setContext and getContext

The context API provides a mechanism for components to ‘talk’ to each other without passing around data and functions as props, or dispatching lots of events. It’s an advanced feature, but a useful one.

Take this example app using a Mapbox GL map. We’d like to display the markers, using the `<MapMarker>` component, but we don’t want to have to pass around a reference to the underlying Mapbox instance as a prop on each component.

There are two halves to the context API — `setContext` and `getContext`. If a component calls `setContext(key, context)`, then any *child* component can retrieve the context with `const context = getContext(key)`.

Let’s set the context first. In `Map.svelte`, import `setContext` from `svelte` and `key` from `mapbox.js` and call `setContext`:

```
import { onDestroy, setContext } from 'svelte';
import { mapbox, key } from './mapbox.js';

setContext(key, {
    getMap: () => map
});
```

The context object can be anything you like. Like lifecycle functions, `setContext` and `getContext` must be called during component initialisation. Calling it afterwards - for example inside `onMount` - will throw an error. In this example, since `map` isn’t created until the component has mounted, our context object contains a `getMap` function rather than `map` itself.

On the other side of the equation, in `MapMarker.svelte`, we can now get a reference to the Mapbox instance:

```
import { getContext } from 'svelte';
import { mapbox, key } from './mapbox.js';

const { getMap } = getContext(key);
const map = getMap();
```

The markers can now add themselves to the map.

> A more finished version of `<MapMarker>` would also handle removal and prop changes, but we're only demonstrating context here.

Context keys

In `mapbox.js` you'll see this line:

```
const key = Symbol();
```

Technically, we can use any value as a key — we could do `setContext('mapbox', ...)` for example. The downside of using a string is that different component libraries might accidentally use the same one; using symbols, on the other hand, means that the keys are guaranteed not to conflict in any circumstance, even when you have multiple different contexts operating across many component layers, since a symbol is essentially a unique identifier.

Contexts vs. stores

Contexts and stores seem similar. They differ in that stores are available to *any* part of an app, while a context is only available to *a component and its descendants*. This can be helpful if you want to use several instances of a component without the state of one interfering with the state of the others.

In fact, you might use the two together. Since context is not reactive, values that change over time should be represented as stores:

```
const { these, are, stores } = getContext(...);
```