# Creating responsive components in Stencil using the ResizeObserver API

Christian Cook 🍪 Follow
Nov 15, 2018 · 5 min read

Responsive design is a key part of the modern web, allowing websites to be viewable across multiple platforms at different sizes. Webpages will adapt to the browser viewport to change the layout, sizes of elements and even add or remove additional elements within the page.

But this can give component developers a real headache when only developing part of a page. How do we know our component will be displayed at X width we designed it at? Will it be in a column on the page or will it be full width? We can only style our components by the dimensions of the entire page using CSS Media Queries… what if we took this one step further and could adapt our component to its surroundings? This is where the ResizeObserver API comes in.

## Enter ResizeObserver API

This API is a lesser-known part of the web spec and is partially supported by mainstream browsers. While it is supported by Chrome and Opera, we are still waiting for support from Apple, Microsoft and Mozilla based browsers. Luckily, if we need to support the latter browsers with our components there are polyfills/ponyfills available which can shim this functionality in. Because of the nature of Stencil we can ship these with the components that require it if we so wish.

With this implemented, we are able to observe for changes in the elements dimensions and modify our component to suit. For example if we had a component which displayed social media sharing buttons, at one size we may wish to display the labels and their respective icons… but in smaller containers we may want it to adapt to only displaying the icons to better fit the space.

## Getting started

Let's get started by creating a brand new Stencil project. In this example we will use the Stencil `component` starter.

```
npm init stencil

ionic-pwa
  app
> component
```

After you have created your project, open the source code in your favourite code editor.

## Making our component responsive

To make our component aware of its surroundings, we need to start observing for changes in its dimensions using the ResizeObserver API. In this example, we will ponyfill in this functionality using the `resize-observer-polyfill` package available on npm.

```
npm install resize-observer-polyfill --save
```

Now that we have this available in our project, we can import it in to our new component's code `./src/components/my-component/my-component.tsx`.

```
import ResizeObserver from "resize-observer-polyfill";
```

We have our ResizeObserver accessible in the component's class, so it can now be used wherever we like. Ideally we need to start watching as soon as the DOM element is available to be read, so we will start watching for changes in the `componentDidLoad()` lifecycle method.

```
@Element() element: HTMLElement;
let ro: ResizeObserver;

componentDidLoad() {
  this.ro = new ResizeObserver(entries => {
  // Do stuff!
  });
  this.ro.observe(this.element);
}
```

First off we are declaring our new instance of `ResizeObserver` and the handler which fires when it detects a change — For now this does absolutely nothing. After this we tell our instance to start observing our component's element reference, so it knows only to fire off when our component changes and no others.

Because we like to write tidy code, as the element is being observed it will also need to stop being observed if/when our component gets destroyed. We can do this by connecting in to the `componentDidUnload()` lifecycle method.

```
componentDidUnload() {
  this.ro.disconnect();
}
```

Simple! We now have our component set up to watch for changes in its dimensions — what next? Depending on the desired behaviour of the component there is a multitude of things that can be done. In this example we will apply a different CSS class to our element so we can show a different visual state depending on the width of our component.

```
applySizeClasses(size: number) {
  let small = false;
  let medium = false;
  let large = false;
  if (size <= 200) small = true;
  else if (size <= 400) medium = true;
  else large = true;
  this.element.classList.toggle("small", small);
  this.element.classList.toggle("medium", medium);
  this.element.classList.toggle("large", large);
}
```

In our snippet above, we are determining whether our component is small, medium or large in width and applying the relevant class depending on the result. This also ensures that any old states are no longer persisted on the element, else it may have all three classes as it gets resized.

Now we will need to put everything together. We can go back to our observer handler and trigger this function with the desired parameters. Here we are looping through the entries returned by our observer (in this scenario it will only be one) and passing in the width of the changed element to our new size function.

```
this.ro = new ResizeObserver(entries => {
  for (const entry of entries) {
    this.applySizeClasses(entry.contentRect.width);
  }
});
```

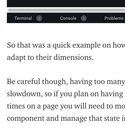
When our component is 200px or less in width, it has the `small` class applied. When it is less than 400px we apply `medium` and for anything wider it has the `large` class — Neat! But visually nothing has changed on our component yet, let's add some styles which changes the thickness of the border depending on the size.

```
:host(.small) {
  border: 1px solid red;
}

:host(.medium) {
  border: 4px solid red;
}

:host(.large) {
  border: 10px solid red;
}
```

And that is all that there is to it! We now have a component that is aware of its surroundings and adapts to its habitat **🌳.

## Show me the money 💸

So how does this all look when put together? In our example we have one component which takes up 100% width of the page, a second component which is locked to 40px wide and a third component which is contained inside a `div` set to 50% width. If we were to change the width of these components either through resizing the page or their respective containers, the components will automatically adapt to their surroundings — Give it a try!




So that was a quick example on how we can make our Stencil components adapt to their dimensions.

Be careful though, having too many `ResizeObserver` on the page can cause slowdown, so if you plan on having a responsive component hundreds of times on a page you will need to move your observer outside of your component and manage that state in the parent container or higher.

Stencijs Web Components Responsive Design



WRITTEN BY
Christian Cook 🍪 Follow
🔔 Technical Director of @eliseofficial | 💜 Co-founder of @DigitalPlymouth | Organiser of @Plymouth_Web | 🖤 Maker of Web Components