

# Web Components

An introduction to the Future



Hello everyone! It is so lovely to be here, thank you for having me. I'm here to talk about web components: what they are, how to use them, and what role they will play in the future of web development.

# My name is Tobias



My name is Tobias and i am from Gothenburg, in the west of Sweden.



I work at **squeed**

---



Here I work as a web developer consultant, and also part time as a frontend evangelist at the company Squeed, spreading knowledge and interest about all things frontend, both within and without the company. And so that's why I'm here today - to talk to you about web components.

But first, I want to take a step back and talk about another amazing invention that makes our lives so much easier every day. I am of course talking about...



---

squeezed

...ice delivery. What a fantastic service! Ice right to your door, you can put it in your ice box to keep your food cold so it doesn't spoil. If it's a hot day you can grab the ol' ice pick and chop off some pieces for your drink. What would we do without ice delivery, right?



Except, of course... no one has ice delivery anymore



squeqd

People just have refrigerators instead. And the reason why is of course obvious. A fridge is in every way more convenient. Cheaper for customers in the long run. Electricity is so much easier to deliver than ice. And when you buy a house or apartment, the fridge is usually already there waiting for you. You might even call the electricity a built in, or native feature, of the house. Even though a lot of ice delivery people had to find new jobs, moving to refrigerators was clearly...

# *Upgrade*

---

squared

...an upgrade for everyone involved.  
You know what else upgrades?

# Web tech upgrades constantly

like ALL the time omg give us a break...



The web. All the time. Sometimes it feels like hardly a day goes by without some new web tech being announced. But one of the most amazing things about web tech is its backwards compatibility. Even though new things are added to the specifications all the time, the old stuff still works. If it didn't, then suddenly a lot of old web pages would just stop working. Everyone would have to constantly upgrade and maintain their web sites, and nothing else would get done.

Ice boxes still work, but no new houses are built with them. We've moved on to a newer technology, and everyone benefits from it, even though the initial transition can be a little cumbersome, costly, or even scary. Let's take a look at another technology:



jQuery! What a fantastic library! When it came around in 2006 it was a tremendous upgrade from the way we used to query and manipulate the DOM. It made web development easier, and allowed us to create better experiences for our users.



But not many people use jQuery anymore. At least... it's not the first thing they reach for when starting a new project, like they would have done 10 years ago.

# Native browser features replaced jQuery



And this is because many of the good practices established by jQuery made it into the JavaScript and CSS specifications, and thus became native browser features. The functionality that jQuery provided became obsolete. Now we are able to do many of the things jQuery did, but without having to ship a bundle of jQuery code to the user. They already have what they need in their web browsers. And so while the transition away from jQuery may have been a little cumbersome, in the end..

# *Upgrade*

squqrd

...it was an upgrade for everyone involved. Kind of like how the ice delivery was replaced by built-in electricity.  
Now, let's talk about technology that many of us probably use every single day.



JavaScript frameworks! What fantastic technology! And I'm talking here about the mostly client-side frameworks like React or Vue. These babies do so much for us, and they enable us to build so many cool things. But why have they become so popular in recent years? Well, there are few major problems that these frameworks help us solve.

The first is that of keeping state and UI in sync. We update the state - the framework automatically updates the UI. We never have to write a DOM-modifying line of code again - the frameworks do all this for us, and they do it well.

The second is that of componentization. As programmers we want to be able to structure our code by splitting it up into logical, reusable chunks. This is good for the longevity of the codebase, and good for the product in the long run. I'm sure you know all about this. But since there hasn't been a native solution for this on the web, we've had to resort to things like template engines and JavaScript tooling to be able to write componentized code on our end, while the browser receives a packaged, bundled, and transformed version of our code that it can understand and run. This actually puts a distance between the code we write and the code the users run. And I argue, that the bigger this distance, the less control and understanding we have over what actually happens in our applications, and the harder it becomes to ship reliable software. But, as long as the tools are reliable, we still benefit greatly from the componentization that they enable us.

There are also additional problems that are solved within frameworks, like CSS encapsulation or single page routing.

# Framework problems



The frontend frameworks are not without their drawbacks of course. There can be a lot of domain specific knowledge that you need to acquire in order to understand and be able to work efficiently with your framework of choice. And, while rare, they can of course ship bugs of their own, which you have little control over. On the users side, we have to include the framework runtime, along with our application-specific code, which increases the total amount of data that needs to be downloaded. This, of course, has its own set of negative consequences. Some frameworks minimise or mitigate this in different ways, but we always have to ship something extra.



Enter, then, web components.

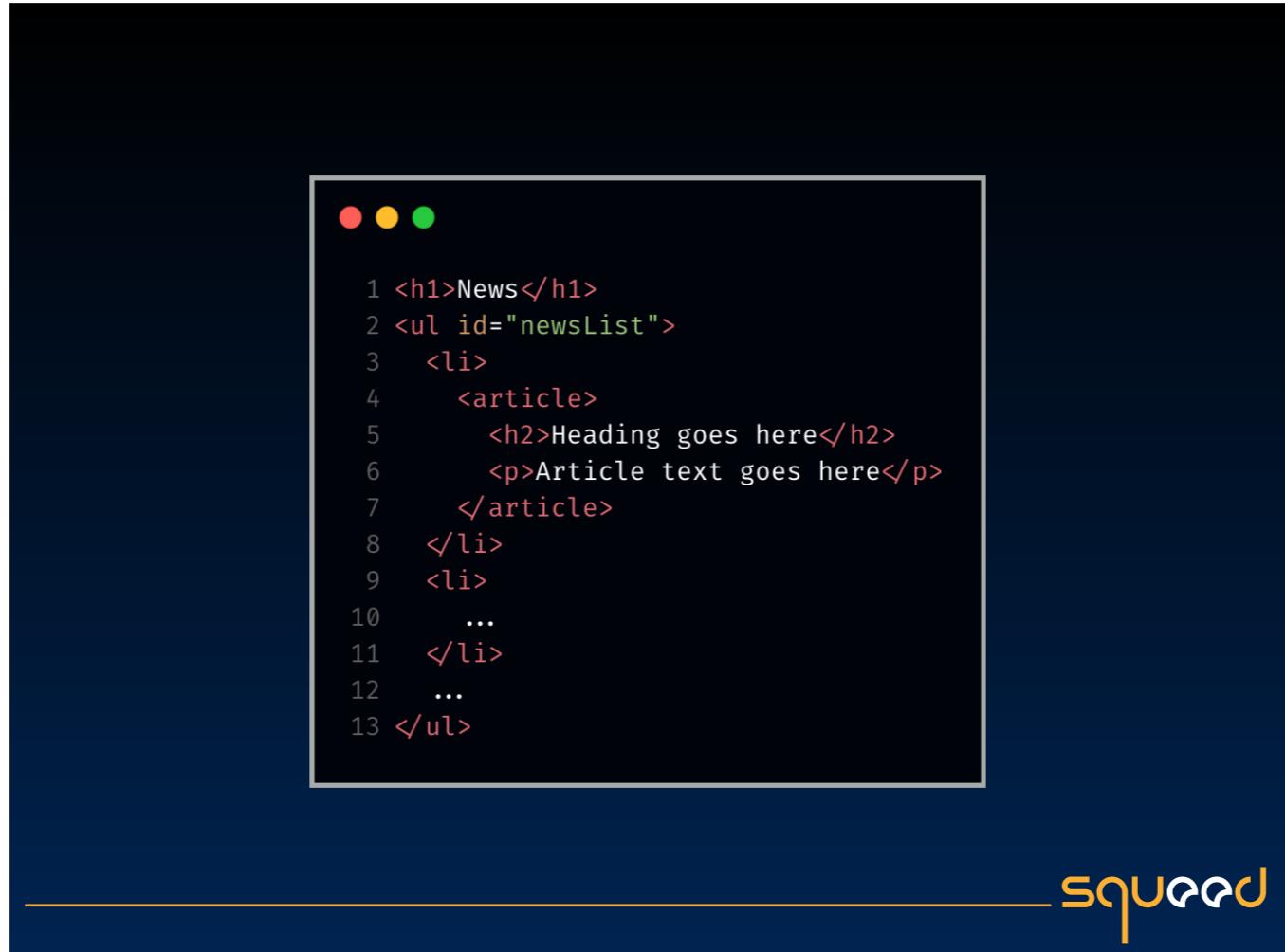
Web components are not a magical solution to every aspect of web development, but we will be able to leverage the power of componentization on a native level. In some circumstances this might lead to a frontend framework not being necessary at all. Sometimes you might find that a combination of technologies produce the best result. But I'll talk more about the use cases later, let's take a closer look at what web components actually are comprised of!

# HTML Templates



Web components are actually not just one thing, but a collection of browser APIs that we collectively call Web components. The first of these APIs that we'll take a look at are HTML templates.

So, let's imagine we have a news page, and every little news article follows the same pattern and has the same markup. It might look something like this:



A screenshot of a terminal window on a dark background. The window has three colored window control buttons (red, yellow, green) at the top left. Inside the window, there is a block of numbered HTML code:

```
1 <h1>News</h1>
2 <ul id="newsList">
3   <li>
4     <article>
5       <h2>Heading goes here</h2>
6       <p>Article text goes here</p>
7     </article>
8   </li>
9   <li>
10    ...
11  </li>
12  ...
13 </ul>
```

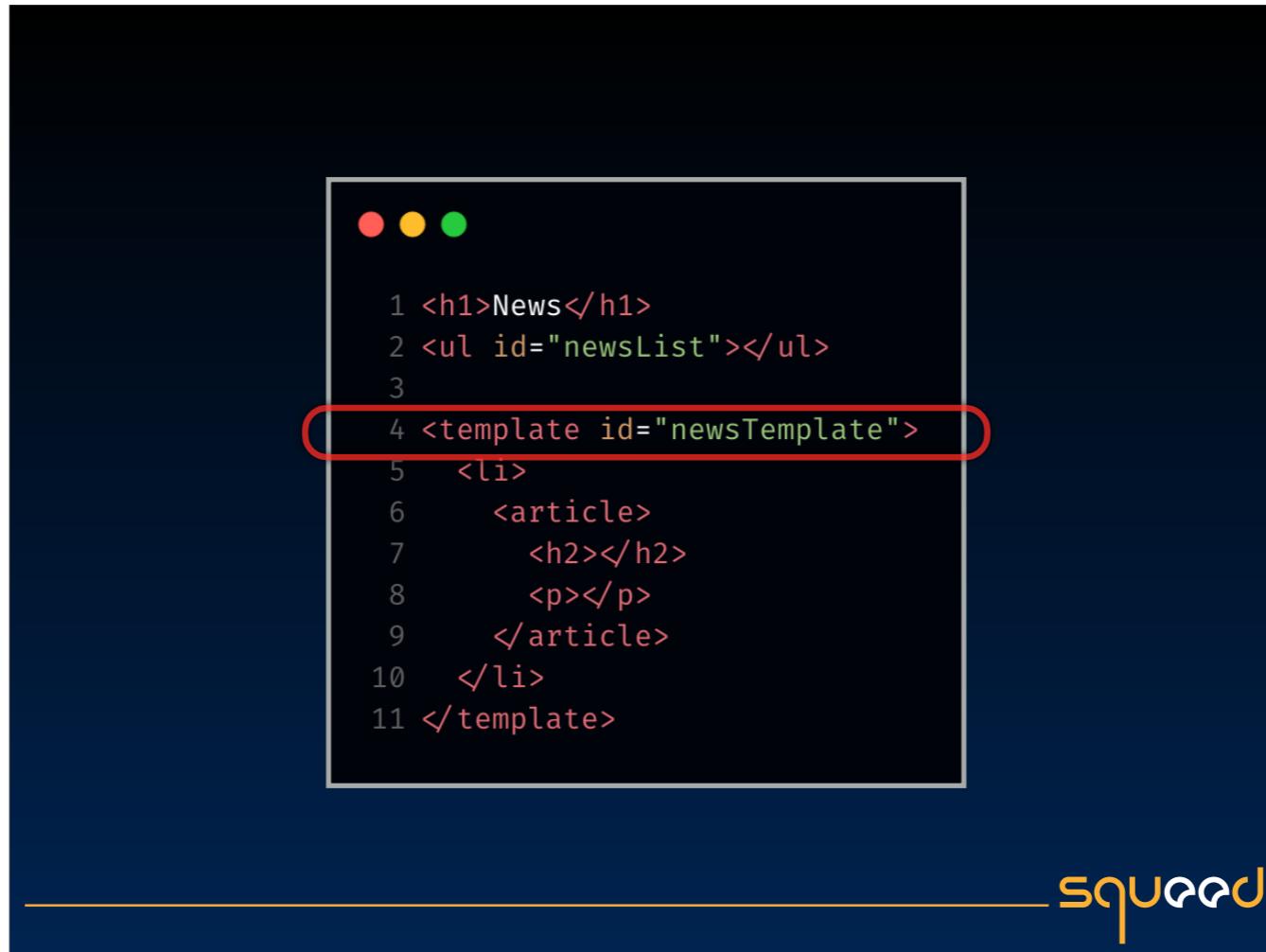
squqrd

We have a list of news articles, wrapped in article-tags, and each of these have a header and some content. We then repeat this for each news item we want to display. Now, of course, we wouldn't want to go in and manually write all of these tags, so what we would do is maybe render this page on the server. The server would get the news articles and then calculate the resulting html and send that to the requesting user. Works great, even though the calculating time on the server adds a little time to the response.

Another thing we could do is to pre-calculate the html document, so that there is no extra time spent on the server and the user gets a full, complete document right away. This is usually achieved using some form of static site generator, and requires one kind of templating language or other. Whenever there is a new news article, we would have to re-generate the documents before the users can request them, so this is not the best strategy if you have data that updates very often, like only seconds between updates, or maybe even faster.

Frontend frameworks solve this by sending a minimal html page to the user, a script bundle, and then it fetches data asynchronously, probably displaying a nice loading animation in the meantime, and then when the data arrives it is rendered out according to our specification, and we don't have to repeat ourselves.

With HTML templates we now have an additional option.



A screenshot of a terminal window on a dark background. At the top left are three colored dots: red, yellow, and green. Below them is some code:

```
1 <h1>News</h1>
2 <ul id="newsList"></ul>
3
4 <template id="newsTemplate">
5   <li>
6     <article>
7       <h2></h2>
8       <p></p>
9     </article>
10   </li>
11 </template>
```

The line `<template id="newsTemplate">` is highlighted with a red oval.

squared

But how would we do this with HTML templates? Right here in our html we can wrap a bunch of markup in the template-tag. Just like, say, the head-tag, this will not be displayed on the page, but still available in the DOM. So what we've wrapped in the template-tag here is the repeating part of our news page - the list item and subsequent article. In order to repeat and populate the page with these items we now have to use a little bit of JavaScript.



```
1 const newsList = document.querySelector("#newsList");
2 const template = document.querySelector("#newsTemplate");
3
4 data.forEach((item) => {
5   const clone = template.content.cloneNode(true);
6   clone.querySelector("h2").textContent = item.headline;
7   clone.querySelector("p").textContent = item.content;
8
9   newsList.appendChild(clone);
10});
```

The screenshot shows a browser's developer tools console window. It contains a block of JavaScript code. The first two lines of the code are highlighted with a red rounded rectangle. The code itself is as follows:

```
1 const newsList = document.querySelector("#newsList");
2 const template = document.querySelector("#newsTemplate");
3
4 data.forEach((item) => {
5   const clone = template.content.cloneNode(true);
6   clone.querySelector("h2").textContent = item.headline;
7   clone.querySelector("p").textContent = item.content;
8
9   newsList.appendChild(clone);
10});
```

squared

Here is the content of a script tag on the same page. The first two lines queries the document for two elements: the list, which we will be putting our list items into, and the template element.



```
1 const newsList = document.querySelector("#newsList");
2 const template = document.querySelector("#newsTemplate");
3
4 data.forEach((item) => {
5   const clone = template.content.cloneNode(true);
6   clone.querySelector("h2").textContent = item.headline;
7   clone.querySelector("p").textContent = item.content;
8
9   newsList.appendChild(clone);
10 });

```

The code demonstrates the use of `document.querySelector` to select the `newsList` and `newsTemplate` elements. It then uses `data.forEach` to iterate over each item in the `data` array. For each item, it creates a clone of the `newsTemplate` element using `cloneNode(true)`. It then updates the `h2` and `p` elements within the cloned template to reflect the current item's `headline` and `content`. Finally, it appends each cloned template to the `newsList`.

squared

We then take our data, which in a real scenario would probably come from a backend, and for each news item in the data we do the following:



```
 1 const newsList = document.querySelector("#newsList");
 2 const template = document.querySelector("#newsTemplate");
 3
 4 data.forEach((item) => {
 5   const clone = template.content.cloneNode(true);
 6   clone.querySelector("h2").textContent = item.headline;
 7   clone.querySelector("p").textContent = item.content;
 8
 9   newsList.appendChild(clone);
10 });

```

The code demonstrates how to clone a template node and append its cloned instances to a list element. A red circle highlights the line `const clone = template.content.cloneNode(true);` to indicate the cloning operation.

squared

First, we clone the template. It is a clone of this kind that we can append to the DOM so that it becomes visible.

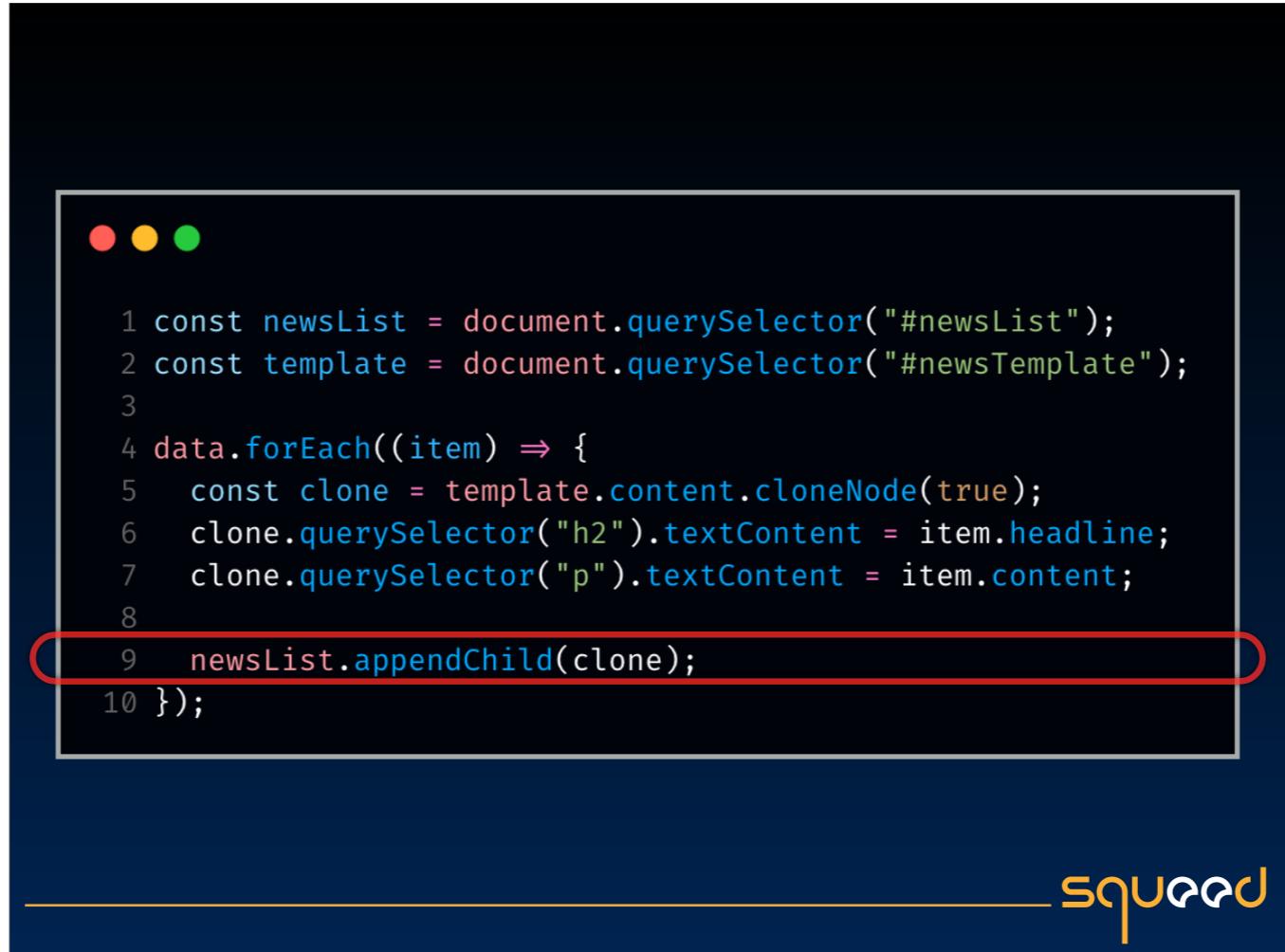


```
 1 const newsList = document.querySelector("#newsList");
 2 const template = document.querySelector("#newsTemplate");
 3
 4 data.forEach((item) => {
 5   const clone = template.content.cloneNode(true);
 6   clone.querySelector("h2").textContent = item.headline;
 7   clone.querySelector("p").textContent = item.content;
 8
 9   newsList.appendChild(clone);
10 });

```

The image shows a terminal window on a Mac OS X desktop. The window has three colored window control buttons (red, yellow, green) at the top left. The background of the window is dark. Inside, there is a white rectangular area containing the provided JavaScript code. Lines 6 and 7 are highlighted with a red rounded rectangle. Below the terminal window, the Mac OS X desktop background is visible, featuring a blue gradient with a subtle geometric pattern. In the bottom right corner of the desktop, there is a small, semi-transparent watermark or logo that reads "SQUARED" in a stylized, orange and blue font.

But before we make it visible we find the h2- and p-tags in the clone and set their text content to the news items heading and content, respectively.



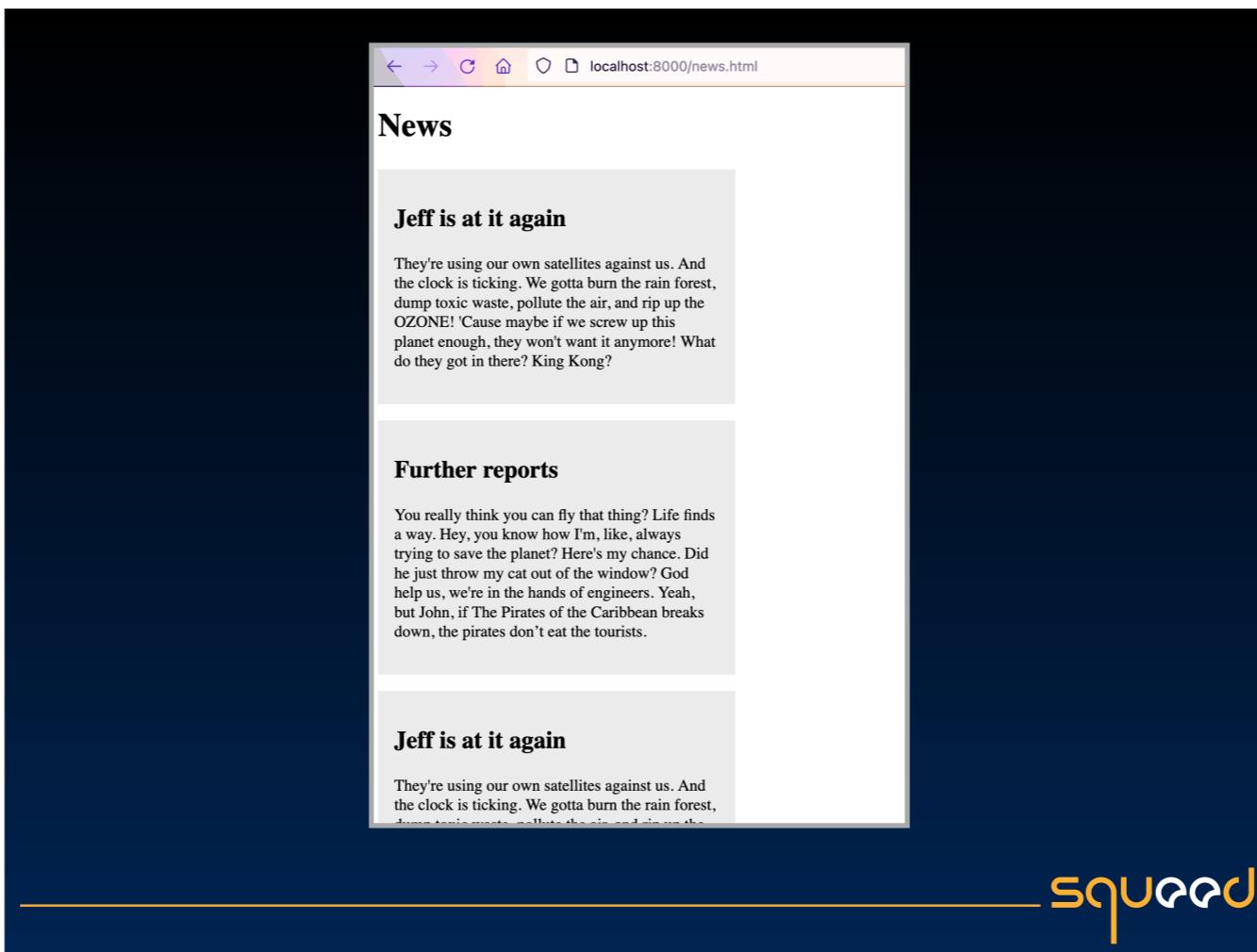
```
 1 const newsList = document.querySelector("#newsList");
 2 const template = document.querySelector("#newsTemplate");
 3
 4 data.forEach((item) => {
 5   const clone = template.content.cloneNode(true);
 6   clone.querySelector("h2").textContent = item.headline;
 7   clone.querySelector("p").textContent = item.content;
 8
 9   newsList.appendChild(clone);
10 });

```

The code demonstrates how to clone a template element and append its clones to a news list. The line `newsList.appendChild(clone);` is highlighted with a red oval.

squared

And then finally, we append the clone to the list element. We do all this for each data item and...



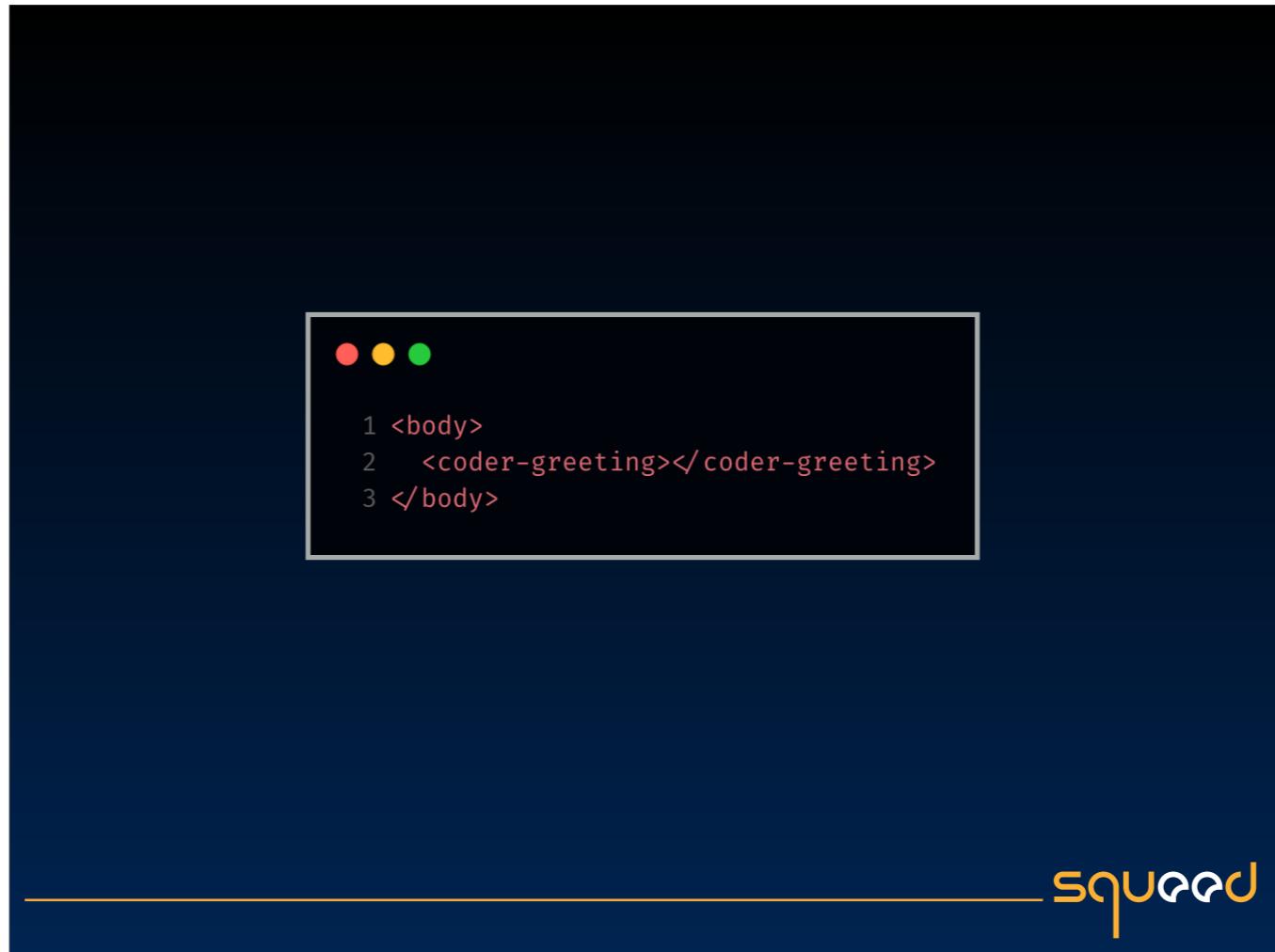
...here is the result!

Using HTML templates we can reuse HTML code, using only native browser APIs. We could have done the same without templates, by just directly inserting the DOM nodes every iteration, but a HTML template is both more performant, and it makes use of the declarative nature of HTML, making the code more easily readable, and closer to what will be rendered in the DOM in the end.

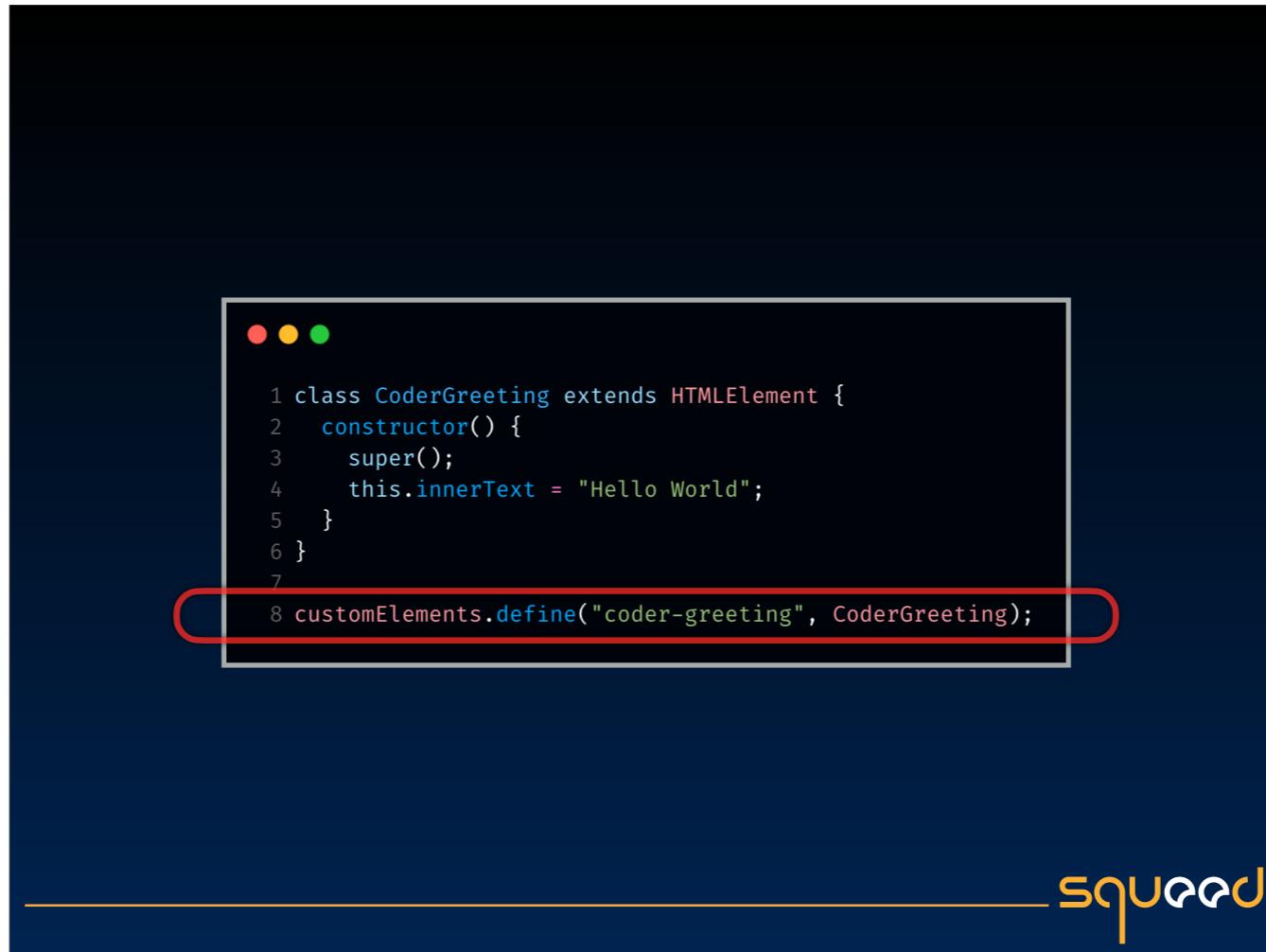
# Custom elements

squared

Custom elements is the second of the APIs that make up web components.



If you take a html document and write a tag that doesn't exist in the specification, like this tag here "coder-greeting", you get nothing. If you put some text content in it, the browser might treat it as a div or something, I think that depends on the browser, but it's invalid html, and should not be used like that. However! With custom elements, we can make it valid.

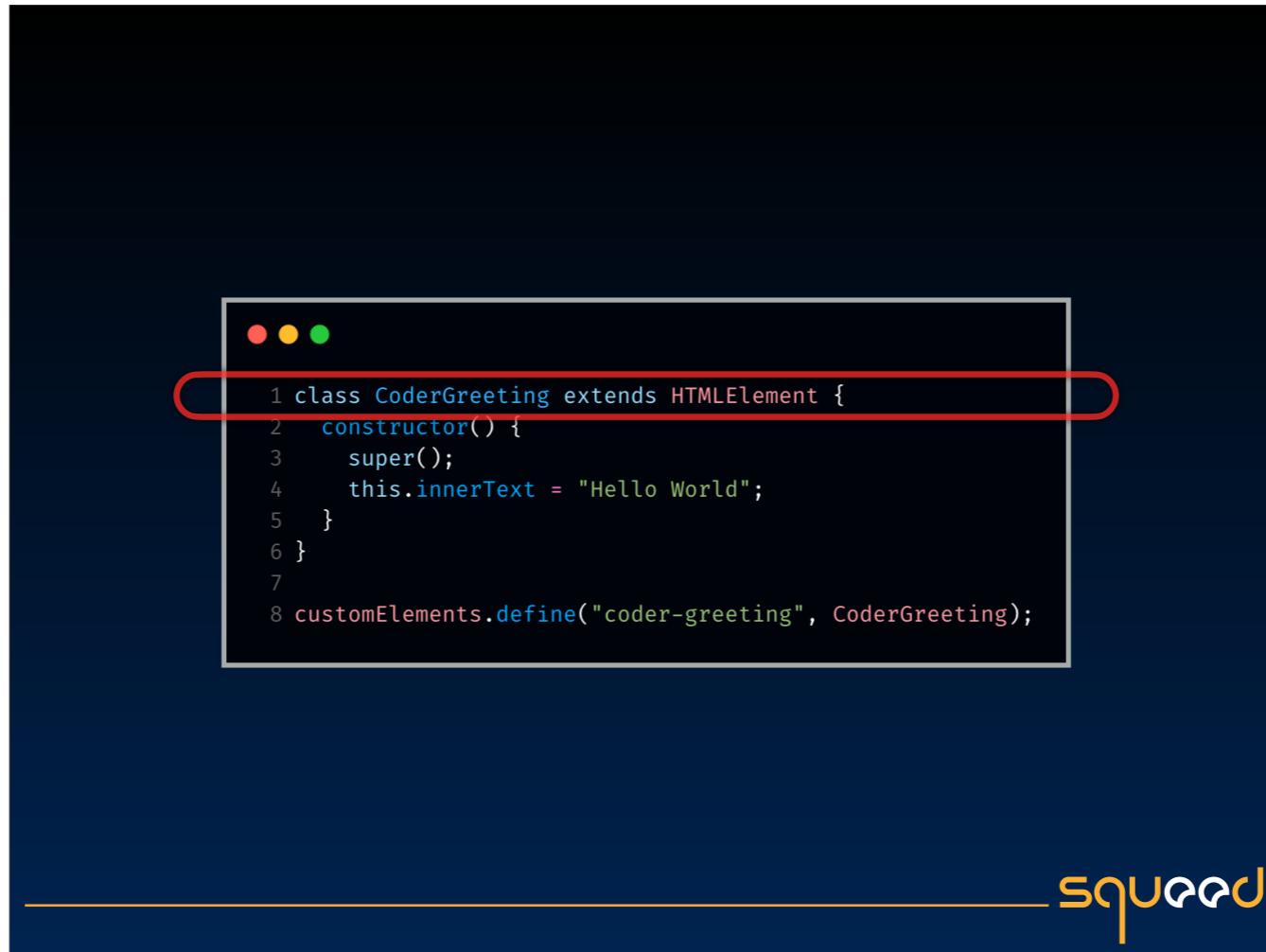


A screenshot of a terminal window on a dark background. The window shows a block of JavaScript code. The last line of the code, which is highlighted with a red rounded rectangle, is:

```
8 customElements.define("coder-greeting", CoderGreeting);
```

squared

Creating a custom element is done with JavaScript's class syntax. You use the custom elements registry to make a new custom element, which is what is done down at the end of this script. Here we give the define-method a string that later will allow us to write a HTML tag with that name, and we give it a class, which contains all the inner workings of the element.



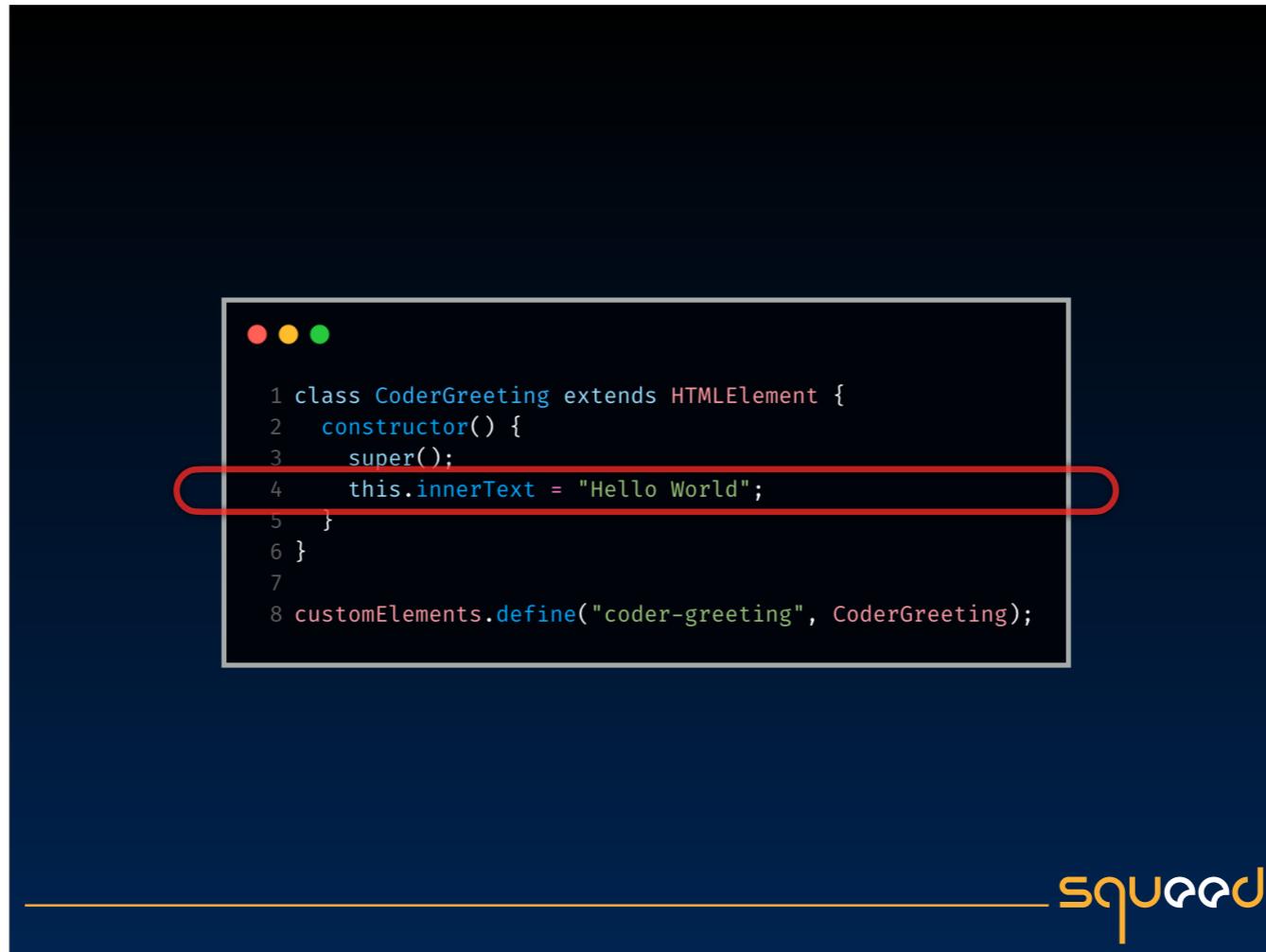
A screenshot of a terminal window on a dark background. The window title bar shows three colored dots (red, yellow, green). The terminal displays the following JavaScript code:

```
1 class CoderGreeting extends HTMLElement {  
2   constructor() {  
3     super();  
4     this.innerText = "Hello World";  
5   }  
6 }  
7  
8 customElements.define("coder-greeting", CoderGreeting);
```

The first line of code, '1 class CoderGreeting extends HTMLElement {', is highlighted with a red rounded rectangle.

squared

Because, it is with JavaScript's class syntax that we specify what our custom element shall do. We create a class that first of all extends the built in class HTMLElement.



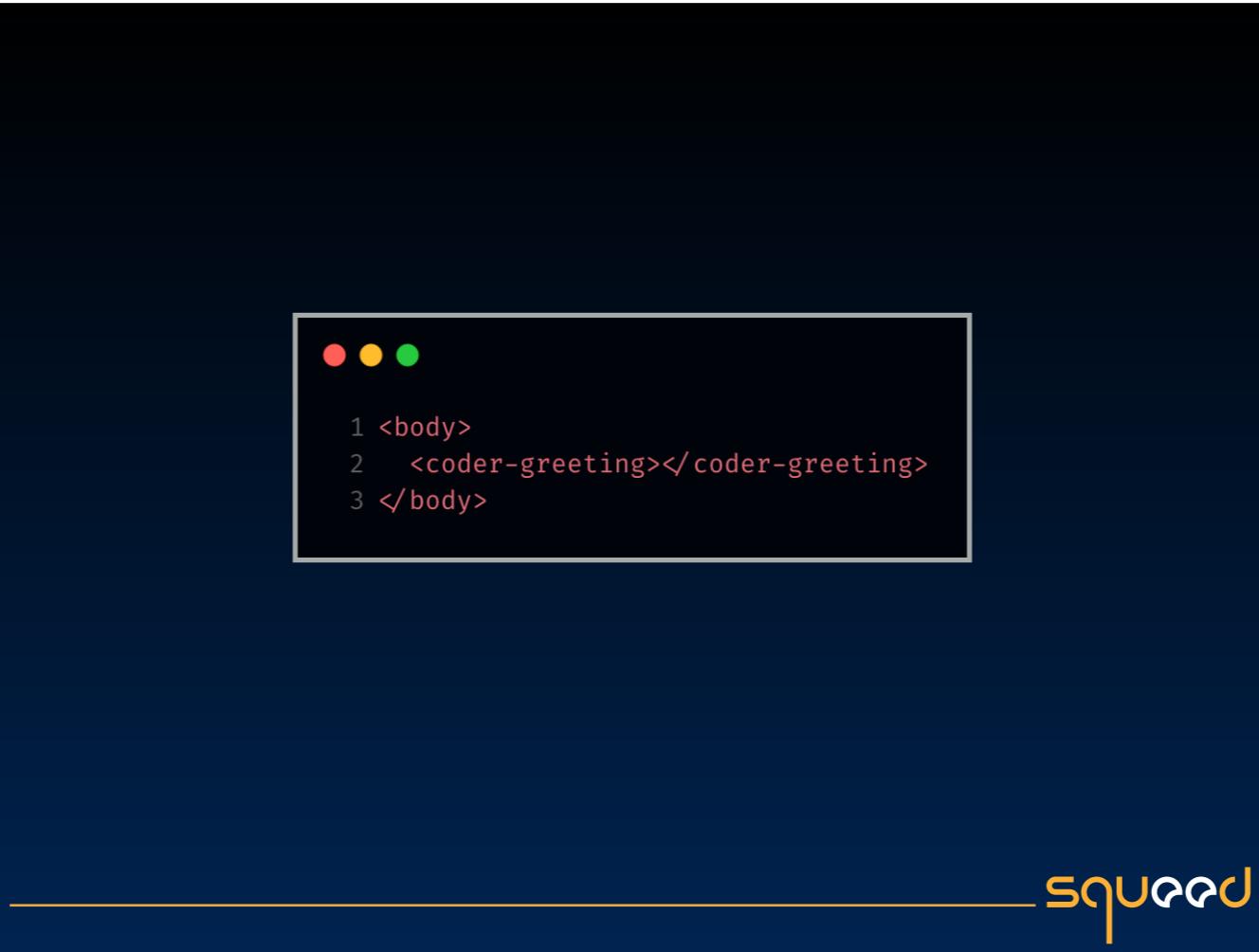
A screenshot of a terminal window on a dark background. The window title bar shows three colored dots (red, yellow, green). The terminal displays the following code:

```
1 class CoderGreeting extends HTMLElement {  
2   constructor() {  
3     super();  
4     this.innerText = "Hello World";  
5   }  
6 }  
7  
8 customElements.define("coder-greeting", CoderGreeting);
```

The line `this.innerText = "Hello World";` is highlighted with a red oval-shaped selection.

squared

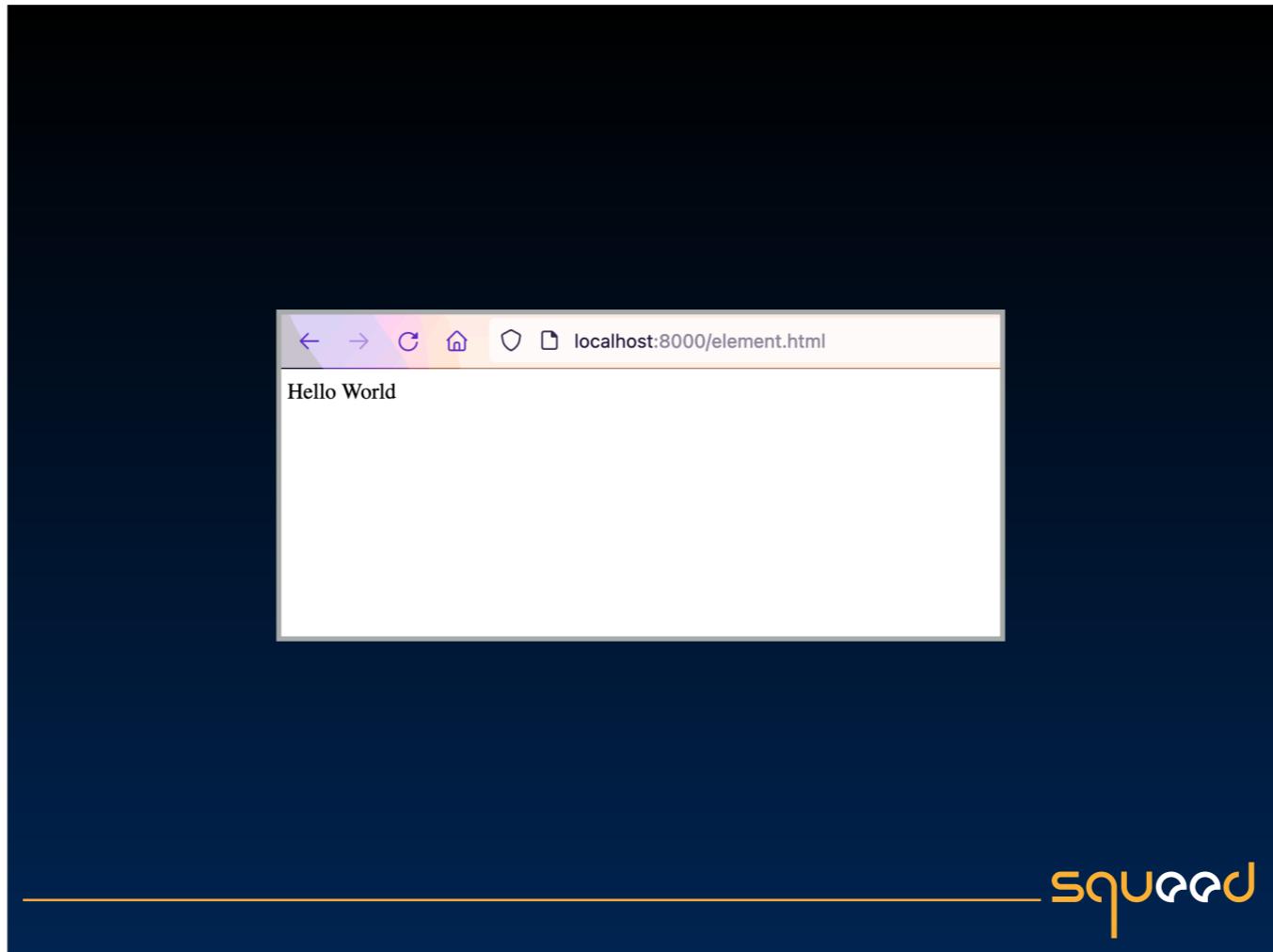
Then, in the constructor - after calling the constructor of the class we extend, which we always must do - we set our custom elements inner text to "Hello world".



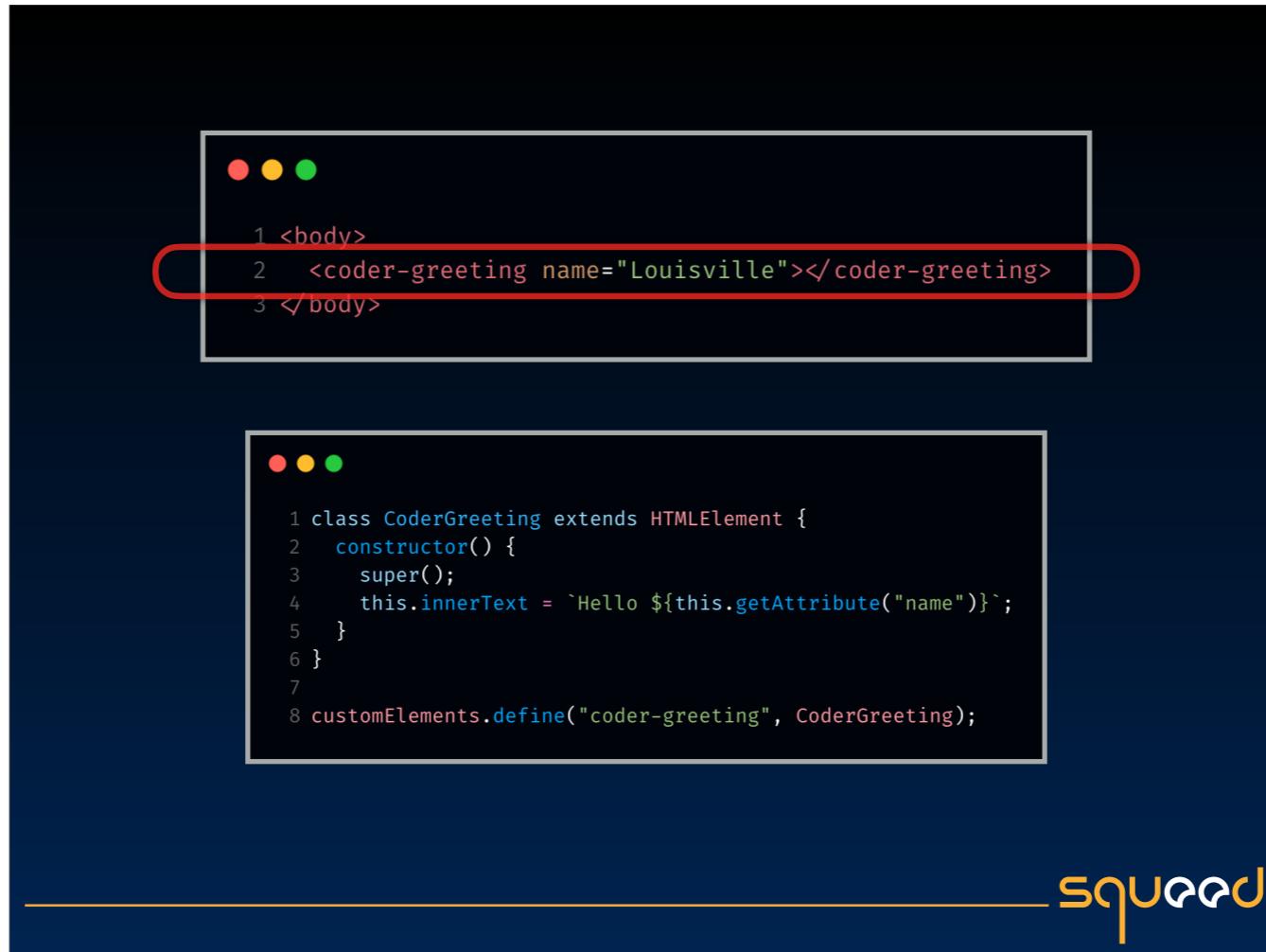
```
● ● ●  
1 <body>  
2   <coder-greeting></coder-greeting>  
3 </body>
```

sqwqwd

And now, we can write a "coder-greeting"-tag in HTML...



And we automatically get a greeting on the screen. That's how simple it can be to create your very own custom element.

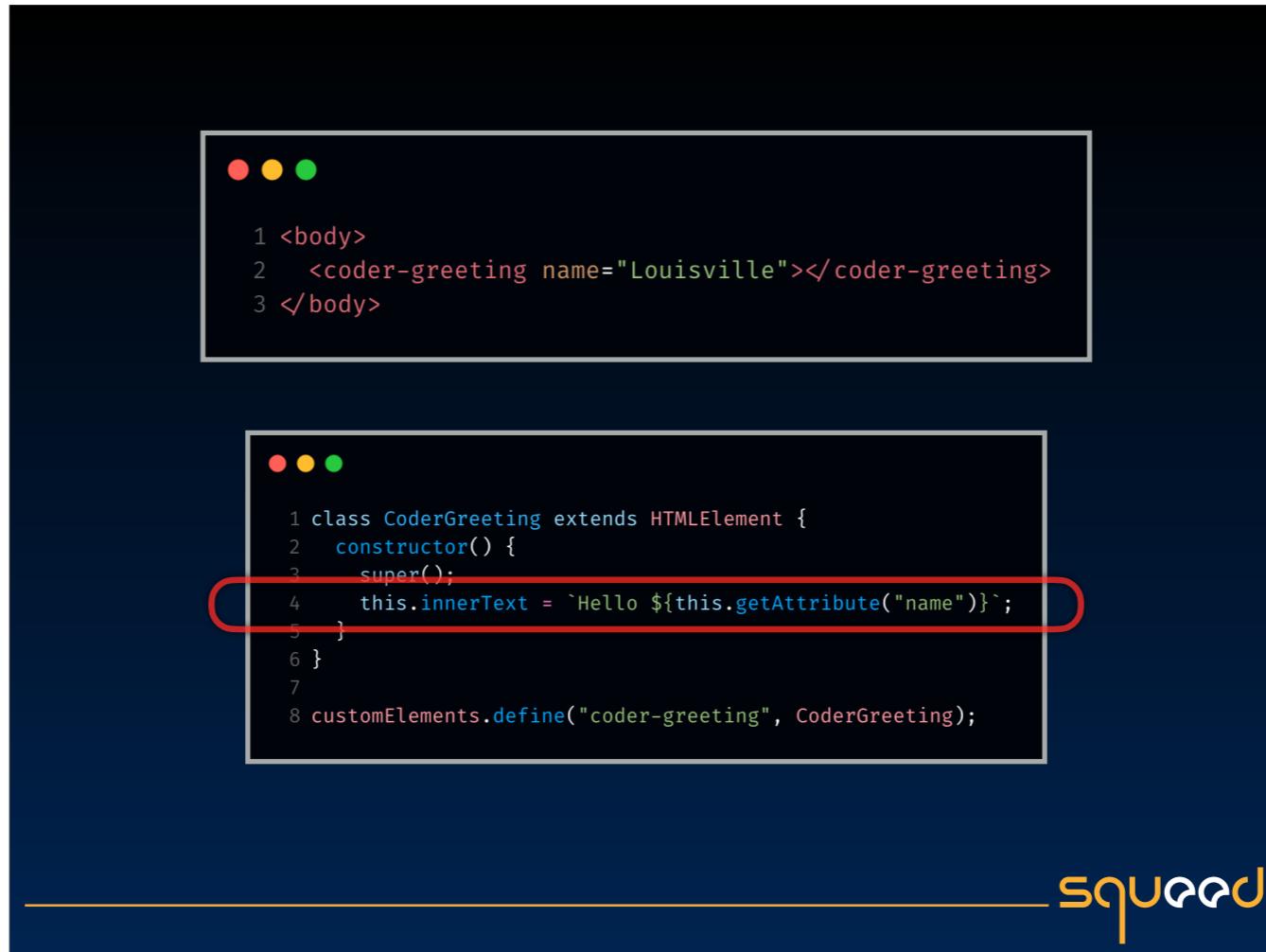


```
1 <body>
2   <coder-greeting name="Louisville"></coder-greeting>
3 </body>
```

```
1 class CoderGreeting extends HTMLElement {
2   constructor() {
3     super();
4     this.innerText = `Hello ${this.getAttribute("name")}`;
5   }
6 }
7
8 customElements.define("coder-greeting", CoderGreeting);
```

squared

Let's also take a quick look at attributes. Any attribute put on the tag in the HTML can be retrieved in the code with the `getAttribute` method, and used in the element. If we put the attribute "name" on the HTML tag...

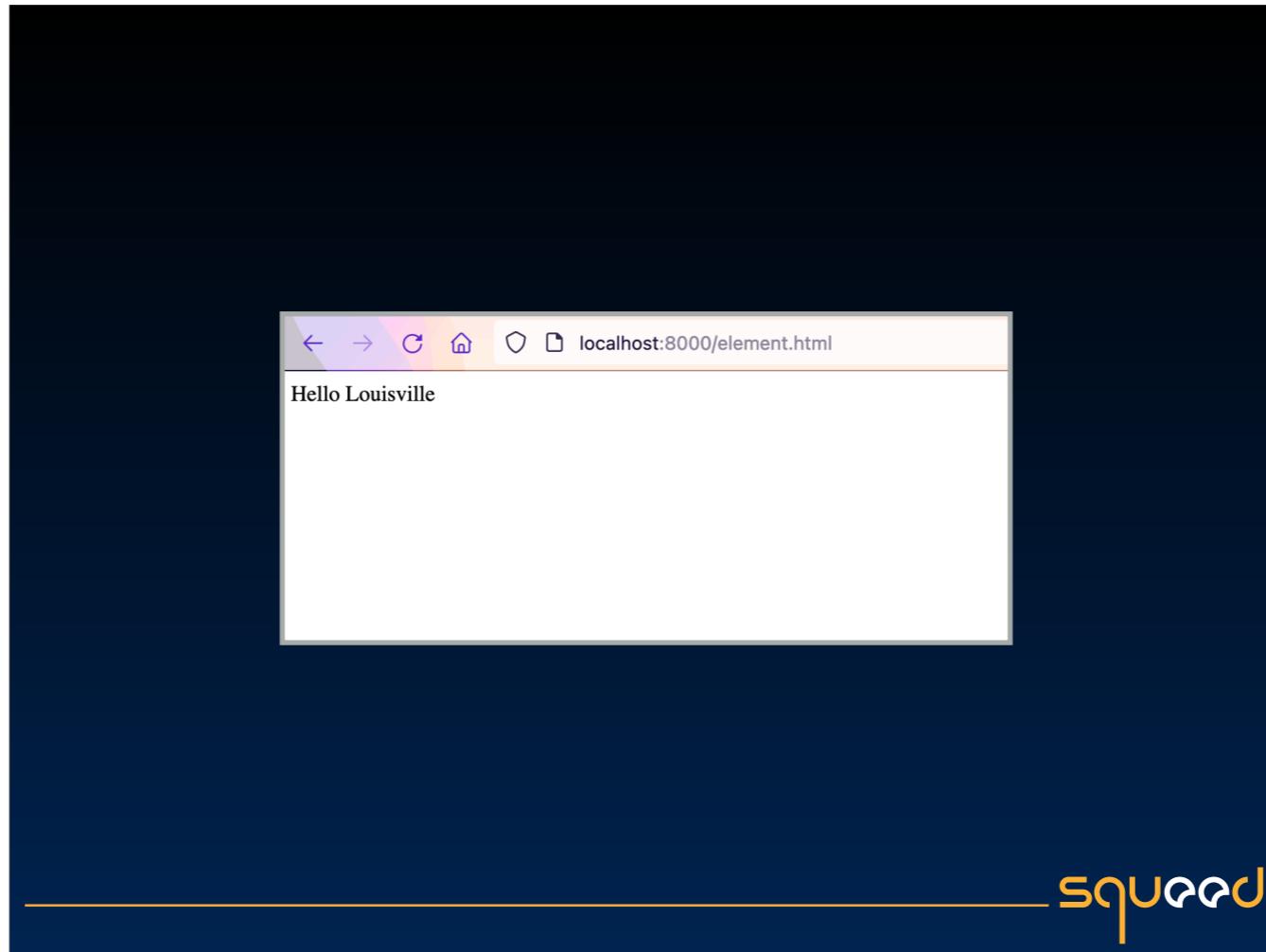


```
1 <body>
2   <coder-greeting name="Louisville"></coder-greeting>
3 </body>
```

```
1 class CoderGreeting extends HTMLElement {
2   constructor() {
3     super();
4     this.innerText = `Hello ${this.getAttribute("name")}`;
5   }
6 }
7
8 customElements.define("coder-greeting", CoderGreeting);
```

squared

...we can then retrieve it in the component class and make it part of the elements inner text.



squoqd

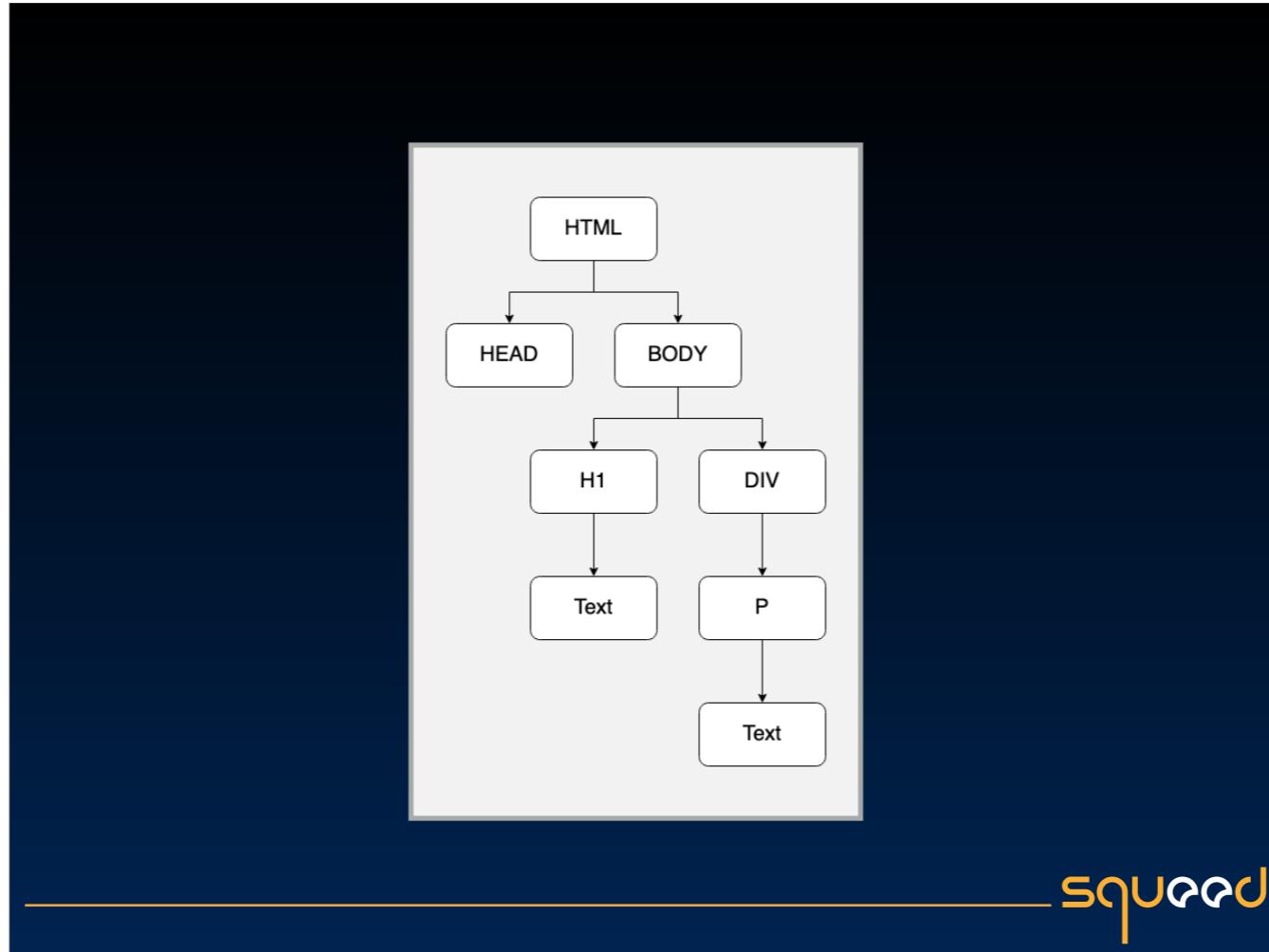
And so we get Hello Louisville!

There is tons more you can do, and we'll get more into that in a minute, but just remember that the API for a custom element is exactly like that of a regular element. GetAttribute, setAttribute, and so on, it's all there.

# The shadow DOM

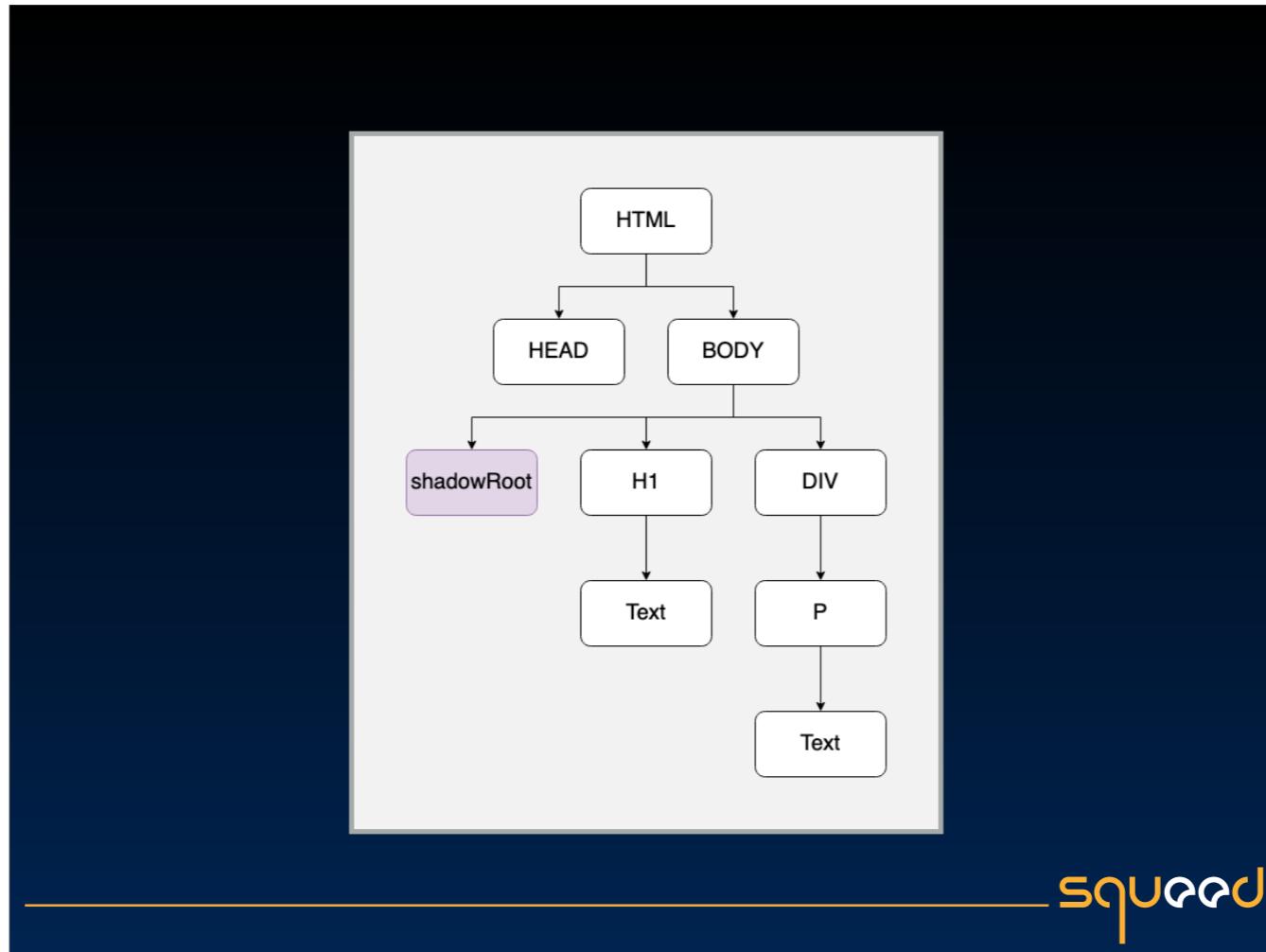


Finally, the third api that powers web components is the shadow dom. Now, I don't know if it's the name of it or what, but I've gotten the impression that this is the most difficult, and the scariest of the concepts, and the hardest one to grasp. But I don't think it's very complicated really. Let's take a look!



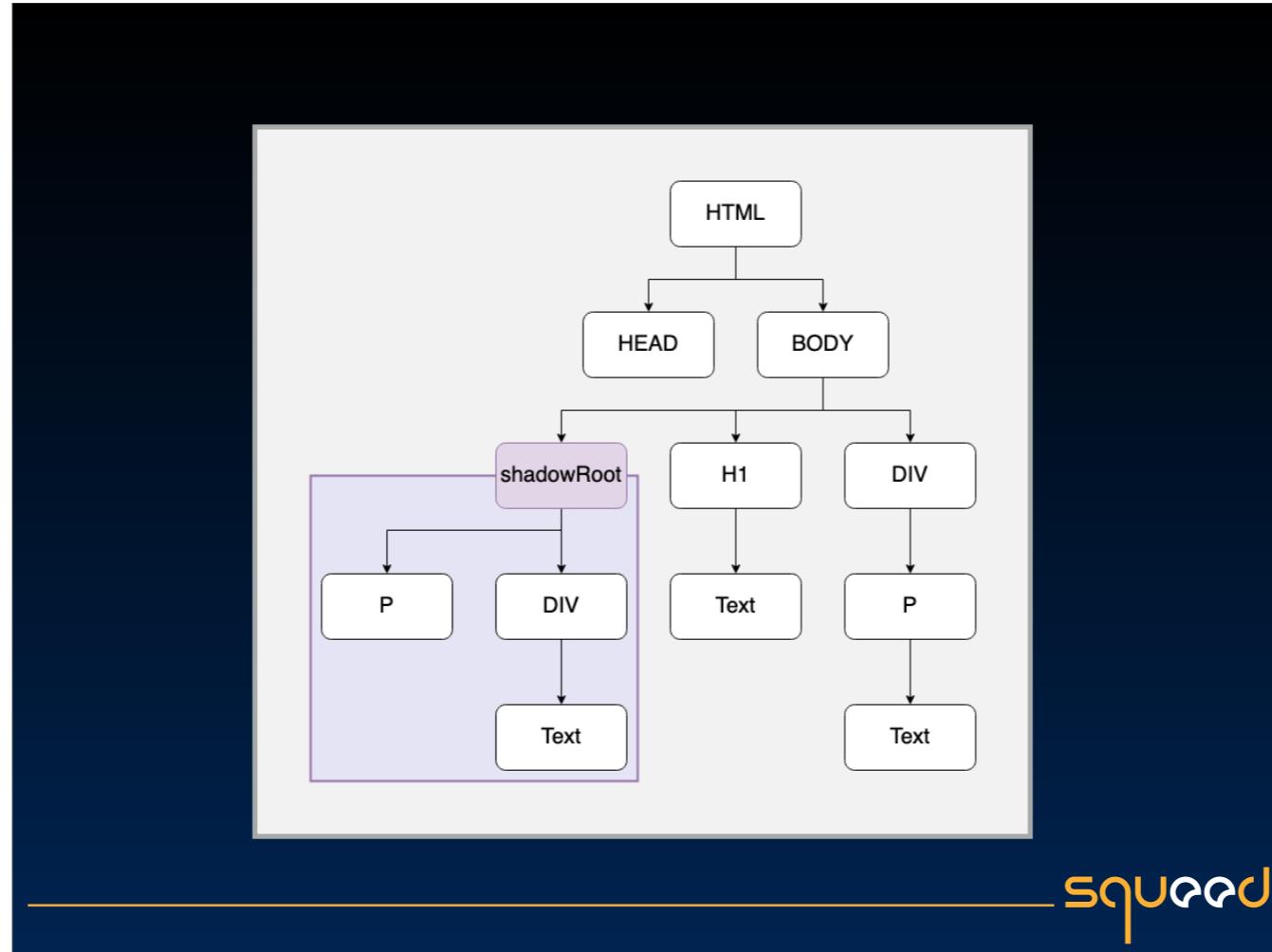
squqrd

So here we have a very small document tree, with an html as the root of the tree, and some html elements nested below. Nothing strange about this, all is normal. Now, you know that we can use JavaScript to attach new nodes to this tree, that's nothing new. But what we can do is we can take one of these elements and attach a so called shadow root to it.



squared

Attaching a shadow root to the body element would make our tree look like this. Now this doesn't affect the tree in any way yet, but we can now attach more, regular nodes to the shadow root.



squqrd

Like this! And what we have here is actually a new, separate tree. Just like the regular tree has the HTML node as its root, this new tree has the shadowRoot node as its root. And what the shadow root does is that it acts like a kind of gatekeeper between the shadow world and the light world. Let's say we insert a style tag somewhere in the light tree, to apply some styling to our page. That styling does not penetrate the shadowRoot. Nothing within the shadow world can be targeted by that CSS. The same goes for styles defined inside the shadow world - nothing in the light world will be able to be targeted. You also can't query for elements over the border. If you use `document.querySelector` you will not find anything on the other side of the shadowRoot. You'll have to find the shadow root itself and then use `shadowRoot.querySelector` in order to find those elements.

There are ways to pierce the shadow veil and have things like styling come through, but we'll talk more about that in a little while. For now, just know that the shadow DOM is a separate document tree that encapsulates its behaviour and does not affect the rest of the document.

# HTML Templates

## Shadow DOM

## Custom Elements

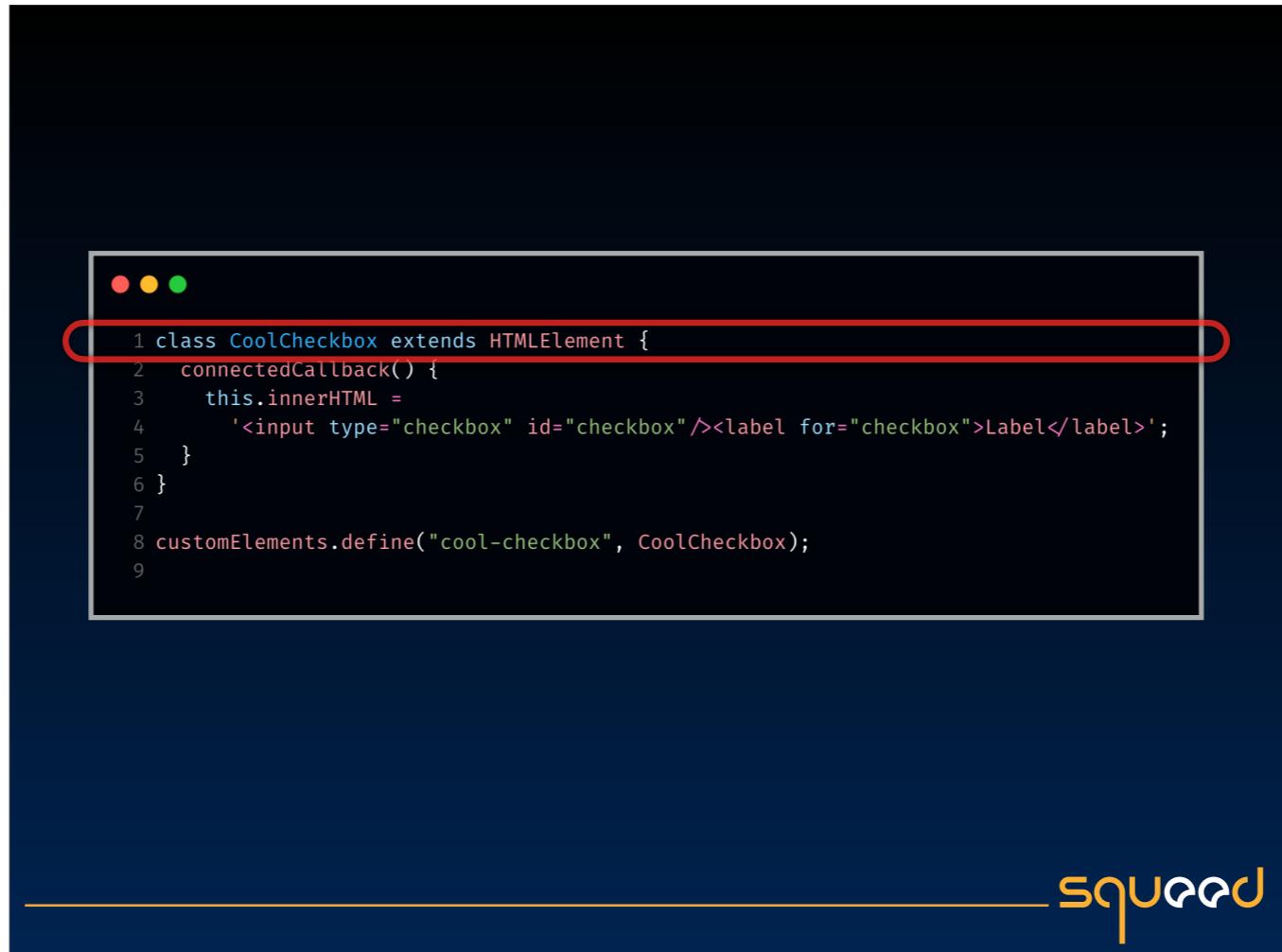


Alright, so now we've looked at HTML templates, custom elements, and the shadow dom. With our powers combined! We now have the ability to...

# Let's code a web component!



...code a web component!



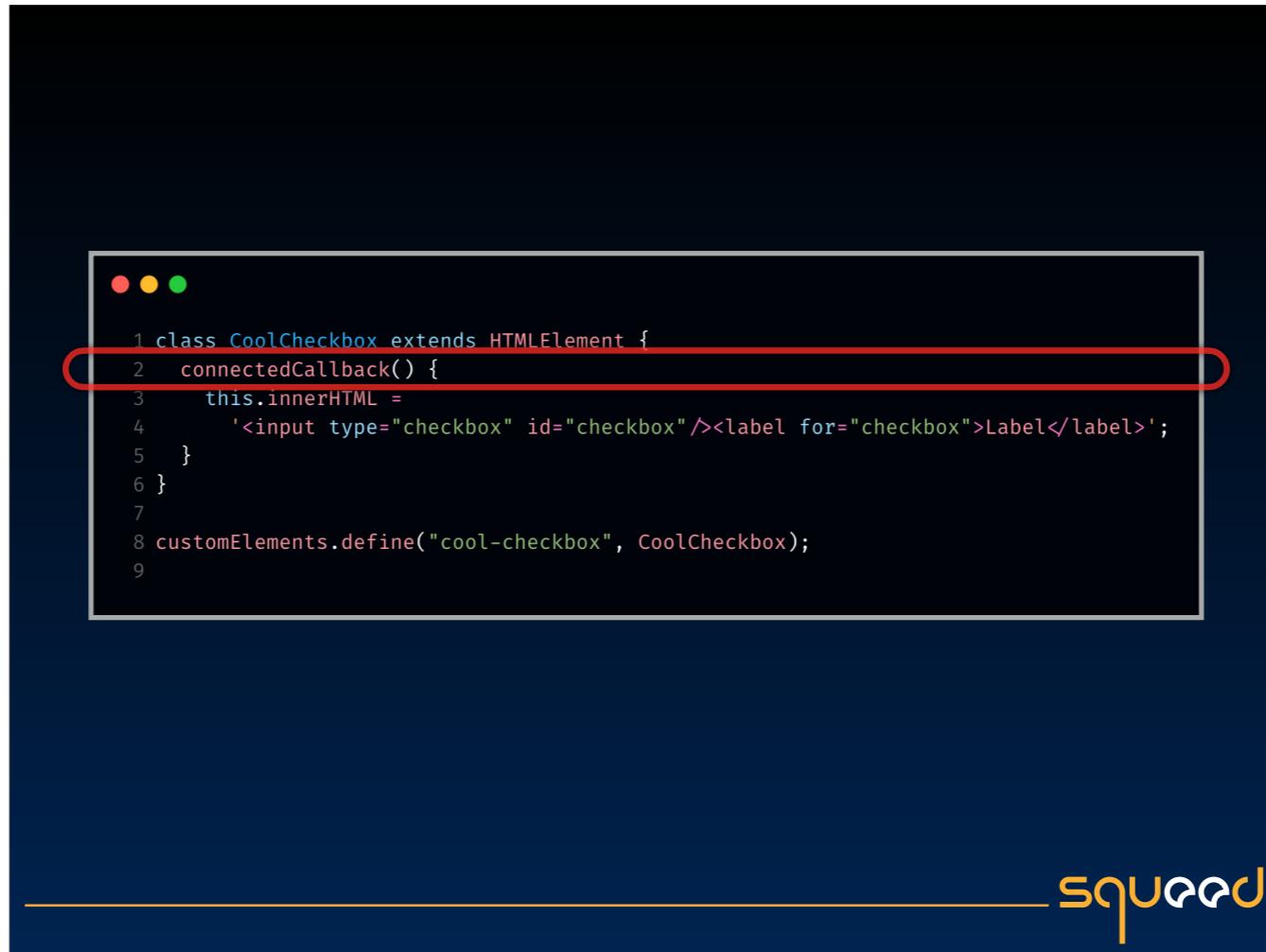
A screenshot of a code editor window with a dark theme. At the top, there are three colored circular icons (red, yellow, green). Below them is a red rectangular selection highlighting the first line of code. The code itself is written in JavaScript and defines a custom element named 'cool-checkbox'. It extends the 'HTMLElement' class and overrides the 'connectedCallback' method to set the element's innerHTML to an HTML string containing an input checkbox and a label. Finally, it uses 'customElements.define' to register the new element. The code editor has a light gray background and a white border around the code area.

```
1 class CoolCheckbox extends HTMLElement {  
2     connectedCallback() {  
3         this.innerHTML =  
4             '<input type="checkbox" id="checkbox" /><label for="checkbox">Label</label>';  
5     }  
6 }  
7  
8 customElements.define("cool-checkbox", CoolCheckbox);  
9
```

squared

The component that we'll be coding is a custom checkbox. This is something you often might find in component libraries, since native checkboxes are notoriously hard to style, and you can get a lot of different results in different browsers.

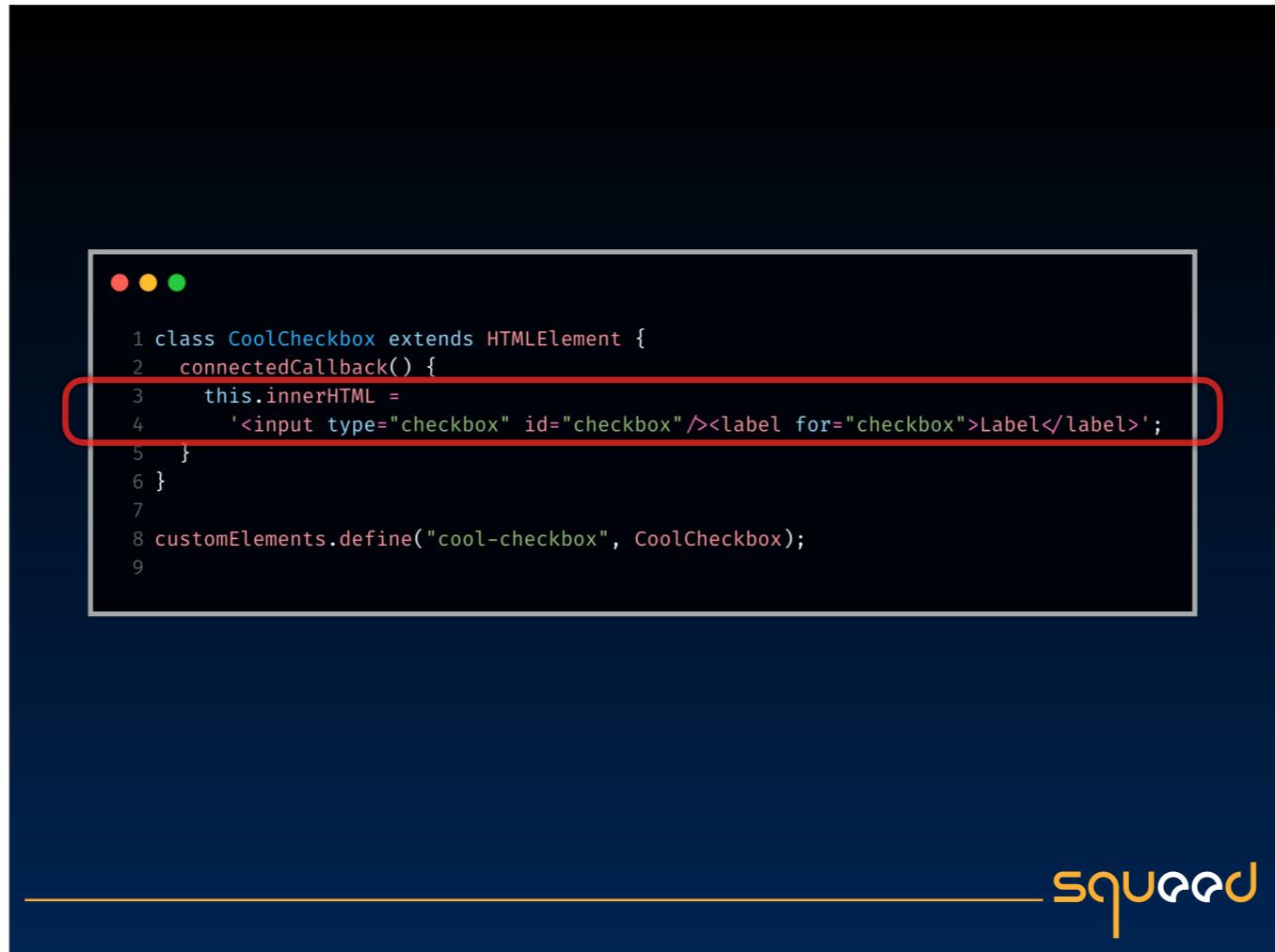
So! The first thing we'll do is create our custom element. We'll call it CoolCheckbox, because obviously it's going to be really cool, and we'll extend HTMLElement, just like we're supposed to.



```
1 class CoolCheckbox extends HTMLElement {  
2     connectedCallback() {  
3         this.innerHTML =  
4             '<input type="checkbox" id="checkbox" /><label for="checkbox">Label</label>';  
5     }  
6 }  
7  
8 customElements.define("cool-checkbox", CoolCheckbox);  
9
```

The screenshot shows a code editor window with a dark theme. At the top left are three colored dots (red, yellow, green). The code itself is written in JavaScript and defines a custom element named 'cool-checkbox'. It uses the standard template literal syntax to define the element's content. The 'connectedCallback' method is highlighted with a red oval. The Squeeb logo is visible at the bottom right of the slide.

Next we define a function called `connectedCallback`. You might remember before, we specified the components content in the constructor. Now, a problem with doing that is that it won't work if the element is appended to the DOM before it has been defined. This might be something you should avoid anyway, but just to be safe it is best to do all of the initial DOM manipulation stuff in the `connectedCallback` function, because this function runs right after the element has been both defined and then placed in the DOM, so it's the ideal place for setting up the components initial state.



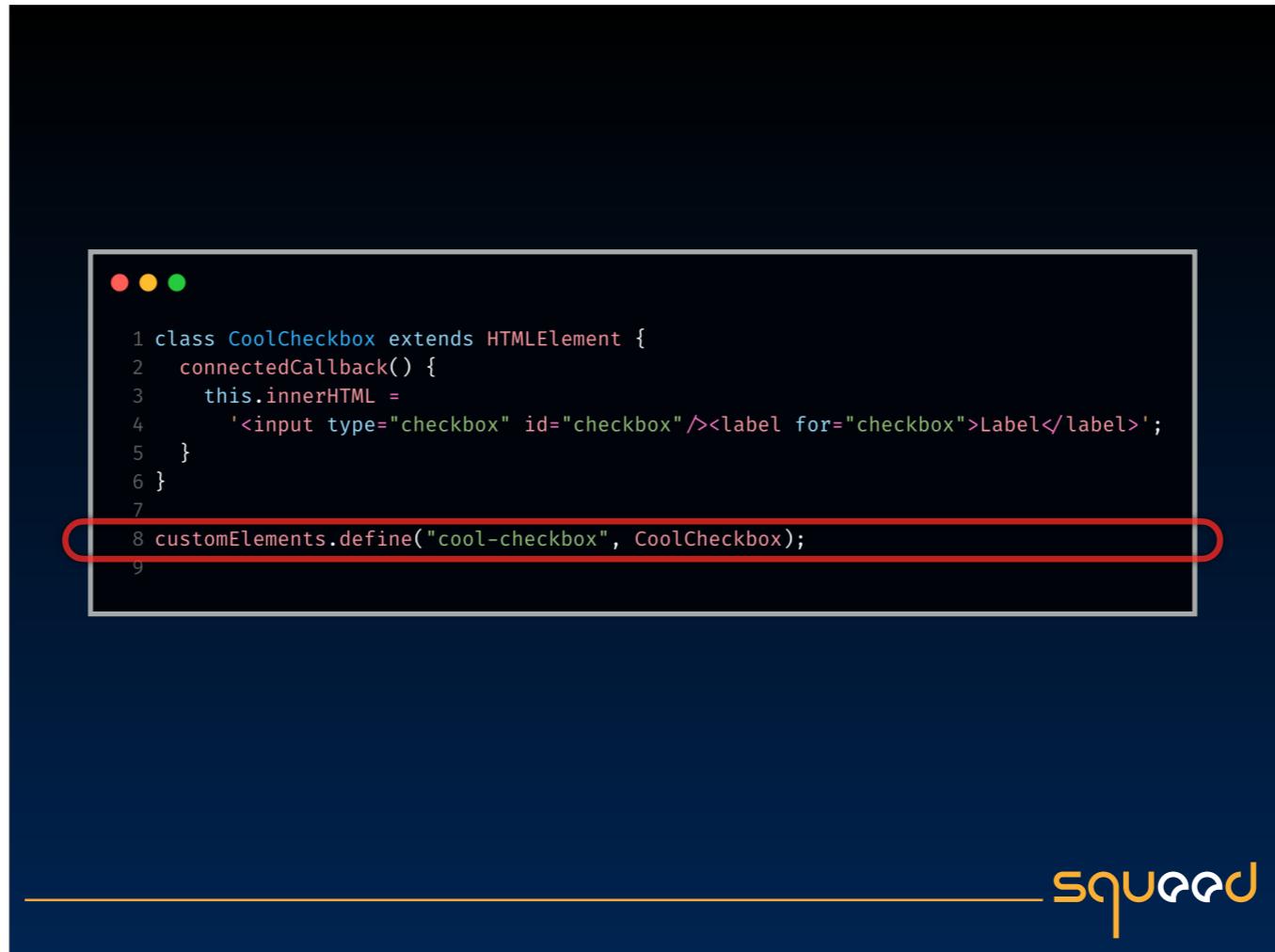
A screenshot of a code editor window with a dark theme. At the top left, there are three colored dots (red, yellow, green). The main area contains the following code:

```
1 class CoolCheckbox extends HTMLElement {  
2     connectedCallback() {  
3         this.innerHTML =  
4             '<input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>';  
5     }  
6 }  
7  
8 customElements.define("cool-checkbox", CoolCheckbox);  
9
```

The line of code `this.innerHTML =` is highlighted with a red rectangular box.

squared

So, in this function, we set the elements innerHTML to be an input of type checkbox, and a corresponding label.



A screenshot of a code editor window with a dark theme. At the top left, there are three colored dots (red, yellow, green). The code editor contains the following JavaScript code:

```
1 class CoolCheckbox extends HTMLElement {  
2   connectedCallback() {  
3     this.innerHTML =  
4       '<input type="checkbox" id="checkbox" /><label for="checkbox">Label</label>';  
5   }  
6 }  
7  
8 customElements.define("cool-checkbox", CoolCheckbox);  
9
```

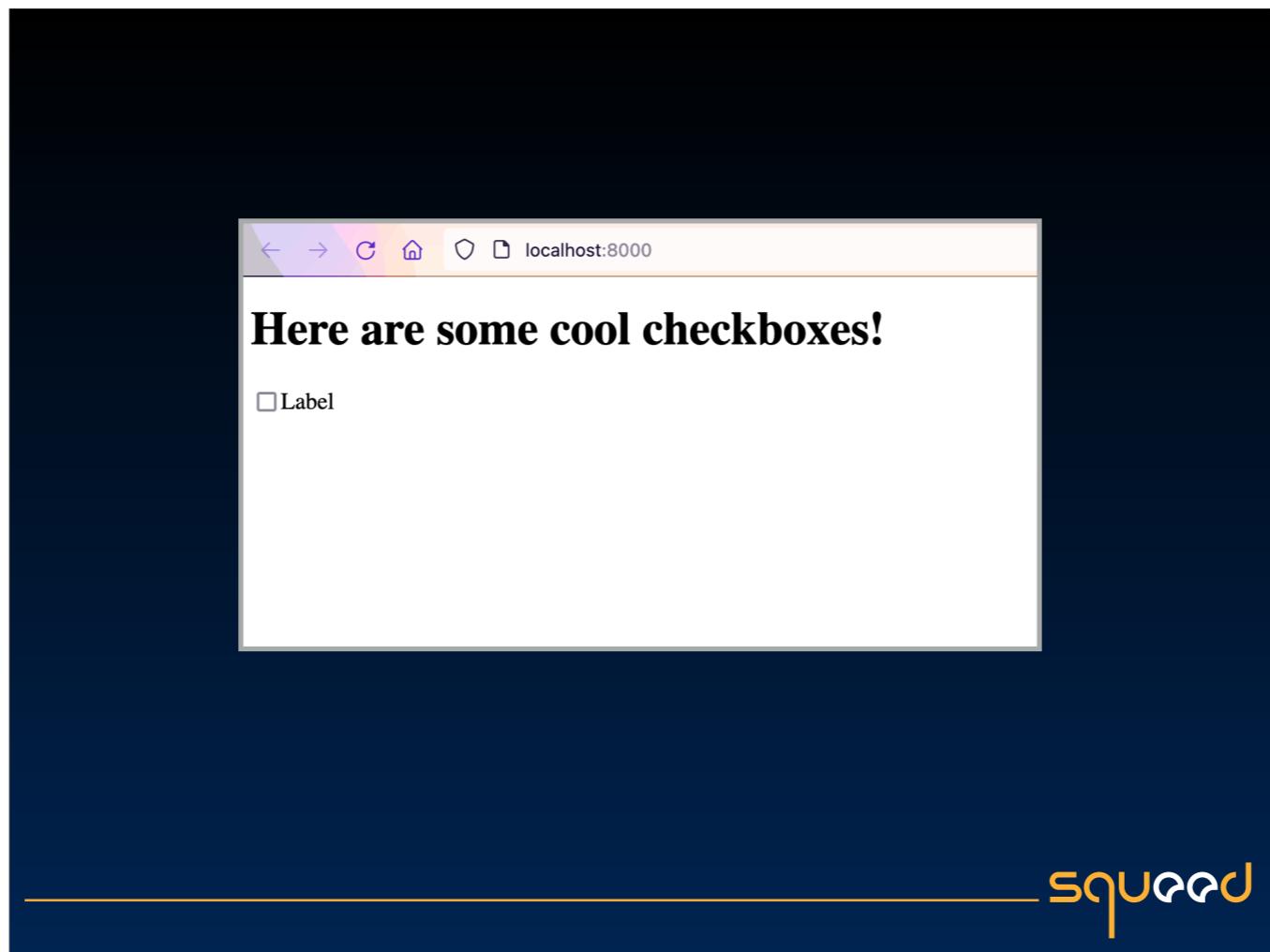
The line `8 customElements.define("cool-checkbox", CoolCheckbox);` is highlighted with a red oval.

At the bottom right of the editor window, there is a logo consisting of the word "squared" in a stylized orange font.

Finally, we define our new custom element in the registry, and it's ready to use!



We can now put it in our HTML file like so...



And we get a perfectly ordinary checkbox! Awesome. Next...

# Use a template



...let's use a HTML template for the content of our component.



```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
4 `;
5
6 class CoolCheckbox extends HTMLElement {
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.appendChild(templateContent);
10  }
11 }
```

squared

Before, we defined a template declaratively in the HTML file...



A screenshot of a terminal window with a dark background and light-colored text. The window has three colored dots (red, yellow, green) in the top-left corner. Inside the window, there is a code editor displaying the following JavaScript code:

```
1 <template>
2   <input type="checkbox" id="checkbox" /><label for="checkbox">Label</label>
3 </template>
4 ;
5
6 class CoolCheckbox extends HTMLElement {
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.appendChild(templateContent);
10  }
11 }
```

squared

...like this, you remember? But we can just as easily do it imperatively in JavaScript.



```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
4 ;
5
6 class CoolCheckbox extends HTMLElement {
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.appendChild(templateContent);
10  }
11 }
```

The screenshot shows a terminal window with a dark background and light-colored text. It displays a snippet of JavaScript code. A red box highlights the line `template.innerHTML =` and the subsequent multi-line string containing an input and a label element. Below the terminal is a blue footer bar with the word "SQUARED" in white.

So that's what we do here, we use `document.createElement` to create a template element, and then we set its inner HTML.



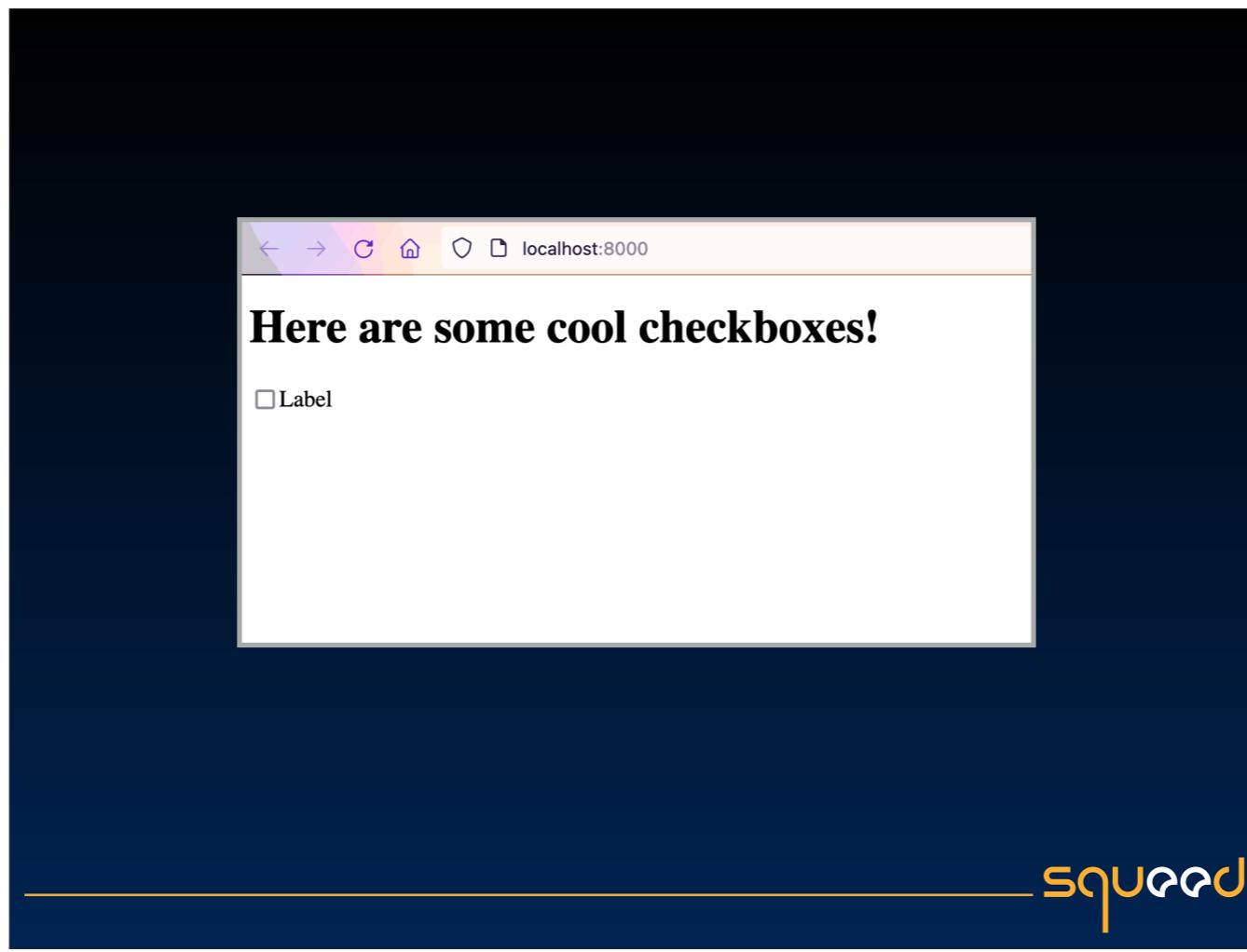
```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
4 `;
5
6 class CoolCheckbox extends HTMLElement {
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.appendChild(templateContent);
10  }
11 }
```

The image shows a terminal window with a dark background. Inside, there is a code editor window displaying a snippet of JavaScript. A red rectangular box highlights the line of code `const templateContent = document.importNode(template.content, true);`. The code itself is as follows:

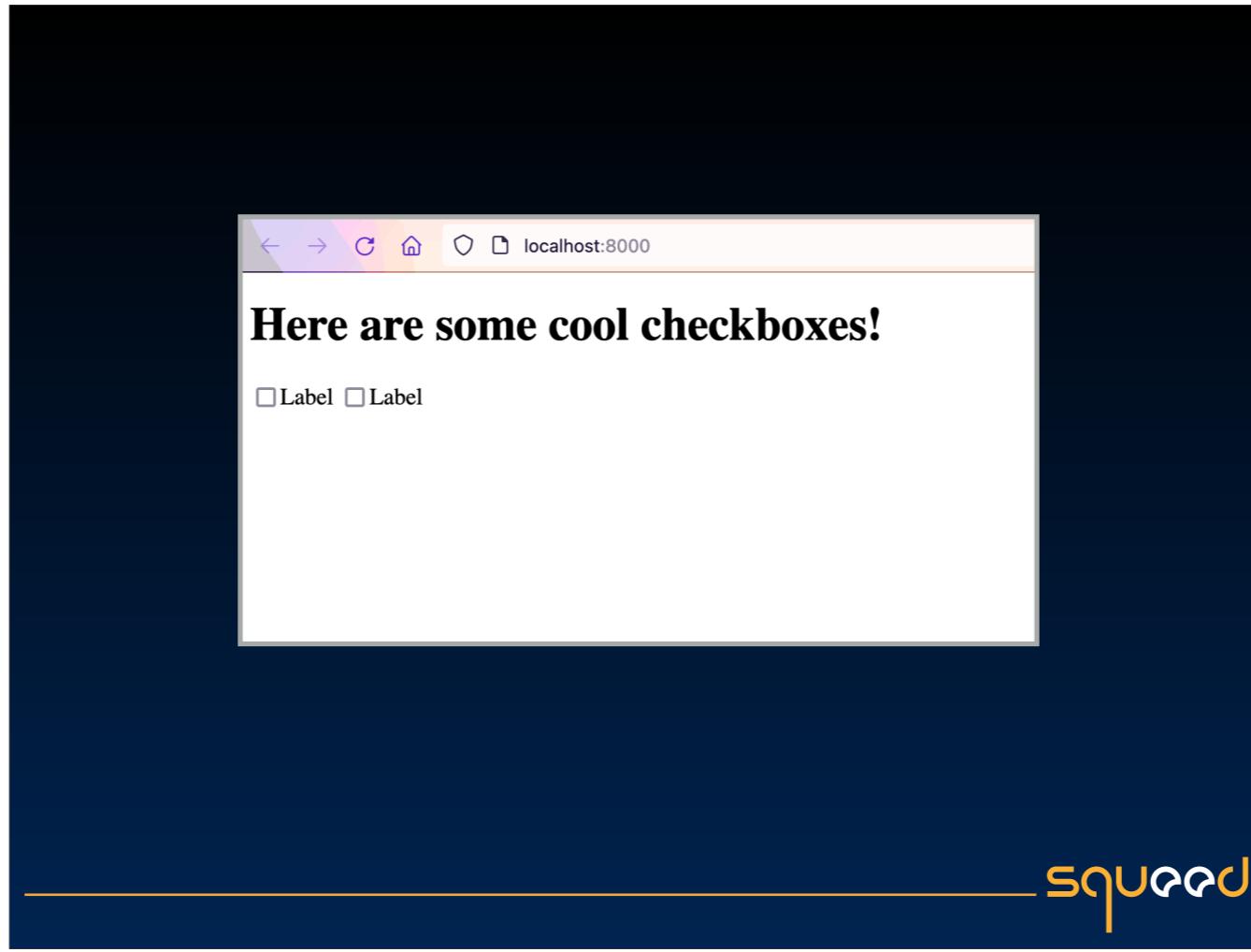
```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
4 `;
5
6 class CoolCheckbox extends HTMLElement {
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.appendChild(templateContent);
10  }
11 }
```

squared

Then, down in the connectedCallback of our custom element we clone the template content using document.importNode and append it to the custom element.



And voila! It looks exactly the same as before.



However, the advantage is that if we put multiple instances of our component on a page, the template will only have to be parsed once, and then it can be reused via cloning, so that the more copies you have on a page, and the more complex they are, the bigger the performance gain.

# Enter the **shadow** **DOM**

---

squared

Now, this was going to be a cool checkbox, and since there's nothing cooler than being dark and mysterious, let's add a shadow DOM!



A screenshot of a code editor showing a file named 'coolcheckbox.js'. The code defines a custom element 'CoolCheckbox' that extends 'HTMLElement'. In the constructor, it calls 'super()' and then 'this.attachShadow({ mode: "open" })'. The 'attachShadow' line is highlighted with a red oval. The code also includes a 'connectedCallback' function that imports a template and appends its content to the shadow root.

```
1 class CoolCheckbox extends HTMLElement {
2   constructor() {
3     super();
4     this.attachShadow({ mode: "open" });
5   }
6
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.shadowRoot.appendChild(templateContent);
10  }
11 }
```

squared

This is done in the constructor, and it's no harder than taking our element and calling `attachShadow` on it. There! Now we have a `shadowRoot` on our custom element. We also pass an option object that contains "mode: open". This is so that the shadow root will allow us to access it with JavaScript from the outside. You will probably always set it to open when you create your shadow roots, because it is by accessing it with JS that we can manipulate its content.

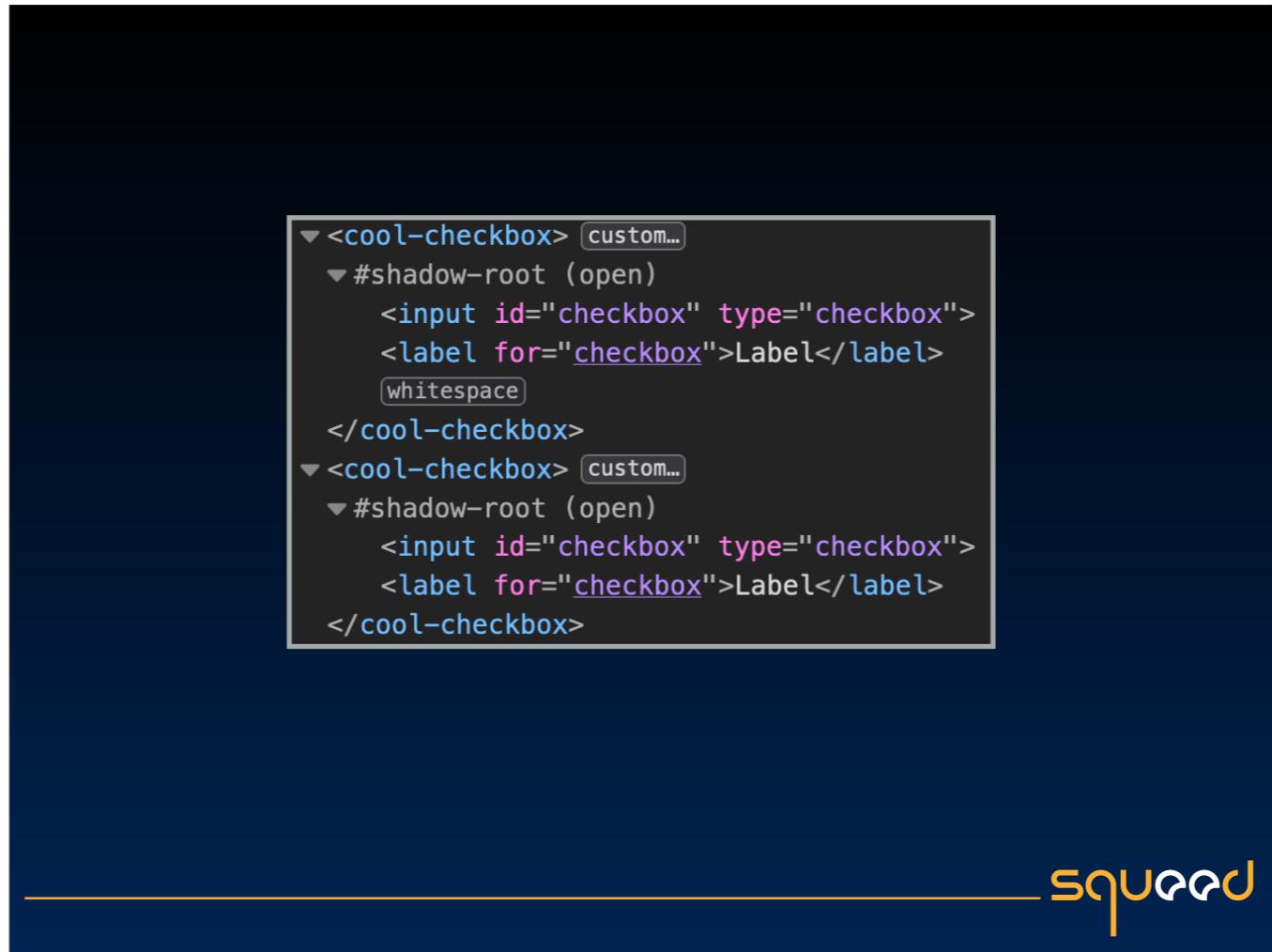


```
1 class CoolCheckbox extends HTMLElement {
2   constructor() {
3     super();
4     this.attachShadow({ mode: "open" });
5   }
6
7   connectedCallback() {
8     const templateContent = document.importNode(template.content, true);
9     this.shadowRoot.appendChild(templateContent);
10  }
11 }
```

The screenshot shows a code editor window with a dark theme. It displays a JavaScript file named 'CoolCheckbox.js'. The code defines a class 'CoolCheckbox' that extends 'HTMLElement'. It includes a constructor that calls 'super()' and attaches an open shadow DOM. The 'connectedCallback' method imports the content of a template element using 'document.importNode(template.content, true)' and appends it to the shadow root. The line 'this.shadowRoot.appendChild(templateContent);' is circled in red.

squared

Something that we immediately do in the connectedCallback function. Here, instead of appending the cloned template content to the element itself (as before), we append it to its shadow root instead. And now, we have a shadow DOM. All of the HTML content from the template that was appended to the shadow root is now living in its own little shadow world, just like we saw in the diagram earlier!

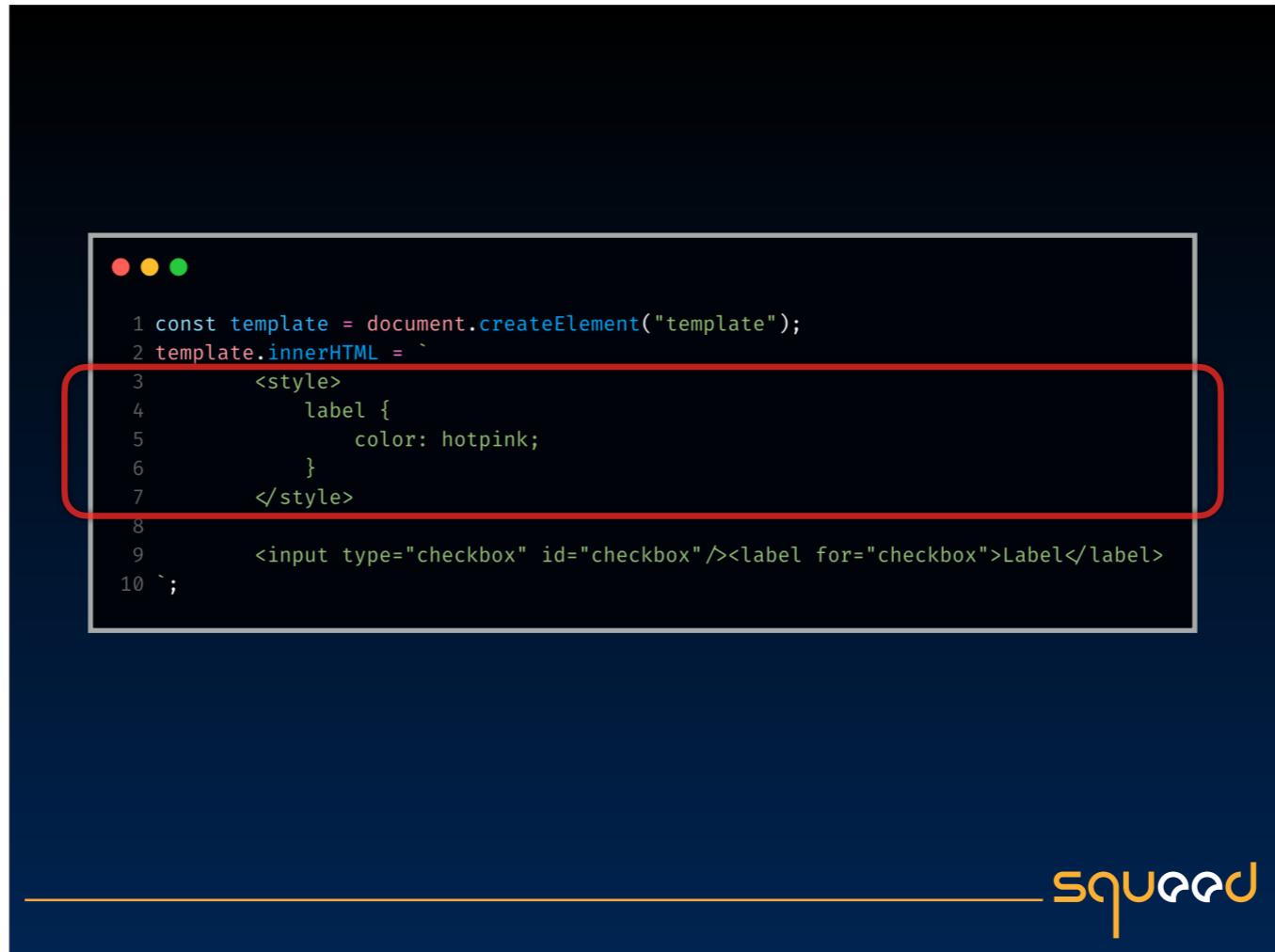


The screenshot shows the browser's developer tools with the element inspector open. It displays a tree structure of HTML elements. Two nodes are highlighted with a yellow border: both are custom elements named <cool-checkbox>. Each of these custom elements has a corresponding shadow root indicated by a small triangle icon next to the element name. The shadow roots contain standard HTML code for an input checkbox and a label. A horizontal line with a yellow arrow points from the left towards the SQUARED logo at the bottom right of the screen.

```
▼ <cool-checkbox> [custom...]
  ▼ #shadow-root (open)
    <input id="checkbox" type="checkbox">
    <label for="checkbox">Label</label>
    whitespace
  </cool-checkbox>
▼ <cool-checkbox> [custom...]
  ▼ #shadow-root (open)
    <input id="checkbox" type="checkbox">
    <label for="checkbox">Label</label>
  </cool-checkbox>
```

And if we take a look at the page structure in the browser dev tools (this is in Firefox), we can see the shadow root on display, just like a regular node in the tree. We can expand it, and under each node is a copy of our template, living in its shadow world, having a good ol' time.

SQURED



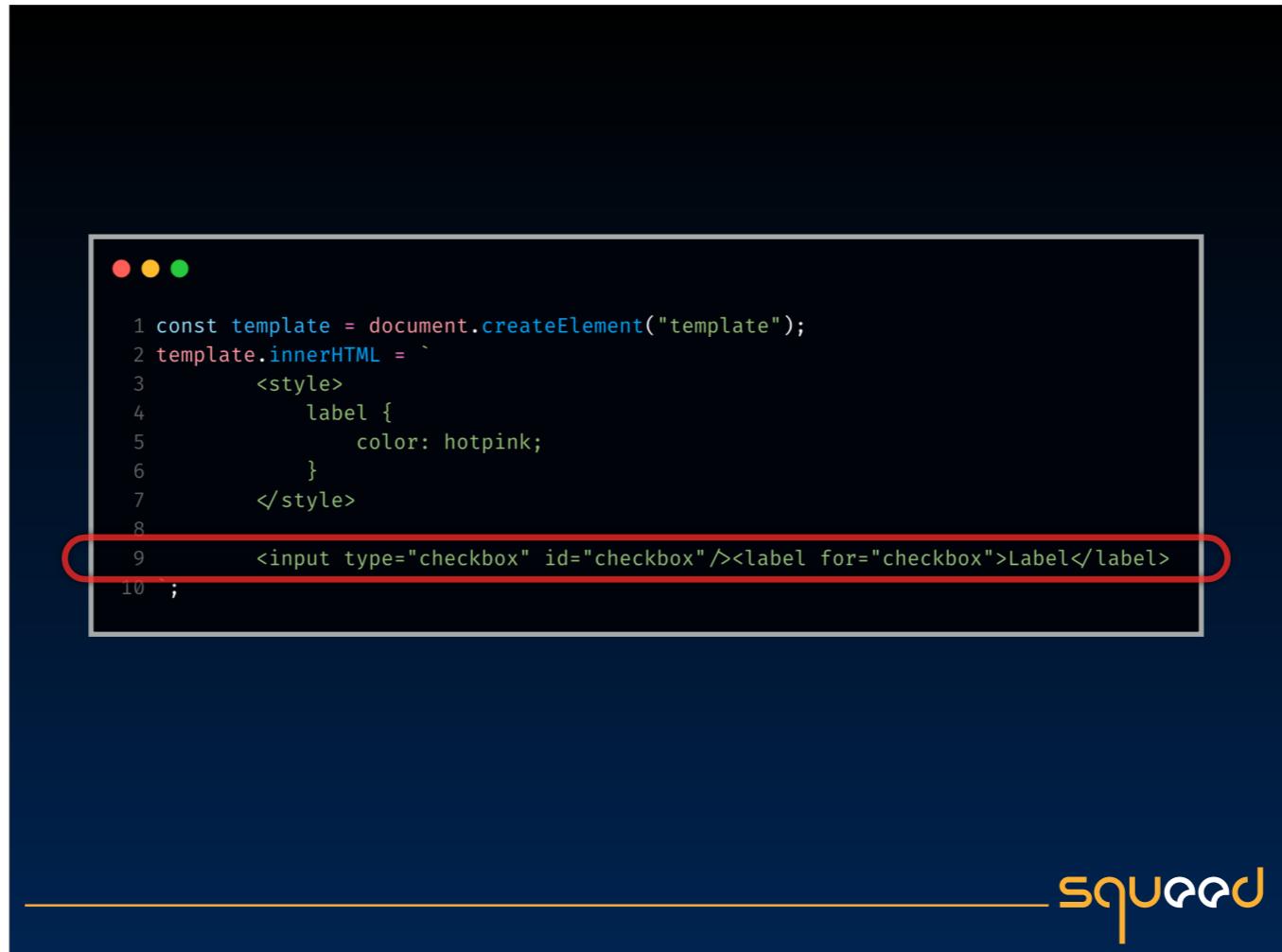
```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <style>
4     label {
5       color: hotpink;
6     }
7   </style>
8
9   <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
10
```

The screenshot shows a browser window with a dark theme. In the top right corner, there are three colored dots (red, yellow, green). Below them is a code editor window displaying the following JavaScript code:

```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <style>
4     label {
5       color: hotpink;
6     }
7   </style>
8
9   <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
10
```

A red rectangular box highlights the entire content of the innerHTML block, specifically the style and label elements. At the bottom right of the slide, there is a logo consisting of the word "SQUEE" in a stylized font where each letter has a different color.

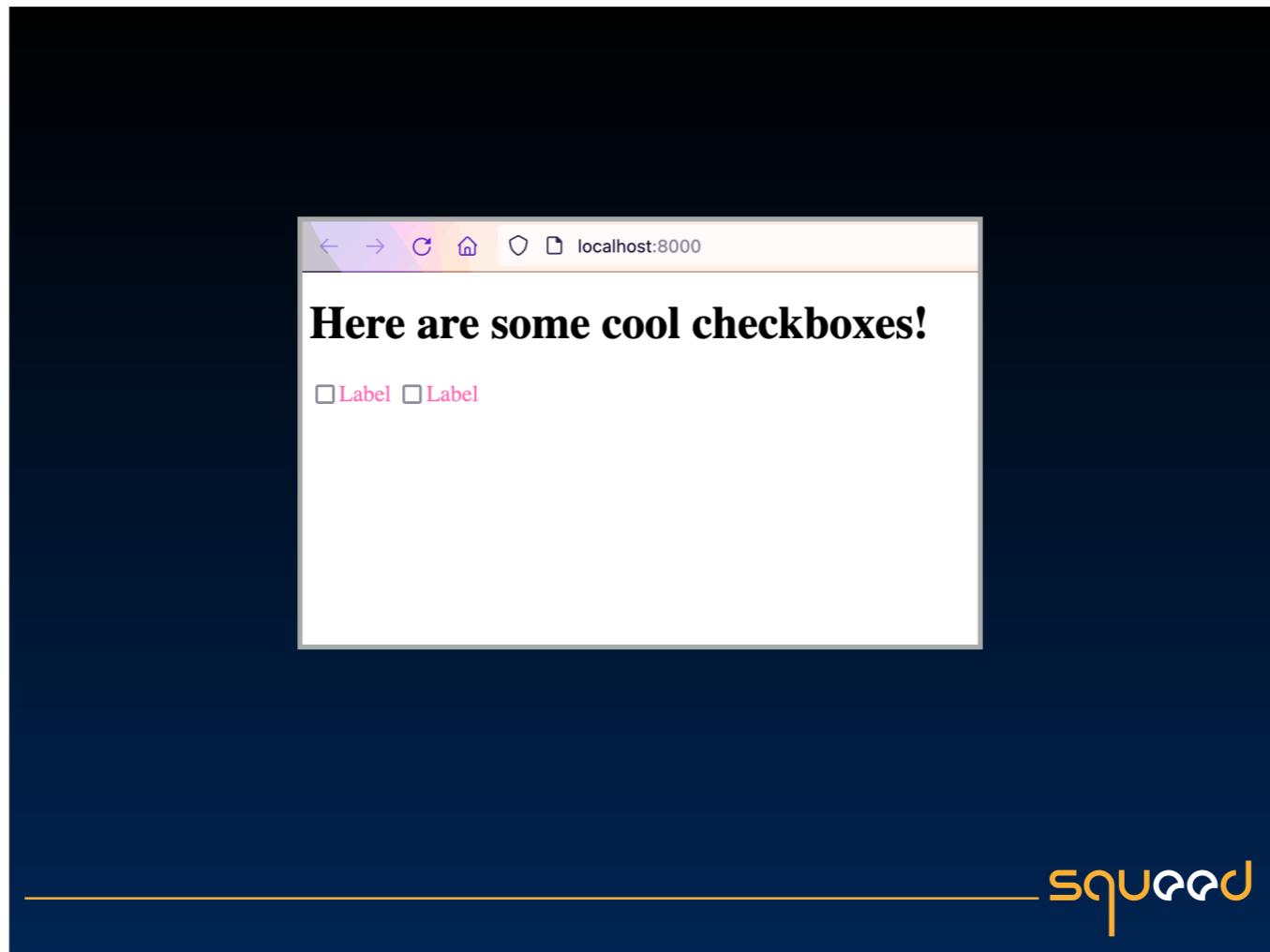
Adding some styles to our checkbox is as easy as adding it to the template. We put a style tag here, and some CSS inside - and as you may notice we're using a very broad selector: label. If this was in the light DOM, we might not want to use such broad selectors, but instead use classes and ids to target more specific elements. But the styles inside the shadow DOM only affect the elements inside the shadow DOM, and as you can see there are only two elements - the input and the label.



```
1 const template = document.createElement("template");
2 template.innerHTML =
3     <style>
4         label {
5             color: hotpink;
6         }
7     </style>
8
9     <input type="checkbox" id="checkbox" /><label for="checkbox">Label</label>
10 
```

squared

Also note that the id on the checkbox is super generic. It's just "checkbox" in this case. Ids are also encapsulated in the shadow DOM, so there will be no id conflict if elements in the light DOM or in other shadow DOMs have the same id. This allows us to write CSS that is this simple. We rarely have to bother with classes and naming methodologies - if we keep the markup in the template simple, we can keep the CSS simple. And as you know, in software development, keeping it simple is a virtue.



And here's the effect. Very hot pink.

# Let the user enter custom data

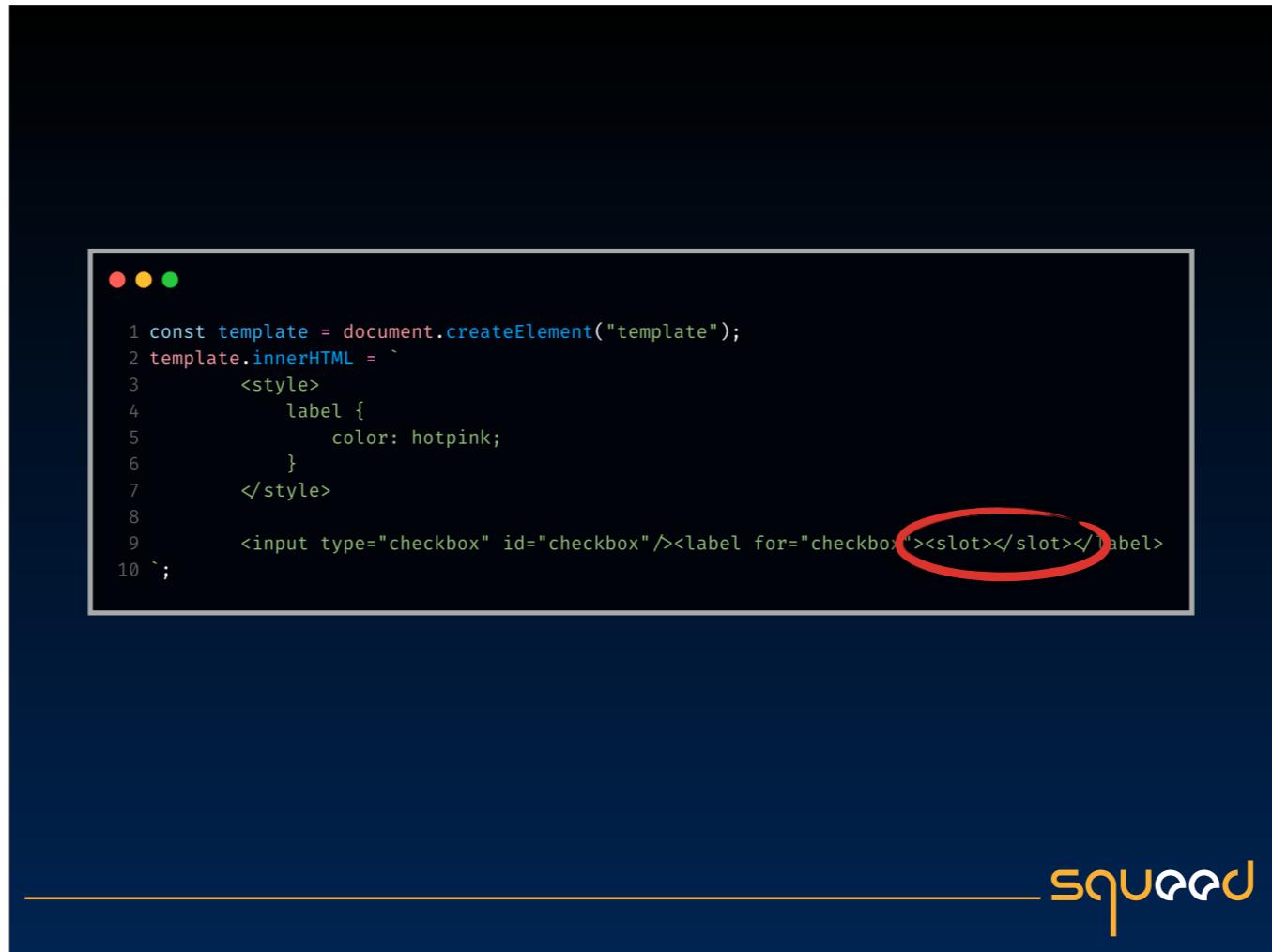


Ok, so a web component can do a lot when it comes to deciding its content, but it wouldn't be very versatile if the user couldn't affect the content and the appearance of the component. And there are multiple ways to do this.

# Using <slot>



One way is using slots. This is a concept that will be very familiar to you if you've worked with slots in Vue. It's also not unlike the concept of children in React.

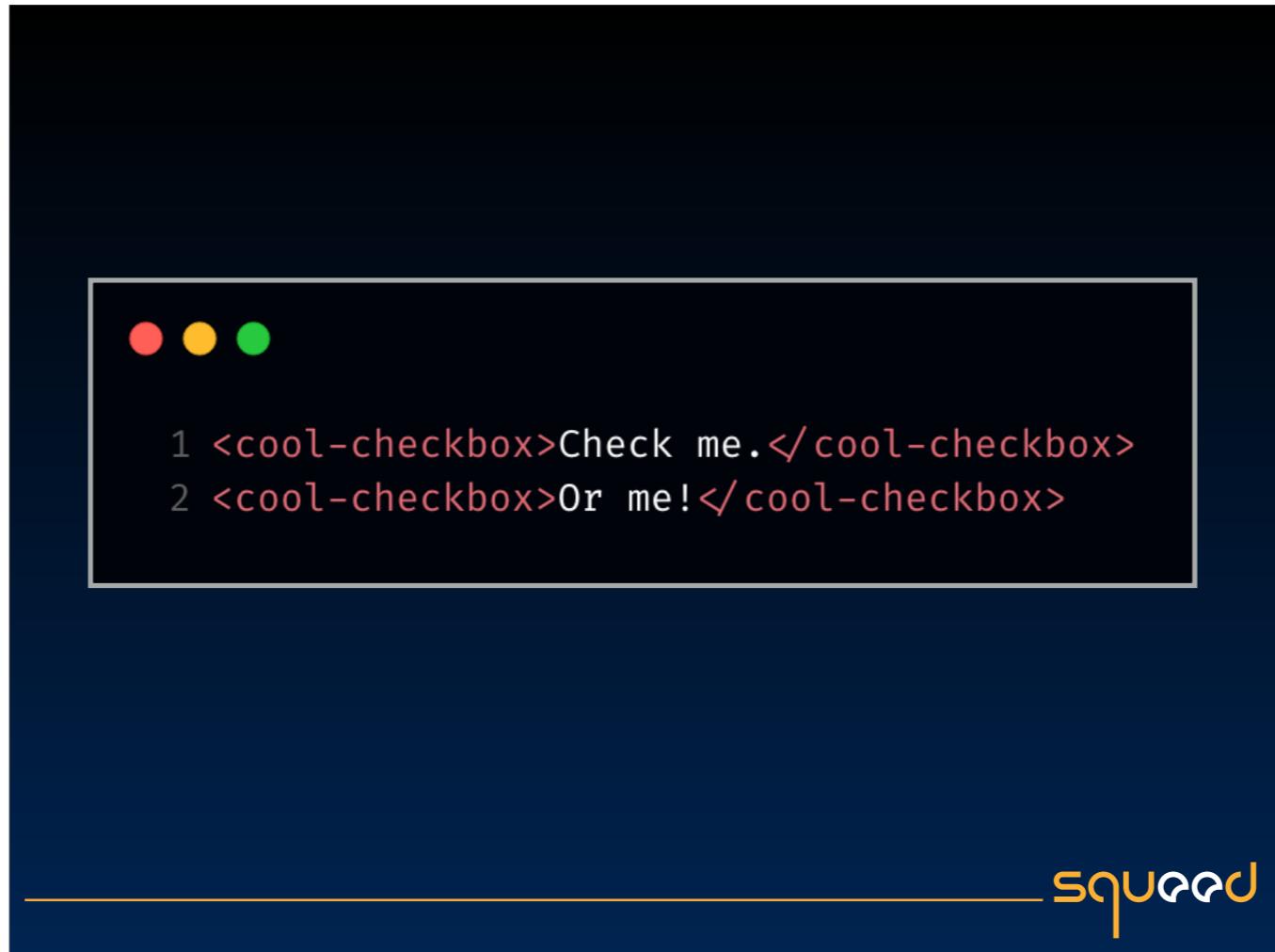


```
1 const template = document.createElement("template");
2 template.innerHTML =
3   <style>
4     label {
5       color: hotpink;
6     }
7   </style>
8
9   <input type="checkbox" id="checkbox"/><label for="checkbox"><slot></slot></label>
10 `;
```

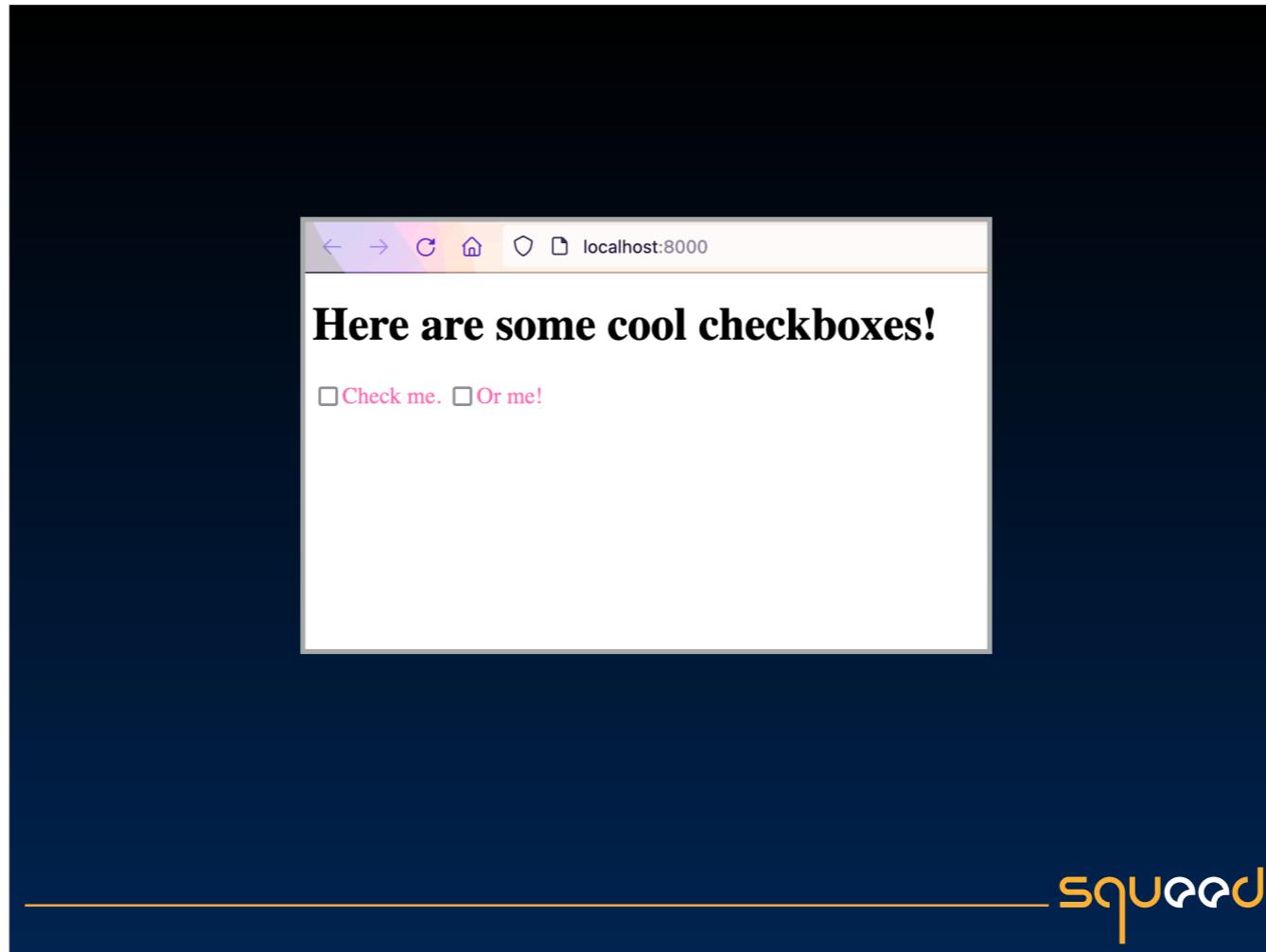
The screenshot shows a terminal window with a dark background and light-colored text. It displays a snippet of JavaScript code. The code creates a template element, sets its innerHTML to a string containing a style block and an input checkbox with a label. A red circle highlights the <slot> tag within the label's content.

squared

What a slot is, is a place in the template that we allow the user to fill with content. We place the slot tags anywhere we want in the template...



...and then we put some content between the tags of our custom element. It can be text, like this, but it can also be any valid HTML. There's really no limit.



And the result is that whatever we put between the element tags ends up inside the slot tags! Pretty simple, right?

It's also possible to have multiple slots, but then you have to give them names, and the user must use corresponding names to place different pieces of content in the different slots.

# Using attributes and properties



The next way to give data to our component is by using attributes and properties.



```
1 <element attribute="value"></element>
```



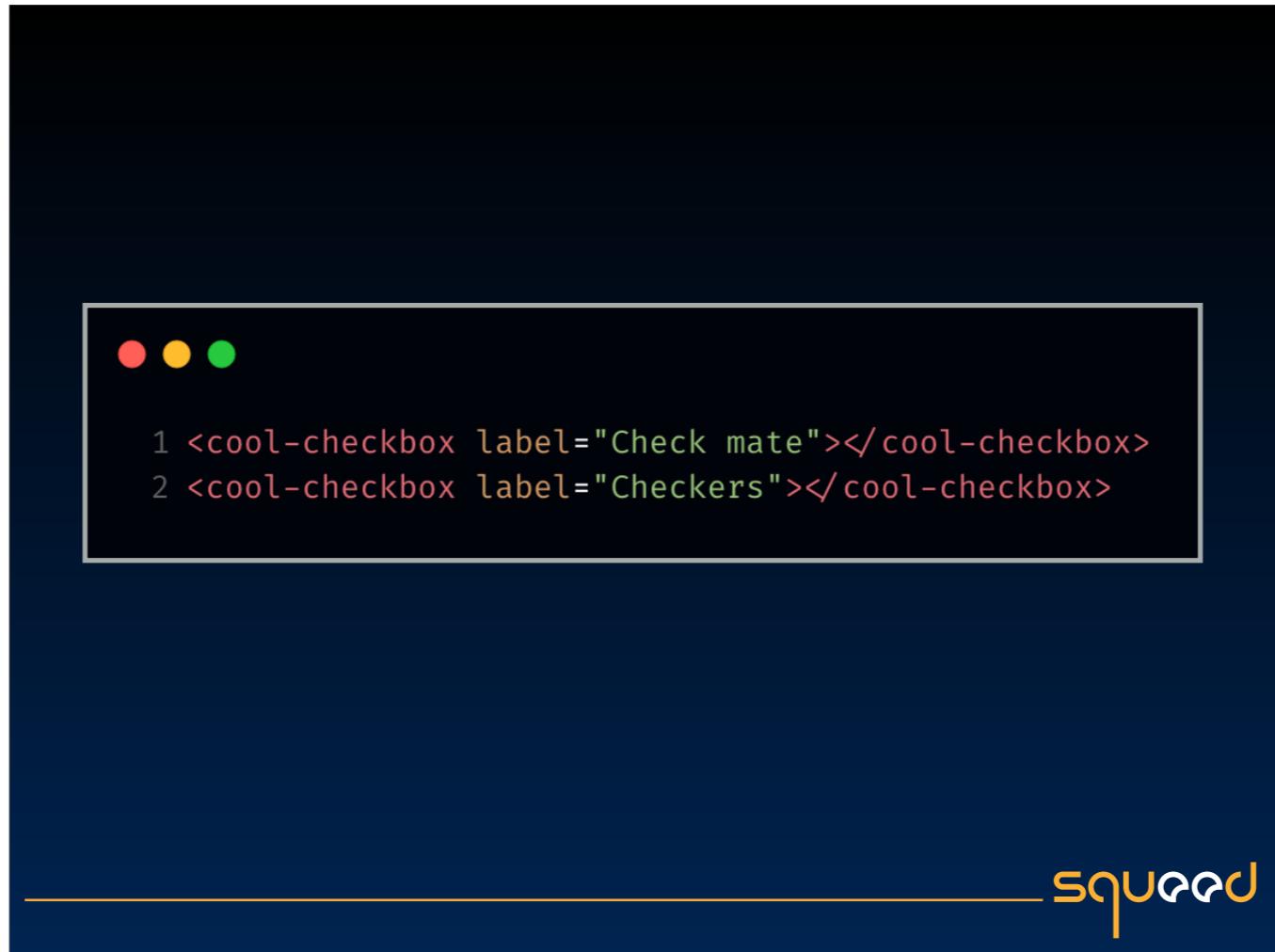
```
1 const element = querySelector('query');
2 element.property = 'value';
```



It's important to know the difference between an attribute and a property.

An attribute is what we set on the HTML tag. We can access this attribute with JavaScript, but it will always be a string.

A property is what we have on a JavaScript object. And when it comes to the DOM, each element in HTML is represented by a JS object. This is what we get when we use things like querySelector. This object has properties that we can access and change, just like any other JS object.



So, let's say we want to set the text of the label via an attribute, instead of a slot. Makes sense in a way, since a label should probably just be a piece of text, right? Well, if we set the attribute on the cool-checkbox tags in the HTML like this, that doesn't mean that we automatically get a property on the corresponding JS object - namely our custom element.



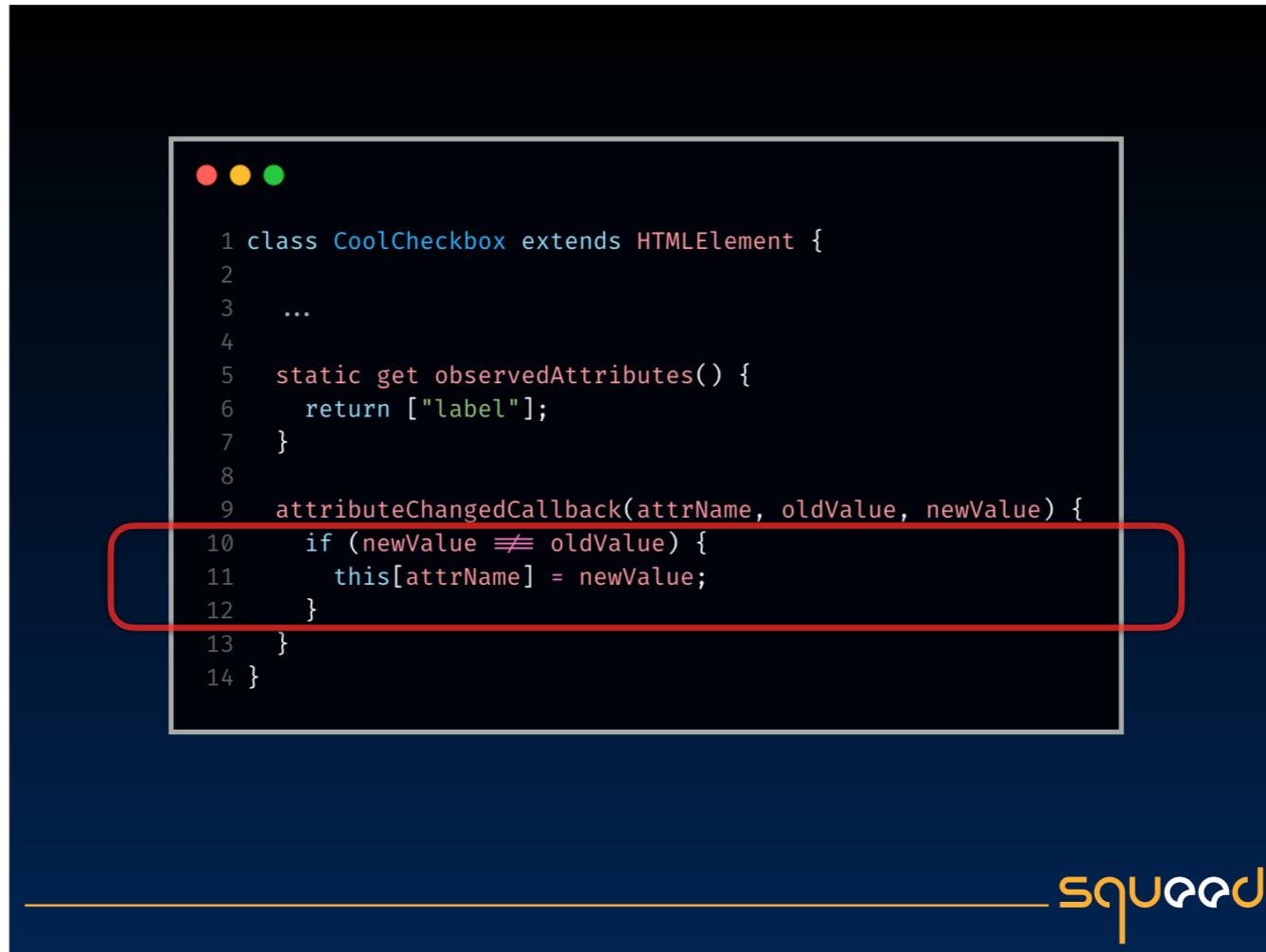
```
1 class CoolCheckbox extends HTMLElement {  
2  
3     ...  
4  
5     connectedCallback() {  
6         const templateContent = document.importNode(template.content, true);  
7         this.shadowRoot.appendChild(templateContent);  
8  
9         this.label = this.getAttribute("label");  
10    }  
11 }
```

The code shows a class named CoolCheckbox that extends the HTMLElement. It contains a connectedCallback function which imports a templateContent from the document and appends it to the element's shadowRoot. A line of code that sets the element's label attribute is highlighted with a red oval.

squared

We have to do that ourselves. And that's what we do here in the connectedCallback function - we get the attribute, and we set it on the element.

So ok, that good, but remember: the connectedCallback function only runs once - when the element is inserted into the DOM. What if the user manages to change the attribute?



```
1 class CoolCheckbox extends HTMLElement {  
2  
3     ...  
4  
5     static get observedAttributes() {  
6         return ["label"];  
7     }  
8  
9     attributeChangedCallback(attrName, oldValue, newValue) {  
10        if (newValue !== oldValue) {  
11            this[attrName] = newValue;  
12        }  
13    }  
14 }
```

The code editor shows a snippet of JavaScript. A red rectangular box highlights the `attributeChangedCallback` method and its body. The method takes three parameters: `attrName`, `oldValue`, and `newValue`. It checks if `newValue` is not equal to `oldValue`, and if so, it updates the corresponding property on the element.

squared

That's where this next lifecycle method comes into play: `attributeChangedCallback`. As the name implies, this method fires anytime one of our attributes is changed, and we can update the property accordingly. Actually, that's not quite the whole truth, I should say that this method fires whenever one of our observed attributes change, and the observed attributes are the ones returned by this getter function you see here:

```
1 class CoolCheckbox extends HTMLElement {  
2  
3     ...  
4  
5     static get observedAttributes() {  
6         return ["label"];  
7     }  
8  
9     attributeChangedCallback(attrName, oldValue, newValue) {  
10        if (newValue !== oldValue) {  
11            this[attrName] = newValue;  
12        }  
13    }  
14 }
```

squared

static get observedAttributes. If any attribute in this list change, then the attributeChangedCallback will fire. This is because there is no limit to what attributes a user can put on any HTML tag, so this ensures that we don't run a bunch of code because of a change to something we don't even care about.

Alright, so now whenever the label attribute is set or changed, the property will be updated. Sync achieved! But... what about the other direction? If we use JS to change the property directly, we also want the attribute in the HTML to update, right? And, even more importantly, we want the actual text on the page to change. This is not something that happens automatically either.



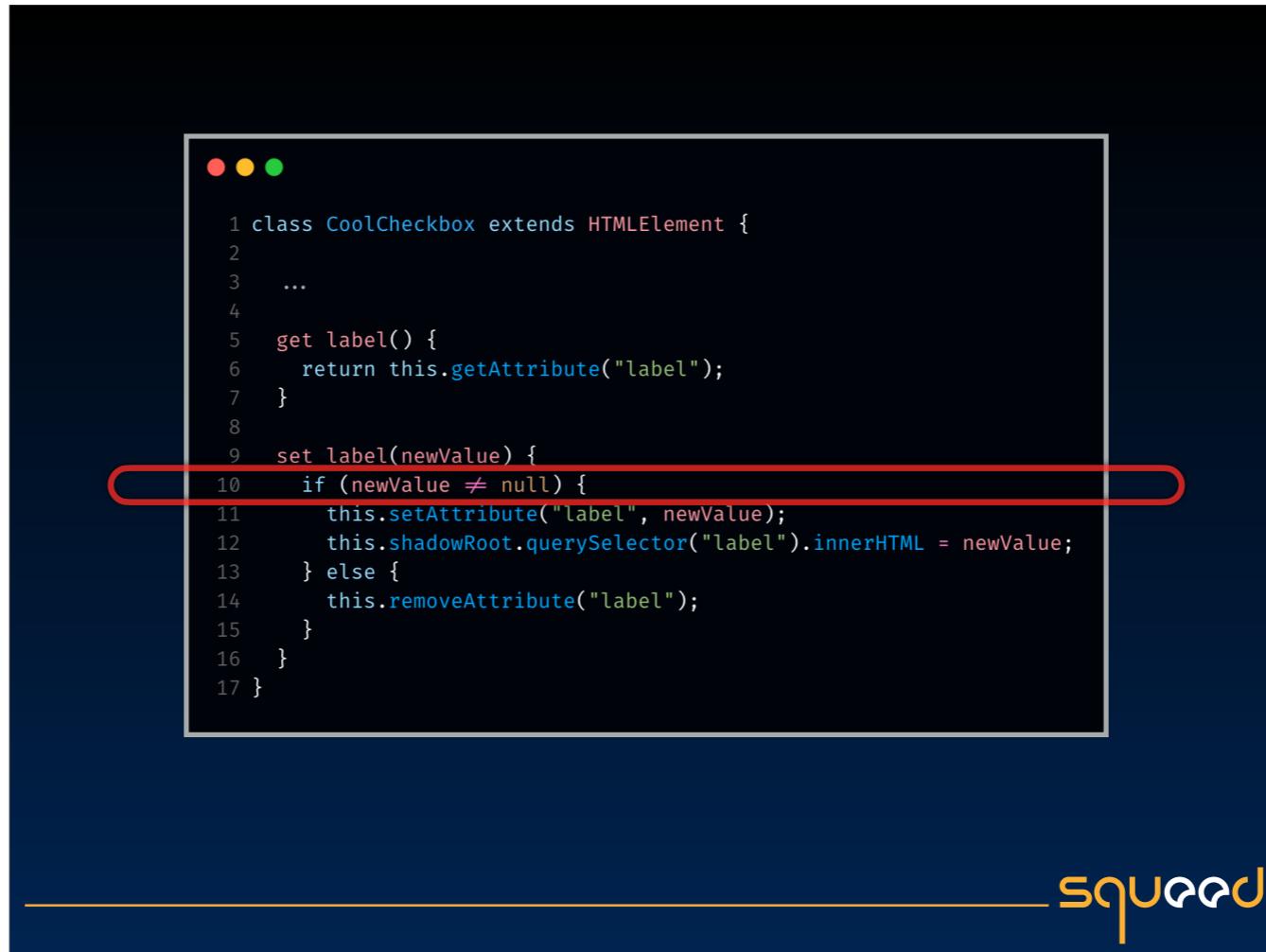
A screenshot of a code editor window on a dark background. The window title bar shows three colored dots (red, yellow, green). The code editor displays a file named 'CoolCheckbox.js' with the following content:

```
1 class CoolCheckbox extends HTMLElement {  
2  
3     ...  
4  
5     get label() {  
6         return this.getAttribute("label");  
7     }  
8  
9     set label(newValue) {  
10        if (newValue != null) {  
11            this.setAttribute("label", newValue);  
12            this.shadowRoot.querySelector("label").innerHTML = newValue;  
13        } else {  
14            this.removeAttribute("label");  
15        }  
16    }  
17}
```

The getter method 'label()' is highlighted with a red rounded rectangle.

squared

So, let's make it happen! The easiest way to accomplish it, I think, is to use a custom getter and setter for our property. The getter is very simple, it just returns the value of the attribute. The setter does a little more.



```
1 class CoolCheckbox extends HTMLElement {  
2  
3     ...  
4  
5     get label() {  
6         return this.getAttribute("label");  
7     }  
8  
9     set label(newValue) {  
10        if (newValue != null) {  
11            this.setAttribute("label", newValue);  
12            this.shadowRoot.querySelector("label").innerHTML = newValue;  
13        } else {  
14            this.removeAttribute("label");  
15        }  
16    }  
17}
```

The image shows a screenshot of a code editor with a dark theme. A red oval highlights the first line of the `set label(newValue)` block, specifically the `if (newValue != null)` condition. The code is written in JavaScript, defining a class `CoolCheckbox` that extends `HTMLElement`. It includes methods for getting and setting the `label` attribute, which involves interacting with the element's shadow root.

squared

Whenever we set the property `label` this code is going to run, and the first thing it does is it checks if the new value of `label` exists at all.

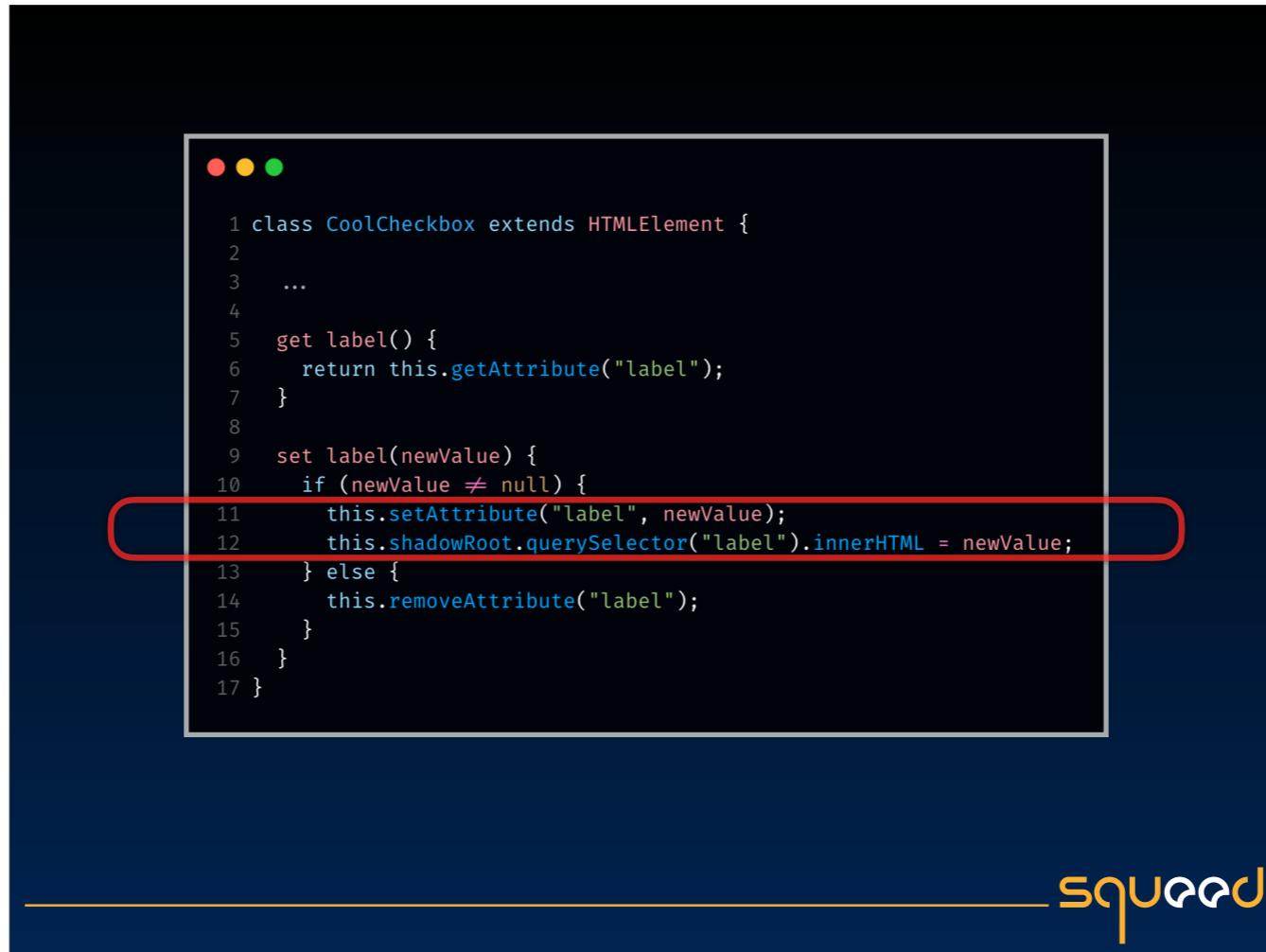


```
1 class CoolCheckbox extends HTMLElement {  
2  
3     ...  
4  
5     get label() {  
6         return this.getAttribute("label");  
7     }  
8  
9     set label(newValue) {  
10        if (newValue != null) {  
11            this.setAttribute("label", newValue);  
12            this.shadowRoot.querySelector("label").innerHTML = newValue;  
13        } else {  
14            this.removeAttribute("label");  
15        }  
16    }  
17}
```

The screenshot shows a code editor window with a dark theme. A red oval highlights the line of code `this.removeAttribute("label");` within the `set label(newValue)` block. The code is written in JavaScript, defining a class `CoolCheckbox` that extends `HTMLElement`. It includes methods to get and set the `label` attribute, utilizing both the standard `getAttribute`/`setAttribute` API and a custom shadow DOM element.

squared

If not, then we reflect that change by removing the attribute from the HTML.



```
1 class CoolCheckbox extends HTMLElement {
2
3     ...
4
5     get label() {
6         return this.getAttribute("label");
7     }
8
9     set label(newValue) {
10        if (newValue != null) {
11            this.setAttribute("label", newValue);
12            this.shadowRoot.querySelector("label").innerHTML = newValue;
13        } else {
14            this.removeAttribute("label");
15        }
16    }
17}
```

squared

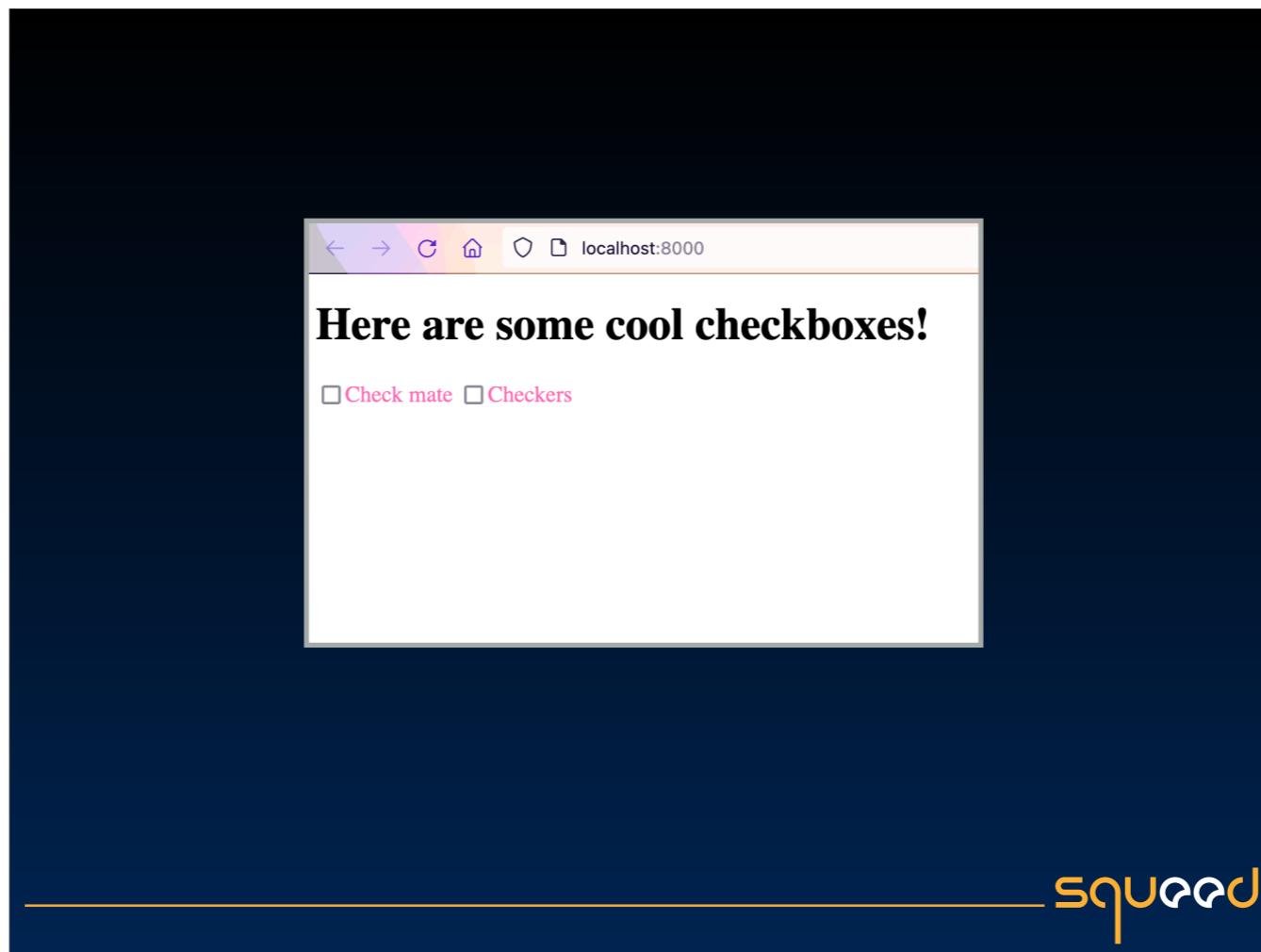
If the label property does indeed have a value, then we set the label attribute to this value, and we also update the inner HTML, so that the change appears visually in the browser.



```
1 <cool-checkbox label="Check mate"></cool-checkbox>
2 <cool-checkbox label="Checkers"></cool-checkbox>
```

squared

Now the label attribute and property are all wired up, stay in sync, and behaves as expected! We use it like this...

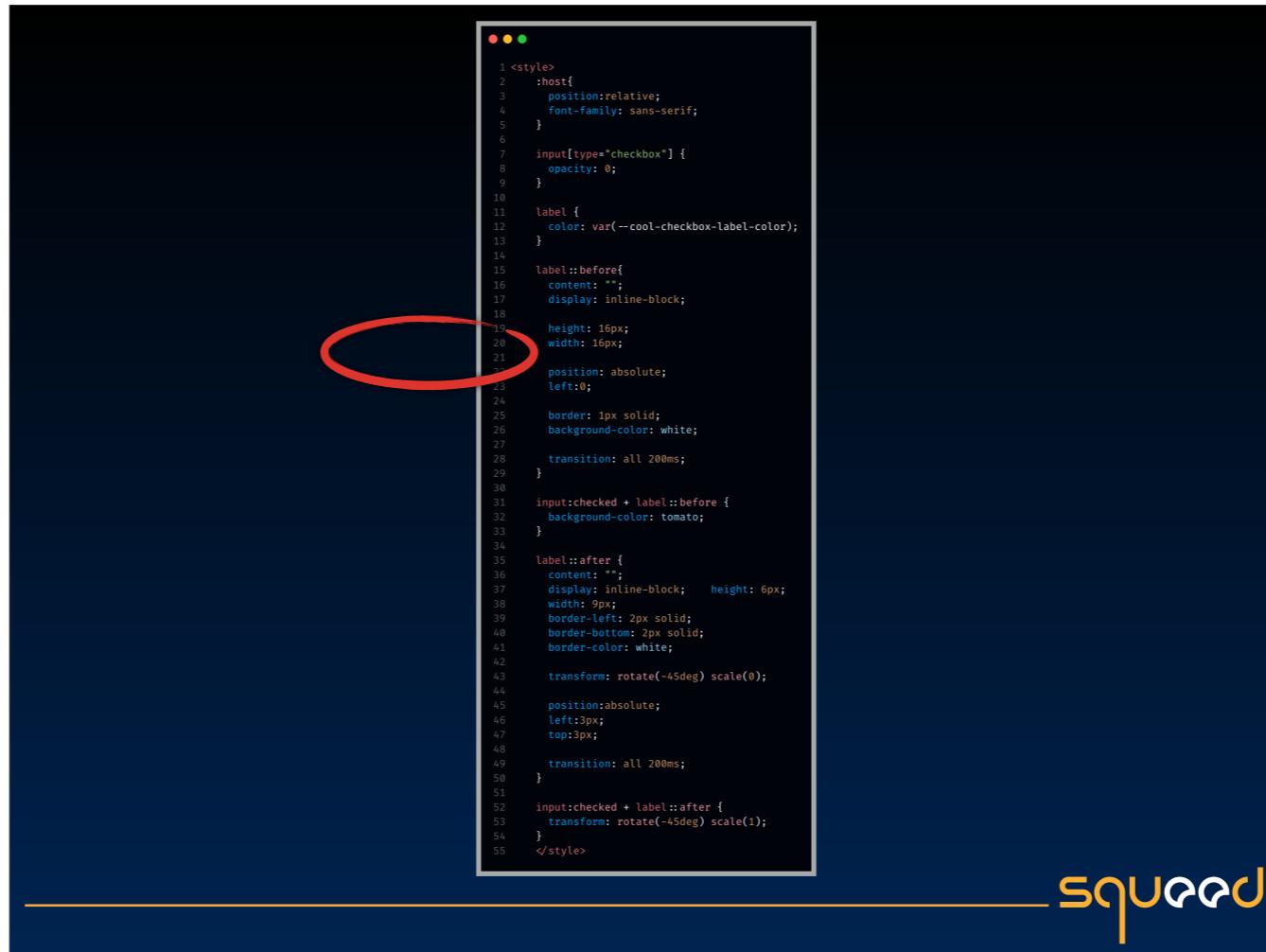


...and it appears just the way we want!

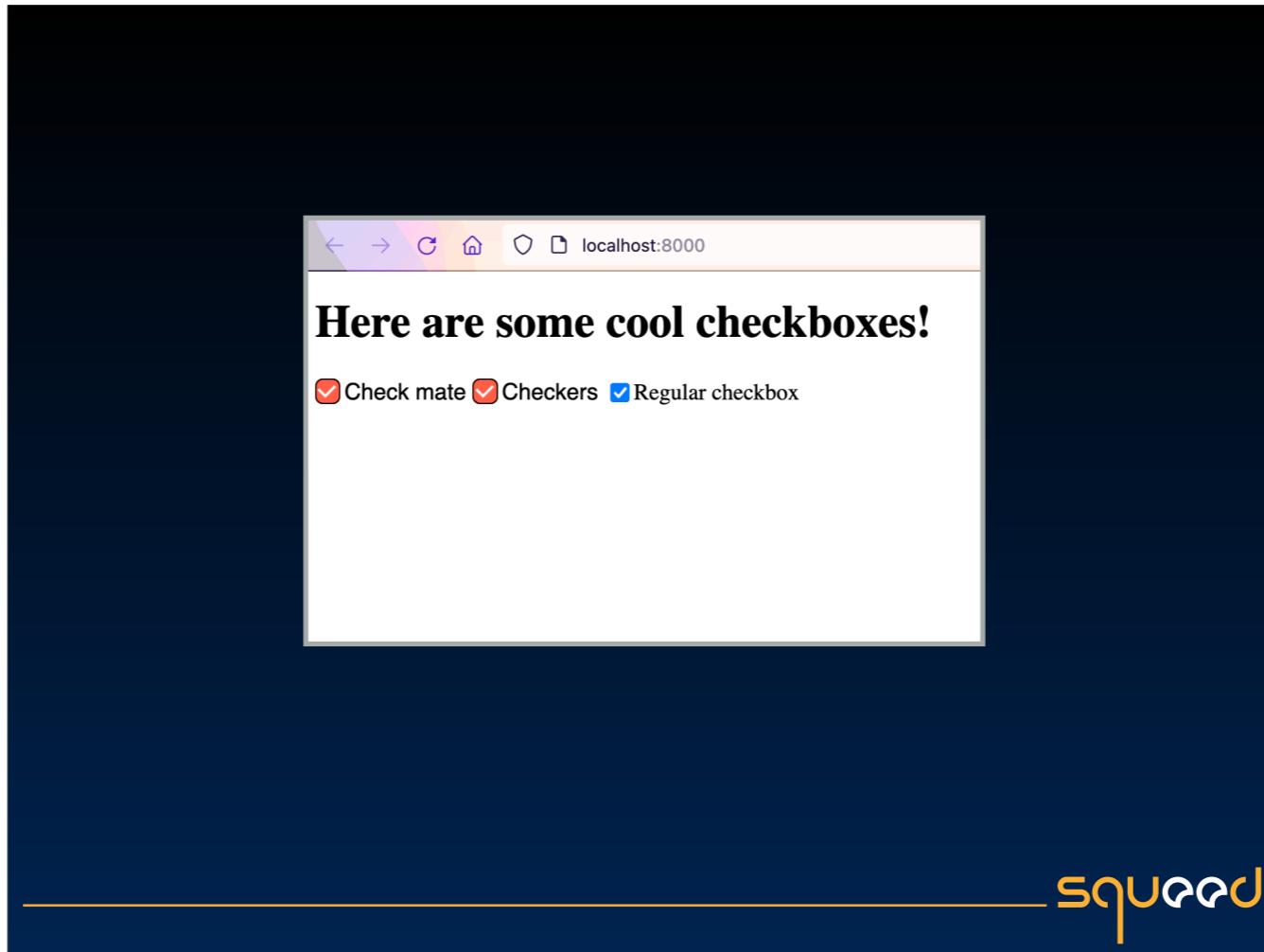
# Piercing the veil



Ok, so the final way of letting the user affect the content and behaviour of our web component is by piercing the shadow veil. I mentioned this earlier - the shadow DOM and light DOM are separate - but there are controlled ways to let information slip through, and that's what we're going to explore now.

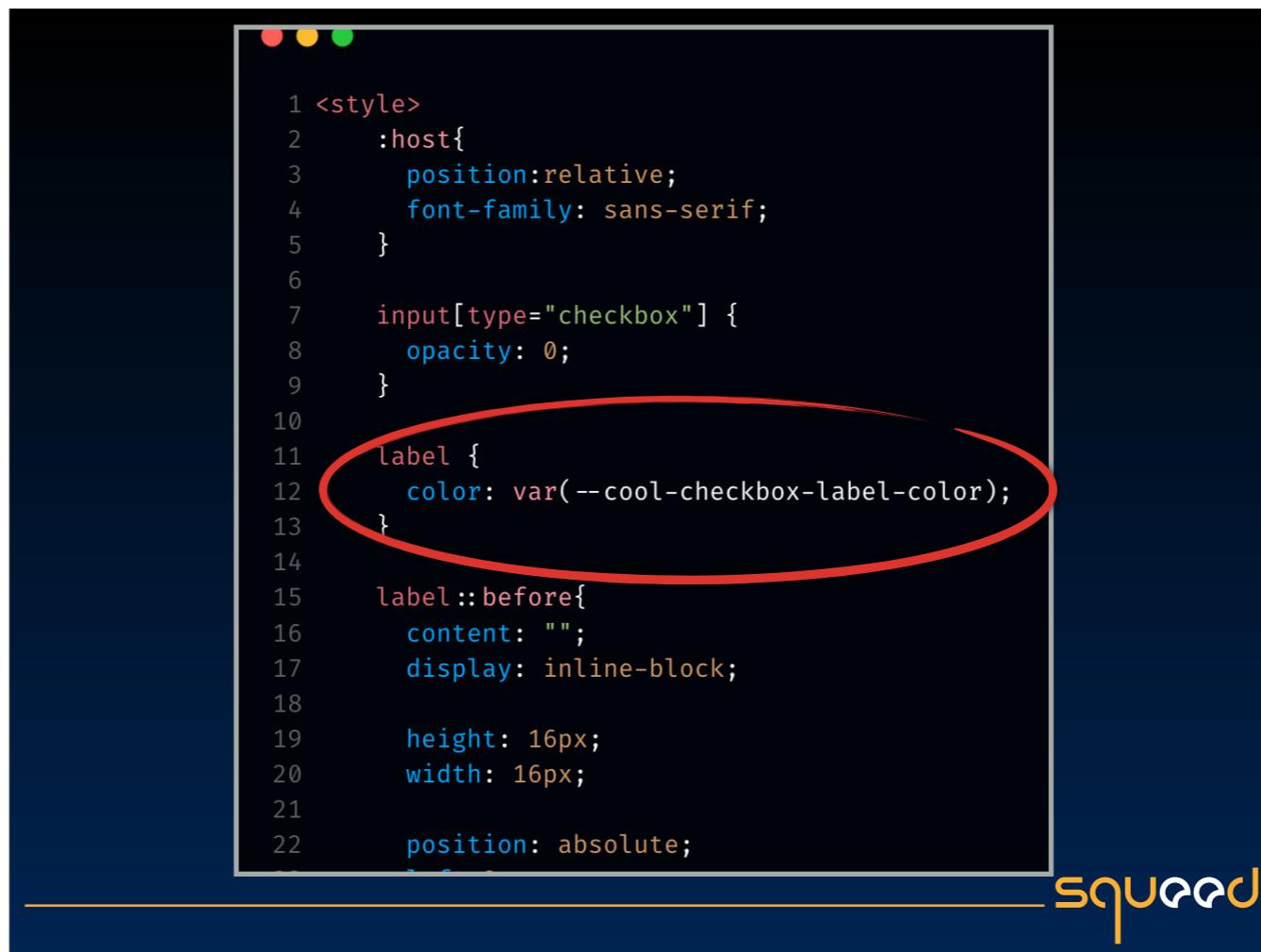


So, first of all I'll add a bunch of CSS to make the checkboxes really custom. We won't go closer into this, because how to style a checkbox is a session all in its own. But there is one interesting part of this CSS that you won't find in the light world - the :host pseudo-selector. This selector targets the host element - that is, in our case, the actual cool-checkbox-element itself. The container for the whole custom element really. One reason for targeting this can be to avoid having to use unnecessary wrappers on the top level since the host element is a wrapper itself.



Anyway, this is what it looks like. Now the checkboxes are truly cool, especially when compared to the regular old checkbox hanging out beside.

Ok, so notice the text colour. Black, right? Well, take a look at this CSS declaration:

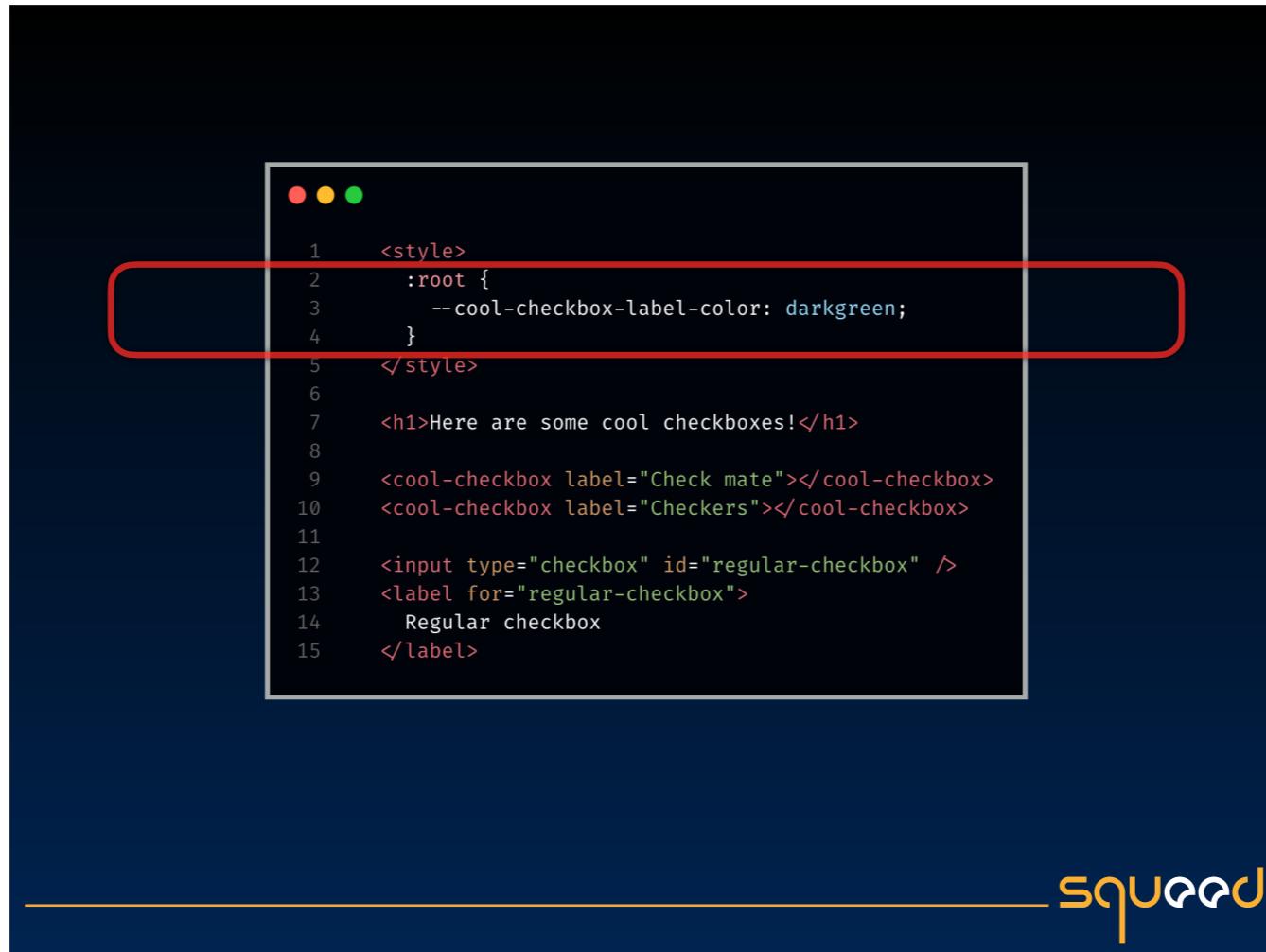


```
1 <style>
2   :host{
3     position: relative;
4     font-family: sans-serif;
5   }
6
7   input[type="checkbox"] {
8     opacity: 0;
9   }
10
11  label {
12    color: var(--cool-checkbox-label-color);
13  }
14
15  label ::before{
16    content: "";
17    display: inline-block;
18
19    height: 16px;
20    width: 16px;
21
22    position: absolute;
23    top: -16px;
24    left: -16px;
25    border-radius: 50%;
```

squqd

Here we set the text colour of the label to that of a CSS custom property. But we haven't declared this property anywhere, so the colour just reverts to its default value. And when it comes to text colour, it is an inherited value, meaning it is set to the computed value of its parent. That's why we only have to set things like colour and font-family once, on the body, then it is inherited throughout the document. And the reason why I bring this up is because inheritance is one way to pierce the shadow veil - inherited values inherit through to the shadow DOM. So if your page uses a certain typeface, your web components will automatically get the same typeface applied to them, without the need to specify this for each and every component.

But what about this custom property then, why is it there at all? Well, custom properties are another way to pierce the veil, and a very controlled one. The custom properties we use in our CSS becomes the api for our users if they want to change the internal styling of the component.

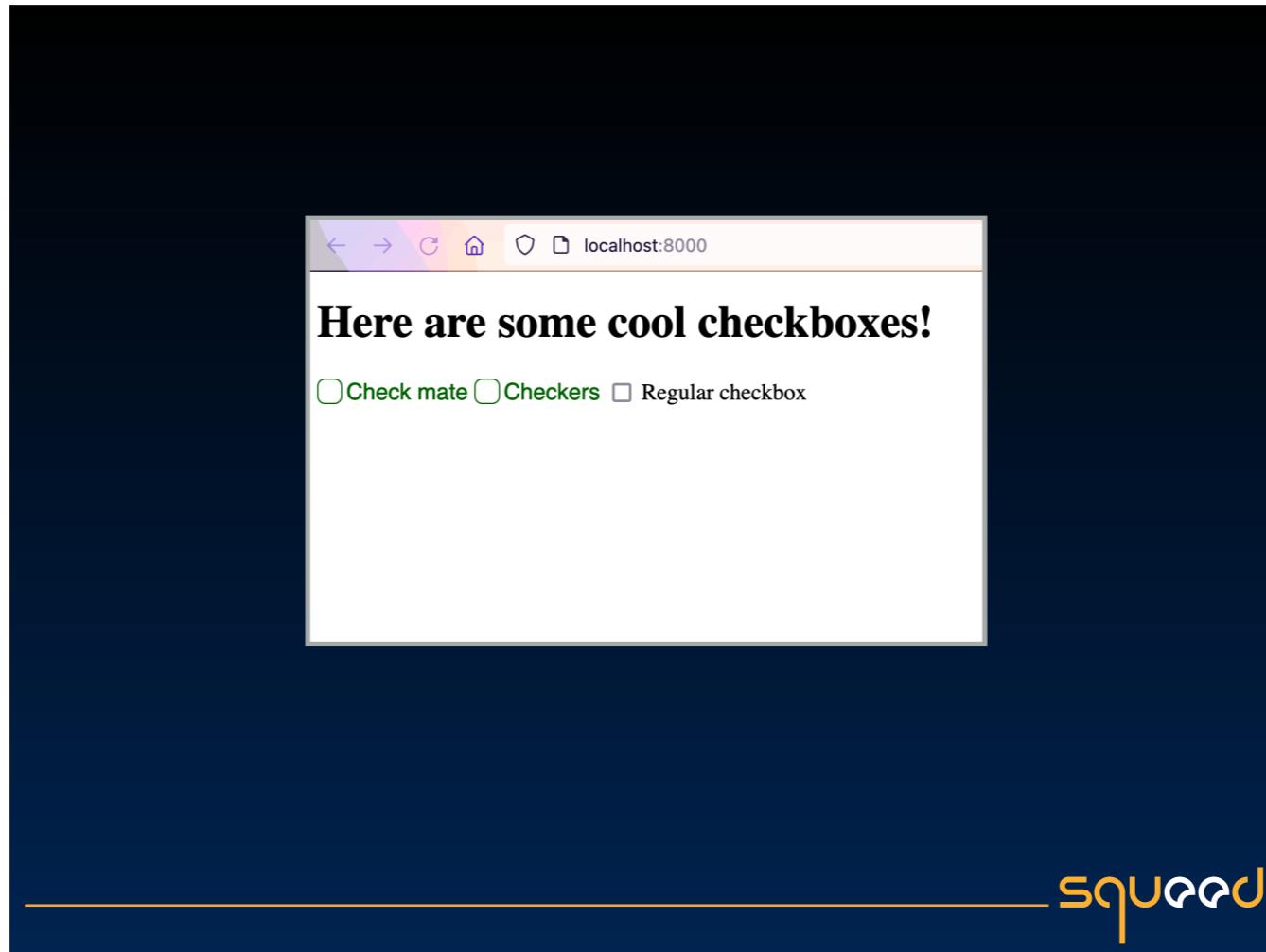


```
1  <style>
2    :root {
3      --cool-checkbox-label-color: darkgreen;
4    }
5  </style>
6
7  <h1>Here are some cool checkboxes!</h1>
8
9  <cool-checkbox label="Check mate"></cool-checkbox>
10 <cool-checkbox label="Checkers"></cool-checkbox>
11
12 <input type="checkbox" id="regular-checkbox" />
13 <label for="regular-checkbox">
14   Regular checkbox
15 </label>
```

The screenshot shows a code editor with a dark theme. At the top left, there are three colored dots (red, yellow, green). Below them is a code block. A red rounded rectangle highlights the first four lines of the CSS section. The rest of the code consists of HTML components: an 

# element, two components with labels "Check mate" and "Checkers", and a standard with a label "Regular checkbox". The SQuared logo is visible at the bottom right.

Like this! Here, out in the light world, we have some styling that defines that custom property: cool-checkbox-label-color. This is picked up by the component and...



...the text turns green! And since custom properties can be pretty much anything, they provide quite a powerful api that we can expose to our users, allowing them to customise pretty much anything we want.

# Sending information back up



We've looked at how a user can send data into the web component to affect things, and now the final thing I want to show you is how to send data from the web component back up to the user. This is done using the good 'ol browser api: events.



```
1 connectedCallback() {  
2   ...  
3   const inputElement = this.shadowRoot.querySelector("input");  
4   inputElement.addEventListener(  
5     "change",  
6     () => this.dispatchEvent(new Event("customevent"))  
7   );  
8 }
```

The SQuRRD logo is visible in the bottom right corner of the terminal window.

It might look like this in the connectedCallback function: we find something to react to (in our case its the checkbox's change event, but it could just as well be a button press, or the completion of a network call or something)



A screenshot of a browser developer tools console window. The code shown is:

```
1 connectedCallback() {  
2   ...  
3   const inputElement = this.shadowRoot.querySelector("input");  
4   inputElement.addEventListener(  
5     "change",  
6     () => this.dispatchEvent(new Event("customevent"))  
7   );  
8 }
```

The line `6 () => this.dispatchEvent(new Event("customevent"))` is highlighted with a red oval.

squared

and then we dispatch a new event on the custom element. Notice that it's not on the shadowRoot this time, but on the element itself.



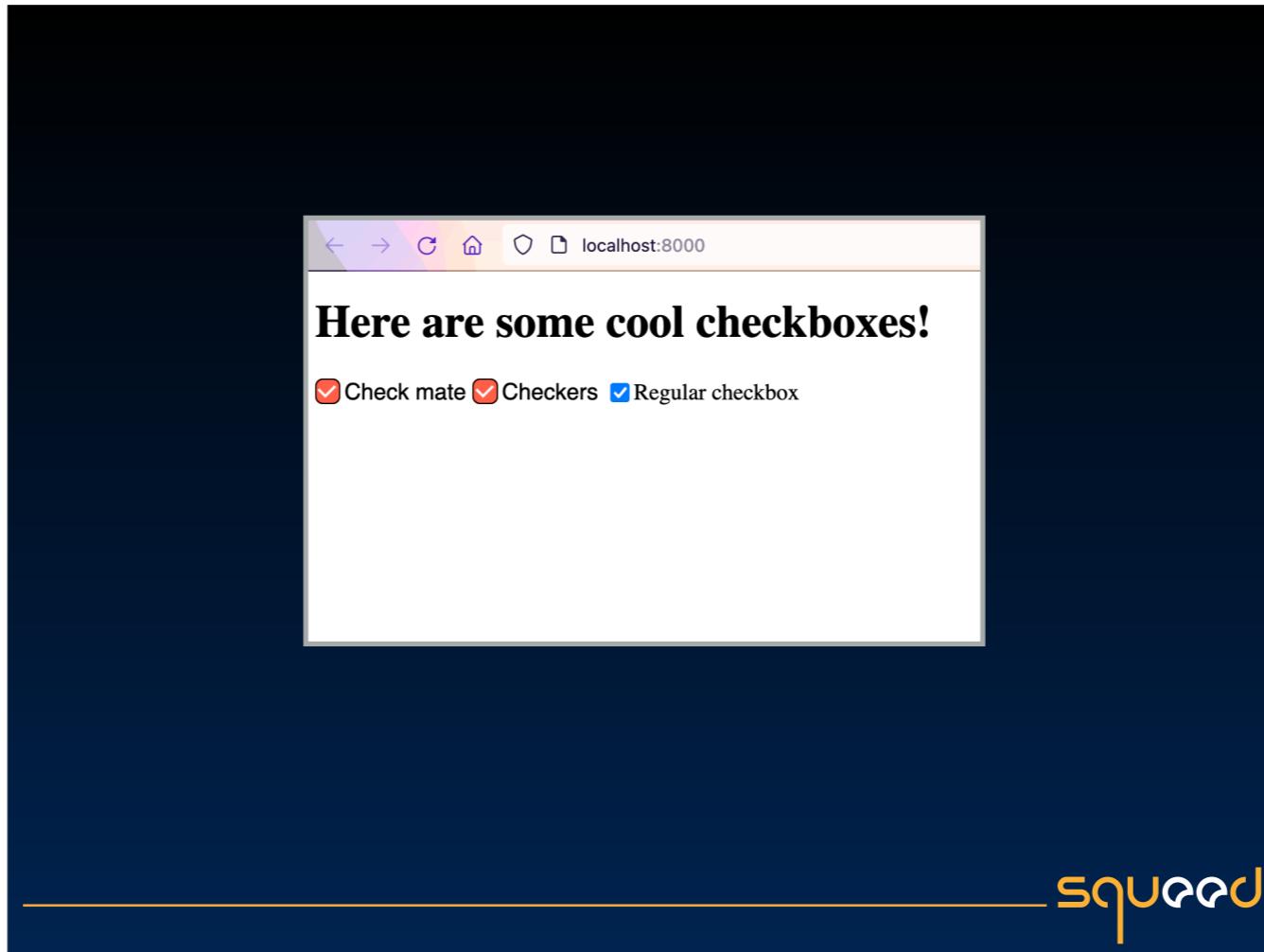
A screenshot of a terminal window with a dark background. In the top-left corner, there are three small colored circles: red, yellow, and green. Below them is a white rectangular box containing the following JavaScript code:

```
1 document
2   .querySelector("cool-checkbox")
3   .addEventListener("customevent", () =>
4     console.log("Something happened in the web component!")
5 );
```

At the bottom right of the terminal window, the word "SQUARED" is written in a stylized orange font.

This is because the user can then attach an event listener to the custom element, and in turn, react to this event being dispatched.

If you're not 100% familiar with the event API, you should also know that in addition to dispatching an event with a name, the event can also contain any kind of data, that the listener can make use of.



Ok, so this concludes the creation of our checkbox web component! We're using a custom element with a shadow DOM that encapsulates styles. We're accepting user data via attributes and CSS custom properties, and we're sending data back up with custom events.

There are loads more advanced stuff that you can do with your web components, but this pretty much covers the basics. I think the main thing to take away from all this is that it's all native browser apis, many of which you are probably already familiar. Hopefully this will make the step to start using web components a little easier for you.

# The component and the frameworks



Using web components in existing web frameworks is so straight forward that I don't even have any example slides for it. As long as you've run the registration code once, then you can just use them like you would use any regular HTML element. Super handy! The only major exception is actually React, which requires a thin wrapper around the components in order for listeners and attributes to work correctly.

# Web component frameworks



As you've seen throughout this talk, the APIs that we use to create our web component are fairly low level. There are many things that need to be wired up manually, which leads to a lot of boilerplate code. Wouldn't it be easier then, to make an abstract class that takes care of all the boilerplate, and then you can inherit from that, instead of inheriting directly from `HTMLElement`? Well, yes, it would, and luckily there are already a bunch of clever people who have thought of this and made such classes that we can use. Some have gone even further than that, but I think it's ok to call all of these "Web component frameworks", even though they are usually very small and lightweight, because they make it so much easier and cleaner to write web components. Underneath though, they use all the technology that we've looked at in this talk, so having that understanding makes it a lot easier to understand and use the frameworks.



Here are a couple of the most popular web component frameworks right now: Lit is a very competent base class that you can inherit from. It only uses native JavaScript syntax, and yet manages to make the api very streamlined. It is a sort of continuation of the Polymer project, and I highly recommend checking it out.

Stencil includes tooling that makes it easy to write collections or libraries of web components - they also use JSX in their syntax, so someone coming from React might feel comfortable there.

There are also many others out there, like solid or lightning.



And there are also existing javascript frameworks that can actually compile their components into web components, such as Vue or Svelte. I will admit though, that I haven't explored how easy or convenient this is myself yet, but the possibility is there!

In conclusion, when it comes to web component frameworks, you will probably want to use one of these when you write web components, at least if you need a moderate amount of complexity, or more. They do make things a lot easier, and they don't add a lot of overhead. But, as with all things, having an understanding of how web components work on a fundamental level will make it easier to understand what the frameworks do, and it will make you a better developer in the long run.

# What does the future hold?



Ok, we're nearing the end of this talk, so let's take a moment to look to the future and ask ourselves: what role will web components play in the future of web development?

# Use cases for web components



Web components are mature enough, and has enough browser support to be used in production today. And they are! In many places. Some people already use web components to build entire web applications. Others use them to build single, universally useable components that can be distributed and used in any context. And, taking this further, a somewhat popular use for web components is to build UI libraries with them, possibly as part of a larger design system. This makes the library components universal, and doesn't force any specific technology on its users - this is especially useful in very large organisations where many different tech stacks already exist, or where experimentation with new things are encouraged.

# Will web components replace JS frameworks?



Some might claim that web components are here to completely replace the current JavaScript frameworks. Will it be so? Well, not very soon I don't think. There are still a lot of problems that these frameworks solve that web components, in and of themselves do not. Not to mention the huge communities and eco systems that surround them. But, you could ask yourself this: will React last and be popular forever? If not, what will replace it, and when?

Regardless of what mine or your predictions are, the fact is that web components are part of the web specification now, and that's not going away. In fact, it will only keep evolving and improve as time goes by. So sooner or later, it might be time for you too...

*Upgrade*

---

squared

to upgrade!

# Thank you

tobiasljungstrom.net  
@midvintr



Thank you very much for listening to this talk, if you want to see more of what I do or get in contact with me you can visit my website: [tobiasljungstrom.net](http://tobiasljungstrom.net), and you can also find me on twitter. Thanks again!