# Chrome Dev Summit 2014: Let's build some apps with Polymer!

**By** Rob Dodson
Rob is a contributor to Web**Fundamentals**

Over the previous year, the Polymer team has spent a lot of time teaching developers how to create their own elements. This has lead to a rapidly growing ecosystem, buoyed in large part by Polymer's Core and Paper elements, and the Brick elements created by the team at Mozilla.

As developers become more familiar with creating their own elements and start to think about building applications, it opens up a number of questions:

- How should you **structure** the UI of your application?
- How do you **transition** through different states?
- What are some strategies to improve **performance**?
- And how should you provide an **offline** experience?

For Chrome Dev Summit, I tried to answer these questions by building a small contacts application and analyzing the process I went through to build it. Here's what I came up with:

## Structure

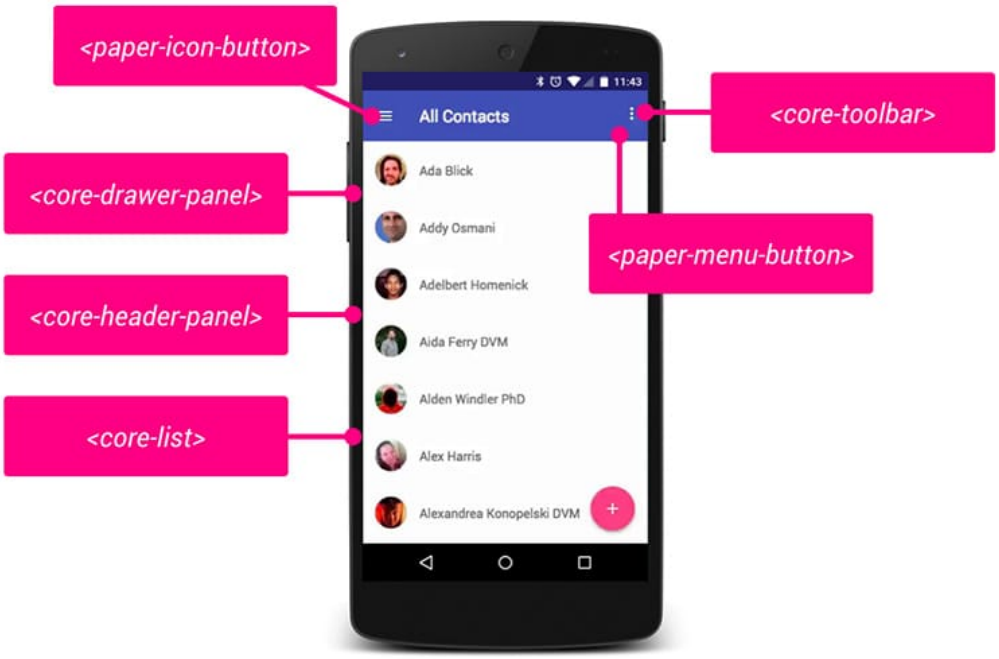
Breaking an application into modular pieces that can be combined and reused is a central tenant of Web Components. Polymer's core-* and paper-* elements make it easy to start with small pieces, like paper-toolbar and paper-icon-button...

## <core-toolbar>

A basic container for controls like tabs or buttons

```
<core-toolbar>
  <paper-icon-button icon="menu">
  </paper-icon-button>
  <div>All Contacts</div>
</core-toolbar>
```


...and through the power of composition, combine them with any number of elements to create an application scaffold.



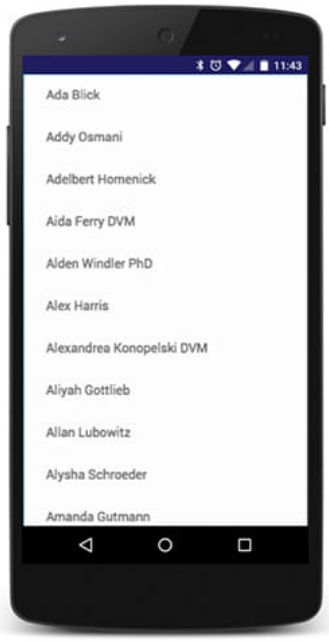

Once you have a generic scaffold in place, you can apply your own CSS styles to transform it into an experience unique to your brand. The beauty of doing this with components is that it enables you to create very different experiences while leveraging the same app building primitives. With a scaffold in place you can move on to thinking about content.

One element that is especially well suited for dealing with lots of content is the `core-list`.

A virtualized, “infinite” list

```
<core-list id=“list” flex>
  <template>
    <div>
      <div>{{model.name}}</div>
    </div>
  </template>
</core-list>
```

```
this.$.list.data = [
  {name: ‘Ada Blick’},
  {name: ‘Addy Osmani’},
  …
];
```

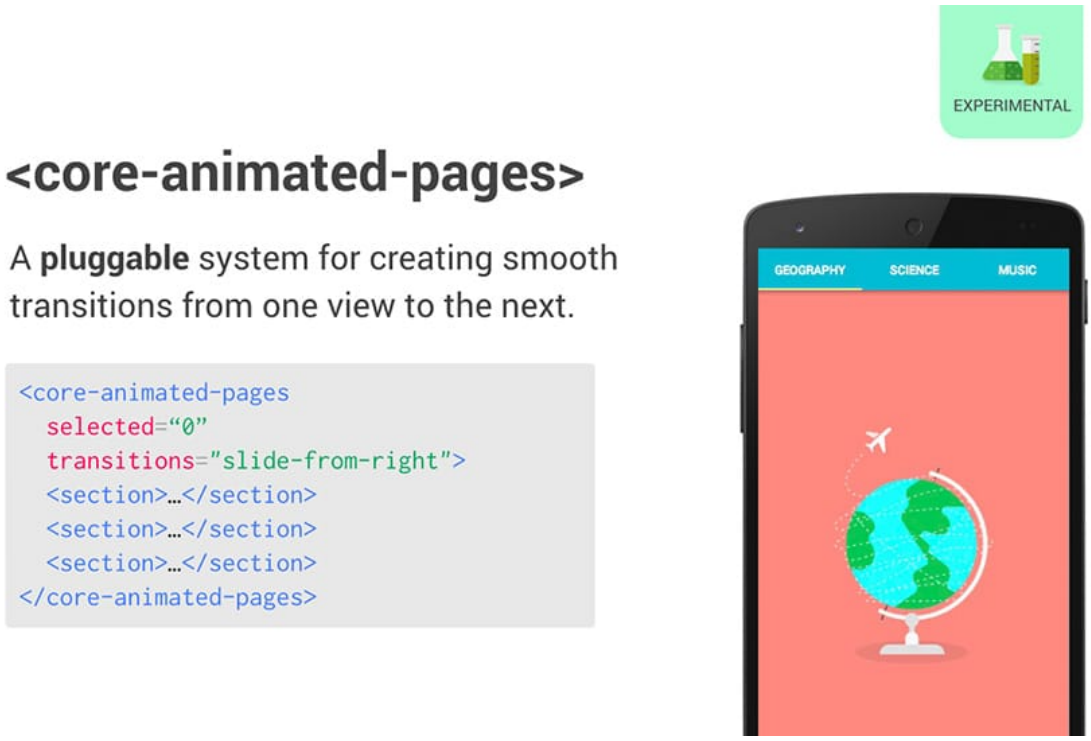



The **`core-list`** can be connected to a data source (basically an array of objects), and for each item in the array, it will stamp out a template instance. Within the template you can leverage the power of Polymer's data binding system to quickly wire up your content.

## Transitions

With the various sections of your app designed and implemented, the next task is figuring out how to actually navigate between them.

Although still an experimental element, **`core-animated-pages`** provides a pluggable animation system that can be used to transition between different states in your application.




But animation is only half of the puzzle, an application needs to combine those animations with a router to properly manage its URLs.

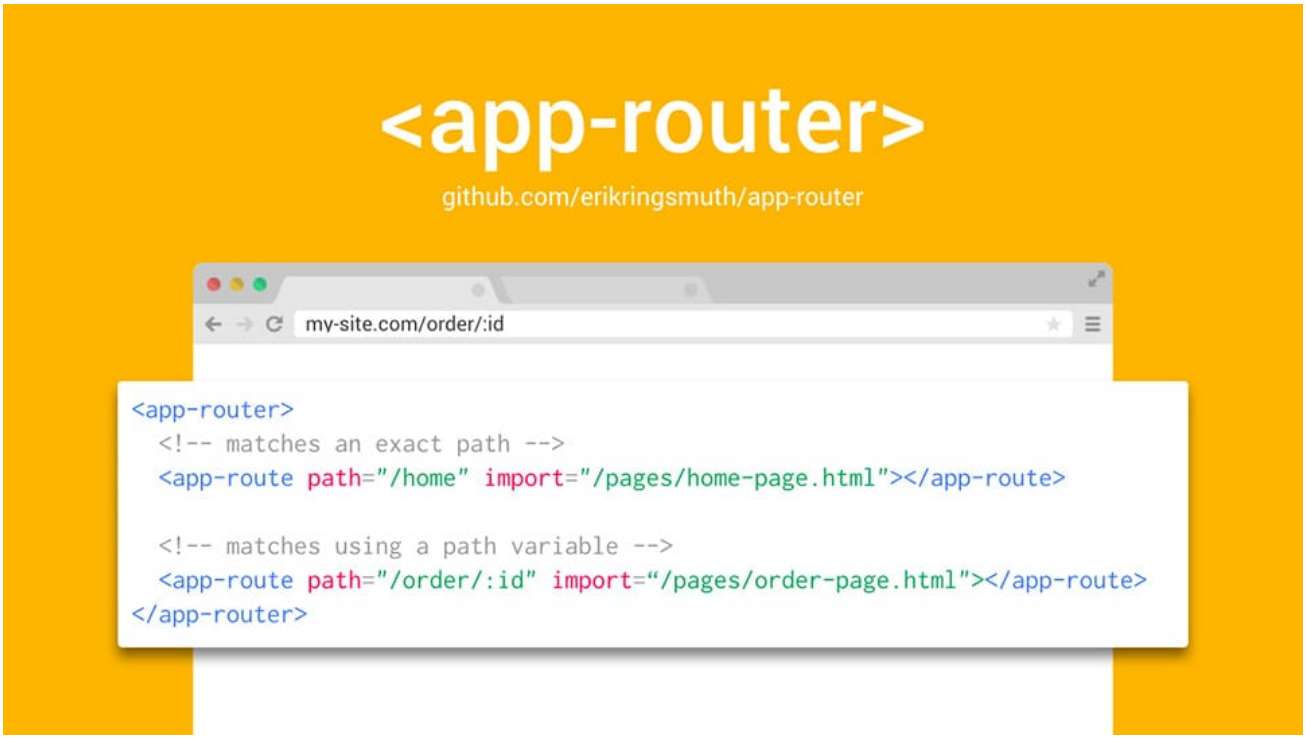
In the world of Web Components routing comes in two flavors: imperative and declarative. Combining **`core-animated-pages`** with either approach can be valid depending on your project needs.

An imperative router (such as Flatiron's Director) can listen for a matching route, and then instruct **`core-animated-pages`** to update its selected item.

```
router.on('/some/route', function() {
  pages.selected = 1;
});
```

This can be useful if you need to do some work after a route matches and before the next section has transitioned in.

On the other hand, a declarative router (like app-router) can actually combine routing and **`core-animated-pages`** into a single element, so managing the two becomes more streamlined.

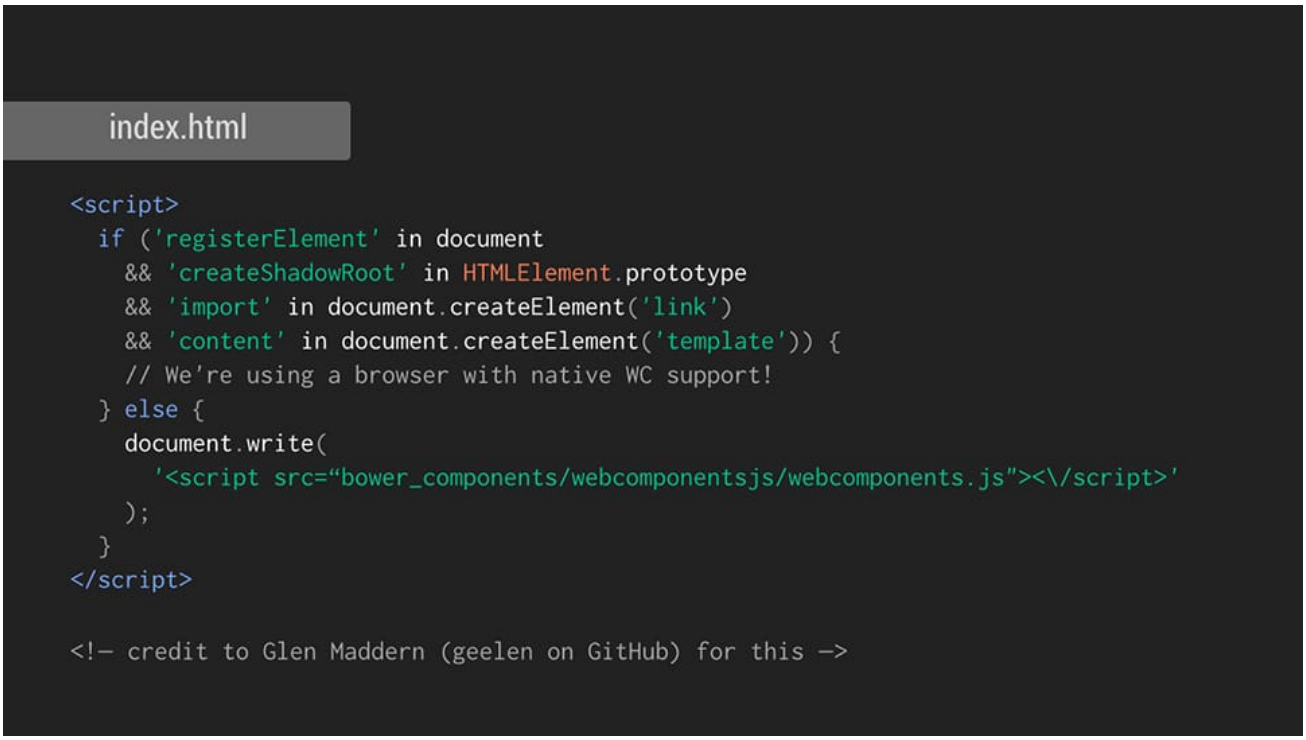



If you'd like more fine grained control, you can look at a library like more-routing, which can be combined with **`core-animated-pages`** using data binding and possibly give you the best of both worlds.

## Performance

As your application is taking shape, you have to keep a watchful eye on performance bottlenecks, especially anything associated with the network since this is often the slowest part of a mobile web application.

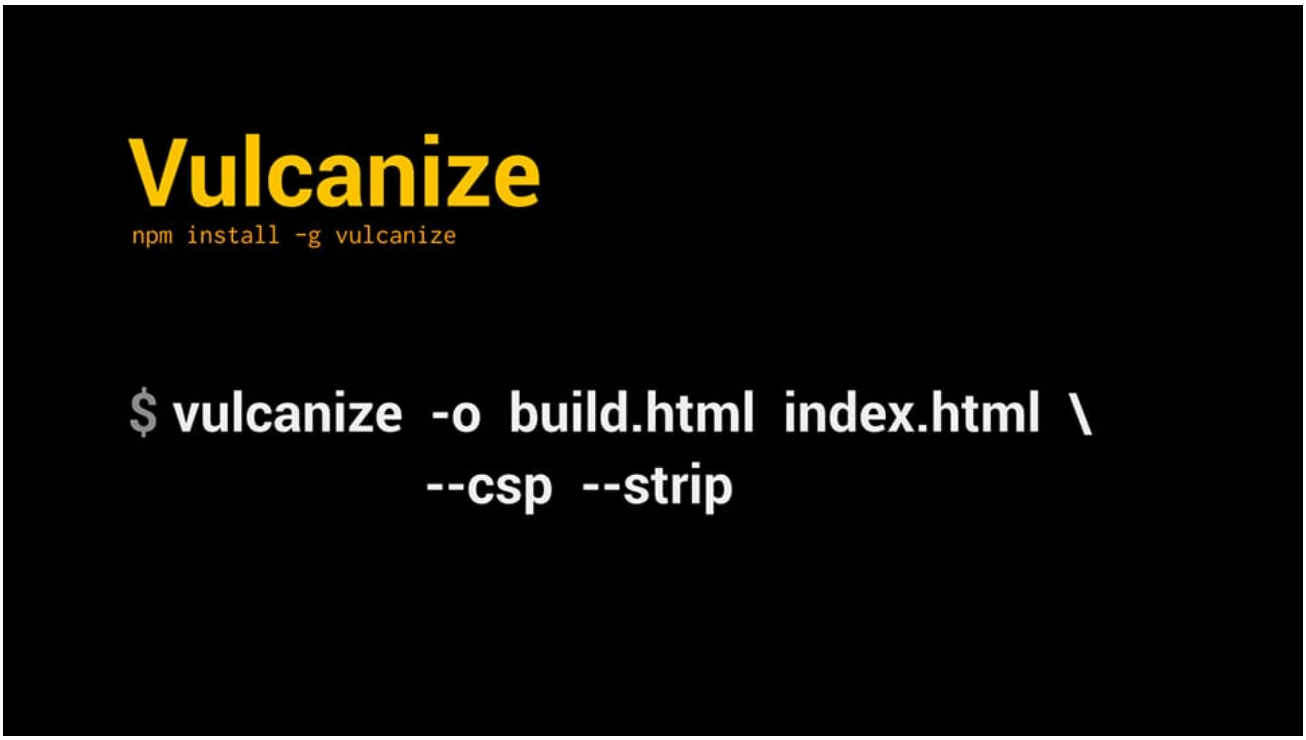
An easy performance win comes from conditionally loading the Web Components polyfills.




There's no reason to incur all that cost if the platform already has full support! In every release of the new webcomponents.js repo, the polyfills have also been broken out into standalone files. This is helpful if you want to conditionally load a subset of the polyfills.

```
<script>
  if ('import' in document.createElement('link')) {
    // HTML Imports are supported
  } else {
    document.write(
      '<script src="bower_components/webcomponentsjs/HTMLImports.min.js"><\/scrip
    );
  }
</script>
```

There are also significant network gains to be had from running all of your HTML Imports through a tool like Vulcanize.




Vulcanize will concatenate your imports into a single bundle, *significantly* reducing the number of HTTP requests that your app makes.

## Offline

But just building a performant app doesn't solve the dilemma of a user with little or no connectivity. In other words, if your app doesn't work offline, then it's not really a mobile app. Today you can use the much maligned application cache to offline your resources, but looking to the future, Service Worker should soon make the offline development experience much nicer.

Jake Archibald has recently published an amazing cookbook of service worker patterns but I'll give you the quick start to get you going:

Installing a service worker is quit easy. Create a `worker.js` file, and register it when your application boots up.

```
// Install Service Worker
if (navigator.serviceWorker) {
  navigator.serviceWorker.register('/worker.js').then(function(reg) {
    console.log('◕‿◕', reg);
  }, function(err) {
    console.log('ʘ_ʘ', err);
  });
}
```

It's important that you locate your **`worker.js`** in the root of your application, this allows it to intercept requests from any path in your app.

In the worker's install handler, I cache a boatload of resources (including the data that powers the app).

worker.js

```
self.addEventListener('install', function(event) {
  // pre cache a load of stuff:
  event.waitUntil(
    caches.open('myapp-static-v1').then(function(cache) {
      return cache.addAll([
        '/',
        '/styles/main.css',
        '/scripts/app.js',
        '/elements/elements.vulcanized.html',
        'https://polymer-contacts.firebaseio.com/all.json'
      ]);
    })
  )
});
```

This allows my app to provide at least a fallback experience to the user if they're accessing it offline.

## Onward!

Web Components are a big addition to the web platform, and they're still in their infancy. As they land in more browsers, it'll be up to us, the developer community, to figure out the best practices for structuring our applications. The above solutions give us a starting point, but there's still much more to learn. Onward to building better apps!

---



*Last updated July 2, 2018.*