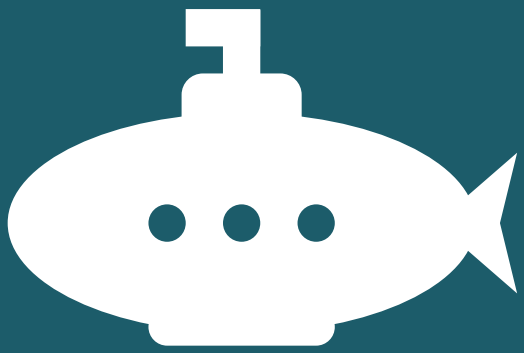
Pocket Guides  Go Make Things

10

WEB APPS

with Vanilla JavaScript



CHRIS FERDINANDI

Web Apps with Vanilla JavaScript

By Chris Ferdinandi

Go Make Things, LLC

v1.0.0

Table of Contents

1. Intro
2. Rendering Content
3. Components and State
4. Handling Events
5. URLs and Single Page Apps
6. URL Routing
7. Placeholders
8. Putting it all together
9. About the Author

Intro

In this guide, you'll learn:

- How to render dynamic content into the DOM.
- How to detect changes to your data and selectively update content.
- How to listen for user interactions and behaviors.
- How to handle URL routing with single page apps.
- How to use placeholder content to improve perceived initial load time.
- How to put it all together and write a working web app with vanilla JavaScript.

A quick word about browser compatibility

This guide makes heavy use of ECMAScript 5 (more commonly known as ES5) and ECMA 6 (ES6) methods and APIs.

My goal for browser support is IE9 and above. Each function or technique mentioned in this guide includes specific browser support information. For methods and APIs that don't meet that standard, I also include information about polyfills—snippets of code that add support for features to browsers that don't natively offer it.

You'll never have to run a command line prompt, compile code, or learn a weird pseudo language (though you certain can if you want to). Let's get started…

Note: *You can extend support all the way back to IE7 with a polyfill service like polyfill.io.*

Rendering Content

To render content into into the DOM, there are two things we need:

1. A template string with your content.
2. An element to render content into.

```
// The template string
var template = '<h1>Hello, world!</h1>';


<!-- The element -->
<div id="app"></div>
```

There are a handful of ways to approach this, but to make it easier, we'll use `render()`, a helper method based on React's method of the same name.

First, I'm going to show you how to use it. Then I'll show you the helper function itself and explain how it works.

Rendering a template into the DOM

`render()` accepts two arguments.

The first is the template you want to render. It can be a string, or a function that returns a string (useful for complex templates or when you need to use `if...else` logic).

The second is the element to render the content into. This can be either a selector, or an actual element.

```
// This works
render('<h1>Hello, world!</h1>', '#app');

// So does this
var app = document.querySelector('#app');
var template = '<h1>Hello, world!</h1>';
render(template, app);

// And so does this
var template = function () {
    return '<h1>Hello, world!</h1>';
};
render(template, '#app');
```

Returning the element

The `render()` method also returns the element back to you, so you can do things with it after your content renders.

```
var app = render('<h1>Hello, world!</h1>', '#app');
app.style.backgroundColor = 'rebeccapurple';
app.style.color = 'white';
```

Event hooks

It also provides a custom event&mdash`render`—that you can hook into to whenever an element is rendered.

You can listen for a specific element to be rendered.

```
var app = document.querySelector('#app');
app.addEventListener('render', function () {
    console.log('It was rendered!');
}, false);

// Logs "It was rendered!" in the console
render('<h1>Hello, world!</h1>', app);
```

You can also detect any element that's rendered, and get the specific element with the `event.target` property.

```
var app = document.querySelector('#app');
window.addEventListener('render', function (event) {

    // The rendered element
    var elem = event.target;

    // Log the console
    console.log(elem);

}, false);

// Logs "<div id="app"><h1>Hello, world!</h1></div>"" in
the console
render('<h1>Hello, world!</h1>', app);
```

Updating content

Any time you want to update your content, call your `render()` function again.

```
// Initial render
render('<h1>Hello, world!</h1>', '#app');

// Update 5 seconds later
window.setTimeout(function () {
    render('<h1>Bonjour, world!</h1>', '#app');
}, 5000);

// Update 10 seconds after that
window.setTimeout(function () {
    render('<h1>Hola, world!</h1>', '#app');
}, 10000);
```

The `render()` method

Here's the helper method that powers all of this.

```
/**
 * Render a template into the DOM
 * @param  {String|Function} template The template to ren
der
 * @param  {String|Node}     elem     The element to rend
er into
 * @return {[type]}                   The element
 */
var render = function (template, elem) {
```

```
    // Set rendering element for the component
    if (typeof template === 'function') {
        template.elem = elem;
    }

    // If elem is an element, use it.
    // If it's a selector, get it.
    elem = typeof elem === 'string' ? document.querySelec
tor(elem) : elem;
    if (!elem) return;

    // Get the template
    template = (typeof template === 'function' ? template
(template.state) : template);
    if (typeof template !== 'string') return;

    // Render the template into the element
    if (elem.innerHTML === template) return;
    elem.innerHTML = template;

    // Dispatch a render event
    if (typeof window.CustomEvent === 'function') {
        var event = new CustomEvent('render', {
            bubbles: true
        });
        elem.dispatchEvent(event);
    }

    // Return the elem for use elsewhere
    return elem;
```

```
};
```

Let's break it down and look at what each piece of the code does.

Set the rendering element for the component

If the template passed in is a function, we give it a property of `elem` and store the target element or select to it. This probably doesn't make much sense now, but we'll be using this in the next chapter to help automatically re-render content when our data changes.

```
// Set rendering element for the component
if (typeof template === 'function') {
    template.elem = elem;
}
```

Get the element

The element can be a selector or an element. If it's a string, pass it into `querySelector()` to get the actual element.

We're also checking to make sure the element actually exists or was found so that we don't throw a JavaScript error and break our app.

```
// If elem is an element, use it.
// If it's a selector, get it.
elem = typeof elem === 'string' ? document.querySelector(
elem) : elem;
if (!elem) return;
```

Get the template

The template can be either a string or a function. If it's a function, we need to run it to get our string. If there's no string, we'll end our function.

```
// Get the template
template = (typeof template === 'function' ? template(tem
plate.state) : template);
if (typeof template !== 'string') return;
```

Render our template into our element

Now we're ready to render our template string into our element, using the `innerHTML` property.

But in order to avoid causing a browser repaint if we don't need to, we're first going to get the current HTML in our element as a string, and compare it to our template.

If they're the same, we won't render. If they're different, we will.

```
// Render the template into the element
if (elem.innerHTML === template) return;
elem.innerHTML = template;
```

Dispatch a custom event

Next, we're going to dispatch our custom event.

This uses the `CustomEvent` API, which lacks support for Internet Explorer and, at time of writing, Safari. As such, we'll want to check that it's supported first. You can also add a polyfill to extend `CustomEvent` support[1] back to IE9.

We're setting `bubbles` to true so that you can listen for `render` events on the `document` or `window`, not just the event it was dispatched on.

```
// Dispatch a render event
if (typeof window.CustomEvent === 'function') {
    var event = new CustomEvent('render', {
        bubbles: true
    });
    elem.dispatchEvent(event);
}
```

Return the event

Finally, we're going to return the rendered element for use elsewhere in

your scripts.

```
// Return the elem for use elsewhere
return elem;
```

Browser Compatibility

The `render()` method works in all moderns browsers, and IE9 and up.

However, it requires a polyfill for `CustomEvent` support[2] in Internet Explorer and Safari. It will render content without it, but you won't be able to use the events feature.

Components and State

If you've never heard that word "state" before in JavaScript—or have but don't know what it means—you're not alone!

State is just data.

So why do they call it state instead of data? Because there's a time-bound aspect to it.

State is data at a particular moment in time. It's the present "state" of your data. Get it?

Components

A component is a template function that has state (or data) attached to it. Here's our "Hello, world" function from last chapter?

```
var template = function () {
    return '<h1>Hello, world!</h1>';
};
```

We can add state to it by giving our function a `state` property with an object as it's value. That object can contain any data relevant to the template.

```
template.state = {
    greeting: 'Hello'
};
```

Our `render()` function from the previous chapter passes our `state` property into the function as an argument. We can use this to add logic to our template.

Let's use `props` as the argument name for our `state`.

```
var template = function (props) {
    return '<h1>' + props.greeting + ', world!</h1>';
};
```

Now, our template will use whatever greeting is in it's current state.

Updating State

To update state, you can reassign a value to to your `state` property as a whole, or any property in the `state` object, just like you would with any other JavaScript object.

```
render(template, '#app');

// Update 5 seconds later
window.setTimeout(function () {
    // Change the greeting to "Bonjour"
    // Update just the `greeting` property
    template.state.greeting = 'Bonjour';
    render(template, '#app');
}, 5000);

// Update 10 seconds after that
window.setTimeout(function () {
    // Change the greeting to "Hola"
    // Update the whole state this time instead
    template.state = {
        greeting: 'Hola'
    }
    render(template, '#app');
}, 10000);
```

Detecting State Changes

There's a good chance that when your state updates, you'll want to render your content again with the new data.

`component()` is a helper method that adds a helper function, `setState()`, that you can use to update the state of your component *and* render (or re-render) your content as the same time. It also adds state to your template, turning it into a component.

Just like `render()`, first I'll show you how to use it, and then explain how it works.

Create your component

The `component()` method accepts three arguments: your template, the state, and the element (or element selector) to render your content into.

It returns the template back to you, so you can use it to setup a template for the first time *or* add state to an existing template.

```
// Create a new template
var template = component(function (props) {
    return '<h1>' + props.greeting + ', world!</h1>';
}, {
    greeting: 'Hello'
}, '#app');

// Or add state to an existing template
var template = function (props) {
    return '<h1>' + props.greeting + ', world!</h1>';
};
component(template, {
    greeting: 'Hello'
}, '#app');
```

The last argument—the node to render content into—is optional.

In the last chapter, I mentioned that the `render()` method adds the `elem` property to your function. If you don't specify an element for your component when setting it up, it will use whatever element (or selector) you used for your last render.

```
// Create your component
var template = component(function (props) {
    return '<h1>' + props.greeting + ', world!</h1>';
}, {
    greeting: 'Hello'
});

// Render your component
// It will use `#app` as your element selector going forw
ard
render(template, '#app');
```

Update state and render

To update state and cause your content to render, use the `setState()` method.

The `setState()` method requires you to pass in an object, and will conduct a shallow merge into your existing state. You can pass in your entire state object, or just the properties you want to update.

```
render(template, '#app');

// Update 5 seconds later
window.setTimeout(function () {
    // Change the greeting to "Bonjour" and re-render the
 template
    template.setState({greeting: 'Bonjour'});
}, 5000);

// Update 10 seconds after that
window.setTimeout(function () {
    // Change the greeting to "Hola" and re-render the te
mplate
    template.setState({greeting: 'Hola'})
}, 10000);
```

The `component()` method

Here's the helper method for creating components.

```
/**
 * Create a component
 * @param  {Function}       template  The template functi
on
 * @param  {Object}         props     The state data
 * @param  {String|Element} elem      The element to rend
er content into
```

```
 * @return {Function}                    The template functi
on
 */
var component = function (template, props, elem) {

    Object.defineProperties(template, {
        elem: {
            value: elem,
            writable: true
        },
        state: {
            value: props,
            writable: true
        },
        setState: {
            value: function (props) {

                // Shallow merge new properties into stat
e object
                for (var key in props) {
                    if (props.hasOwnProperty(key)) {
                        template.state[key] = props[key];
                    }
                }

                // Render the element
                render(template, template.elem);

                // Return the elem for use elsewhere
                return template.elem;
```

```
            }
        }
    });

    // Return the elem for use elsewhere
    return template;

};
```

Let's break it down and look at what each piece of the code does.

Define object properties

The `Object.defineProperties()` method let's you add multiple properties to an object. Unlike simply assigning them —`template.someProperty = ...`—this method let's you specify whether or the property value can be changed or deleted, and if it should be iterated over in loops.

We use it to add our properties to the `template` function.

```
Object.defineProperties(template, {
    // ...
});
```

Add the element as a property

If an element or element selector is passed in as an argument, we set it as a property. We'll use it later when we automatically call our `render()` function inside the `setState()` method.

```
elem: {
    value: elem,
    writable: true
}
```

Add state

Next, we need to add state to our template.

```
state: {
    value: props,
    writable: true
}
```

Create the `setState()` method

Finally, we'll add the `setState()` method to our template.

This loops through any passed in object properties and merges them into the existing template state. Then it calls our `render()` function, passing in the template and the `elem` property for the template. Finally, it returns the rendered element.

```
setState: {
    value: function (props) {

        // Shallow merge new properties into state object
        for (var key in props) {
            if (props.hasOwnProperty(key)) {
                template.state[key] = props[key];
            }
        }

        // Render the element
        render(template, template.elem);

        // Return the elem for use elsewhere
        return template.elem;

    }
}
```

Browser Compatibility

The `component()` method works in all moderns browsers, and IE9 and up.

Handling Events

For your app to be truly interactive, you need to listen to user inputs and interactions and update your state accordingly.

We can use the browser-native `addEventListener()` method for this. You can find a full list of available events on the Mozilla Developer Network. [3]

```
var btn = document.querySelector('#click-me');
btn.addEventListener('click', function (event) {
    console.log(event); // The event details
    console.log(event.target); // The clicked element
}, false);
```

Multiple Targets

The vanilla JavaScript `addEventListener()` function requires you to pass in a specific, individual element to listen to. You cannot pass in an array or node list of matching elements like you might in jQuery or other frameworks.

To add the same event listener to multiple elements, you also **cannot** just use a `for` loop because of how the `i` variable is scoped (as in, it's not and changes with each loop).

Fortunately, there's an easy way to detect events on multiple elements with a single listener: event bubbling.

Instead of listening to specific elements, we'll instead listen for *all* clicks on a page, and then check to see if the clicked item has a matching selector.

```
// Listen for clicks on the entire window
window.addEventListener('click', function (event) {

    // If the clicked element has the `.click-me` class, it's a match!
    if (event.target.matches('.click-me')) {
        // Do something...
    }

}, false);
```

This works for other types of events as well.

Multiple Events

In vanilla JavaScript, each event type requires it's own event listener. Unfortunately, you *can't* pass in multiple events to a single listener like you might in other frameworks.

But... by using a named function and passing that into your event listener, you can avoid having to write the same code over and over again.

```
// Setup our function to run on various events
var someFunction = function (event) {
    // Do something...
};

// Add our event listeners
window.addEventListener('click', someFunction, false);
window.addEventListener('scroll', someFunction, false);
```

Use Capture

The last argument in `addEventListener()` is `useCapture`, and it specifies whether or not you want to "capture" the event. For most event types, this should be set to `false`. But certain events, like `focus`, don't bubble.

Setting `useCapture` to `true` allows you to take advantage of event bubbling for events that otherwise don't support it.

```
// Listen for all focus events in the document
document.addEventListener('focus', function (event) {
    // Run functions whenever an element in the document
comes into focus
}, true);
```

Preventing the default behavior

You can prevent the default behavior from happening with `event.preventDefault()`. You would use it to, for example, prevent a link for triggering a page reload or stop a form from submitting data.

```
document.addEventListener('click', function (event) {
    event.preventDefault();
    // The rest of your code...
}, true);
```

An Example of Event Handling

If you were building a todo app, you'd want users to be able to add items to their list. Here's a sample template for that list that will generate an unordered list with all of the todos.

```
// Create the template
var todoList = function (props) {
    var items = '';
    for (var i = 0; i < props.todos.length; i++) {
        items += '<li>' + props.todos[i] + '</li>';
    }
    return '<ul>' + items + '</ul>';
};

// Add state
component(todoList, {todos: []}, '#todo-list');
```

And here's a form users can use to add items to their list (we're not going to worry about marking them as complete or anything in this example).

```
<form id="add-todos">
    <label for="add-todo-item">Add a task to your list</label>
    <input type="text" name="add-todo-item" id="add-todo-item">
    <button type="submit">Add Todo</button>
</form>
```

And here's the event listener we would use to listen for new todos and update our list.

```
// Listen for form submit events
// This let's us capture both submit button clicks AND so
meone hitting enter on the keyboard
document.addEventListener('submit', function (event) {

    // If the submitted form is the `#add-todos` form
    if (event.target.matches('#add-todos')) {

        // Get the input field
        var todo = event.target.querySelector('#add-todo-
item');

        // If there's no todo item, bail
        if (todo.value.length < 1) return;

        // Update our state
        todoList.setState({todos: todoList.state.push(tod
o.value)});

        // Clear our todo field
        todo.value = '';

    }

}, false);
```

Browser Compatibility

`addEventListener` works in all modern browsers, and IE9 and above.

The `matches()` method, mentioned under event bubbling, is also supported in all modern browsers, and IE9 and up. *But*... older versions of certain browsers used a vendor prefixed version instead of the official spec. You should use a polyfill[4] to ensure consistent behavior in all browsers.

URLs and Single Page Apps

One of the bigger challenges with single page apps (apps that serve different views and layouts from a single HTML file) is handling URL routing.

In this chapter, you'll learn how to provide unique URLs for different pages, update the browser history so that the forward and back buttons works properly, and show different content based on those URLs.

The History API

The History API is a browser API that provides a way for you to update URLs and interact with the browser history.

Updating the URL

The `pushState()` method lets you update the browser URL without triggering a page refresh. It accepts three arguments.

1. `state` - Info about the new URL. You can access this data for the current URL in the browser history with the `history.state` property.
2. `title` - The new page title, though it's ignored by most browsers (more on that later).
3. `url` - The URL of the new page. This can only be a URL on the current domain (no external links, for security reasons).

```
history.pushState({
    page: 'settings'
}, 'Settings | My Awesome App', 'my-awesome-app.com?p=set
tings');
```

`pushState()` creates a new item in the browser history. Clicking the forward and backward buttons will update the URL in the address bar.

There's also a `replaceState()` method that works exactly the same way, but instead of creating a new item in the browser history, it replaces the current one.

Detecting forward and back button clicks

The `onpopstate` event is triggered whenever someone clicks the forward or backward buttons in their browser. You can use it to update the page content when someone uses those buttons to navigate.

```
windown.onpopstate = function (event) {
    if (history.state && history.state.page === 'settings
') {
        // Render new content for the settings page
    }
};
```

You can also use it with `addEventListener`, which is my preferred approach.

```
window.addEventListener('popstate', function (event) {
    if (history.state && history.state.page === 'settings
') {
        // Render new content for the settings page
    }
}, false);
```

Note: *The* `pushState()` *and* `replaceState()` *methods don't trigger an* `onpopstate` *event.*

Query Strings

Query strings are key/value pairs in a URL.

You've almost certainly seen them before. They look like this.

```
http://my-awesome-app.com?search=vanilla%20javascript&ref
erer=https://twitter.com
```

In the example above, `search` has a value of `vanilla javascript` (the `%20` is a URL encoded space), and `referer` has a value of `https://twitter.com`.

Query strings are perfect for single page apps for a few reasons:

1. Servers by default do not treat them as a different page. `my-awesome-app.com` and `my-awesome-app.com/search/vanillajs` are viewed as two separate pages, but `my-awesome-app.com?search=vanillajs`

is viewed by the server as `my-awesome-app.com`.

2. You can easily encode information about a page/view into your URL.
3. Google and other search engines recognize and index them as unique URLs (if the content is different).

I like to use `p` as the key for page names. For example, `my-awesome-app.com?p=settings`.

Getting query string values

The `getParams()` helper method, adapted from CSS Tricks, gets all of the query strings in a URL and returns an object of key/value pairs.

Here's how to use it.

```
// Example with the current browser URL
getParams(window.location.href);

// Example with a string
// Returns {
//   p: 'settings',
//   referer: 'https://twitter.com'
// }
getParams('my-awesome-app.com?p=settings&referer=https://
twitter.com');
```

And here's the method itself.

```
/**
 * Get the URL parameters
 * source: https://css-tricks.com/snippets/javascript/get
-url-variables/
 * @param  {String} url The URL
 * @return {Object}     The URL parameters
 */
var getParams = function (url) {
    var params = {};
    var parser = document.createElement('a');
    parser.href = url;
    var query = parser.search.substring(1);
    var vars = query.split('&');
    for (var i=0; i < vars.length; i++) {
        var pair = vars[i].split("=");
        params[pair[0]] = decodeURIComponent(pair[1]);
    }
    return params;
};
```

Real URLs

Many single page apps today use real URLs. Instead of `my-awesome-app.com?p=settings`, you would use `my-awesome-app.com/settings`.

In order to use this approach, you need to be able to configure your server to user your `index.html` file for all paths or routes. If you don't and someone tries to go directly to a page within your app instead of the homepage (for example, `my-awesome-app.com/settings`), they'll get a `404` error.

Parsing the URL pathname

The `parsePathname()` method is a function I wrote to extract variables from the pathname (the `/something` part) of a URL.

Pass in the pattern to match against, and the pathname to parse. The pattern can be a basic string.

```
// matches: '/settings'
// doesn't match: '/settings/username'
var pattern1 = '/settings';

// matches: '/settings/username'
// doesn't match: '/settings'
var pattern2 = '/settings/username';
```

Or, it can include parameters you want to extract from the URL pathname. Your parameter names should be wrapped in parentheses and start with a colon: `(:myVariable)`. Use `(*)` for wildcards (these won't be matched or extracted).

```
// matches: '/books/rowling/harry%20potter'
// matches: '/books/adams/hitchhikers'
// doesn't match: '/books/adams/hitchhikers/part-2'
var pattern1 = '/books/(:author)/(:book)';

// matches: '/publisher/adams/hitchhikers'
// matches: '/publisher/1979/hitchhikers'
var pattern2 = '/publisher/(*)/(:book)';
```

It returns an object with the pattern, any parameters, and the url.

```
{
    id: pattern, // The pattern if it matches, or null if
 it doesn't
    params: {},  // Any parameters extracted from the URL
    url: url     // The URL itself
};

// Example
parsePathname('/books/(:author)/(:book)', '/books/adams/h
itchhikers');

// Returns
{
    id: '/books/(:author)/(:book)',
    params: {
        author: 'adams',
        book: 'hitchhikers'
    },
    url: '/books/adams/hitchhikers'
}
```

parsePathname()

And here's the `parsePathname()` helper method.

```
/**
 * Parse a URL pathname for variables
 * @param  {String} pattern The pattern to match
```

```
 * @param  {String} url     The pathname to parse
 * @return {Object}         Details from the pathname
 */
var parsePathname = function (pattern, url) {

    // Variables
    var map = {};
    var keys = [];
    var matches = false;

    // If the URL is an exact match for the pattern, retu
rn it
    if (url === pattern || url + '/' === pattern) {
        return {
            id: pattern,
            params: {},
            url: url
        };
    }

    // Add a trailing slash to the URL if one is missing
    url = url.slice(-1) === '/' ? url : url + '/';

    // Push variables in the pattern to our key array and
 replace them with regex match grouping
    var newPattern = pattern.replace('(*)', '.+?').replac
e(/\(:.+?\)/g, function(match) {
        var key = match.slice(2, -1);
        keys.push(key);
```

```
        return '([^/]*)';
    });

    // Test the URL against the pattern.
    // If it's a match, pull the variables out into the m
ap
    var test = url.replace(new RegExp('^' + newPattern +
'/$'), function() {
        matches = true;
        for (var i = 0; i < keys.length; i++) {
            map[keys[i]] = arguments[i+1];
        }
    });

    // Return the data
    return {
        id: (matches ? pattern : null),
        params: map,
        url: url
    };
};
```

Putting it all together

You can use `pushState()` to update browser URL without reloading the page, `onpopstate` to handle browser buttons, and query strings or a real URL for the different pages in your web app.

There are still a few missing pieces, though:

1. A page-level state you can hook into in your components for styling purposes (as in, some JavaScript object that indicates what page should be displayed). You *could* use `history.state`, but I'd prefer something independent from it.
2. A way to trigger rendering when the page state changes. Remember, `onpopstate` events aren't triggered by `pushState()`.
3. A way to reliably update the page title in the tab or menu bar.
4. A way to handle URLs for pages in our app that don't exist (as in, what to do about `404` errors).

We'll take a look at how to address them in the next chapter.

Browser Compatibility

Query strings work in any browser, as do the `getParams()` and `parsePathname()` helper methods.

The History API works in all modern browsers, and IE10 and up. You can add more backwards compatibility with a polyfill. history.js[5] is the most popular. Alternatively, you could allow a full page refresh instead of Ajax page loading.

URL Routing

The `router()` function is a helper method I wrote to handle some of the heavy lifting with URL routing and address some of the unanswered challenges I mentioned at the end of the last chapter.

It does a few things:

1. Creates a page-level state you can access inside your components.
2. Provides a hook to render content on URL changes.
3. Updates the document title.
4. Handles `404` errors.

As always, I'll first explain how to use it, and then dig into how it works.

Passing links through the router

You can route links in your app through your router by giving them the `[data-route]` data attribute.

```
<a data-route href="?p=settings">Settings</a>
```

Creating URL routes

The `router()` method accepts two arguments: your URL routes, and an optional object of settings that you can configure.

The routes

For each route in your `routes` object, the key should be the query string value (if using query strings) or URL pattern (if using real URLs) for that page.

The value is an object with some details about the page. `title` is the title for that page. If the page should redirect to another page, also include the `redirect` key, and use the query string value or URL pattern for the page it should redirect to as the value.

```
// Example with Query Strings
var routes = router({

    // The settings page
    // The query string for this page is "?p=settings"
    settings: {
        title: 'Settings' // The page title
    },

    // The homepage
    // The query string for this page is "?p=home"
    home: {
        title: 'My Awesome App'
    },

    // Pages with no query string
    '': {
        redirect: '?p=home' // Redirect to "?p=home"
    }
```

```
});

// Example with URL Pathnames
var routes = router({

    // The settings page
    '/settings': {
        title: 'Settings' // The page title
    },

    // The homepage
    '/': {
        title: 'My Awesome App'
    },

    // The old homepage, which we now redirect
    '/home': {
        redirect: '/'
    }

});
```

Configuring router settings

You can also configure a few settings for your router. Every one is optional.

```
var routerSettings = {

    // The selector to use for your links
    // If your app has no external links, you could set t
his to `a` and skip the data attribute
    selector: '[data-route]',

    // Links to ignore, useful if you use a more broad `s
elector`
    ignore: '[data-route-ignore]',

    // The query string key to use
    // You might, for example, set this to `page` instead
    // URLs would look like this: "?page=settings"
    key: 'p',

    // The default title for app pages.
    // This gets appended to the end of your route-specif
ic title.
    // For example, if this was set to "My App" and your
settings page title was "Settings | ", the rendered page
title would be "Settings | My App"
    title: '',

    // If true, listen for onpopstate events and route th
e updated URL
    onpopstate: true,

    // If true, force a full page reload instead of using
 pushState()
```

```
    // By default, this is set to true automatically when
 pushState() isn't supported, false when it is
    forceReload: !(history.pushState),

    // The type of routing to use (query strings or URL p
athnames)
    // valid values: query|path
    type: 'query'

};

var routes = router({
    settings: {
        title: 'Settings' // The page title
    },
    ...
}, routerSettings);
```

Rendering content when the URL changes

Every time a URL is routed, the `router()` function emits a custom `route` event you can hook into to call the appropriate `render()` functions.

```
window.addEventListener('route', function () {
    render(template, '#app');
}, false);
```

State is added to your URL routes, and contains the current page, a full list of all parameters from the URL, and the URL itself.

```
// URL: "?p=settings&user=Chris"
routes.state = {
    id: 'settings',
    params: {
        p: 'settings',
        user: 'Chris'
    },
    url: '?p=settings&user=Chris'
};

// URL: "/settings/chris"
routes.state = {
    id: 'settings',
    params: {
        page: 'settings',
        user: 'Chris'
    },
    url: '/settings/chris'
};
```

You can hook into it in your templates.

```
var template = function (props) {
    var str = '';
    if (routes.state.id === 'settings') {
        str =
            '<h1>Settings</h1>' +
            '<p>Configure your settings</p>';
    } else {
        str = '<h1>' + props.greeting + ', world!</h1>'
    }
    return str;
};
```

The Router API

The `router()` function will automatically route any URLs when links with the `[data-route]` attribute (or whatever other selector you've set) are clicked.

But you can also manually run some of its internal processes.

```
// Route any URL (and trigger a `route` event)
routes.route('?p=home');

// Add a page to the routes.state, but don't trigger a a
`route` event or update the URL
routes.setRoute('?p=home');

// Update the URL with the current state
routes.push();

// Replace the current item in browser history with the c
urrent state
routes.replace();
```

The `router()` method

Here's the `router()` helper method.

```
/**
 * Setup URL router
 * @param  {Object} routes  The URL paths
 * @param  {Object} options Custom options
 * @return {Object}         The routes
 */
var router = function (routes, options) {

    // Defaults
```

```
    var settings = {
        selector: '[data-route]',
        ignore: '[data-route-ignore]',
        key: 'p', // The query string key to use
        title: '',
        onpopstate: true,
        forceReload: !(history.pushState),
        type: 'query' // valid: query|path
    };

    // Shallow merge new properties into settings object
    for (var key in options) {
        if (options.hasOwnProperty(key)) {
            settings[key] = options[key];
        }
    }

    /**
     * Get the URL parameters
     * source: https://css-tricks.com/snippets/javascript
/get-url-variables/
     * @param  {String} url The URL
     * @return {Object}     The URL parameters
     */
    var getParams = function (url) {
        var params = {};
        var parser = document.createElement('a');
        parser.href = url;
        var query = parser.search.substring(1);
        var vars = query.split('&');
```

```
        for (var i=0; i < vars.length; i++) {
            var pair = vars[i].split("=");
            params[pair[0]] = decodeURIComponent(pair[1])
;
        }
        return params;
    };

    /**
     * Parse a URL pathname for variables
     * @param  {String} pattern The pattern to match
     * @param  {String} url     The pathname to parse
     * @return {Object}         Details from the pathname
     */
    var parsePathname = function (pattern, url) {

        // Variables
        var map = {};
        var keys = [];
        var matches = false;

        // If the URL is an exact match for the pattern,
return it
        if (url === pattern || url + '/' === pattern) {
            return {
                id: pattern,
                params: {},
                url: url
            };
```

```
        }

        // Add a trailing slash to the URL if one is miss
ing
        url = url.slice(-1) === '/' ? url : url + '/';

        // Push variables in the pattern to our key array
 and replace them with regex match grouping
        var newPattern = pattern.replace('(*)', '.+?').re
place(/\(:.+?\)/g, function(match) {
            var key = match.slice(2, -1);
            keys.push(key);
            return '([^/]*)';
        });

        // Test the URL against the pattern.
        // If it's a match, pull the variables out into t
he map
        var test = url.replace(new RegExp('^' + newPatter
n + '/$'), function() {
            matches = true;
            for (var i = 0; i < keys.length; i++) {
                map[keys[i]] = arguments[i+1];
            }
        });

        // Return the data
        return {
            id: (matches ? pattern : null),
            params: map,
```

```
            url: url
        };
    };

    /**
     * Get the route state from query strings
     * @param  {String} url The URL
     * @return {Object}     The route state
     */
    var getRouteFromQuery = function (url) {

        // Get the query string parameters and the page i
d
        var params = getParams(url);
        var id = params[settings.key] ? params[settings.k
ey].toLowerCase() : '';

        // If route has redirect, call setRoute with redi
rect URL
        if (routes[id] && routes[id].redirect) {
            routes.setRoute(routes[id].redirect);
            return;
        }

        return {
            id: routes[id] ? id : 404,
            params: params,
            url: url
        };
    };
```

```
,,

/**
 * Get a route state from a pathname
 * @param  {String} url The URL
 * @return {Object}     The route state
 */
var getRouteFromPath = function (url) {

    // Make URL case-insensitive
    url = url.toLowerCase();

    // Loop through each route and test against URL
    for (var route in routes) {
        if (routes.hasOwnProperty(route)) {
            var parsed = parsePathname(route, url);
            if (parsed.id) {
                return parsed;
            }
        }
    }

    // Otherwise return 404
    return {
        id: 404,
        params: {},
        url: url
    };

};
```

```
/**
 * Setup the route properties
 */
Object.defineProperties(routes, {
    state: {
        value: {
            id: '',
            params: {},
            url: null
        }
    },
    setRoute: {
        value: function (url) {

            // Bail if there's no URL
            if (!url) return;

            // Get the route state
            var state;
            if (settings.type === 'path') {
                state = getRouteFromPath(url);
                if (!state) return;
                routes.state.id = state.id;
                routes.state.params = state.params;
                routes.state.url = state.url;
            } else {
                state = getRouteFromQuery(url);
                if (!state) return;
                routes.state.id = state.id;
```

```
                routes.state.params = state.params;
                routes.state.url = state.url;
            }

            return routes.state;
        }
    },
    push: {
        value: function () {
            if (settings.forceReload) {
                if (window.location.href === routes.state.url) return;
                window.location.href = routes.state.url;
            }

            if (!history.pushState) return;

            // Update the page title
            document.title = (routes[routes.state.id] && routes[routes.state.id].title ? routes[routes.state.id].title : '') + settings.title;

            // Don't run if already current page
            if (history.state && history.state.url && history.state.url === routes.state.url) return;
            history.pushState(
                routes.state,
                (routes[routes.state.id] && routes[routes.state.id].title ? routes[routes.state.id].title : ''
```

```
) + settings.title,
                    routes.state.url
                );
            }
        },
        replace: {
            value: function () {
                if (!history.replaceState) return;

                // Update the page title
                document.title = (routes[routes.state.id]
 && routes[routes.state.id].title ? routes[routes.state.i
d].title : '') + settings.title;

                // Don't run if already current page
                if (history.state && history.state.url &&
 history.state.url === routes.state.url) return;
                history.pushState(
                    routes.state,
                    (routes[routes.state.id].title ? rout
es[routes.state.id].title : '') + settings.title,
                    routes.state.url
                );
            }
        },
        route: {
            value: function (url) {
                if (!url) return;
                routes.setRoute(url);
```

```
                routes.push();
                var event = new CustomEvent('route', {
                    bubbles: true,
                    detail: {
                        state: routes.state
                    }
                });
                document.dispatchEvent(event);
                return routes.state;
            }
        }
    });

    /**
     * Add event listeners
     */

    // Listen for router link clicks
    window.addEventListener('click', function (event) {
        if (!event.target.matches(settings.selector) || e
vent.target.matches(settings.ignore)) return;
        event.preventDefault();
        routes.route(event.target.href);
    }, false);

    // Listen for popstate events
    if (settings.onpopstate) {
        window.addEventListener('popstate', function (eve
nt) {
            if (!history.state.url) return;
```

```
            routes.route(history.state.url);
        }, false);
    }

    return routes;

};
```

Let's break it down and look at what each piece of the code does.

Merging options into defaults

First, we establish default setting values, and merge any options passed into the `router()` method into them.

```
// Defaults
var settings = {
    selector: '[data-route]',
    ignore: '[data-route-ignore]',
    key: 'p',
    title: '',
    onpopstate: true,
    forceReload: !(history.pushState),
    type: 'query'
};

// Shallow merge new properties into settings object
for (var key in options) {
    if (options.hasOwnProperty(key)) {
        settings[key] = options[key];
    }
}
```

Get parameters

Next, I've included our `getParams()` and `parsePathname()` helper methods for getting query string and URL parameters.

```
/**
 * Get the URL parameters
 * source: https://css-tricks.com/snippets/javascript/get
-url-variables/
 * @param  {String} url The URL
 * @return {Object}     The URL parameters
 */
var getParams = function (url) {
    // ...
};

/**
 * Parse a URL pathname for variables
 * @param  {String} pattern The pattern to match
 * @param  {String} url     The pathname to parse
 * @return {Object}         Details from the pathname
 */
var parsePathname = function (pattern, url) {
    // ...
};
```

Get the route from a query string

If using query strings, this gets the page parameter from the URL, checks to see if it exists in your routes, and returns back the state object.

```
/**
 * Get the route state from query strings
 * @param  {String} url The URL
 * @return {Object}     The route state
 */
var getRouteFromQuery = function (url) {

    // Get the query string parameters and the page id
    var params = getParams(url);
    var id = params[settings.key] ? params[settings.key].
toLowerCase() : '';

    // If route has redirect, call setRoute with redirect
 URL
    if (routes[id] && routes[id].redirect) {
        routes.setRoute(routes[id].redirect);
        return;
    }

    return {
        id: routes[id] ? id : 404,
        params: params,
        url: url
    };
};
```

Get the route from a pathname

If using URL pathnames, this loops through each route until it finds a match. Then it returns the route state.

```
/**
 * Get a route state from a pathname
 * @param  {String} url The URL
 * @return {Object}     The route state
 */
var getRouteFromPath = function (url) {

    // Make URL case-insensitive
    url = url.toLowerCase();

    // Loop through each route and test against URL
    for (var route in routes) {
        if (routes.hasOwnProperty(route)) {
            var parsed = parsePathname(route, url);
            if (parsed.id) {
                return parsed;
            }
        }
    }

    // Otherwise return 404
    return {
        id: 404,
        params: {},
        url: url
    };

};
```

Add properties to the routes

I'm again using `Object.defineProperties` to add properites to the routes that were passed in to `router()`.

Route State

The `routes.state` property sets up our baseline state.

```
state: {
    value: {
        id: '',
        params: {},
        url: null
    }
}
```

The `routes.setRoute()` method sets our route state.

If using query strings, it gets the query string or pathname (depending on your settings) from whatever URL is passed into it. Then it gets the route state form the URL using one of the methods mentioned above.

```
setRoute: {
    value: function (url) {

        // Bail if there's no URL
        if (!url) return;

        // Get the route state
        var state;
        if (settings.type === 'path') {
            state = getRouteFromPath(url);
            if (!state) return;
            routes.state.id = state.id;
            routes.state.params = state.params;
            routes.state.url = state.url;
        } else {
            state = getRouteFromQuery(url);
            if (!state) return;
            routes.state.id = state.id;
            routes.state.params = state.params;
            routes.state.url = state.url;
        }

        return routes.state;
    }
},
```

Updating the URL

The `routes.push()` method updates the URL based on the `state.page` of your routes. It also uses `document.title` to update the title of the document.

If `forceReload` is set to true, it forces a page reload instead of using `pushState()`.

```
push: {
    value: function () {
        if (settings.forceReload) {
            if (window.location.href === routes.state.url
) return;
            window.location.href = routes.state.url;
        }

        if (!history.pushState) return;

        // Update the page title
        document.title = (routes[routes.state.id] && rout
es[routes.state.id].title ? routes[routes.state.id].title
 : '') + settings.title;

        // Don't run if already current page
        if (history.state && history.state.url && history
.state.url === routes.state.url) return;
        history.pushState(
            routes.state,
            (routes[routes.state.id] && routes[routes.sta
te.id].title ? routes[routes.state.id].title : '') + sett
ings.title,
            routes.state.url
        );
    }
}
```

The `routes.replace()` method works almost the same exact way as

`routes.push()`, but uses `replaceState()` and doesn't force a page reload.

```
replace: {
    value: function () {
        if (!history.replaceState) return;

        // Update the page title
        document.title = (routes[routes.state.id] && rout
es[routes.state.id].title ? routes[routes.state.id].title
 : '') + settings.title;

        // Don't run if already current page
        if (history.state && history.state.url && history
.state.url === routes.state.url) return;
        history.pushState(
            routes.state,
            (routes[routes.state.id].title ? routes[route
s.state.id].title : '') + settings.title,
            routes.state.url
        );
    }
}
```

Routing the URL

The `routes.route()` method actually routes the URL for us.

First, it runs `setRoute()` to get the route state. Then it runs `routes.push()` to update the URL. Finally, it emits our custom `route` event. Finally, it returns the route state for use in other scripts and returns the route state.

```
route: {
    value: function (url) {
        if (!url) return;
        routes.setRoute(url);
        routes.push();
        var event = new CustomEvent('route', {
            bubbles: true,
            detail: {
                state: routes.state
            }
        });
        document.dispatchEvent(event);
        return routes.state;
    }
}
```

Event Listeners

The handle our click events, `router()` sets up a click event listener and checks to see if the clicked link has our route selector.

```
// Listen for router link clicks
window.addEventListener('click', function (event) {
    if (!event.target.matches(settings.selector) || event
.target.matches(settings.ignore)) return;
    event.preventDefault();
    routes.route(event.target.href);
}, false);
```

If `settings.onpopstate` is true, it also listens for `popstate` events and routes the update URL through `routes.route()`.

```
// Listen for popstate events
if (settings.onpopstate) {
    window.addEventListener('popstate', function (event)
{
        if (!history.state.url) return;
        routes.route(history.state.url);
    }, false);
}
```

Browser Compatibility

The `router()` method works in any browser, and IE9 and up.

Placeholders

Depending on which parts of your user interface you're rendering with JavaScript and how you're fetching your data (Ajax vs. local storage, and so on), there may be parts of your UI that aren't ready when the page first loads.

Placeholder content—sometimes called an app shell—can help improve the perceived load time of your app.

// @todo insert image

Your placeholders mimick the completed layout of the app. They're typically gray, and often have a slight animation to them to imply that the real content is loading.

Let's look at how to use them in your web app.

Creating placeholders

Placeholder content can be created with just a little bit of HTML and CSS. First, reate a `<div>` and add the `.placeholder` class.

```
<div class="placeholder"></div>
```

Then we'll add this CSS to our app.

```
/**
 * Setup keyframes for pulsing animation
 */
```

```
@-webkit-keyframes loadingPlaceholders {
    0% {
        background-color: lightgray;
    }
    50% {
        background-color: #e5e5e5;
    }
    100% {
        background-color: lightgray;
    }
}
@keyframes loadingPlaceholders {
    0% {
        background-color: lightgray;
    }
    50% {
        background-color: #e5e5e5;
    }
    100% {
        background-color: lightgray;
    }
}

/**
 * Style the placeholder
 */
.placeholder {
  -webkit-animation: loadingPlaceholders 1.5s ease-in inf
inite;
          animation: loadingPlaceholders 1.5s ease-in inf
```

```
  animation: loadingPlaceholders 1.5s ease-in-out infi
nite;
  background-color: #e5e5e5;
}
```

Customizing Placeholders

While the CSS above adds the basic functionality, you'll want to style your placeholders to match the layout of your app.

I use a handful of modifier classes to create different shapes to match my content.

```
.placeholder-hero {
    height: 20em;
}

.placeholder-heading {
    height: 3em;
    width: 55%;
}

.placeholder-sentence {
    height: 1.5em;
    margin-bottom: 0.5em;
}

.placeholder-sentence-last {
    width: 85%;
```

```
    width: 85%;
}

.placeholder-paragraph {
    height: 8em;
    margin-top: 1.625em;
}

.placeholder-btn {
    height: 3em;
    width: 8em;

}

.placeholder-thumbnail {
    border-radius: 50%;
    height: 9em;
    width: 9em;
}

.placeholder-hero,
.placeholder-heading,
.placeholder-paragraph,
.placeholder-btn,
.placeholder-thumbnail {
    margin-bottom: 1.625em;
}
```

And you use them like this.

```
<div class="placeholder placeholder-hero"></div>

<div class="placeholder placeholder-thumbnail"></div>

<div class="placeholder placeholder-paragraph"></div>

<div class="placeholder placeholder-sentence"></div>
<div class="placeholder placeholder-sentence"></div>
<div class="placeholder placeholder-sentence placeholder-
sentence-last"></div>

<div class="placeholder placeholder-btn"></div>
```

Putting it all together

To make this all tangible, let's work on a project together. We'll build a web app that let's you randomly pick who's going to drive from a group of friends.

The starter template and complete project code are included in the source code[6] on GitHub.

Getting Setup

I've dropped some placeholder markup into the template to help you get started.

It includes an empty `<div>` with an ID of `#app`. This is where we'll render all of the content for our app. There's also a `<script>` tag in the footer for our JavaScript.

```
<!-- The app container -->
<div id="app"></div>

<script>
    // Codes goes here...
</script>
```

There's also some light CSS to give form labels a bit of style.

```
label {
    display: block;
    font-weight: bold;
    margin-bottom: 0.5em;
}
```

How it's going to work

Our app will have three sections:

1. A list of all potential drivers.
2. A form you can use to add new drivers to the list.
3. A button that shuffles the list and picks a driver.

Render our base layout

The first thing we need to do is render our base layout.

We want unique `<div>` containers for each of our three content sections so that we can independently render and re-render them as users add new drivers and then select one. We also probably want to add a heading for our app.

The finished markup should look something like this.

```
<div id="app">
    <h1>Who's Driving</h1>
    <div id="driver-list"></div>
    <div id="driver-actions">
        <form id="add-driver">
            <label>Type the name of your driver and hit e
nter</label>
            <input type="text" name="new-driver" id="new-
driver" autofocus>
            <button class="btn" type="submit">Add Driver<
/button>
        </form>
        <form id="pick-driver"><button class="btn btn-sec
ondary" id="pick-driver-submit">Pick Who Drives</button><
/form>
    </div>
    <div id="selected-driver"></div>
</div>
```

The `#driver-actions` content shouldn't ever need to be updated, since the user is just entering names there, so we can render out all of it's content on load.

Create a layout component

Let's first add our `render()` and `component()` helper functions.

```
var render = function (template, elem) {
    // ...
};

var component = function (template, props, elem) {
    // ...
};
```

Next, let's set up our HTML above as a component, and render it into the app. We don't need to set a state for the layout, since it's really just a container for the rest of the page content.

```
// The base layout
var layout = component(function () {
    var template =
        '<div id="app">' +
            '<h1>Who\'s Driving</h1>' +
            '<div id="driver-list"></div>' +
            '<div id="driver-actions">' +
                '<form id="add-driver">' +
                    '<label>Type the name of your driver
and hit enter</label>' +
                    '<input type="text" name="new-driver"
 id="new-driver" autofocus>' +
                    '<button class="btn" type="submit">Ad
d Driver</button>' +
                '</form>' +
                '<form id="pick-driver"><button class="bt
n btn-secondary" id="pick-driver-submit">Pick Who Drives<
/button></form>' +
            '</div>' +
            '<div id="selected-driver"></div>' +
        '</div>';
    return template;
});

// Render the base layout
render(layout, '#app');
```

Create the driver list

Next, let's set up our list of potential drivers. This will be a simple unordered list with each driver's name.

Before we build our template, let's set up the state (or data) for our component. We'll call it `drivers`, and give it a default value of an empty array. As the user adds potential drivers, we'll push them into the array.

```
// The list of potential drivers
var driverList = component(function (props) {
    // Our template will go here...
}, {drivers: []});
```

In our template, we'll loop through each driver in our state (`props.drivers`) and create a list item. If there are no drivers, we'll `return` an empty string. Otherwise, we'll return our unordered list with each driver in it. Then we can render our list into the DOM.

```
// The list of potential drivers
var driverList = component(function (props) {

    // If their are no drivers, bail
    if (props.drivers.length < 1) return '';

    // Setup our template
    var template = '';

    // For each driver, create a list item
    for (var i = 0; i < props.drivers.length; i++) {
        template +=
            '<li>' +
                props.drivers[i] +
            '</li>';
    }

    // Otherwise, return our unordered list
    return '<ul>' + template + '</ul>';

}, {drivers: []});

// Render the driver list
render(driverList, '#driver-list');
```

Add drivers to the list

Now we're ready to start adding potential drivers to the list.

To do this, we *could* listen for clicks on our "Add Driver" button. A better way, though, is to listen for `submit` events. This will let users click the "Add Driver" button *or* hit the enter key for faster submissions.

We'll use event bubbling to listen for all form submissions and filter out the form to add drivers.

```
// Listen for form submissions
document.addEventListener('submit', function (event) {

    // Check if the form is the driver list form
    if (event.target.id === 'add-driver') {
        // Add driver to the list
    }

}, false);
```

Next, we need to do a few things:

1. Prevent the form from submitting.
2. Get the input field in our form.
3. Check that a name was entered (if it's empty, we'll do nothing).
4. Update the state for our `driverList` component using `setState()` to cause the list to render again.
5. Clear the input field.

```
// Listen for form submissions
document.addEventListener('submit', function (event) {
```

```
    // Check if the form is the driver list form
    if (event.target.id === 'add-driver') {

        // Prevent the form from submitting
        event.preventDefault();

        // Get the input field
        // If there's no driver name, bail
        var driver = event.target.querySelector('#new-dri
ver');
        if (driver.value.length < 1) return;

        // Update the state for our `driverList` componen
t and render again
        driverList.state.drivers.push(driver.value.trim()
);
        driverList.setState({
            drivers: driverList.state.drivers
        });

        // Clear the input field
        driver.value = '';

    }

}, false);
```

Try it out! Type a name and hit enter to see the list update.

Create our selected driver component

Once we have some potential drivers, we need to pick one to actually drive. For this, we need to add a selected driver component with it's own state.

We'll give it a state of `selected` with a default value of `null`. If there's no selected driver, we won't show anything. If there is a driver, though, we'll display their name in bold, followed by the words, "is driving."

We can render our component right into the DOM.

```
// The selected driver
var selectedDriver = component(function (props) {
    if (!props.selected) return;
    return '<p><strong>' + props.selected + '</strong> is
 driving</p>';
}, {
    selected: null
});

// Render the selected driver
render(selectedDriver, '#selected-driver');
```

Pick a driver

Now it's time to pick a driver. We can use our same `submit` event listener to detect clicks or keyboard interactions with the "Pick Who Drives" button.

Just like last time, we want to prevent the default form submission behavior.

```
// Listen for form submissions
document.addEventListener('submit', function (event) {

    // Check if the form is the driver list form
    if (event.target.id === 'add-driver') {
        // ...
    }

    // Check if the form is the pick driver button
    else if (event.target.id === 'pick-driver') {

        // Prevent the form from submitting
        event.preventDefault();

        // Pick our driver...
    }

}, false);
```

To pick our driver, we want to grab our drivers from the `driverList.state` object, randomly shuffle the order of our array, and pick the first one in the new, shuffled list.

After some Googling, I found this great article from Frank Mitchell[7] on the best way to shuffle arrays with JavaScript. He does a better job explaining it than I could, so go read the original for the details.

The one change I made was the clone our original array and shuffle the clone instead of the original. We want to keep the driver order the same in our `driverList`.

Add this helper method to the script.

```
/**
 * Shuffle our array of drivers and return the first item
 * Source: https://www.frankmitchell.org/2015/01/fisher-y
ates/
 * @param  {Array} array The array to shuffle
 * @return {String}      The first item in the shuffled a
rray
 */
var pickDriver = function (array) {

    if (array.length < 1) return '';

    // Create a clone of the array so that we don't shuff
le the original one
    var arr = array.slice();

    var i = 0;
    var j = 0;
    var temp = null;
```

```
    for (i = arr.length - 1; i > 0; i -= 1) {
        j = Math.floor(Math.random() * (i + 1));
        temp = arr[i];
        arr[i] = array[j];
        arr[j] = temp;
    }

    return arr[0];

};
```

Now, we can pick a random driver and set our `selectedDriver` state.

```
// Listen for form submissions
document.addEventListener('submit', function (event) {

    // Check if the form is the driver list form
    if (event.target.id === 'add-driver') {
        // ...
    }

    // Check if the form is the pick driver button
    else if (event.target.id === 'pick-driver') {

        // Prevent the form from submitting
        event.preventDefault();

        // Pick a driver
        var picked = pickDriver(driverList.state.drivers)
;

        // Update the state
        selectedDriver.setState({
            selected: picked
        });

    }

}, false);
```

At this point, you now have a fully functional web app. Nice work!

Now let's add a few extras.

Remove a driver from the list

There's a good chance that at some point your user will want to remove a driver from the list. Someone backs out of plans, or they type the name in wrong, and so on.

Let's add a link next to each driver in the list that our user can click to remove that name from the list. We'll give it a `[data-driver-remove]` data attribute that we can hook into in an event listener. We'll give it a value of the driver's name.

We should also assign the link a `role` of `button`, so that assistive technologies like screen readers know what to expect.

```
// The list of potential drivers
var driverList = component(function (props) {

    // If their are no drivers, bail
    if (props.drivers.length < 1) return;

    // Setup our template
    var template = '';

    // For each driver, create a list item
    for (var i = 0; i < props.drivers.length; i++) {
        template +=
            '<li>' +
                props.drivers[i] +
                ' <a role="button" data-driver-remove="'
+ props.drivers[i] + '" href="#">(remove)</a>' +
            '</li>';
    }

    // Otherwise, return our unordered list
    return '<ul>' + template + '</ul>';

}, {drivers: []});
```

Now, we can add a `click` event listener. We'll again use event bubbling and check for the `[data-driver-remove]` data attribute.

```
// Listen for click events
document.addEventListener('click', function (event) {

    // If a remove driver link was clicked
    if (event.target.hasAttribute('data-driver-remove'))
{
        // Remove the driver...
    }

}, false);
```

We need to prevent the default link behavior. We'll use `indexOf` to get the driver from our `driversList.state.drivers` array, using the name stashed in our data attribute. Then, we'll use the `splice()` method to remove it from the array.

Then, we'll use the `setState()` method on our `driverList` component to update the DOM.

```
//Listen for click events
document.addEventListener('click', function (event) {

    // If a remove driver link was clicked
    if (event.target.hasAttribute('data-driver-remove'))
{

        // Prevent the link from doing anything
        event.preventDefault();

        // Get the driver from our state
        var driver = driverList.state.drivers.indexOf(eve
nt.target.getAttribute('data-driver-remove'));

        // Remove the driver from state
        driverList.state.drivers.splice(driver, 1);

        // Update the state
        driverList.setState({
            drivers: driverList.state.drivers
        });

    }

}, false);
```

Add a settings page

Let's add a settings page where users can delete all drivers out of their app at once.

This is, frankly, completely unneccessary for an app like this. But I want to make sure you get a chance to practice URL routing with single page apps.

The support this, let's first add the `router()` helper method to our script.

```
var render = function (template, elem) {
    // ...
};

var component = function (template, props, elem) {
    // ...
};

var router = function (routes, options) {
    // ...
};
```

Add routes

We only need three routes for this app: `home`, `settings`, and an empty string that we'll redirect to `home` for people who visit for the first time without a query string in the URL.

We'll also pass in a title of `Who\'s Driving` (the `\` is to escape the single quote) as a setting so that it will get added to each page.

```
var routes = router({
    settings: {
        title: 'Settings | '
    },
    home: {
        title: ''
    },
    '': {
        redirect: '?p=home'
    }
},
{
    title: 'Who\'s Driving'
});
```

Now that we have routes set up, we need to listen for `route` events. When one occurs, we'll render our app again.

```
// Listen for route events
window.addEventListener('route', function () {

    // Render the base layout
    render(layout, '#app');

    // Render the driver list
    render(driverList, '#driver-list');

    // Render the selected driver
    render(selectedDriver, '#selected-driver');

}, false);
```

Remove our standalone `render()` functions

Currently, we're using the `render()` method to render our `layout`, `driverList`, and `selectedDriver` components when the page loads, and then again on `route` events.

We can make the code a bit more DRY (an acronym for "Don't Repeat Yourself") and easier to maintain by removing the standalone `render()` functions that happen on page load. We'll instead rely on our `route` event to handle the initial page load as well.

How? We'll use the `route()` method from the router API, and pass in the initial page URL using `window.location.href`.

```
// Listen for route events
window.addEventListener('route', function () {

    // Render the base layout
    render(layout, '#app');

    // Render the driver list
    render(driverList, '#driver-list');

    // Render the selected driver
    render(selectedDriver, '#selected-driver');

}, false);

// Render the page for the first time
routes.route(window.location.href);
```

Add a link to the settings page

Now we have a router set up, but we're not passing any links through it.

Let's add a link to the settings page just below the heading. We're using query strings, so the link `href` value will be `?p=settings`.

```
// The base layout
var layout = component(function () {
    var template =
        '<div id="app">' +
            '<h1>Who\'s Driving</h1>' +
            '<p><a data-route href="?p=settings">Settings
 &rarr;</a></p>' +
            '<div id="driver-list"></div>' +
            '<div id="driver-actions">' +
                '<form id="add-driver">' +
                    '<label>Type the name of your driver
and hit enter</label>' +
                    '<input type="text" name="new-driver"
 id="new-driver" autofocus>' +
                    '<button class="btn" type="submit">Ad
d Driver</button>' +
                '</form>' +
                '<form id="pick-driver"><button class="bt
n btn-secondary" id="pick-driver-submit">Pick Who Drives<
/button></form>' +
            '</div>' +
            '<div id="selected-driver"></div>' +
        '</div>';
    return template;
});
```

Refresh the page. You should see a settings link now. If you click it, the title in the tab/navbar will update, and so will the URL. If you called `history.state` in the Console tab of developer tools, you'll see

something like this.

```
// history.state
{
    id: "settings",
    params: {
        p: "settings"
    },
    url: "03%20-%20project%20complete%20with%20url%20routing.html?p=settings"
}
```

Show different content based on route state

Next, we need to conditionally show content based on the route. To do that, we'll hook into the `routes` state in our components.

Let's start with the the `layout` component. We'll use an `if...else` statement to show different content based on the `routes.state.id`.

```
// The base layout
var layout = component(function () {
    var template;
    if (routes.state.id === 'settings') {
        template =
            '<h1>Settings</h2>' +
            '<p><a data-route href="?p=home">&larr; Pick who\'s driving</a></p>' +
```

```
            '<div id="settings"></div>';
    } else {
        template =
            '<div id="app">' +
                '<h1>Who\'s Driving</h1>' +
                '<p><a data-route href="?p=settings">Sett
ings &rarr;</a></p>' +
                '<div id="driver-list"></div>' +
                '<div id="driver-actions">' +
                    '<form id="add-driver">' +
                        '<label>Type the name of your dri
ver and hit enter</label>' +
                        '<input type="text" name="new-dri
ver" id="new-driver" autofocus>' +
                        '<button class="btn" type="submit
">Add Driver</button>' +
                    '</form>' +
                    '<form id="pick-driver"><button class
="btn btn-secondary" id="pick-driver-submit">Pick Who Dri
ves</button></form>' +
                '</div>' +
                '<div id="selected-driver"></div>' +
            '</div>';
    }
    return template;
});
```

We should also conditionally render our `driverList` and `selectedDriver` components if the route ID is `home`.

```
// The list of potential drivers
var driverList = component(function (props) {

    // Only render on the homepage
    if (routes.state.id !== 'home') return;

    // If their are no drivers, bail
    if (props.drivers.length < 1) return '';

    // Setup our template
    var template = '';

    // For each driver, create a list item
    for (var i = 0; i < props.drivers.length; i++) {
        template +=
            '<li>' +
                props.drivers[i] +
                ' <a role="button" data-driver-remove="'
+ props.drivers[i] + '" href="#">(remove)</a>' +
            '</li>';
    }

    // Otherwise, return our unordered list
    return '<ul>' + template + '</ul>';

}, {drivers: []});

// The selected driver
var selectedDriver = component(function (props) {
```

```
    // Only render on the homepage
    if (routes.state.id !== 'home') return;

    if (!props.selected) return;
    return '<p><strong>' + props.selected + '</strong> is
 driving</p>';

}, {
    selected: null
});
```

Go ahead and refresh the browser to see this in action. You can now toggle between two different views.

Add a settings component

Next, let's create a component for our settings that we'll use to add a "Remove All Drivers" button. We don't need to add state to this one since the UI is not conditional on data.

We'll only render this when the route ID is `settings`.

```
// The settings form
var settings = component(function () {
    if (routes.state.id !== 'settings') return;
    var template =
        '<form id="driver-settings">' +
            '<button type="submit">Remove All Drivers</bu
tton>' +
        '</form>';
    return template;
});
```

Next, we need to add a `render()` function for our settings component into our `route` event listener.

Remove all drivers

Finally, we need to remove all drivers when the “Remove All Drivers” form is submitted. We can use our existing `submit` event listener for this.

We’re going to reset the `driverList.state.drivers` value to an empty array. We’ll also wipe out any selected driver if one exists. Then, we’ll redirect users back to the homepage by calling the `routes.route()` method, with `?p=home` as the URL.

```
// Listen for form submissions
document.addEventListener('submit', function (event) {

    // Check if the form is the driver list form
    if (event.target.id === 'add-driver') {
        // ...
    }

    // Check if the form is the pick driver button
    else if (event.target.id === 'pick-driver') {
        // ...
    }

    // Check if the form is the settings form
    else if (event.target.id === 'driver-settings') {

        // Reset the `driverList` state to an empty array
        driverList.state.drivers = [];

        // Reset the `selectedDriver` state to null
        selectedDriver.state.selected = null;

        // Redirect user back to the homepage
        routes.route('?p=home');

    }

}, false);
```

And with that, our app is complete.

Congratulations! You just created your first web app with vanilla JavaScript!

About the Author

Hi, I'm Chris Ferdinandi. I help people learn JavaScript.

I love pirates, puppies, and Pixar movies, and live near horse farms in rural Massachusetts. I run Go Make Things with Bailey Puppy, a lab-mix from Tennessee.

You can find me:

- On my website at GoMakeThings.com.
- By email at chris@gomakethings.com.
- On Twitter at @ChrisFerdinandi.

1. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/polyfills/CustomEvent.js↩
2. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/polyfills/CustomEvent.js↩
3. https://developer.mozilla.org/en-US/docs/Web/Events↩
4. https://github.com/cferdinandi/vanilla-javascript-cheat-sheet/blob/master/polyfills/matches.js↩
5. https://github.com/browserstate/history.js↩
6. https://github.com/cferdinandi/web-apps-source-code/↩
7. https://www.frankmitchell.org/2015/01/fisher-yates/↩