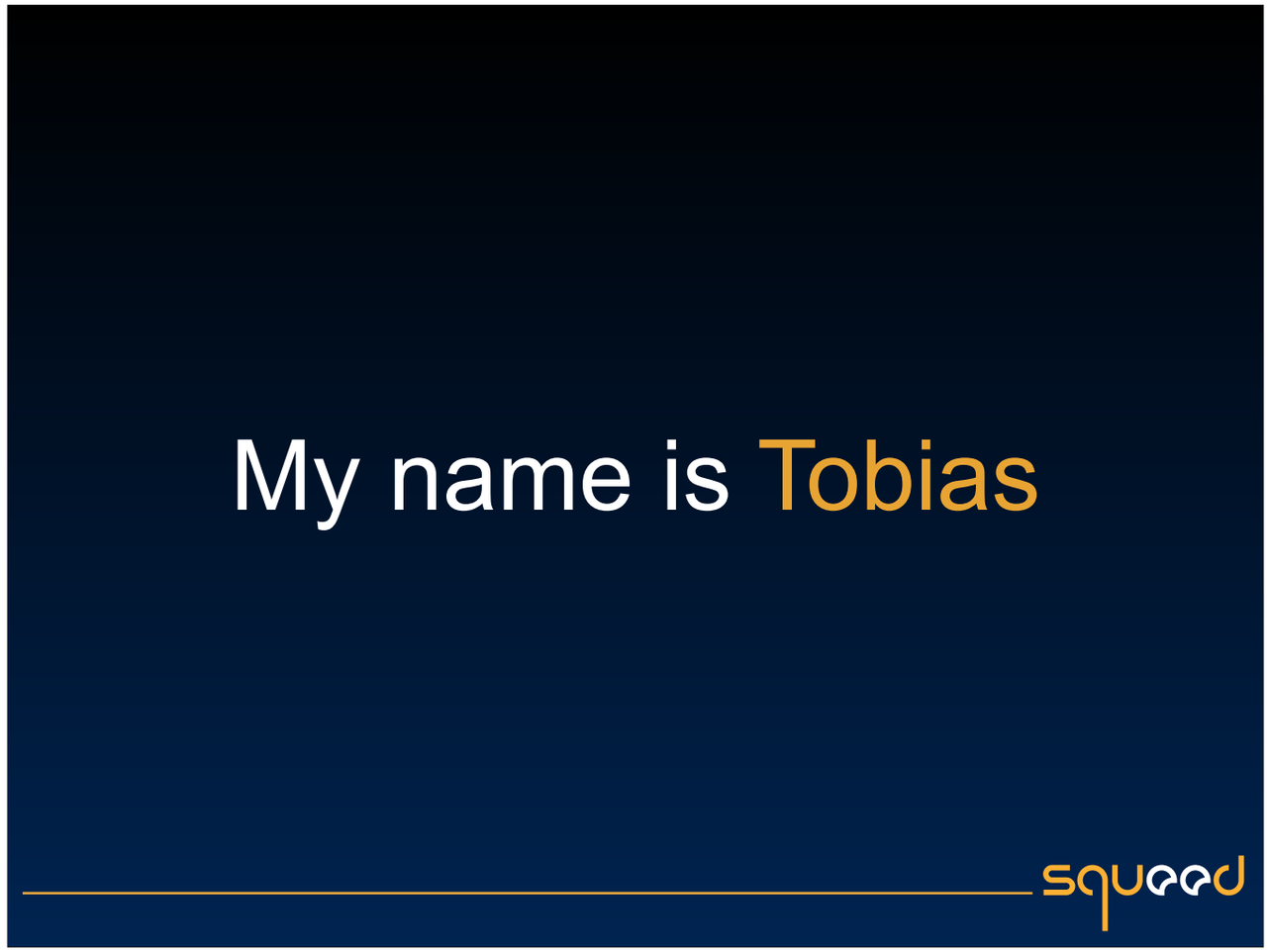
Hello everyone! It is so lovely to be here, thank you for having me. I'm here to talk about web components: what they are, how to use them, and what role they will play in the future of web development.



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

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...

...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

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...

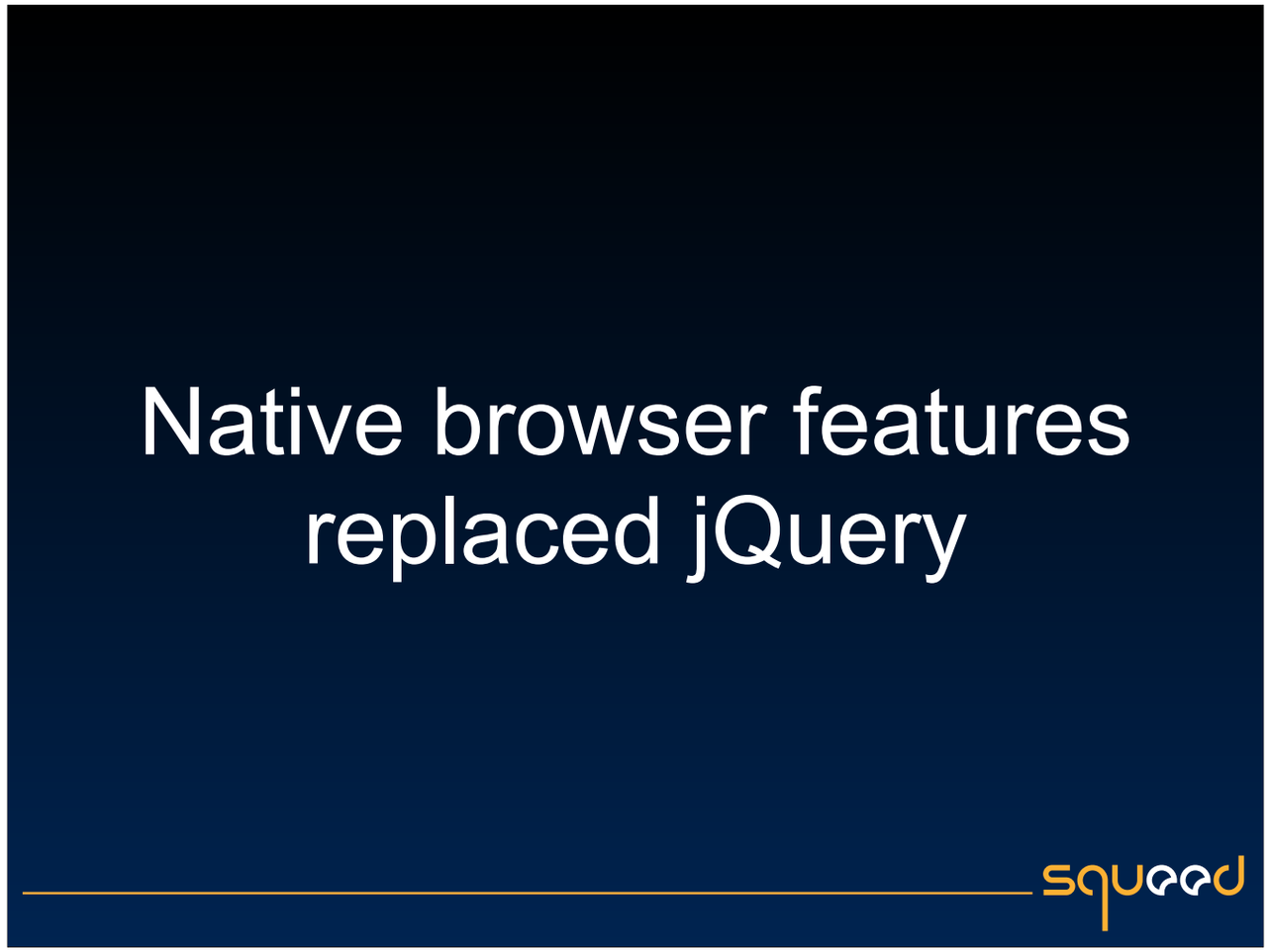
...an upgrade for everyone involved.
You know what else upgrades?

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:

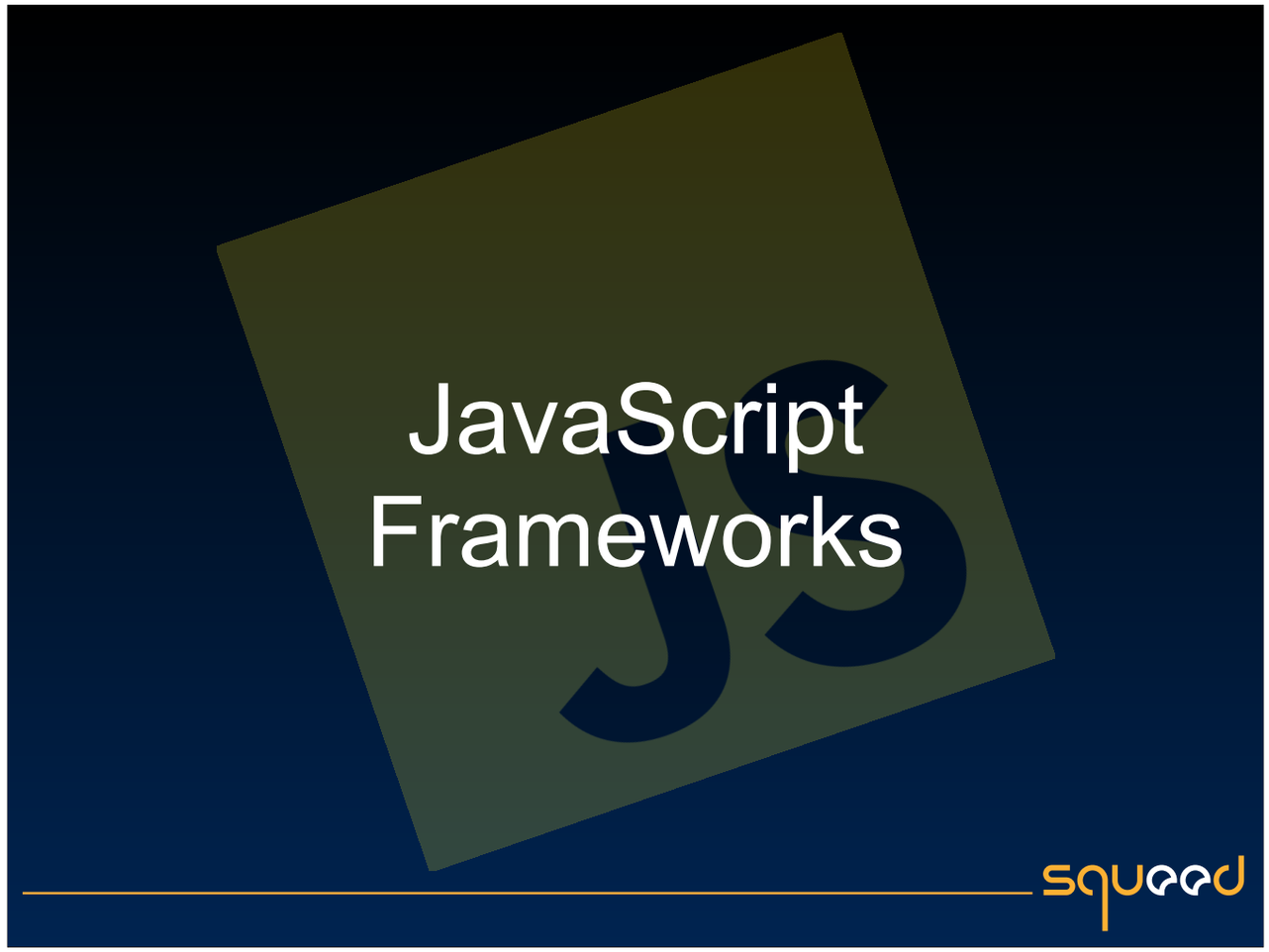
jQuery! What a fantastic library! When it came around in 2006 it was a tremendous upgrade from the way we used to query and manipulate the DOM. It made web development easier, and allowed us to create better experiences for our users.

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.



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..

...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

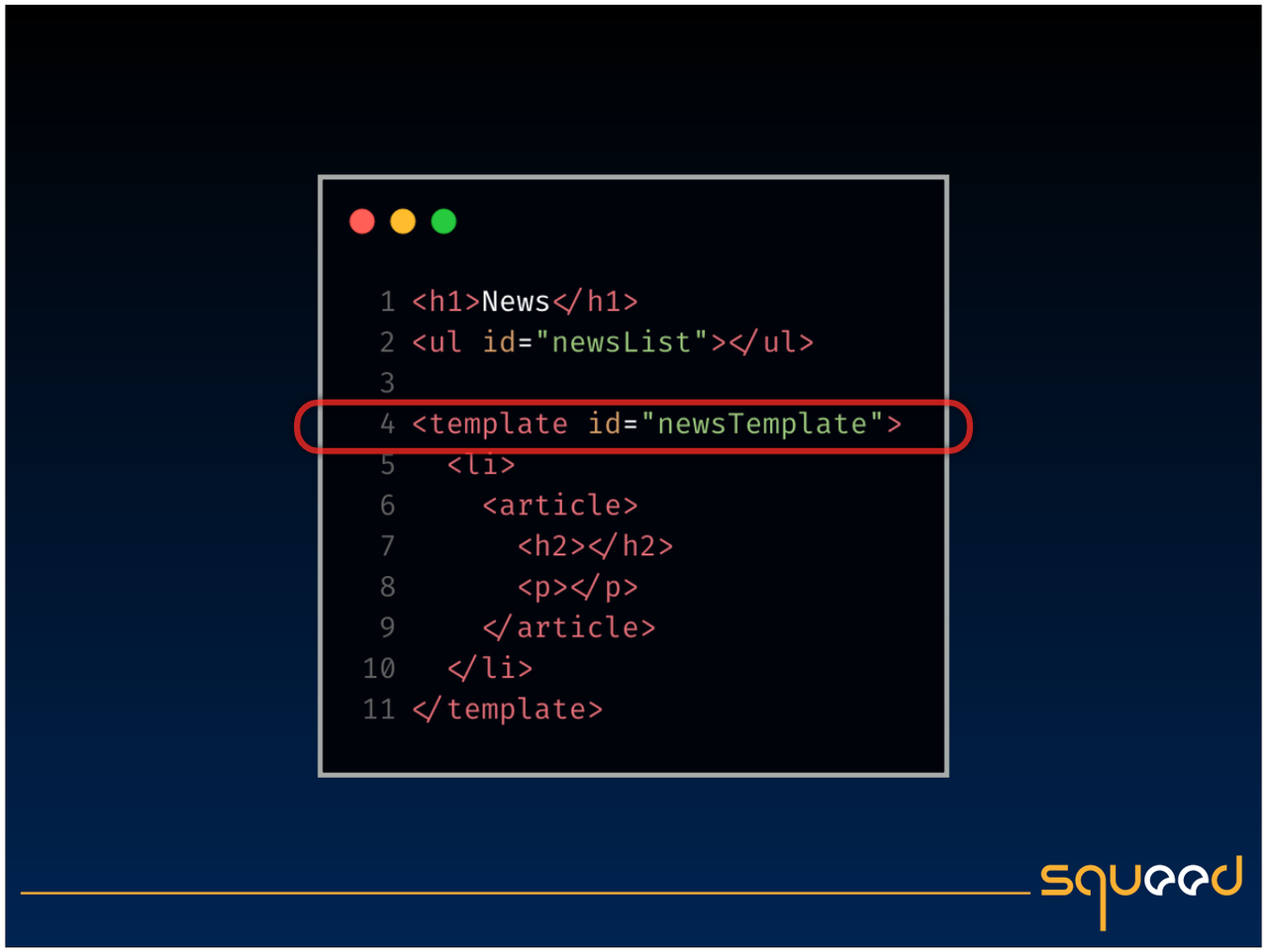
Web components are actually not just one thing, but a collection of browser APIs that we collectively call Web components. The first of these APIs that we'll take a look at are HTML templates.
So, let's imagine we have a news page, and every little news article follows the same pattern and has the same markup. It might look something like this:

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 new 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.



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 });
```

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.

```js
const newsList = document.querySelector("#newsList");
const template = document.querySelector("#newsTemplate");

data.forEach((item) => {
  const clone = template.content.cloneNode(true);
  clone.querySelector("h2").textContent = item.headline;
  clone.querySelector("p").textContent = item.content;

  newsList.appendChild(clone);
});
```

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:

```js
const newsList = document.querySelector("#newsList");
const template = document.querySelector("#newsTemplate");

data.forEach((item) => {
  const clone = template.content.cloneNode(true);
  clone.querySelector("h2").textContent = item.headline;
  clone.querySelector("p").textContent = item.content;

  newsList.appendChild(clone);
});
```

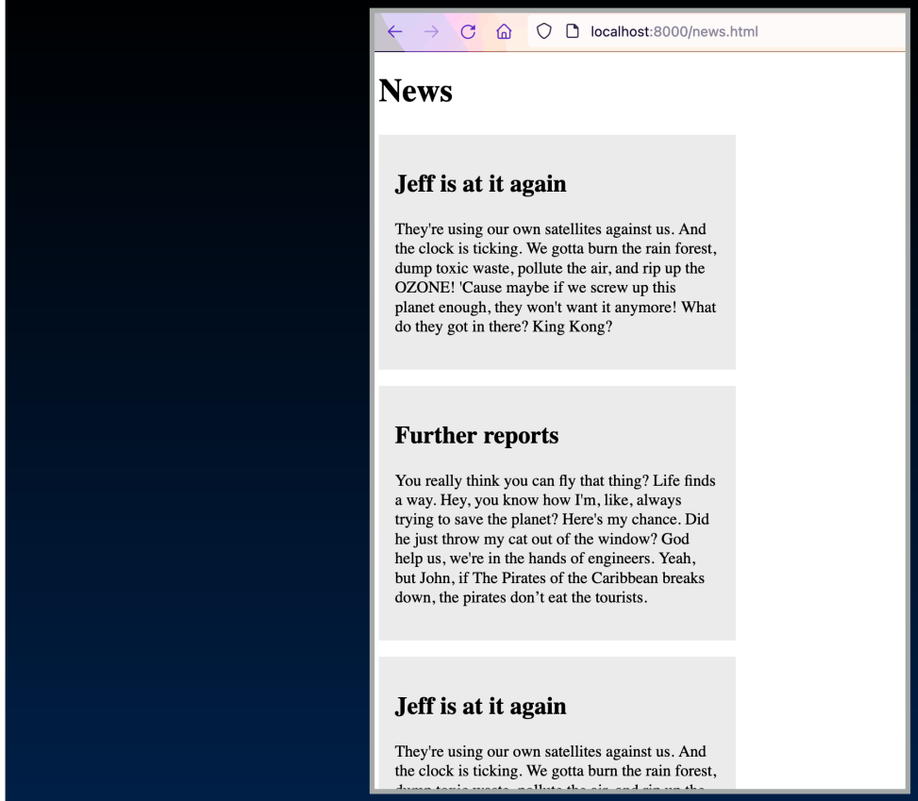

First, we clone the template. It is a clone of this kind that we can append to the DOM so that it becomes visible.

```
const newsList = document.querySelector("#newsList");
const template = document.querySelector("#newsTemplate");

data.forEach((item) => {
  const clone = template.content.cloneNode(true);
  clone.querySelector("h2").textContent = item.headline;
  clone.querySelector("p").textContent = item.content;

  newsList.appendChild(clone);
});
```

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.

```js
const newsList = document.querySelector("#newsList");
const template = document.querySelector("#newsTemplate");

data.forEach((item) => {
  const clone = template.content.cloneNode(true);
  clone.querySelector("h2").textContent = item.headline;
  clone.querySelector("p").textContent = item.content;

  newsList.appendChild(clone);
});
```

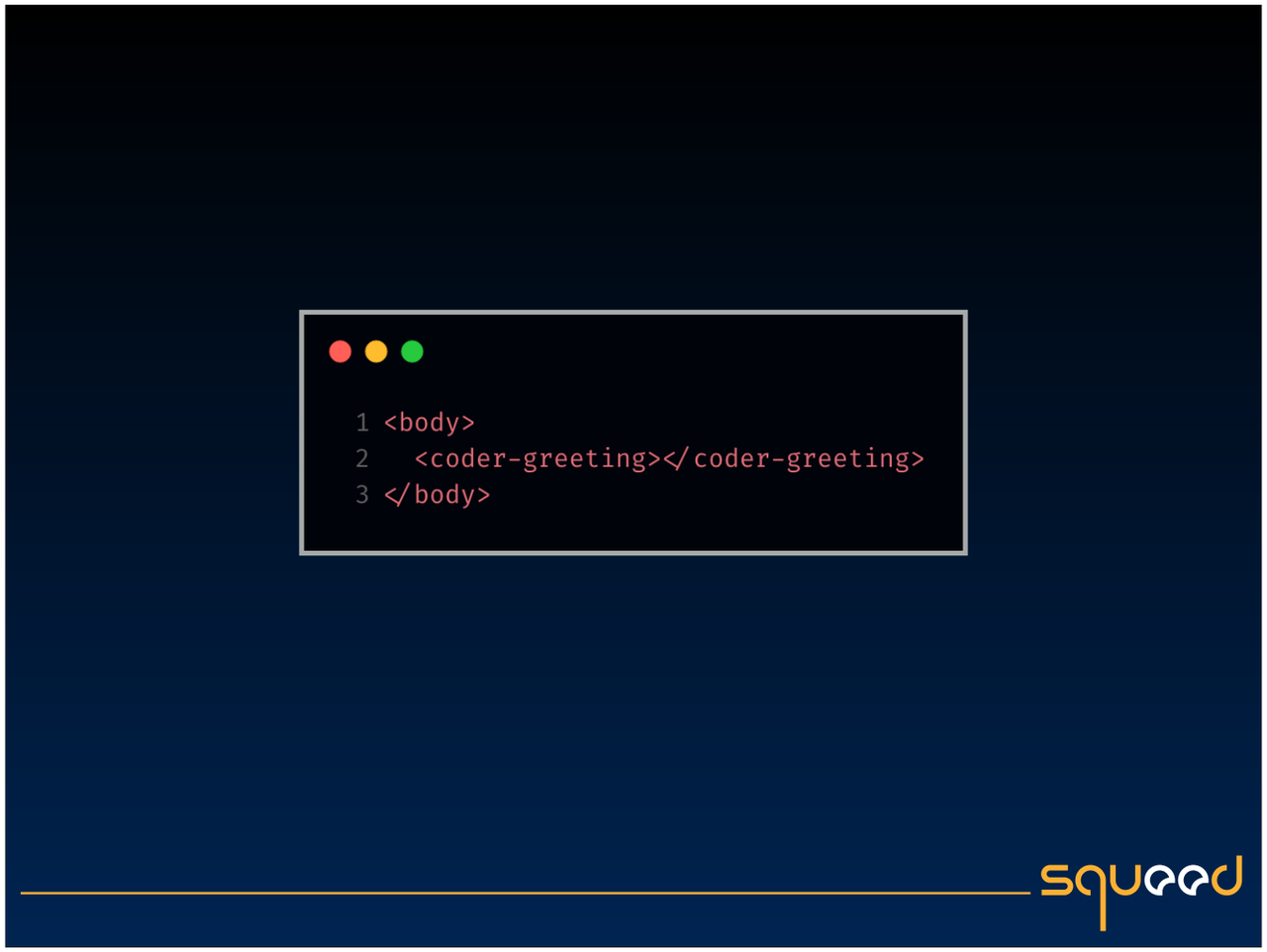

And then finally, we append the clone to the list element. We do all this for each data item and...



...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.

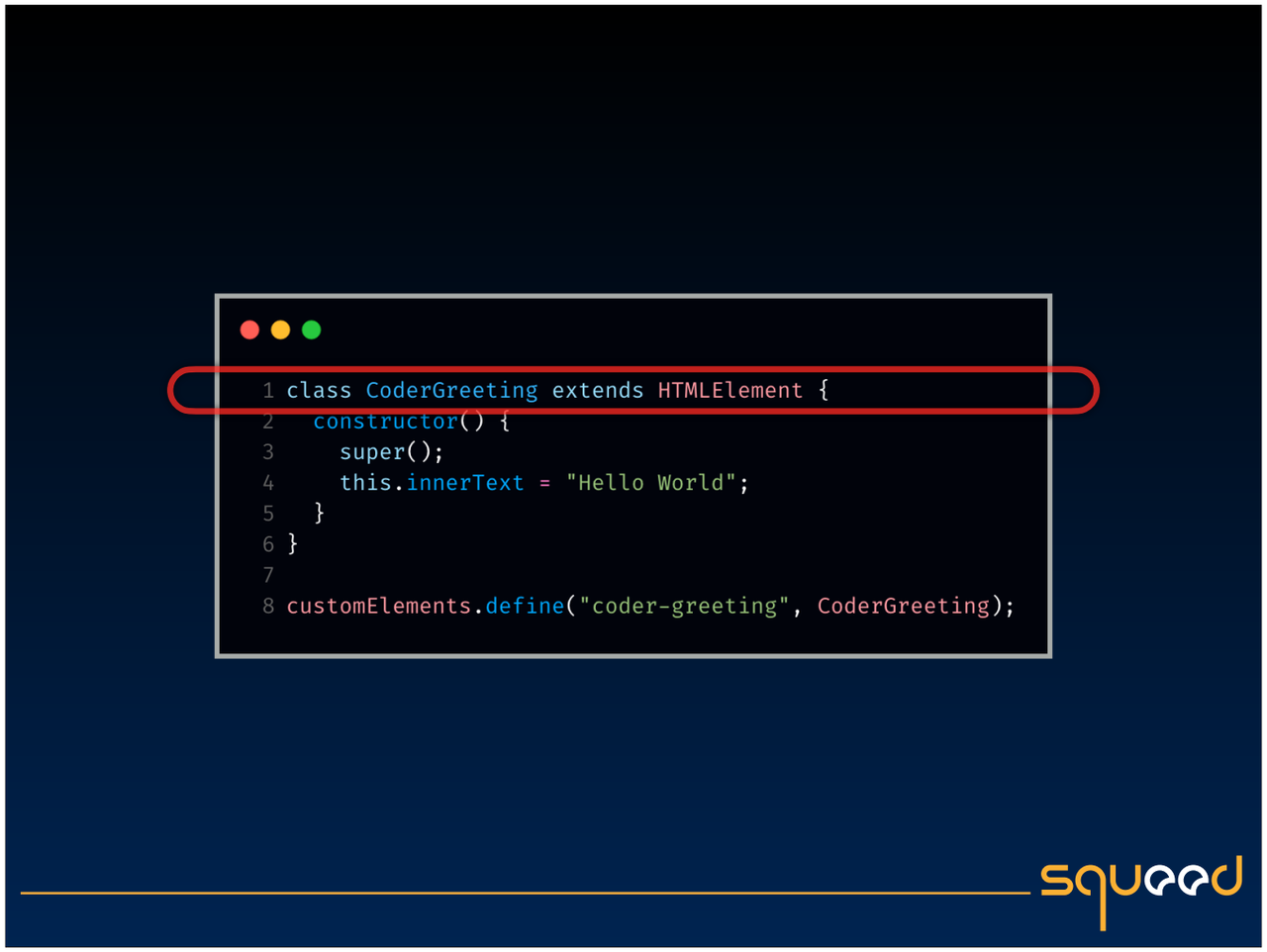
Custom elements

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

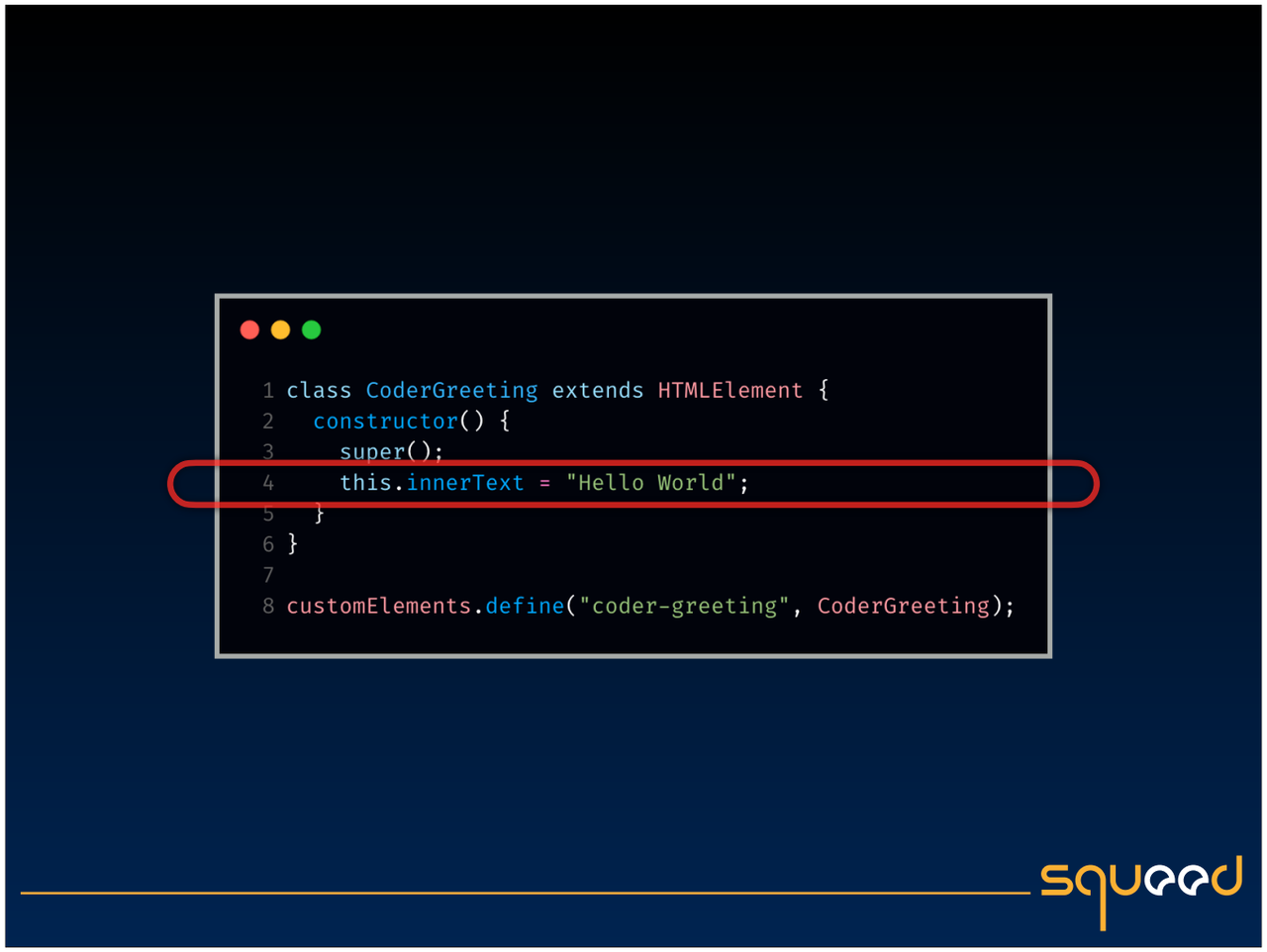


If you take a html document and write a tag that doesn't exist in the specification, like this tag here "coder-greeting", you get nothing. If you put some text content in it, the browser might treat it as a div or something, I think that depends on the browser, but it's invalid html, and should not be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

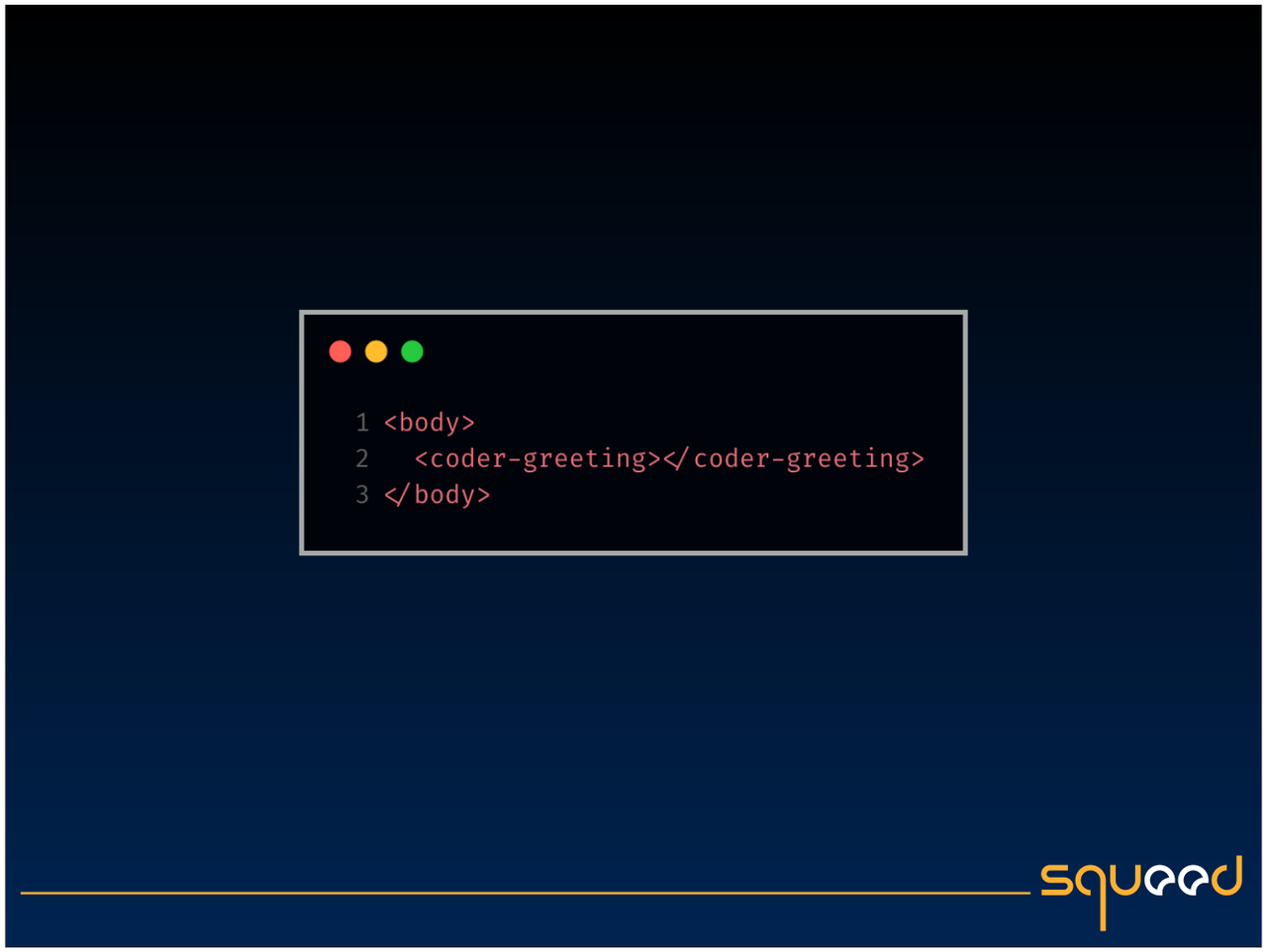
Creating a custom element is done with JavaScripts class syntax. You use the custom elements registry to make a new custom element, which is what is done down at the end of this script. Here we give the define-method a string that later will allow us to write a HTML tag with that name, and we give it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



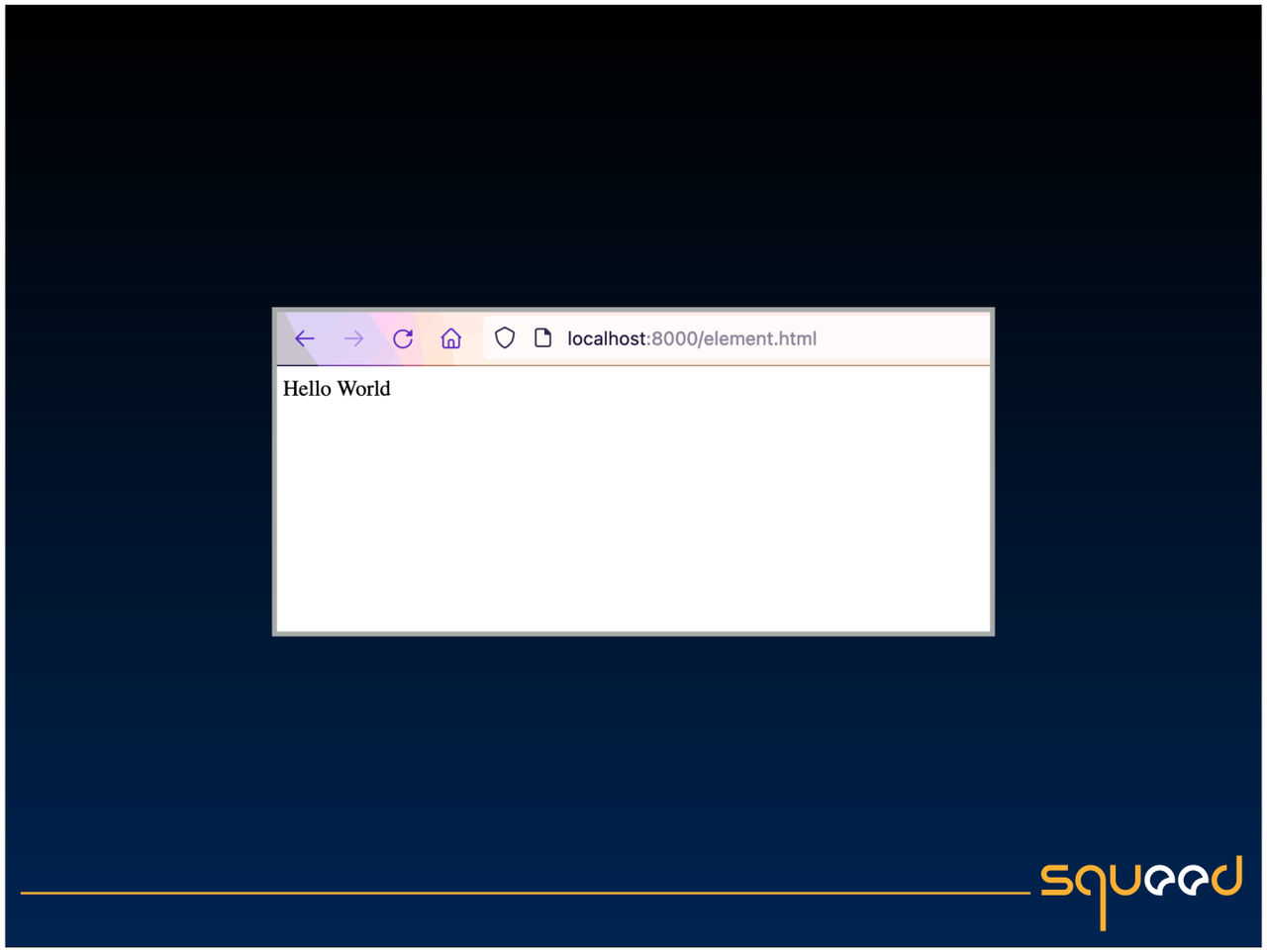
Because, it is with JavaScripts class syntax that we specify what our custom element shall do. We create a class that first of all extends the built in class HTMLElement.



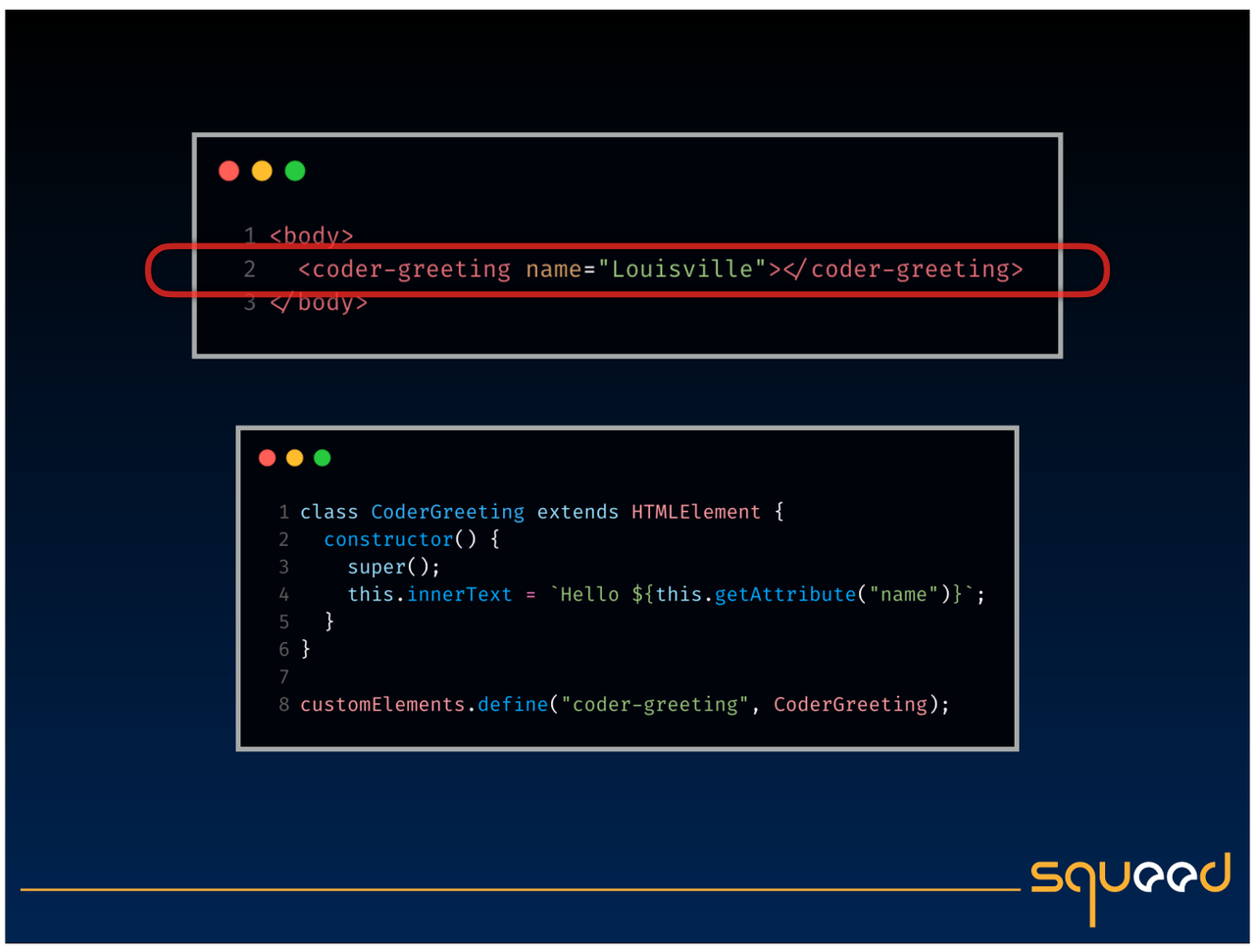
Then, in the constructor - after calling the constructor of the class we extend, which we always must do - we set our custom elements inner text to "Hello world".



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.

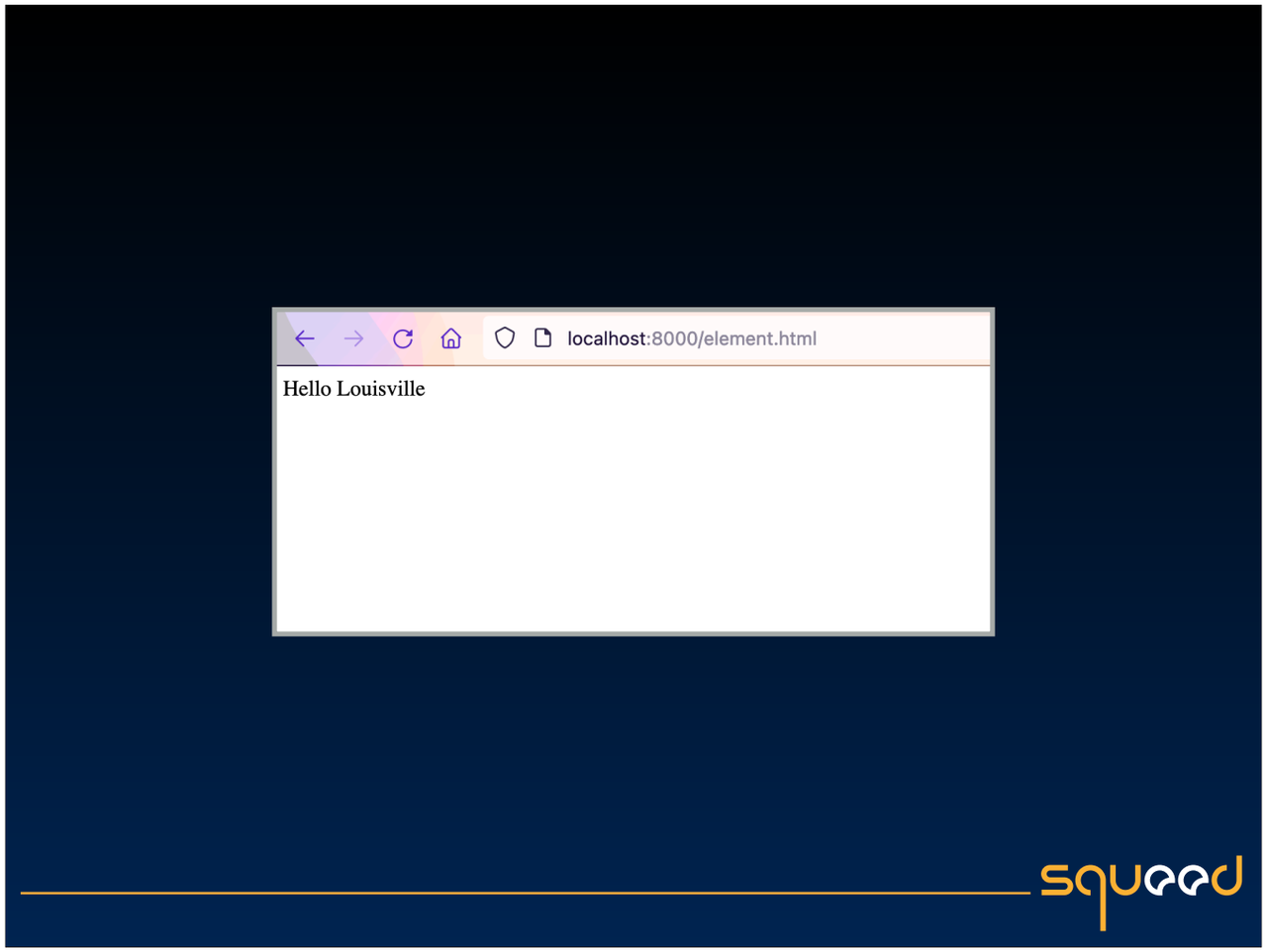


Let's also take a quick look at attributes. Any attribute put on the tag in the HTML can be retrieved in the code with the getAttribute method, and used in the element. If we put the attribute "name" on the HTML tag...

```
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);
```

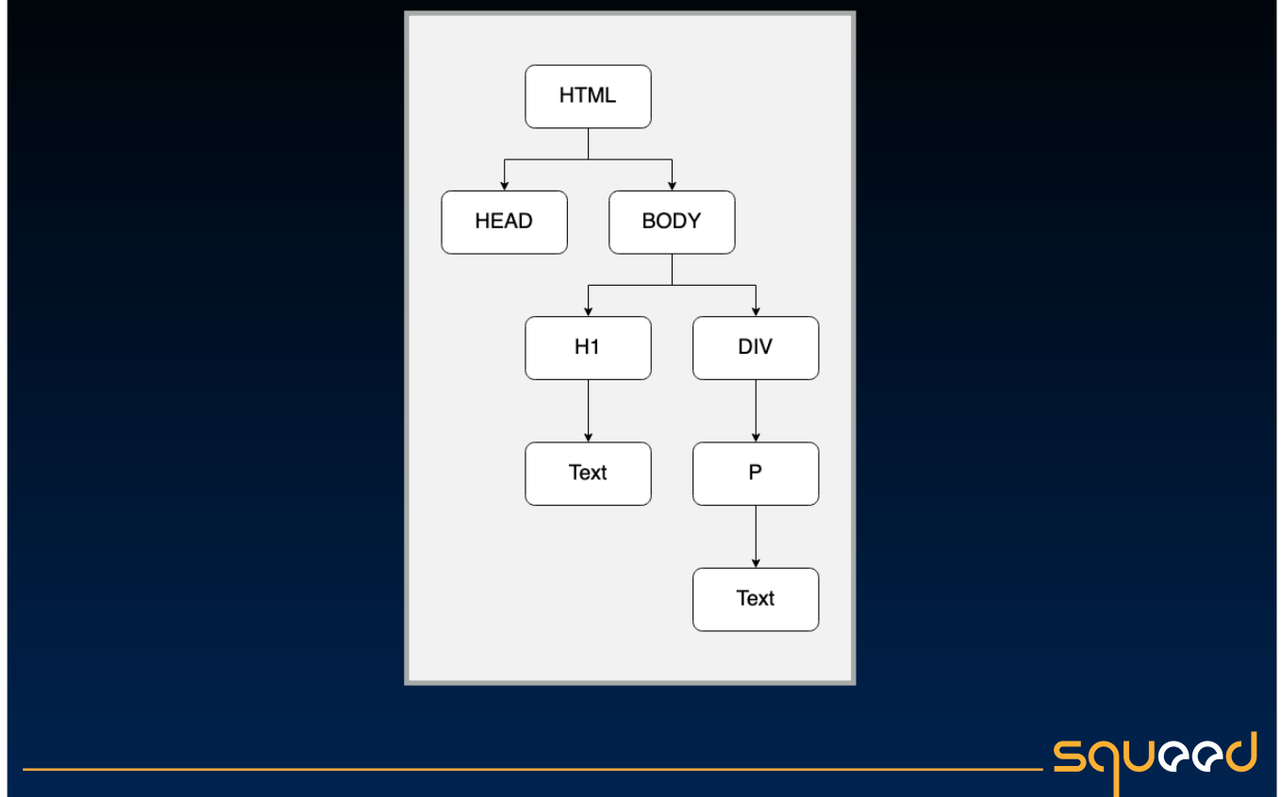
...we can then retrieve it in the component class and make it part of the elements inner text.



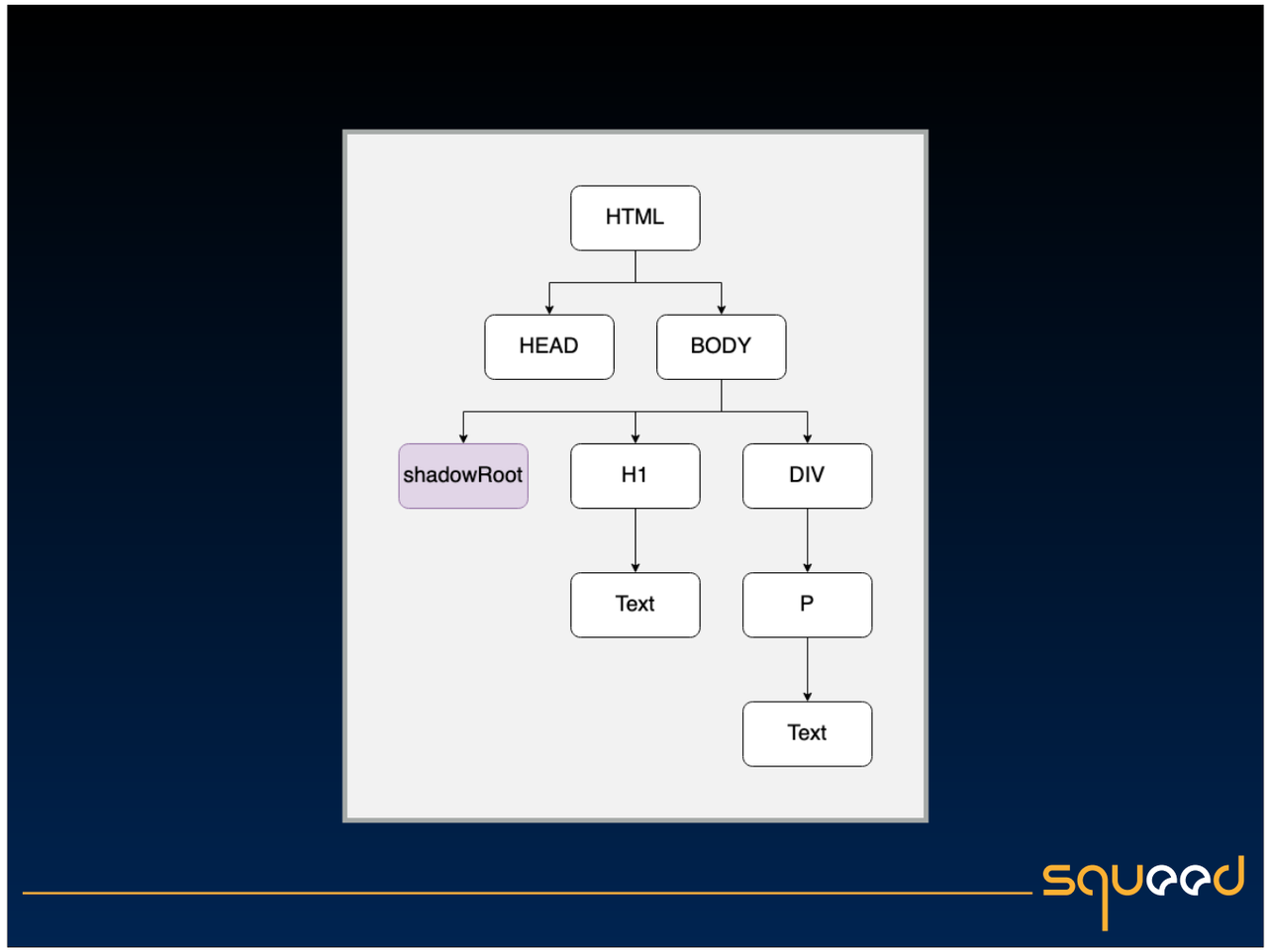
And so we get Hello Louisville!

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.

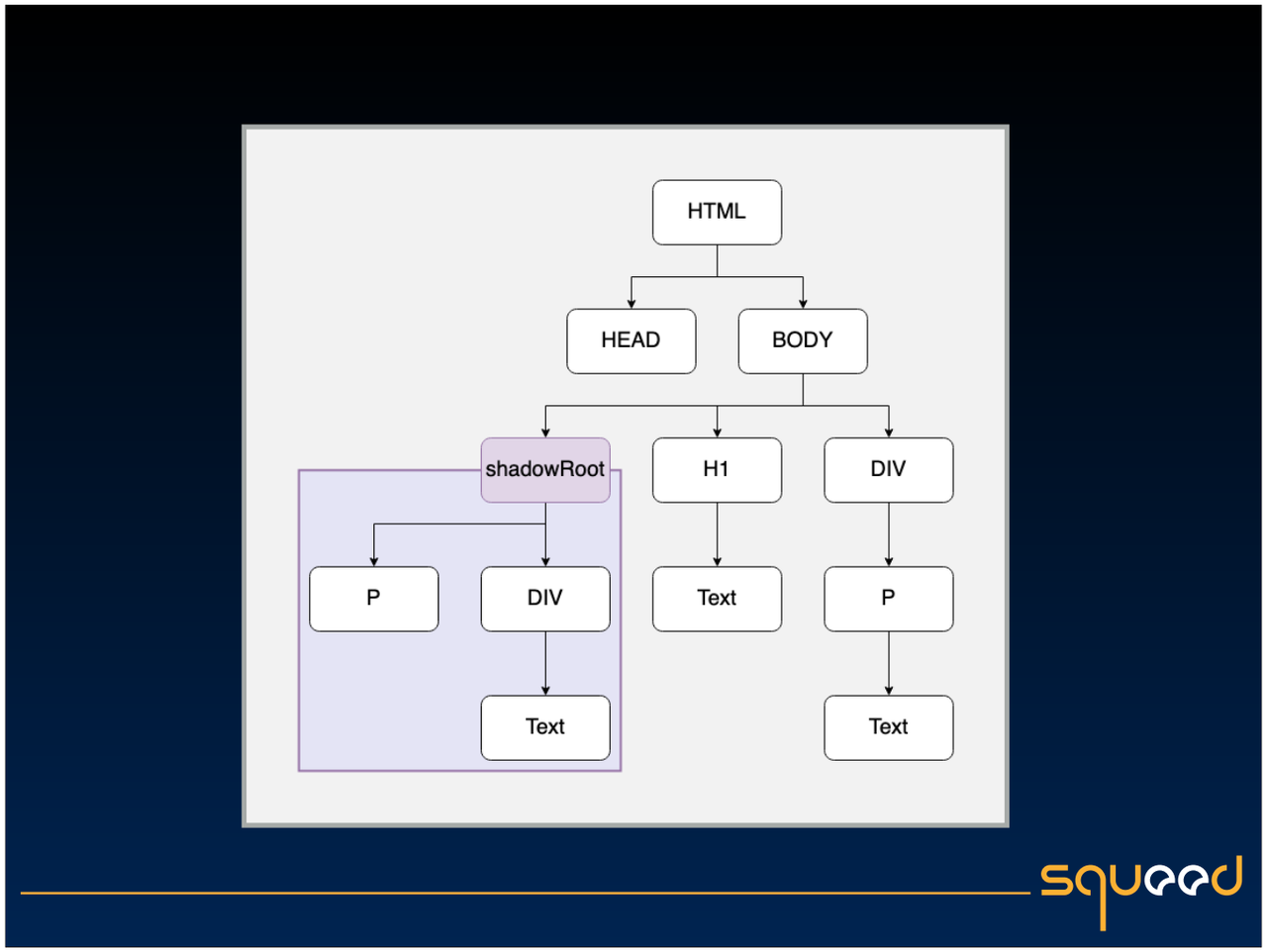
Finally, the third api that powers web components is the shadow dom. Now, I don't know if it's the name of it or what, but I've gotten the impression that this is the most difficult, and the scariest of the concepts, and the hardest one to grasp. But I don't think it's very complicated really. Let's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



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.

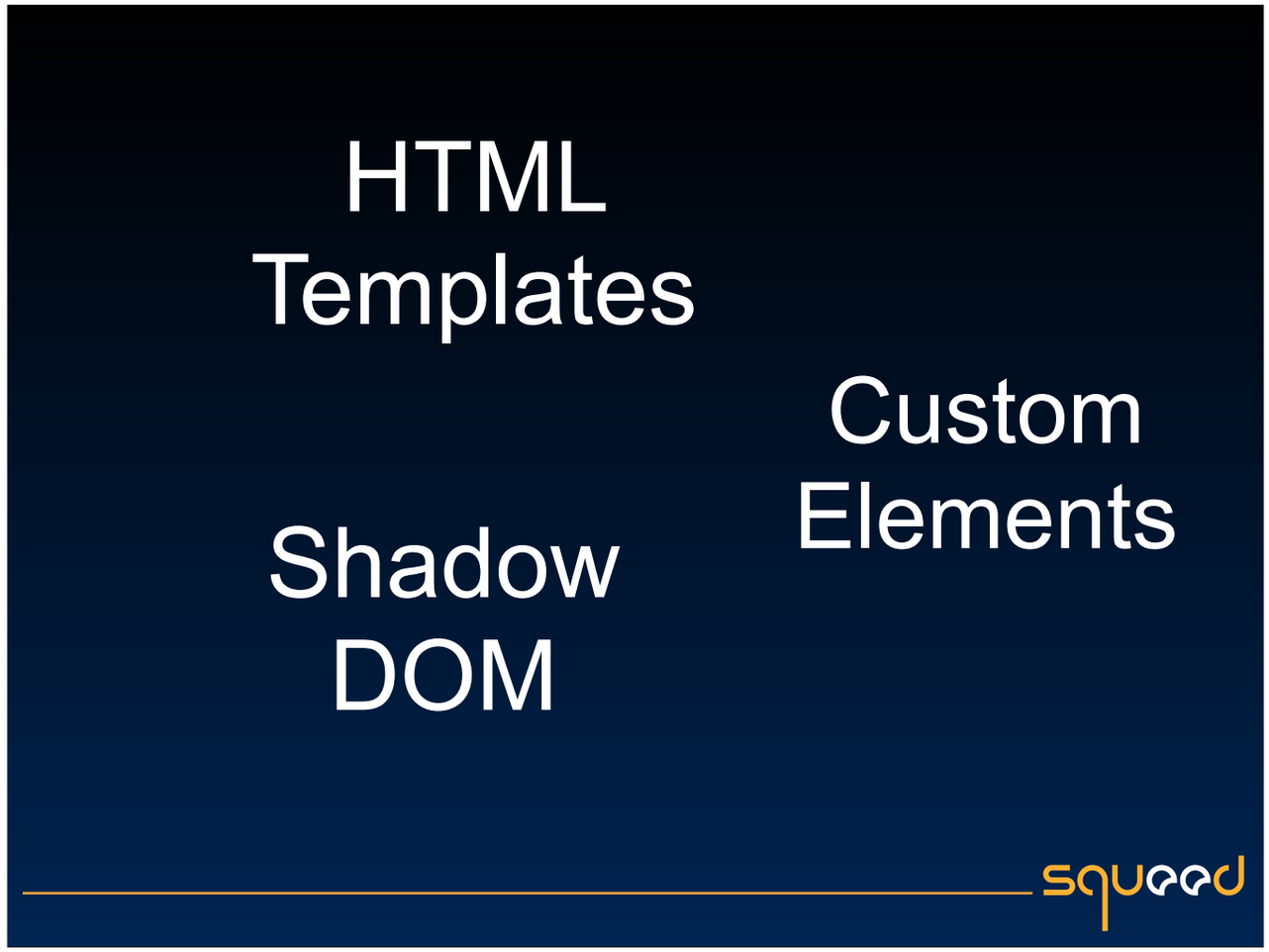


Attaching a shadow root to the body element would make our tree look like this. Now this doesn't affect the tree in any way yet, but we can now attach more, regular nodes to the shadow root.



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.



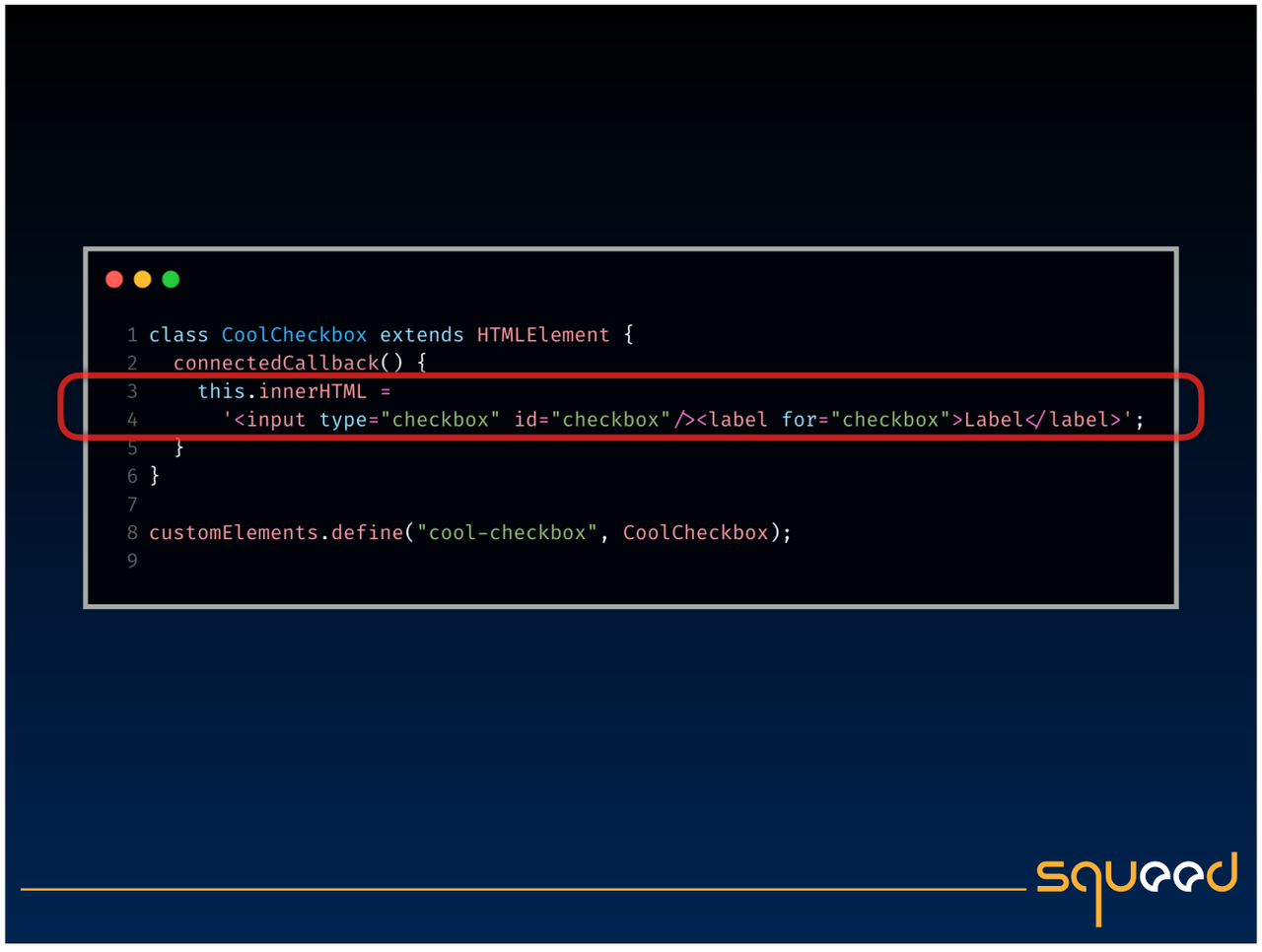
Alright, so now we've looked at HTML templates, custom elements, and the shadow dom. With our powers combined! We now have the ability to...

...code a web component!

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.

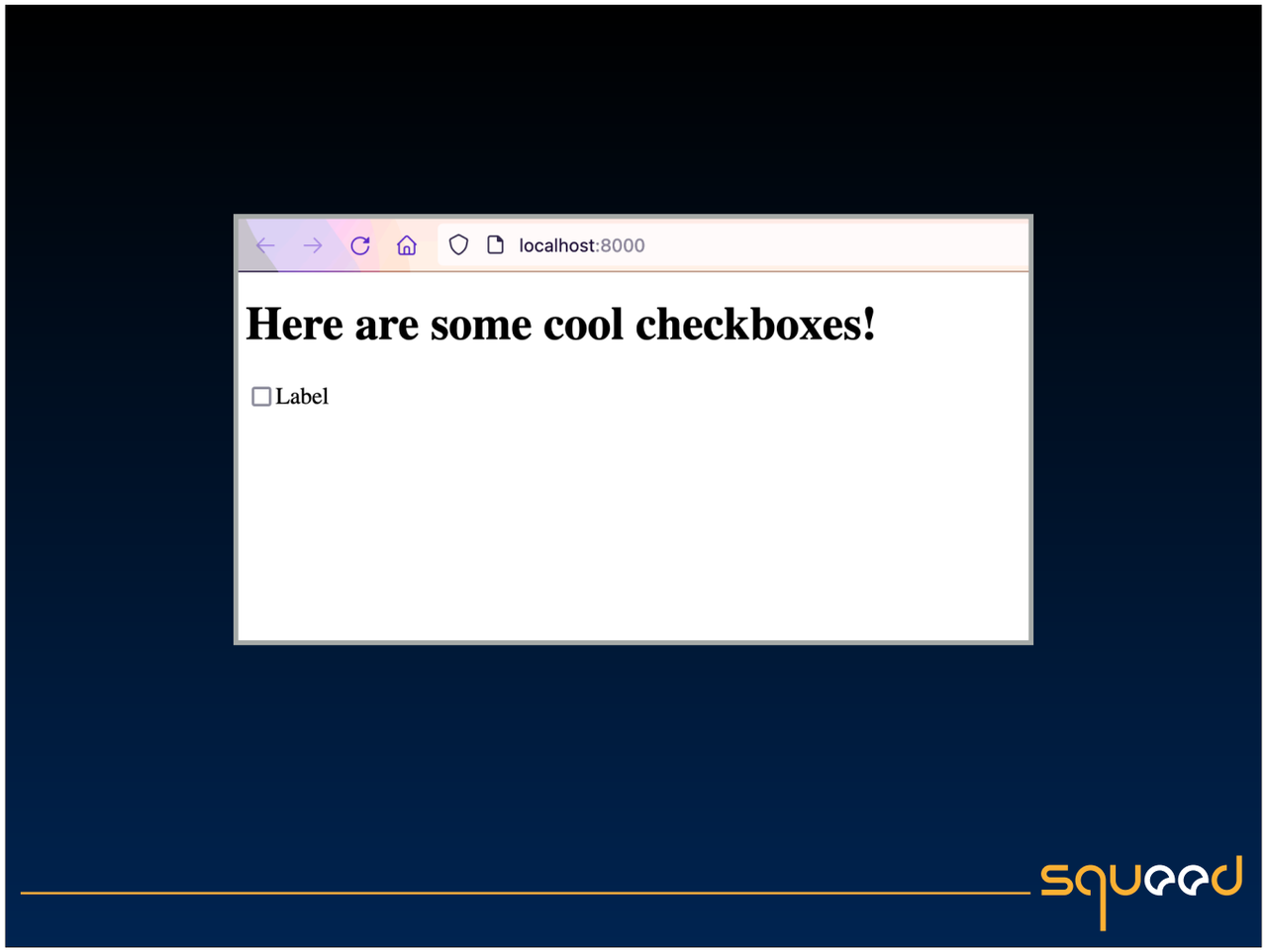
Next we define a function called connectedCallback. You might remember before, we specified the components content in the constructor. Now, a problem with doing that is that it won't work if the element is appended to the DOM before it has been defined. This might be something you should avoid anyway, 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.



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

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...

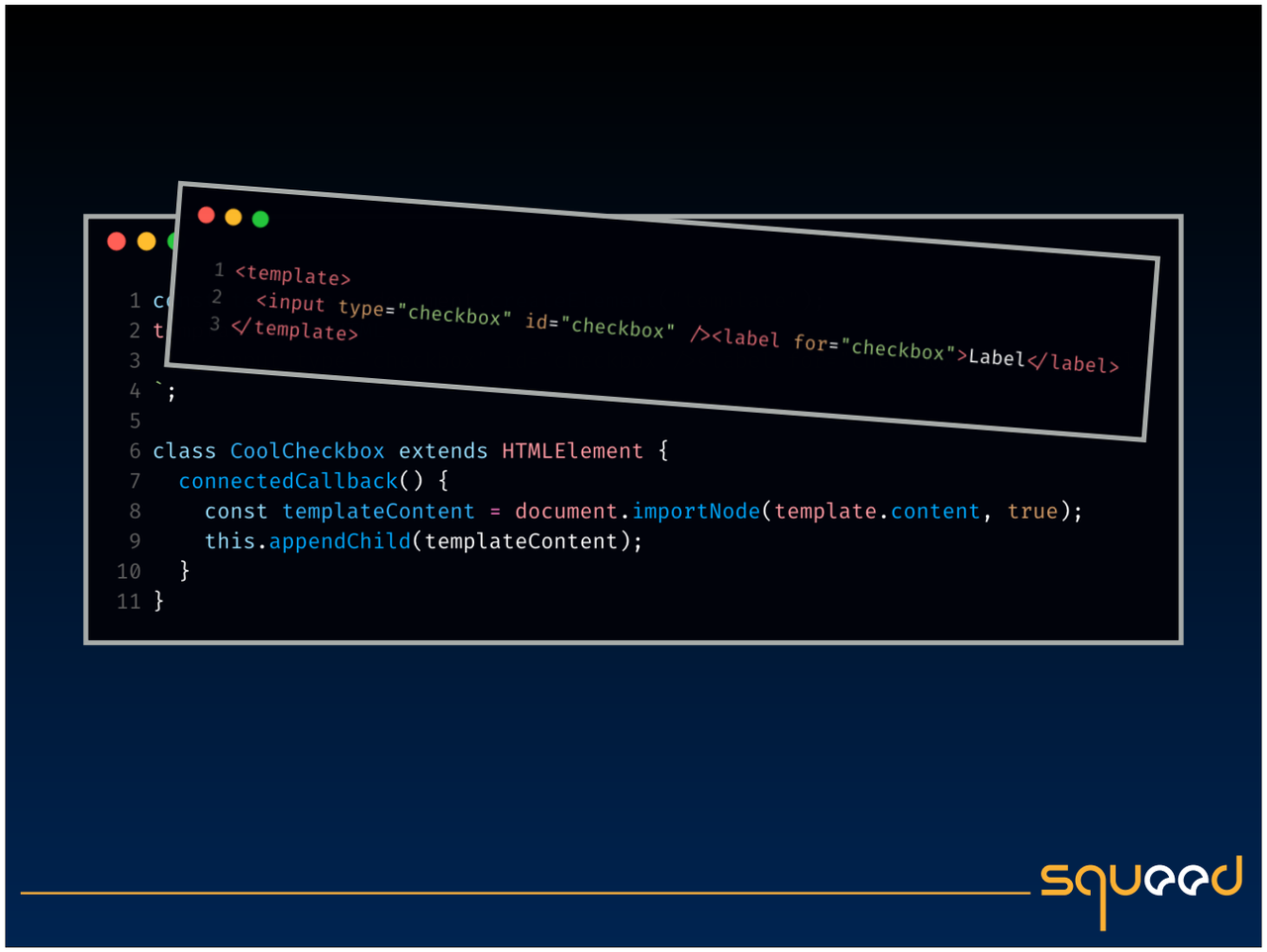
Use a template

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

```
const template = document.createElement("template");
template.innerHTML = `
    <input type="checkbox" id="checkbox"/><label for="checkbox">Label</label>
`;

class CoolCheckbox extends HTMLElement {
  connectedCallback() {
    const templateContent = document.importNode(template.content, true);
    this.appendChild(templateContent);
  }
}
```

squeed

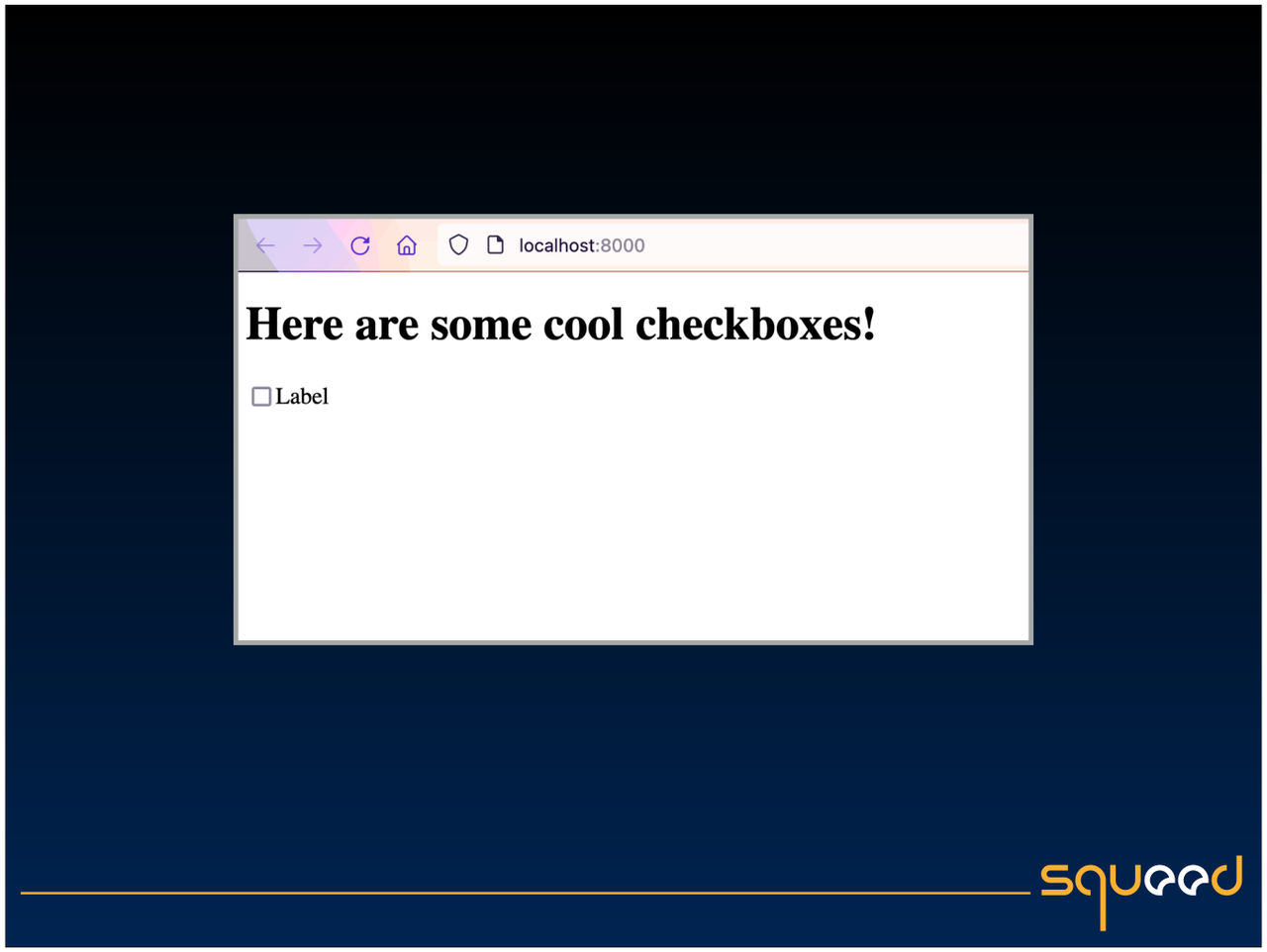
Before, we defined a template declaratively in the HTML file...



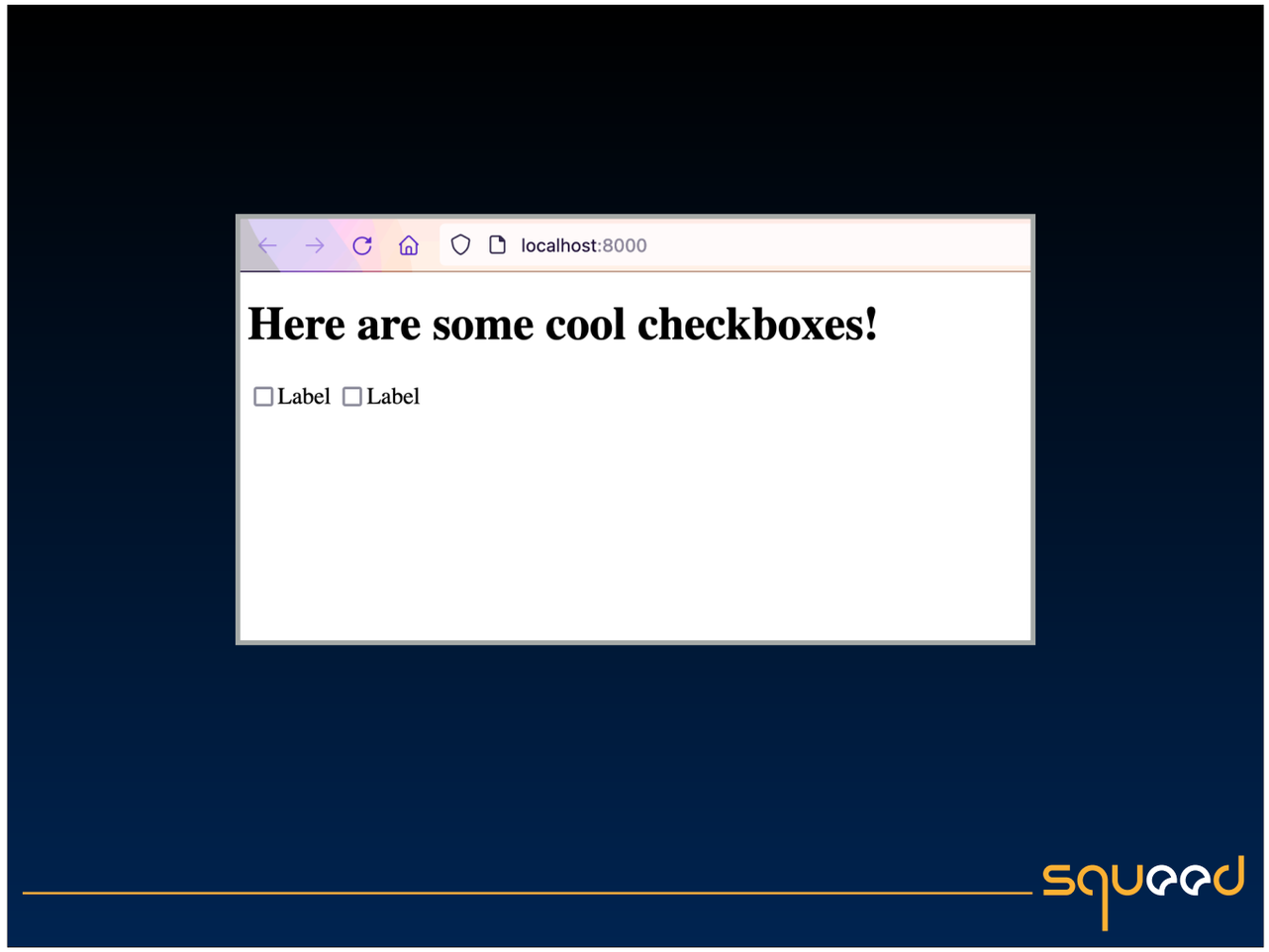
...like this, you remember? But we can just as easily do it imperatively in JavaScript.

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

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.

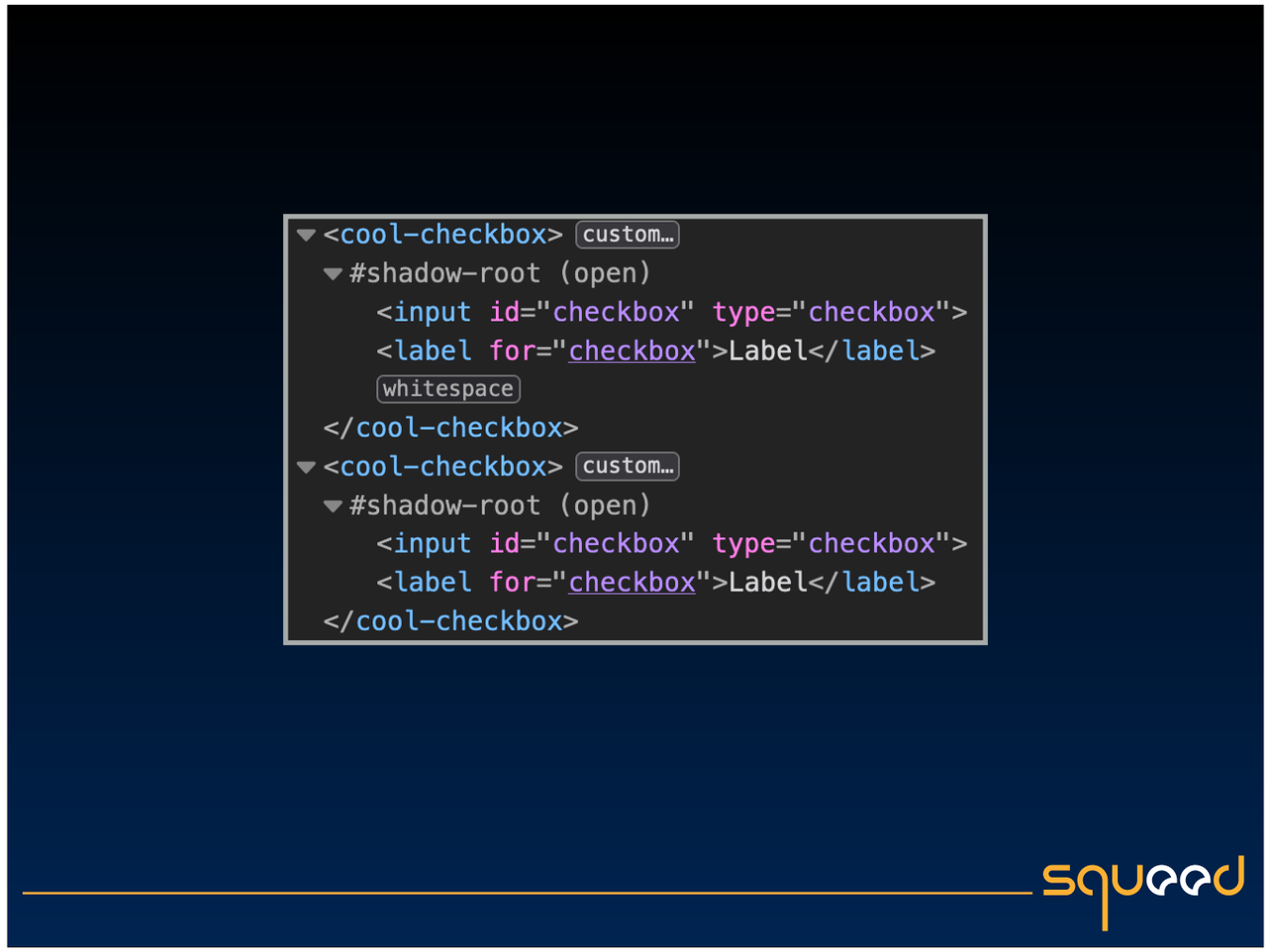


However, the advantage is that if we put multiple instances of our component on a page, the template will only have to be parsed once, and then it can be reused via cloning, so that the more copies you have on a page, and the more complex they are, the bigger the performance gain.

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!

This is done in the constructor, and it's no harder than taking 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.

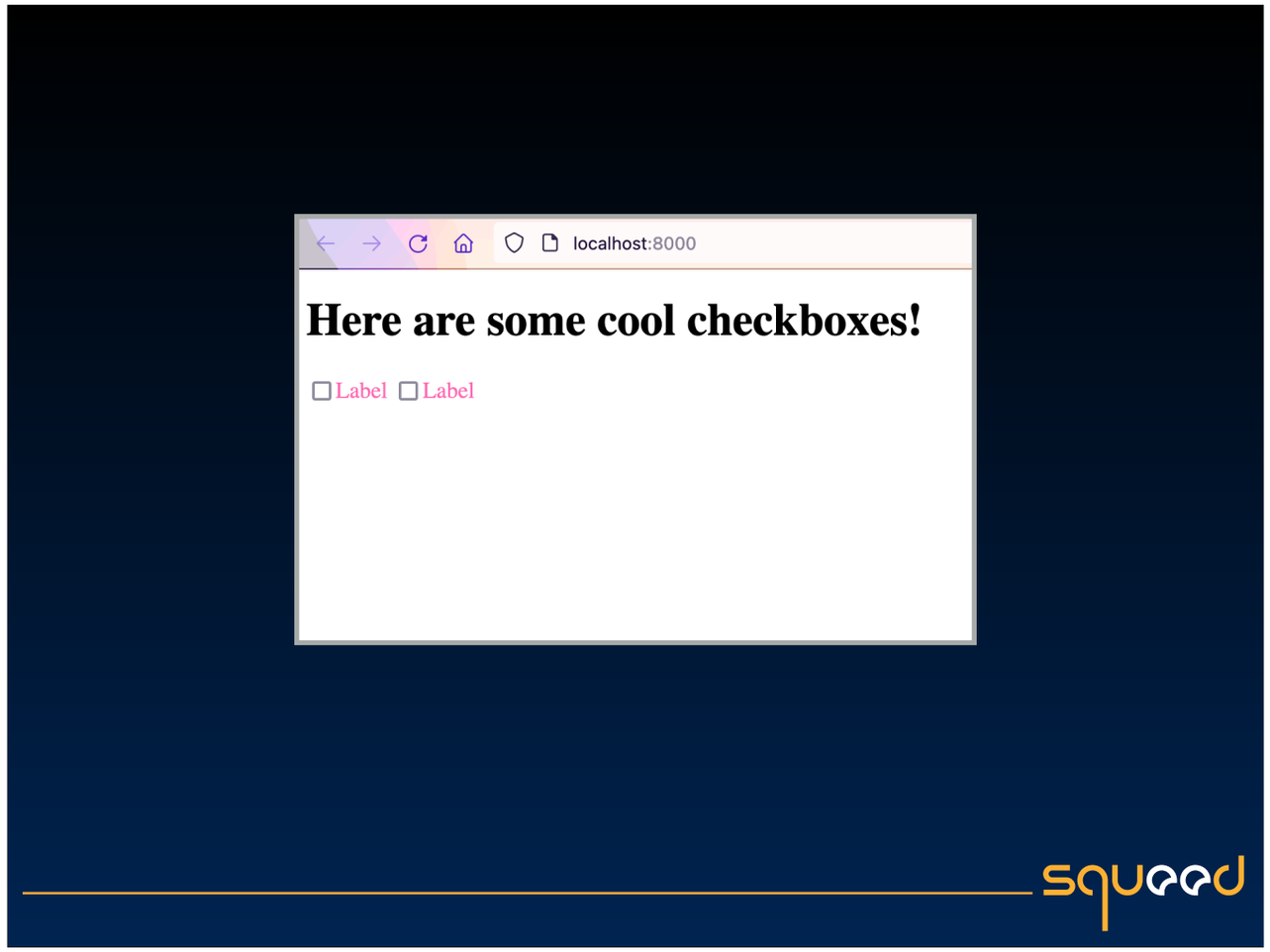
Something that we immediately do in the connectedCallback function. Here, instead of appending the cloned template content to the element itself (as before), we append it to its shadow root instead. And now, we have a shadow DOM. All of the HTML content from the template that was appended to the shadow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



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.

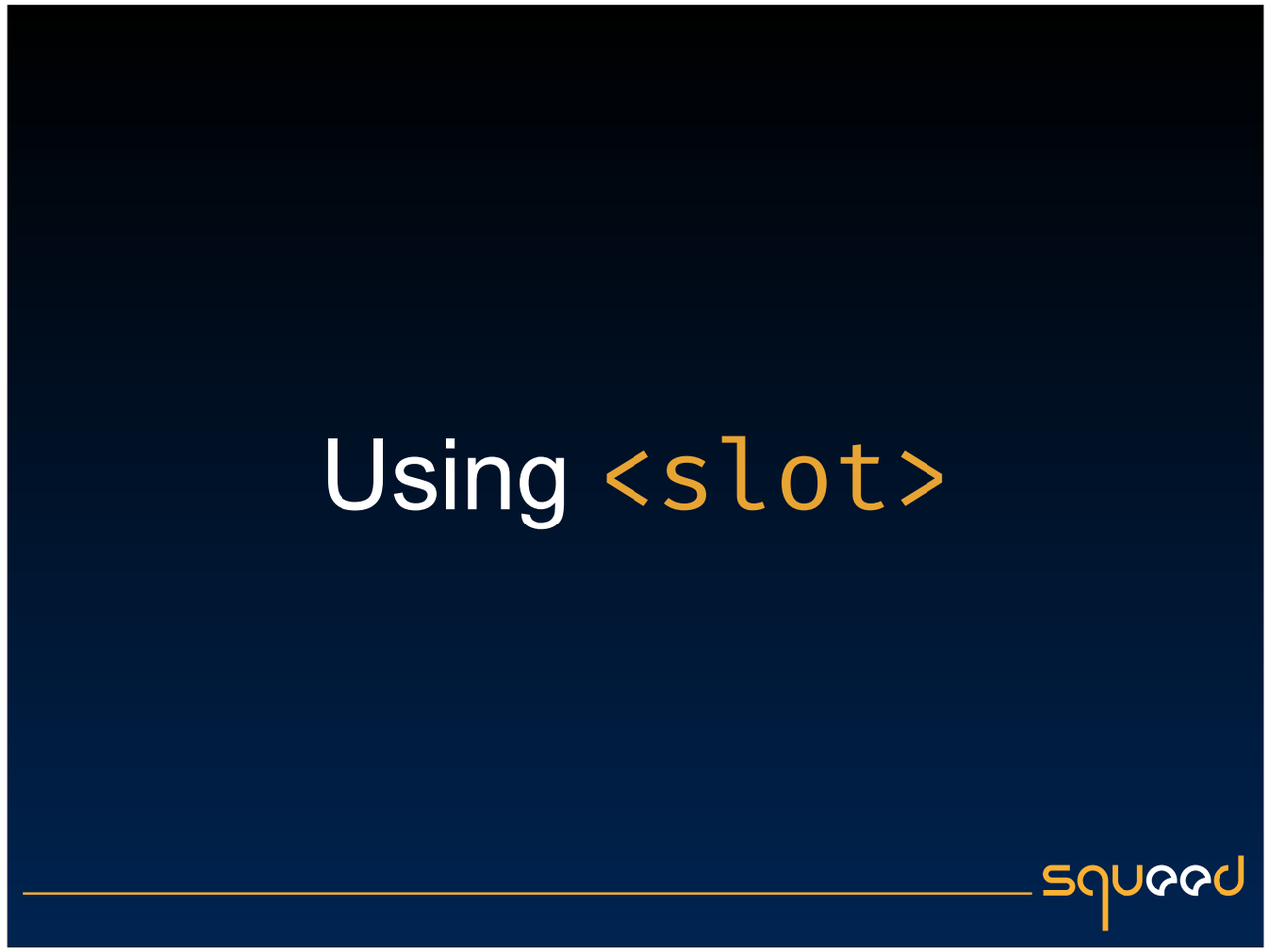
Adding some styles to our checkbox is as easy as adding it to the template. We put a style tag here, and some CSS inside - and as you may notice we're using a very broad selector: label. If this was in the light DOM, we might not want to use such broad selectors, but instead use classes and ids to target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

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.

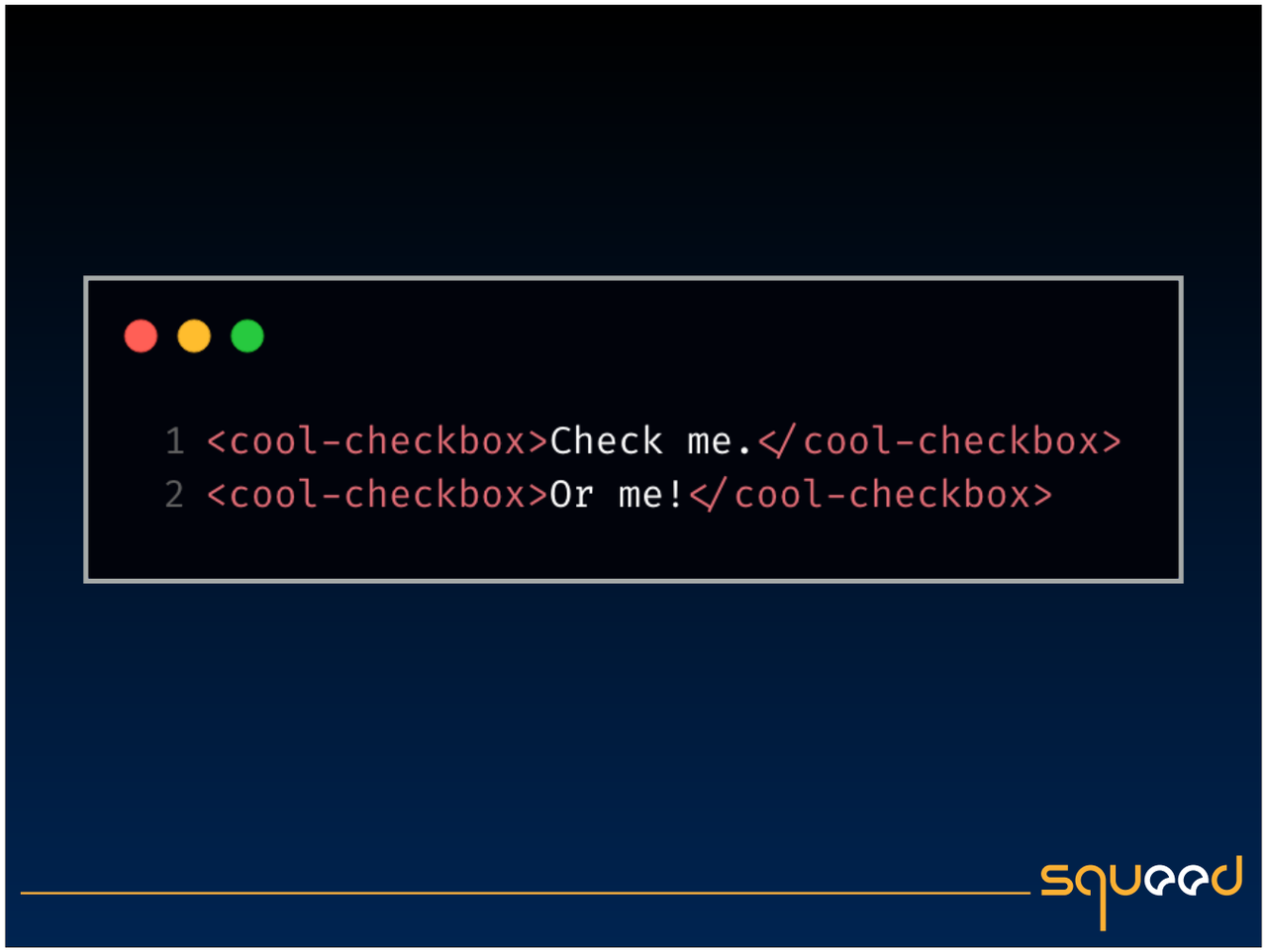
Ok, so a web component can do a lot when it comes to deciding its content, but it wouldn't be very versatile if the user couldn't affect the content and the appearance of the component. And there are multiple ways to do this.



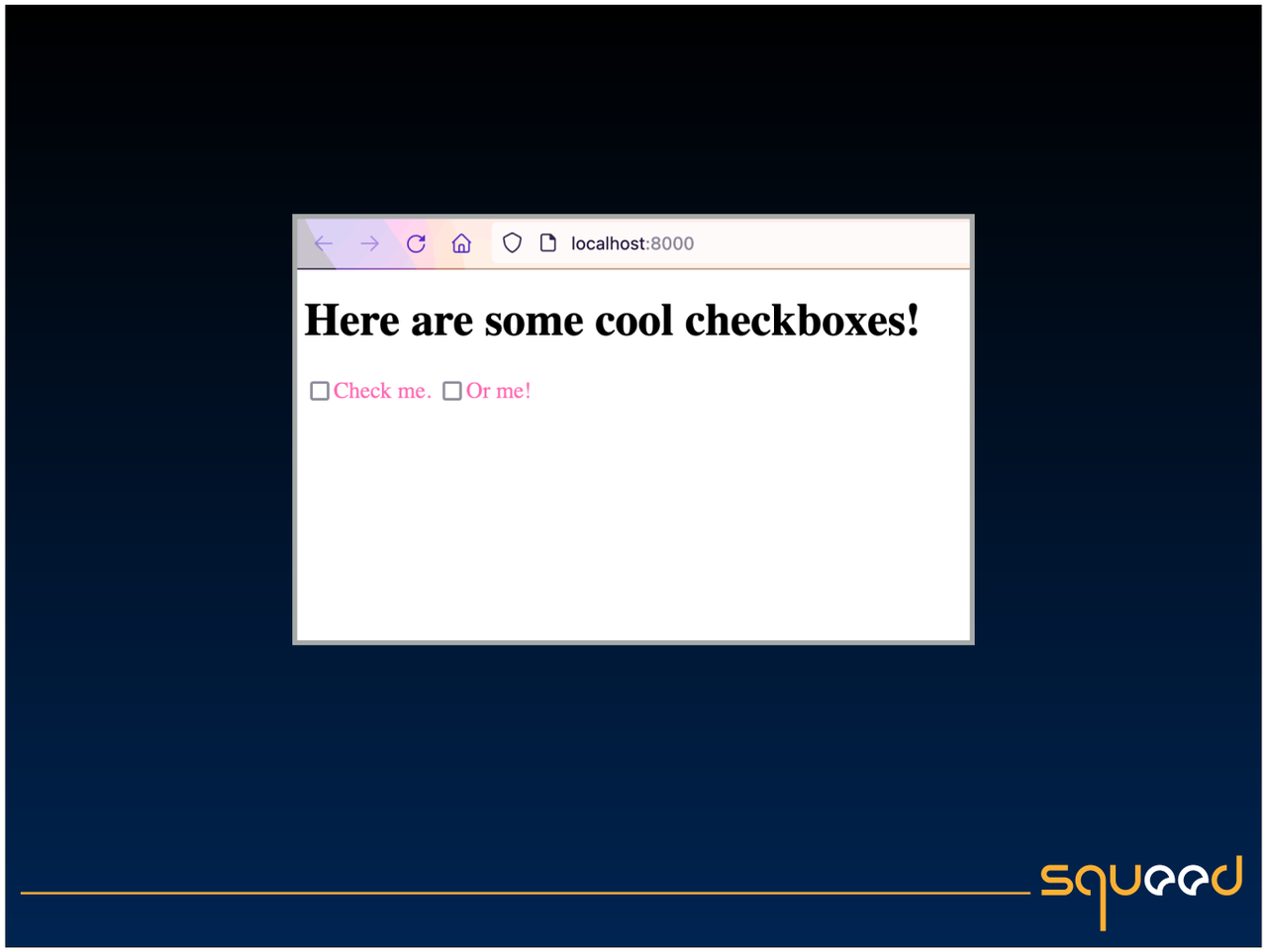
One way is using slots. This is a concept that will be very familiar to you if you've worked with slots in Vue. It's also not unlike the concept of children in React.

```
const template = document.createElement("template");
template.innerHTML = `
        <style>
            label {
                color: hotpink;
            }
        </style>

        <input type="checkbox" id="checkbox"/><label for="checkbox"><slot></slot></label>
`;
```


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...



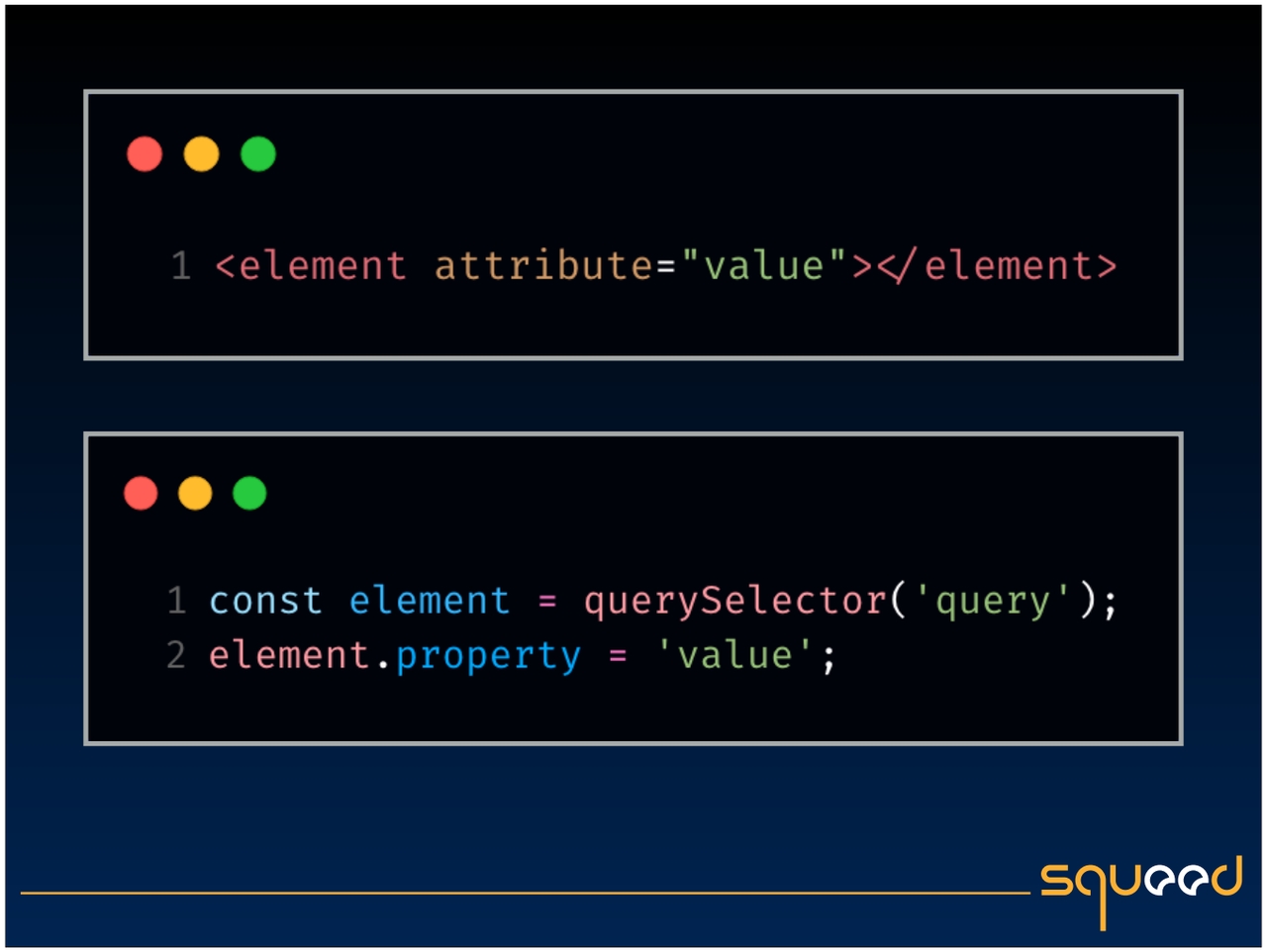
...and then we put some content between the tags of our custom element. It can be text, like this, but it can also be any valid HTML. There's really no limit.



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.

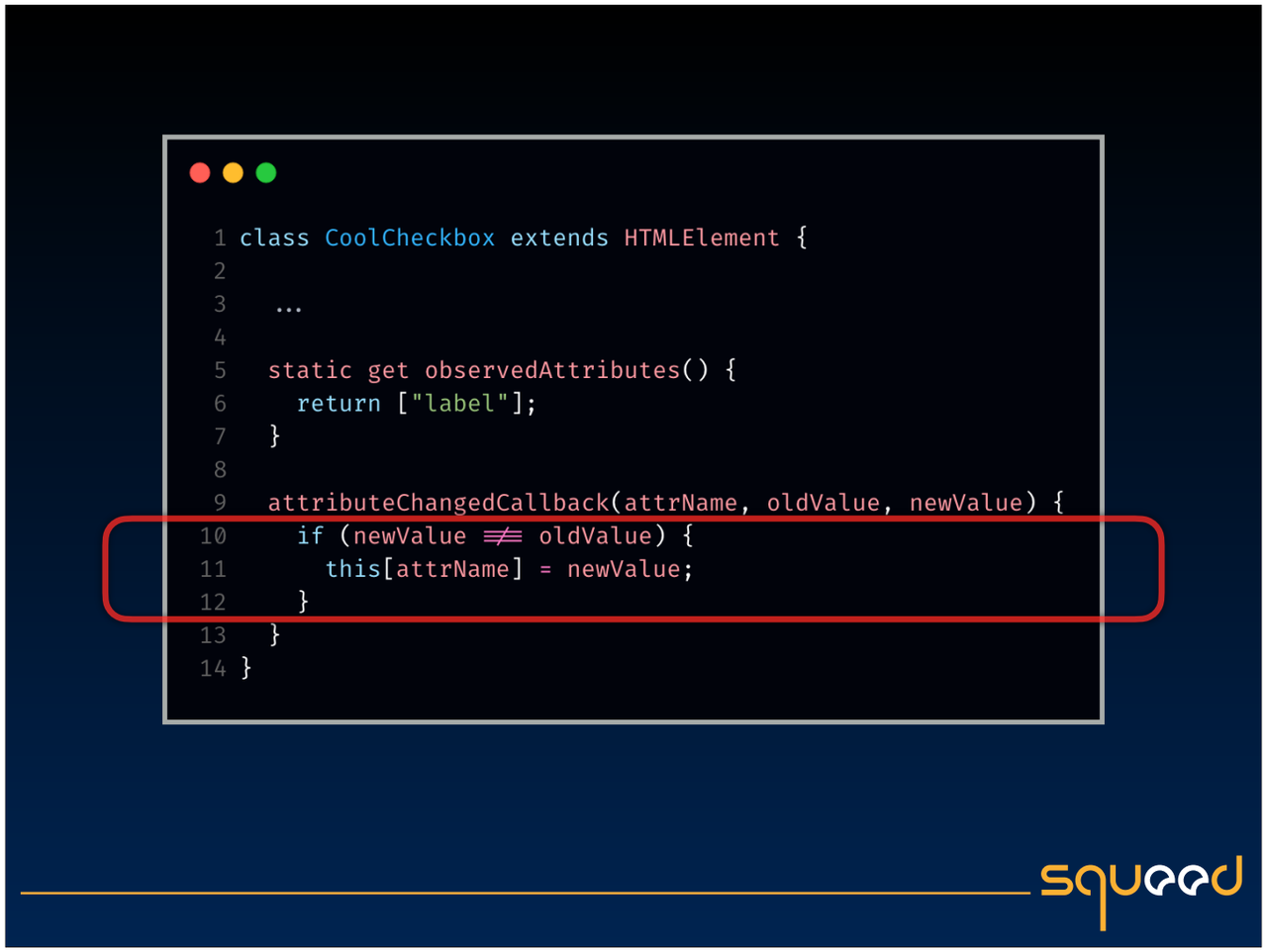


It's important to know the difference between an attribute and a property.
An attribute is what we set on the HTML tag. We can access this attribute with JavaScript, but it will always be a string.
A property is what we have on a JavaScript object. And when it comes to the DOM, each element in HTML is 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.

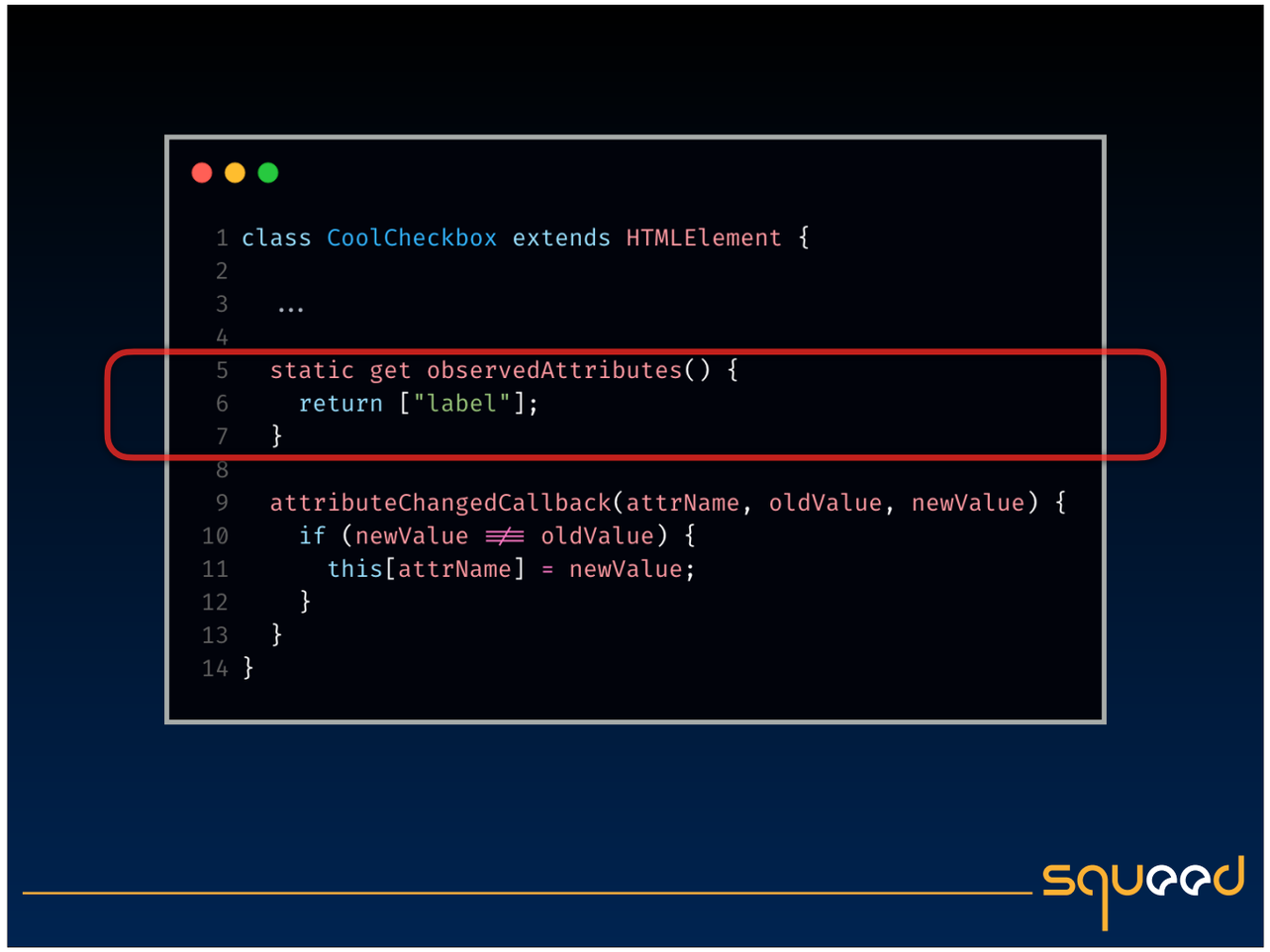
So, let's say we want to set the text of the label via an attribute, instead of a slot. Makes sense in a way, since a label should probably just be a piece of text, right? Well, if we set the attribute on the cool-checkbox tags in the HTML like this, that doesn't mean that we automatically get a property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

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?



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:



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.

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.

```
class CoolCheckbox extends HTMLElement {

  ...

  get label() {
    return this.getAttribute("label");
  }

  set label(newValue) {
    if (newValue ≠ null) {
      this.setAttribute("label", newValue);
      this.shadowRoot.querySelector("label").innerHTML = newValue;
    } else {
      this.removeAttribute("label");
    }
  }
}
```


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.

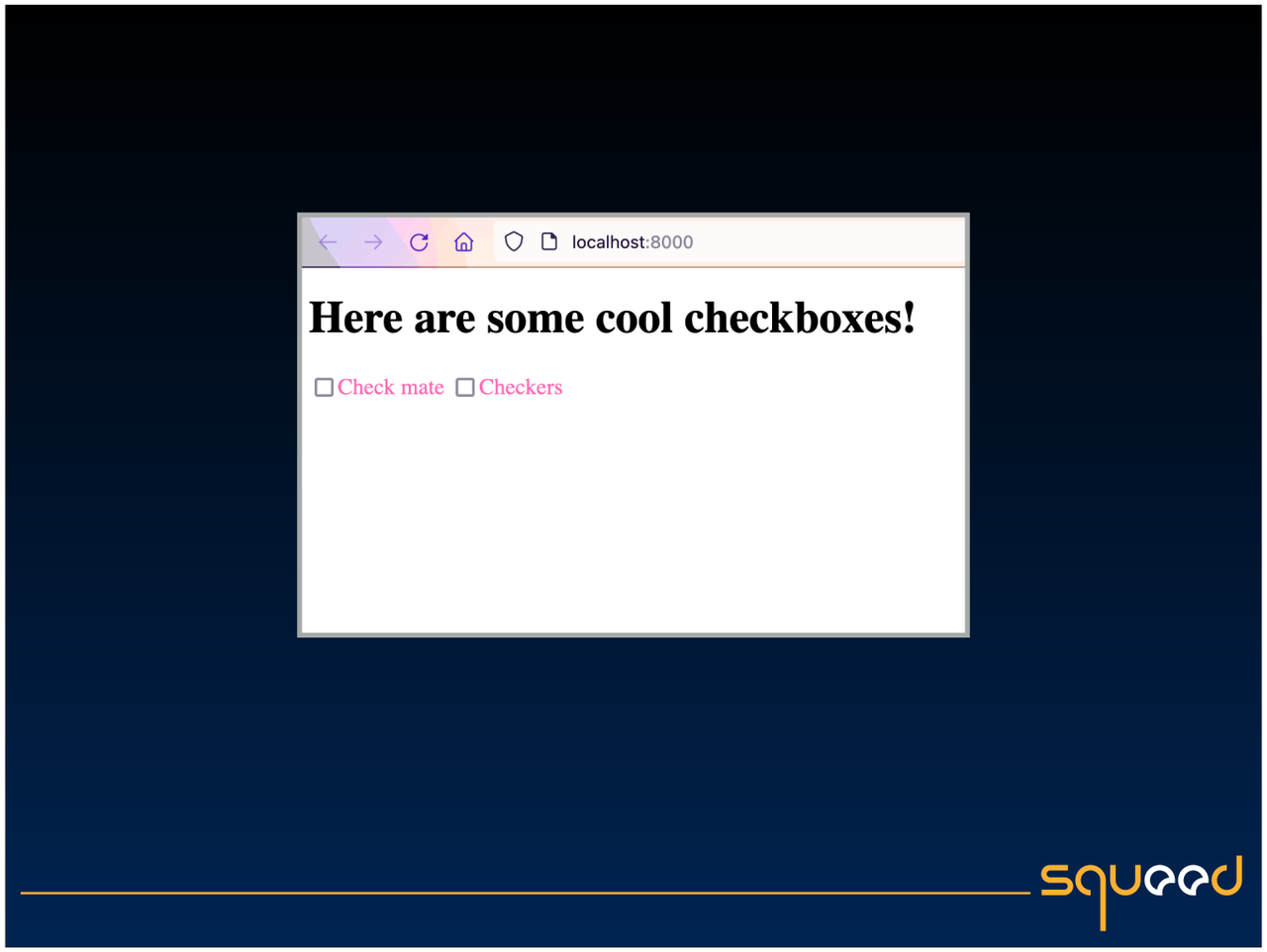
```
class CoolCheckbox extends HTMLElement {

  ...

  get label() {
    return this.getAttribute("label");
  }

  set label(newValue) {
    if (newValue ≠ null) {
      this.setAttribute("label", newValue);
      this.shadowRoot.querySelector("label").innerHTML = newValue;
    } else {
      this.removeAttribute("label");
    }
  }
}
```

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

```
class CoolCheckbox extends HTMLElement {

  ...

  get label() {
    return this.getAttribute("label");
  }

  set label(newValue) {
    if (newValue ≠ null) {
      this.setAttribute("label", newValue);
      this.shadowRoot.querySelector("label").innerHTML = newValue;
    } else {
      this.removeAttribute("label");
    }
  }
}
```

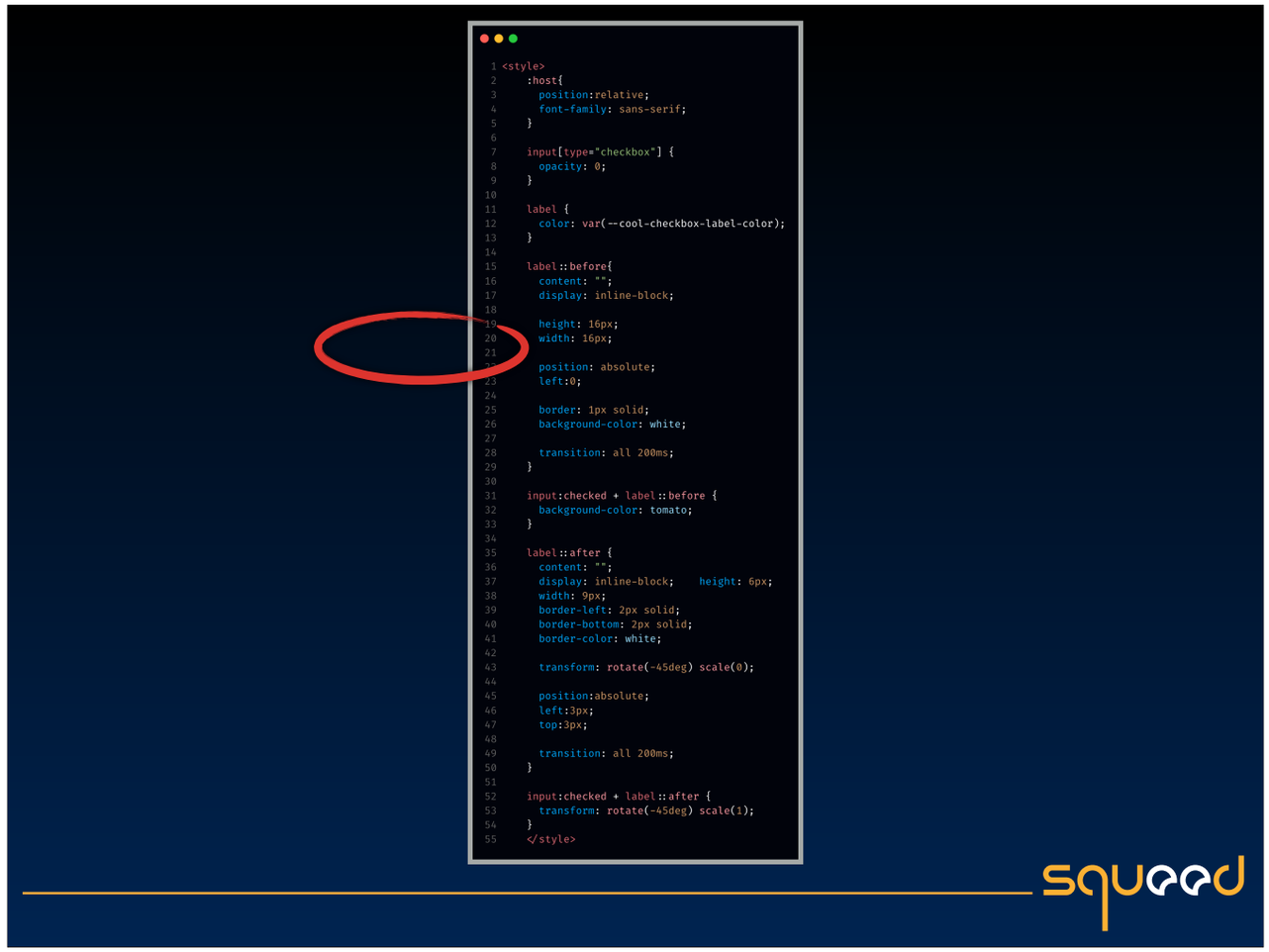

If the label property does indeed have a value, then we set the label attribute to this value, and we also update the inner HTML, so that the change appears visually in the browser.

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

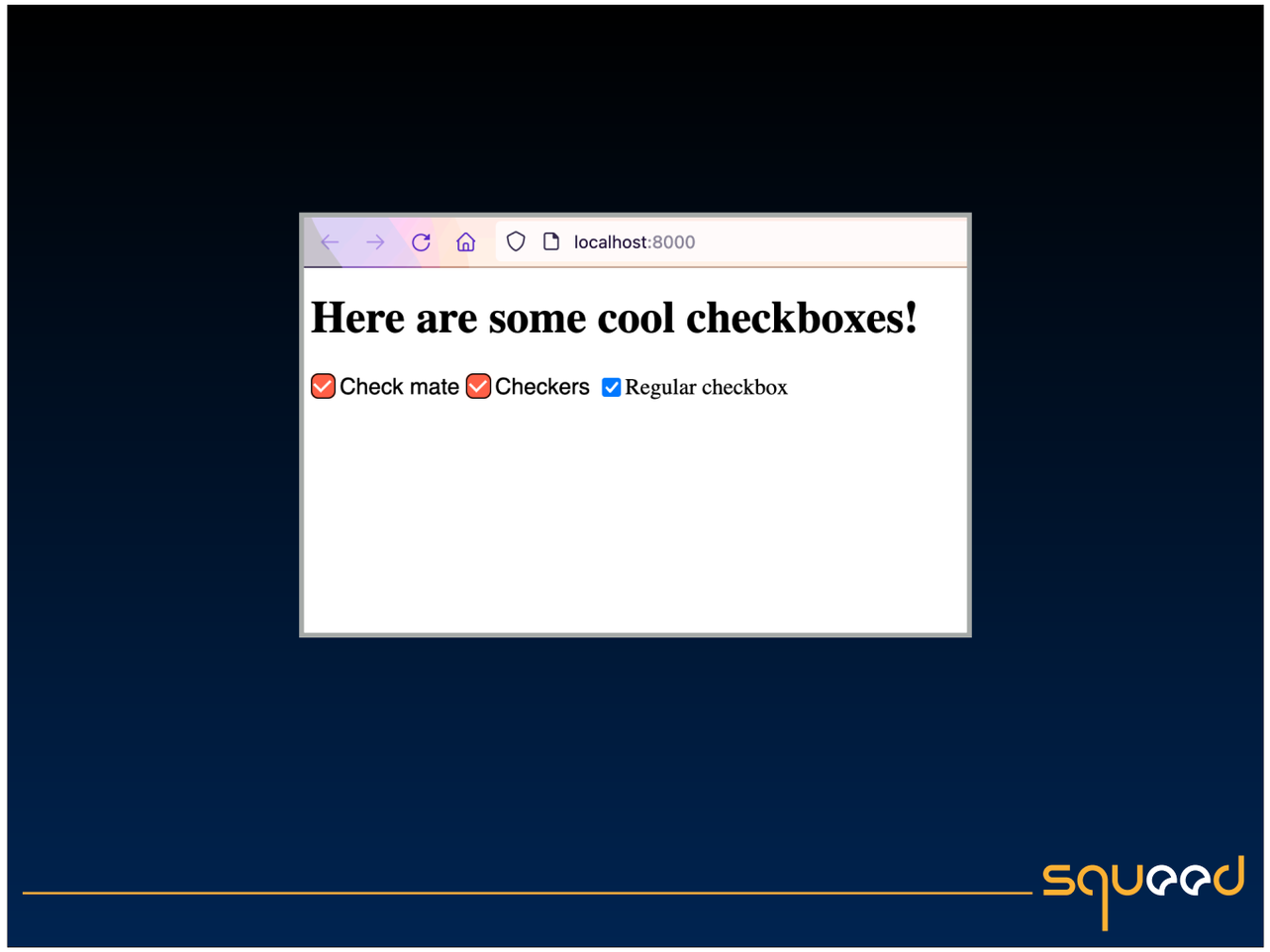


...and it appears just the way we want!

Ok, so the final way of letting the user affect he 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 lets information slip through, and that 's what we're going to explore now.

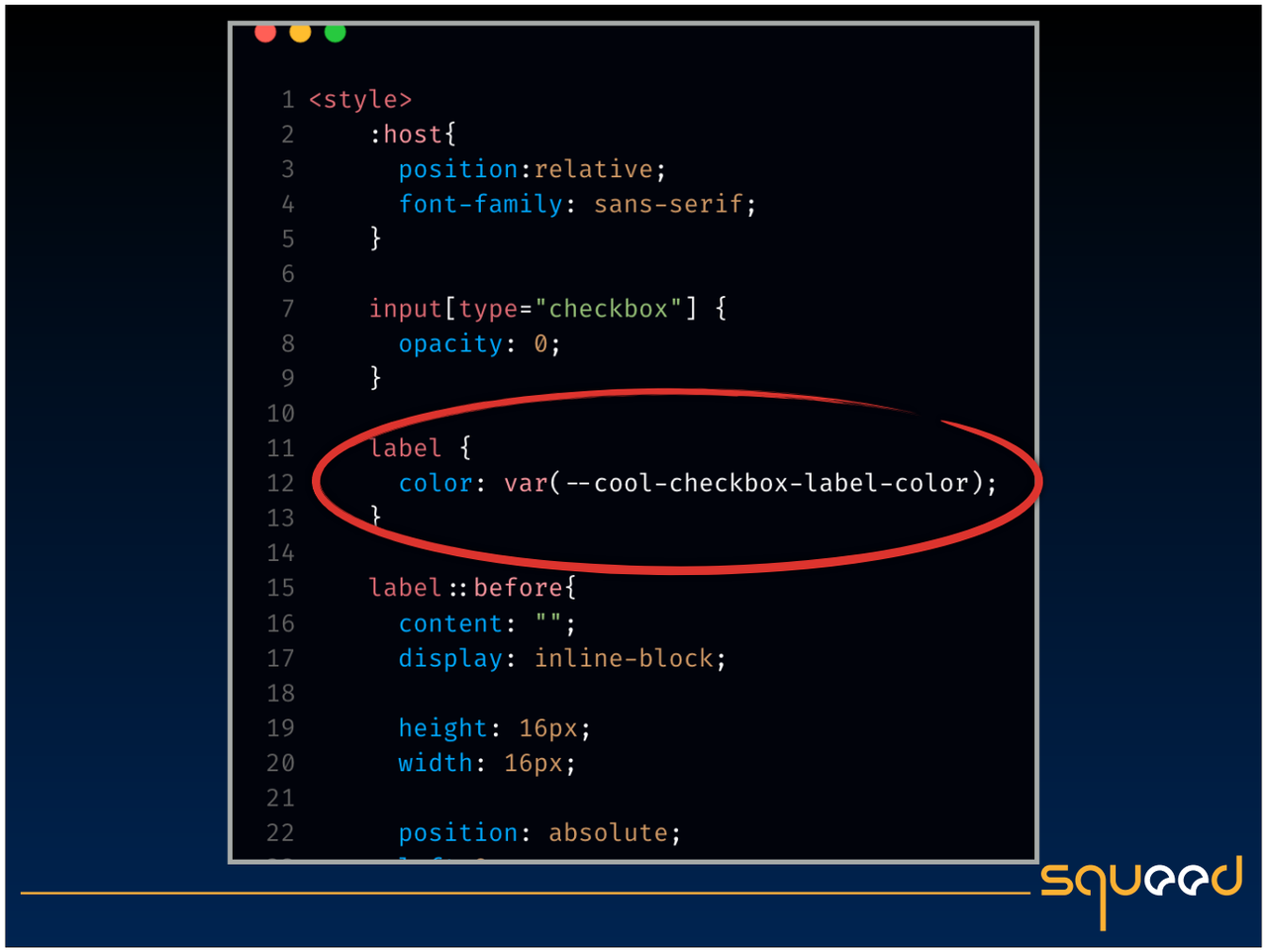


So, first of all I'll add a bunch of CSS to make the checkboxes really custom. We won't go closer into this, because how to style a checkbox is a session all in its own. But there is one interesting part of this CSS that you won't find in the light world - the :host pseudo-selector. This selector targets 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.



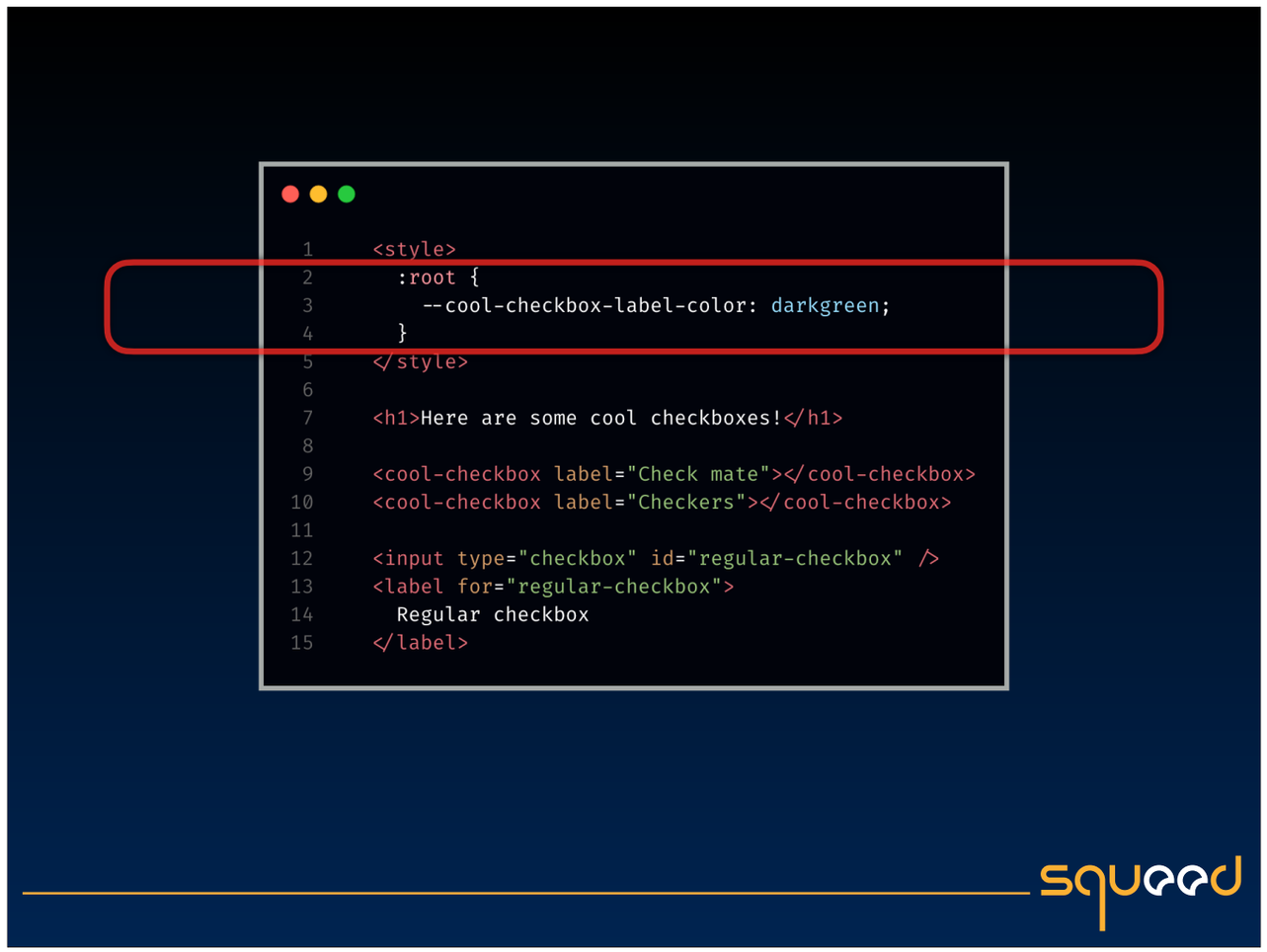
Anyway, this is what it looks like. Now the checkboxes are truly cool, especially when compared to the regular old checkbox hanging out beside.

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

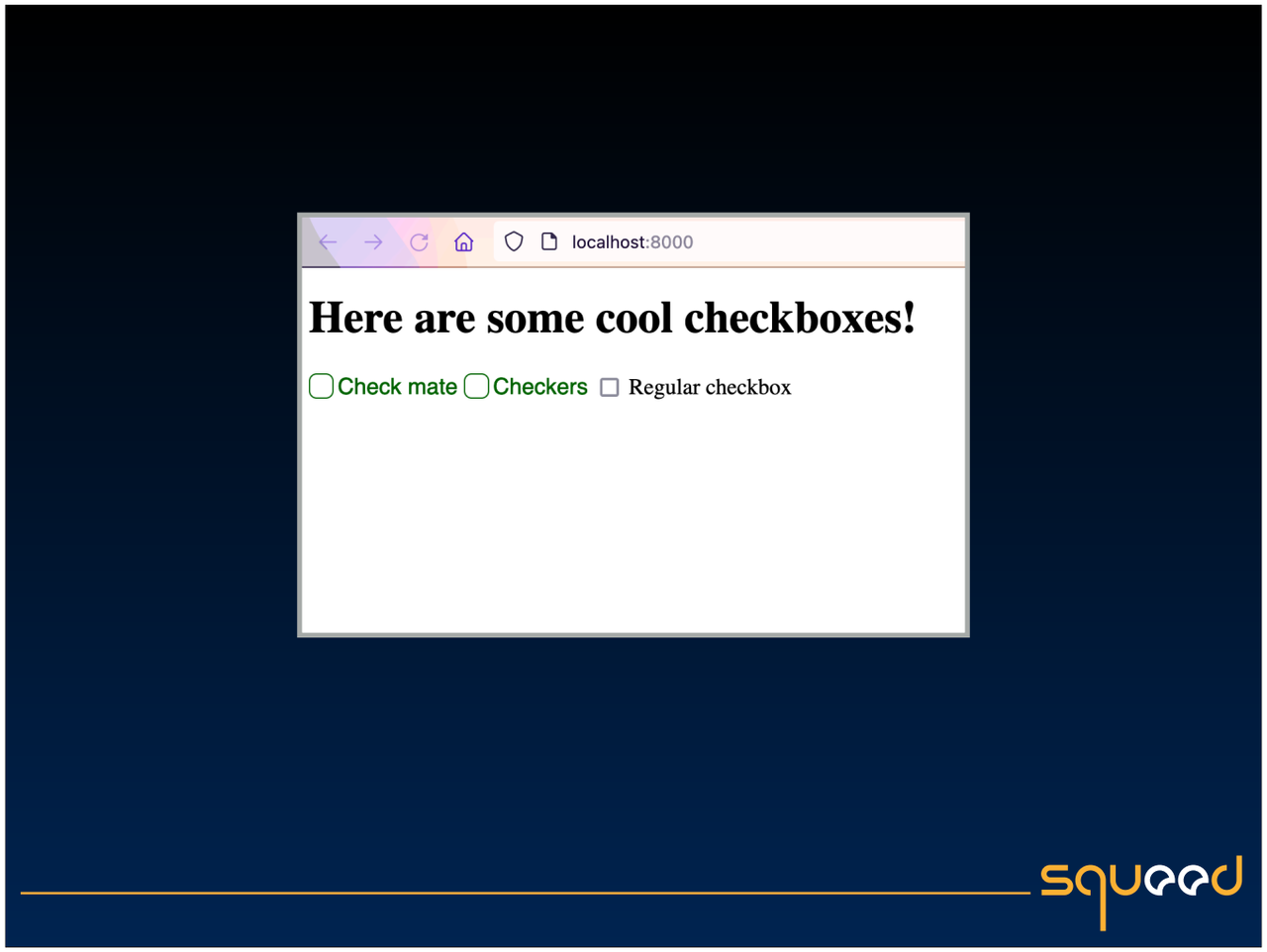


Here we set the text colour of the label to that of a CSS custom property. But we haven't declared this property anywhere, so the colour just reverts to its default value. And when it comes to text colour, it is an inherited value, meaning it is set to the computed value of its parent. That's why we 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.

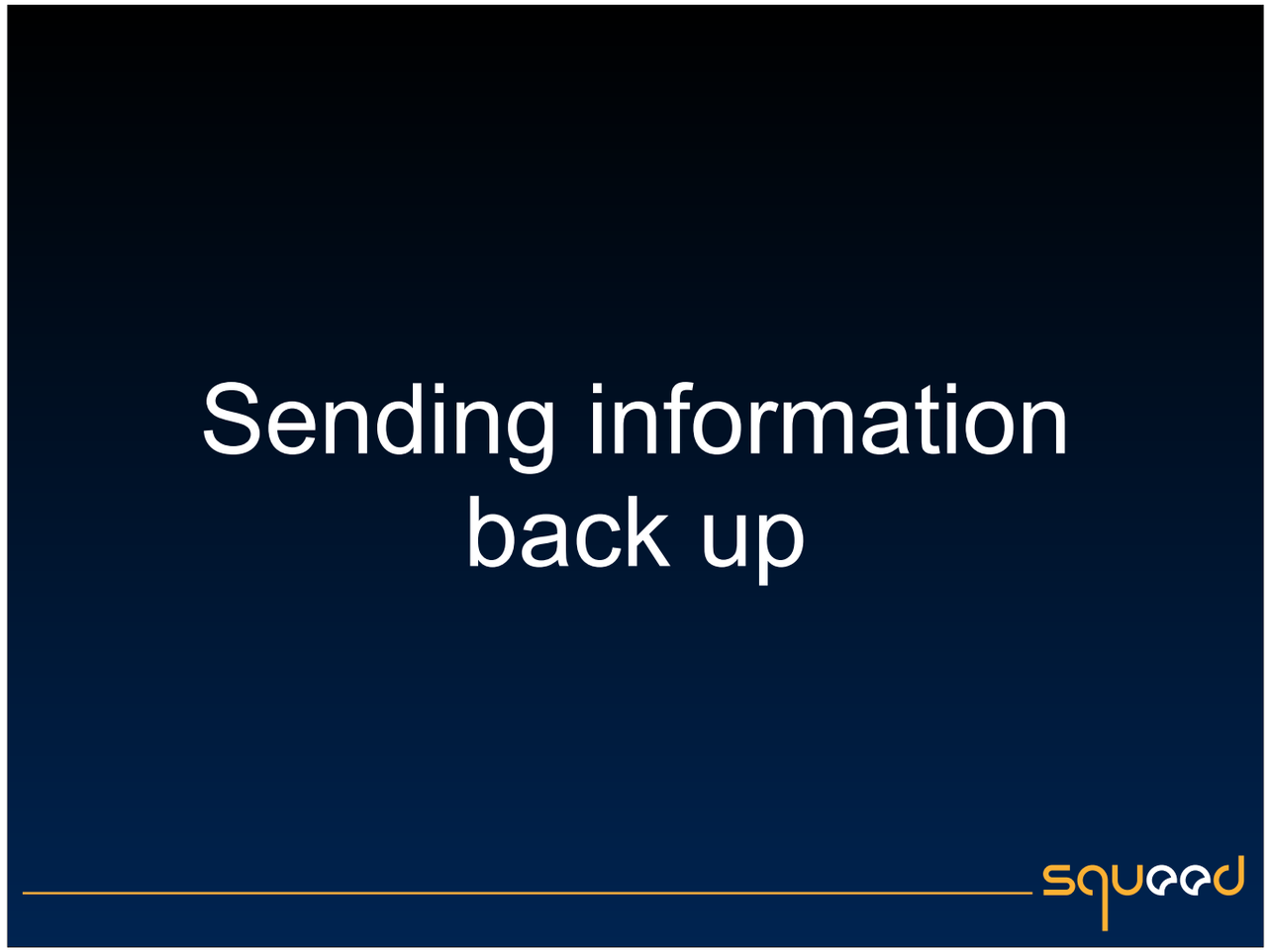
But what about this custom property then, why is it there at all? Well, custom properties are another way to pierce the veil, and a very controlled one. The custom properties we use in our CSS becomes the api for our users if they want to change the internal styling of the component.



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.

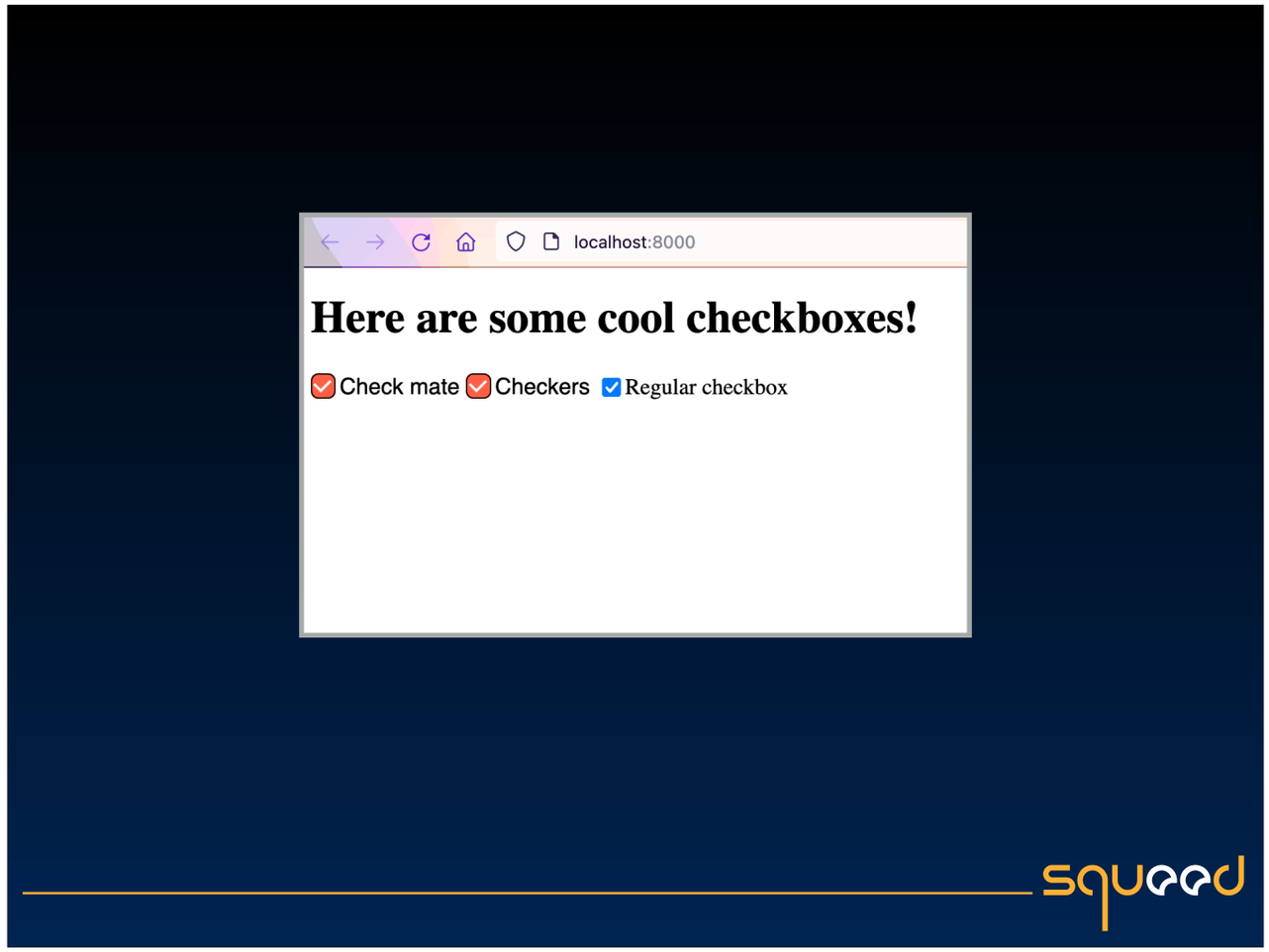


We've looked at how a user can send data into the web component to affect things, and now the final thing I want to show you is how to send data from the web component back up to the user. This is done using the good 'ol browser api: events.

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)

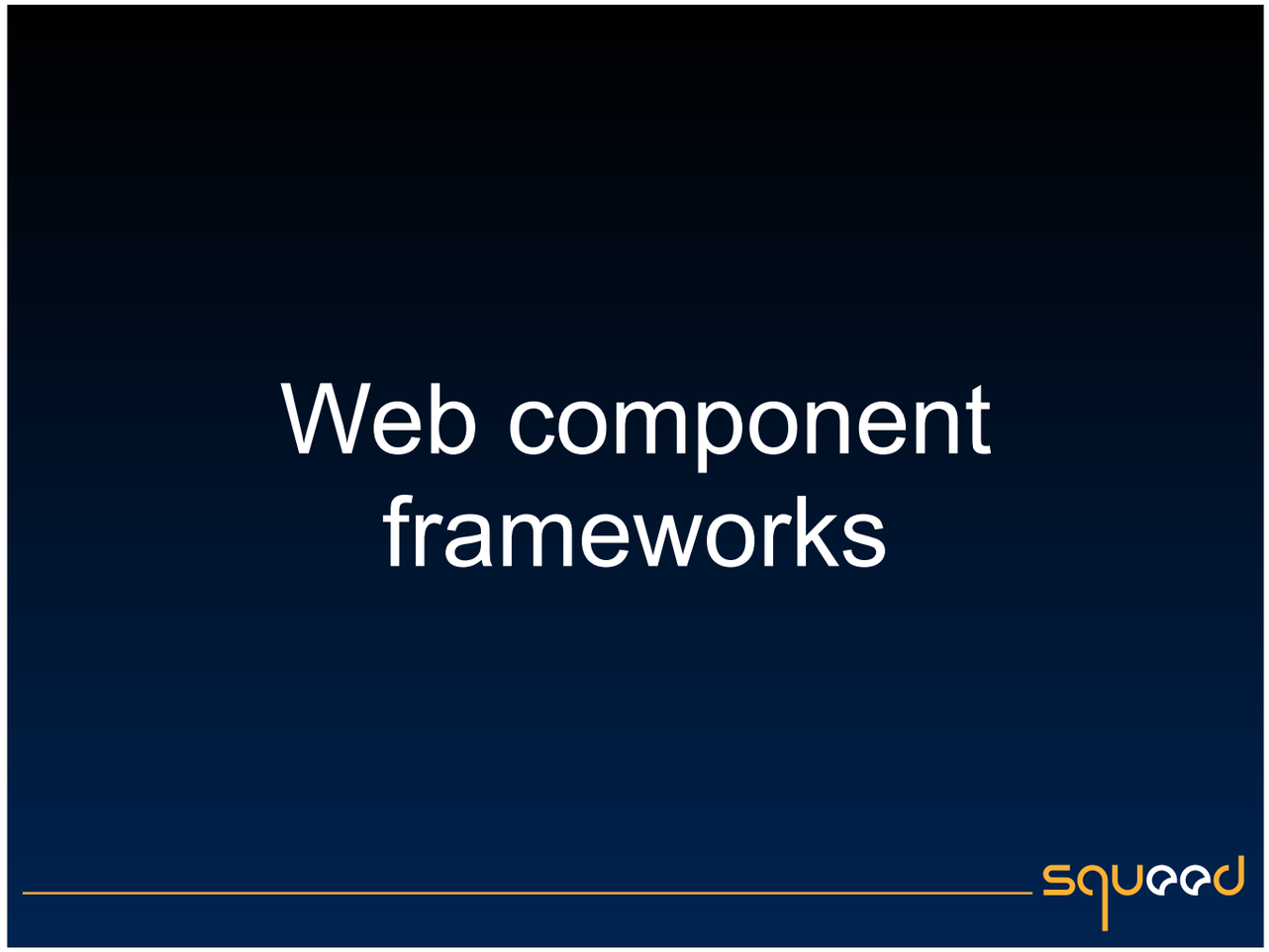
and then we dispatch a new event on the custom element. Notice that it's not on the shadowRoot this time, but on the element itself.

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.

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.

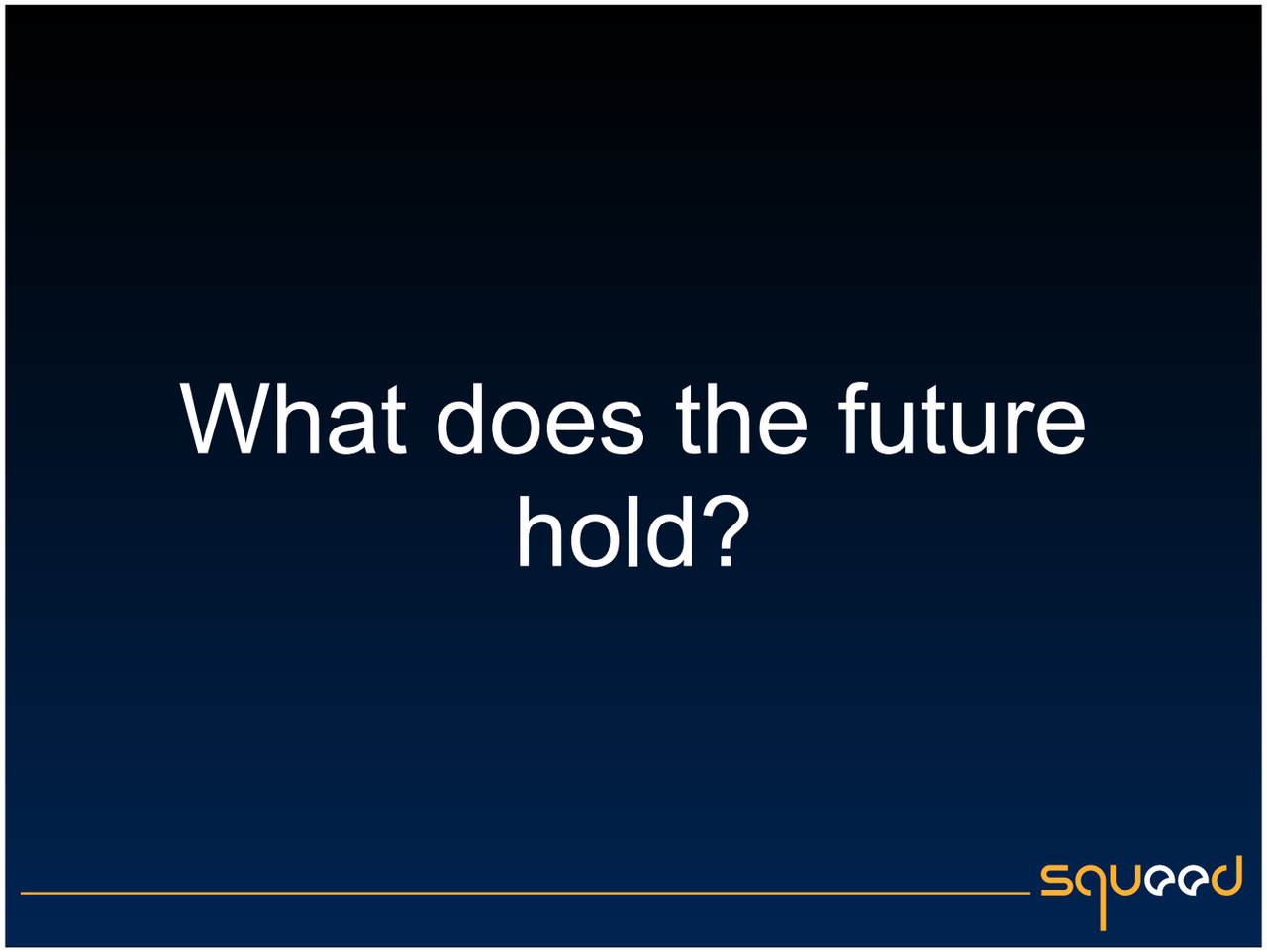


As you've seen throughout this talk, the apis that we use to create our web component are fairly low level. There are many things that need to be wired up manually, which leads to a lot of boilerplate code. Wouldn't it be easier then, to make an abstract class that takes care of all the boilerplate, 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 though 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!

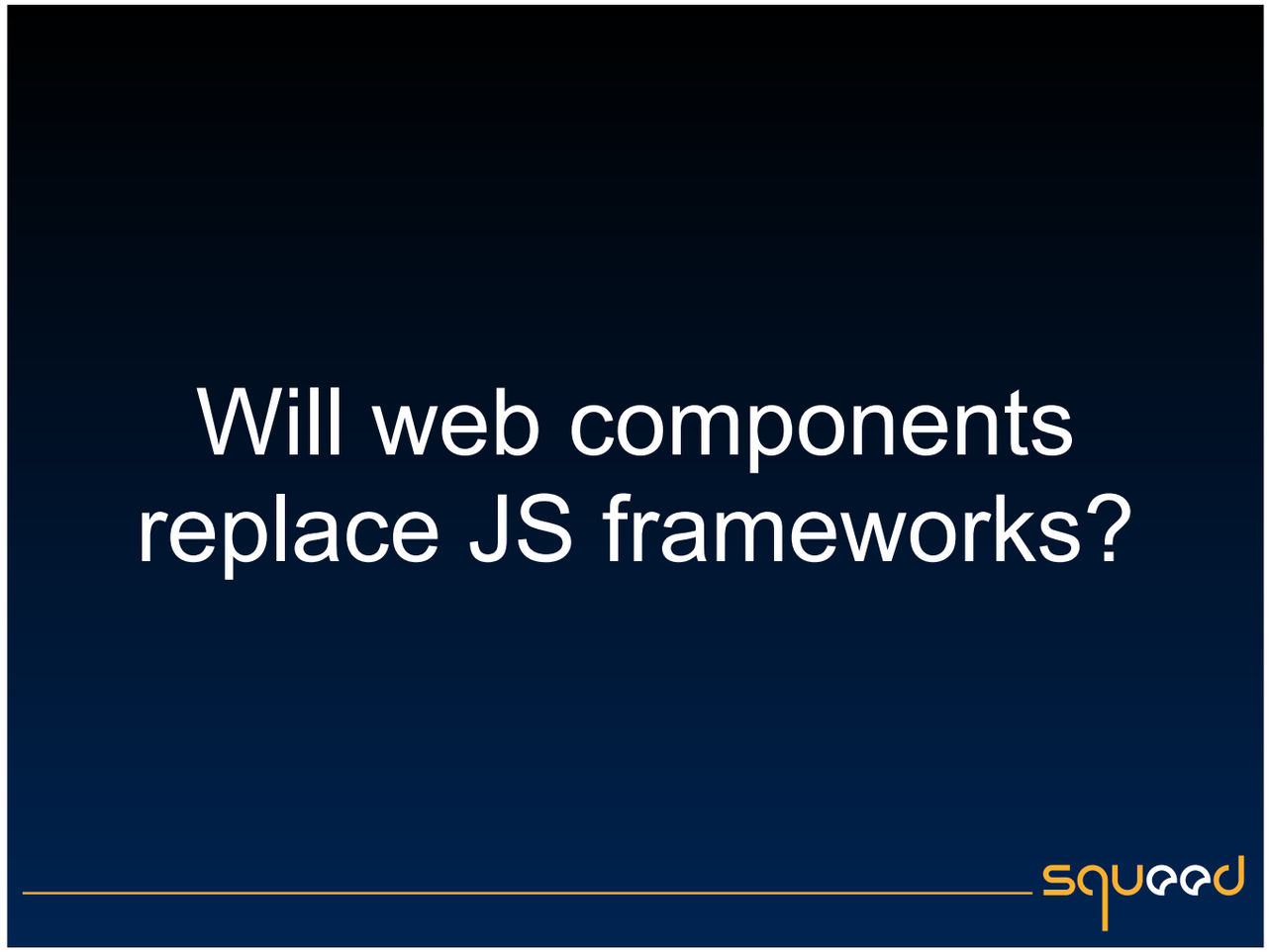
In conclusion, when it comes to web component frameworks, you will probably want to use one of these when you write web components, at least if you need a moderate amount of complexity, or more. They do make things a lot easier, and they don't add a lot of overhead. But, as with all things, having an 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.



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 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.



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...

to upgrade!

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, and you can also find me on twitter. Thanks again!