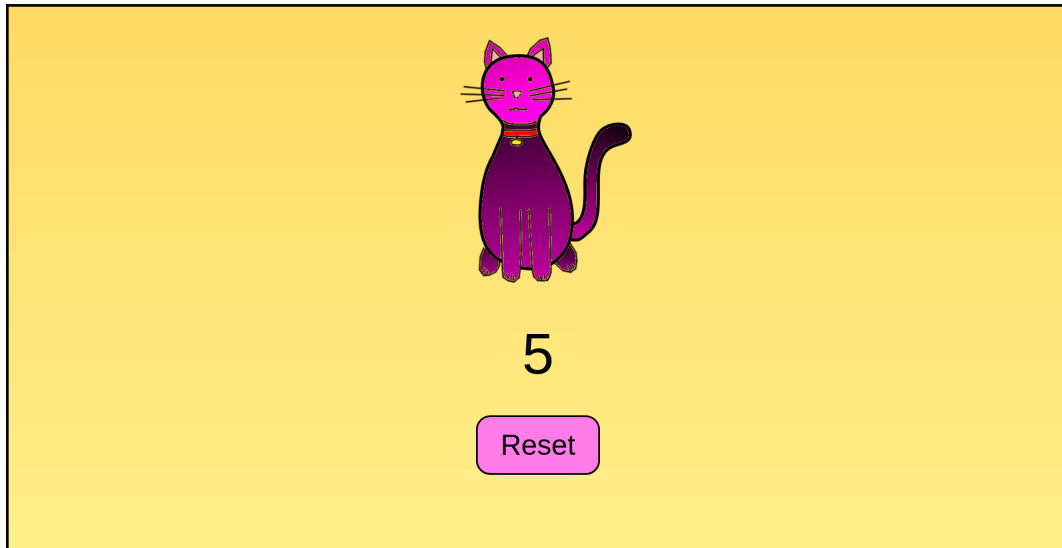


Figure 2.22: This is what happens now when you click the cat 5 times

There's some shorthand for achieving what we just did with `parseInt`, which is just to put a “+” before the name of the variable. This will convert it into a number.

```
1 document.getElementById("cat").onclick = function () {
2     let counter = document.getElementById("counter")
3     let currentCount = +counter.innerText;
4     counter.innerText = currentCount + 1;
5 }
```

Now we just need to hook up the reset button and we're done. We can find the button with `document.getElementById("reset")` and give it an onclick function which finds the counter and sets its “innerText” property back to zero.

```
7 document.getElementById("reset").onclick = function() {
8     document.getElementById("counter").innerText = 0;
9 }
```

We can test this out and see that it works! Before we finish this project though, there's just one more almost-insignificant optimisation to make. At the moment, `document.getElementById("counter")` gets called every time we click the cat or the reset button, when really it only needs to be called once. At the start of the script we can run this function, and store the result in a variable.

```
1 const counter = document.getElementById("counter");
```

```
2
3 document.getElementById("cat").onclick = function () {
4     let currentCount = +counter.innerText;
5     counter.innerText = currentCount + 1;
6 }
7
8 document.getElementById("reset").onclick = function() {
9     counter.innerText = 0;
10 }
```

Let's talk for a second about why we used `const` there instead of `let` like we used for the other variables.

In JavaScript there are three main ways of declaring variables — `const`, `let`, and `var`. We use `const` when we know that the value of a variable is never going to change, as this lets the compiler be slightly more memory efficient in ways that you don't have to care about. We use `let` for all other variables. Never use `var`. We hate `var`. The difference between `let` and `var` is one of scoping. For example, if you define a variable with `let` inside a function, loop, or other code block, you won't be able to access that variable outside of the code block. With `var` on the other hand, you'll be able to access it all over the place, which is gross. There should never be a situation where you would have to use `var`.

That being said, our little clicker game is done. To recap, here is the HTML:

```
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Clicker game</title>
5         <link rel="stylesheet" href="./css/styles.css">
6     </head>
7     <body>
8         
9         <div id="counter">
10             0
11         </div>
12         <button id="reset">Reset</button>
13         <script src="./js/script.js"></script>
14     </body>
15 </html>
```

Here is the CSS

```
1 body, html {
2     width: 100%;
```

```

3     height: 100%;
4 }
5
6 body {
7     background-image: linear-gradient(to bottom, #ffd36b, #ffe78f);
8     margin: 0;
9     position: absolute;
10    top: 0;
11    left: 0;
12 }
13
14 *:focus {
15     outline: none;
16 }
17
18 #counter {
19     font-size: 100px;
20     font-family: Arial, Helvetica, sans-serif;
21     text-align: center;
22 }
23
24 #reset {
25     font-size: 50px;
26     border: 3px solid black;
27     background-color: #f09ae9;
28     cursor: pointer;
29     padding: 20px 40px;
30     margin: 50px auto 0 auto;
31     display: block;
32     border-radius: 25px;
33 }
34
35 #reset:hover {
36     background-color: #9e6299;
37 }
38
39 #reset:active {
40     background-color: #694165;
41 }
42
43 #cat {
44     display: block;
45     margin: 50px auto;
46     cursor: pointer;
47     height: 450px;
48     transition: height 150ms ease, margin 150ms ease, transform 500ms ease;
49     transform: rotate(0);
50 }
51
52 #cat:hover {
53     height: 500px;

```

```
54     margin: 25px auto;
55 }
56
57 #cat:active {
58     transform: rotate(360deg);
59 }
```

And here is the JavaScript

```
1  const counter = document.getElementById("counter");
2
3  document.getElementById("cat").onclick = function () {
4      let currentCount = +counter.innerText;
5      counter.innerText = currentCount + 1;
6  }
7
8  document.getElementById("reset").onclick = function() {
9      counter.innerText = 0;
10 }
```

You can find the full project file at <https://github.com/slippedandmissed/LoveOfGod> and you can check out the website for yourself at <https://slippedandmissed.github.io/LoveOfGod/clicker>



## 3 Setting up a Node.js server

So far we've been focusing on the front-end of our website. However, we also need to set up a server which can listen out for requests, and send our website back to a client. We can write this server in almost any high level programming language. Two of the most common options for this are PHP and Node.js.

Node.js is not in itself a language. It is actually what is known as a “JavaScript runtime”<sup>1</sup> and it allows us to write our back-end server code in JavaScript. This is handy, because front-end code is also in JavaScript, and this minimises the number of programming languages we have to learn. In this book we will write our server in Node.js and this approach is called “Full-stack JavaScript development”.

You can install Node.js from <https://nodejs.org/en/download/> and once you have done so you will have access to the command-line tools “node” and “npm”. NPM stands for node package manager, and is what we will be using to manage the dependencies of our server.

The first step is to create a project directory. I will call this “node-server-tutorial”. Once inside this folder, run the command

```
npm init
```

from your terminal or command prompt. You will then be prompted for several pieces of information such as a name for the project (package) but if you're not sure then you can always press enter without typing anything to either leave the field blank or use the default value (shown in brackets). After the command has run you will see a file has been created in your project directory called “package.json” and its contents will look something like this:

```
1 {
2   "name": "server-tutorial",
3   "version": "1.0.0",
4   "description": "An example node server",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
```

---

<sup>1</sup><https://nodejs.org/en/about/>

### 3 Setting up a Node.js server

```
8   },
9   "author": "Morgan Saville",
10  "license": "ISC"
11 }
```

This is written in a language called JSON (JavaScript Object Notation) and it is, as the name suggests, the same syntax we use to define objects in JavaScript. For now, the only important part to note is where it defines the value “main” equal to “index.js”. This tells Node.js where our server code is written. This file however does not exist yet, so we can go ahead and create it in the same directory. Our project folder now has this structure:

```
node-server-tutorial
├── index.js
└── package.json
```

To write our server we need to use a node package called express. We can install this using npm with the following command

```
npm install express --save
```

Now we can create our server by writing the following code into “index.js”

```
1  const express = require("express");
2  const app = express();
3  const port = 3000;
4
5  app.listen(port, () => {
6    console.log(`App listening on port ${port}`);
7  });
```

Let’s go through this code one bit at a time. The first line uses the “require” keyword to bring the express package into our code. This returns a function which we are assigning to the variable **express**. Since we will never change the value of this variable – i.e. it will always be referring to the function we’ve pulled in – we can use the **const** keyword to declare it.

In the second line, we invoke this function. It returns an object which represents our web application. We store this object in another constant variable called **app**.

In the third line we define some port number on which the app should listen for requests from clients. This can be really whatever positive integer you like, although you should note that some ports are reserved for specific processes<sup>2</sup>.

The next part is a bit trickier. The `app` object has a function called `listen` which makes the app listen for requests on a given port. However, this function actually takes two parameters. The first one is the port, and the second one is a function called a “callback”. Sometimes certain functions (such as `app.listen`) can take a long time (relatively speaking) to execute, and so if we have some more code which relies on this function having finished, instead of waiting for the function to finish and then just running this code we can pass the function a callback. To do this is to tell the function “do your thing, and when you’re done, call this function” and in the meantime we can go and run some other code.

The syntax for writing functions in JavaScript can be quite confusing. There are three main ways of defining a function. The first is to use the `function` keyword followed by the function name, the parameter list, and the code like so.

```
function add(x, y) {  
  return x+y;  
}
```

This will likely be familiar to you if you have experience in other high level languages such as Java, Python, or C#. However, JavaScript allows you to define functions without actually giving them a name like so:

```
function (x, y) {  
  return x+y;  
}
```

You might be wondering how you could even invoke a function if it doesn’t have a name. Well the answer is simply to assign it to a variable.

```
let add = function (x, y) {  
  return x + y;  
}
```

Now you can call `add()` just like before.

---

<sup>2</sup><http://www.meridianoutpost.com/resources/articles/well-known-tcpip-ports.php>

### 3 Setting up a Node.js server

So far the above two methods of making a function called “add” which returns the sum of its two parameters are functionally identical. However, there is a third, very subtly different method. This is called arrow notation, and it uses an arrow “=>” instead of the `function` keyword.

```
let add = (x, y) => {  
  return x + y;  
}
```

For our purposes at the moment, these three methods are all the same. However, using arrow notation is actually different in terms of what is known as “context”, but this is something we will cover later. For now, just know that these are the three main ways of defining a function. Looking back at our server code, you can see that the two parameters passed to `app.listen` are the port, and then a callback function defined using arrow notation. It takes no parameters, and it simply outputs a log message.

You may also notice the weird format of the message

```
`App listening on port ${port}`
```

Defining a string using the backtick character ``` rather than a quote mark `'` or a double-quote mark `”` means that the string is a so-called “template string”. This allows you to embed variables or expressions inside the string by putting a dollar sign followed by the variable or expression inside curly brackets. The above string is equivalent to:

```
"App listening on port " + port
```

We can now launch the server by running the command

```
node index.js
```

In the console you should see the message:

```
App listening on port 3000
```

And indeed, if you navigate to the url `http://localhost:3000/` you will see the following page.

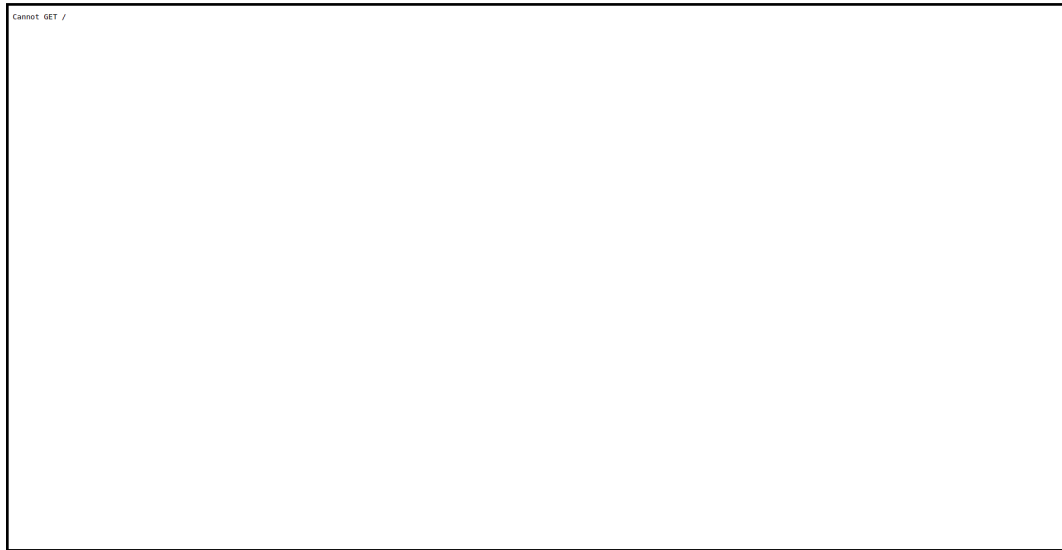


Figure 3.1: The server is up and running

A little underwhelming, sure, but it means that the server is up and running. The error message stems from the fact that even though the server is up, we haven't written any code which actually handles the requests, or sends data back.

There are several types of HTTP request, but the two main ones are GET and POST. A GET request is typically used when you want to access a resource. When you type a URL into your web browser, the browser is performing a GET request. You can send some parameters along with your request by putting a question mark (?) after the URL and then putting a variable name, followed by an equals sign (=) and a value for that variable. You can send multiple variables by putting an ampersand (&) after the first value and then continuing with the second variable name etc.

For example, all YouTube videos are on the page `https://www.youtube.com/watch`. The server only knows which video to actually show you based on the value you provide for the parameter `v`.

<https://www.youtube.com/watch?v=dQw4w9WgXcQ>

While GET requests send their parameters as part of the URL itself, POST requests send them in the background, where the user typically can't see them. For example, if you post a comment on a blog, the comment data is sent to the server via a POST request. If you therefore copied and pasted the URL to a friend and they clicked it, it would not result in the same comment being posted from their account — the data is not part of the URL.

### 3 Setting up a Node.js server

At the moment, we need to handle a GET request to the location “/”, meaning the root of the website. We can do this using the function `app.get` which takes two parameters. The first is a route (the location being requested. Here it is “/”). The second parameter is a callback. This callback takes two objects as its parameters, called `req` and `res`. This callback will be invoked every time there is a GET request to “/”.

```
1  const express = require("express");
2  const app = express();
3  const port = 3000;
4
5  app.get("/", (req, res) => {
6    // Do something here
7  });
8
9  app.listen(port, () => {
10    console.log(`App listening on port ${port}`);
11  });
```

The `req` object contains information about the request itself, for example the GET parameters passed in by the URL, and information about the client. The `res` object has functions which we can use to send a response to the client. For example, let’s just send back a message of our choosing.

```
5  app.get("/", (req, res) => {
6    res.send("All the birds died in 1986 due to Reagan killing them and
    ↪ replacing them with spies that are now watching us. The birds work
    ↪ for the bourgeoisie.");
7  });
```

If we kill the server by pressing Ctrl+C in the terminal window which is running it, and then restarting by running

```
node index.js
```

And then refresh our browser window, we will see our message.



Figure 3.2: Our message is being rendered

Don't forget to restart the server every time you make changes to "index.js".

Now that we've got text being sent back to the client, we can move on to sending back HTML. Now it's not just enough to use `res.send` to send back the HTML as a string to the client, because the HTML contains links to some CSS and some JavaScript which will also be requested separately. For us, these are "static", meaning that their content doesn't depend on the GET parameters, and we don't need to generate them with code. This means we can put our HTML, CSS and JavaScript into a subdirectory in our project folder called "static". I will use the clicker game code from the previous chapter. Our project directory now looks like this:

```
node-server-tutorial
├── node_modules
│   └── ...
├── static
│   ├── css
│   │   └── styles.css
│   ├── img
│   │   └── wafflecone.png
│   ├── js
│   │   └── script.js
│   └── index.html
├── index.js
├── package-lock.json
└── package.json
```

If you're wondering where "node\_modules" and "package-lock.json" came from, they were generated when we ran

### 3 Setting up a Node.js server

```
npm install express --save
```

And you don't need to give them much mind. We now need to add in a line of code which tells the server that every time a client requests a resource relative to the root of the webpage, it should find that resource inside the “static” directory and send it back. For example, if the client requests “/index.html”, the server should send back the contents of “static/index.html”. If the client requests “/img/wafflecone.png”, the server should send back “static/img/wafflecone.png” etc.

Luckily, express makes this very easy for us.

```
5 app.use(express.static("static"));
```

Our finished server code now looks like this:

```
1 const express = require("express");
2 const app = express();
3 const port = 3000;
4
5 app.use(express.static("static"));
6
7 app.listen(port, () => {
8   console.log(`App listening on port ${port}`);
9 });
```

And if we restart the server and navigate to `http://localhost:3000/` we will see our clicker game.



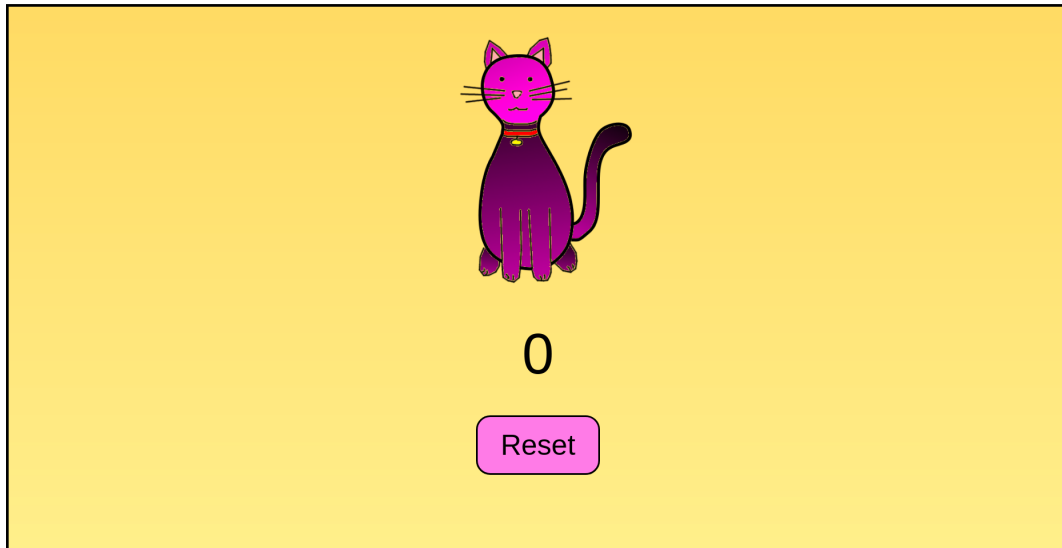


Figure 3.3: The server is now serving our clicker game

Another question now arises: why did the server send us “index.html” when we only requested the root of the website “/”? Well, this is just another convention of web requests. If no filename is specified, by default the client receives the file named index if there is one. It’s just something arbitrary to remember.



## 4 Painting By Numbers